AEGIS OUTLINE

PHILOSOPHY of AEGIS

- Integrated System
- Object orientation
- Managers as Model for Data Abstraction

OVERVIEW of AEGIS CONCEPTS

- Processes
- Object–Based File System
- Naming
- Mapping / Address Space Management
- Memory Management
- Networking
- Protection

OBJECTS

- Storage and Disk Structures
    - pvol, lvol, bat, vtoc
    - important bootstrapping information
        in the lv_label
- UIDs, Attributes, Segmentation, Locating
- Locking (local)

NAMING

- Directories
- /, //, 'Node_Data, WD, ND
- Links (hard and soft)

ACCESSING OBJECTS

Address Spaces (asids, global)
Mapping Objects (mst)
Active Objects (ast)
Paging/Purifier

NETWORK FILE SYSTEM

Remote vs. Local
Paging Server, Remote File Server
Asknode

INTERPROCESS COMMUNICATION

The Ring
Packets and Sockets
Major Clients of Sockets
MBX

SECURITY

Acls, Registry, Protected Subsystems
Login, SIDs

PROCESS MANAGEMENT (Supervisor Mode)

Process Switching (dispatching)
Interrupt Handling
Processor Scheduling
Synchronization (eventcounts)
Mutual Exclusion
Special CPU B Handling

Process Creation and Deletion
Clocks and Time–Driven Events

PROCESS MANAGEMENT (User Mode)

Program Management
Parsing
Program Levels, Procceses and Fork
Mapped Segment Manager (ms)
Storage Allocator (rws)
The Loader, KGT
Libraries; Global and Private

PROCESS MANAGEMENT
(Error and Fault Handling)

Kinds of Faults
Supervisor Mode Fault Handling/Generation
User Mode Fault Generation
Fault Handlers
Dynamic Cleanup Handlers
Static Cleanup Handlers
Mark/Release Handlers

STREAMS

The Stream Table
Opening Streams
The Generic Switch Call
Some Special Switch Calls
The D_File Manager
Other Managers

FROM POWER–UP TO LOGIN

Physical / Virtual Address Space Layout

MD

- SIO vs. Display KBD
- Service / Normal
- Boot Device Selection
- Commands : Internal vs. External
  - (LD, LO, EX)

SYSBOOT / NETBOOT

- Aegis initialization
  - required directories and files
  - creating the first level 2 process

THE BOOT SHELL

- ENV / Libraries
  - the basic idea (SH, DM, SPM)
  - firmware (PEB and COLOR)
  - global libraries
  - startup–files (where and why)

DISKLESS NODES

NETWORK SERVERS

- SPM / CRP
- SIOLOGIN
- SF HELPER
- ALARM SERVER

THE APPLICATION LEVEL

PST
NETSTAT
HPC
NETLOG
DB
FST
TB
COMPILER/BINDER

GPIO

MULTIBUS Limits
Device Driver Considerations

Philosophy & Overview of *AEGIS*

Philosphy: 3 perspectives

- market
- hardware technology
- system architecture technology

Overview: textbook OS taxonomy

- processor management
- address space management
- memory management
- file system
- network
- I/O device management

Apollo Computer

The premier supplier of workstations
for the technical professional

Maximize the productivity of the technical professional via:

1. ability to run large, mainframe class application programs tailored to his profession

2. high user <–> computer bandwidth

3. network for cooperation and sharing with others

Implications:

1. a. Fast, 32 bit CPU
   b. Virtual memory

2. a. Bit–mapped display
   b. Window–oriented user environment

3. a. Distributed system
   b. Net–wide access to files

AEGIS is the operating system that resulted to support these objectives.

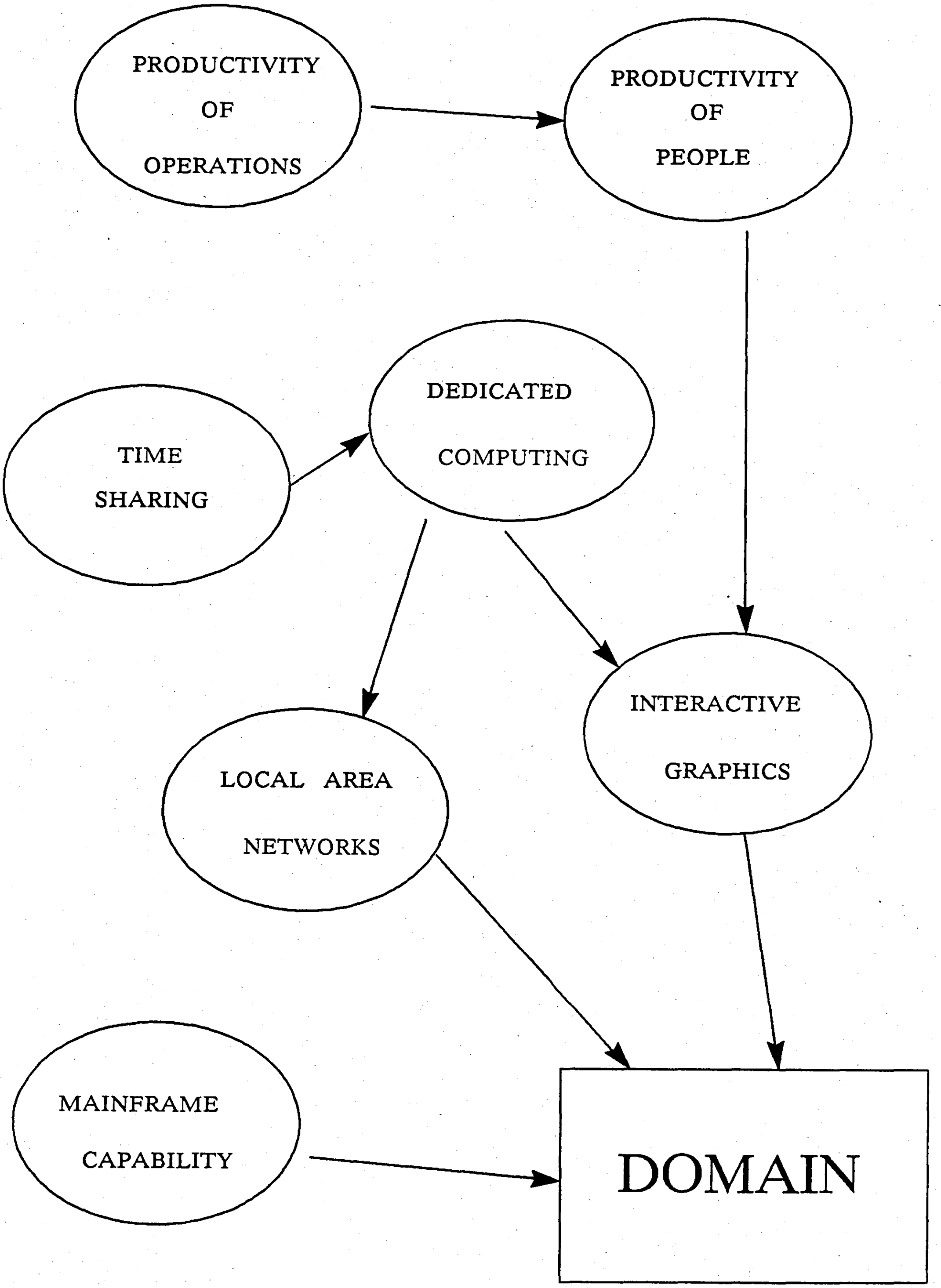
Hardware Technology

1. VLSI CPU's
2. 64k RAM
3. WInchester disks

Pioneered by the Alto at Xerox PARC, started to see other systems:

Nu Machine (MIT)
SUN Machine (Stanford)

This new, cheaper computing power was changing the focus on how computing was done....


PRODUCTIVITY OF OPERATIONS
PRODUCTIVITY OF PEOPLE
TIME SHARING
DEDICATED COMPUTING
LOCAL AREA NETWORKS
INTERACTIVE GRAPHICS
MAINFRAME CAPABILITY
DOMAIN

System Architecture Technology

Operating systems

- Multics (MIT)

  Bill Podeska
  Bernie Stumpf

  - original implementation
  - restructuring studies
- Hydra, Medusa (CMU)
- System/38 (IBM)

Distributed systems

- Pilot (Xerox PARC)
- WFS (Xerox PARC)

Languages

- Mesa (Xerox PARC)
- CLU (MIT)
- Alphard (CMU)
- Smalltalk (Xerox PARC)
- Ada (DoD)

Key attributes of AEGIS

AEGIS is a

- *distributed*
  - *integrated*
  - *local area network*
- *object–oriented*
- *personal workstation*

operating system.

Distributed Systems

Advantages:

- **robustness, reliability**
  - *when one node fails, system still runs*
- **incremental expansion of computing power**
  - *just keep on adding nodes*
- **potential for higher performance**
  - *run computations in parallel*

Problems:

- **partial failures**
  - *if you need the node that failed...*
- **"richer" set of errors**
  - *not just "up" or "down"*
- **replication needed for reliability**
  - *hard to do automatically*
- **parallelism needs to be explicitly programmed**
  - *no automatic decomposition today*
- **sharing & cooperation**
  - *can be hard to get back to timesharing level*

Where does Aegis fit?

Lots of different kinds of distributed systems.

- **VAXcluster: a distributed multi–computer**
  - **meant to act exactly like one big VAX**
    - *good sharing & cooperation*
    - *all the problems of timesharing*

- **ARPAnet: communicating, autonomous hosts**
  - **seperately owned and administered**
    - *limited sharing & cooperation*
      - *remote login, file transfer, mail*

Aegis falls somewhere in between.

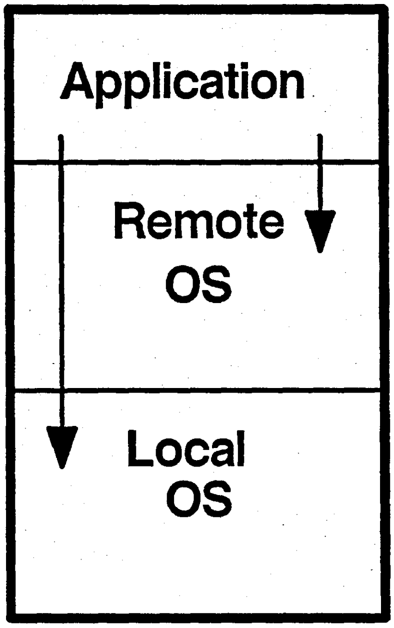
Structural Implications

- distributed systems are naturally structured differently than centralized ones

- Aegis was built from the ground up to be distributed

"Local access is the **special** *case"* — *PHL*
"...but it still has to be fast" — *PJL*

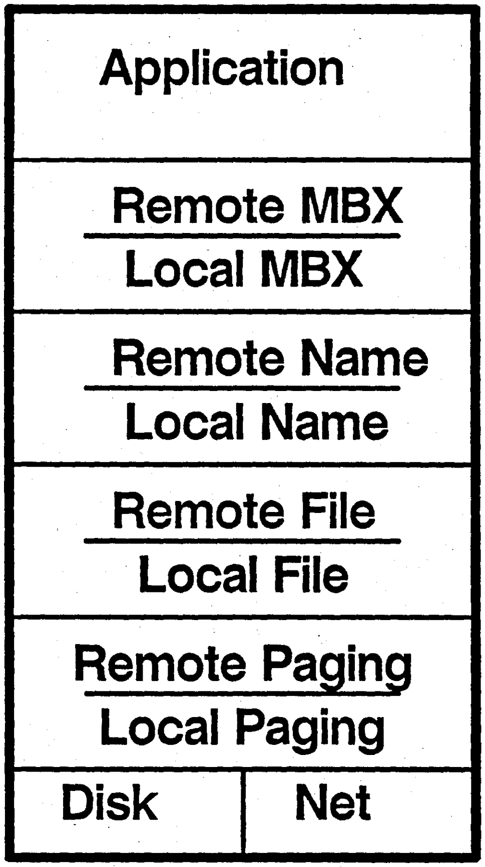
Contrast to Post–Hoc Distributed Systems



A complete remote OS is layered on top of a complete local OS; applications determine which is being requested at each use.

Aegis Structure I

In Aegis, each component has a local and remote part within it.



Application

Remote MBX

Local MBX

Remote Name

Local Name

Remote File

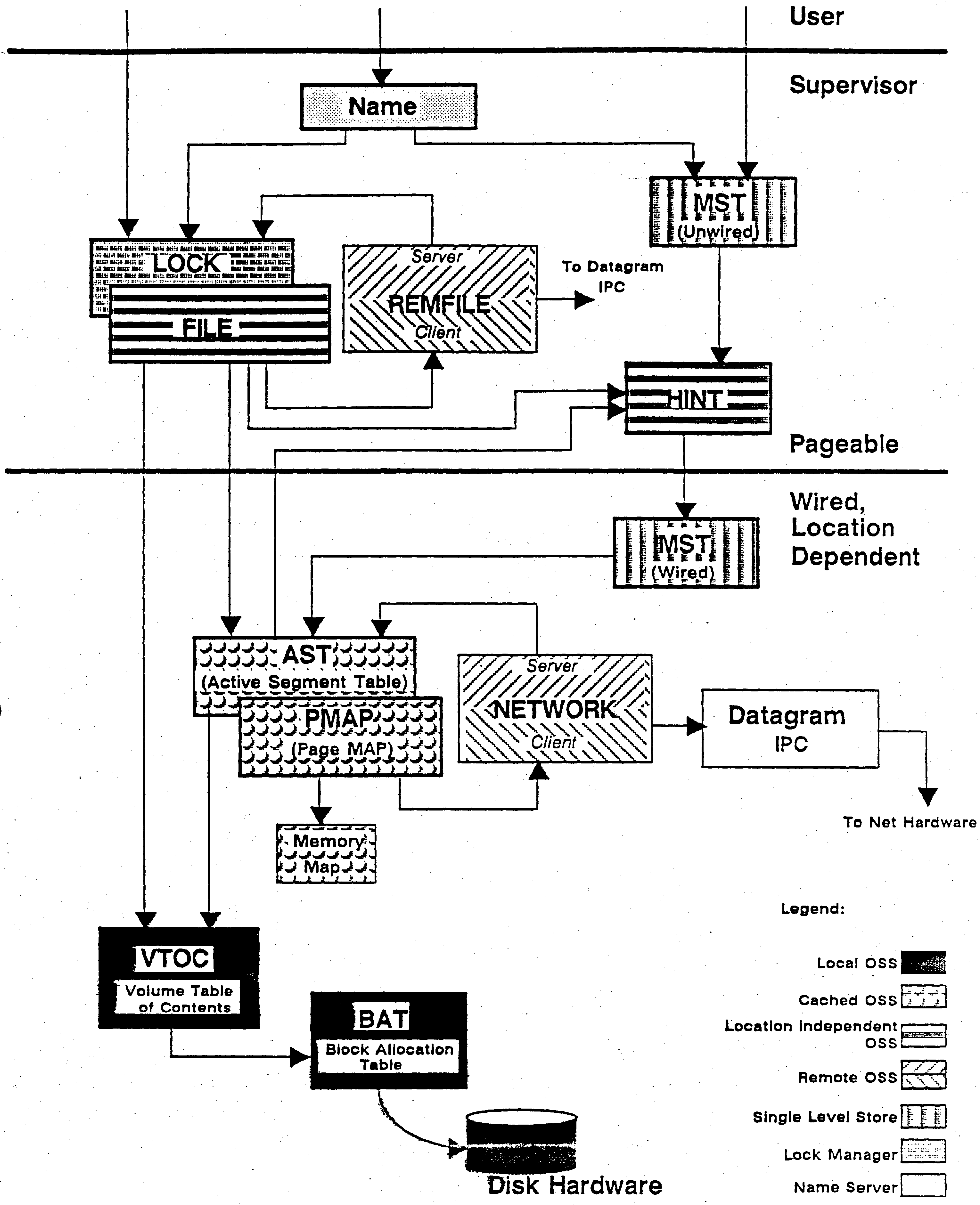
Local File

Remote Paging

Local Paging

Disk

Net



File System Structure

Aegis Structure II: Net–Wide Caching

Another example of "ground up" distribution:

Network–wide caching of objects would probably not have been feasible without having designed it in from the start.

The file locking operations were specifically designed to allow cache control in addition to concurrency control.

Personal Workstation Implications

With a network of personal workstations:

- **(potentially) can share what's important**
  - *information, programs*
  - *expensive peripherals*
- **don't share what's not important**
  - *CPU cycles: they're cheap*
- **you can decide how to use your node**
  - *autonomy*

Potential advantages:

- **cooperation & sharing**
  - *use network*
- **dedicated, controllable performance**
  - *you allocate your node*
- **high user <–> computer bandwidth**
  - *CPU is close to the display*
  - *highly interactive user environment*
- **simpler OS if only run one user**

Simpler OS

Protection

- all computation on a node is on behalf of a single person
- don't worry about maliciousness
- just worry about accidents

Fairness of resource allocation

- just do what the owner says

Accounting

- is in terms of the whole node

Structure

- can put software in user space
  - easier to modify, debug, replace

Openness

- more facilities can be made accessible if needn't worry about above items

Problems with Personal Workstation Model

How to manage tension between autonomy and cooperation.

- autonomy means independence
- cooperation means dependence

Solution: make cooperation voluntary; but how?

- need mechanisms
- usually, cooperation & autonomy go along machine boundaries
  - on same machine: cooperate
  - on different machine: autonomy
- not good enough for personal workstations

Problems II

How to provide traditional system services:

- identifying users to the system
- printing
- backup
- mail
- storage of community information
  - at project, department, organization and corporate levels
- data integrity
- data privacy
- communication gateways
- background computation (batch)

Partial solution: use "servers" to provide them

- dedicated nodes running trusted applications

Cooperation vs. Autonomy
Why are both needed?

Cooperation:

- **need to cooperate with colleagues to get your job done**
  - *personal workstation didn't change that!*

Autonomy:

- **need to control resources of own node**
  - *in order to get controllable response*
- **need to control sharing**
  - *to protect privacy of data*
- **need to manage own data files**
  - *to guarantee data integrity*
- **need to operate when network is down**
  - *need enough independence to do so*

Server Issues

Protection:

- all programs on same server node trust each other

Fairness of resource allocation:

- they also trust each other to be reasonable in their resource use

Accounting:

- is up to each server to do in an application specific way

Local Area Network Implications

Local area networks are sufficiently different from other kinds of networks that different techniques need to be used to take advantage of them.

Bandwidth:

- **typical networks are orders of magnitude slower than the memory bus**
- **LAN's are faster: ours has 2/3 the bandwidth of the memory bus of a DN400.**

Error rates:

- **typical network error rates: 10**–4 or so.**
- **LAN error rates much lower**

SO:

- **minimize CPU time to "get on and off the wire"**

  don't spend it trying to optimally utilize network bandwidth
- **don't worry as much about errors**

  use simple retransmission techniques

Problem Oriented Protocols

Don't use the traditional OSI "layered" architecture

- **make a very cheap datagram service.**
- **don't use virtual circuits, sessions, presentation layer.**
- **take advantage of operation semantics to cheaply do what those layers normally do.**
- **use "end–to–end" argument.** avoid acknowledgements.

Examples:

- *idempotent operations*
- *transaction IDs*
- *"natural" state* don't specially package data.

P–O–P Examples

Idempotent operations

- *has same effect if done twice in a row as if done once.*
    - *example: read page N of a file*
- *use simple two message protocol*
    - *RR: request/response*
    - *retransmit on time out*
    - *duplicate requests no problem*
- *saves an acknowledge message (RRA)*

Transaction IDs

- *eliminate duplicate replies*
    - *tag each request with a unique number*
    - *discard replies with duplicate TIDs*

Natural state

- *for non–idempotent operations*
- *save request TID in a database that was needed anyway*
- *discard requests with duplicate TIDs; resend old response*
- *example: lock database*

Integrated Distributed System

System provided, user selectable mechanisms that:

- *Preserve* **autonomy.**
- *Permit* **cooperation & sharing (when desired).**

Provide the user with a unified system:

- **name files, not hosts**
- **system wide user identification**

Integrated Implications

Network wide file system:

- **to make sharing easy**

Network transparency:

- **location transparency:**
  - **all resources accessed in same way, regardless of their location**
  - *easier software development*
  - *supports incremental changes to system*
  - *easier to realize increased reliability*
  - *simpler user model*
- **name transparency:**
  - **name doesn't imply location**
  - *allows relocation, substitution*

Uncatalog object
and re-catalog it on a
remote node ; object
doesn't move

Control mechanisms:

- **access control**
- **network wide user identification**

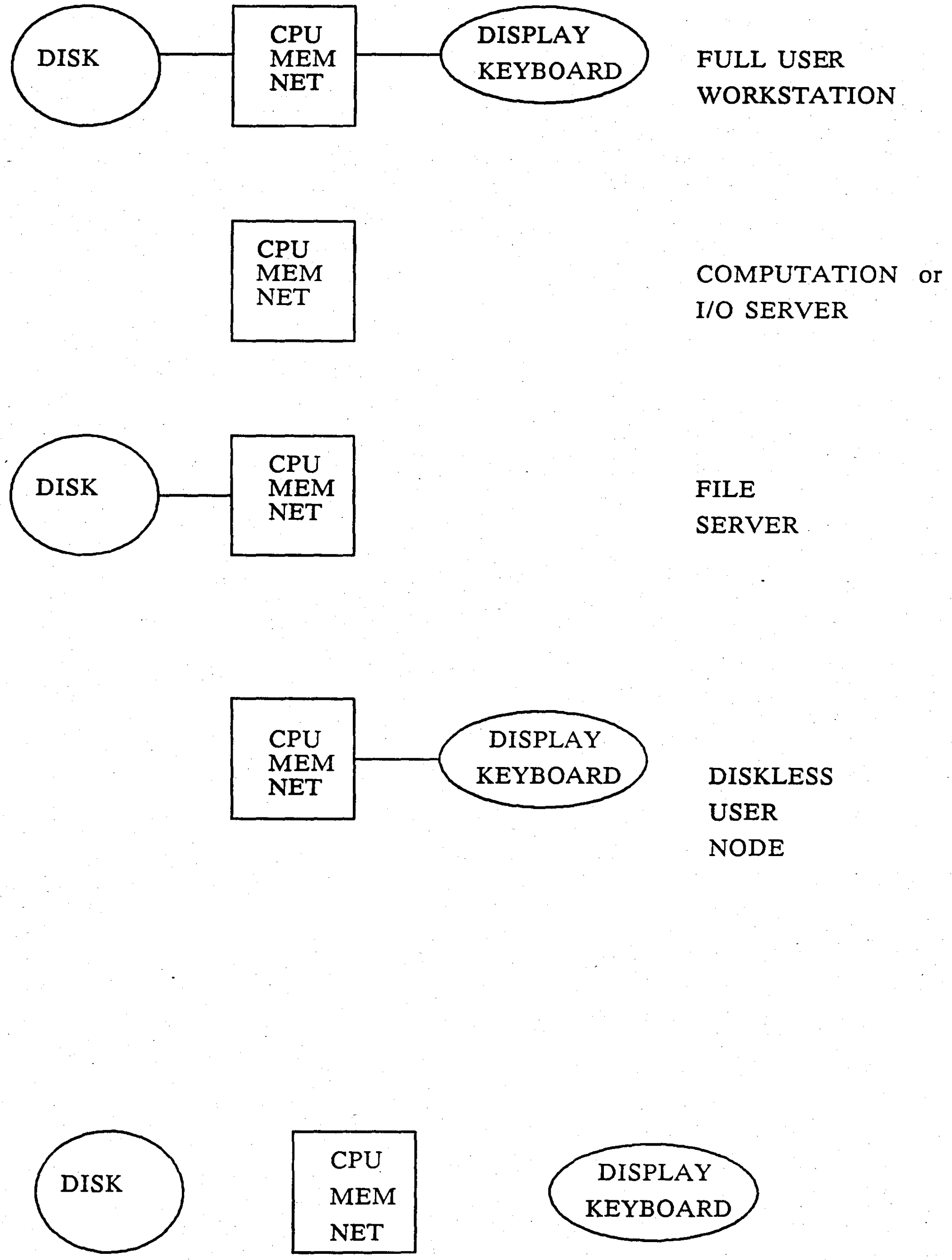
Integrated Implications II

Reliability criterion:

- must always be able to access information on own node, even if network down
- if two nodes are up and want to cooperate, then no single failure will stop them
  - so, third parties must be replicated

Functional integration:

- each node has a complete set of OS facilities
  - *so can run when network down*
  - *also for performance reasons*



DISK — CPU MEM NET — DISPLAY KEYBOARD

FULL USER WORKSTATION

CPU MEM NET

COMPUTATION or I/O SERVER

DISK — CPU MEM NET

FILE SERVER

CPU MEM NET — DISPLAY KEYBOARD

DISKLESS USER NODE

DISK

CPU MEM NET

DISPLAY KEYBOARD

MODULAR WORKSTATION DESIGN

Object Orientation

Object:

- **user level: some sealed data plus operations**
- **OS level: a storage container for uninterpreted data, plus a type tag that**
  - **identifies the object's manager**
  - **tells how to interpret the data.**

Only way to get to an object is through its manager.

autonomous **Managers:**

- **each module is manager of some** *object.*
- **object is some meaningful (OS) entity**
  - *disk block, process, file, directory, etc.*
- **manager handles all details of "its" objects**
- **interface to manager gives all permissable operations; completely defines object to clients**
  - *clients only manipulate object through the interface*
- **manager is solely responsible for the integrity of its objects**
  - *knowledge of representation (data structures) confined to manager*
  - *managers correctness depends only on itself, managers of components*

Objects II

Why?

- understandable semantics for modules;
  a principle for OS decomposition into modules
- managers are orthogonal and independent
  - can isolate bugs to one manager
  - can find manager to change to make an enhancement

Protection

Need access control to allow you to choose with whom to share and cooperate.

Can't protect data on a node from the node owner:

- has physical access

SO:

- allow each node to protect own data against access from the network
- don't try to protect data from deliberate efforts of node owner
- try to make accidents improbable

Aegis Interface

	MANAGERS
Single Level Store	MST
Object Storage System	FILE
Low Level IPC	MSG
Naming Server	NAME
Processes	PROC2, EC2

Faults	FAULT, FIM fault interceptor manager
Display	COLOR, SMD, SMDU
I/O	MT, LPR, PBU,, DISK, VOLX,TERM
Protection	ACL
Info	AS address space, BAT block allocation table, ASKNODE, PROC1, VTOC, CAL, NETWORK, object storage OS, PEB, TPAD, NETLOG, GET_BUILD_TIME, OSINFO
Misc	TIME, UID, VFMT UID_LIST

Processes

- independent, asynchronously executing
- 33 total 8 processes reserved for O.S.
- one is the Display Manager (user process)
- Shell windows are processes, edit pad windows are not
- Serarate address space per process
  asid
  - * for protection
  - * because the address space is too small (less than 10 MB min.)
- Address Space
  - * 256 (or 16) Megabyte
  - * objects mapped into it
  - * R/W with ordinary instructions
- Object Types
  - * programs, libraries, data
- Aegis is in each address space

Processes 2

– Synchronization and Communication

* Shared Objects (communication)

same object in AS of > 1 proc.
both observe changes
restricted to 1 machine

* Eventcounts (synchronization)

processes can wait on an EC
processes can "advance" EC
to wake up waiters
also restricted to 1 machine

* IPC (MBX)

both comm. and synch.

sends data, wakes up receiver

network wide!

local, too; exactly the same

semantics (but more efficient)

Processes 3

Dispatching   Scheduling

- dynamic (recalculates)
- priority based
- priority is inversely proportional to the amount of CPU time used
    - attempts to give interactivity priority
    - paging is currently a problem
- Priority boost ?.
    - delta added to the priority computed above
    - Dispaly Manager gets it ?.
    - It is not user settable
- Process Layering

  PM
  PROC2
  PROC1

PROC1

lives in wired portion of process,

- Synchronized with EC1
- A finite number of them (33)
- Wired state
- State = registers
  PSW
  ASID
  locks
- Runs only in global space
- Needed to implement Virtual Memory

  * purifier

  *paging server

  * file server

  ...

6 processes needed to implement kernel { run in same address space, use same data bases

2 unused

PROC2

- Synchronized with EC2

- Runs in its own address space

- Can use Virtual Memory

- Potentially unwired state

    * eventually bind and unbind

    * copies state in VM

mutex lock

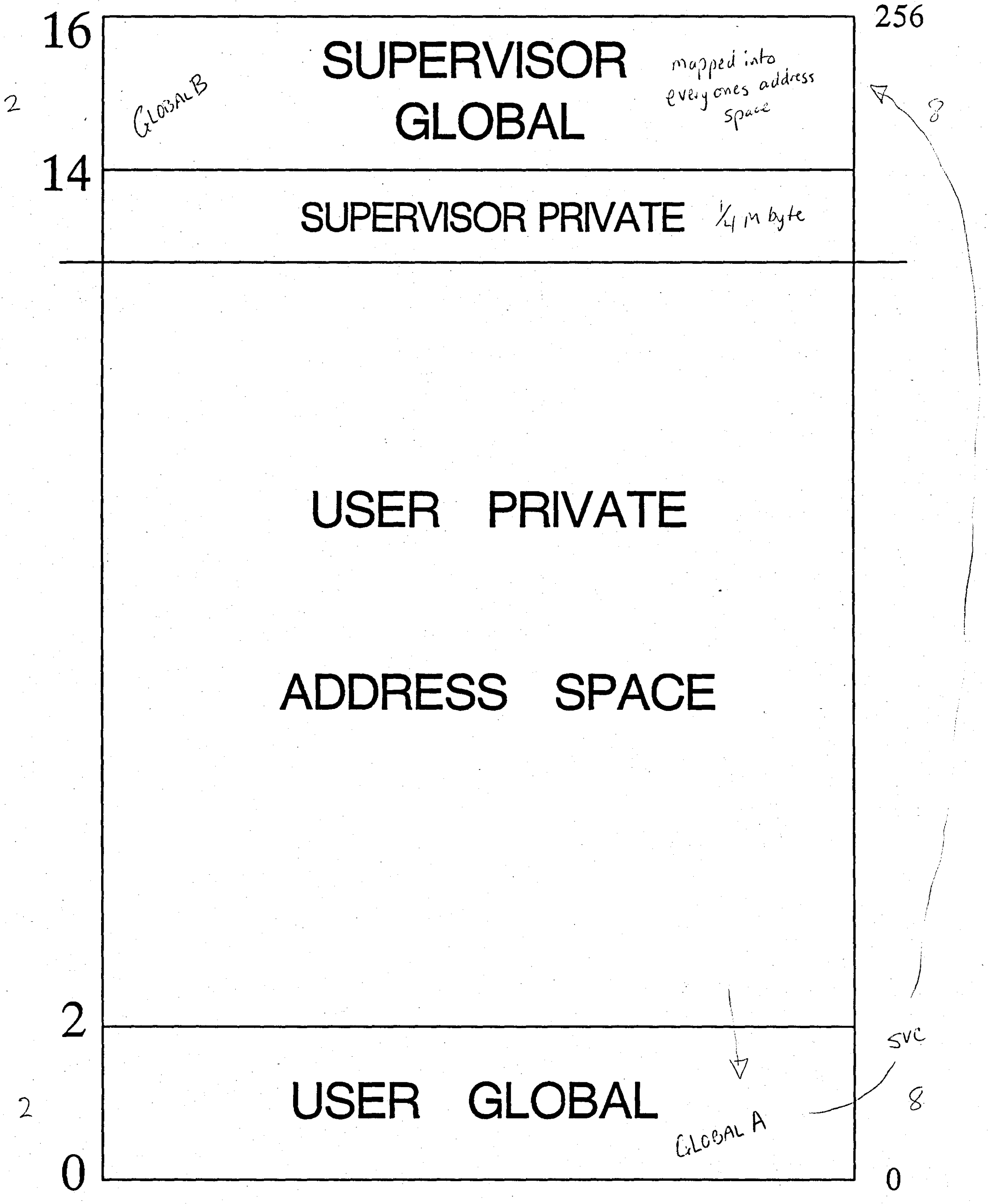
ML

- Mutex Lock

- Uses EC1

- Deadlock detection

Virtual address space



16

SUPERVISOR GLOBAL

GLOBAL B

mapped into everyones address space

14

SUPERVISOR PRIVATE

1/4 M byte

USER PRIVATE

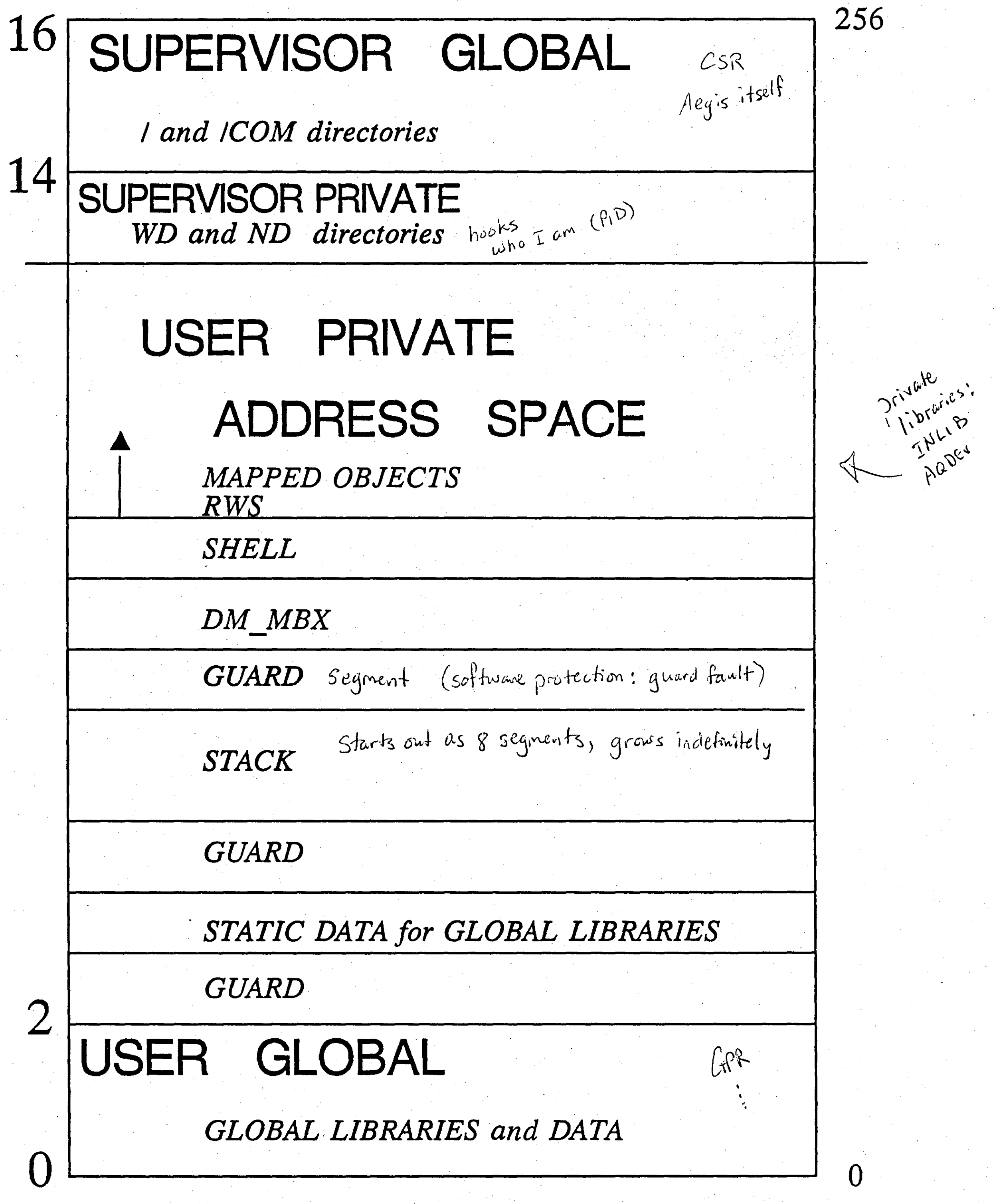
ADDRESS SPACE

2

USER GLOBAL

GLOBAL A

0



16

256

SUPERVISOR GLOBAL

CSR
Aegis itself

/ and /COM directories

14

SUPERVISOR PRIVATE

WD and ND directories

hooks
who I am (PID)

USER PRIVATE

ADDRESS SPACE

Private
libraries:
INLIB
AQDev

MAPPED OBJECTS
RWS

SHELL

DM_MBX

GUARD Segment (software protection: guard fault)

STACK Starts out as 8 segments, grows indefinitely

GUARD

STATIC DATA for GLOBAL LIBRARIES

GUARD

2

USER GLOBAL

GPR
⋮

GLOBAL LIBRARIES and DATA

0

0

each segment is 32 k

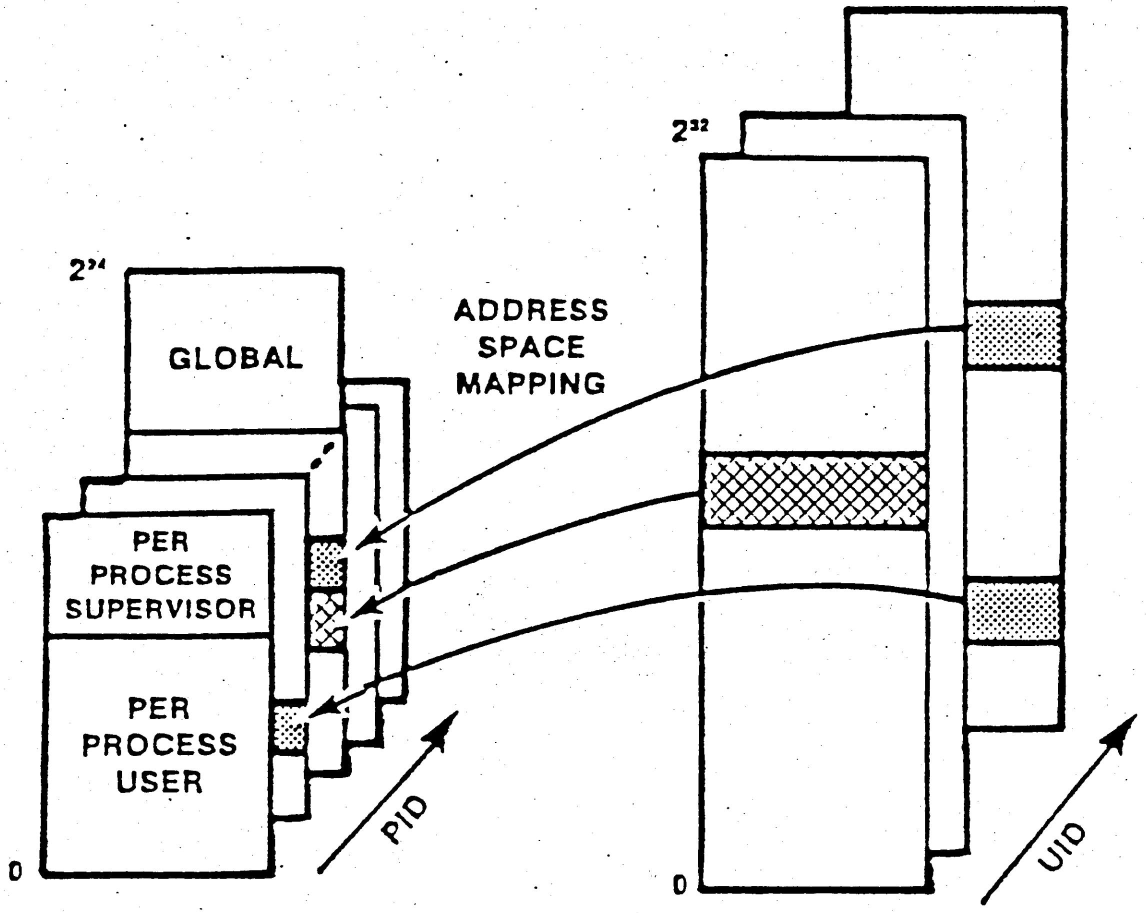
```
      VA Range     Obj Start   Pathname              don't see supervisor global

     8000 -     FFFF        0   /sys/node_data/global_data
    10000 -    1FFFF        0   /lib/pmlib
    20000 -    37FFF        0   /lib/syslib.460
    38000 -    3FFFF        0   /lib/vfmt_streams
    40000 -    47FFF     8000   /sys/node_data/global_data
    48000 -    67FFF        0   /lib/streams
    68000 -    7FFFF        0   /lib/error
    80000 -    9FFFF        0   /lib/swtlib
    A0000 -    A7FFF        0   /lib/pbulib
    A8000 -    AFFFF    10000   /sys/node_data/global_data
    B0000 -    BFFFF        0   /lib/ftnlib
    C0000 -    E7FFF        0   /lib/gprlib
    E8000 -    FFFFF        0   /lib/clib
   100000 -   117FFF        0   /lib/shlib
   118000 -   11FFFF        0   /lib/auxlib
   120000 -   127FFF    18000   /sys/node_data/global_data
   128000 -   137FFF        0   /lib/tfp
   138000 -   13FFFF        0   /lib/x25lib
   140000 -   147FFF    20000   /sys/node_data/global_data
   148000 -   14FFFF        0   /sys/node_data/stream_$sfcbs
   800000 -   897FFF        0   -- temporary file --  stack
   898000 -   89FFFF        0   /sys/node_data/dm_mbx
   8A0000 -   8A7FFF        0   /com/sh
   8A8000 -   8AFFFF        0   -- temporary file --  stack
   8B0000 -   8B7FFF        0   /com/las
   8B8000 -   8DFFFF    98000   -- temporary file --  stack
   8E0000 -   8E7FFF        0   /f/las.big
  F788000 -  F797FFF        0   /f
  F798000 -  F7A7FFF        0   //node_28f4

2368 KB mapped.
```

Single Level Store

- Direct access to objects via machine instructions
- "Map" an object into a portion of a process' address space
- Only page in the needed pieces
- Similar to Multics, IBM System/38, and Xerox Pilot
- Distributed over the whole network

backing store for any object
does not have to be on the local node:
it can be anywhere.

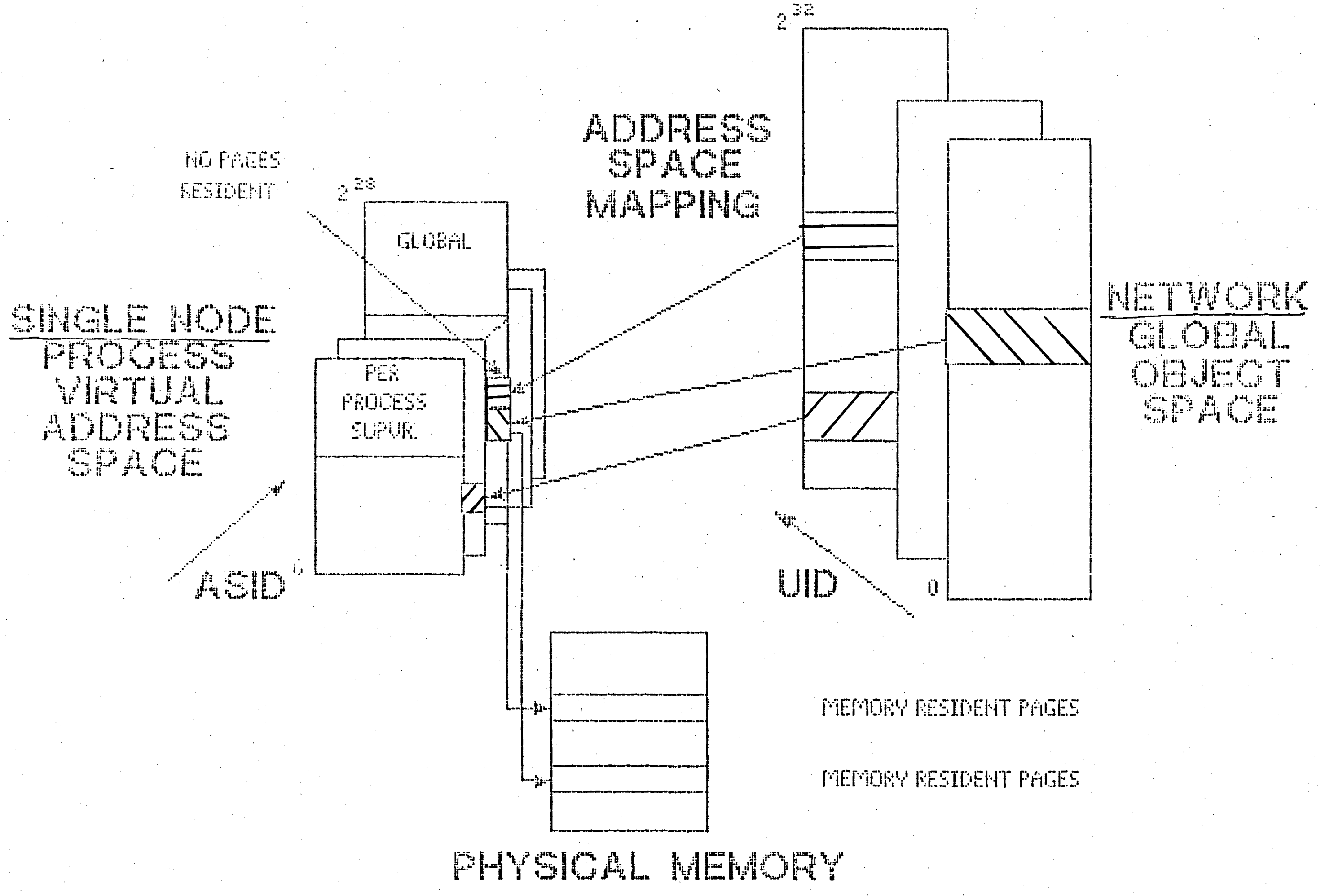
OPERATING SYSTEM MAPPING



SINGLE NODE
PROCESS
VIRTUAL ADDRESS
SPACE

NETWORK
GLOBAL
OBJECT SPACE

OPERATING SYSTEM MAPPING



Libraries

- the environment for programs
  - all callable entry points not bound with the program

    dynamically bound as program is run.

    Actual symbols reference is left in the program

  - most of the system services are made available through libraries (nucleus calls are in a library)
- dynamic linking to libraries

  reason for this: everything is totally relocatable within system.

  - symbolic references left in program (the name of the proc/subr/func)
  - resolved by the loader when the program is invoked
  - uses the KGT (known global table)
- loading vs. installing
  - programs are loaded

    GPIO

  - libraries are installed, entries are kept in the KGT

two KGT's : per process <u>lifo</u> this one searched first by loader.
system-wide

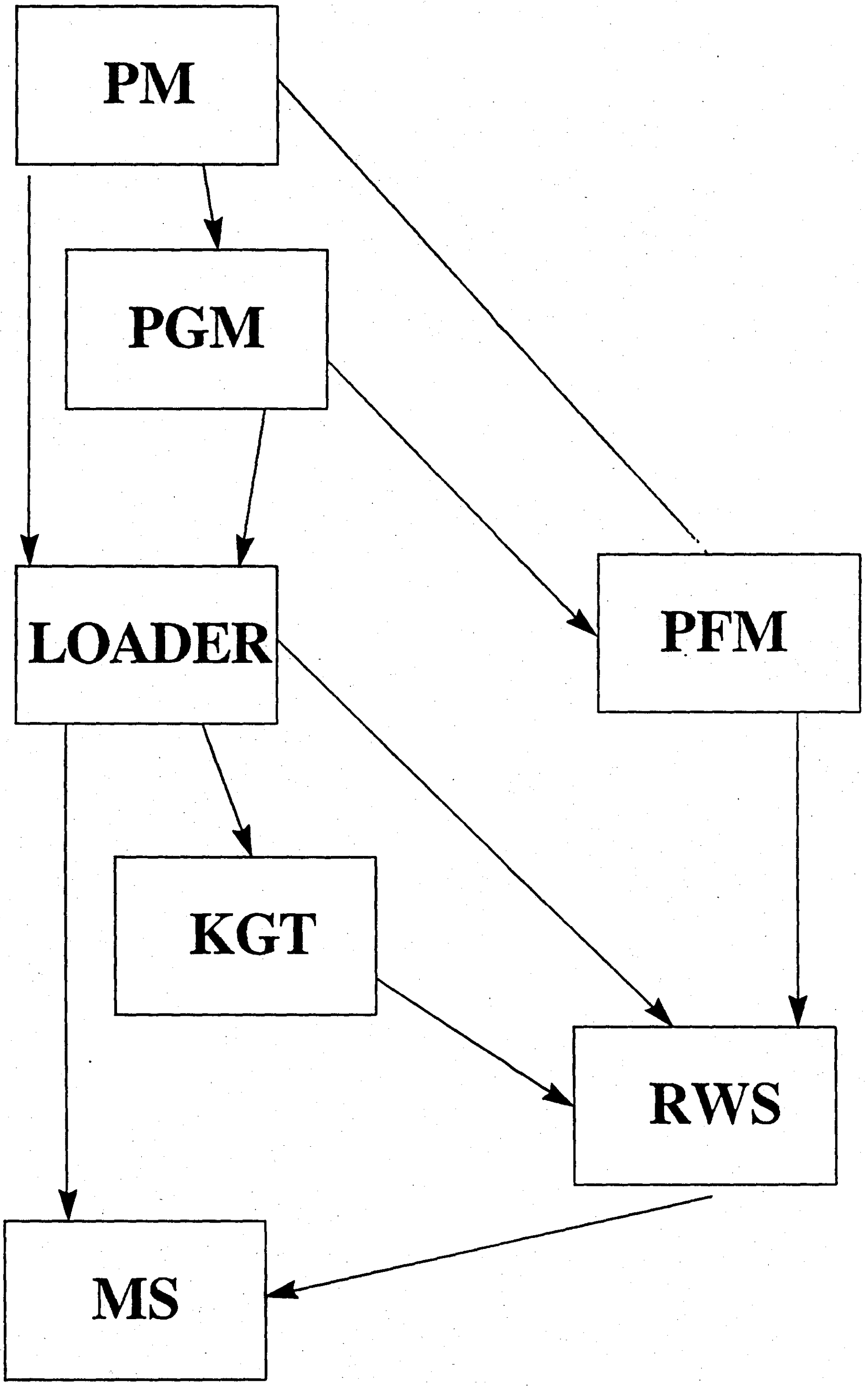
Global vs. Private Libraries

Global

- in the Address Space of all processes
- automatic
- don't need to be loaded when each process is created
- more efficeint sharing (hardware)
- installed when the system comes up (ENV)

Private

- in the AS of processes that load it
- installed after the system comes up
- not enough global space for all libraries
- each process does INLIB
  still sharable, but more costly (because of virtual stuff... two MST pointers to resolve)
- INLIB command


PM
PGM
LOADER
PFM
KGT
RWS
MS

Programs

- a file system object
- a kind of procedure (or set of ...)
- special convention for invocation
  - args are an array of strings
  - redirection upon invocation
  - not normally in AS, must be mapped
- resource management unit
  - all resources a program acquires are released when program exits

    open streams are closed
    mapped objects are unmapped
    scratch space is released
    database areas are cleaned up

  - extensible

    mark/release handlers
    new managers install their own

Memory Management

Demand Paged Virtual Memory

- LRU replacement
- purifier (write–behind) every 10 seconds goes through 1/6 of memory

ASTE's

- hold disk addresses for "active" objects
- also object attributes
- 128 ASTE's per megabyte

as many ASTE's as real memory

active segment table entries (copies of VTOC for the object ; disk/network addes

Sequential access

- touch ahead (read ahead)
- allocate for disk locality

Random access to very large files

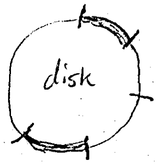
not good

- large: more than 4 meg/meg of main memory
- causes 2 disk I/O per page
  - one for file map
  - one for the page

block allocation table

BAT step = 3

at rev 9 = 2



if it's in AST, you don't have to look it up in the VTOC. Thus, AST is a type of cache.

invol 10 undocumented option to invol that lets you set the BAT step

File System Management

File system =

Object storage system

+ Naming server

+ Streams

Streams

Traditional device independent sequential I/O, plus

- seek
- record structure
- locate mode

Operations:

- Open, Close, Read, Write
  - a.k.a. get_rec, put_rec
- "handle" is a stream ID (small integer)

Implementation:

- "switch"
  - uses type UID
  - calls type dependent manager
- Files:
  - map into the address space (window)
  - slide the window over file
  - access via "load/store"
  - move mode – copies data into caller's buffer
    - no nucleus intervention
  - touch ahead automatically set depending on access
    read

Object Storage System ← OSS

- network transparent data access
- access files anywhere in the network as if they were local
- port Fortran, C, Pascal programs without change
- preserve investment
- only a 90% solution

***BUT a very important one ! ***

Totally distributed systems are not built in a day!

- object orientation
- all operations are operations on some object
- a 'natural' way to distribute

Software Environment

Aegis Operating System

- Objects
    - * named by UID
- Object attributes
    - * UID of ACL
    - * UID of type descriptor
    - * physical storage descriptor
    - * misc. (DTM, DTU, etc.)

* whether or not it can be mapped into supervisor space.

Supported Object Types

- alphanumeric text
- record structured data
- IPC "mailboxes"
- IPC "pipes"
- executable procedure
- directories
- ACLs
- serial I/O ports
- magnetic tape drives
- display bit maps
- ...

Create own object types + managers at SR9;
Extensible streams.

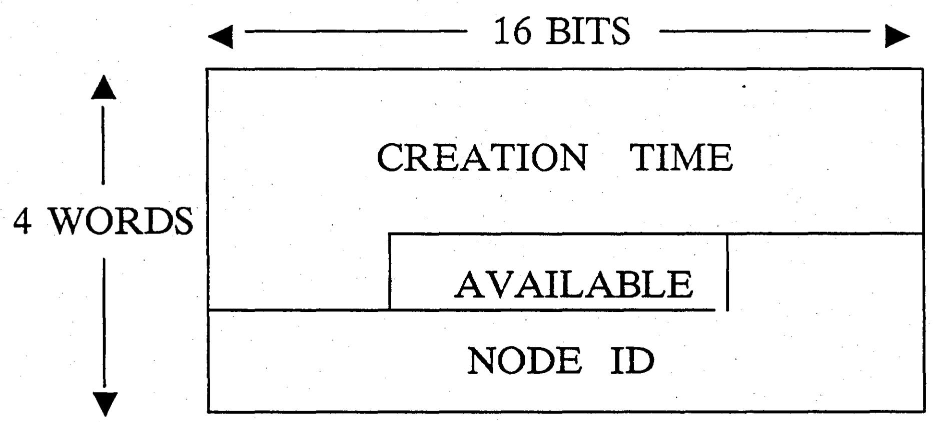
Internal/External Names

- External Name
  - * user visible, human usable
  - * text string
- Internal name
  - * computer convenient "handle" for an object
- Choices for form of internal name
  - * UID
  - * "structured name" name itself tells you where it is stored.
- UID is choice of Aegis
  - * just like a bit string that uniquely identifies an object
  - * but doesn't tell how to find it
  - * like a Social Security Number

- Structured name
  - * multiple components
  - * gives location of, or route to, object
  - * may or may not be reused
  - * may or may not be one–to–one with object

UIDs

- * 64 BIT UNIQUE NAME
- * NEVER (EVER) REUSED
- * CONCRETE REPRESENTATION



16 BITS

4 WORDS

CREATION TIME

AVAILABLE

NODE ID

- * OBJECTS ARE ACCESSED BY MAPPING INTO THE VIRTUAL MEMORY
- * OBJECT ACCESS IS NETWORK TRANSPARENT

Certain UID's will never be created : "Canned UIDs" used to bring up certain components of the O.S. they are programmed into the Boot proms.

WHY UIDs ?

- location independence
- absolute names with respect to processes, nodes
- simple nucleus interface
- uniform naming for all objects, by most levels
- composite objects
- typed objects

Locating Objects

- Make the task easier by restricting locations
  - * don't let objects move Can't have removable volumes
  - the way it is today * require objects to be on the same volume as the directory in which it is cataloged
  - * establish equivalence classes among volumes
  - * no restrictions; broadcast Can't have internet compatibility
- Requirements
  - * removable volumes
  - * internet environment compatibility

file_locate call

- Use "hints"

  * from node ID in UID
  * from "hint manager": takes hints from anywhere: directory manager, user ...

- Improve algorithm over time

  1. look local, then the node on which the object was created.

  2. local; hint manager; then the node of creation

  3. modify 2. to try remote first if the node ID in the UID is remote

Concurrency Control
(a.k.a. the stale cache problem)

Single level store

- SLS makes no consistency guarantee
  (property: purely local use is OK)

- Locking and timestamp techniques

  * lock before use; unlock after
  * timestamp detects stale data

? dtm/u

- Lock (an object)

MBX

  * send message to home node
    (acts as a coordinator)
  * get back version number
    (timestamp)
  * discard stale pages
    (ones with older timestamps)

- Unlock

  * send modified pages back to
    home node
  * send message to release lock

Pure data: read-only
? impuredata : r/w

- Page In
  - * returns page's version number
  - * check version number against current one
  - * return error if no match
- Page Out
  - * bumps version number, returns it
  - * checks, rejects if not owner requesting
- Client Protocols
  - * Possible because cache flushing operations are exported

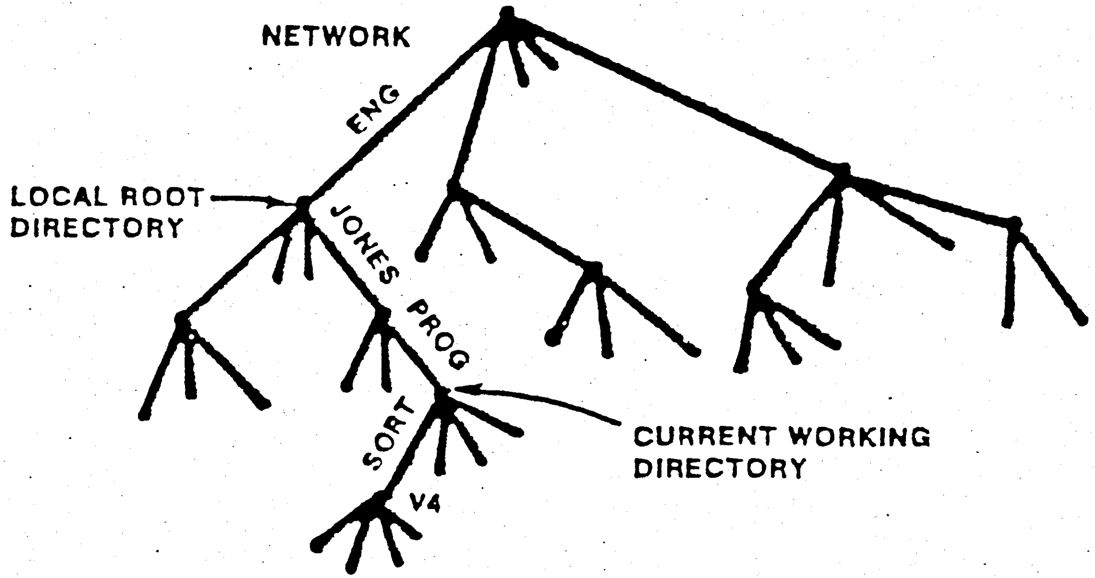
Uniform Name Space

- Same "absolute " file name refers to the same object, anywhere in the network

- Allows file names to be exchanged without changing meaning

- Means data, programs are more easily shared

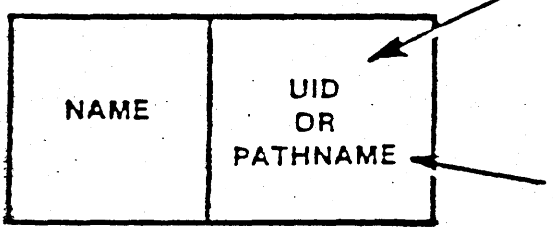
USER NAME SPACE



SYNTAX

//ENG/JONES/PROG...NETWORK WIDE
/JONES/PROG/SORT...LOCAL ROOT RELATIVE
SORT/V4...WORKING DIRECTORY RELATIVE

DIRECTORY OBJECT



POINTS TO NEXT DIRECTORY OR TARGET OBJECT

PATHNAME SUBSTITUTED IN NAME (LINK)

Naming

Text string names

- hierarchical tree structure
    - * "path name"
    - * made up of "component names"
    - * for example, /x/y/z
- directory objects
    - * component name => UID
    - * component name => path name links
- absolute path name
    - * starts at "root" directory
    - * leads to UID of an object
    - * valid network wide, like UID

Network Management

Sockets:

- datagram service
- IDs are small integers
- services are at "well known" sockets
- reply sockets allocated as needed

for instance; Socket 4 is the paging socket

MBX: implemented on top of ↑

- virtual circuit service
- IDs are UIDs, names
- "advertise" service in name space
- is not in the nucleus

I/O Management

Barely any; all special cased

- disk
- serial I/O
- network
- magtape
- line printer

all done by different special case managers.

Protection

User identification

- registry

Access Control Lists (ACLs)

Protected Subsystems

data protected from user, but not necessarily from a program that the user invokes.

Registry

- system wide registry of people, projects, and accounts
- identifies a user to the system, not just a node
- replicated for reliability, availability
- each node owner doesn't have to be a system administrator.

Can't have acl's
without accounts (registry)

Why not just OSS and SLS ?

- good if data << computing
  - user pays computing cost
  - automatic caching
- not so good if computing << data
  - cost of moving data high
- not so good: exposes representation of data ~~ot~~ to the whole network
- good when one process is computing on distributed data
- not so good when many , distributed processes are working on distributed data
  - more processes =>
    1 more reliability
  - more processes =>
    more performance
  - need synchronization

General Distributed Computing Tools

- Remote procedure calls
- Concurrent programming
- Replicated objects
- Consistency control
- “Yellow Pages”
- Remote process invocation and migration
- Debugging

Basic AEGIS Vocabulary

- UID
  - * Unique Identifier
- Object
  - * Anything where existence is associated with a UID (e.g. Files, Volumes, Processes)
- File
  - * Disk Resident Object

1056 bytes (data +header) for disk
1048 " packet " for network

- Page
  - * Smallest spearable unit of Memory, Disk, Object (1024 bytes for us)
- Segment
  - * 32–page grouping of Virtual Memory of object––smallest MAP–ABLE unit
- Mapping
  - * Associates Virtual Memory Segment with Object Segment

Disk Glossary

- Physical Volume

  * A disk

- Disk Block

  * 1056 byte section on a disk
    (32 byte header/1024 byte data)

- Logical Volume

  * A section of a physical volume that is
    completely self–describing and contained
    (Usually one L. V. per P. V.)

- Physical–Volume Label

  * Single disk block that describes the
    Physical Volume

- Logical–Volume Label

  * Single disk block that describes the
    Logical Volume

- Disk Address (DADDR)

  * Disk block number as an offset from the
start of Logical–Volume (usually)

Disk Block HEADER

- Reliability
- Recoverability

32 bytes in addition to 1024 data bytes

1056 total

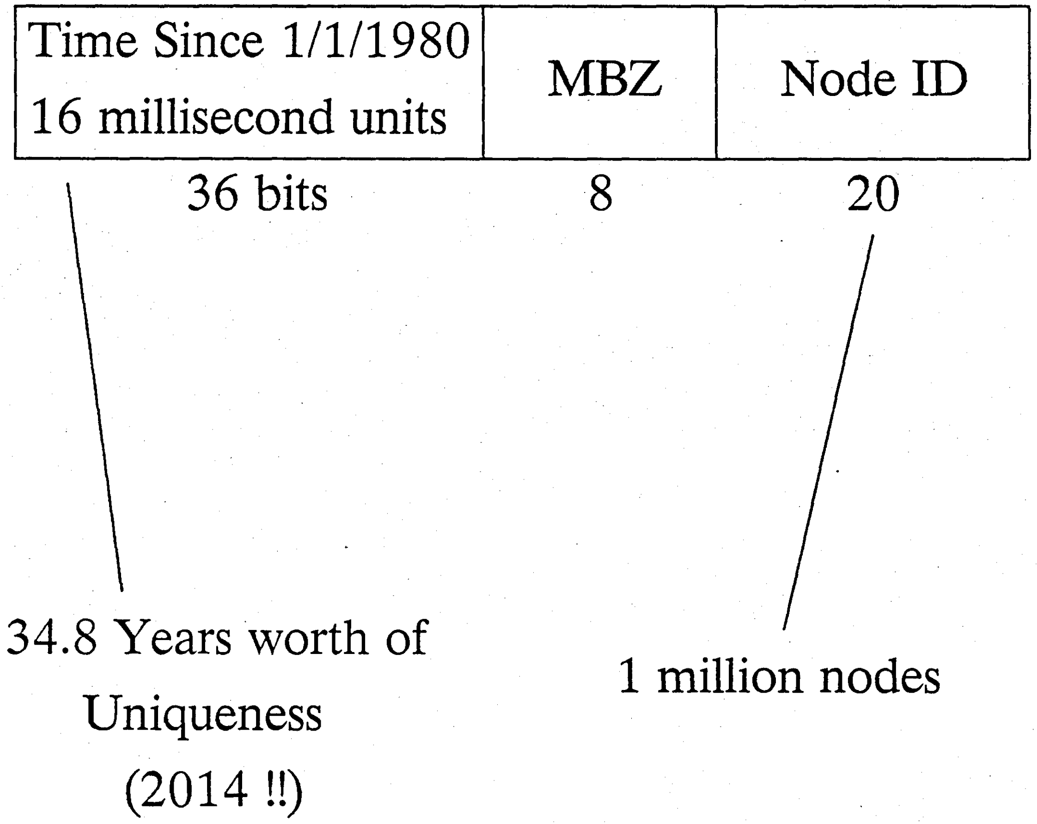
UID of object to which block belongs
Page# in file
Time written
Checksum of Data
Disk Address

originated so that you could reconstruct a VTOC, etc. never used.

scatter/gather

floppy drives don't have disk block headers.

Anatomy of a UID



Time Since 1/1/1980 16 millisecond units	MBZ	Node ID
36 bits	8	20

34.8 Years worth of
Uniqueness
(2014 !!)

1 million nodes

∴ We're not worried yet !!

“Canned” UID’s

- Hand constructed by R & D
- To identify “SPECIAL” objects

  * Examples:

  “Canned” ACLs--
  .%.%.%.%
  FNDWRX
  0001800F,0
  time node_ID

  * Disk Structures

  PHYS_VOL_LABEL
  00000200,0

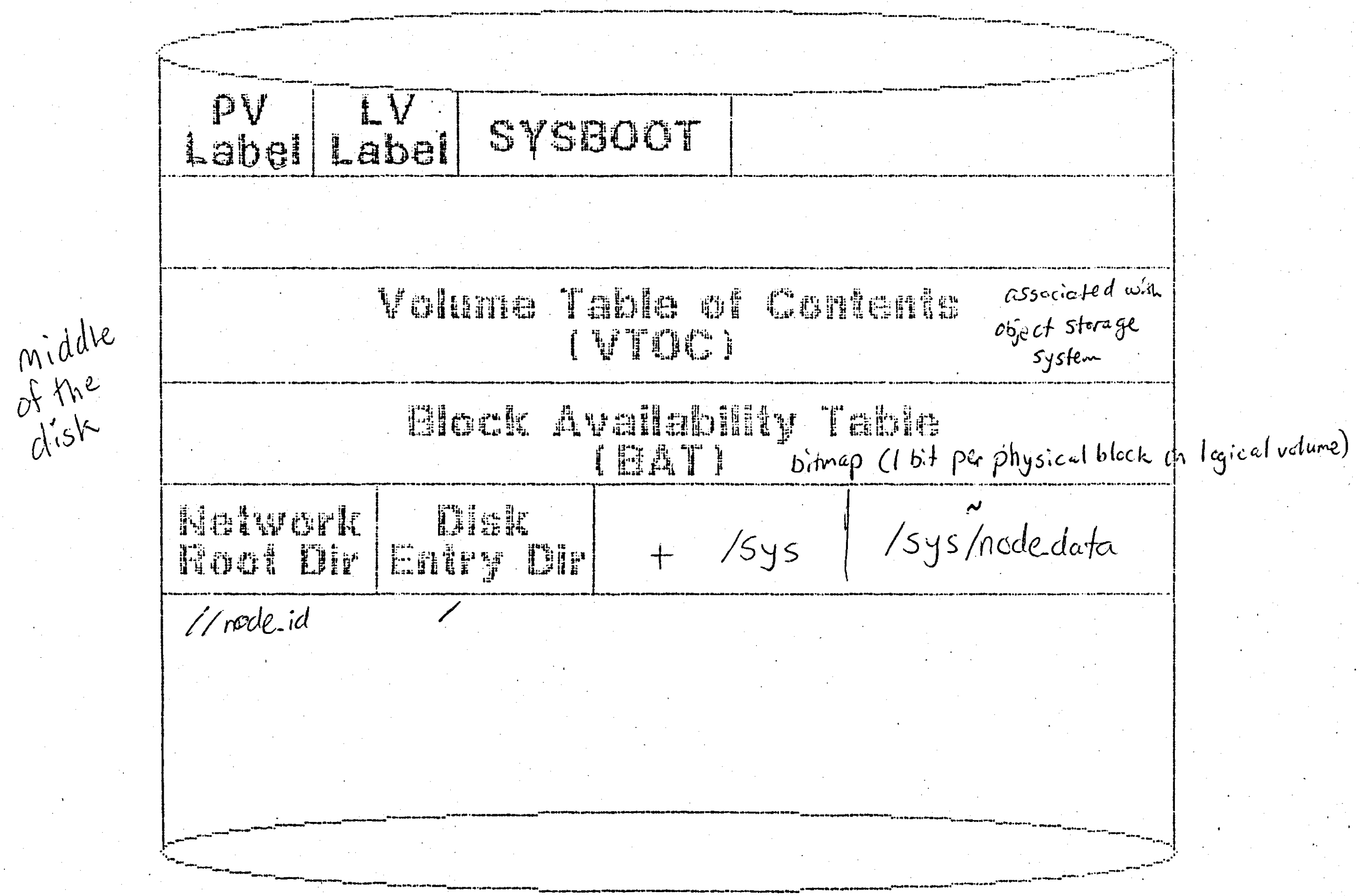
  Disk address zero

  Chuvol doesn't change canned UID's

  * “Canned” People (!)

  USER 00000500,0

DISK STRUCTURE


PV Label
LV Label
SYSBOOT
Volume Table of Contents (VTOC)
associated with object storage system
middle of the disk
Block Availability Table (BAT)
bitmap (1 bit per physical block on logical volume)
Network Root Dir
Disk Entry Dir
+ /sys
/sys/~/node.data
//node_id
/

PHYSICAL VOLUME LABEL

VERSION NUMBER
"APOLLO"
PHYSICAL VOLUME NAME
PHYSICAL VOLUME UID
BLOCK COUNT
BLOCKS PER TRACK
TRACKS PER CYLINDER
DISK ADDRESS (DADDR) OF LOGICAL VOLUME 1
DISK ADDRESS (DADDR) OF LOGICAL VOLUME 2
• • •

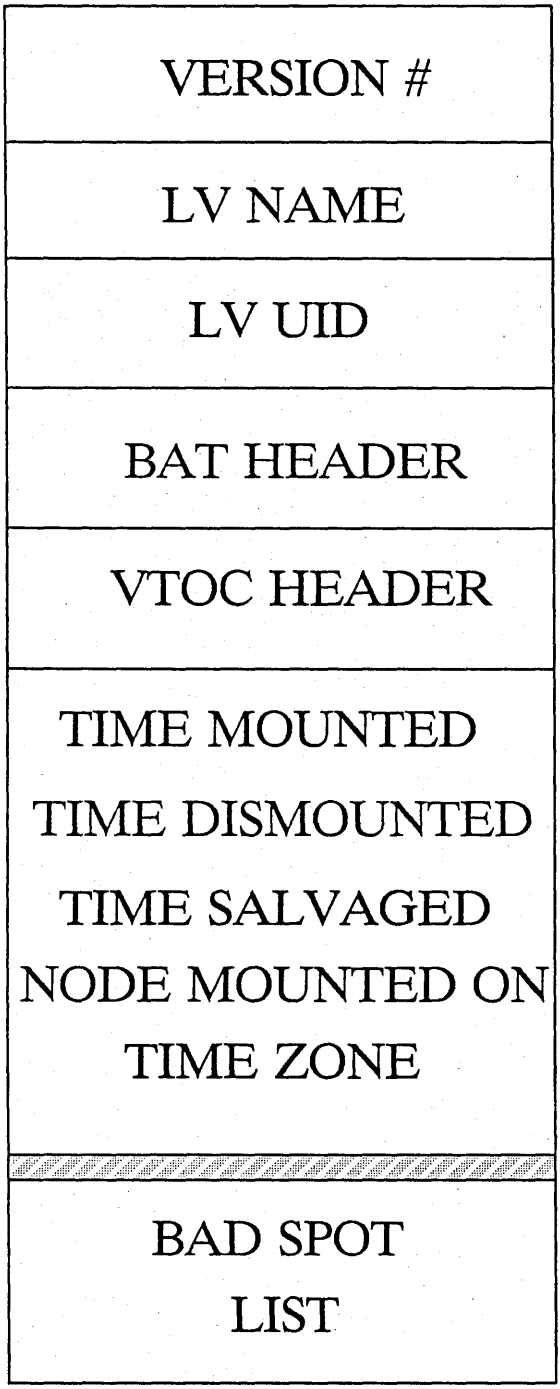
Describes the DISK

Locates Logical Volumes

(up to 10 per Physical Volume)

plus Alternate Logical Volume Labels

LOGICAL VOLUME LABEL



VERSION #

LV NAME

LV UID

BAT HEADER

VTOC HEADER

TIME MOUNTED
TIME DISMOUNTED
TIME SALVAGED
NODE MOUNTED ON
TIME ZONE

BAD SPOT
LIST

FREE BLOCK
MANAGEMENT

VOLUME TABLE
OF CONTENTS

VOLUME

MAINTENANCE

number of hash buckets
calculated in Invol from
average file size (5 blocks default
and total number of blocks.

use this to optimize applications

BAT HEADER

NUMBER OF BLOCKS REPRESENTED	fold to Invol
NUMBER OF FREE BLOCKS	lvolfs
DISK ADDRESS OF FIRST BAT BLOCK	
BLOCK NUMBER REPRESENTED BY THE FIRST BIT IN THE BAT	
NEEDS SALVAGING FLAG	

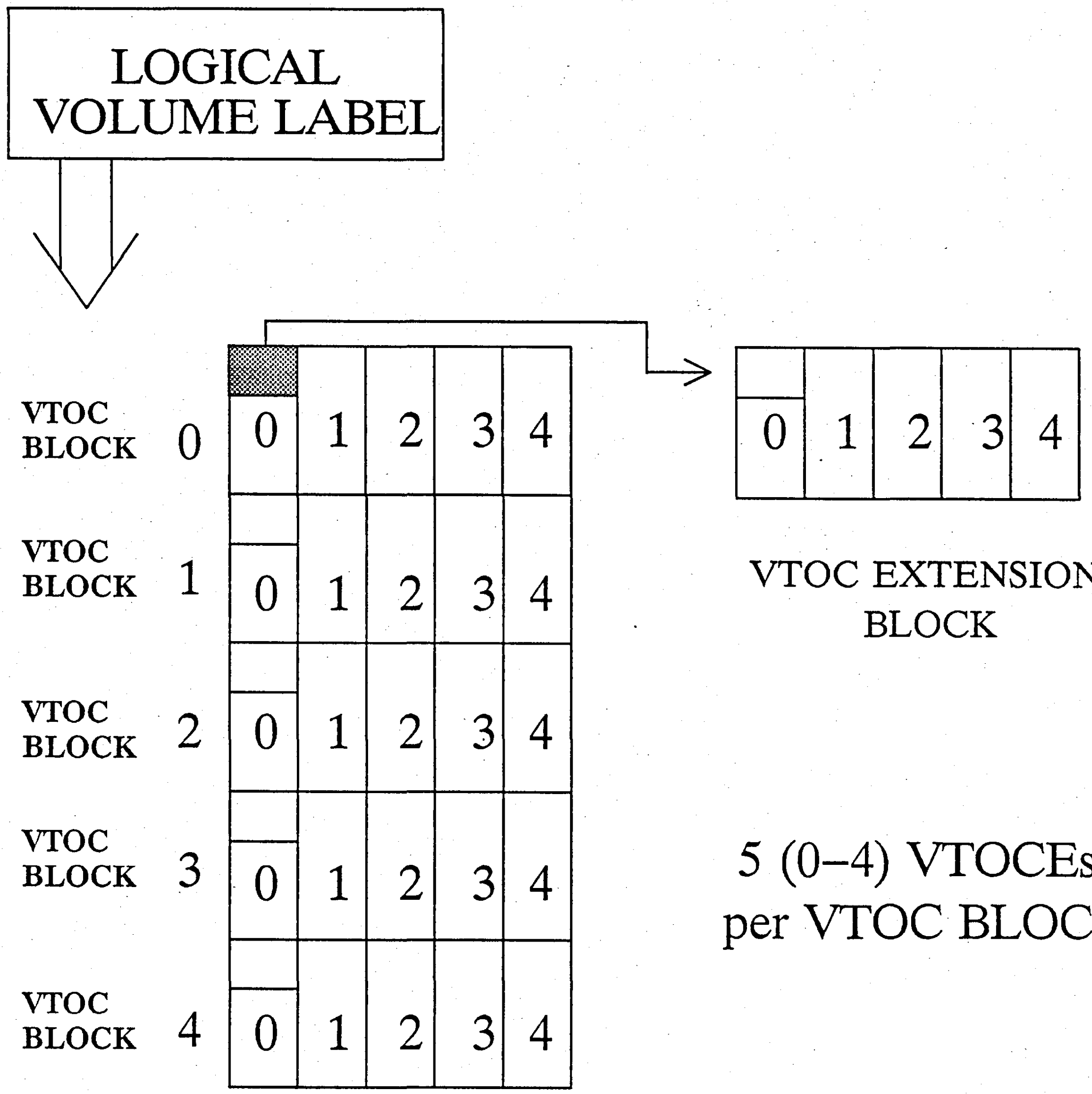
VTOC HEADER

NUMBER OF HASH BUCKETS
NUMBER OF BLOCKS USED
VTOCX OF NETWORK ROOT DIRECTORY
VTOCX OF LOGICAL VOLUME ENTRY DIRECTORY
VTOCX OF OS PAGING FILE
VTOCX OF SYSBOOT BOOT FILE
VTOC MAP

VTOC indexes
point to VTOC Entries

the paging file
is always contiguo
Used at boot time.
O.S paging file is
the backing store for
3 segments of the
O.S. (all at address
but two of them are
always wired
↓
las -bs

VOLUME TABLE OF CONTENTS
VTOC



LOGICAL VOLUME LABEL

VTOC BLOCK 0

VTOC BLOCK 1

VTOC BLOCK 2

VTOC BLOCK 3

VTOC BLOCK 4

VTOC EXTENSION BLOCK

5 (0–4) VTOCEs per VTOC BLOCK

? VTOCX (hashed VTOC index)

↓

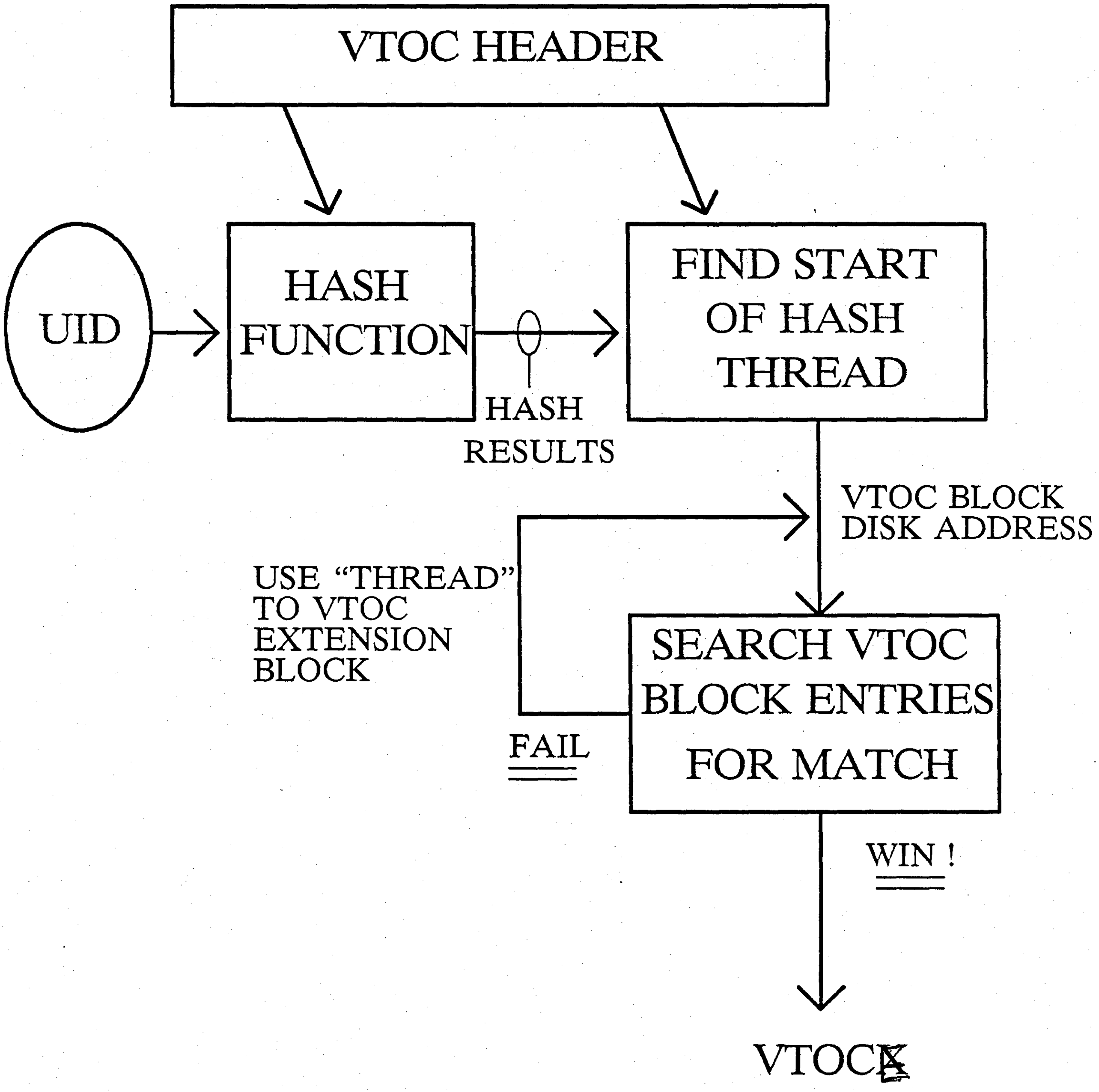
VTOCE (contains file map for object)

↓

file

Cataloging means entering a name/UID pair in a directory object.

USING THE VTOC



VTOC HEADER

UID

HASH FUNCTION

HASH RESULTS

FIND START OF HASH THREAD

VTOC BLOCK DISK ADDRESS

USE "THREAD" TO VTOC EXTENSION BLOCK

SEARCH VTOC BLOCK ENTRIES FOR MATCH

FAIL

WIN !

VTOC~~X~~E

VTOC ENTRY

VTCOE (vee–toe–chee)

HDR	DATA BLOCK POINTERS FOR SEGMENT #0	L1	L2	L3

32K
32 pages

0 file
1 dir
3 link

SYS TYPE	Permanent: can't delete PERM 1bit	immutable (not implemented) IMM 1bit	UID 64 bit	canned TYPE UID 64bit	CURR LEN	BLKS USED

ACL UID	DIR UID pointer to parent	DTU	DTM file version	REF CNT how many files using this object

acl's use ref. counts

LOCK KEY ~~not used~~

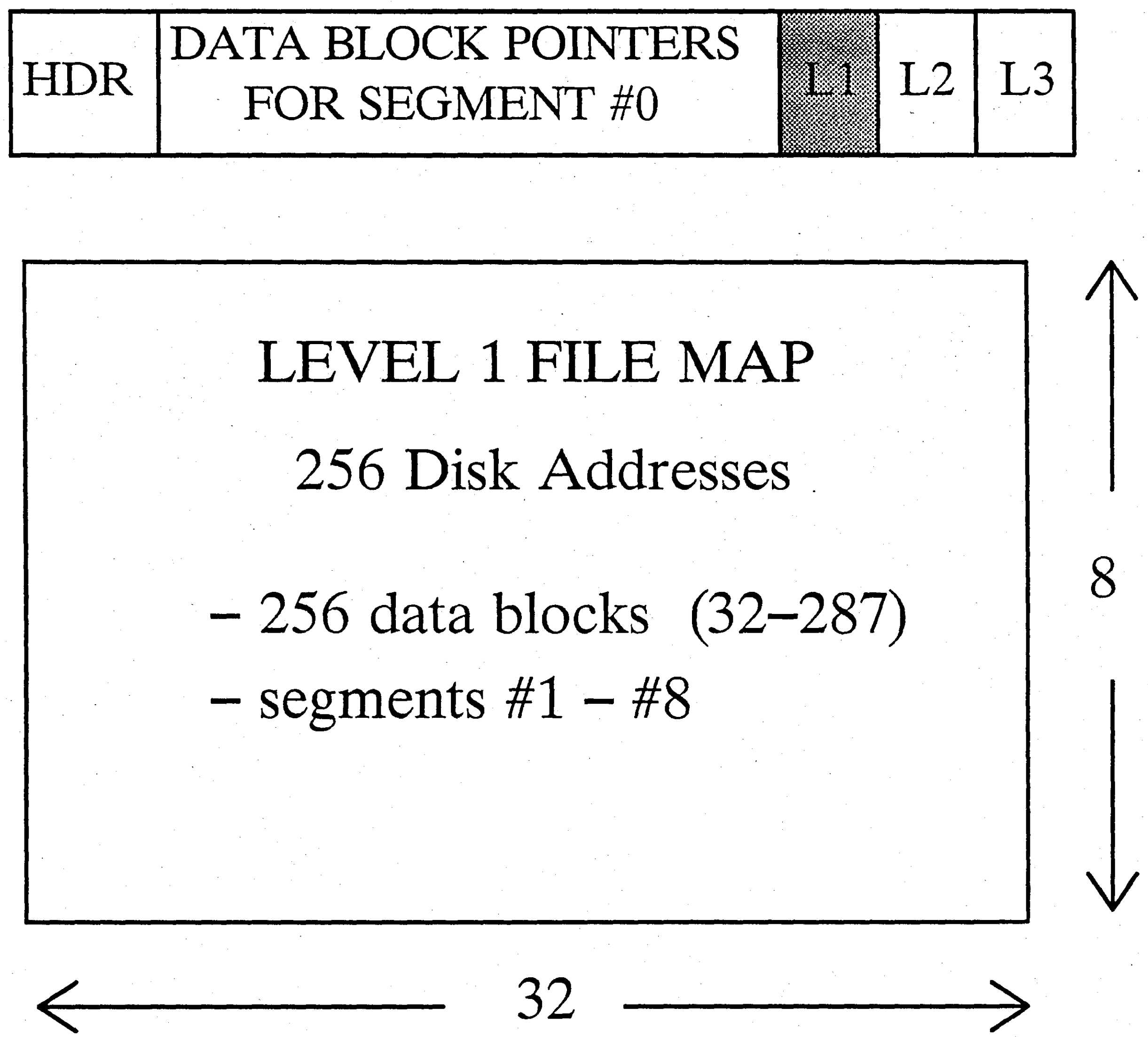
links aren't objects, they're only names.

VTOC HEADER

file_$ attributes call

AST is a cache over the VTOC

VTOC ENTRY
VTCOE (vee–toe–chee)



HDR

DATA BLOCK POINTERS FOR SEGMENT #0

L1

L2

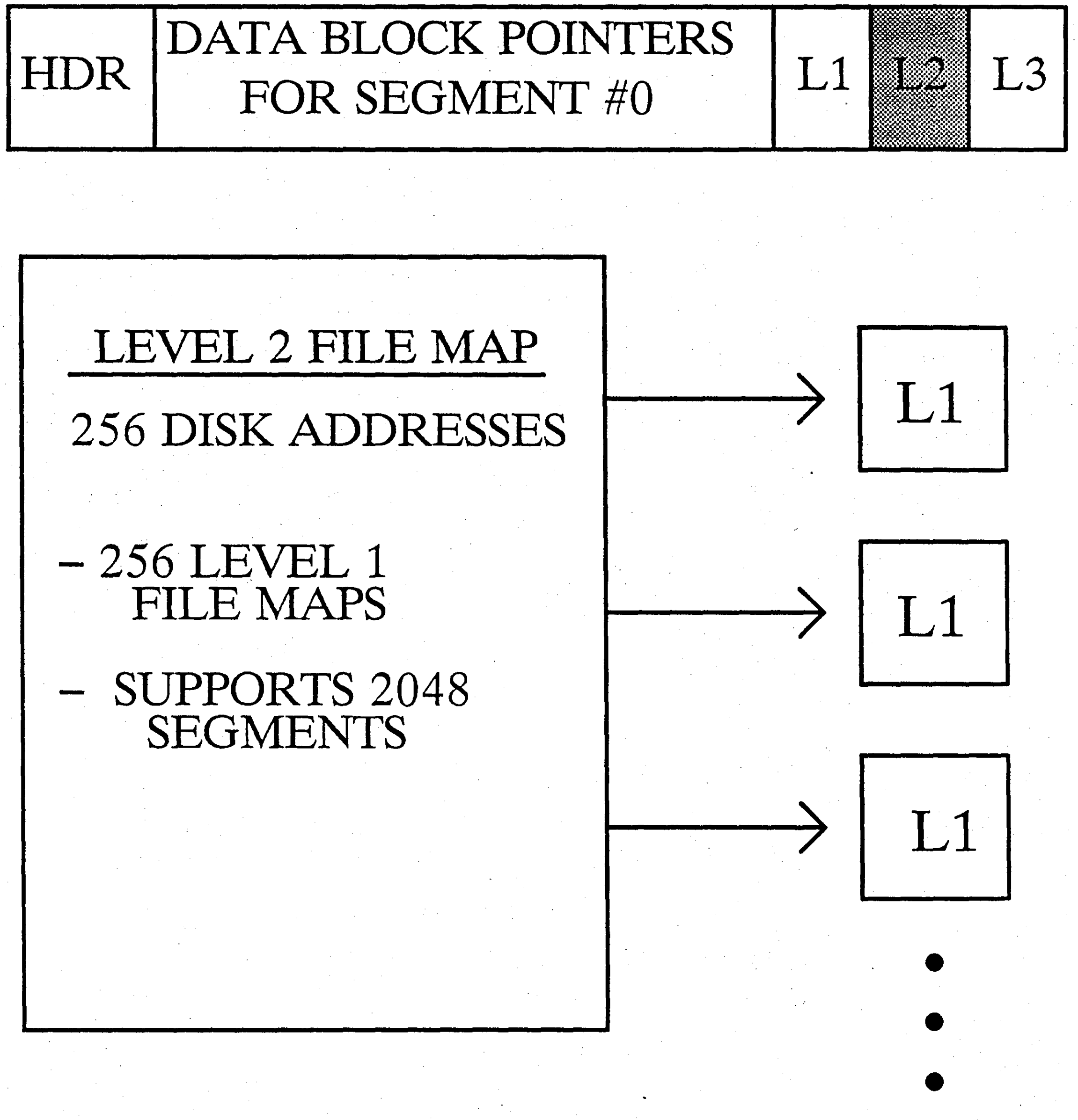
L3

LEVEL 1 FILE MAP

256 Disk Addresses

- 256 data blocks (32–287)
- segments #1 – #8

VTOC ENTRY
VTCOE (vee–toe–chee)



HDR | DATA BLOCK POINTERS FOR SEGMENT #0 | L1 | L2 | L3

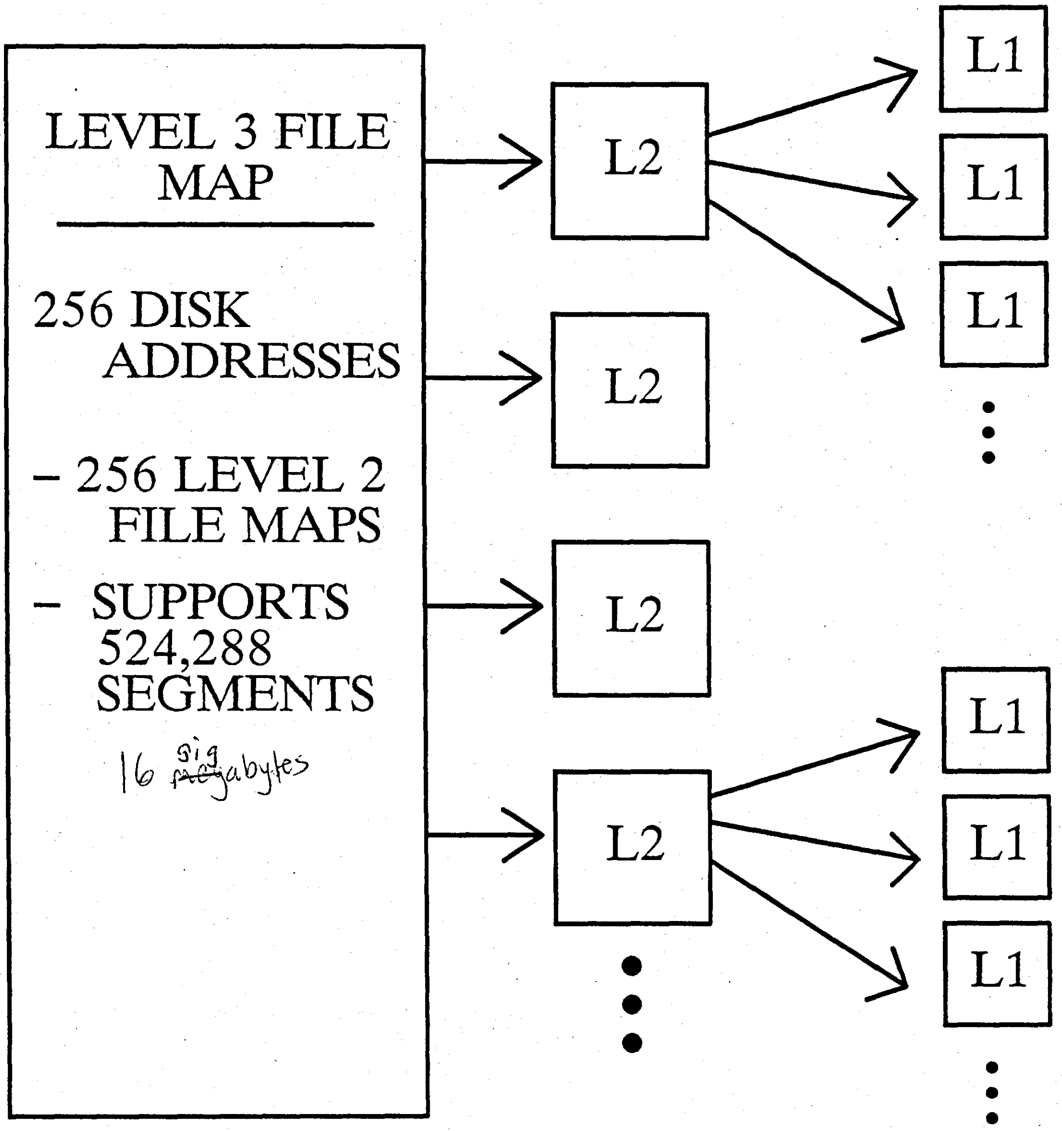
LEVEL 2 FILE MAP

256 DISK ADDRESSES

- 256 LEVEL 1 FILE MAPS
- SUPPORTS 2048 SEGMENTS

VTOC ENTRY
VTCOE (vee–toe–chee)

HDR	DATA BLOCK POINTERS FOR SEGMENT #0	L1	L2	L3



LEVEL 3 FILE MAP

256 DISK ADDRESSES

- 256 LEVEL 2 FILE MAPS
- SUPPORTS 524,288 SEGMENTS

16 ~~mega~~ gigabytes

Penalty for big files: Maybe 7 seeks to get a page.

More physical memory -> more AST (active segments)

Example

create backing store on disk

- FILE_CREATE (LOCal_UID, UID, ST)

  1. Find the volume that holds LOC_UID
  2. Call UID_$GEN to get a UID
  3. Build a VTOCE–hea~~g~~der for the new file.
  4. Add the VTOCE to the VTOC

DONE!

Allocating Blocks on Disk

- Strategy

  * Nearest available block to last allocated block

  * taking into account the "BAT" step

- Mechanism

  * Read the appropriate part of the "BAT" into memory

  * Find FREE blocks and change in memory copy of BAT (Write it back later . . .) in memory most of the time

Note: SALVOL's biggest job is to fix the BAT, since the ON–DISK copy is almost always out–of–date!

Salvol reads all the VTOCE's to update the BAT (rebuild it)

Apollo Virtual Memory

- The Idea

  - * Lots of processes with independent address spaces (256 MB or 16 MB)

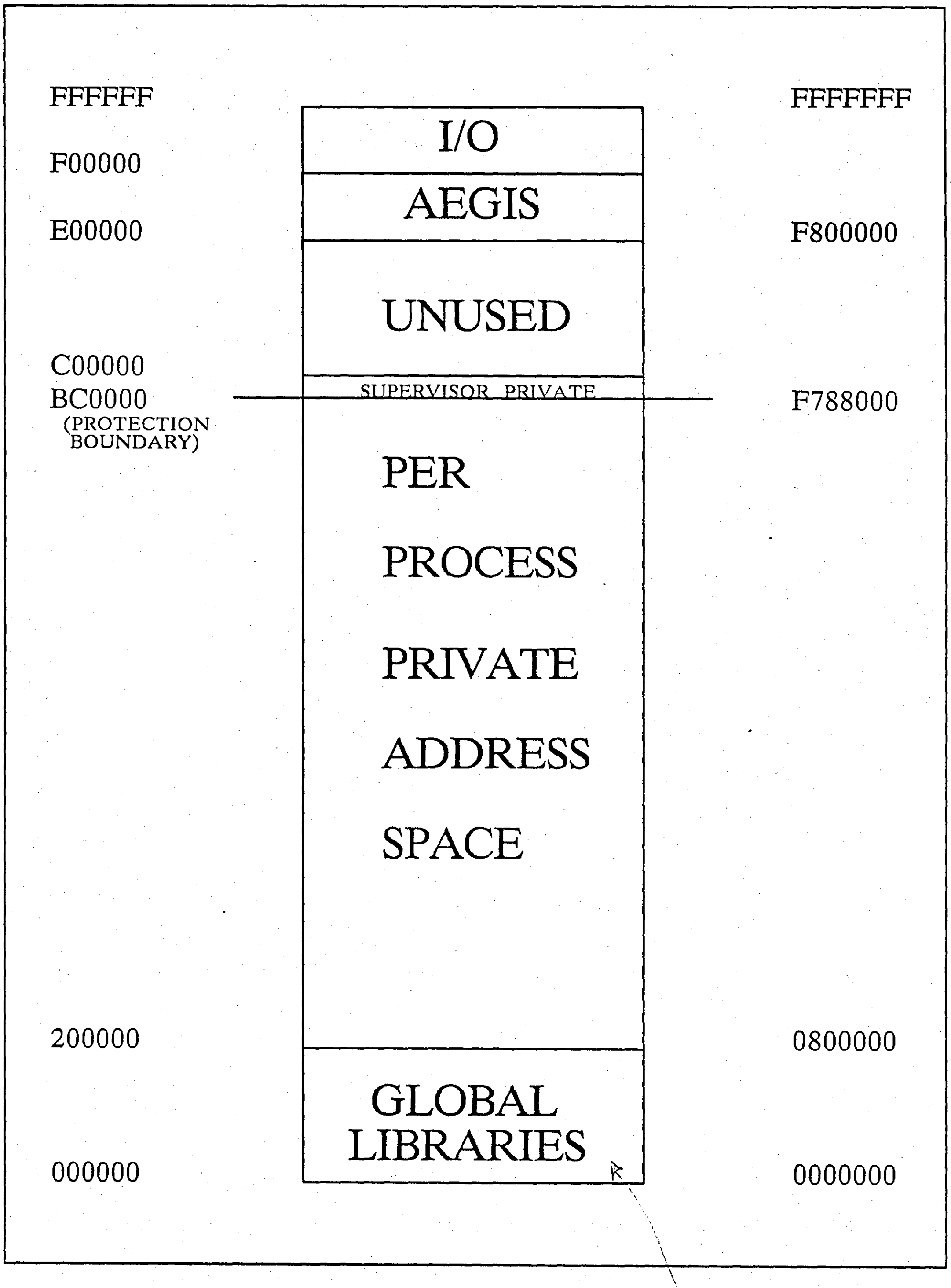
  - * Some stuff GLOBAL to all processes

  - * Divide A.S. into 32 Kbyte segments

  - * Divide objects into 32 Kbyte segments

  - * Some processes will live only in the nucleus and won't need private space. . .only GLOBAL!

PROCESS ADDRESS SPACE



FFFFFF
F00000
E00000
C00000
BC0000
(PROTECTION BOUNDARY)
200000
000000

I/O
AEGIS
UNUSED
SUPERVISOR PRIVATE
PER PROCESS PRIVATE ADDRESS SPACE
GLOBAL LIBRARIES

FFFFFFF
F800000
F788000
0800000
0000000

env installs
a canned list of names.

Virtual Memory Glossary

- ASID: Address Space Identifier

0 is Aegis
1 is DM or SPM

- MAPPING
  - * Binding V.A. Segments with OBJECT Segments

- MST: Mapped Segment Table (One per process)

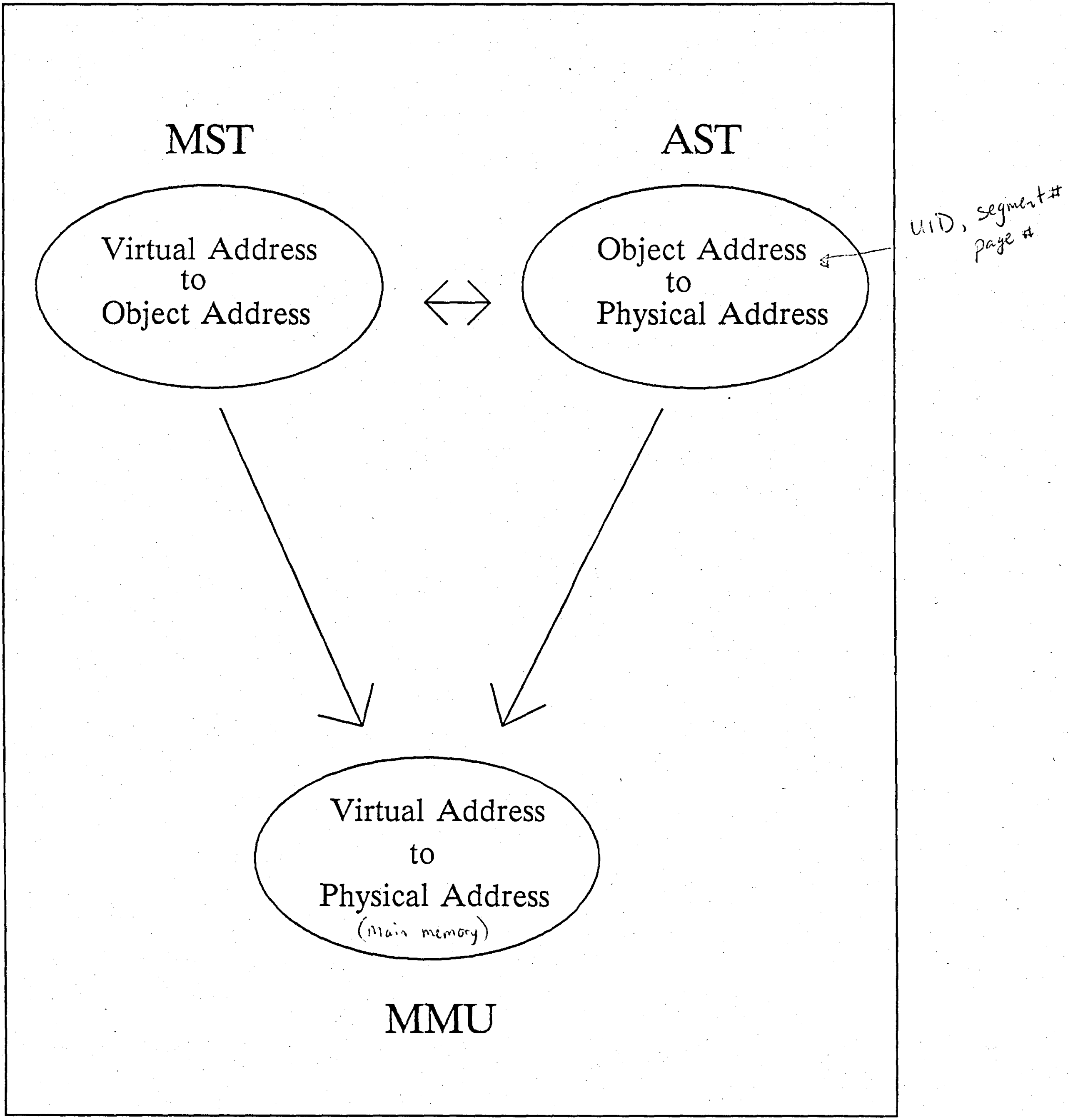
- Active Segments
  - * Object segments whose information and data are cached in physical memory.

- AST: Active Segment Table

- PMAP
  - * Disk Address & Physical Address (if resident) of each page in an object segment

VIRTUAL MEMORY

The Main Players



MST

Virtual Address to Object Address

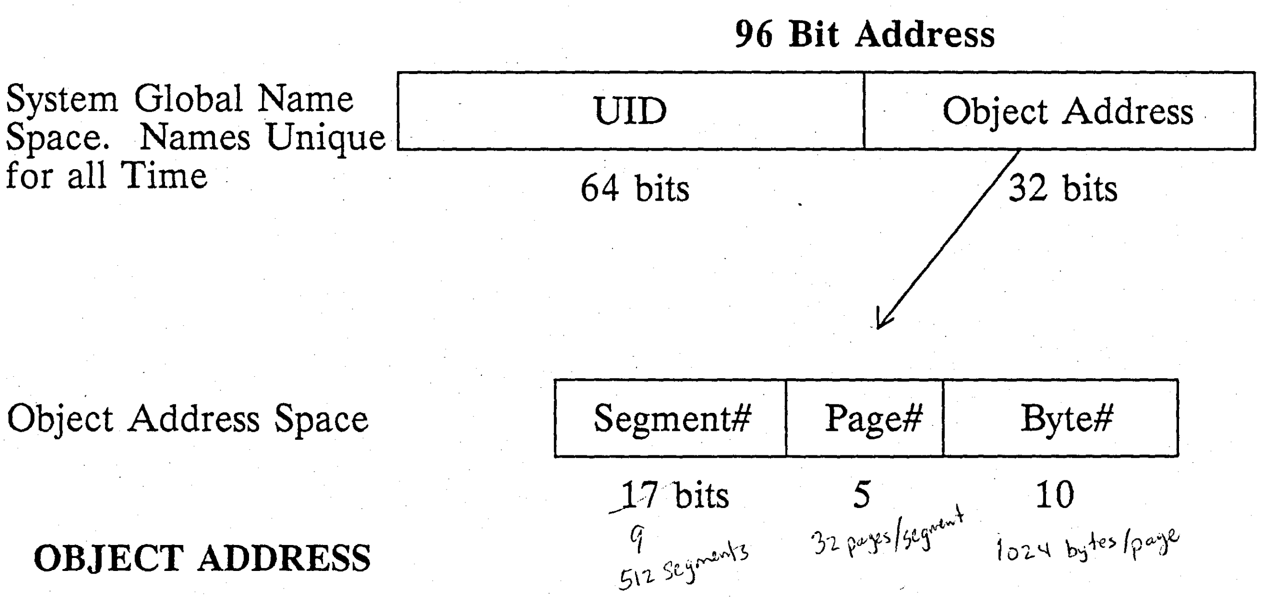
AST

Object Address to Physical Address

UID, segment # page #

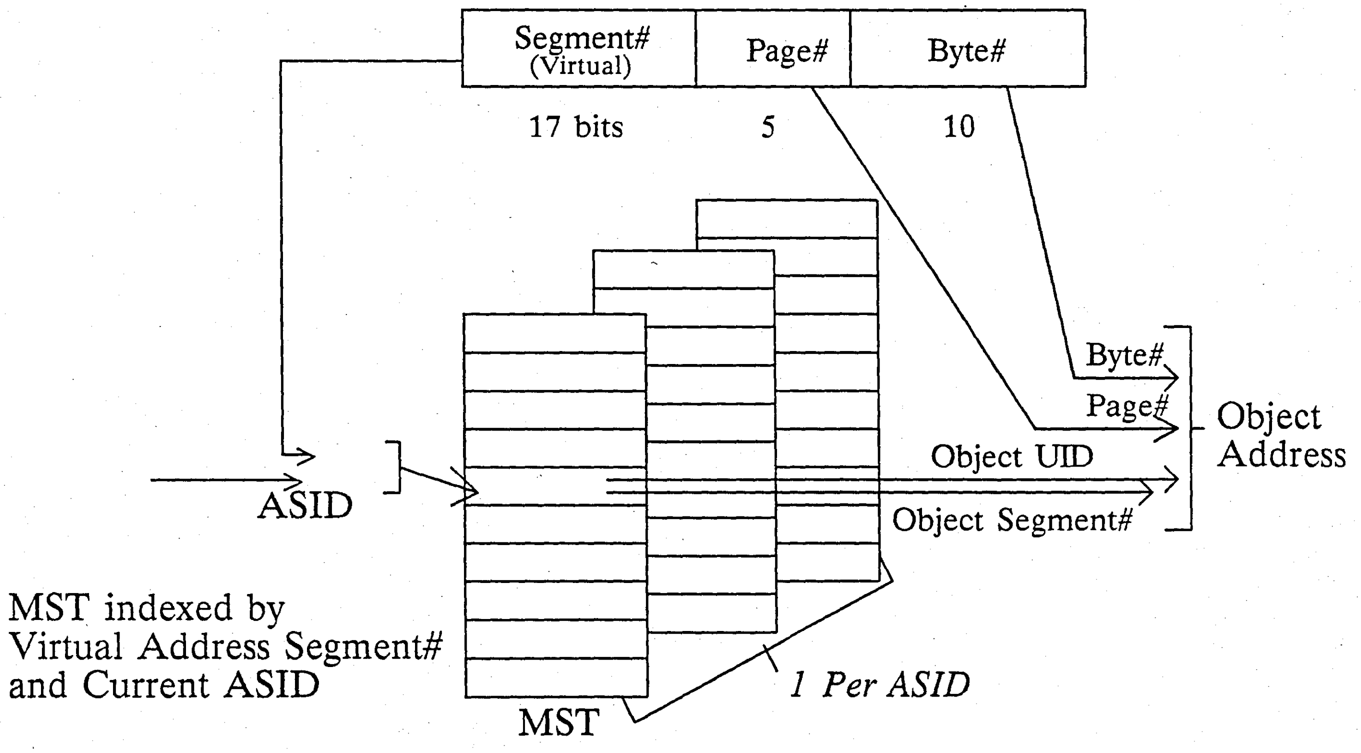
Virtual Address to Physical Address (main memory)

MMU



OBJECT ADDRESS

VIRTUAL ADDRESS



TERN (DNX60) Virtual Addressing

–> Virtual Addressing differs slightly

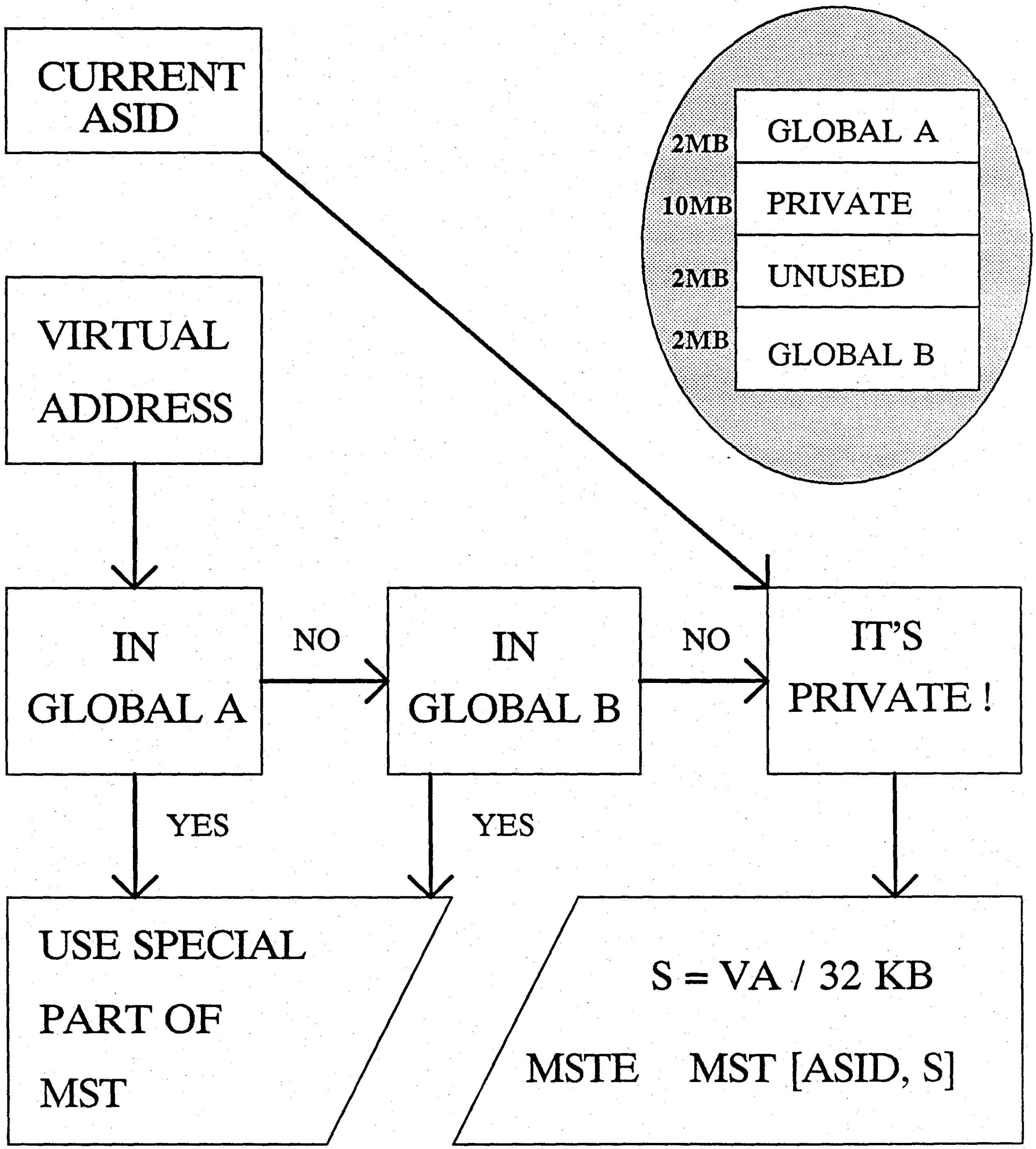
Region#	*Segment#*	*Page#*	*Byte#*
5	*12*	*5*	*10*

Why: 1) Simplifies table organization for big address space

2) Simplifies hardware/microcode

BUT: it's transparent to everyone but AEGIS memory management code

Finding the RIGHT MST



CURRENT ASID

VIRTUAL ADDRESS

IN GLOBAL A

NO

IN GLOBAL B

NO

IT'S PRIVATE !

YES

YES

USE SPECIAL PART OF MST

S = VA / 32 KB

MSTE   MST [ASID, S]

MAPPED SEGMENT TABLE ENTRY

(MSTE)

OBJECT UID	UID of the Object
OBJECT SEGMENT NUMBER	Segment within the Object
EXTEND OK FLAG	Can the File be Extended ?
ACCESS	Access Righs
GUARD	Is this a Guard Segment ?
HINT ASTE INXED DE	Performance Enhancement
LOCATION VTOCX	Disk or Network

- Now improved with “Touch Ahead Count”

THE ACTIVE SEGMENT TABLE

- An Array of AST Entries (ASTEs)

- Each ASTE is a cache entry over the VTOC

ASTE HEADER	OBJECT SEGMENT PAGE MAP (PMAP)

- ASTE Header

  * Object UID
  * Object Segment Number
  * ACL UID
  * Location   if remote, MST maps it on remote AST
  * DTM

- Object Segment Page Map

  * 32 PMAP Entries (PMAPEs); one per page in the segment
  * Current PPN   physical page number
  * Disk Address (DADDR)

Object Address –> Physical Address (UID, SEG#, PAGE#, BYTE#)

1. Find ASTE for (UID, SEG#). If not in AST, read VTOC and fill in an ASTE.

2. Look in PMAP for the ASTE to get the disk address for page “PAGE#”.

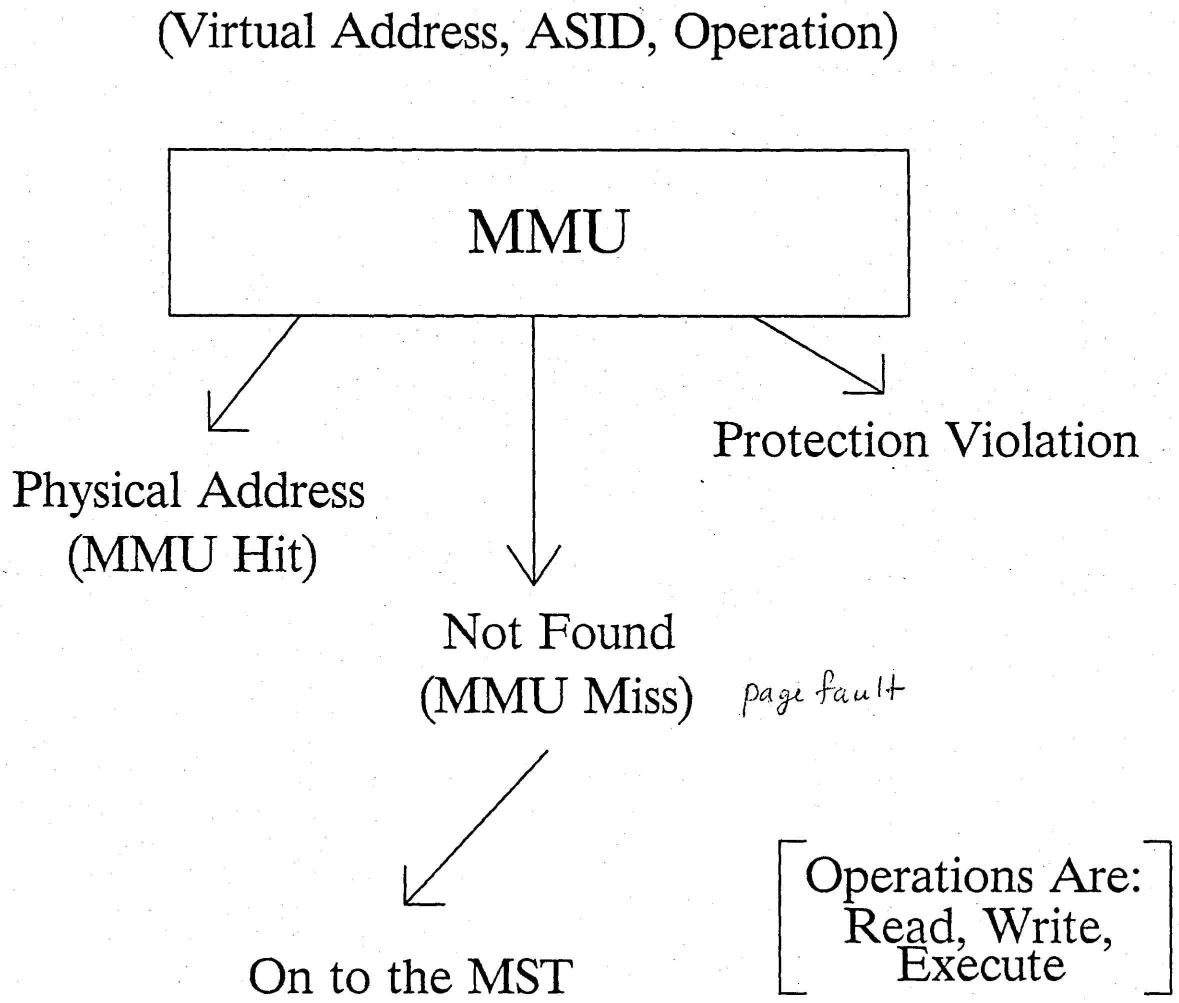
3. Find a free physical memory page.

4. Read the disk.

5. Update the PMAP.

6. Load the MMU (so it can succeed next time!).

Memory Management Unit (MMU)



(Virtual Address, ASID, Operation)

MMU

Physical Address
(MMU Hit)

Not Found
(MMU Miss) page fault

Protection Violation

On to the MST

Operations Are:
Read, Write,
Execute

VIRTUAL ADDRESS
TO
OBJECT ADDRESS

Any Object Segment may be:

— MAPPED BUT NOT ACTIVE

not activated until you touch it, (refer to it)

— ACTIVE BUT NOT MAPPED

when objects are unmapped they aren't automatically removed from AST. Another example, activated on remote node with hold count.

— MAPPED TO MORE THAN ONE ADDRESS SPACE SEGMENT WITHIN A SINGLE ADDRESS SPACE

— MAPPED TO DIFFERENT ADDRESS SPACE SEGMENTS IN DIFFERENT PROCESSES

Same ASTE

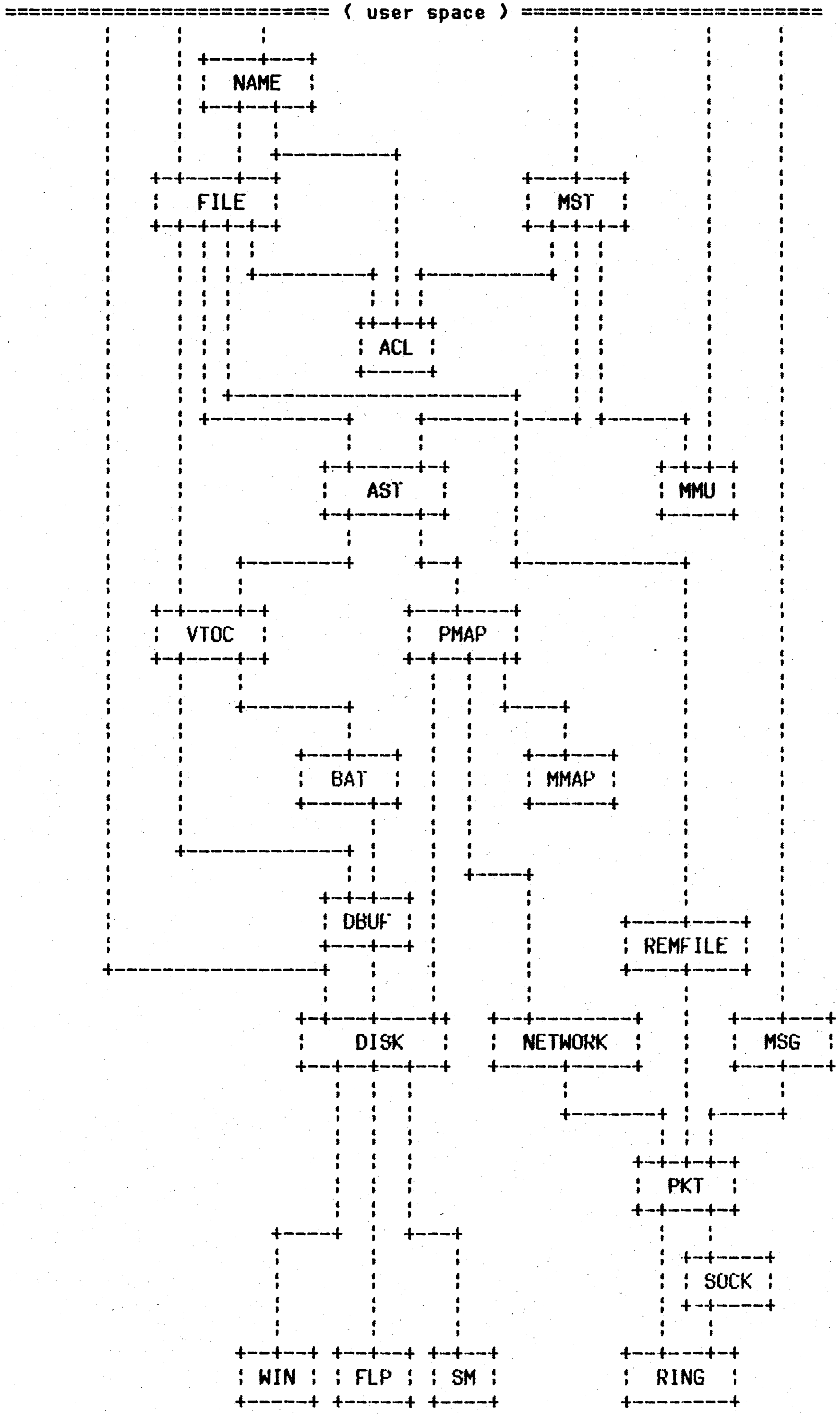
MST

Virtual Address	UID	segment #	location	access	
300000	U_a	0	Node – 2	r w	ASID 1
308000	U_a	1	Node – 2	r w	
301000	U_b	0	Node – 2	r	
300000	U_b	0	Node – 2	r	ASID 2
308000					
301000	U_b	0	Node – 2	r	

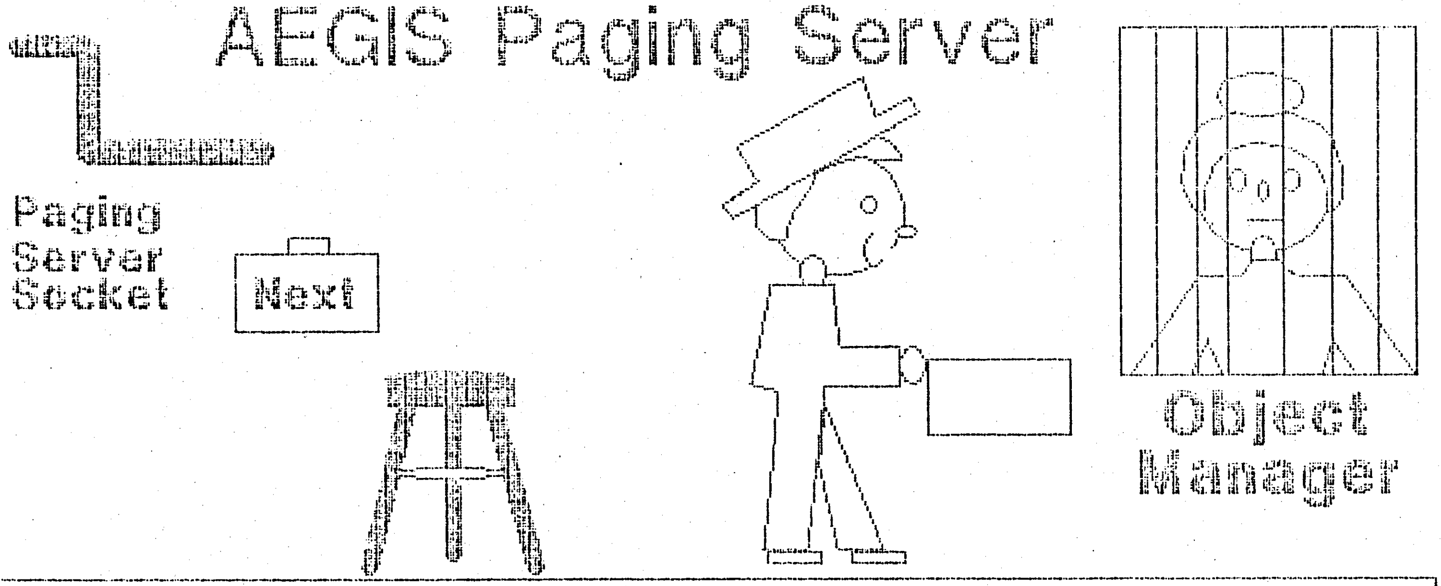
AST

UID	segment #	attribs	page map
U_a	1	....	(32 daddrs & ppns)
U_b	0	....	(32 daddrs & ppns)
U_a	0	....	(32 daddrs & ppns)

EXAMPLE: MST & AST in a running system


(user space)
NAME
FILE
MST
ACL
AST
MMU
VTOC
PMAP
BAT
MMAP
DBUF
REMFILE
DISK
NETWORK
MSG
PKT
SOCK
WIN
FLP
SM
RING

AEGIS Paging Server



Paging Server
MENU OF SERVICES

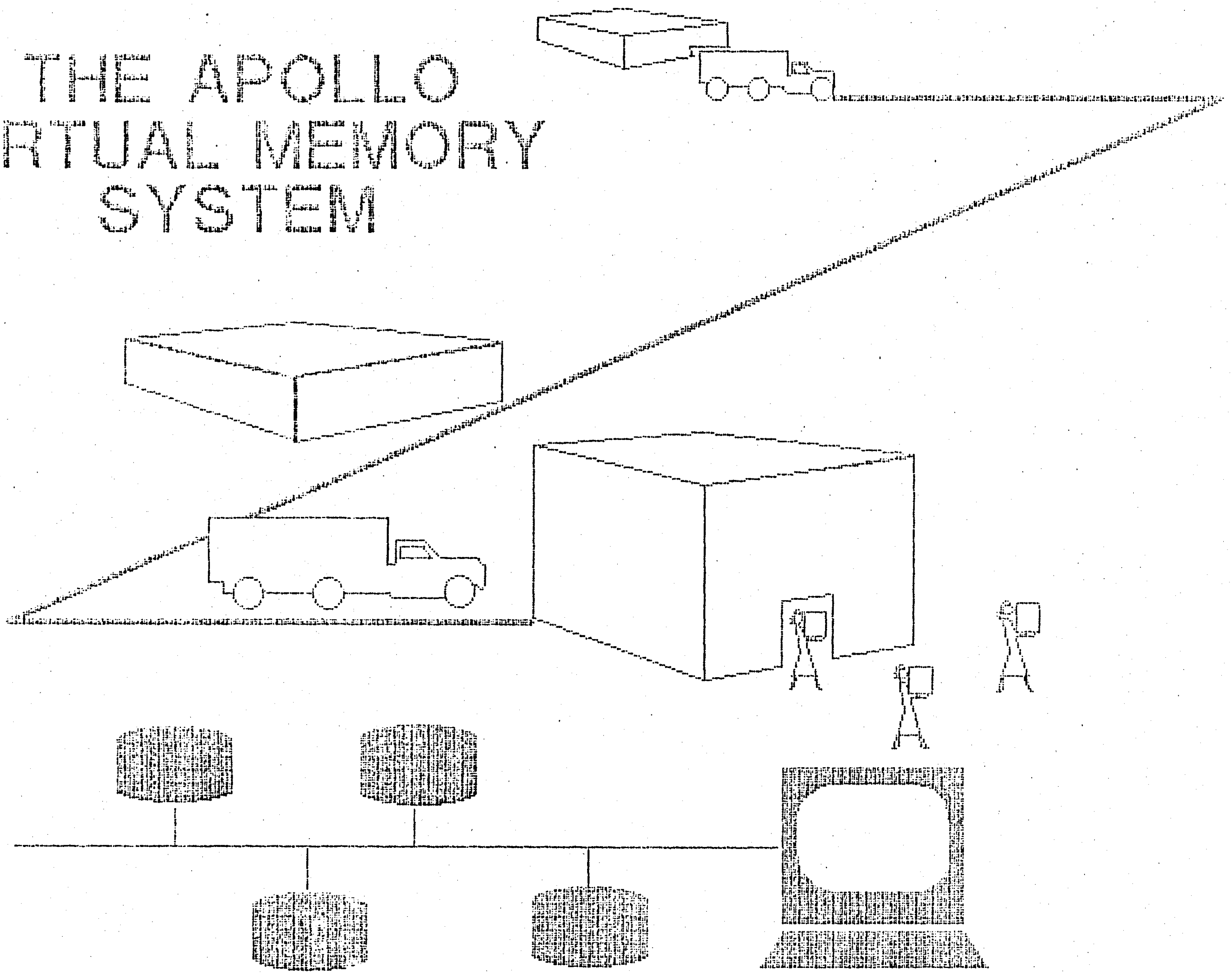
☒ Page-In
UID 12345678.9ABCDEF0
Page # 7

□ Page-Out
UID ____
Page # ____
Data ____

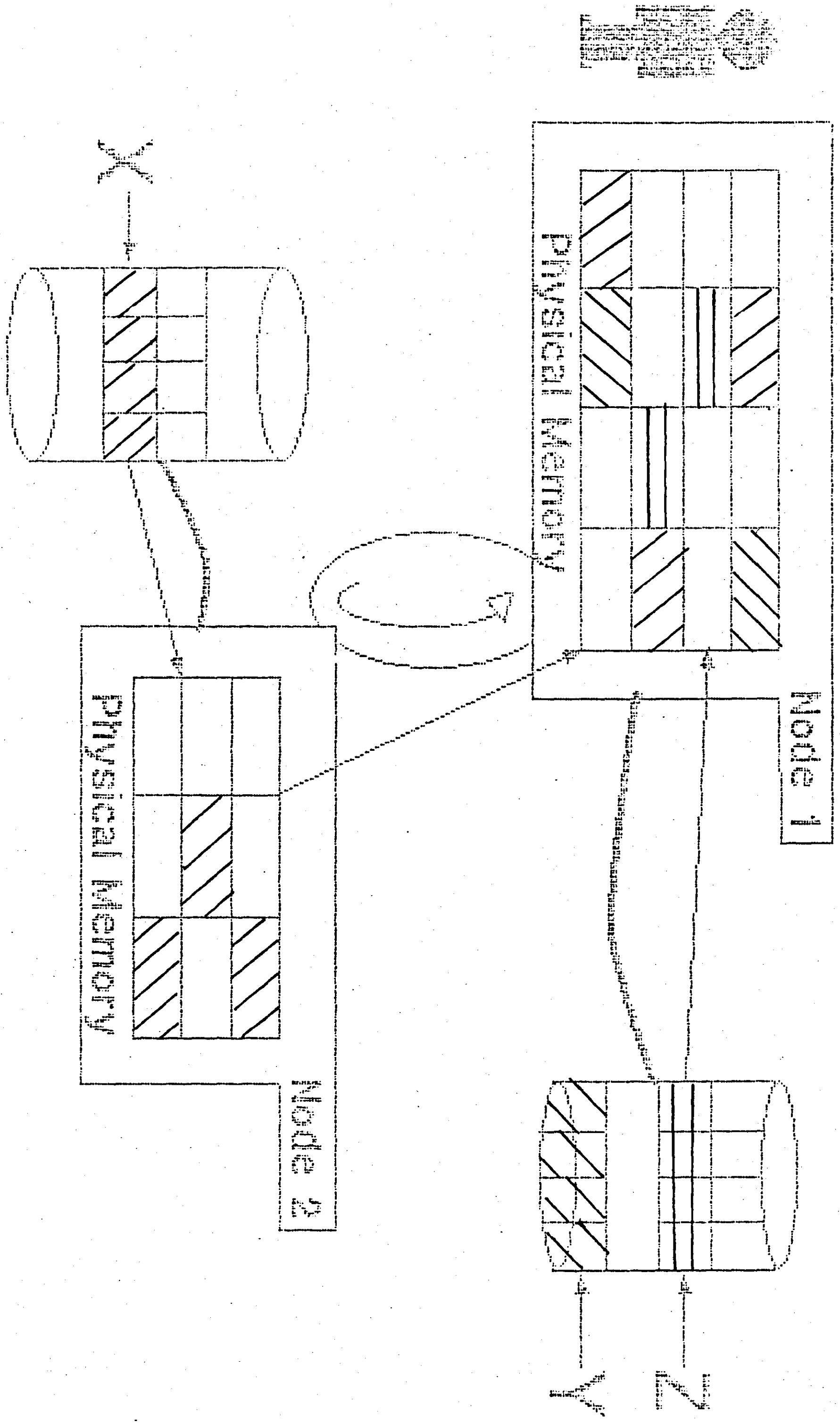
□ Info Request
UID ____
Type of Info Desired ____

□ Echo

THE APOLLO VIRTUAL MEMORY SYSTEM



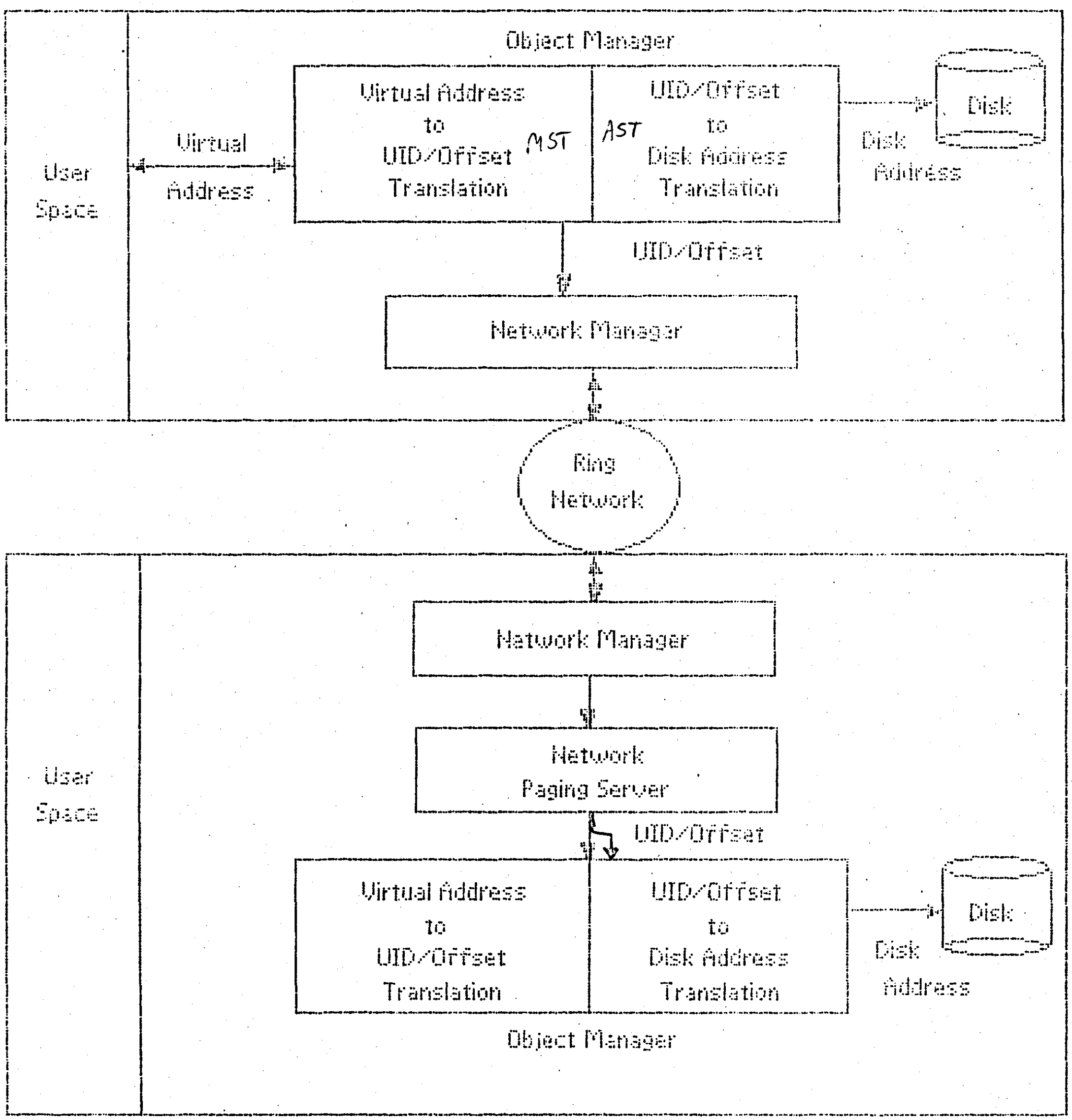
Network-wide Virtual Memory



netsvc -P (paging remote server pool size)

(minimum size 50)

DOMAIN VIRTUAL MEMORY SYSTEM



NETWORK FILE SYSTEM

Remote–file server

handles file level operations

lock, unlock, directory–lookup,
get–attributes, create, delete

Arguments are passed from the client to the server, the server executes the call and passes back the answer.

Remote paging server

handles paging operations

page–in, page–out, attributes

based on unique **object addresses**

(uid, segment #, page #)

FILE SERVER

Menu of Services

File Services

- LOCK
- UNLOCK
- CREATE
- DELETE
- TRUNCATE
- INFORMATION
- NAME LOOKUP

⋮

Node Information Services

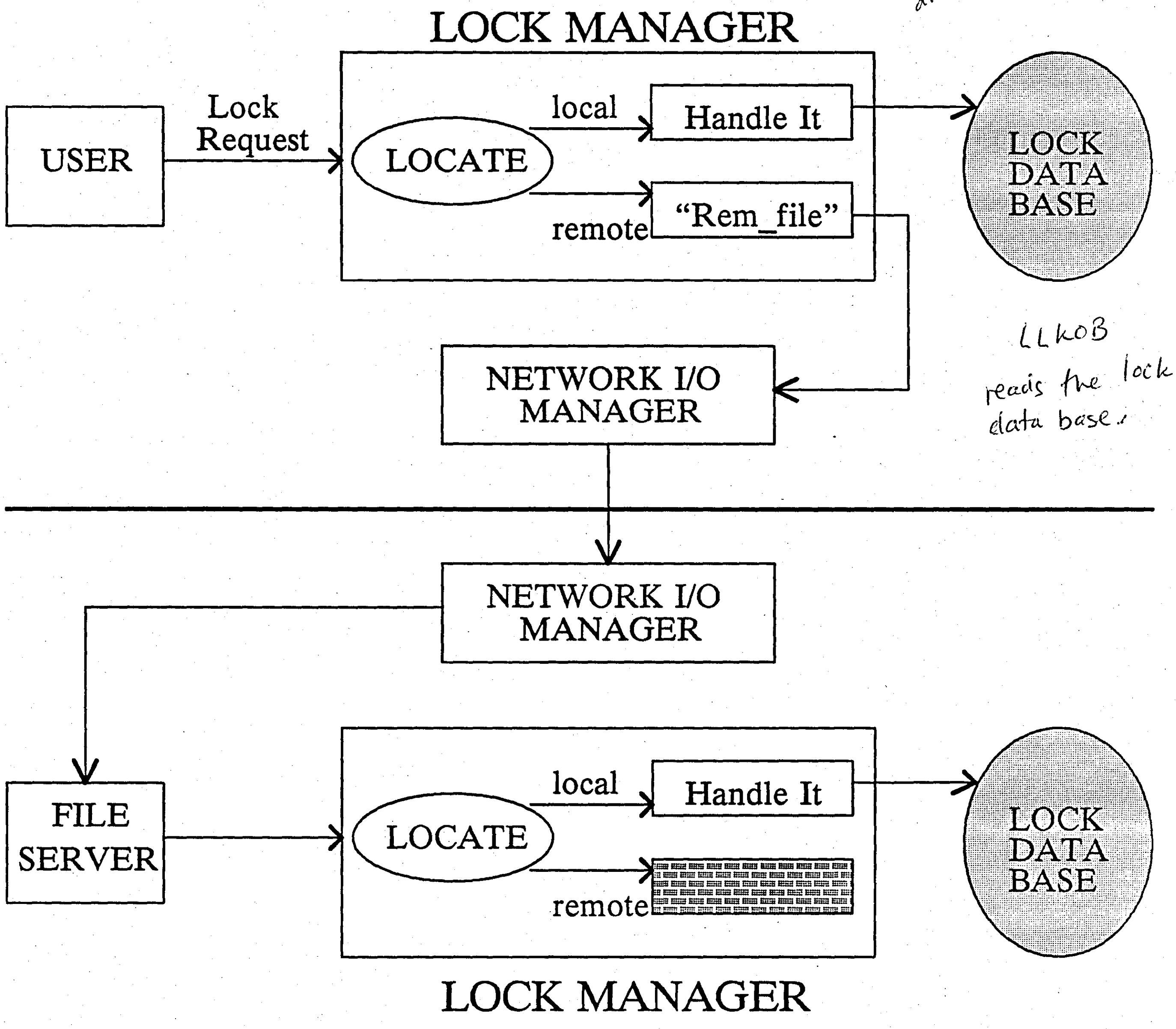
- VOLUME FREE SPACE
- ACTIVE PROCESS INFORMATION
- I/O STATISTICS
- TIME
- HELP WITH LCNODE

⋮

LD-51

LOCK REQUEST

To the lock manager: files + objects have homes. Files are always locked locally.



LOCK MANAGER

USER

Lock Request

LOCATE

local

Handle It

remote

"Rem_file"

LOCK DATA BASE

NETWORK I/O MANAGER

NETWORK I/O MANAGER

FILE SERVER

LOCATE

local

Handle It

remote

LOCK DATA BASE

LOCK MANAGER

LLKOB reads the lock data base.

LOCKING OBJECTS

CONCURRENCY CONTROL (2 models)

(1) n readers XOR 1 writer

any number of readers,
or exactly one writer.

(2) cowriters

any number of readers,
or any number of writers all from the same node.

Shared memory?

the object doesn't have to be on the same node as the writers.

LOCKING MODES (3 kinds)

(1) READ ONLY

(2) READ & WRITE

(3) READ – INTENDING – WRITE

(warning that I'll change to READ & WRITE before I'm done)

THE ROLE OF THE LOCK MANAGER

Enforce concurrency rules at lock time

Control all LOCAL files

Cooperate on REMOTE files

Maintain the LOCK TABLE

Support the distributed system

Help manage the object caches
(flushing when needed)

Pass authorization information
to paging system through
the object's **lock key.**

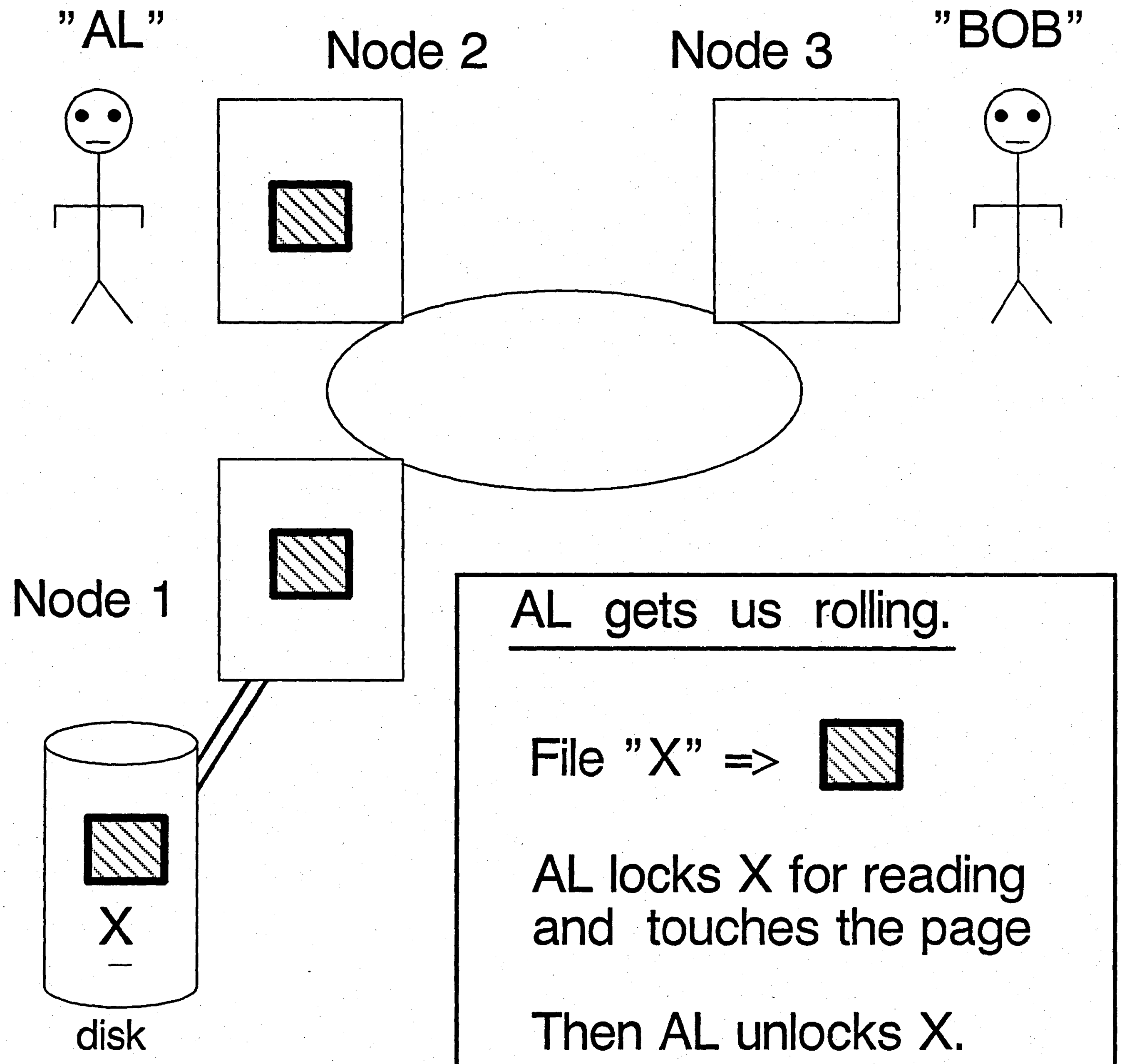
Lock Managers Tools

- Lock Table: Database
- Authorization Control
  - * Set Object Lock–key

    ZERO means read–only

    NODE_ID means only that node may write
- V. M. Cache Control
  - * Get object DTM
  - * Flush cache if needed
  - * Purify

    send changes home



STEP 1

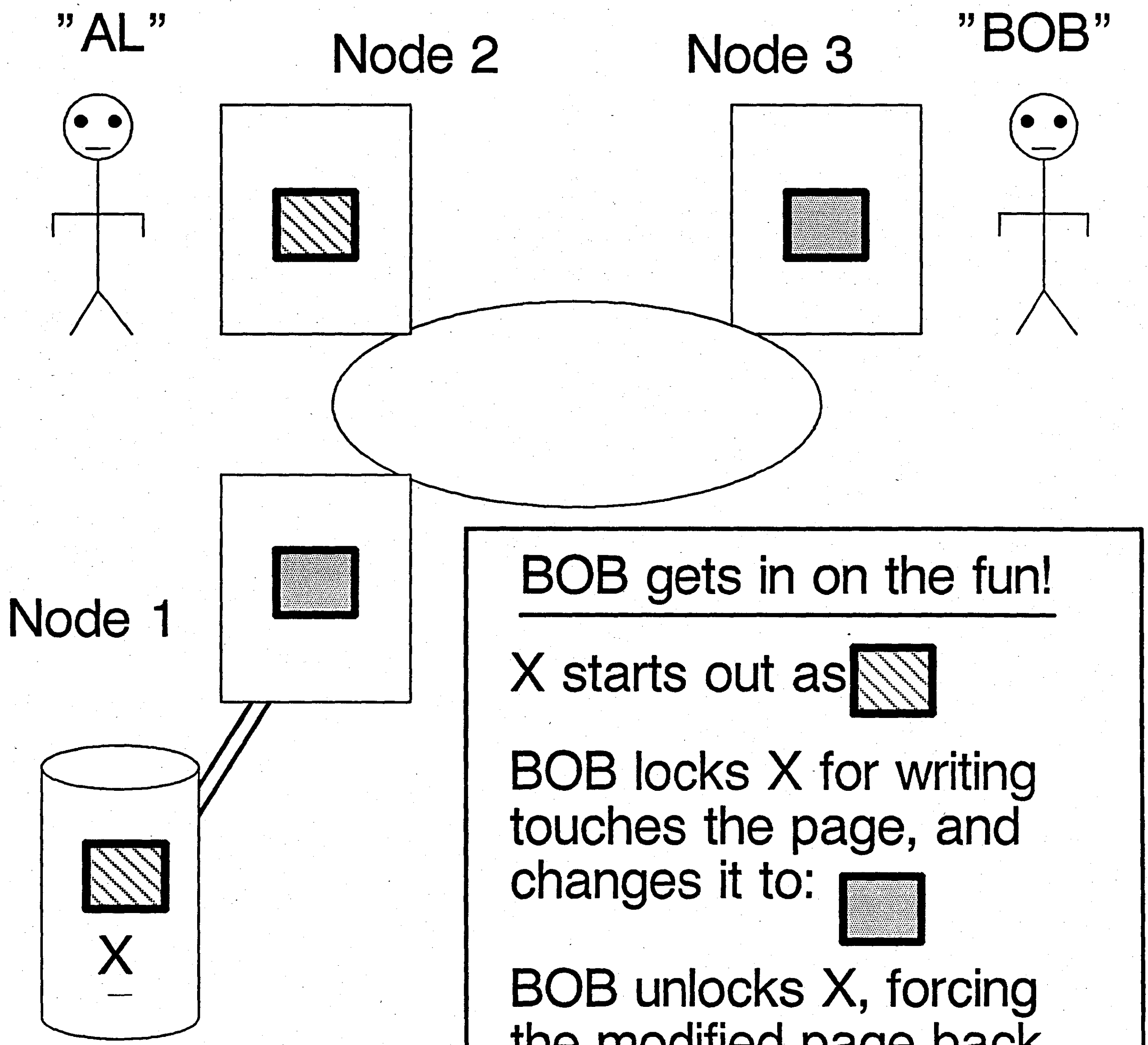
AL gets us rolling.

File "X" => ▨

AL locks X for reading and touches the page

Then AL unlocks X.

Note that Node 2 keeps it's copy of X in case it's needed again soon.



STEP 2

BOB gets in on the fun!

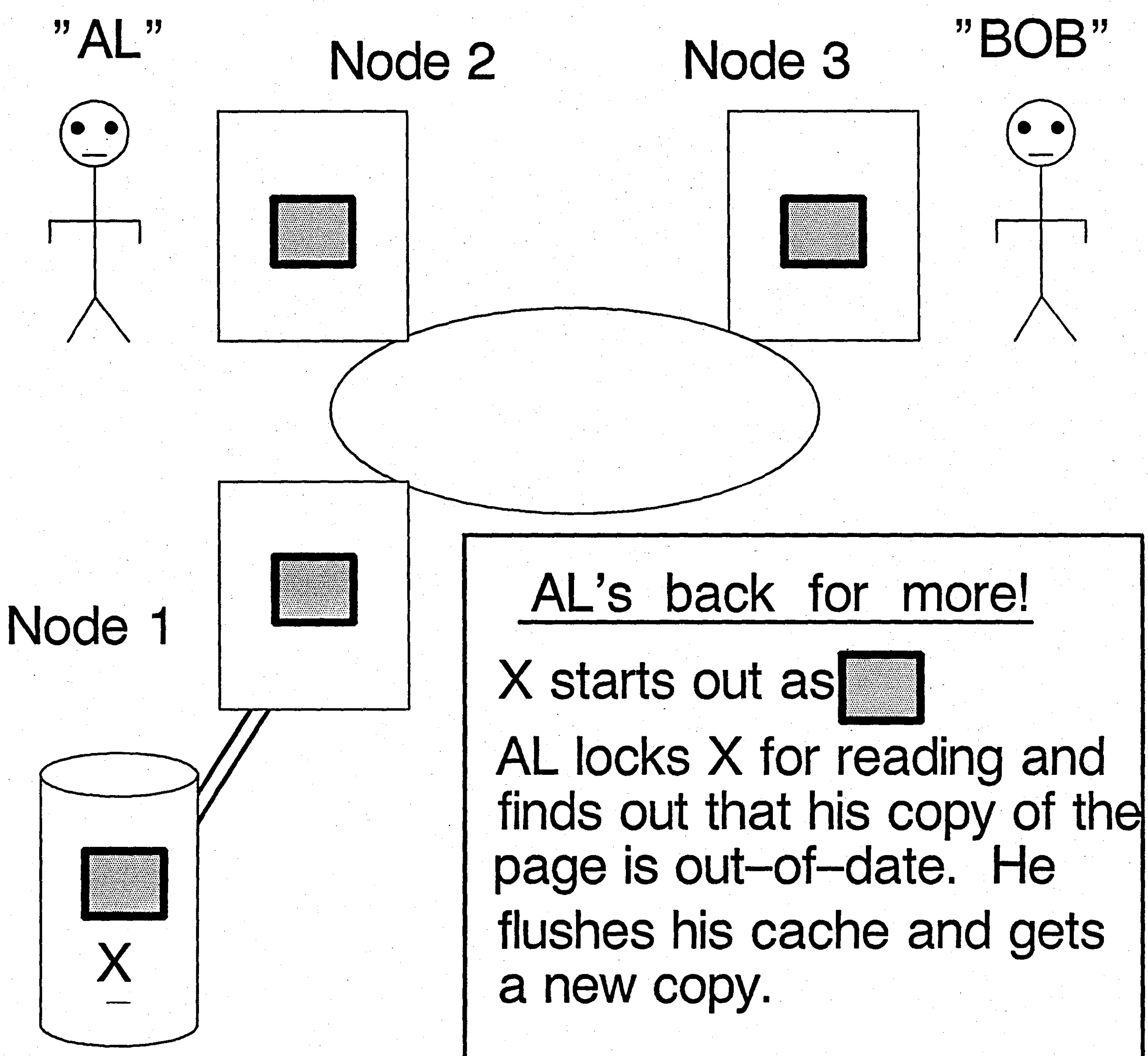
X starts out as ▨

BOB locks X for writing touches the page, and changes it to: ▒

BOB unlocks X, forcing the modified page back to Node 1.

Note that Node 2 doesn't know.

Note that the disk doesn't get updated right away.



AL's back for more!

X starts out as ▨

AL locks X for reading and finds out that his copy of the page is out–of–date. He flushes his cache and gets a new copy.

Note that if X hadn't changed, AL wouldn't have needed a new copy.

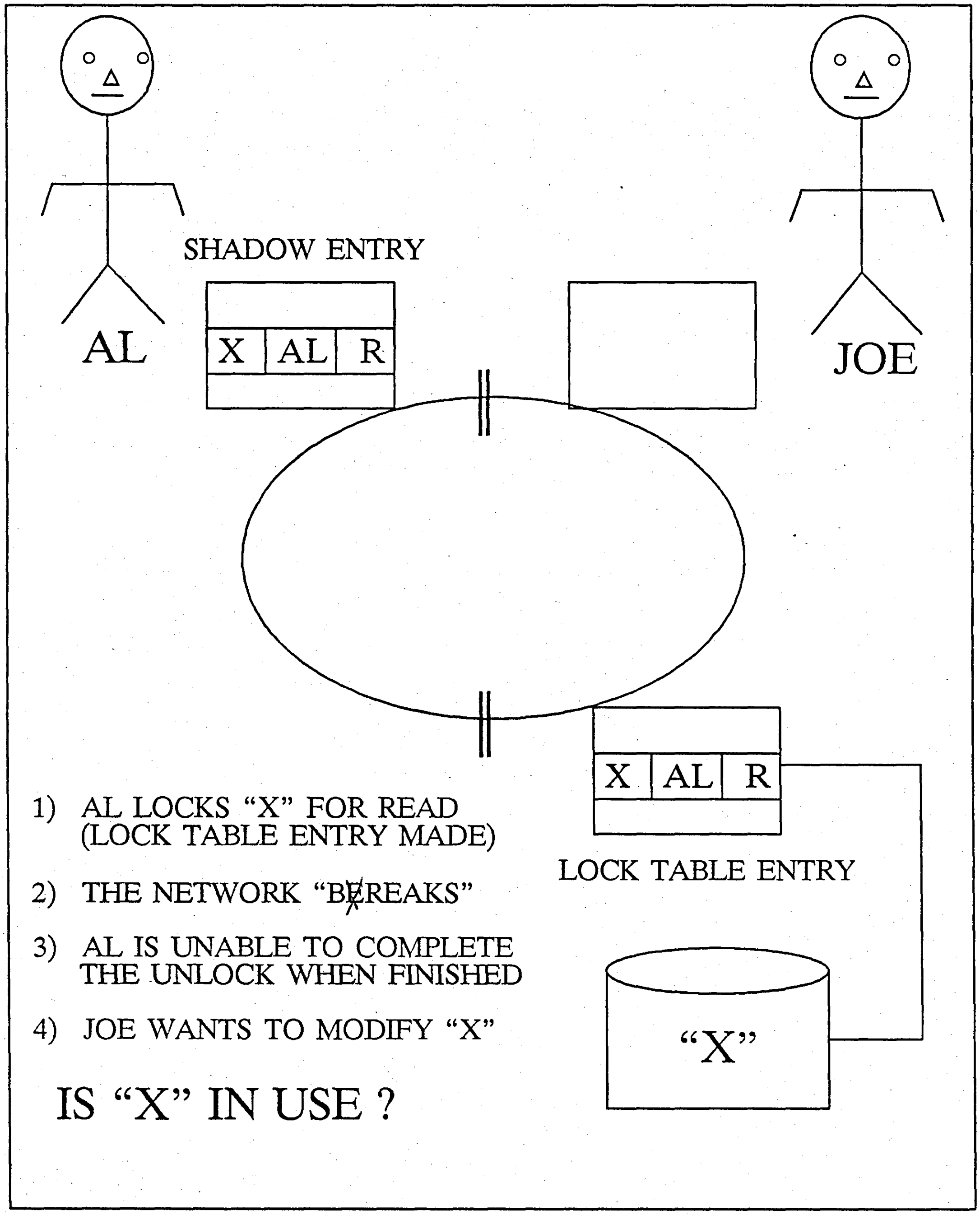
Note that AL's bad copy of the page isn't flushed until AL locks X again.

STEP 3

page purifier

writes modified pages to disk --
it "purifies" the page.
(modified objects are "impure")

ORPHAN LOCKS



1) AL LOCKS “X” FOR READ (LOCK TABLE ENTRY MADE)

2) THE NETWORK “B~~E~~REAKS”

3) AL IS UNABLE TO COMPLETE THE UNLOCK WHEN FINISHED

4) JOE WANTS TO MODIFY “X”

IS “X” IN USE ?

node name

LLKOB -N "X" -R

obj node_id

"X" AL

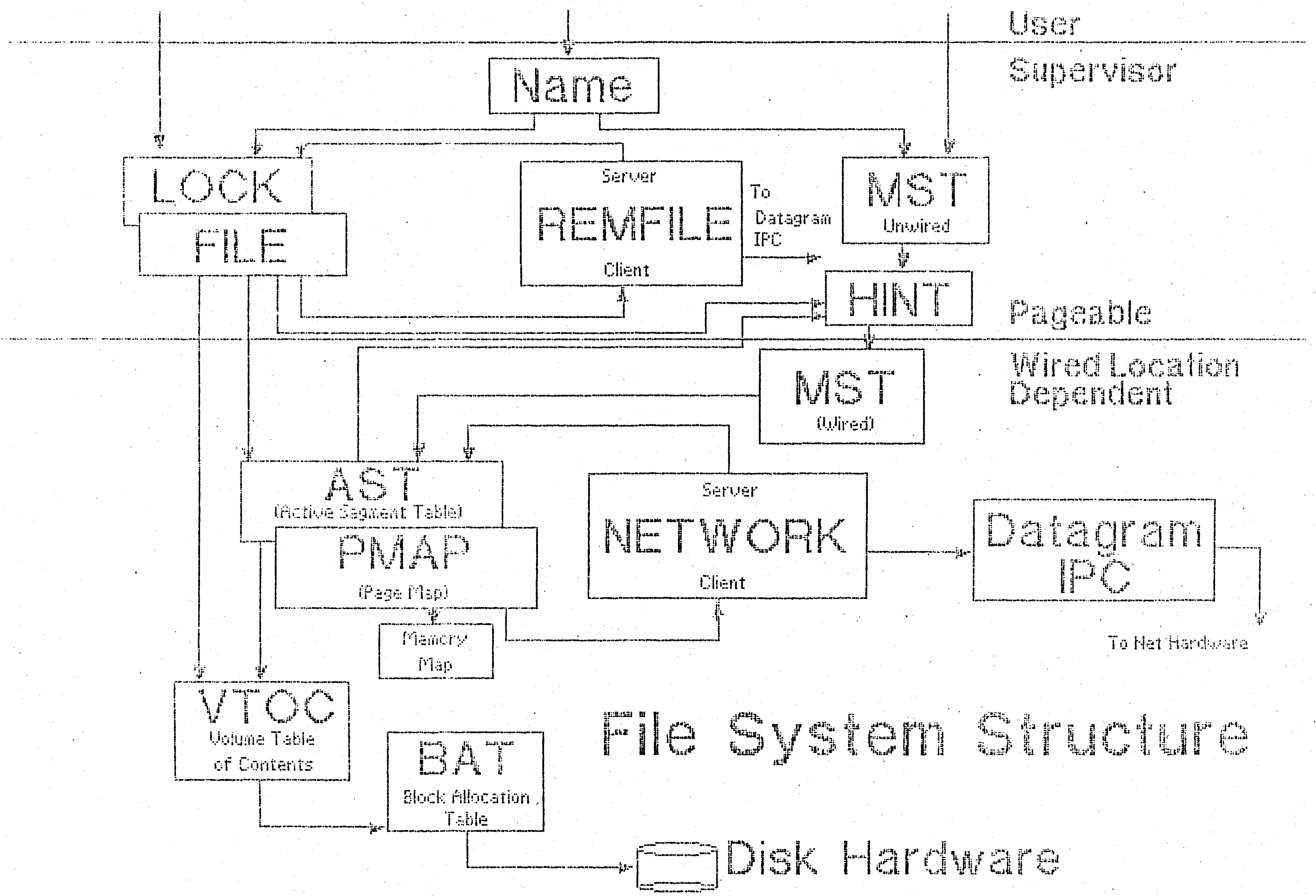
ULKOB -FORCE

LD //AL

AL not found

ULKOB "X" -F

File System Structure



Naming Vocabulary

- Naming Server
    * Set of routines that store and retrieve (NAME, UID) mapping.
- Directories
    * The file storage database used by the naming server.
- “Resolve”
    * The Naming Server operation that takes a name and returns a UID.
- “GPATH” (get–path)
    * The Naming Server operation that takes a UID and returns a name.

NAMING VOCABULARY 2

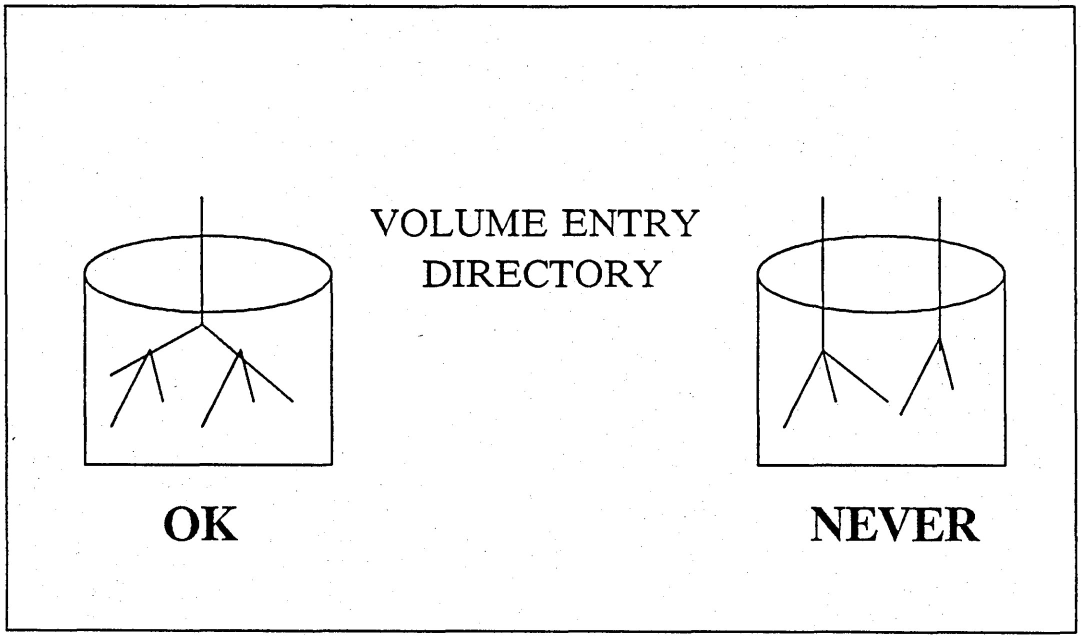
- Soft Links : A Naming Server facility that allows text substitution in names during "name resolve"

LINKS Are not objects → don't have VTOCE? ∴ don't have reference count

- Hard Links : A facility supported by the Naming Server that allows more than one name to be paired with a single UID (needed to support AUX)

- Entry Directory : The directory c~~er~~reated by INVOL to be the root of all named objects on a Logical Volume

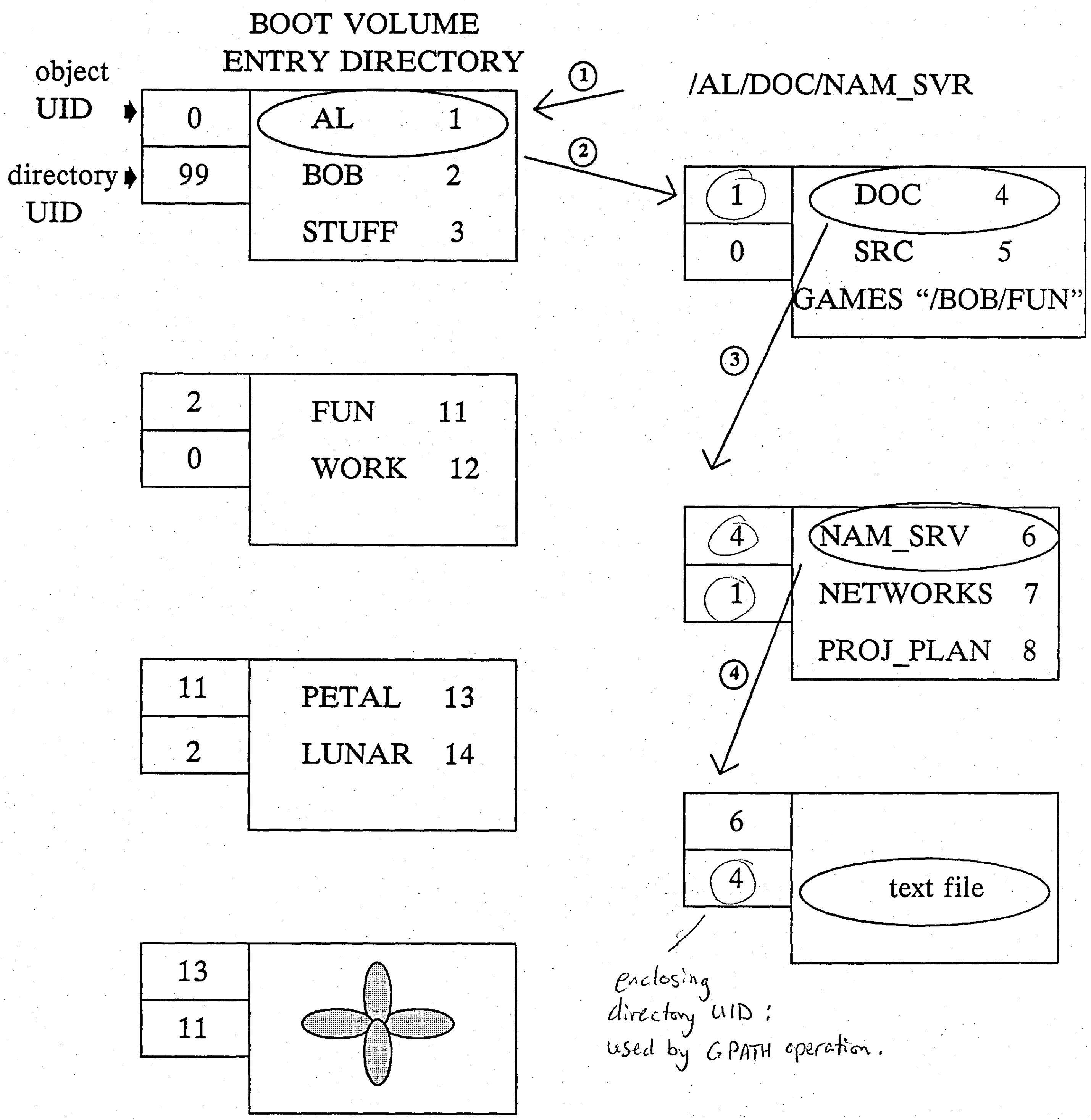
Cataloged in Aegis global space when node comes up



Naming Vocabulary (Cont'd)

- Node entry directory (always /) you can have upto 10 logical volumes mounted, bu only one can be mounted as the boot volume.
  - * The entry directory of the boot volume.
- Network Root
  - * The special directory created by INVOL to hold the node entry directory (NAME, UID) pairs for nodes in the network. "//" ALWAYS refers to the network root directory "hidden" on the BOOT VOLUME.
- Initial ACL's
  - * The Naming Server facility to allow newly created files to inherit their ACL based on the directory that holds their name.

NAME RESOLUTION



BOOT VOLUME
ENTRY DIRECTORY

object UID

directory UID

0	AL	1
99	BOB	2
	STUFF	3

① /AL/DOC/NAM_SVR

②

1	DOC	4
0	SRC	5
	GAMES	"/BOB/FUN"

③

2	FUN	11
0	WORK	12

4	NAM_SRV	6
1	NETWORKS	7
	PROJ_PLAN	8

④

11	PETAL	13
2	LUNAR	14

6	
4	text file

13
11

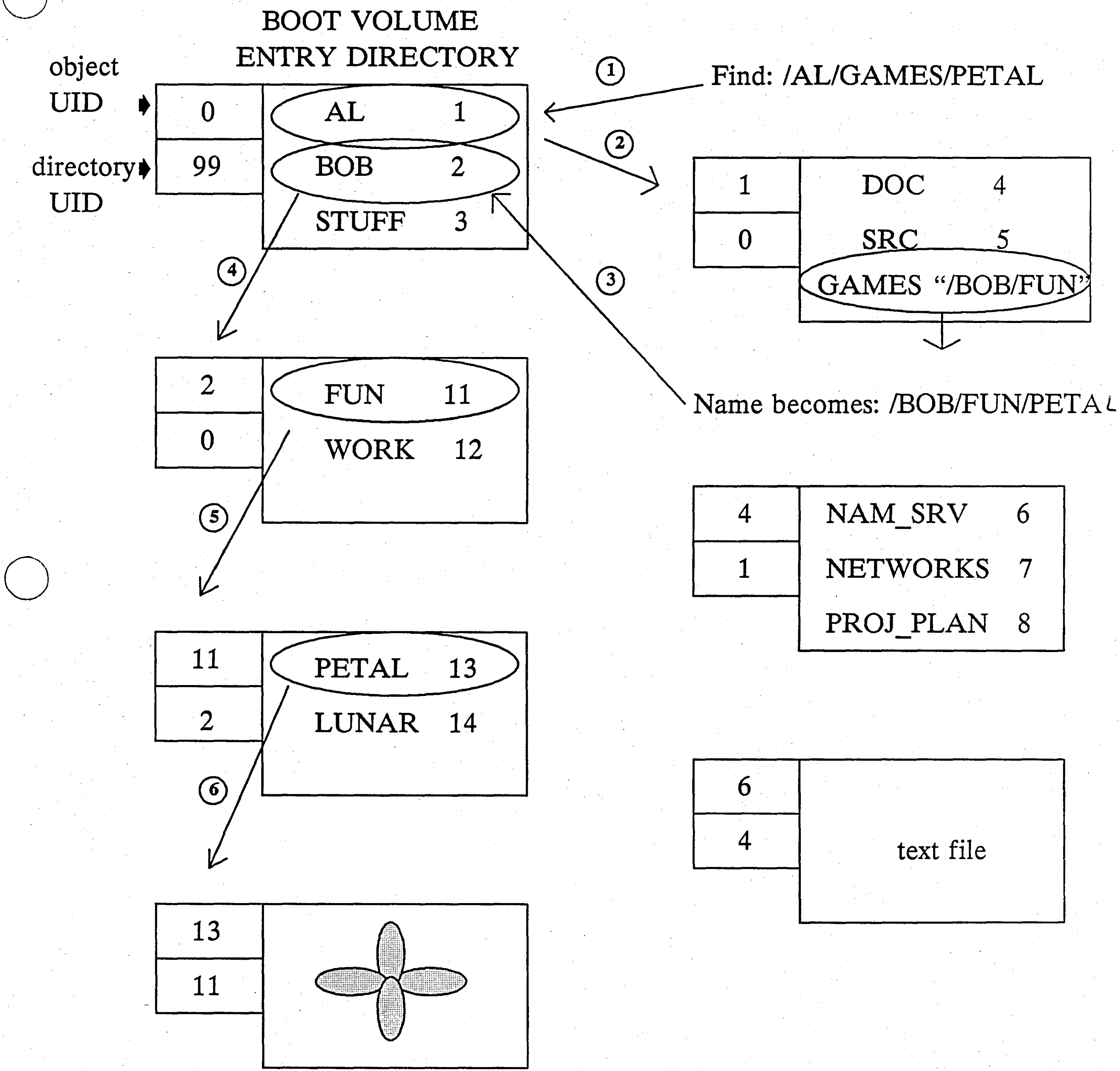
enclosing directory UID: used by GPATH operation.

Ctob associates a name with a UID.

Create or copy a file; object is created on logical volume of enclosing directory.

RULE: All objects live on the same logical volume as their enclosing directory.

NAME RESOLUTION



BOOT VOLUME ENTRY DIRECTORY

object UID

directory UID

0	AL 1
99	BOB 2
	STUFF 3

① Find: /AL/GAMES/PETAL

②

1	DOC 4
0	SRC 5
	GAMES "/BOB/FUN"

③ Name becomes: /BOB/FUN/PETAL

④

2	FUN 11
0	WORK 12

⑤

11	PETAL 13
2	LUNAR 14

⑥

13	
11	

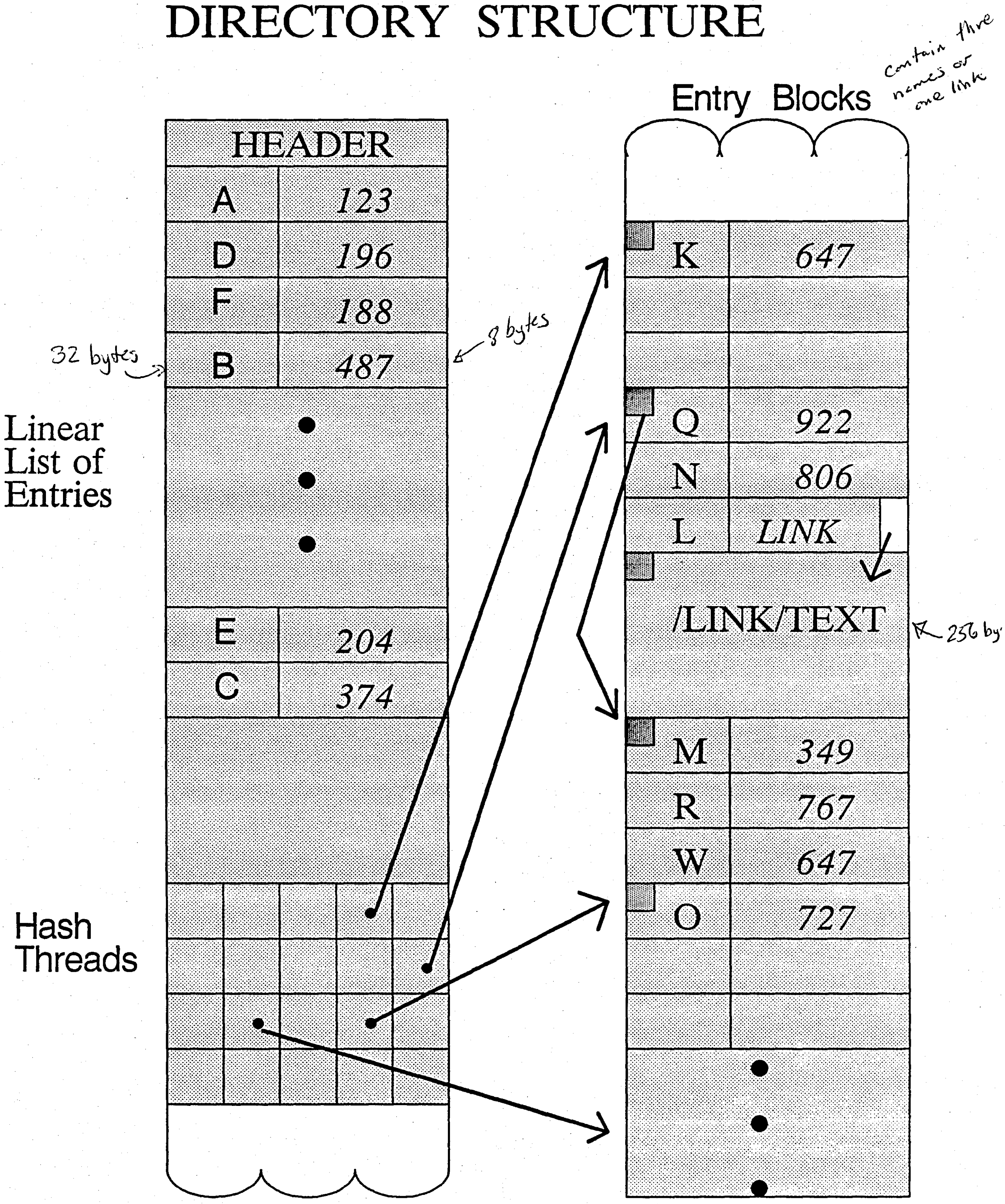
4	NAM_SRV 6
1	NETWORKS 7
	PROJ_PLAN 8

6	
4	text file

the OS paging file is
an unnamed permanent file.

Find-orphans :
to find uncataloged UIDs.
(An orphan object is a UID
without a name)

DIRECTORY STRUCTURE



One disk access for directories with 18 or less files.

limit of directory size is 2 segments → ~1300 names

ADVANCED NAMING TOPICS

Why SALD (salvage–directory)

internal directory structure contains hash threads that can be damaged when the system crashes.

COLOCATION OF NAME and OBJECT

un–necessary for correct operation but necessary for sanity!

HARD LINKS (needed for AUX)

UNIX allows a file to have many names, as long as all of the names live on the same disk volume.

Salvage commands:

Salvol

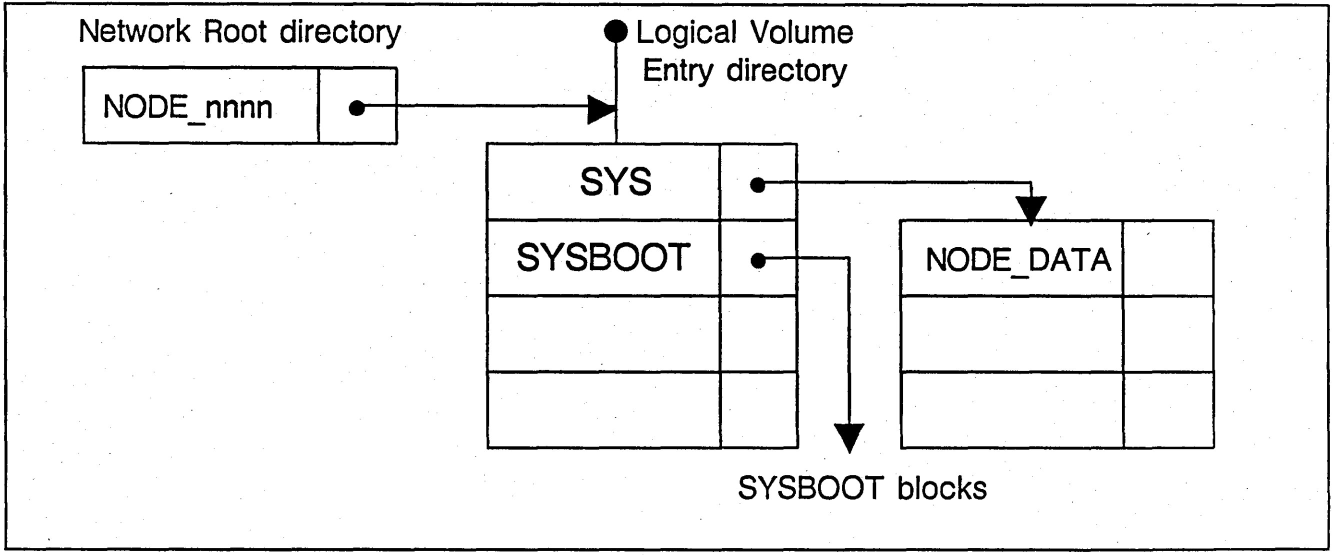
Sald
Salacl (consolidates ACL objects)
Salrgy (updates registries)

MTVOL AND CTNODE

Background:
When a logical volume is created with INVOL, it is given 5 things:

1) A Network Root // + VTOC + BAT
2) An entry directory for the volume /
3) A SYSBOOT file entry
4) /SYS directory
5) 'NODE_DATA directory

Each of these has a UID, let us say UID1, UID2, UID3, UID4 and UID5, respectively. The initial state of the network root is to contain the pair (NODE_nnnn, UID2). The initial state of the entry directory is to contain the pairs (SYSBOOT, UID3), (SYS, UID4) and /SYS contains ('NODE_DATA, UID5).



When a system is running, its network root is accessed through the naming convention of "//". "//" ALWAYS refers to the network root directory on the BOOT LOGICAL VOLUME. The node entry directory is accessed through the naming convention "/". "/" ALWAYS refers to the logical volume entry directory on the BOOT LOGICAL VOLUME.

// is not cataloged anywhere – it's the only directory that's not cataloged.
Each node has its own local // directory.

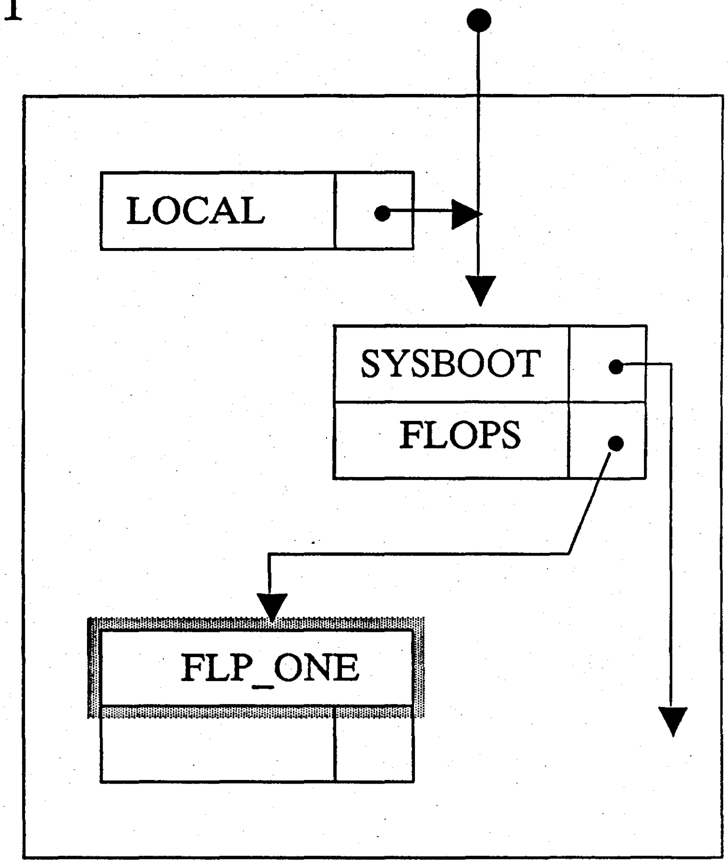
CTNODE uses the asknode manager.

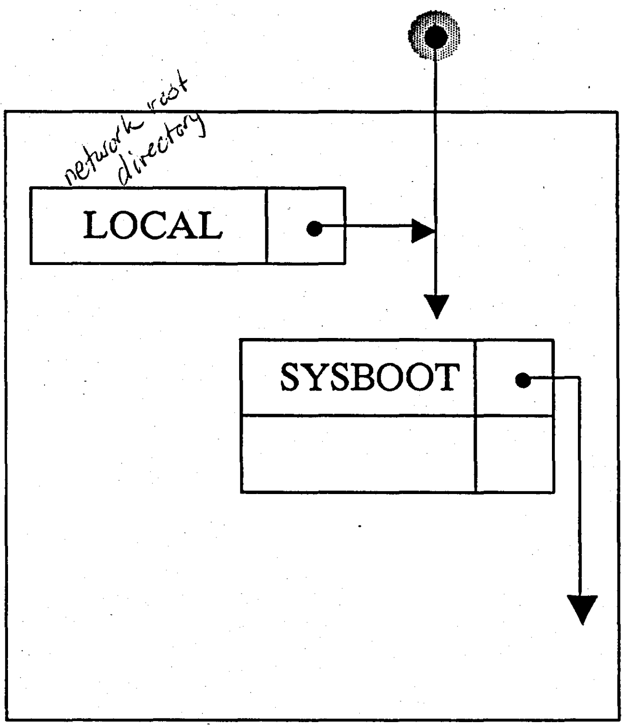
Can't do a file locate on a canned UID
because no node-id info in a canned UID.

wildcarding at // level is different
than anywhere else.

MTVOL

MTVOL F 1 /FLOPS/FLP_1





Winchester Logical Volume One

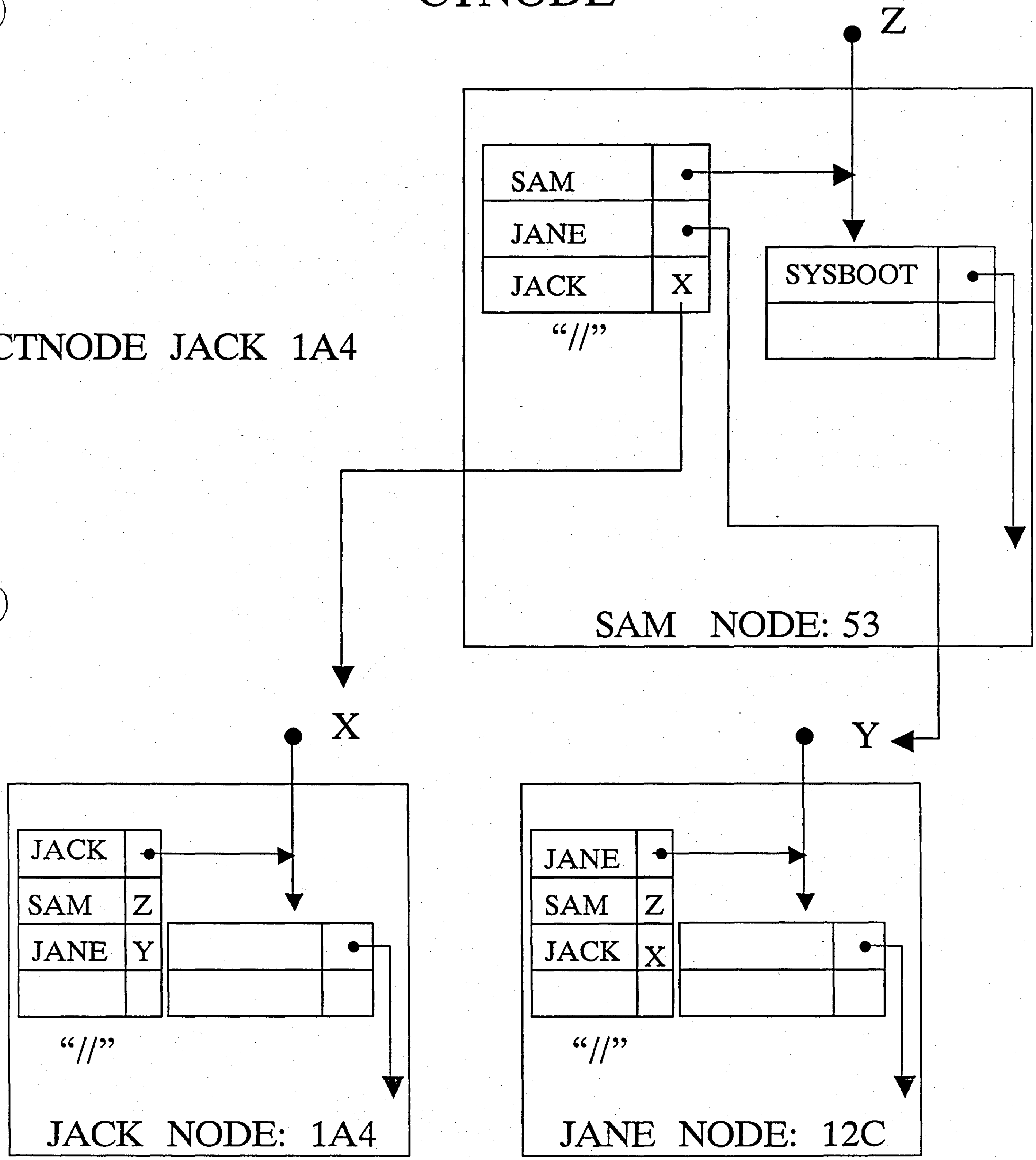
and

Boot Logical Volume

disk controller table entry

CTNODE

CTNODE JACK 1A4



Co–locating Names & Objects

- System architecture does NOT require it.

- SANITY DEMANDS IT!

- So. . .Released utilities ENFORCE IT!

Naming Issues Today (1/85)

1. Set of Legal Characters

2. Case Sensitivity

3. Character “Conflicts”

   (.   ~   ‘   /)

4. Component name length

5. Directory size limit

– AUX/UNIX compatibility issue.

VM Performance Issues

- Disk through–put
  - File layout

    Ordered seeks as of SR9
  - Touch–ahead

    go after a file, you always get four pages minimum

    ↘ 32
- Network through–put
  - Touch–ahead
  - Paging server queuing

    another big non-linear performance degradator
  - Expoliting overlap
- Page replacement
  - Purifier
  - LRU
- ASTE Replacement

  thrashing for large files. improve on the algorithm.
  - LRU

Networking at Apollo

1. The Ring

2. Packets & Sockets

The datagram service is "msg"

3. Clients of Sockets
    - Paging Server
    - File Server
    - NETMAN
    - MBX

MBx is built on top of MSG. MBx is like a virtual circuit model. (protocol)

The Apollo Ring Network

- Ours is a TOKEN–PASSING RING network

  * TOKEN PASSING

  A special bit–pattern circulates through the network ("passing" from node–to–node). In order to transmit a message, a node must have control of this TOKEN.

  * RING

  The nodes are connected in a circle.

flow of information is counter-clockwise

4 types of sync characters

Why a ring like ours?

1. Token–passing for distributed control of communications hardware.

2. Graceful degradation under heavy traffic bursts.

3. Automatic acknowledge of successful transmission.

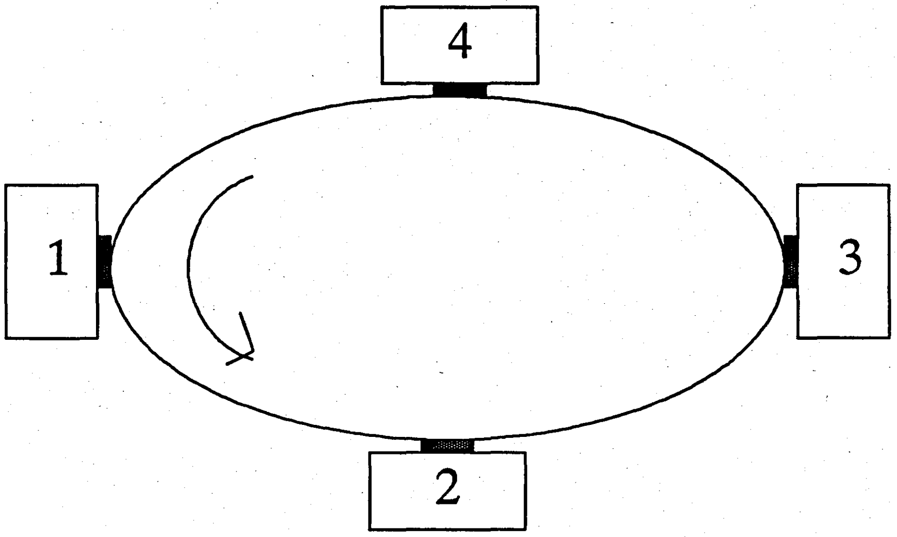
acknowledge is built into the transmission technique.

4. Allows different "WIRING" technologies.

    * e.g. Fiber, microwave

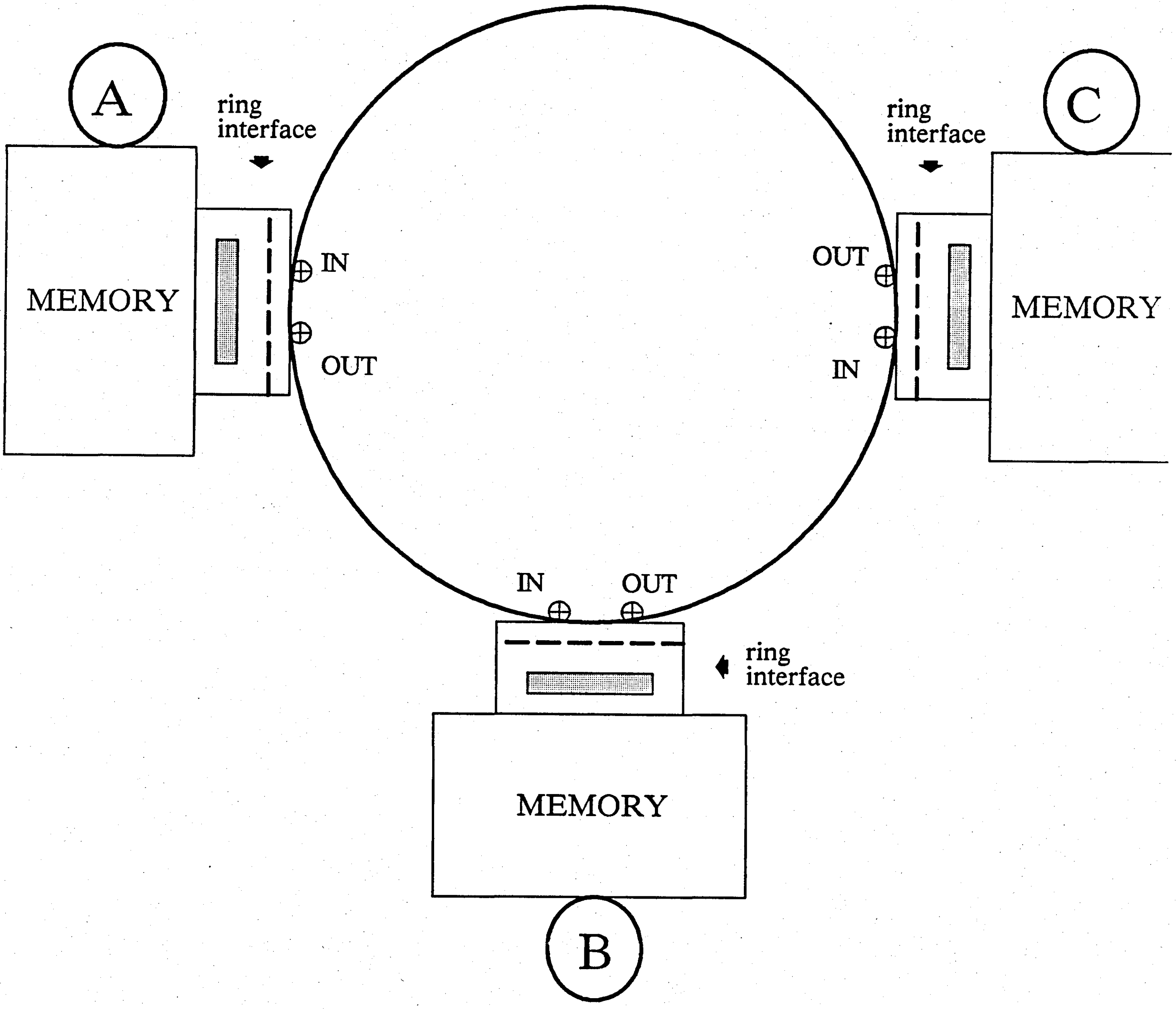
if you're running slow,
it's not the bandwidth of network –
it's the request rate on the paging server.

THE APOLLO RING NETWORK



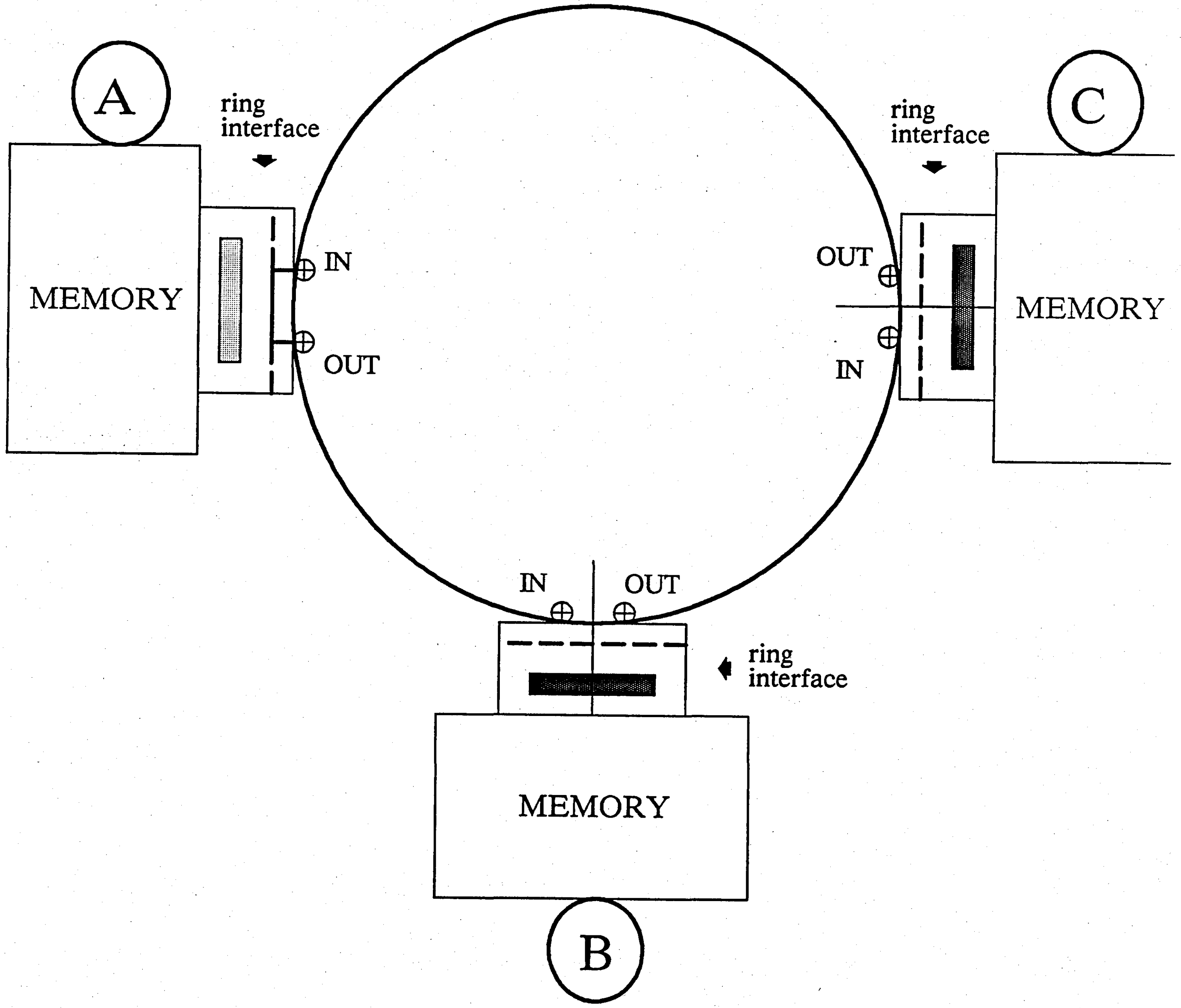
- Every message goes "through" every node (ring hardware)
- Only targeted receiver "processes" the message (DMA into memory, change the ACK byte)
- The transmitter "removes" the message after one full circle
- The transmitter examines the ACK byte to see if the intended receiver got the message (altered the ACK byte)

THE APOLLO RING NETWORK



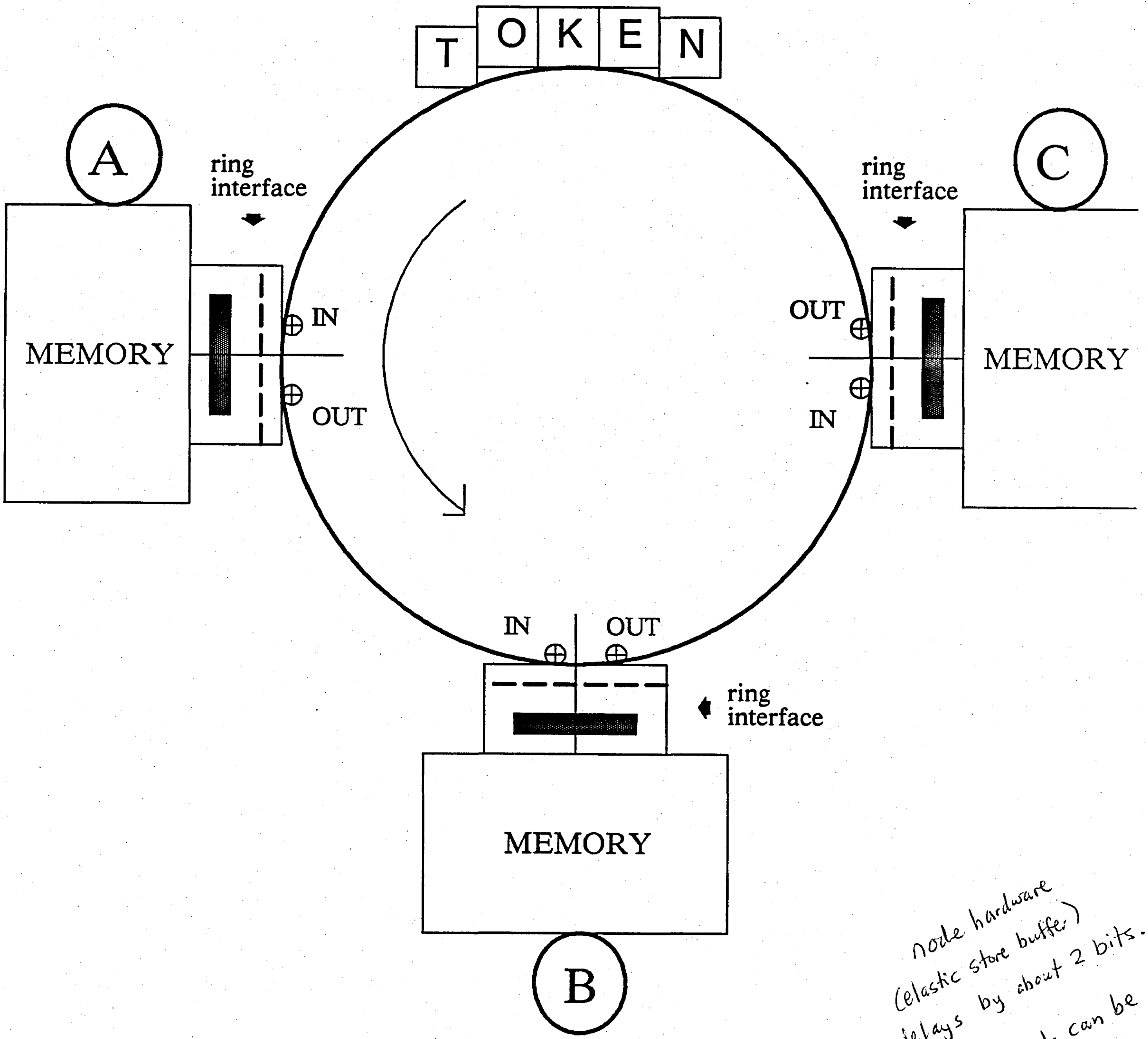
THE APOLLO RING NETWORK

"A" Disconnected



THE APOLLO RING NETWORK

IDLE - no node wants to TRANSMIT



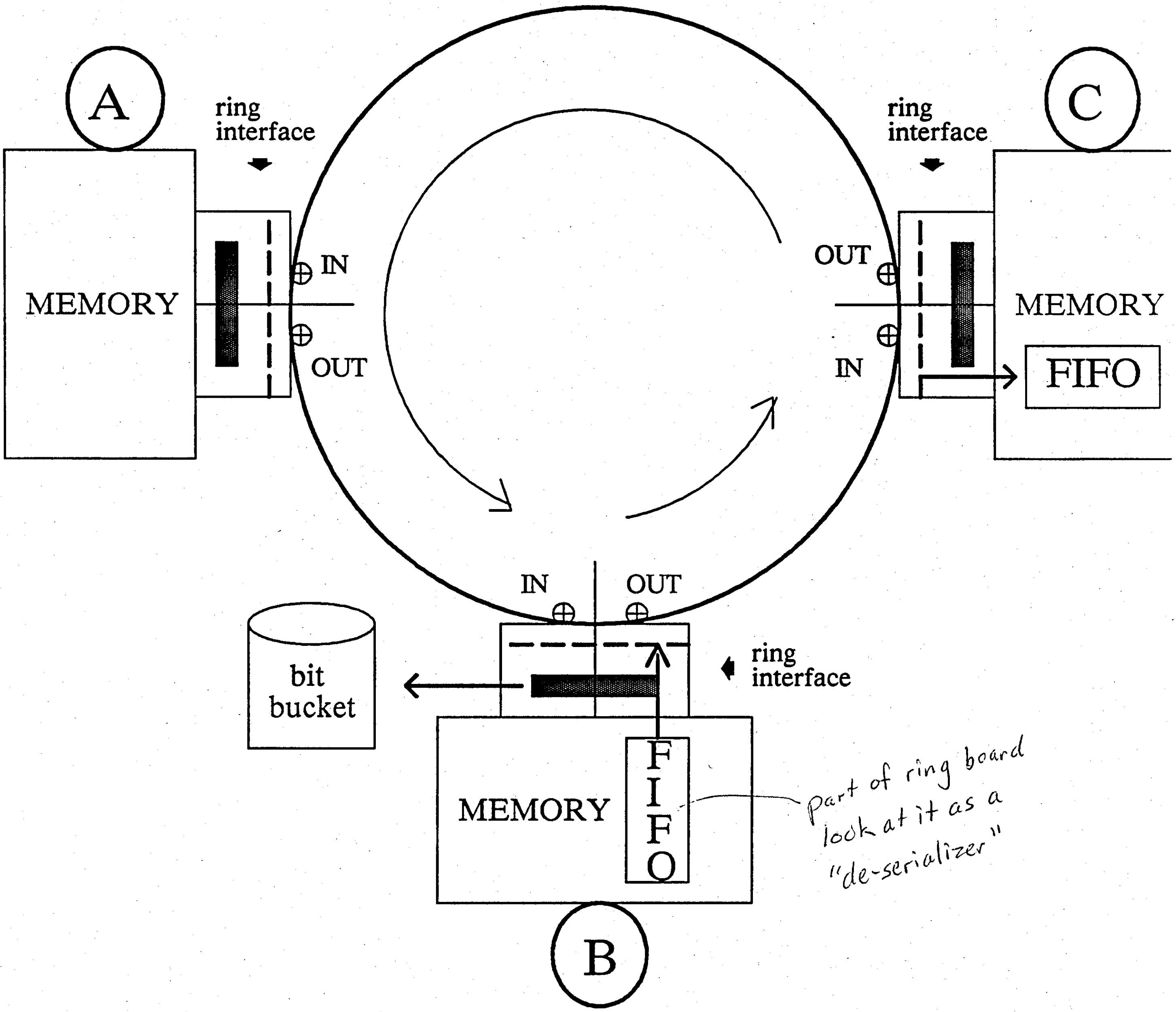
node hardware (elastic store buffer) delays by about 2 bits.

delay of 6 bits can be switched in if it doesn't see a recognizable token

netstat tells you if the delay has been switched in.

THE APOLLO RING NETWORK

"B" sends to "C" and watches for the ACK fields



part of ring board look at it as a "de-serializer"

hardware never retries, the software retries.

WACK (wait acknowledge)

PACKETS

&

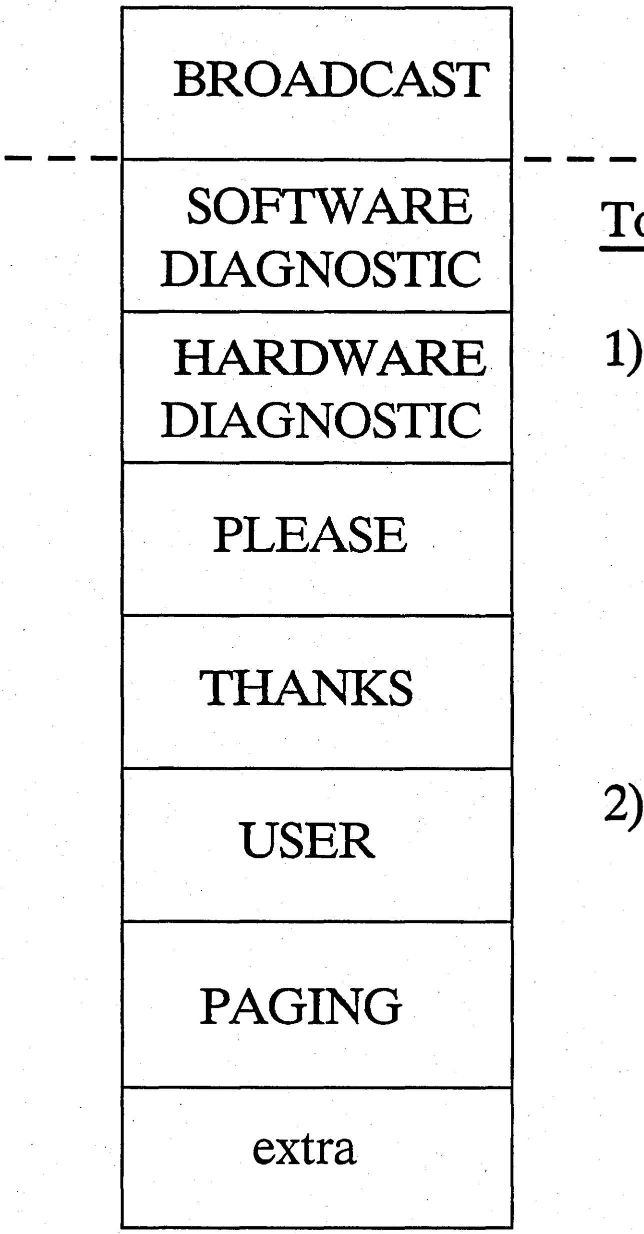
SOCKETS

LD -U gives you the UID of files in directory

TOKEN → message header (last bit toggled)

PACKET HEADER	TO NODE	
	TYPE	HDWR HDR
	EACK	
	FROM NODE	
	TO SOCK	AEGIS NETWORK
	FROM SOCK	PROTOCOL
	TRANS ID	SOFTWARE HDR
	USER HDR DATA	CLIENT HDR (e.g.PAGING SERVER)
PACKET DATA	MSG-SEP	HDWR SUPPLIED
	USER DATA	CLIENT DATA (OPTIONAL)
	CRC	HDWR SUPPLIED
	ACK FIELD	HDWR SUPPLIED

THE TYPE FIELD



To receive a packet :

1) The "To Node" must match or BROADCAST must be set

AND

2) The "To Node" must be willing to accept packets of this TYPE

anded with type mask.
type mask is set by netsvc:

ring + disk use same bus:
If there is a long seek on disk, (DMA is progress)
WAK will be sent: (I want to receive it, but I'm not ready)

Apollo Network Sockets

- Queues of received packets

Socket depth is 7 packet Packets are 2k (1½ k). 14k per socket that has to be wired.

- Identified by "simple" numbers (e.g. "1", "4")

Performance implications when there are a lot of paging requests. The packets overflow the queue,

- Numbers unique within a node, but not unique across nodes
- Two "kinds"— Well–known and Reply

  * Well–known

  Used by System Services (e.g. Paging Server uses Socket #1 in every Apollo node)

  * Reply

  Used by clients of well–known sockets

  Allocated as needed from a pool

  Provide a "return address" to be sent with service requests.

LCNODE is built on top of ASKNODE Manager.

Sockets are dynamically wired as they are needed. (2N-1) where N = number of processes

phase-lock loop (for synch)

bi-phase modulation. (clock is 2x signal ~~speed~~ freq.

Clock is sent by every node, packet is retransmitted by every node.

Elastic store buffer (1-2 bits) allows node to take up slight variations in clock, signal.

Clients of “Socket”

1. Paging Server

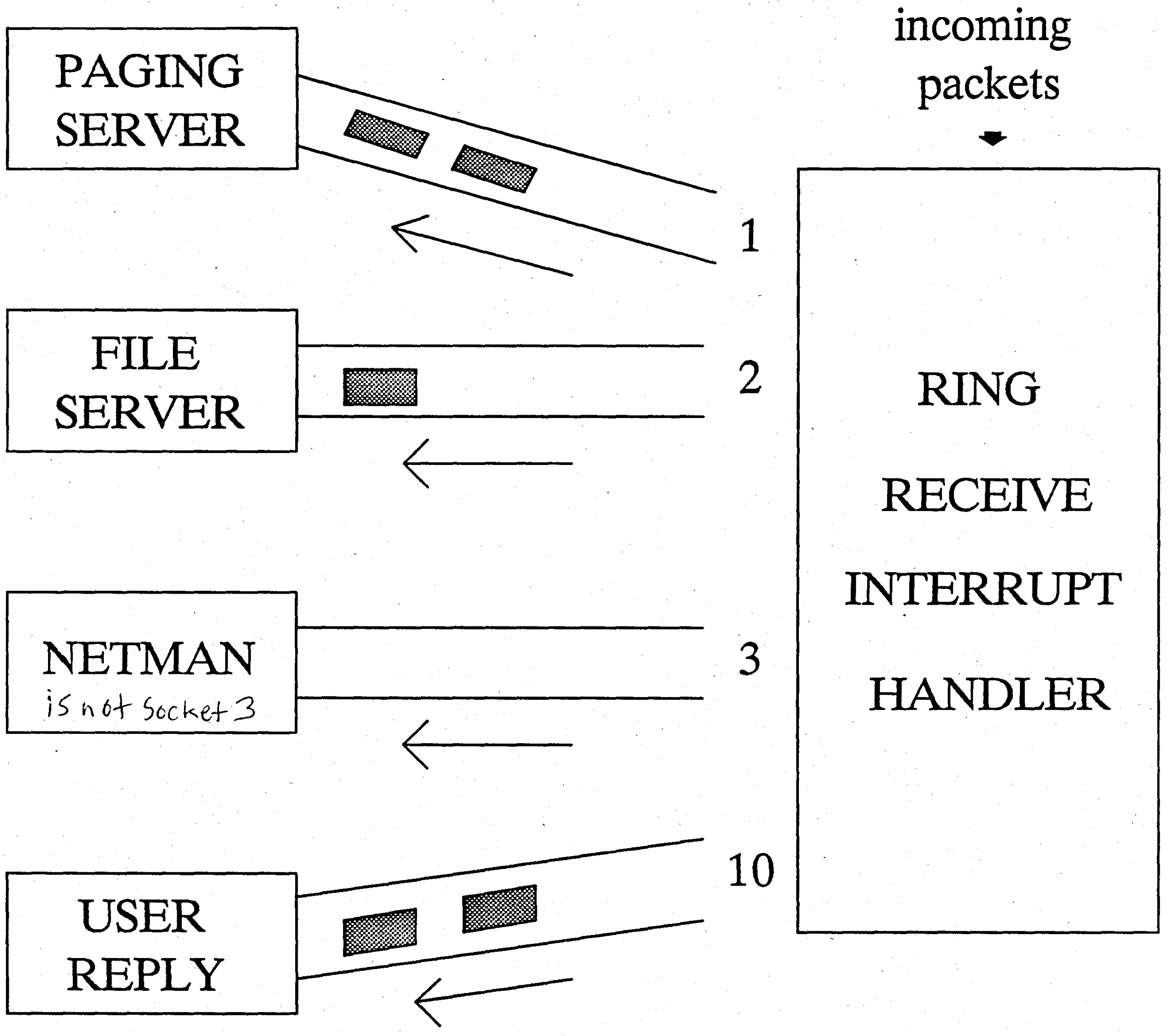
2. File Server/Information Server

3. Netman

4. MBX

- Each of these servers is assigned a well–known socket number. To obtain service, a client must address a packet containing the REQUEST to a (NODE, SOCKET) pair. (Paging server on node 1BA can receive paging requests on Socket #1 at node 1BA.

SOCKETS



To decline incoming packets, the Interrupt Handler examines the Packet Software Header for the Target Socket Number

To save copying :

DMA into wired buffer,
buffer is threaded into user address space
mapped and unwired.

Socket Service

1. DATAGRAM

2. Unreliable

   - Can lose/discard packets
   - Can arrive out of sequence
   - Can deliver duplicates

3. The ONLY Apollo packet delivery mechanism.

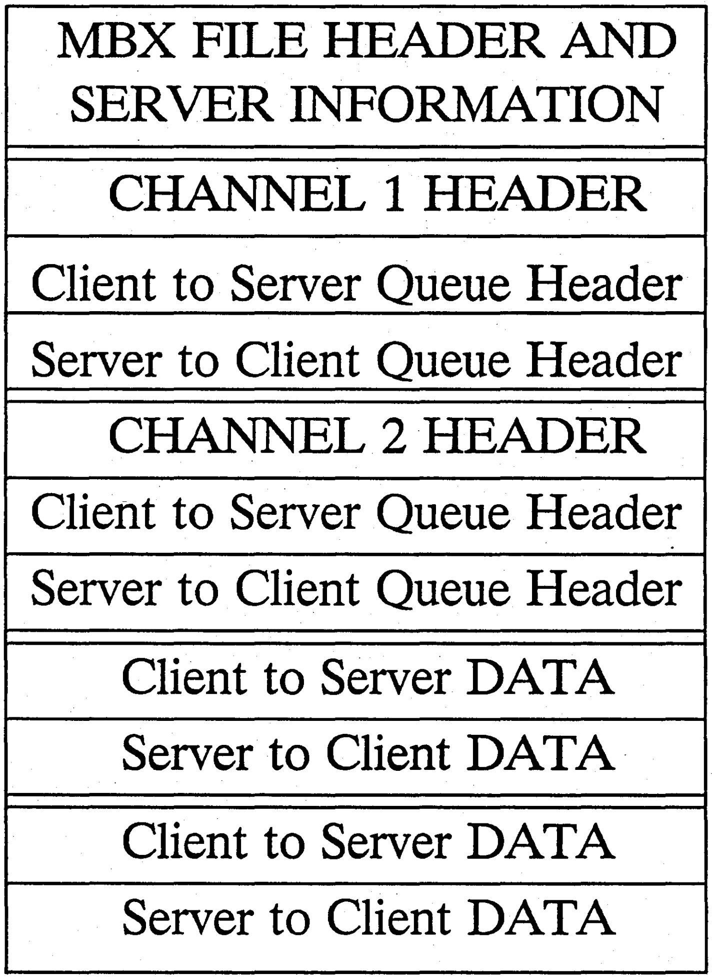
4. Available to user space through the (unreleased/undocumented) "MSG" interface.

User Available IPC

MBX

- Interprocess
- Intra– and Inter– node
- User callable
- Fully documented
- Full–duplex virtual circuits
    * Flow control
    * Guaranteed delivery
- Identified by pathnames

A MAILBOX



MBX FILE HEADER AND SERVER INFORMATION

CHANNEL 1 HEADER

Client to Server Queue Header

Server to Client Queue Header

CHANNEL 2 HEADER

Client to Server Queue Header

Server to Client Queue Header

Client to Server DATA

Server to Client DATA

Client to Server DATA

Server to Client DATA

- "Owned" by the SERVER
- SERVER specifies the number ofchannels and the size of the DATA area
- Shared memory (co–writers)

You don't have to be on same node as your mailbox.

whole cloth objects have no backing stored: they can never be paged, (permanently wired).

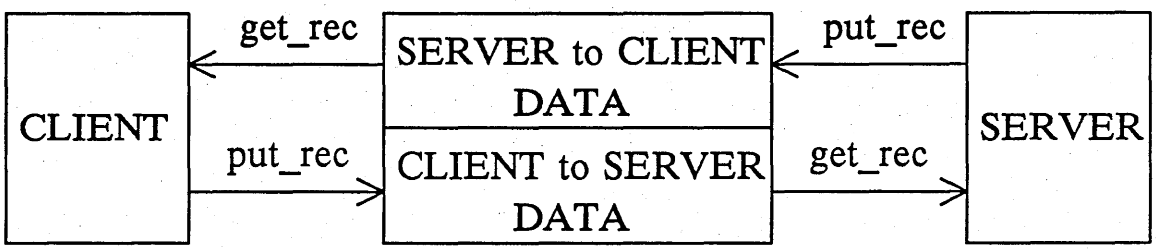
MBX

SERVER AND CLIENT CO-RESIDENT

MBX file

Shared memory mapped for cowriters

Cowriters must be on same node.



CLIENT

get_rec

put_rec

SERVER to CLIENT DATA

CLIENT to SERVER DATA

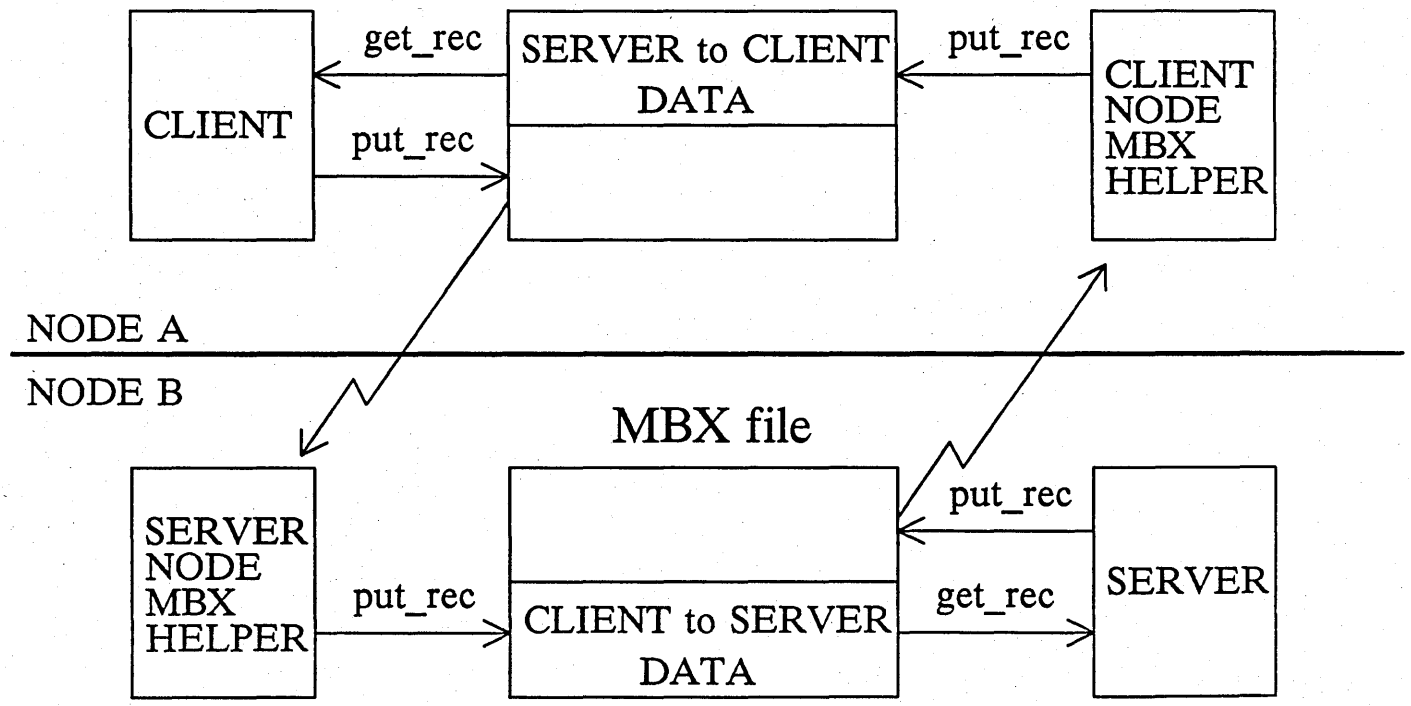
put_rec

get_rec

SERVER

SERVER AND CLIENT ON DIFFERENT NODES

SYSMBX file



CLIENT

get_rec

put_rec

SERVER to CLIENT DATA

put_rec

CLIENT NODE MBX HELPER

NODE A

NODE B

MBX file

SERVER NODE MBX HELPER

put_rec

CLIENT to SERVER DATA

put_rec

get_rec

SERVER

MAILBOX SERVER INFORMATION

SERVER HANDLE and FLAGS
SERVER OPEN TIME
MBX LOCK
ANY CHANNEL EVENTCOUNT
ANY ROOM EVENTCOUNT
QUEUE SIZE
NUMBER OF CHANNELS
SET OF OPEN CHANNELS
SET OF CHANNELS WITH DATA
SWEEP INDEX

is a mutex lock : if you're trying to write and this lock is set, you have to wait.

A QUEUE DESCRIPTOR

USAGE AND FLAGS

BYTES IN EVENTCOUNT

BYTES OUT EVENTCOUNT

REMOTE BYTES NEEDED

QUEUE START OFFSET

QUEUE END OFFSET

QUEUE IN OFFSET

QUEUE OUT OFFSET

QUEUE OUT REMAINING

IN FRAGMENTED PUT

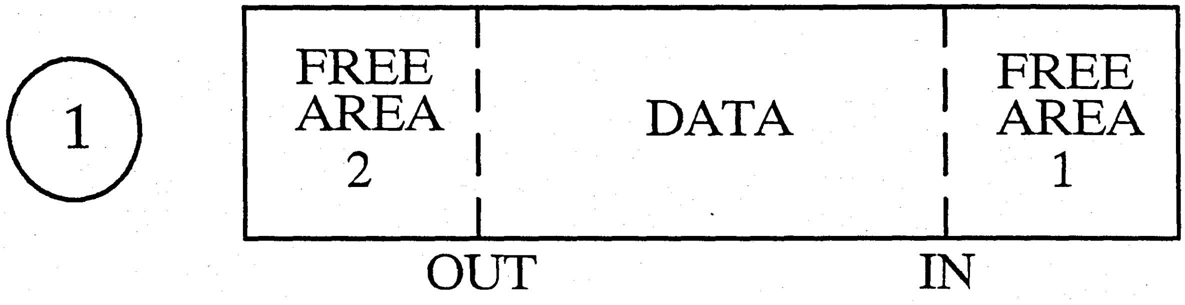
FRAGMENTED START

FRAGMENTED LENGTH

}
UNUSED
LOCAL
REMOTE
EOF_PENDING

set by cntrl/z
or end of channel

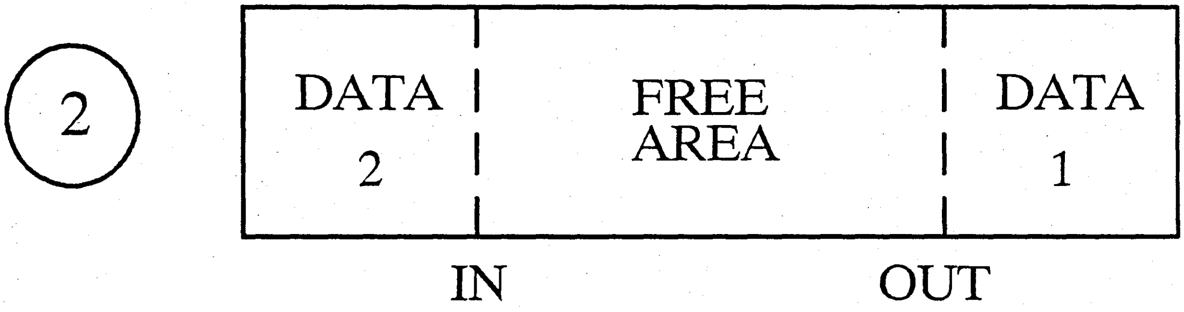
CIRCULAR QUEUES



1

FREE AREA 2 | DATA | FREE AREA 1

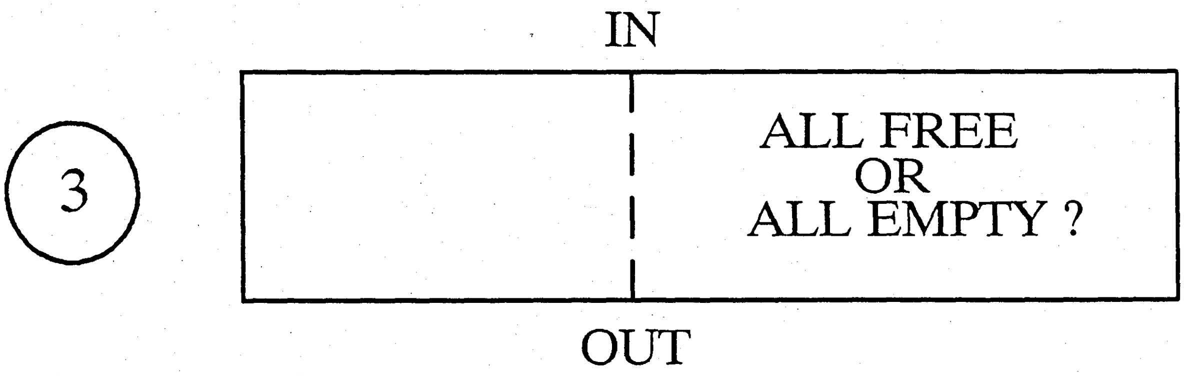
OUT IN



2

DATA 2 | FREE AREA | DATA 1

IN OUT



3

IN

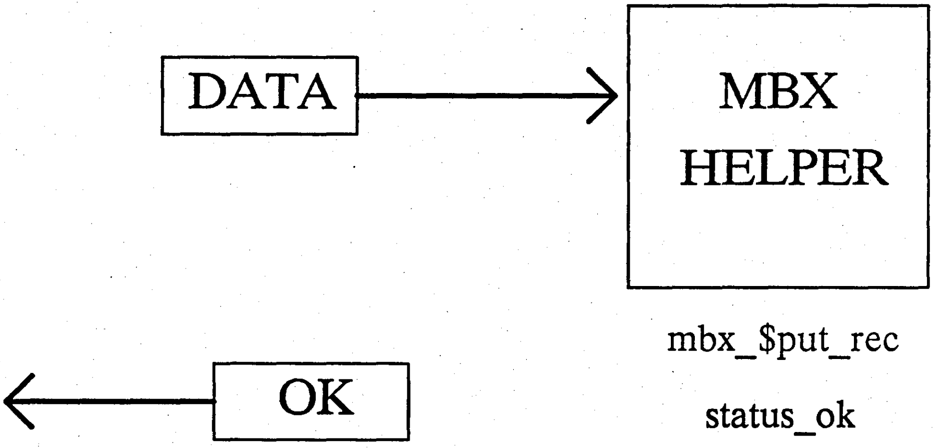
ALL FREE OR ALL EMPTY ?

OUT

FREE IFF BYTES IN = BYTES OUT

MESSAGES

NORMAL CASE



DATA

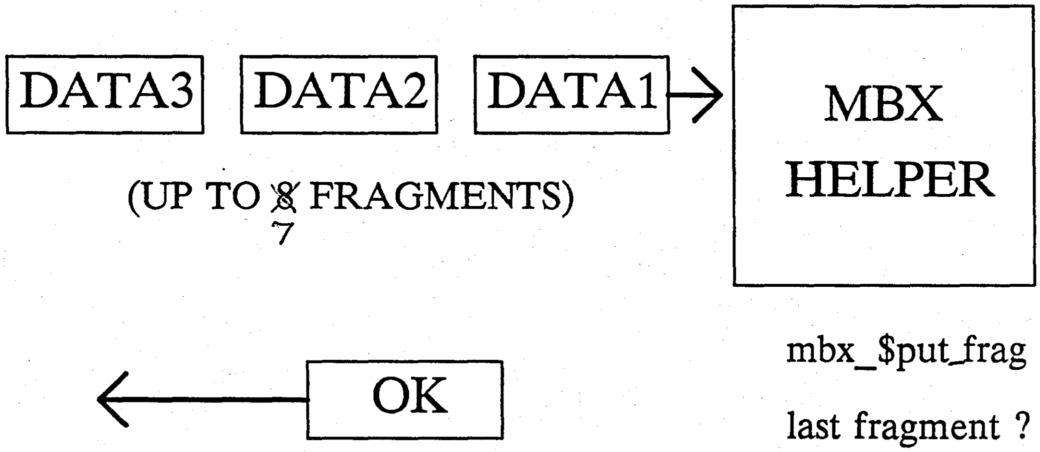
MBX HELPER

OK

mbx_$put_rec

status_ok

FRAGMENTED CASE



DATA3 DATA2 DATA1

MBX HELPER

(UP TO ~~8~~ 7 FRAGMENTS)

OK

mbx_$put_frag

last fragment ?

status_ok

AEGIS Process Management

- Topics:
  - * Process Switching (dispatching) context switching
  - * Interrupt Handling
  - * Processor Scheduling
  - * Synchronization (eventcounts) EC
  - * Mutual Exclusion ML
  - * Special CPU B Handling
  - * Process Creation & Deletion all PCB's are wired from OS init
  - * Asynchronous Fault Delivery
  - * Clocks & Time–Driven Events

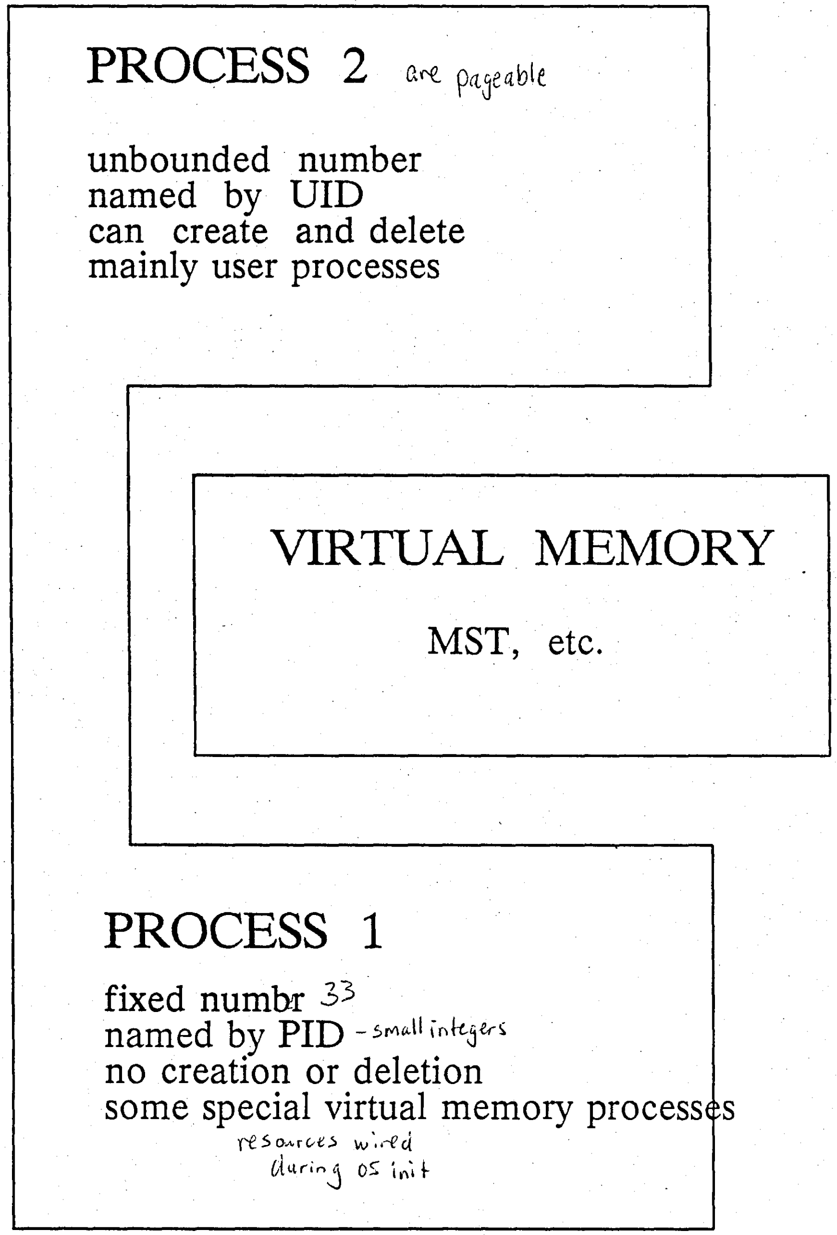
AEGIS Process Management (Cont'd)

– Managers:

Process management managers

- * Level One Processes (PROC 1)
- * Level Two Processes (PROC 2)
- * Level One Eventcounts (EC)
- * Level Two Eventcounts (EC2)
- * Mutex Locks (ML)
- * Timers (Time)

WHY TWO LEVELS ?



PROCESS 2 are pageable

unbounded number
named by UID
can create and delete
mainly user processes

VIRTUAL MEMORY

MST, etc.

PROCESS 1

fixed numbr 33
named by PID - small integers
no creation or deletion
some special virtual memory processes
resources wired
during OS init

Proc 2 + Proc1 managers
reside in nucleus.
PM manager is user
Space code.

What is a Level One Process?

- Processor State
  * Stack Pointers (SSP, USP) supervisor stack pointer A7' user stack pointer
  * Address Space ID (ASID) for binding with Proc2's
  * Virtual Time Clock running total of the time the process has used
  * "Resource Lock" Set

- Scheduling Information
  * Scheduling Priority "resource lock" set is part of this
  * Resource Lock Set
  * Remaining Time Slice
  * Time Since Last Wait
  * State:

    bound means it can be scheduled
    waiting on an ec somewhere
    suspended unschedulable DS: suspend DC; continue
    suspend pending try to suspend a process with a resource lock.
    TSE with resource lock
    (time slice end

Resource Locks

- Not really locks at PROC1 level
- Control deadlock detection
- Control scheduling priority
  - * A process with a resource lock has proirity over a process with none
  - * A process with an "important" resource lock has proirity over a less important one

Resource Locks (Cont'd)

- Control ability to turn on CPU B
    * A process with an lock higher than OK_ON_B can run on CPU B

      Signal to dispatcher: "I'm not going to take a page fault"
    * A process witn no locks or whose highest lock is less than OK_ON_B cannot run on B
- Prevent process suspension
- User–mode code never holds a resource lock

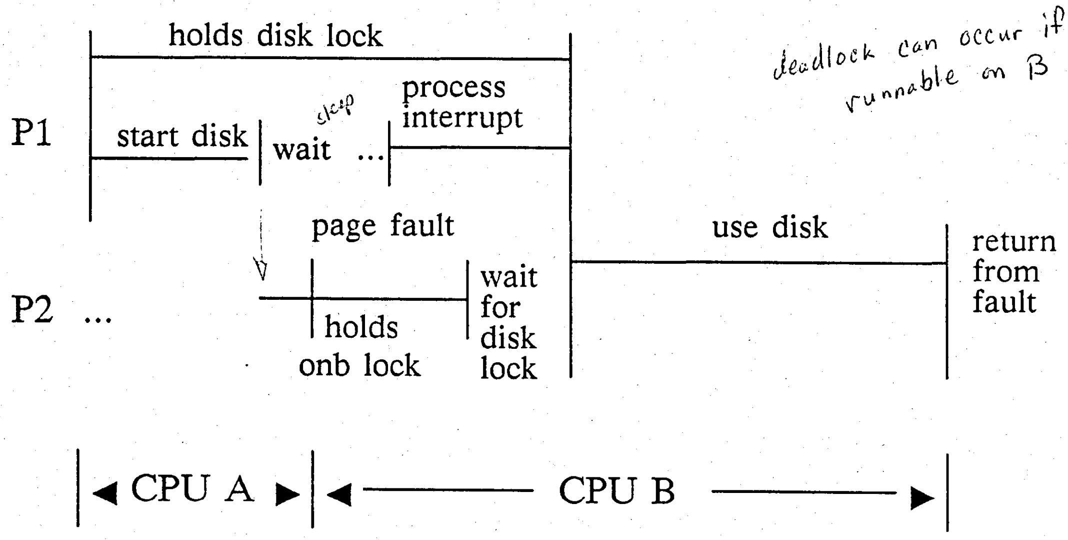
Example : A Disk Driver

- needs exclusive access to the device

- must be runnable on CPU B

- wants high priority

- a time line :



holds disk lock

P1

start disk

sleep

wait ...

process interrupt

page fault

use disk

return from fault

P2 ...

holds onb lock

wait for disk lock

◄ CPU A ►

◄ CPU B ►

deadlock can occur if not runnable on B

Resource Locks

If you have a resource lock that makes you non-suspendable by scheduler, you have guaranteed atomic operation.

network_$server_lock	{ 00 1 }
mt_$lock	{ 01 2 }
ml_$free3	{ 02 4 }
ml_$free4	{ 03 8 }
ml_$free5	{ 04 10 }
file_$lock_lock	{ 05 20 }
ec2_$lock	{ 06 40 }
smd_$respond_lock	{ 07 80 }
smd_$request_lock	{ 08 100 }
disk_$mnt_lock	{ 09 200 }
term_$lock	{ 10 400 }
proc1_$create_lock	{ 11 800 }
onb_$lock	{ 12 1000 faulted to CPU B }
bok_$lock	{ 13 2000 runnable on B }
vtuid_$lock	{ 14 4000 }
vtoc_$lock	{ 15 8000 }
bat_$lock	{ 16 10000 }
ast_$lock	{ 17 20000 }
pag_$lock	{ 18 40000 }
ml_$free6	{ 19 80000 }
flp_$lock	{ 20 100000 }
win_$lock	{ 21 200000 }
ring_$xmit_lock	{ 22 400000 }
ml_$free7	{ 23 800000 }
	{ the next two locks are the highest }
time_$proc_lock	{ 24 1000000 clock process only }
time_$lock	{ 25 2000000 clock process database }

- boundary for running in wired code.

to increase priority, acquire locks

to have a lock means it's in your PCB.

if you try to acquire a lower lock than you already have, the system crashes (deadlock detection + solution)

The PROC1 Database

- The Process Control Block (PCB)
    * Stores processor state & scheduling information
    * One per level one process
- The PCB Array
    * Array [pid_t] of pcb_t
    * pid_t = 1...32
- The Currently Running Process
    * PROC1_$CURRENT
- The Ready List
    * A linked list of PCBs
    * Ordered by CPU scheduling priority
- All PROC1 data is wired

PROC1 Operations

- Scheduling
  - * PROC1_$CHG_PRI (pid, priority_increment)

    increment/decrement CPU priority

    assigns new time slice

    returns old priority
  - * PROC1_$SET_TS (pid, new_time_slice),

    used only internally and by clock process

PROC1 Operations (Cont'd)

- Resource Locks

  * PROC1_$SET_LOCK
    (lock_no)

    crash system if higher lock already held

  * PROC1_$CLR_LOCK
    (lock_no)

    crash if not held or not highest lock held

  *PROC1_$SPECIAL_CLR_LOCK

    used for CPU B–A transition

More PROC1 Operations

- SUSPEND/RESUME
  - * PROC1_$SUSPEND (pid)

    returns boolean –> success

    set SUSPEND_PENDING otherwise:
  - * PROC1_$SUSPEND_EC
    advanced when actually suspended
  - * PROC1_$SUSPENDP (pid)

    returns boolean –> process now suspended
  - * PROC1_$RESUME (pid)

More PROC1 Operations

- Inquiry

    * PROC1_$GET_CPUT
      (virtual_time)

    * PROC1_$GET_INFO
      (pid, info_record)

pass through calls from proc2

- Bind/Unbind

    * PROC1_$BIND
      (start_pc, stack_ptr, stack_base)

        allocate PCB
        build call frame on stack
        make ready
        returns new pid

    * PROC1_$UNBIND (pid)

        suspend process
        make PCB available
        (unbound)

- Allocate Supervisor Stack

  * PROC1_$ALLOC_STACK
    (size_needed)

    determined by
    trial + error.

    returns STACK_PTR

    wires pages of new stack

  * PROC1_$FREE_STACK
    (stack_ptr),

  * PROC1_$CREATE (start,
    stack_size)

    combination of
    alloc_stack + bind

    not really create—just a
    combination of
    ALLOC_STACK and BIND

    used only for special nucleus
    processes

Implementing PROC1 Calls

- Rule: Ready = Current return
  - * Except when interrupts are disabled inside PROC1
- Procedure
  1. Check validity of call
  2. Disable interrupts
  3. Modify PCB
  4. Reorder ready list
  5. Dispatch

Dispatching

- Procedure

    * IF ready < > current THEN

        save CPU state of current
        establish CPU state of ready

    * Enable interrupts

    * Return

- Only hard part is maintaining time slice/virtual clock

    * Special timer clip holds remaining time slice

- Null process
    * pid = 2
    * Always ready
    * Always lowest priority (∅) means it can't have a resource lock
    * Just loops looks at ready list, if it's out of order → crash the system (priorities are not in linear order)
- What if highest priority process not readable on CPU B?
    * Determined by resource locks
    * Just run null process

Cntrl/return → puts you in mnemonic debugger.

you're running in the null process, so you can't do anything that involves page faults, or the system will go away.

Interrupt Handling

- Interrupts vector directly to driver—no special interrupt queueing or dispatching mechanism

interrupt occurs, traps through page 9

- Most interrupt handlers are very simple—just advance an eventcount and return—actual interrupt processing done by driver in requesting process

a lot of the code at the proc1 level has been rewritten in Assembler.

- PROC1_$INT_ADVANCE
  - Jump to here to advance an eventcount and return from an interrupt
  - Push all registers on stack, plus eventcount address
  - Must be done in assembly language
  - INT_ADVANCE simply calls a special version of

normally, just save a couple of registers and keep track → little PCB's

Only use one interrupt level (6)

EC_$ADVANCE that doesn't dispatch or enable interrups, then calls dispatch if this interrupt is returning to level 0

- PROC1_$INT_EXIT
  - * Use to simply return from interrupt
  - * Jump here with all registers intact
  - * Calls dispatch if necessary, then RTE

SCHEDULING ALGORITHM

- READY LIST IS ORDERED BY THE FOLLOWING 48 BIT QUANTIY (VIEWED AS A SINGLE INTEGER)

1 - 16

32 BIT RESOURCE LOCK SET	16 BIT PRIORITY

- PRIORITY VARIES FROM 1 TO 16 WITH 16 BEING THE HIGHEST

- NULL PROCESS HAS PRIORITY ZERO

- THE PRIORITY OF A NEW PROCESS IS 16

- PRIORITY IS DECREMENTED BY ONE AT EACH TIME SLICE END

- PRIORITY IS INCREMENTED BY ONE FOR EACH 1/4 SECOND OF WAIT TIME WHEN A PROCESS FINISHES EC_$WAIT gives interactivity an edge.

- A PROCESS IS ADDED TO THE READY LIST AT THE END OF ITS PRIORITY CLASS. THIS IMPLEMENTS ROUND-ROBIN SCHEDULING FOR PRIORITY ONE.

- IF A TIME SLICE END OCCURS WHILE A PROCESS HOLDS A RESOURCE LOCK, IT IS MOVED TO THE END OF ITS PRIORITY CLASS WHEN THE LAST RESOURCE LOCK IS CLEARED (TSE_ONB IN THE PCB) SCHEDULING STATE)

- THE TIME SLICE VALUES ARE LARGER FOR LOW PRIORITY PROCESSES AND SMALLER FOR HIGH PRIORITY PROCESSES. PRIORITY 16 GETS 1/10 SEC. , PRIORITY 1 GETS 1/2 SEC. (MAX. IN 16 BITS)

- THE DISPLAY MANAGER ALWAYS HAS PRIORITY 16

Level One Eventcounts

- Operations
  - * EC_$WAIT (ec1, ec2, ec3, value1, value 2, value 3,)
  - * EC_$WAITN (ec_ptr_list, value_list, count)

    these both return ordinal of first EC in list which is satisfied
  - * EC_$ADVANCE (ec)
  - * EC_$READ (ec)

    returns current value

    normally done by inline code for speed
  - * EC_$INIT (ec)

    initializes an eventcount

Level One Eventcounts (Implementation)

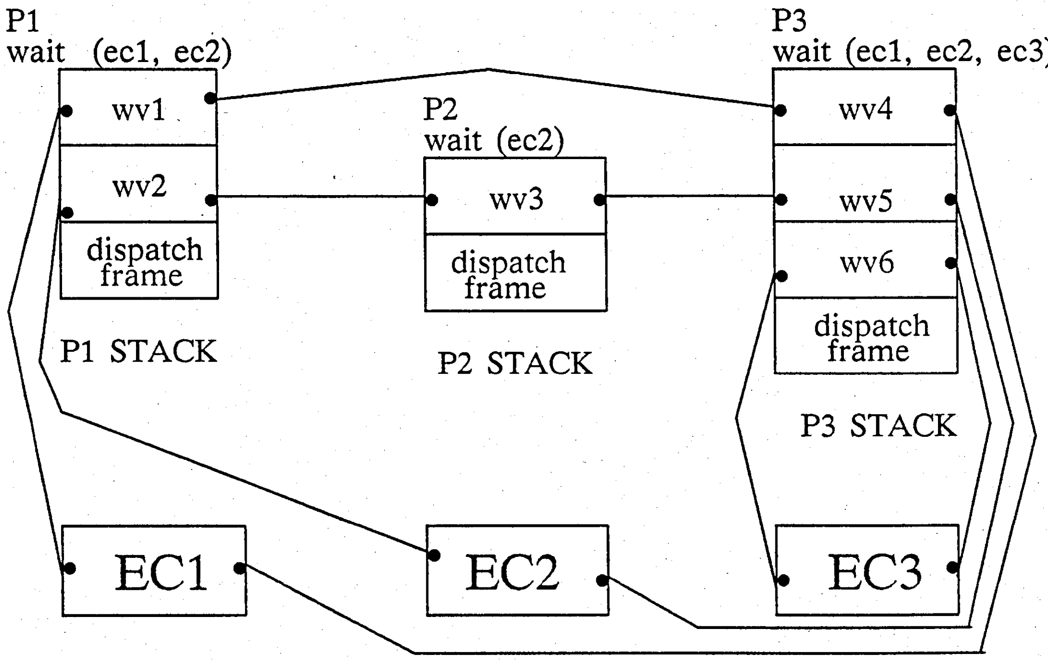
- Integrated with PROC1 Manager

- Format

  Value
  Waiters list head
  Waiters list tail

- Waiters list nodes allocated in process stack

  * wait value
  * PCB pointer
  * forward/backward waiters list links



Mutual Exclusion

- Operations

  * ML_$LOCK (resource_lock)

    obtain exclusive use of resource

    crash if RESOURCE_LOCK < = highest currently held lock (enforced by PROC_$SET_LOCK)

  * ML_$UNLOCK (resource_lock)

    release exclusion

    crash if RESOURSE_LOCK < > highest currently held lock

Mutual Exclusion (Implementation)

- Data

  * One eventcount and one lock byte for each of the 32 resource locks

- ML_$LOCK

  1. Call PROC1_$SET_LOCK— must be done first

  2. Try to set lock bit (BSET instruction) return in successful

  3. Get a "ticket" (eventcount value to wait for)

     * Must be done disabled

     * Guarantees FIFO ordering

Mutual Exclusion (Cont'd)

- ML_$UNLOCK

  1. Clear lock byte
  2. If ticket value = EC value there are no waiters –> return
  3. Advance eventcount

–Reality

  * Because these calls are very heavily used, they have been merged with PROC1, refer to PCBs directly, and are carefully coded in assembly language

Special Considerations For
2 CPU (68000)Systems

- 3 B–A Returns

  * Normal

  CPU A proceeds normally.

  * Error

  Cause bus error on A. Usually generates user mode fault.

  * Interrupt

  Cause interrupt on A. Used when process returning to A is not the highest priority. Vectors directly to PROC1_$INT_EXIT.

Special Considerations For 2 CPU (68000) Systems

- Multiple Faults in Same Instruction

  * It can happen on B–A return that an interrupt is desired because ready < > current. However, it may not happen due to second page fault in some instruction. PROC_$SET_LOCK detects this and fixes the ready list.

- Force Dispatch

  * It may happen on CPU B that ready = current but current cannot run on B. A special version of dispatch is used by PROC1_$CLR_LOCK to force a process switch.

Timer Hardware

- Battery operated “digital watch”

  * Retains date and time

  * Used only at node boot

  * Updated by standalone calendar utility

  * Not as accurate as real digital watch (~ 1 part in 10^4) — 1 in 10^5

The Real Time Clock

- Two generally accessible external variables
  - TIME_$CLOCKH—The high 32 bits of the 48 bit system time. Incremented by 1 at each interrupt from 4 usec timer (every 1/4 sec).
  - TIME_$CLOCKH_EC—An eventcount which is advanced everytime TIME_$CLOCKH is incremented.
- One procedure call
  - TIME_$CLOCK (real_time)

    Returns the full 48 bit system by reading the 4 usec timer.

Real–Time Events

- Operations

  * TIME_$WAIT (rel_abs, expiration_time)

    Blocks caller until a relative or absolute expiration time.

  * TIME_WAIT2 (rel_abs, exp_time, eventcount)

    Waits for expiration time, or for one arbitrary eventcount.

    Returns boolean –> event–count went off, no timer.

  * TIME_$ADVANCE (rel_abs, exp_time, eventcount)

    Advances eventcount when EXP_TIME is reached.

Virtual Time Events

- Handled by interrupt routine for ^{8}usec timer

- Per–process virtual time queue

- Handles repeating events, like time–slice–end

- Future virtual–time events

  * UNIX signals

  * Working set memory management

The Clock Process

- A special high priority, wired, system process (pid #3)
- Handles real–time events and time–slice ends
- One big loop waiting on a single clock process EC
- Real–time event processing
  - * List of all real–time events, ordered by absolute expriation time
  - * 32 usec timer loaded with next event
  - * Interrupt from this timer advances clock process EC
  - * Clock process discovers expired events, advances associated EC, and dequeues them.

Level Two Process Manager

- Creates and deletes user processes
- Manages UID process name space
- Passes through some PROC1 calls
- Allocates user stack files
- Maintains level 2 process stack
  - * user stack UID
  - * UNIX process ID information
  - * whether a process is an "orphan"
  - * whether a process should be stopped at logout
  - * process group UID used for fault delivery
- Implements asynchronous faults

LEVEL TWO PROCESS MANAGER

User Stack Allocation

- Maintains a pool of used user stack files to avoid file_$create / file_$delete overhead
- PROC2_$ALLOC_STACK_FILE
- PROC2_$FREE_STACK_FILE
- PROC2_CLEANUP_STACKS (subject_id)

Pass Through Operations

- PROC2_$SUSPEND (puid)
  Waits for successful suspension if necessary
- PROC2_$RESUME (puid)

Inquiry Operations

- PROC2_$LIST (puid_list, list_size, process_count) PST calls thi
  returns a list of active level 2 processes
- PROC2_$GET_INFO(p2_uid, info_buf, buf_size)
- PROC2_$WHO_AM_I (p2_uid)
- PROC2_$MY_PID
  return level 2 and level 1 names of current process

Miscellaneous

- PROC2_$MAKE_SERVER (p2_uid)
  make given process a "server"
  server processes are not stopped at logout

Create / Delete Operations

- PROC2_$CREATE
  (stack_uid, start_pc, is_orphan, new_uid)
  allocate a new address space and map the user stack (stack_uid); allocate a supervisor stack and bind all to a level one process; process will execute starting at start_pc in user mode; allocate new process group UID of orphan
- PROC2_$FORK (s~~at~~tack_uid, start_pc, new_uid)
  like PROC2_$CREATE but different treatment of new address space for UNIX; a forked process is never an orphan
- PROC2_$MAKE_ORPHAN (p2_uid)
  make the given process an orphan
- PROC2_$DELETE
  delete the calling process and release all the resources; calls almost all nucleus managers to cleanup their per–process data; if orphan, frees the user stack; otherwise advances the process termination eventcount; cannot currently delete other processes

LEVEL TWO EVENTCOUNTS

- Like level one except that eventcounts are unwired and can be anywhere in Virtual Memory

- Level two calls can also wait on level one eventcounts – they are recognized by their special addresses, obtained from manager specific calls that return them

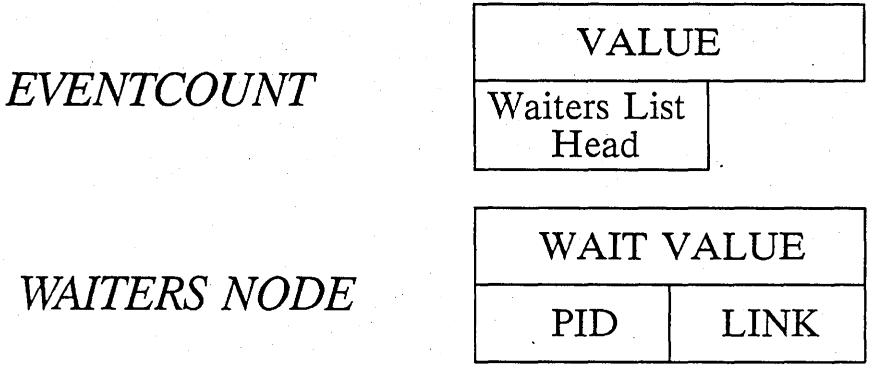
- Level two eventcount calls do not work over the network

- Operations are almost identical to level one; manager name is EC2 Documented in System Programmer manual

LEVEL TWO EVENTCOUNT IMPLEMENTATION

Data Structures

- One level 1 ec per process; all EC2_$WAIT calls wait on~~e~~ this
- Each level two ec heads a linked list of WAITERS NODES:



EVENTCOUNT

VALUE

Waiters List Head

WAITERS NODE

WAIT VALUE

PID

LINK

- EC2_$WAIT
  For level 2 ec : allocate and chain a waiters node
  For level 1 ec: include in ec_$waitn call
- EC2_$ADVANCE
  Runs in user mode for speed if no waiters;
  Increment value; if waiters list is not null, call EC2_$WAKEUP (an SVC)
- EC2_$WAKEUP
  Search waiters list for any satisfied wait values
  If found, remove from list and advance the level one ec of the corresponding process

User Mode Process/Program Management

- ◇ Program Levels, Processes, and Fork
- ◇ The Stack File
- ◇ Mapped Segment Manager (MS)
- ◇ Storage allocator (RWS)
- ◇ The loader, KGT, etc.
- ◇ Libraries, global and private

The User Program Environment

◇ Contains:

- A storage (virtual memory) allocator
- A mapped file manager
- A stream manager
- Some "standard" streams
- Some program arguments
- Exception handling mechanisms

◇ Semi–isolated

- Parent affects child only by
  - passing arguments
  - passing streams
  - inherited state
  - pre–arranged sharing
- Child affects parent only by
  - returned status
  - "permanent" side–effects
  - pre_arranged sharing

◇ Design Trade–offs

- What state to inherit automatically
- What system calls should have "permanent" side–effects (e.g. gpr_$init, stream_$create, pad_$def_pfk)

New Process vs. Same Process

◇ **Goal: make them identical except for**

- performance
- potential concurrency — No concurrency if you do pgm_$wait
- address space available

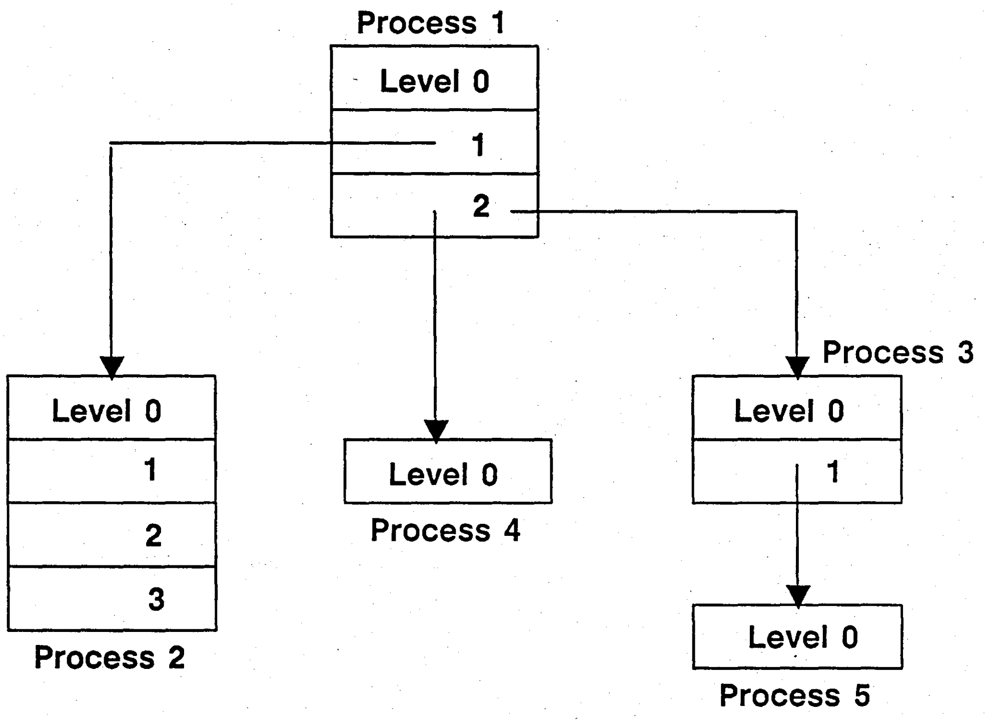
◇ **Reality:**

- Substantial performance penalty for new process
- New process can't use private libraries
- Complex export–import operations required to use most resources in new process — most managers (e.g. gpr, smd, gpio, magtape) don't implement.
- pgm_$invoke for new process not documented

pgm_$invoke [ ] — null set of arguments makes it a child process

◇ **Result: customer use of multiple processes is very limited**

Program Environment Tree



Each small box is a separate program environment
Within a process, program levels form a stack

Calls That Create Program Environments

◇ **pgm_$invoke_s(name, name_len, argc, argv, sidc, sidv, flags, ecp, status1, status2)**

- makes a new process if
  - ○ pgm_$wait NOT in flags
    - – creation record left mapped in parent
    - – parent can wait for termination and check status
  - ○ pgm_$background in flags
    - – creation record unmapped
    - – process disappears when done
  - ○ program is a protected subsystem
    - – caller waits for termination

◇ **pgm_$exec(name, namelen, argc, argv, env, status)**

- like pgm_$invoke, except
  - ○ never makes a new process
  - ○ first exits current level with partial cleanup
  - ○ doesn't rearrange streams

to support UNIX exec

Miscellaneous Process–related Calls

◇ **pm_$finish(ecp, status)**

- Waits for process termination
- Returns its status
- Unmaps creation record
- Releases stack file
- Note: this call should be made even if ec2_$wait is used

◇ **pm_$make_orphan(ecp, p2uid, status)**

- Makes process an orphan
- Returns process UID (all subsequent references must use this instead of ecp)
- This operation cannot be undone

Process Names

◇ Processes are initially unnamed

◇ Name can be assigned by creator or by process itself

◇ Names are just process UIDs, cataloged in ‘node_data/proc_dir

◇ Name can only be set once (because there is no way to tell DM to change name in banner)

◇ Several PM_$ calls to set/inquire process names

Fork

- ◇ **pm_$fork(is_vfork, parent_SP, child_puid, child_suid, ecp, status)**

- ◇ **Makes a new process**
  - copies the parent's stack file
  - copies the parent's address space, except that references to parent's stack are replaced with references to child's stack

- ◇ **Managers with global state (e.g. streams) must be informed**
  - streams pre–fork/post–fork
  - pfm_$static_fork

Vfork

◇ **Push a program level**

◇ **Make a new process**

- Address space is an EXACT duplicate of parent

◇ **Parent waits until child executes PGM_$EXEC**

- Child's activity during this time limited mainly to streams operations

◇ **When child executes PGM_$EXEC**

- Address space is cleared
- Equivalent of new process pgm_$invoke is done, using already created process
- New stack file is initialized at this point

◇ **Parent resumes execution, and pops a program level to recover streams state**

Stack File Allocation

Holds ALL per-process read-write data

File offset		Virtual Address
0	Creation record . termination eventcount . termination status . arguments . exported streams . program to execute . login info . UNIX context	200000
8000	Per process static data for global libraries	208000
30000	guard segment	230000
38000	User mode execution stack	238000
78000	guard segment	278000
80000	Storage managed by RWS	various

Mapped Storage Manager (MS)

- maps objects into the private address space

- handles object locking and unlocking

- objects are automatically unmapped and unlocked at level exit

- based on kernel FILE and MST managers

- used by EVERYBODY, including other PM services
  (read / write storage manager)

MS_$MAPL (name, len, start, length, conc, access,
extend_ok, length_mapped, status):
univ_ptr

- maps the area of the file ‘name’ (‘len’ chars) starting at offset ‘start’ for ‘length’ bytes
- returns the virtual address of the first byte mapped (function value), and the number of bytes mapped (‘length_mapped’)
- locks the file according to (conc, access); ‘conc’ specifies the desired concurrency control:

ms_$nr_xor_1w	N readers XOR 1 writer
ms_$cowriters	N readers and N writers*
ms_$none	no locking

- *cowriters must be on the same node
- ‘access’ specifies the desired access to the file:

ms_$r	read
ms_$rx	read, execute
ms_$wr	write, read
ms_$wrx	write, read, execute
ms_$riw	read intend to write

- allows file growth if extend_ok is true

MS_$MAPL_UID (uid, start, length, conc, access,
extend_ok, length_mapped,
status): univ_ptr

- similar to MS_$MAPL, except 'uid' is specified in lieu of 'name' and 'len'

MS_$CRMAPL (name, len, start, length, conc,
status): univ_ptr

- similar to MS_$MAPL, but creates the object and catalogs it under 'name', 'len'
- object is mapped for read / write
- extend_ok is true (it MUST be!)
- object is made permanent

MS_$CRMAPL_UID (uid, start, length, conc,
status): univ_ptr

- similar to MS_$MAPL_UID except that an object is created and its uid is returned
- object is NOT made permanent

the only permanent unnamed file is the OS paging file.

MS_$CRTEMP (location, len, start, length, conc, status): univ_ptr

- like MS_$CRMAPL but creates a temporary, unnamed object
- 'location', 'len' descibe the volume on which the temporary object is to be created

MS_$REMAP (va, start, length, length_mapped, status): univ_ptr

- unmaps a portion of the object at 'va' and maps a new section ('start', 'length')
- object stays locked as before

MS_$ADDMAP (va, start, length, length_mapped, status): univ_ptr

- maps an additional part of object mapped at 'va'
- object at 'va' is not unmapped
- object remains locked as before
- object is unlocked when the oldest part is unmapped

MS_$UNMAP (va, length_mapped, status)

- unmaps the object specified by 'va' and 'length_mapped'
- unlocks the object if this 'va' was returned from from a procedure other than MS_$ADDMAP

MS_$UNMAP_PARTIAL

used by loader

- unmaps part of a mapping done by one of the MS_$xxMAPxx procedures
- does not unlock the object

MS_$RELOCK (va, access, status)

- changes the lock on an object
- access must be 'ms_$r' or 'ms_$rw'

MS_$ATTRIBUTES (va, attributes, actlen, maxlen, status)

- returns the attributes of the object mapped at ‘va’
- attributes include:
  permanent flag
  immutable flag
  current length
  disk blocks used
  date/time used, modified, created

MS_$TRUNCATE (va, length, status)

- truncates object mapped at ‘va’ to ‘length’ bytes

MS_$MK_PERMANENT (va, opts, name, len, status)

- makes a temporary object (created with MS_$CRTEMP) permanent and names it
- optionally creates a backup file if an object with an identical name exists

MS_$MK_TEMPORARY (va, status)

- makes a permanent file (mapped at ‘va’) temporary
- drops its name

MS_$MK_IMMUTABLE (va, status)

– makes the object mapped at 'va' immutable

MS_$NEIGHBORS (va1, va2, status): boolean

– determine if the objects mapped at 'va1' and 'va2' reside on the same disk volume

MS_$FW_FILE (va, status)

– causes the file mapped at 'va' to be force–written to disk

– doesn't return until the forced write completes

MS_$FW_PARTIAL (va, length, status)

– force writes part of the object mapped at 'va'

– 'length' bytes are force–written

– doesn't return until the force write is complete

MS_$STREAMS_FLAG (va, flag, status)

– sets an internal flag saying, "the mapping at this virtual address is owned by a STREAMS type manager"

– needed because of UNIX 'exec' primitive

– required because of mangers orientation to 'Mark/Release' instead of 'Resouces'

Storage Allocation (RWS)

◇ Basic call:

- p := rws_$alloc_rw_pool(size, {rws_$std_pool | rws_$streams_tm_pool})
- Allocates non-returnable vanilla virtual memory
- Recovered at program termination
- rws_$streams_tm_pool used to avoid recovery at pgm_$exec (because streams are supposed to stay open across EXEC.

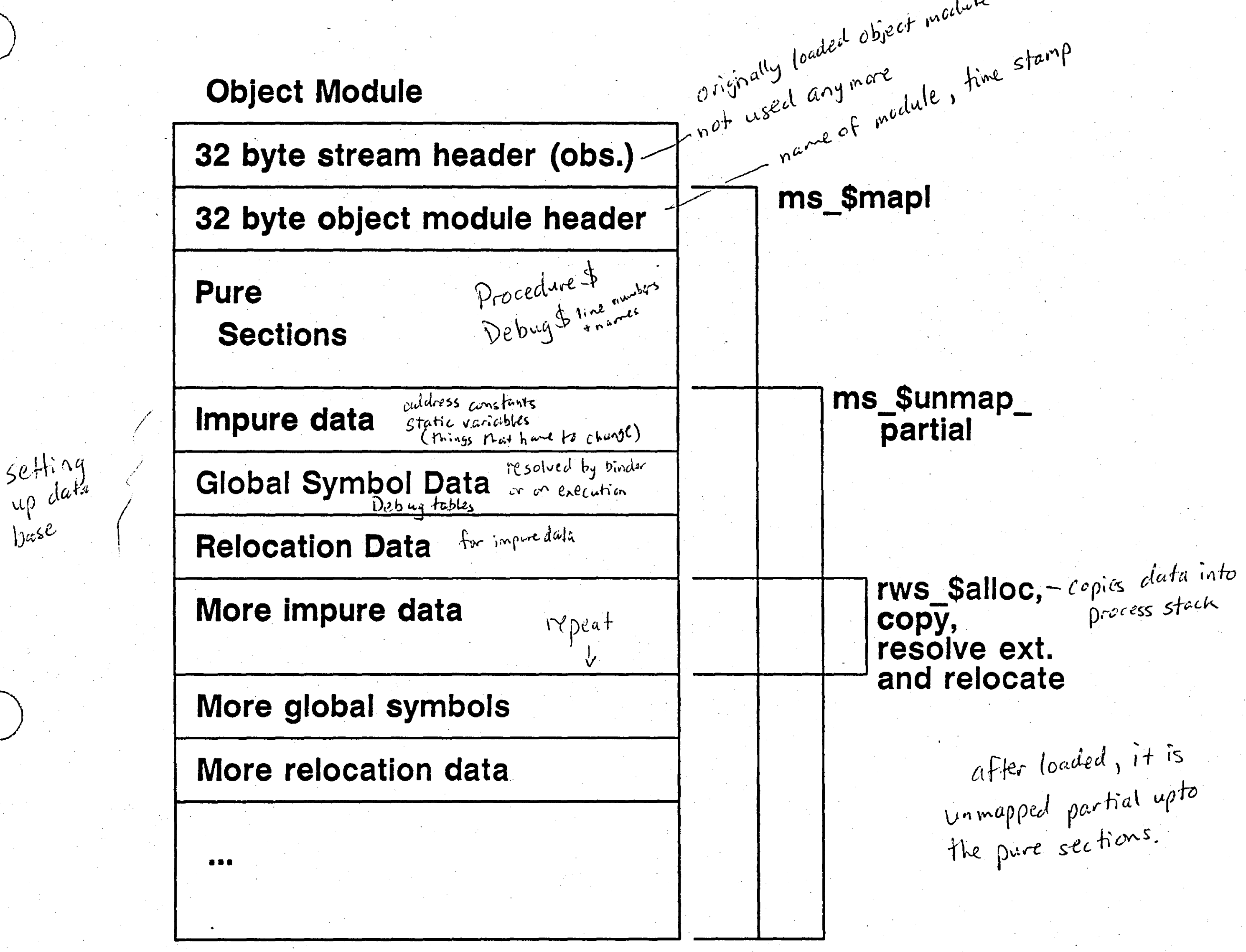
◇ Implementation

- Maintain high water mark in stack file
- Allocate and ms_$mapl in multiples of a segment
- Maintain VM high water mark within a given stack allocation
- Just push and pop high water marks at program level transitions. MS cleanup takes care of the rest

◇ Heap allocation

- rws_$alloc_heap_pool and rws_$release_heap
- Layered on rws_$alloc_rw_pool
- Maintains special free-lists for small blocks
- 16 bytes overhead precedes each allocated block
- Not notably fast

The Loading Process



Object Module

32 byte stream header (obs.)

32 byte object module header

Pure Sections

Impure data

Global Symbol Data

Relocation Data

More impure data

More global symbols

More relocation data

...

ms_$mapl

ms_$unmap_ partial

rws_$alloc, copy, resolve ext. and relocate

originally loaded object module with streams
not used anymore

name of module, time stamp

Procedure $
Debug $ line numbers + names

address constants
static variables
(things that have to change)

resolved by binder
or on execution

Debug tables

for impure data

repeat ↓

setting up data base

copies data into process stack

after loaded, it is unmapped partial upto the pure sections.

◇ **Note that normal cleanup of MS and RWS managers takes care of unloading**

The more modules, the more fiddling
with setting up data bases, entry control blocks: the longer the loading process.

When you execute a program,
and see an unresolved global,
you use the KGT to resolve it.

P.M.- load from process manager

* Dynamic initialization is much, much faster.
the data stays in memory because you touch it.

to process

Private Libraries (INLIB)

◇ Start with normal load

◇ Enter marked global symbols into private KGT

◇ Call main program - do initialization here

◇ Persists only until termination of current program level

◇ Hence INLIB is an internal shell command

run shlib through binder
with global switch to see
all the globals.

bind -sys <system command> to
see the globals called by the command.

Unresolved Globals

◇ **Never terminate any loading process**

◇ **Generate TRAP instruction, followed by symbol name, in DATA$**

◇ **When trap occurs at run time, KGT is tried again**

Second chance to resolve global

runtime dynamic resolution

- if successful, TRAP is replaced by JMP
- otherwise fault handling proceeds

afterwards no further attempt at resolution.

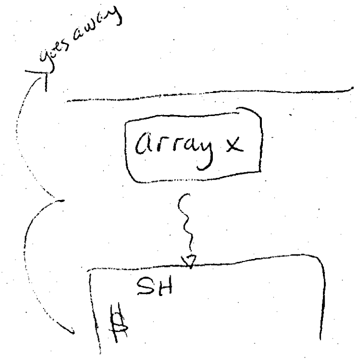
Pascal external is a named section.

Threekinds of Known globals

System KGT

per process KGT

program level KGT



Global Libraries

loading applies to programs
Installing does more; it builds KC

◇ Installed ~~Loaded~~ by ENV in response to DM, SH, SPM, or GO

Knows what libraries to install and where they are.

◇ Use mst_$map_global instead of ms_$mapl – used for private asid mapping.

Global libraries made read only after loaded and KGT is built.

◇ Use globrws_$alloc_rw for DATA$ section

◇ Use privrws_$alloc_rw for impure sections other than DATA$

this gets created per process.

- Skip initialization – main program doesn't run
- Map stack file into appropriate range of private address space in pm_$init

◇ Make DATA$ read-only after loading is complete

- Shared storage managers initialized first

◇ Main program called in every new process /lib /userlib.private

- Hence should be avoided if library is not always needed

◇ "Dynamic linking" not possible because you can't write there.

Can't be any unresolved globals at all. If your userlib.private uses a GPR call in its main program, for instance, then the GPRLIB must be bound to USERLIB.PRIVATE.

links set up in impure area

Error and Fault Handling

- ◇ **Kinds of faults**
- ◇ **Supervisor mode fault handling/generation**
- ◇ **User mode fault generation**
- ◇ **Fault handlers**
- ◇ **Dynamic Cleanup Handlers**
- ◇ **Static Cleanup Handlers**
- ◇ **Mark/Release**

Error and Fault Handling

◇ **Kinds of faults**

◇ **Supervisor mode fault handling/generation**

◇ **User mode fault generation**

◇ **Fault handlers**

◇ **Dynamic Cleanup Handlers**

◇ **Static Cleanup Handlers**

◇ **Mark/Release**

Kinds of Faults

◇ Program error

- Unimplemented instruction
- Odd address error
- Reference to illegal invalid address not in your address space
- Access violation don't have rights
- Reference to unresolved global
- Guard fault (stack overflow)

◇ System error

- Network failure (e.g. too many transmit retries)
- Disk full
- Disk error

◇ Asynchronous

- Quit
- Stop
- UNIX signal (e.g. child death)

use alarm server to monitor disk

Supervisor Mode Fault Handling (synchronous)

◇ Address–related faults

- These are all page faults that cannot be resolved, either because of a user program error, or due to system failure
- Assign appropriate status code
- On 68000 systems, return to CPU A with a bus error
- If fault occurred in supervisor mode:
    - ○ If address in supervisor range, crash system
    - ○ Otherwise, report both supervisor and user mode state
- Go to fim_$com to report fault to user mode

◇ CPU–detected faults

fault interceptor module

- Just set the status code, and go to fim_$com

◇ Common fault handling

- Push a fault frame on the user mode stack
- If this causes another fault, process dies
- Fault frame contains registers, PC, status, etc.
- Fault frame flagged with 16#DFDF
- Force supervisor stack to contain a simple exception frame with PC set to the user mode fim (set by fim_$install)
- RTE

Asynchronous Fault Generation

◇ **Set desired fault status in fim_$trace_status**

◇ **Set trace-trap bit in supervisor stack of process to receive fault**

◇ **Advance fim_$quit_ec to get process out of nucleus if necessary — long waiters also wait on this and fim_$quit_value**

◇ **When trace-trap occurs, use fim_$trace_status, and go to fim_$com to complete fault handling normally**

◇ **Disabling handled in user mode support**

◇ **User mode must acknowledge fault (using fim_$acknowledge) before further asynchronous faults can occur**

Set supervisor bit in machine status register to get above protection boundary.
svc_trap

Multiple Asynchronous Faults

◇ proc2_$trace_fault(p2_uid, fault_status, status)

- Error if a fault is pending which has not yet been acknowledged by fim_$acknowledge
- DM says ”another fault is pending for this process”
- May be inhibited in user mode by pfm_$inhibit, due to user program or system library error in missing a re-enable
- May be hung in nucleus in a call (network retry is typical) that doesn't wait on fim_$quit_ec
- User fim may be trashed and getting faults in the fault handler before previous fault can be acknowledged

◇ proc2_$trace_fault_enq(p2_uid, fault_status, status)

- Enqueues multiple faults
- Subsequent faults delivered after fim_$acknowledge
- Used by UNIX signal mechanism to avoid losing faults

Process Groups

◇ **This mechanism supports AUX**

◇ **It only affects asynchronous fault delivery**

◇ **A parent and its child (either pm_$fork or pgm_$invoke) are in the same process group**

◇ **A background process (pgm_$background to pgm_$invoke, or pgm_$make_orphan) starts a new process group**

◇ **A process may decree itself to be in a new process group**

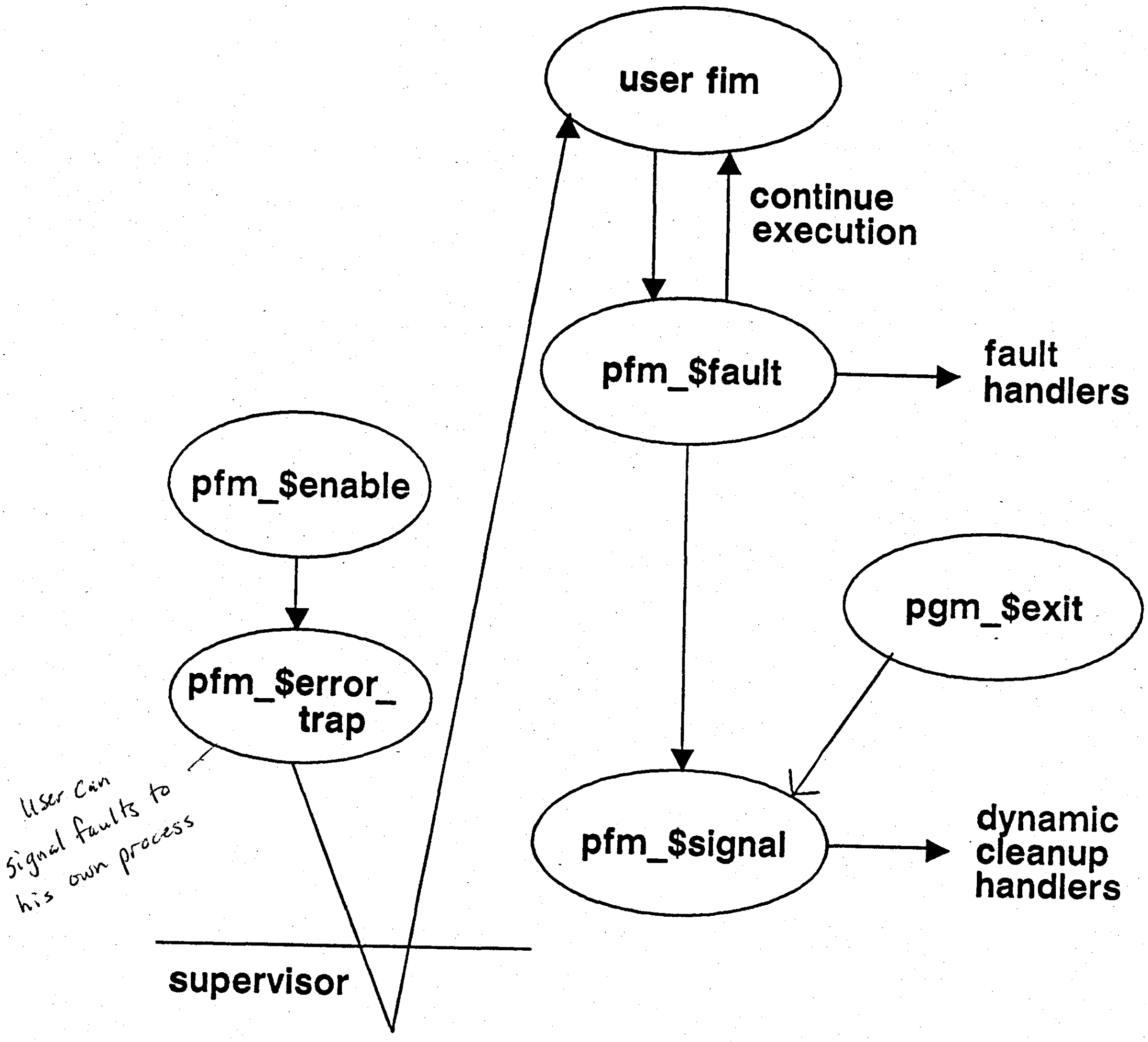
◇ **A process group is denoted by a UID**

◇ **proc2_$trace_fault_pgroup and proc2_$trace_fault_pgroup_enq**

- Deliver faults to all members of process group
- Process UID may ~~me~~ be used to denote the process group it is in
- The DM uses this form of the call for quits

You can send a fault to a group of processes.

User Mode Fault Layering



all transitions from user space to supervisor space are done via traps.

Fault Handlers

In pascal the function must be in an external module.

◇ **Always "static" (i.e. not related to call stack)**

not in normal order on your stack frame.

◇ **Established by pfm_$establish_fault_handler(func_ptr)**

- Returns handle for later release
- Func_ptr is a Pascal (or C) function pointer whose single argument is the fault frame constructed in the nucleus

◇ **Called in inverse order of establishment, by pfm_$fault**

◇ **Not called on asynchronous faults if inhibited**

◇ **Return value from fault handler can cause fault to be ignored, if restart is possible**

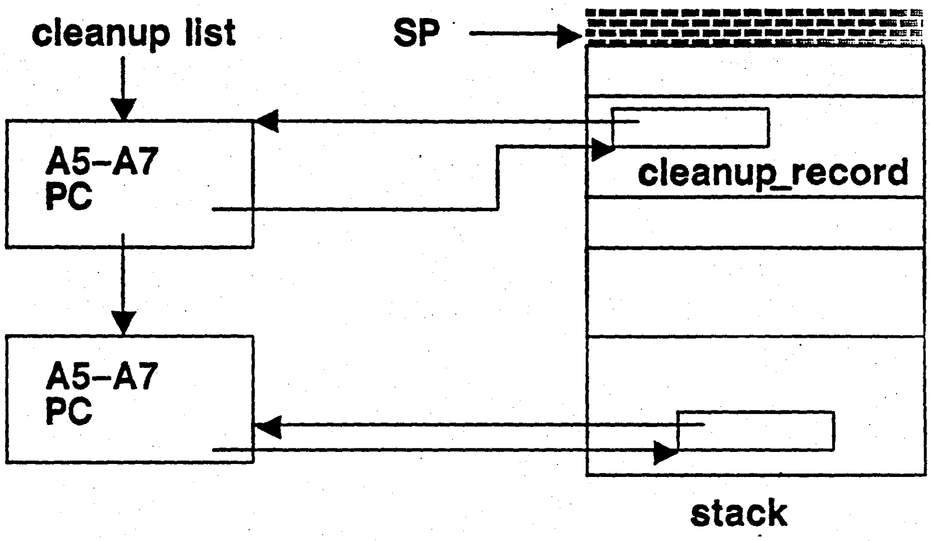
- restartability is recorded in the fault frame by the nucleus, depending on the nature of the fault -- addressing faults are usually not restartable
- if a fault handler says to ignore the fault, no further fault handlers are called, and the program is restarted
- if no fault handler says to ignore the fault, then proceed to pfm_$signal, and dynamic cleanup handlers

all asynchronous faults are restartable : use return rather than exit.

Dynamic Cleanup Handlers

Fault handlers are always static

◇ **Associated with active call frames on stack**



◇ **Activated (not called) by pfm_$signal**

- thus includes all program termination except return from main program

◇ **Return to exception handling only by resignal**

◇ **Cleanup handler automatically released when activated.**

◇ **pfm_$inhibit done automatically**

Dynamic Cleanup Handlers (page 2)

◇ **Consistency checking**

- cleanup list scanned for handler with SP >= current SP
- cleanup record checked for overwriting due to reuse of stack frame exited without pfm_$release_cleanup

◇ **These cleanup handlers are moderately expensive in relation to a simple procedure call. We are working on a cheaper mechanism**

◇ **We should really have language support for this, but...**

Typical Cleanup Handler Usage

```
VAR
    cleanup_rec: pfm_$cleanup_rec;

BEGIN
    ...
    status := pfm_$cleanup(cleanup_rec);
    IF status.all = pfm_$cleanup_set THEN
    BEGIN
        { normal operation }
        pfm_$release_cleanup(cleanup_rec);
    END
    ELSE BEGIN
        { cleanup the mess we started }

        {  depending on the operation we desire, either: }
        PFM_$ENABLE;
        RETURN; { turns fault into normal bad status from
                  this procedure }

        {  OR  }
        pfm_$signal( status );  { resignal other cleanup
                                  handlers }
    END;
END;
```

Disabling Asynchronous Faults

◇ **pfm_$inhibit**

- Increment inhibit counter

◇ **pfm_$fault**

- If fault is asynchronous (recorded in fault frame by nucleus fim) and inhibit count is not zero~~k~~), record status and ignore fault.

◇ **pfm_$enable**

- Decrement inhibit counter

- If zero, and status recorded by pfm_$fault, then pfm_$error_trap

◇ **Many system calls (e.g. ec2_$wait_svc, but not ec2_$wait) will return error status if asynchronous faults are inhibited and one occurs**

◇ **Note: these calls ONLY inhibit asynchronous faults. Since it is very difficult to prevent asynchronous faults altogether, it is best to use a cleanup handler if you need to be robust and can afford the cost.**

Program Initiation/Termination

◇ **A. K. A. Mark/Release**

◇ **pm_$proc_mark**

- called by pgm_$invoke after program is loaded and streams switched
- pm_$level <– pm_$level + 1
- call mark/release handlers
- establish normal cleanup handler
- set status/severity to status_$OK
- if not cleanup, call main program
- call pm_$release

◇ **pm_$proc_release**

- call static cleanup handlers
- pm_$level <– pm_$level – 1
- call mark/release handlers

◇ **pgm_$set_severity**

- Set status.code (used in pm_$mark) to the severity value

Alternative to
System mark/release handlers

Static Cleanup Handlers

Best one to
use for private libraries;
only goes through code once
whereas dynamic handlers
are executed for each
program level.

◇ **Executed (called) at program termination, from the level at which handler was established**

◇ **Established via pfm_$static_cleanup(ecb_addr, status)**

◇ **Called in inverse order of establishment**

◇ **Calling sequence is**

- handler(false, new_level_number, termination_status, is_exec)

◇ **No actual relation to fault handling**

◇ **Preferred method of cleanup for managers in global or private libraries (better than a mark/release handler)**

◇ **Try to avoid depending on managers other than MS, RWS, STREAMS in your static cleanup handler, since other managers' cleanup routines may be called before yours (we should fix this, but are not sure how)**

Mark/Release Handlers

◇ **Like static fault handlers except:**

- called on all level transition, both up and down

◇ **Use when**

- you need to keep client status at each level
- you need to initialize default state for new programs
- you have to "init" call where you could conveniently establish a static cleanup handler
- almost all programs will use your services (e.g. streams)

◇ **Otherwise use a static cleanup handler, established in your "init" call, and released in your "terminate" call.**

Fault State and Traceback Recording

◇ Information reported by FST and TB commands

◇ At the end of pfm_$fault, and before pfm_$signal, the registers, etc., in the fault frame are copied to a global buffer for later use. Alsok the stack is scanned (if possible) and routine names and line numbers are put in another global buffer

◇ Traceback collection sometimes gets a second fault

◇ pfm_$fault_info traces your own stack.

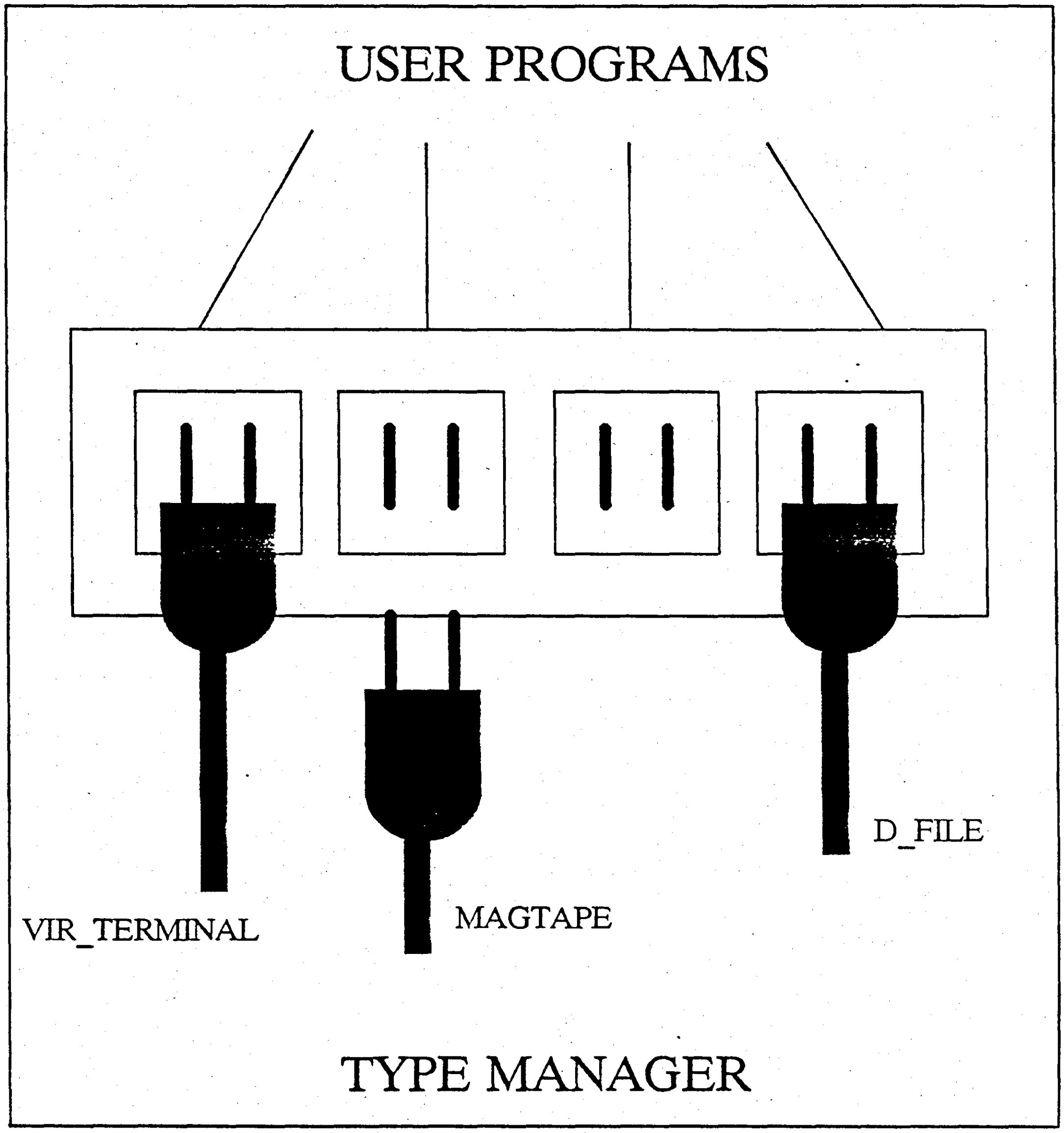
◇ pfm_$trace_info given UID of a process, it will trace stack of process.

object module will be 9-15% smaller if created with no debug info.

THE STREAM MANAGER

- Device Independent I/O
- A Big Switch

Switch box switches to appropriate type manager.



USER PROGRAMS

VIR_TERMINAL

MAGTAPE

D_FILE

TYPE MANAGER

Topics

- The Stream Table
- Opening Streams
- The Generic Switch Call
- Some Special Switch Calls

disk

- The D_FILE Manager
- Other Managers

THE STREAM TABLE

- The Database of the Switch itself
- Array [0...127] of stream_table_entry

pfcb:
process file control block

- Each entry is :

UID

HANDLE

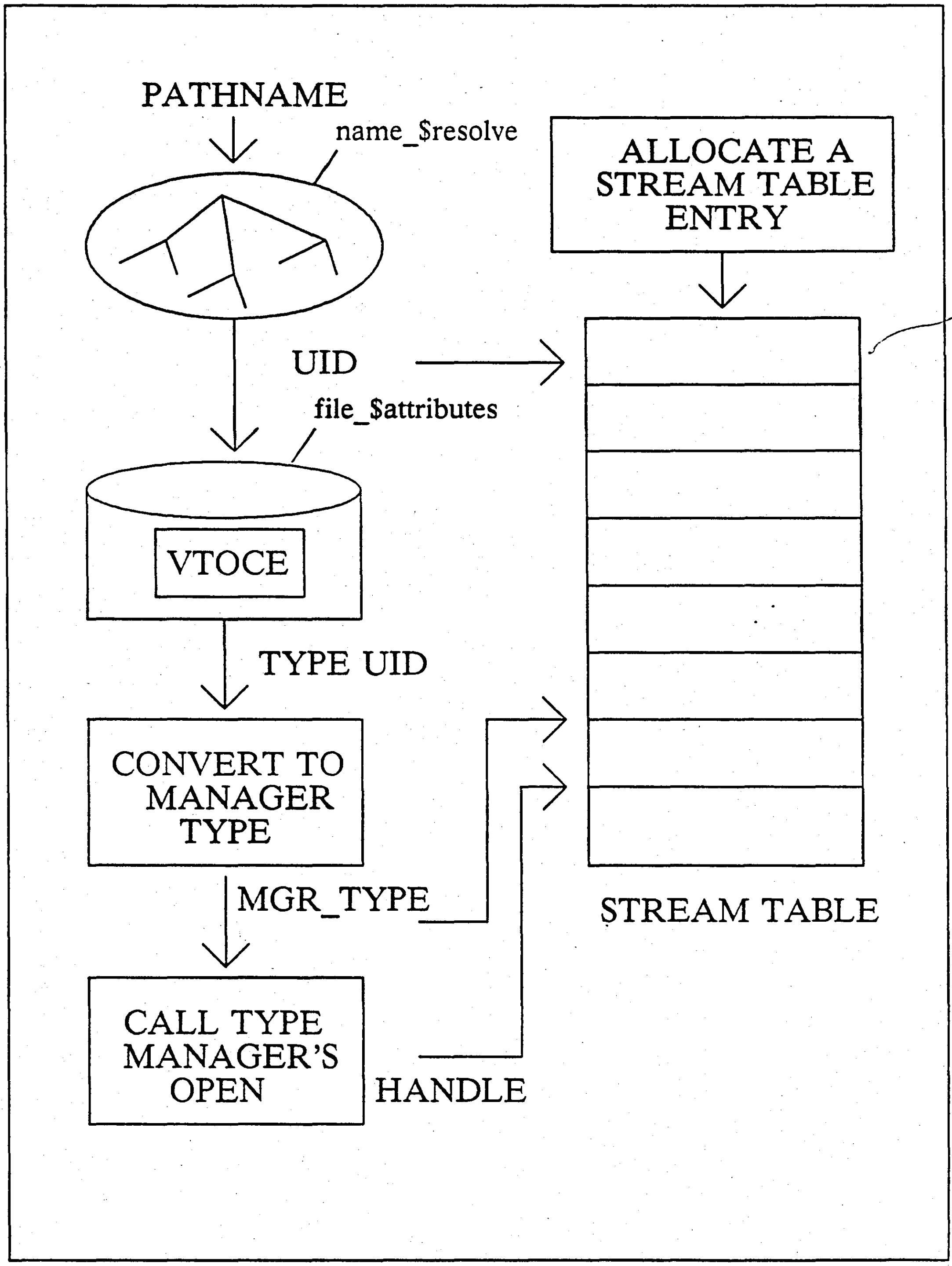
MANAGER TYPE

OPEN PM_LEVEL

SOME UNIX BITS :

* close_on_exec
* ndelay

OPENING A STREAM



PATHNAME

name_$resolve

UID

file_$attributes

VTOCE

TYPE UID

CONVERT TO MANAGER TYPE

MGR_TYPE

CALL TYPE MANAGER'S OPEN

HANDLE

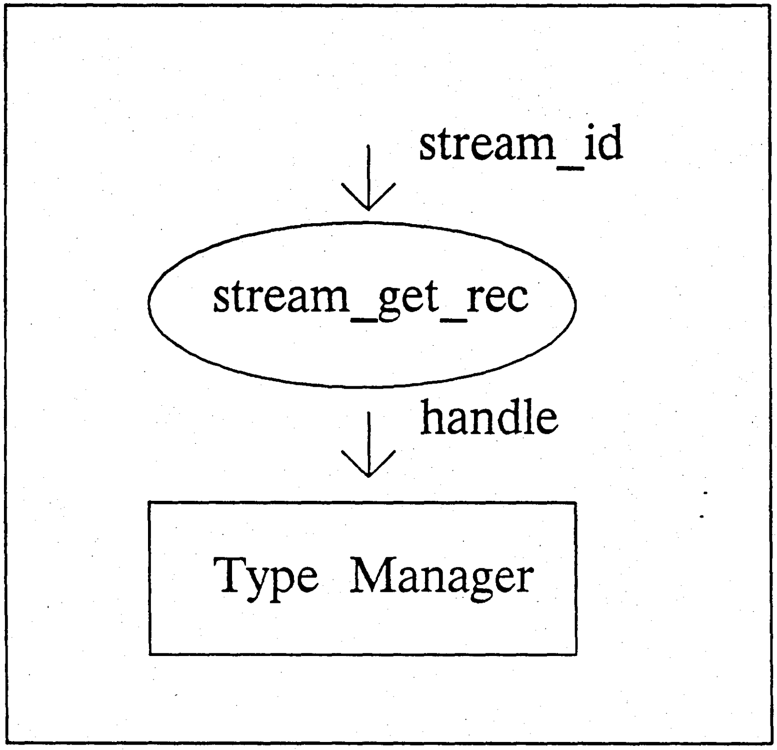
ALLOCATE A STREAM TABLE ENTRY

STREAM TABLE

Pfcb
in writable
per process data
area (your stack

A TYPICAL CALL

- stream_$get_rec



```
WITH stream_table[stream_id] DO
      CASE manager_type OF
      d_file: dfile_$get_rec(handle,args...)
      vir_term: vt_$get_rec(handle,args...)
                        ...
                          virtual terminal
END
```

Stream Table Operations

- STREAM_$SWITCH

    * Move stream table entry to a different stream id.

    * Caller can specify new sid — otherwise allocate downward from 127

- STREAM_$REPLICATE and

  different Pfcb entries when closed, its not closed for both.

  STREAM_$DUP

  two programs share the same seek key, when closed its closed for both,

    * Copy stream table entry to a different sid

    * Two resulting streams are indistinguishable by type manager

    * PM_OPEN_LEVEL and some other STREAM_TABLE values may differ

    * MGR_$REPLICATE is called to increment replication count

    * DUP & REPLICATE differ in order of allocating new sid

Inquire/Redefine

- Mixture of switch attributes and manager specific attributes—manager called only if switch can't do operation itself.
- Pathname operations done in switch, since manager is pathname independent.
- Best to operate on only one attribute per call, so sensible errors can be reported.
- Growing number of inquires that manager must answer makes manager implementation tedious.
- MGR_$INQUIRE must be able to open object temporarily, for inquire by name.

IMPORT/EXPORT

- Like replicate, except new stream is in a different process.

- Used to pass standard streams to a new process.

- Both manager data and stream table data, which are not shared, must be packed for export.

  export
- STREAM_$GET_XP_BUF
  - Call MGR_$EXPORT to package data
  - Add STREAM_TABLE data
  - Caller provides buffer (in creation record for PGM_$INVOKE)
  - Also called by PAD_$CREATE[_WINDOW]

if streams are passed
correctly you can use
them as pipes between
processes.

std input of processA → std output ~~to~~ process B

IMPORT/EXPORT (Cont'd)

export

- STREAM_$OPEN_XP_BUF
  - * Allocate and fill STREAM_TABLE entry
  - * Call MGR_$IMPORT
  - * Called by PM_$INIT in new, process
- STREAM_$FORK
  - * Just call MGR_$FORK—data already copied

Manager Specific Functions

- Operations that are not common to all types of streams
  - e.g. PAD_$USE_FONT, SIO_$CONTROL
- They take a STREAM_ID as argument, however
- These entries must look in the stream table to find their handles, and to check that the stream is open and has the right type.
- MGR_$CREATE is a manager specific function because there is no open stream involved, and no object from which to derive the type.
- STREAM_$CREATE is mis–named. It should be D_FILE3_CREATE.

Creates a UASC file on the disk.

The D_FILE Manager

- The file structure
  - * VTOCE, stream header
- The open stream structure
  - * PFCB, SFCB
- "Windowing" rather than mapping entire file
- Data Organization
  - * D_FILE1

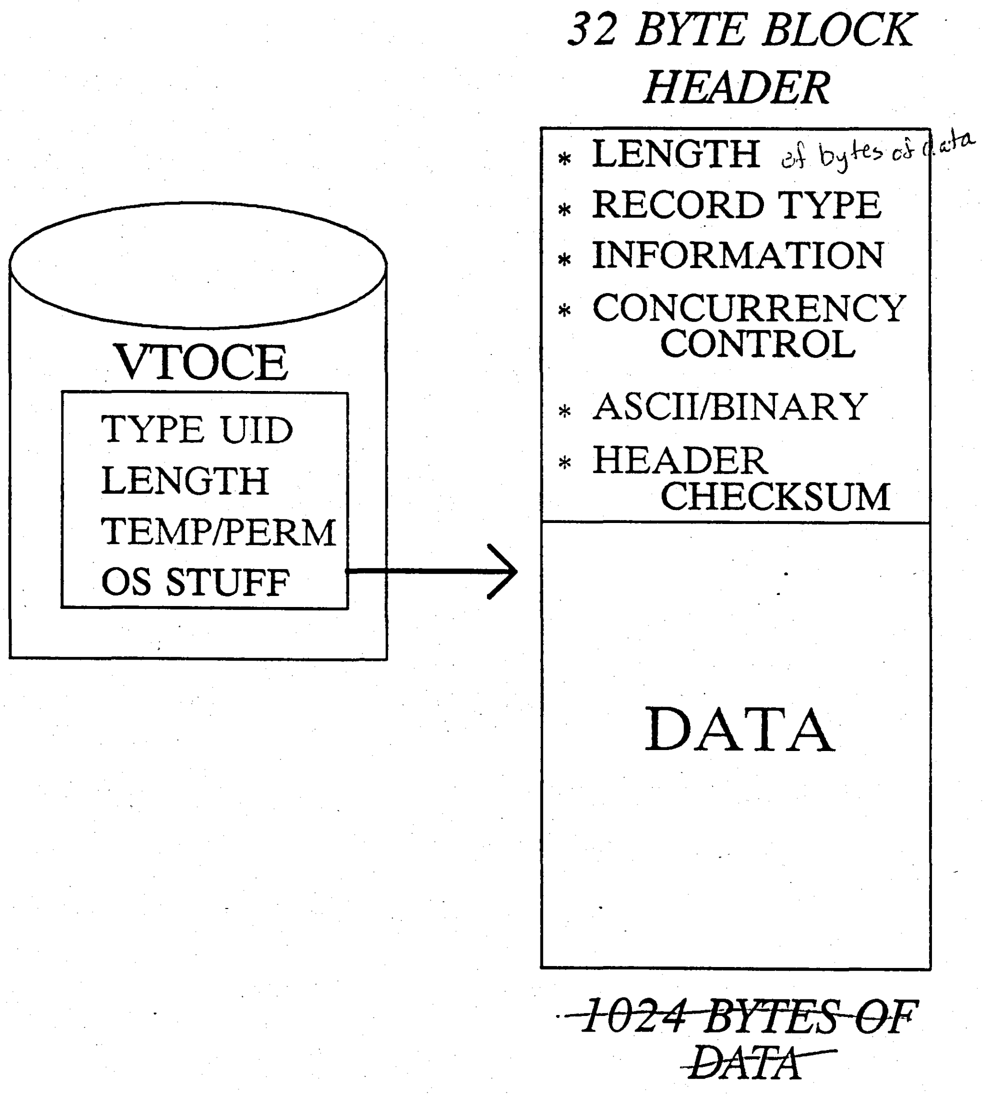
    Counted Records (REC)
  - * D_FILE2

    like bitmaps

    Byte Stream (UNDEF)
  - * D_FILE3

    Byte Stream (UASC) defined internally by CR's
- Locking and Concurrency

THE FILE STRUCTURE



VTOCE

TYPE UID
LENGTH
TEMP/PERM
OS STUFF

32 BYTE BLOCK HEADER

- ∗ LENGTH of bytes of data
- ∗ RECORD TYPE
- ∗ INFORMATION
- ∗ CONCURRENCY CONTROL
- ∗ ASCII/BINARY
- ∗ HEADER CHECKSUM

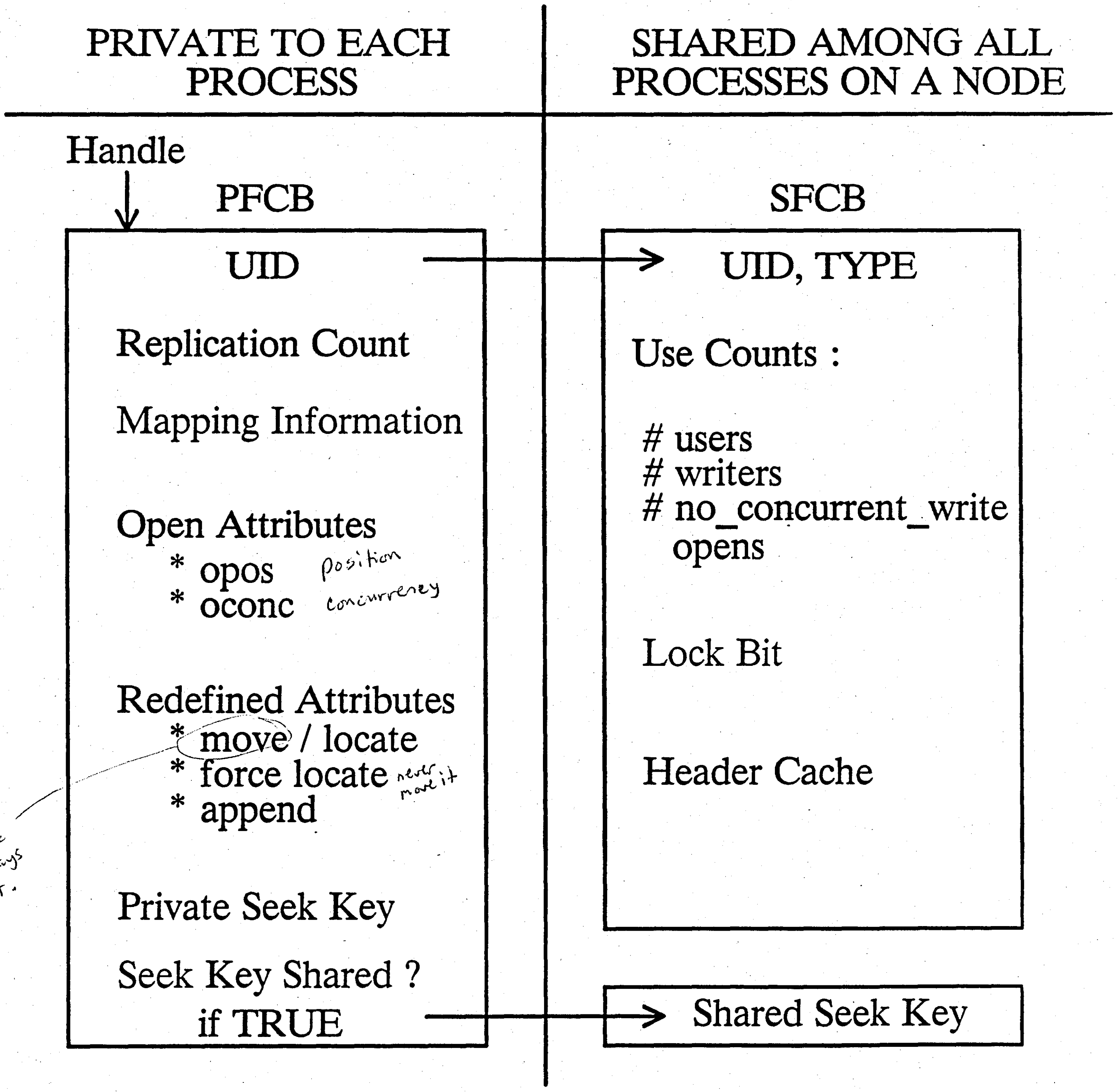
DATA

~~*1024 BYTES OF DATA*~~

to look at non-uasc
use: db map file
then dump from
Start + 20

use chpat to
insert control characters
into a file.

THE OPEN STREAM STRUCTURE



PRIVATE TO EACH PROCESS

SHARED AMONG ALL PROCESSES ON A NODE

Handle

PFCB

UID

Replication Count

Mapping Information

Open Attributes
- * opos position
- * oconc concurrency

Redefined Attributes
- * move / locate
- * force locate never move it
- * append

Private Seek Key

Seek Key Shared ? if TRUE

SFCB

UID, TYPE

Use Counts :

# users
# writers
# no_concurrent_write opens

Lock Bit

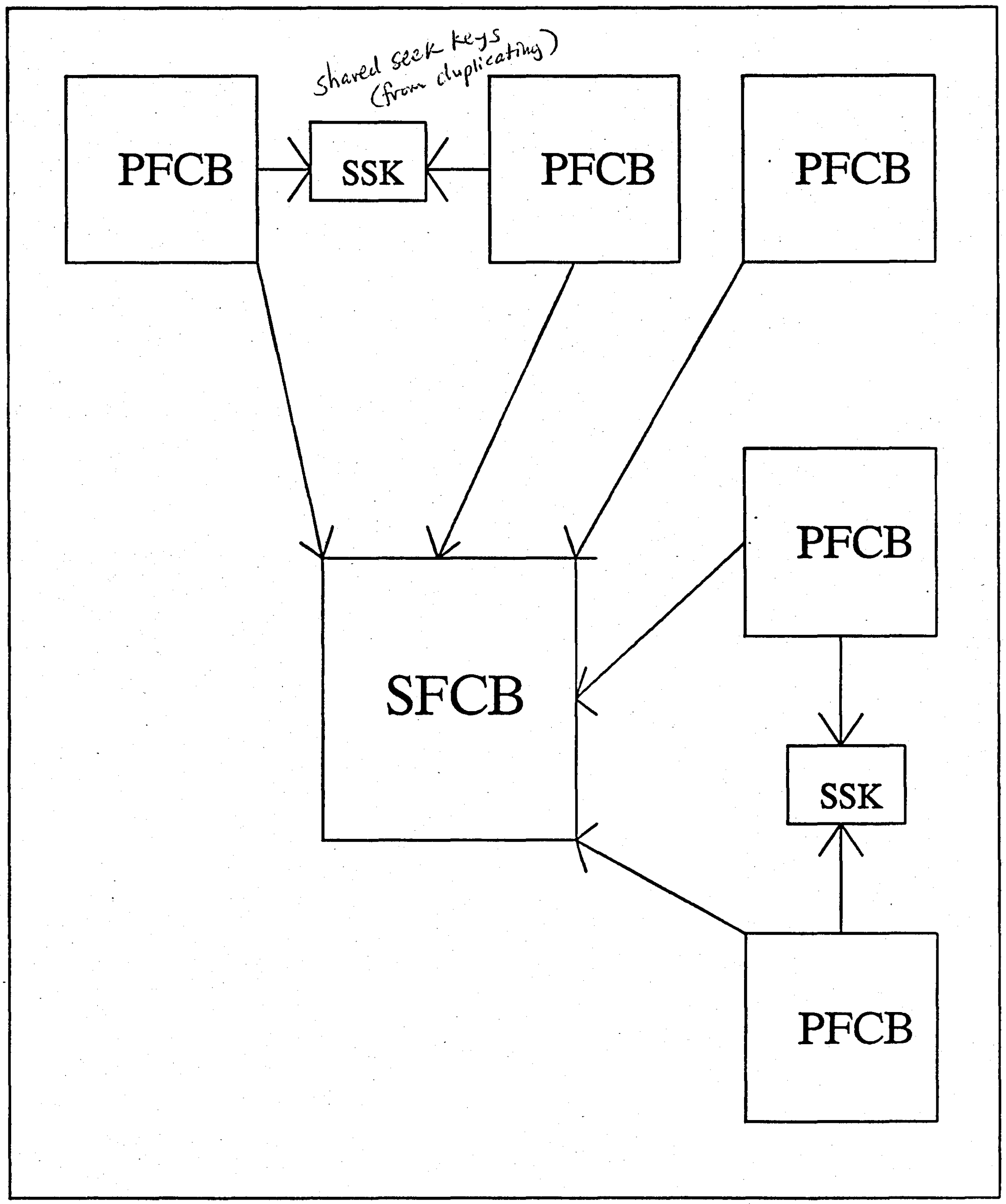
Header Cache

Shared Seek Key

force move: move mode will always move it.

no way of creating a stream for cowriters, must use redefine, and open it for cowriters.

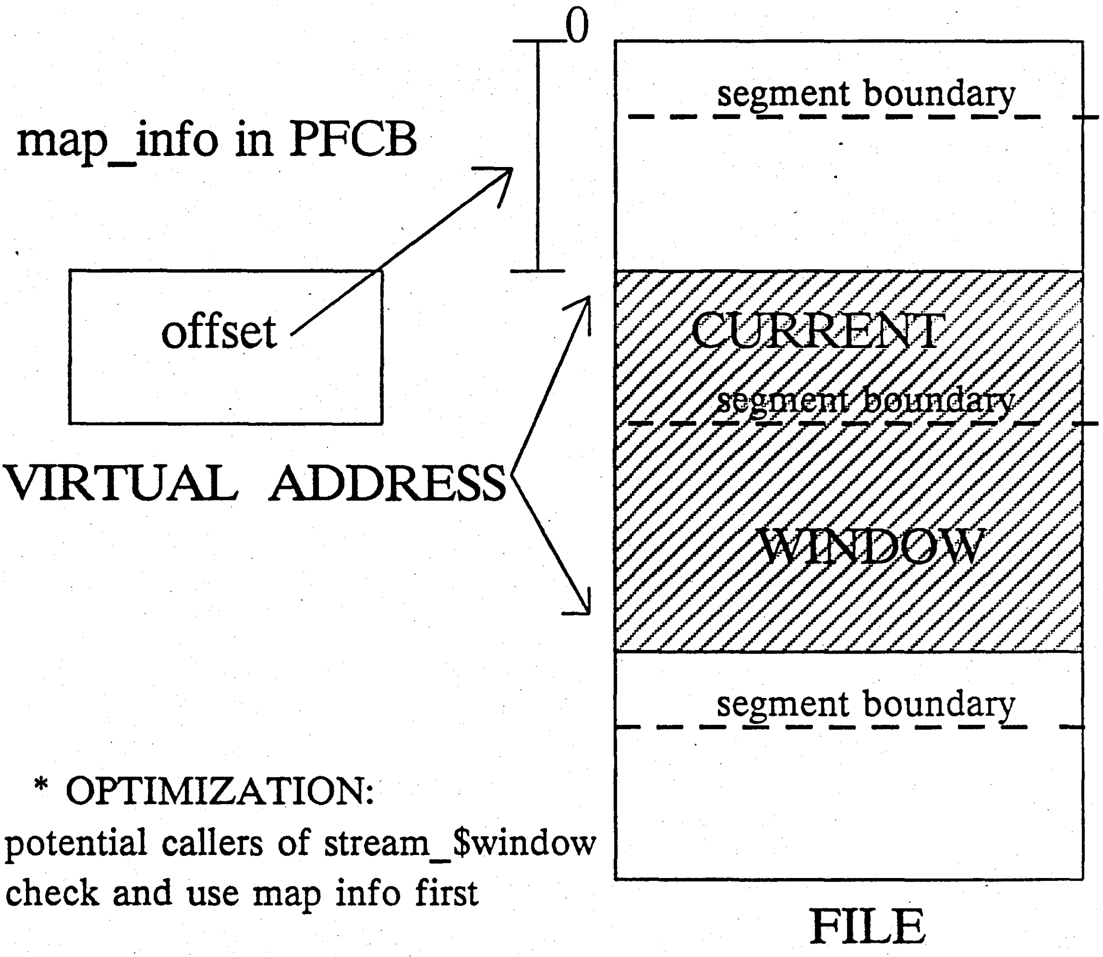
ONE TO MANY RELATIONSHIP



WINDOWING

- The d_file managers do "I/O" by mapping files
- ~~16~~ 9 1/2 MB may be too small to map a whole file
- So, we move a window over the file

VA := stream_window(PFCB, offset, lenth)



* OPTIMIZATION:
potential callers of stream_$window
check and use map info first

if it's already in the window that has been mapped read it, if it isn't move the window then get it.

Data Organization

- Byte Stream
  - * UNDEF : D_FILE2
  - * UASC: D_FILE3
- File (except header) is "pure data"
- Seek key is 4–byte file offset
- No "record" seek
- GETREC/GETBUF
  - * UNDEF
    Return the number of bytes requested, up to EOF
  - * UASC

    GETREC: return # of bytes requested, up to EOF/newline. Say how many bytes would be returned if the buffer were big enough.
    GETBUF: same as UNDEF

DATA ORGANIZATION

- Counted Records : (REC = d_file1)

  * 4 byte count followed by data
  * The count (hence data) always word aligned
  * 8 byte seek key

So you can go
back or forwards:
allows record seeks.

Record Offset	Byte Offset in file

- 2 Subtypes :

  * V : Variable Length
  * F2 : Fixed Length

    allows record seeks

    if set by Redefine, causes error on Putrec if length is wrong

Data Operation (Cont'd)

- Creation

    D-file3

    * STREAM_$CREATE makes UASC/ASCII

    D-file1 or 2

    * STREAM_$CREATE_BINARY makes REC/binary

    * All others must be made by redefine.

Locking & Concurrency

- Files locked only once per node

- SFCB reflects actual concurrent use on the node

- Special lock call (FILE_$LOCK_STREAM) used to support the following sequence:

  * Process 1 — open F

  * Process 2 — open F

  * Process 1 — close F

Locking & Concurrency (Cont'd)

- If both openers and file header agree on concurrent access (including at least one writer) then USE_COUNT in SFCB control access

- SFCB is locked on each read/write whenever file and opener allow concurrency

  * Lock is done by bitset & periodic retry

  * Timeout yields "unable to obtain needed resources"

  * ULKOB also releases streams lock, and invalidates SFCB. Subsequent operation gets "internal fatal error—table verify failed".

Other Managers

- NULL_DEV
  - * EOF on read, bit bucket on write
- DUMB_TERMINAL
  - * READ/WRITE SIO lines
  - * Disk object used to determine type and line number

SIO 1,2,3
SIO Ø (keyboard, tpad, mou

- VIR_TERMINAL

used by DM only,
None of these calls are released.

  - * Display manager input/ transcript pads
- DM_EDIT_PAD
  - * Allows only subset of pad operations and close
- MBX_FILE
  - * Interface to MBX manager for clients

Other Managers (Cont'd)

- PIPE_FILE
  - * UNIX pipes
- DIRECTORY
  - * UNIX format directory reader
- MAGTAPE
  - * STREAM level interface to MAGTAPE support
- CASE_HM history manager ~~Com~~
  - * CASE (DSEE) history manager reader

interleaved delta file with compressed leading spaces,

- All but NULL_DEV, CASE_HM use PFCB variant
- Only D_FILE, transcript pads, use SFCB

HPC uses SIO line as an interrupt source for sampling rate (uses it as a clock)

PROTECTION

Identifying and Authenticating Users

Subject ID (SID)

Registry

Access Control Lists

Protected Subsystems

Locksmith

Identifying Users

Subject ID (SID)

who is accessing the object:

person
project
organization
protected subsystem

PPO

- abbreviation for :
  person, project, organization
- a user
- if the subsystem is important : PPOS

Representation :

- each component of the SID (PPOS) is a UID

Authenticating Users

Establishing the user's identity and authorization to use the system

- a. k. a. "login"

Network Registry

- database of text string PPO to UID translations
- database of accounts

  subset of PPO combinations that can log in

  password

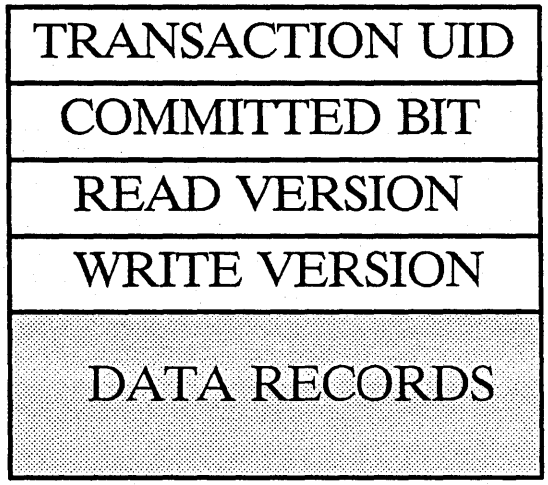
  home directory

Local Registry

- one per node (use when network down)
- last 10users 25 days ~~10~~ users to log in on that node
- guarantees login on your own node

Registry Algorithms

Registry file format (PPO and ACCT)



TRANSACTION UID

COMMITTED BIT

READ VERSION

WRITE VERSION

DATA RECORDS

Atomic Transaction

- all or nothing
- roll forward / roll back

Read Algorithm

- find one, read it

Update Algorithm basics

- make change to one copy
  (clear committed bit)
- “commit” it
- propogate changes to all the rest

Update and Recovery

Update

- lock *all* resigtry copies for RIW

  login can still happen
- pick one to update
- clear the comitted bit (force write)
- generate new transaction UID (time stamp)
- make changes; force write
- set committed; force write
- propogate changes to all copies

Crash Recovery

- find the latest committed copy

  make sure the clocks are in sync!
- overwrite all the rest with it

  rolls foward if changes finished

  rolls backward if changes unfinished

  takes advantage of the replication

  no separate before / after images
- done before each update

  no work (just checking) if no crash

Propogation: same as crash recovery

REGISTRY

- A network–wide, distributed, replicated database

- Contains people's names, projects, organizations (PPO)

- Contains accounts: subset of all PPO's that are authorized to log in (ACCT)

  * Password

  * Home directory

- Why Replicated?

  * Availability in face of failures

  * PARTIAL FAILURE

    A fact of life for distributed systems

REGISTRY LOCATOR

/REGISTRY/REGISTRY

3 ENTRIES
//node1/registry/rgy_site
//node2/registry/alt_site
//node3/registry/alt_site

1st one in file

The LOCATOR file is a list of locations of a distributed object.

SEARCH FOR ONE !

PARTIAL INFORMATION IS A FACT OF LIFE IN A DISTRIBUTED SYSTEM

COMMITTED	LOCK
TRANS UID	

COMMITTED	LOCK
TRANS UID	

COMMITTED	LOCK
TRANS UID	

Canned accounts
use Canned UIDS.

locksmith doesn't do
any ACL checking at all.

NORMAL CASE

COMMITTED YES	LOCK NO
TRANS UID 11:00 AM	

COMMITTED YES	LOCK NO
TRANS UID 11:00 AM	

COMMITTED YES	LOCK NO
TRANS UID 11:00 AM	

START UPDATE

COMMITTED NO	LOCK RIW
TRANS UID 2:00 PM	

COMMITTED YES	LOCK RIW
TRANS UID 11:00 AM	

COMMITTED YES	LOCK RIW
TRANS UID 11:00 AM	

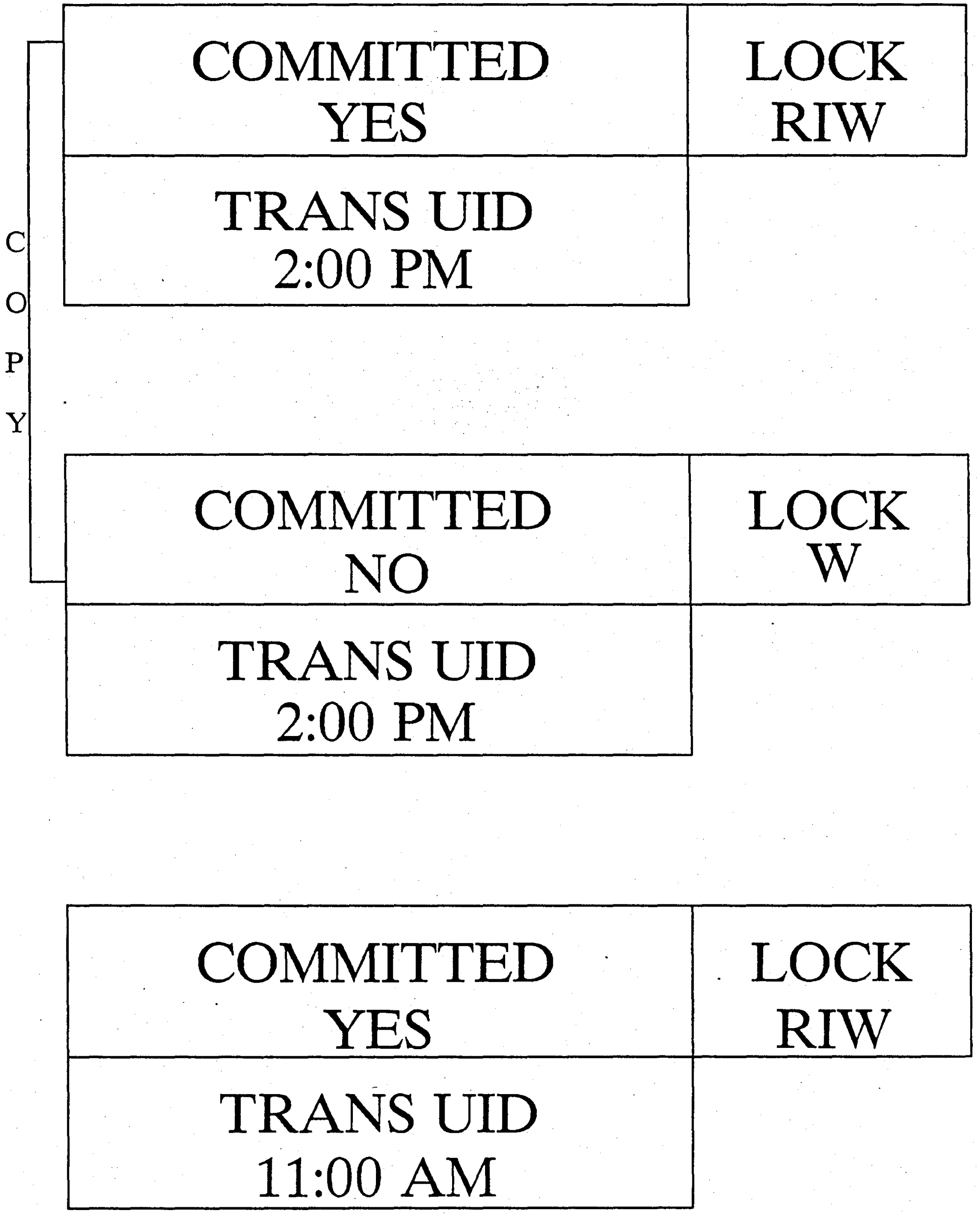
read intend to write

COMMIT UPDATE

COMMITTED YES	LOCK W
TRANS UID 2:00 PM	

COMMITTED YES	LOCK RIW
TRANS UID 11:00 AM	

COMMITTED YES	LOCK RIW
TRANS UID 11:00 AM	


COMMITTED YES
LOCK RIW
TRANS UID 2:00 PM
C O P Y
COMMITTED NO
LOCK W
TRANS UID 2:00 PM
COMMITTED YES
LOCK RIW
TRANS UID 11:00 AM

1 PROGATION DONE

COMMITTED YES	LOCK RIW
TRANS UID 2:00 PM	

COMMITTED YES	LOCK RIW
TRANS UID 2:00 PM	

COMMITTED YES	LOCK RIW
TRANS UID 11:00 AM	

ALL DONE

COMMITTED YES	LOCK NO
TRANS UID 2:00 PM	

COMMITTED YES	LOCK NO
TRANS UID 2:00 PM	

COMMITTED YES	LOCK NO
TRANS UID 2:00 PM	

ACLs

Basic: list of (SID, rights) entries

Rights

–files:	dwrx
–directories:	dcalr
–all:	pgn

Initial ACLs

stored in directory

ACL given to newly created files and directories

inherited by new directory

ACL Format

Version
Type (file, dir)
Default Node
Number of Entries
Subsystem Manager
Subsystem Data
ACL Entries

Entry format: PPOSNER

PPO: person, project, organization UIDs
S: subsystem UID (not currently used)
N: node to which rights apply
E: expiration date (not currently used)
R: rights bits (32)

All access checking is done in the nucleus by the ACL-manager.

Protected Subsystems

A way to restrict access to certain objects to certain programs

The protected subsystem has a UID

The “certain objects”:

- –have subsystem UID in the “subsystem data” field of their ACL
- –called “protected” or “sealed” data

The “certain progams”:

- –have subsystem UID in the “sub-system manager” field of their ACL
- –called “subsystem manager”

Subsystem managers

- –have complete control over access
- –have all rights to protected data

protective subsystems are Subordinate to ACL's. If you have access via ACL's, You have access to the subsystem.

Protected Subsystems II

Commands:

CRSUBS

– *create a new protected subsystem*

ENSUBS

– *enter a subsystem at shell level*

– *examine, debug protected data and managers*

– *make new managers, protect data*

SUBS

–make new manager, protected data

–increase priveledge

– print subsystem status of an object

name of owning subsystem

name of subsystem that the program manages

XSUBS

–execute a shell program as a protected subsystem manager

Protected Subsystems III

Protected subsystem creation

- copy shell into /sys/subsys/*name*
- generate subsystem UID

  it's the UID of the shell!
- set subsystem manager field of shell
- now have a shell to use to protect data, make new managers

Protected subsystem invocation

- pgm_$invoke sees its a manager
- creates new process for it

Protected Subsystems IV
(Rights Checking)

Outside
- *when not running in a manager*
- *in a manager, but without increased priviledge*
- get ordinary *"base"* rights from ACL

Inside
- *in manager, with increased priviledge*
- get all rights

Increased priviledge
- *"UP", "DOWN"* calls
- why ?

  prevents trickery

  pass subsystem data where manager expected ordinary object

Protected Subsystems V
(and miscellaneous)

"Login" protected subsystem

- ships with system
- has one extra priviledge:
  it can set SID
- it promises to do so only after checking PPO, password in registry

Subsystem names

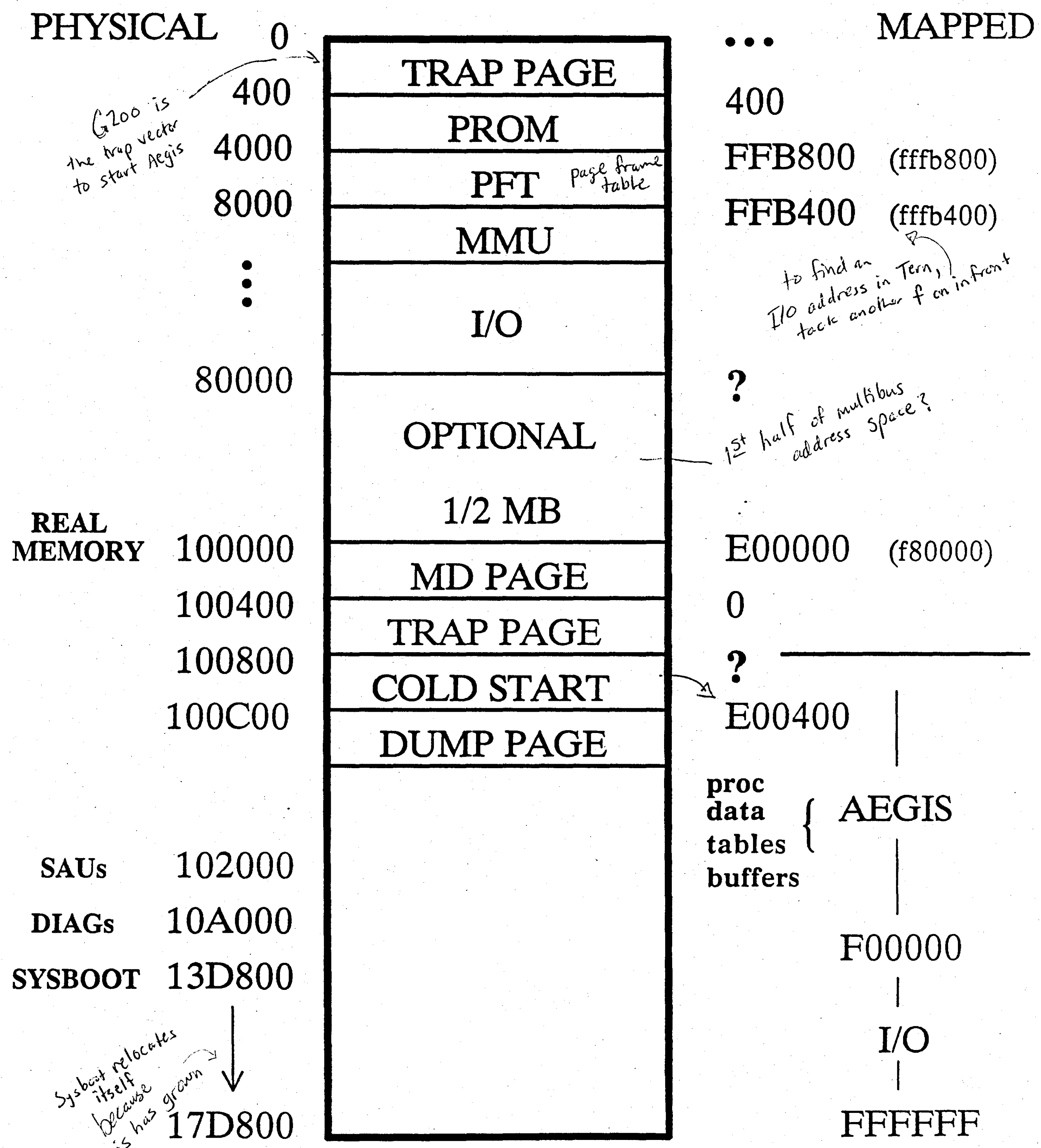
- look up subsystem UID in /sys/subsys
- find object whose ACL has that UID in subsystem manager field
- use its name
- if none *on that node* can't get name

Locksmith

- a project and a protected subsystem
- has all rights to EVERYTHING

How Aegis comes up.

ADDRESS SPACE



PHYSICAL			MAPPED	
	0	TRAP PAGE	...	
	400	PROM	400	
	4000	PFT	FFB800	(fffb800)
	8000	MMU	FFB400	(fffb400)
	⋮	I/O		
	80000	OPTIONAL 1/2 MB	?	
REAL MEMORY	100000	MD PAGE	E00000	(f80000)
	100400	TRAP PAGE	0	
	100800	COLD START	?	
	100C00	DUMP PAGE	E00400	
SAUs	102000			
DIAGs	10A000			
SYSBOOT	13D800			
	17D800			

proc
data
tables
buffers
{ AEGIS

F00000

I/O

FFFFFF

G200 is the trap vector to start Aegis

page frame table

to find an I/O address in Term, tack another f on in front

1st half of multibus address space?

Sysboot relocates itself because Aegis has grown

Aegis mapped between f80000? and Os_proc_end label

PROM

runs with interrupts disabled - it polls all devices.

- 0 – 3FFF Physical ½ segment
- Major Pieces
  - * SYS INIT (SIOS, MMU, I/O) -find all the controllers
  - * Boot Logic
  - * Device Drivers

    DISPLAY

    SIOS

    DISKS—WIN, FLP, SM

    RING (ETHER?)

    LEDS
  - * Diagnostics
  - * MD CMDS & PARSING
- Runs Disabled – does not service interrupts (reason for double carriage return)
- Runs Either Physical or Mapped, All I/O Mapped

all PROM I/O is done in mapped mode.

PROM (Cont'd)

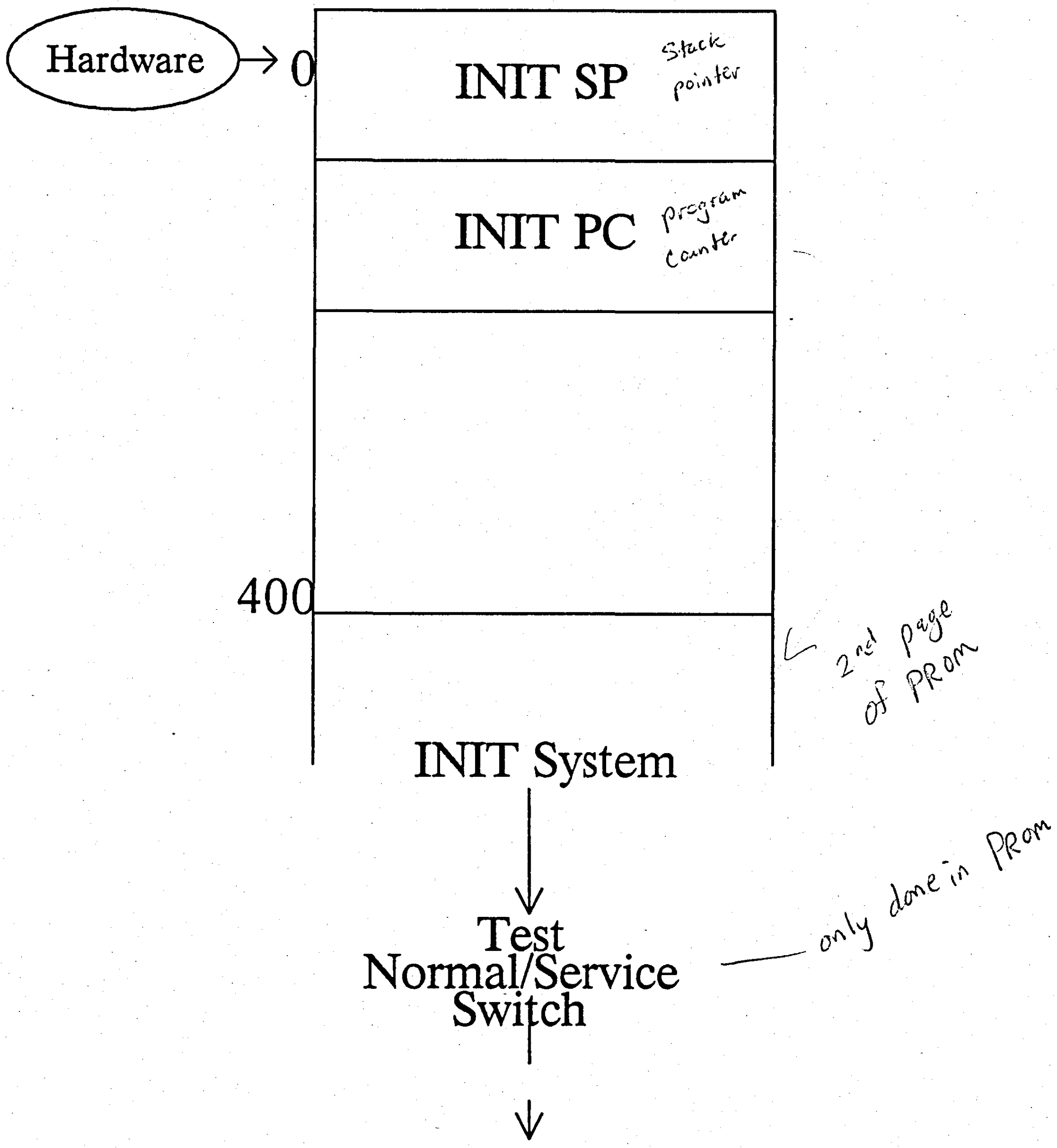
- Machine ID at 100 physical location

0	Old DN400, 420, 600
1	DN420, 600
2	DN300
3	DSP80
4	DNx60
5	DN550

SAu

How the PROM knows what SAu to look at.

Power–On
(Reset Switch)



Hardware → 0

INIT SP

Stack pointer

INIT PC

program Counter

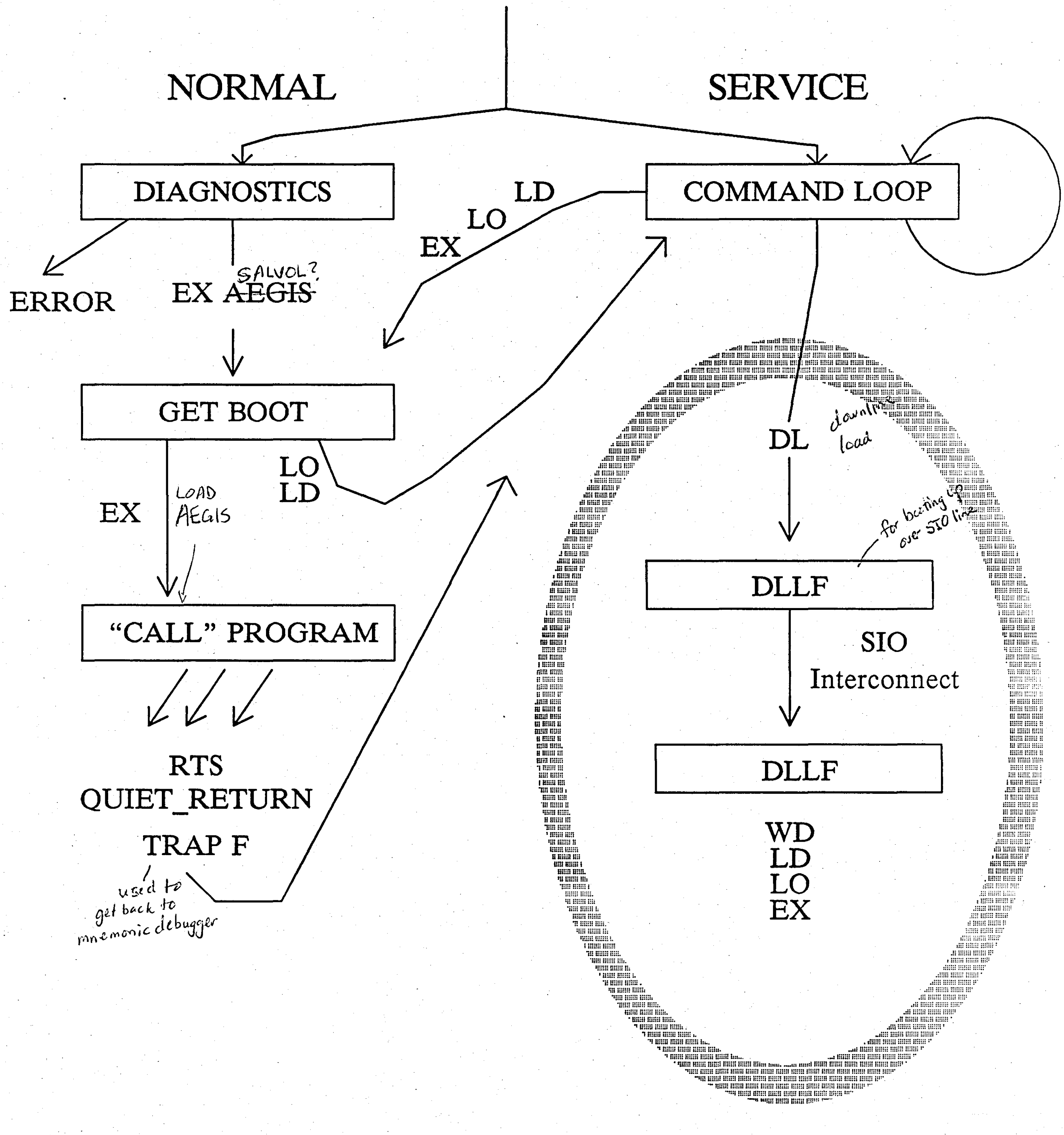
400

2nd page of PRom

INIT System

Test Normal/Service Switch

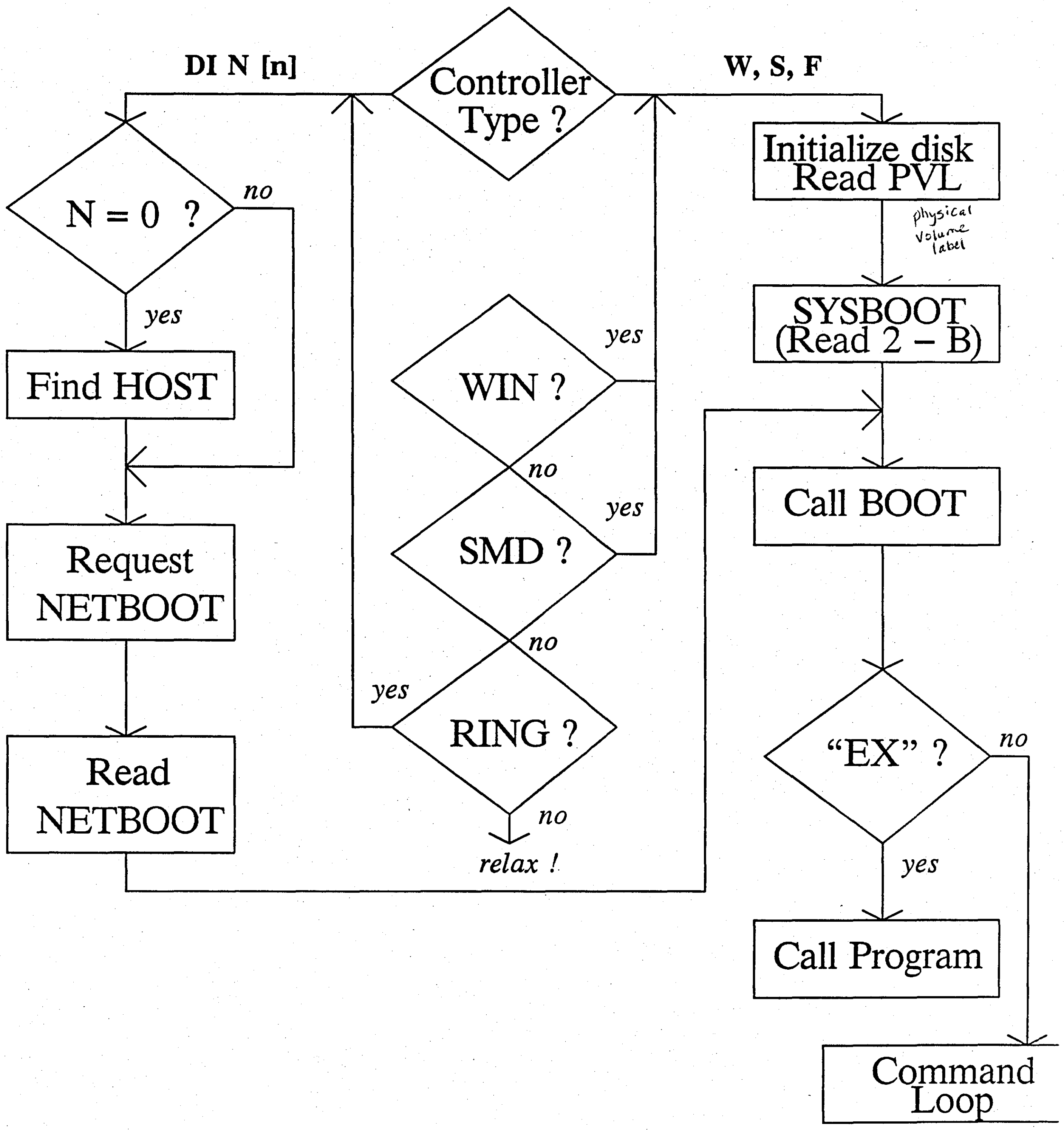
only done in PRom



SIO Ø is input only
Other side of SIO Ø is used for clock. It can't do xon/xoff with a keyboard.

Prom has autobaud capability, but Aegis doesn't.

GETTING A BOOT



physical volume label

Prom knows how to build and broadcast a packet

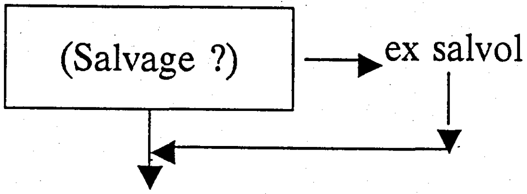
SYSBOOT and NETBOOT

- Parse commands, pick driver

SYSBOOT

READ:

- PV Label
- LV Label



- Root Directory (/)
  - * Find /SAUn
- VTOCE for /SAUn
- /SAUn directory
  - * Find program
- VTOCE for program
- Program

  (Right machine_ID)
- Done

  (Return "GO" flag to MD)

LD

NETBOOT

my place or yours ?

- Chat with NETMAN
- Read file
- Get UIDs :
  - * paging file
  - * /
  - * //
- DONE !

Get UIDs

- Resolve “//”

- Resolve “/”

- Resolve “ ‘NODE_DATA.nnn

300 blocks minimum because OS paging file is in here. Good candidate to get rid of.

  * UNLOCK

  * CREATE

  * SET DEFAULT ACLs

- Resolve “ ‘NODE_DATA.nnn/
  OS_PAGING_FILE”

288 blocks

  *CREATE or EXTEND

If you ever replace Aegis, you should replace net~~man~~boot too because they both know size of OS paging' If they don't agree, boot fails.

- Copy ‘NODE_DATA/SHELL

- Copy /SYS/SYSDEV –> 'NODE_DATA.nnn/DEV

for device streams

- Copy /SYS/DM/STARTUP_ TEMPLATES
  - * Add KBD 2 if DN300
  - * Use /SYS/SPM/STARTUP_TEMPLATES if server (DSP80)
- REPLY WITH UIDs of
  - * //
  - * /
  - * 'NODE_DATA/OS_PAGING_FILE
- PROBLEM? Run:
  - * NETMAN in window
  - * NETMAN –DB

~~Netstat -sens~~
Netstat -since

Run SPM in window
when having trouble
with CRP.

Problem:
run Netman in window
with the -db switch to
see what's going on.

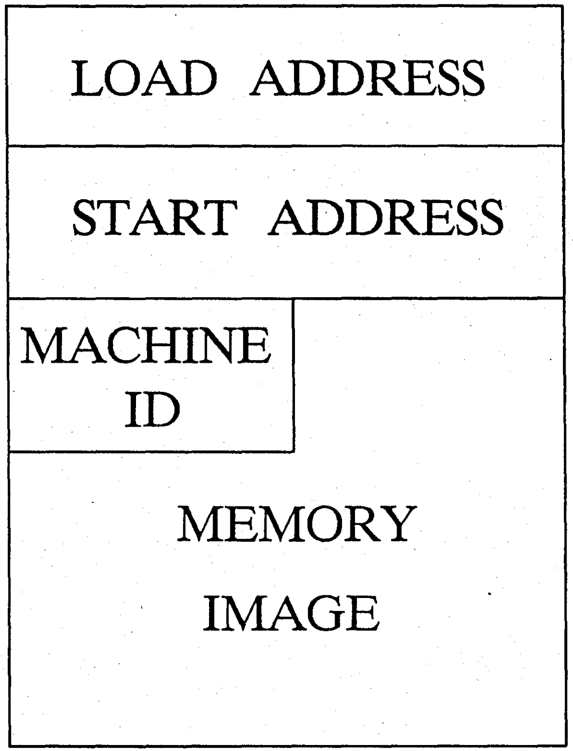
-db is an option on almost
everything
(try -debug also)

RFC FORMAT

All programs that run at sysboot. program is an absolute image.

write test programs, put in SAU and run them from the MD.

Aegis has this format; dump file, look at first 4 bytes to see the load address.



LOAD ADDRESS

START ADDRESS

MACHINE ID

MEMORY IMAGE

Sysboot is machine independent (it runs in mapped mode) Aegis and all RFC files are machine dependent.

RFC – Run File Converter

"Calling" Sequence :

controller type

MUNCH (ctype, unit, lv_num, flags, os_data)

flags = set of (new_prom, dtty, normal)

os_data = Paging file UID
Root directory UID
Node UID (host)
His node ID

AEGIS Initialization Sequence

* Save ARGS for PROM

all done by COLD

* Copy TRAP PG to 100400

* Initialize MMU 1:1

* Initialize OS TRAP /FAULT Vectors

* Turn On ECCC/Parity

writes all of memory to clean out any ECC

* Call OS_$INIT to Do Hard Stuff

AEGIS Initialization Sequence

OS_INIT

- * Initialize I/O Devices
- * Initialize Managers—Clock, UID PROC1, SMD, DTTY, EC2, DBUF

  disk buffers

  dumb terminal
- * Mount BOOT VOL & Verify Calendar
- * Initialize VM MGRS—MST, AST, FILE

  virtual memory
- * Fix Up Address Space (Activate Segs, Wire, Whole Cloth)

  → PCBs, wires down segments.

  Whole cloth pages are pages that have no backing store.
- * Create OS Processes—Clock, Term Helper, Purifer, Net Servers

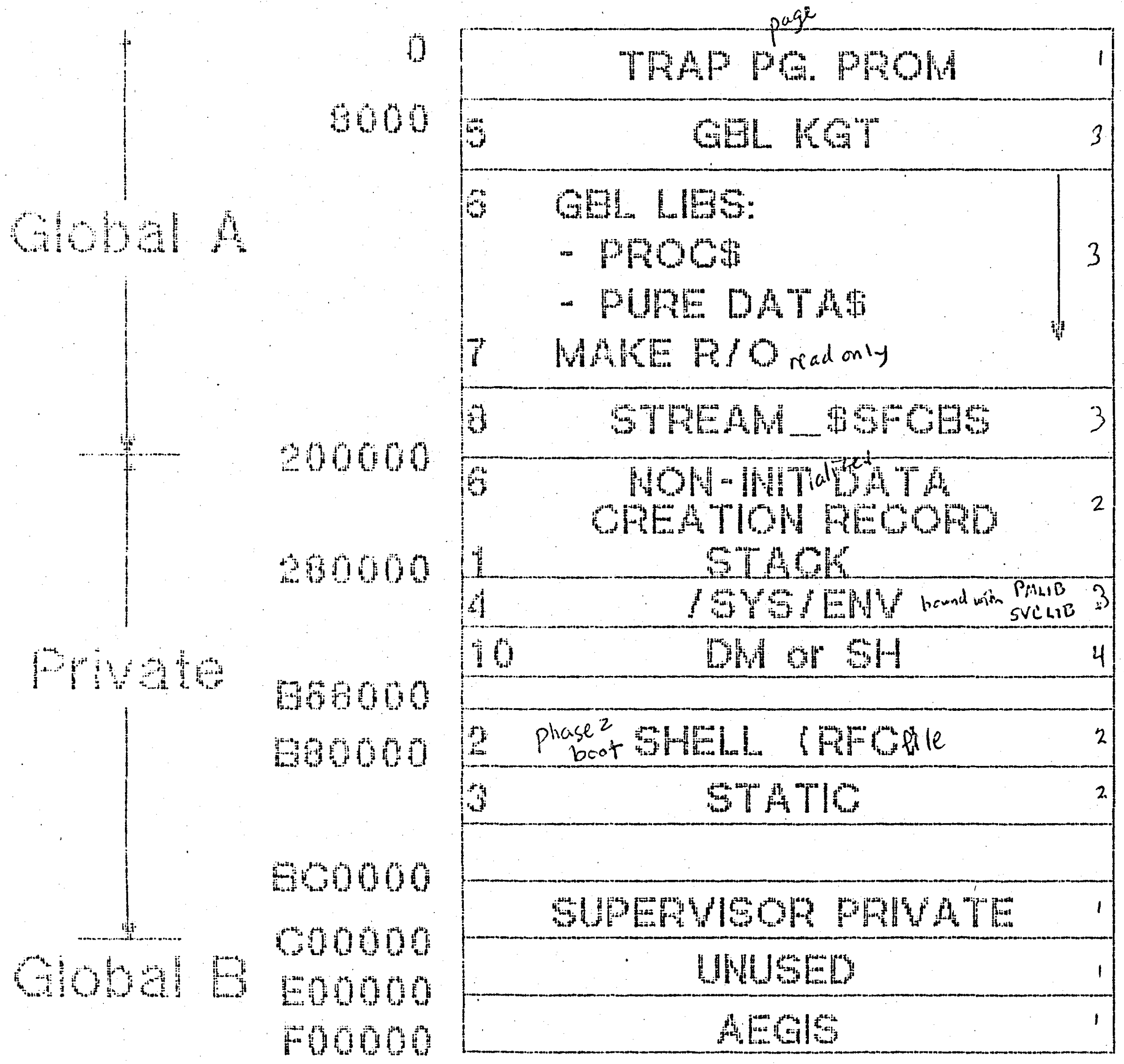
  Level 1 processes

  /BSCom/LAS.BS

  tmpfile 0 uid1 wired OS procedure
  " 0 uid1 pageable OS data
  " 0 uid1 whole cloth

  3 things mapped to same disk file
- * Become Process 1

  DM, or SPM
- * Initialize PROC2 MGR
- * PROC2_$ STARTUP



Global A	0		TRAP *page* PG. PROM	1	
	8000	5	GBL KGT	3	1, 2: PROC2 __ $INIT 3: SHELL
		6 7	GBL LIBS: - PROC$ - PURE DATA$ MAKE R/O *read only*	3	4: PM __$LOAD ENV 5,6,7: PM$INIT __FIRST 8: STREAM __$PROC __ INIT PM __$INIT: $INIT LIBS *CONNECT STREAMS
		8	STREAM__$SFCBS	3	10: LOAD DM or
Private	200000	6 1	NON-INIT*ialized* DATA CREATION RECORD STACK	2	SH or SPM NODE __DATA/STACK
	280000	4	/SYS/ENV *bound with PMLIB SVCLIB*	3	
		10	DM or SH	4	
	B68000				
	B80000	2	*phase 2 boot* SHELL (RFC*file*	2	NODE __DATA/SHELL
		3	STATIC	2	NODE __DATA/STATIC
	BC0000				
Global B	C00000		SUPERVISOR PRIVATE	1	
	E00000		UNUSED	1	• [illegible] DIRS
	F00000		AEGIS	1	

Bootshell

- RFC'ed PGM

- Mostly vestigial resting point now

- Commands

    * Version of MD

    * VM, FS commands

        map files unmap files

        WD, LD, MAP, UMA

    * boot shell commands

        /BSCOM

        LAS, CPBOOT, DLT

- "GO" "DM" "SH" "SPM" ->
  loads ENV & passes flag

- Runs as USER.NONE.NONE
  except for DM, GO, SH, SPM

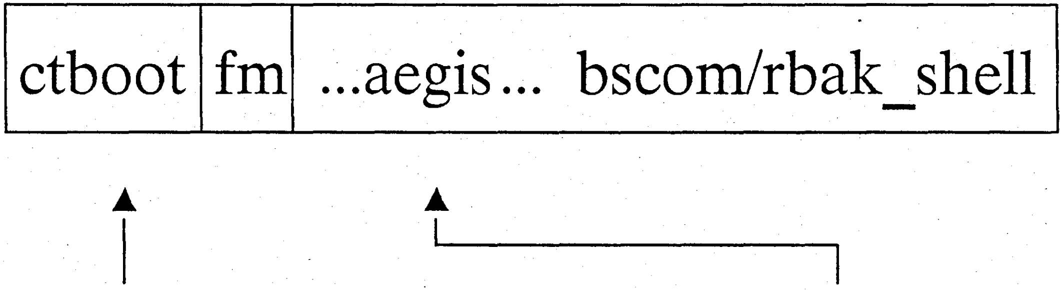
TAPE BOOT

Why ? DN550 has no floppy, so how do you load software on a new disk ?

The NEW Invol creates /sys/node_data

From PROM > DI C ex (any SAU)

Cartridge Tape :



ctboot | fm | ...aegis... bscom/rbak_shell

"CPBOOT /SYS –DEV CT" "WBAK –SYSBOOT'

AEGIS : "NO, LET'S NOT PAGE TO THE CARTRIDGE TAPE"

PROC2_$INIT: "IF BOOTED FROM TAPE, FIRST RBAK BSCOM/RBAK_SHELL – the RFC shell

RBAK_SHELL : LIKE SHELL, BUT RBAK FILE #1 BEFORE CONTINUING

THEN; "GO", LOGIN, INSTALL FROM TAPE

FILES REQUIRED DURING BOOT 01/18/85

REQUESTING AGENT	FILE	
PROM	/SYSBOOT (records 2-B on track 0)	
if tern:	/SAUn/WCS.UC	(microcode file)
	DCODE.UC	(instr. decode RAM contents)
	SPAD.UC	(scratchpad constants and temps)
	ULOAD	(program to load the above)
SYSBOOT	/SAUn/AEGIS	(AEGIS load file)
	/SAUn/SALVOL	(only if salvage required)
AEGIS	[os paging file]	(uncatalogued)
	//	(UIDs found and saved by NAME_$INIT)
	/	
	/COM	
	/SYS/NODE_DATA	
	/SYS/PEB_MICROCODE or PEB2_MICROCODE(1)	
	`NODE_DATA/SHELL(2)	(mapped by PROC2_$INIT)
SHELL	Can play games by moving any bitmap you want into here. → /SYS/APOLLO_LOGO(3)	
	* `NODE_DATA/STARTUP_SHELL(3)	boot shell example; put in LO filename to run a program (cmd file to override dflts)
	/SYS/ENV	(SHELL tells him what to run)
ENV	/LIB/?*	
	/SYS/DM/DM	"GO" command or normal boot -OR-
	/SYS/BOOT	"SH" or boot from SIO line -OR-
	/SYS/SPM/SPM	"SPM" or normal boot on server node
DM	* `NODE_DATA/DEV/SIO1	DM error message goes here when it crashes.
	/SYS/DM/FONTS	
	`NODE_DATA/STARTUP[.19L, .COLOR](3)	
	/SYS/BOOT	
BOOT	/REGISTRY/REGISTRY(4)	(+PPO,Account files pointed to)
	LOCAL_REGISTRY	
	LOCAL_SITE/?*	
	/COM/SH	

Notes

(1) PEB is disabled if microcode file not found.
(2) If booted from cartridge tape, the tape is first searched for BSCOM/RBAK_SHELL.
(3) Optional -- system will manage without it.
(4) If no registries are available, you can login only as USER.NONE.NONE.

STARTUP FILES

```
                                                         06/29/84
      "= = = = >"  =>  unconditionally executes
      "- - - - >"  =>  executes if it exists

  ---
 :
 :
 :  Netman copies /sys/dm/startup_templates (startup, startup.19l,
 :  startup.color to `node_data -------------------------------------+
 :  (If booting node is a DN300, only STARTUP.19L is copied,         :
 :  and a "kbd 2" command is tacked onto the end.)                   :
 :                                                                   :
 :                                                                   :
  ---                                                                :
                                                                     :
      AEGIS                                                          :
        :                                                            :
        :                                                            :
        V                                                            :
  `NODE_DATA/SHELL - - - - - - >  `NODE_DATA/STARTUP_SHELL *         :
        :                                                            :
        :                            (override of default starting   :
        :                             of dm, sh, or spm) load and execute   V
        V                                          whatever you want
   /SYS/DM/DM      = = = = = >  `NODE_DATA/STARTUP        here. (420 portrait)
        :                                   STARTUP.19L   (300, 320, 460, 550)
        :                                   STARTUP.COLOR (600, 660)
        :
        :                            (define dm windows, start netman,
        :                             mbx_helper, etc., kbd command)
        :
        :              (If `node_data/startup[.xxx] isn't found, the DM will look for
        :              /sys/dm/startup[.xxx], but this is undocumented & not shipped.)
        :
        :          = = = = = >  /SYS/DM/STD_KEYS[2]
        :
     (LOGIN)                         (default key definitions)
        :
        :          - - - - - >  USER_DATA/KEY_DEFS[2]
        :
        :                            (personal key defs from last login)
        :
        :          - - - - - >  `NODE_DATA/STARTUP_LOGIN [.19L, .COLOR]
        :               else
        :          - - - - - >  /SYS/DM/STARTUP_LOGIN [.19L, .COLOR]
        :
        :                            (per-login processes, first window,
        :                             by convention, points to -----------+
        :                                                                 :
        :          - - - - - >  USER_DATA/STARTUP_DM [.19L, .COLOR] <----+
        :
        :                            (personal key defs, bgc, etc.,
        :                             optionally points to ---------------+
        V                                                                 :
     /COM/SH       - - - - - >  USER_DATA/STARTUP.SH        <-------------+

                                     (check mail, netsvc, etc.)
```

CRASHES

NODE IS

HUNG SLOW IN MD (“>”)

USE

NETSTAT ← since

NPST -all PST -L1 -PA

LSYSERR

LOOK FOR

DISK / NETWORK ERRORS
SICK SIO ?
MEMORY PROBLEMS
NETWORK TRAFFIC
READY LIST MESSED UP (PST)
VTOC (SALVOL)

CRASHES

NODE IS

HUNG SLOW IN MD (“>”)

CHECK

LIGHTS ?
CURSOR ?
NETWORK ?
KEYBOARD ?
SERVICE MODE ?

CTL RETURN

∞ Loop
Network
Lost Interrupt
Ready List

if out of order, then delete PBu.lib ! GPIO is major culprit in screwing up ready list,

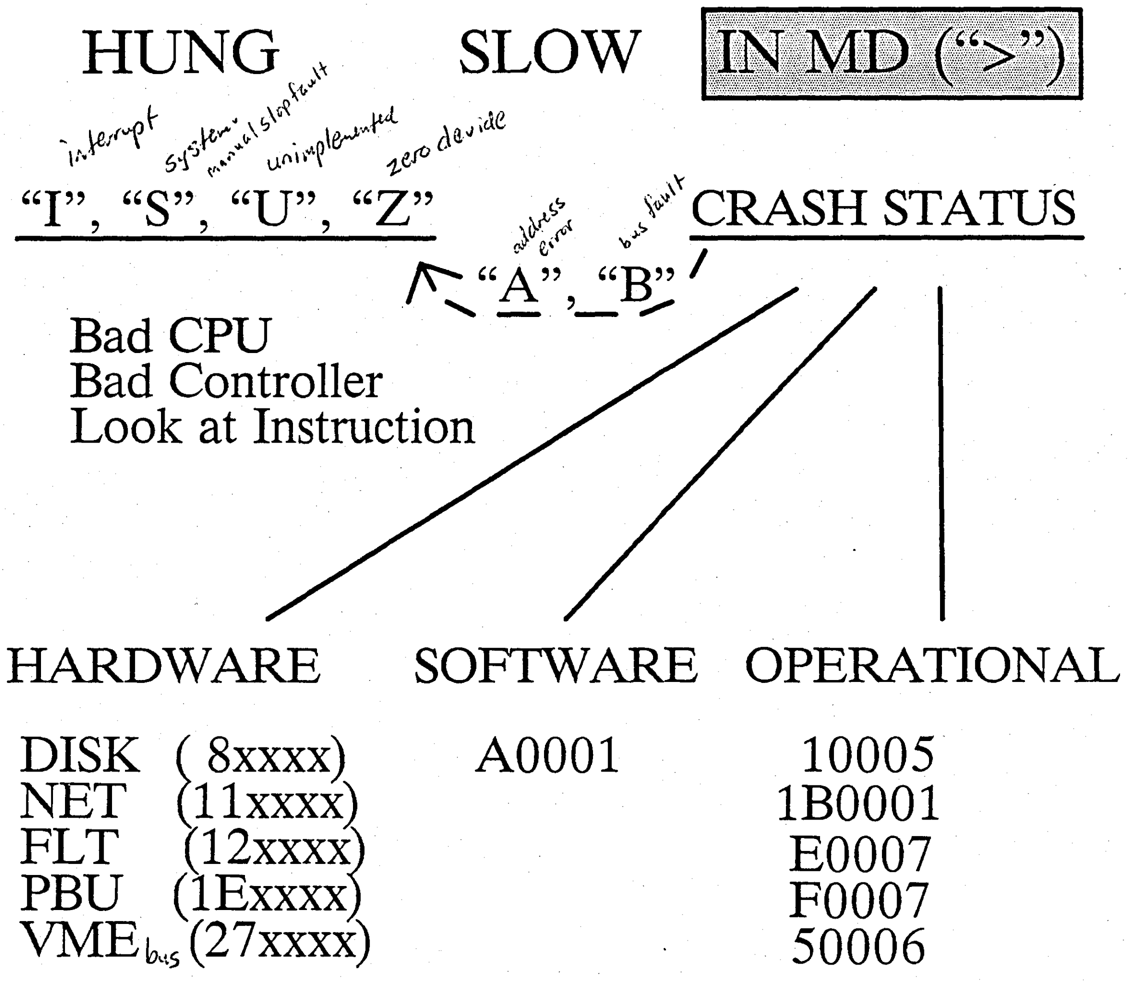
RESET

Double Bus Error
Disabled Loop
(e.g. MMU)
Bus Locked
(bad controller)
Sick CPU

LED lights

CRASHES

NODE IS



HUNG

SLOW

IN MD (">")

interrupt, system, manual stop fault, unimplemented, zero devide

"I", "S", "U", "Z"

address error, bus fault

"A", "B"

CRASH STATUS

Bad CPU
Bad Controller
Look at Instruction

HARDWARE

DISK (8xxxx)
NET (11xxxx)
FLT (12xxxx)
PBU (1Exxxx)
VME bus (27xxxx)

SOFTWARE

A0001

OPERATIONAL

10005
1B0001
E0007
F0007
50006

Cntrl/return

:

to start again:

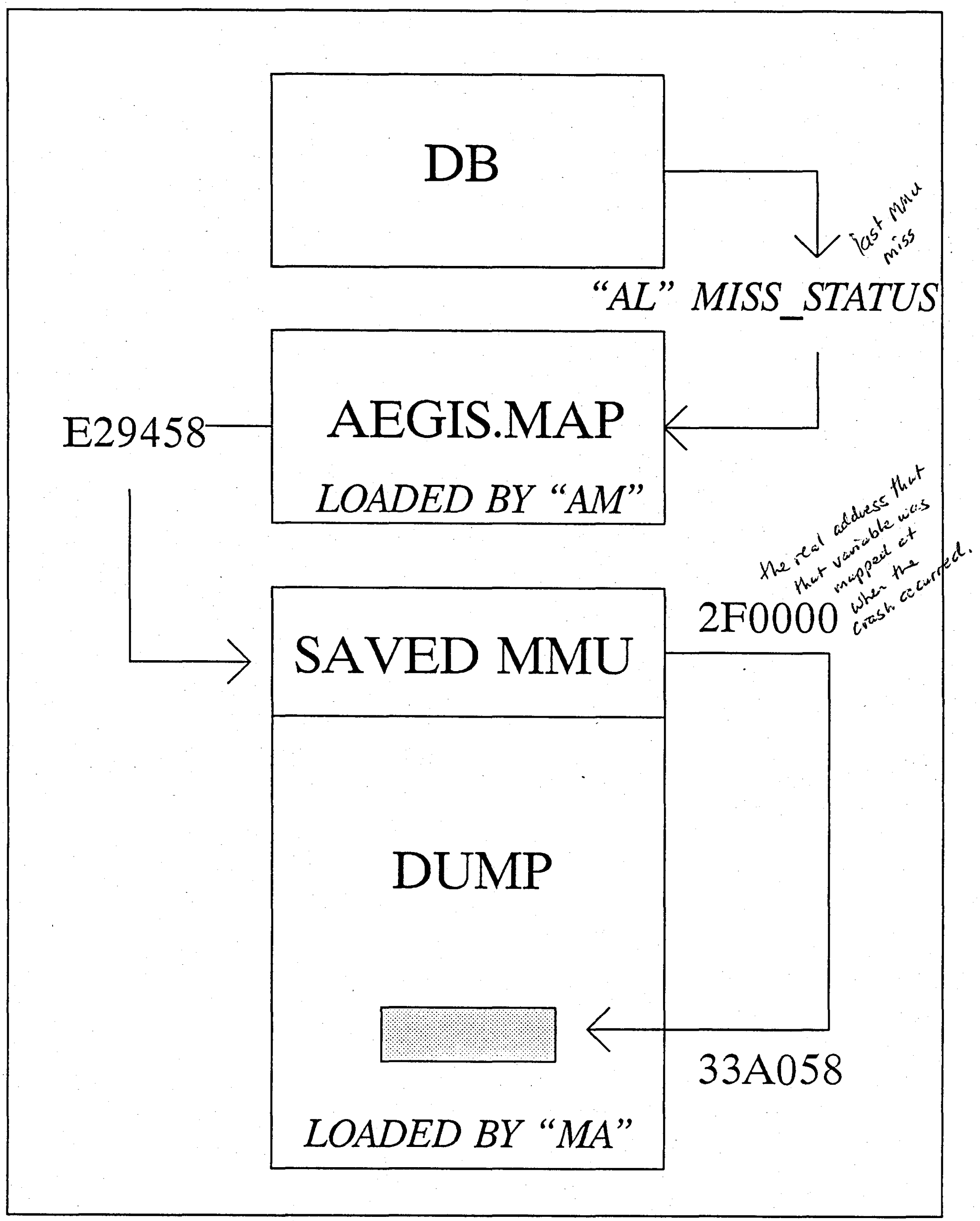
> G

<trap>

> G * + 2

(screen screwed up)

^F (refresh screen)



Sysboot doesn't need Aegis.map -- it's primarily there for people to look at it.

DB CRASH ANALYSIS

- State of the machine:

  status display registers

  ST, DR, DN460, DP, RL, GD,

  timestamp

  TS, MST <asid>, VM

- Error History

  DS, MR, LE

- Disk Status

  DCT, DVT, PVL, LVL

- AEGIS Variables

  state of VME bus

  MISS_STATUS, VME_$SAVE

  NETWORK_$DISKLESS whether or not machine was diskless

  TIME_$CLOCKH

  PARITY_INFO,

  DCTE.BLK_HDR_PTR^

  CPU_B_PBU_SWITCH

SPM

~~SMP~~/CRP

SPM is always the one that creates remote processes.

- Server Process Manager
  - * Services requests to create processes on this node
  - * Supports CP, CPO, CPS requests
  - * Replaces DM on DSP–type nodes
  - * Requires MBX_HELPER
- Create Remote Process
  - * Makes requests of remote SPMs
  - * Supports CP, CPO, CPS requests
  - * Provides streams for CP requests "window on remote process"
  - * Requires MBX_HELPER

The only process
that can call the
login routine and
change SID's.

SPM Details

- If Process 1: (DSP, DM Replacement)
  - * INIT process name directory open STD streams

'node_data/proc_dir

- Set name to "SERVER_PROCESS_MANAGER"
- Set WD, ND, to "/"
- Process arguments
  - * HIGH, LOW = priority of spawned processes
  - * MBX = mailbox to open on
  - * NLOGIN = processes get SID of SPM
- Process 'NODE_DATA/STARTUP.SPM
- Create mailbox ('NODE_DATA/SPM_MBX)

SPM Details (Cont'd)

- Wait for things to happen

  * Invocation requests on mailbox

  * MBX_HELPER problems (restart)

  * Shutdown (if PROCESS_1)

CRP Details

- Processes Options (–DB)
- If CP, Creates Remote Mailbox
  - * 'NODE_DATA/CRP_MBX.n
- Opens Channel on Remote
  - * SPM_MBX
- Issues Invocation Request
- Waits
  - * SPM_MBX for Response
  - * CRP_MBX.N for Opens (CP)
- Closes SPM_MBX Channel
- Waits and Services Inputs
  - * STDIN –> CRP_MBX
  - * CRP_MBX –> STDOUT
- Honors Certain Pad Function Calls

CRP Details (Cont'd)

- Faults

  * QUIT, INTERRUPT

    forwarded only

  * ALL OTHERS

    Stop fault needed to stop CRP

    forwarded & signaled

- Invocation Flavors

  * CP

    opens streams to MBX_UID passed

    invokes SPMLOGIN passing command line

  * CPO and CPS

    opens streams to /DEV/NULL

    invokes SPMSID passing command line

- Processes are Marked as “Servers”
- SPMLOGIN & SPMSID must be stamped in LOGIN subsystem
- I/O Anomalies for CP’d Processes
    - * Prompts
    - * Type–ahead Forwarded Immediately
    - * No Graphics or Pad Calls Supported
- ACLS
    - * on SPM node

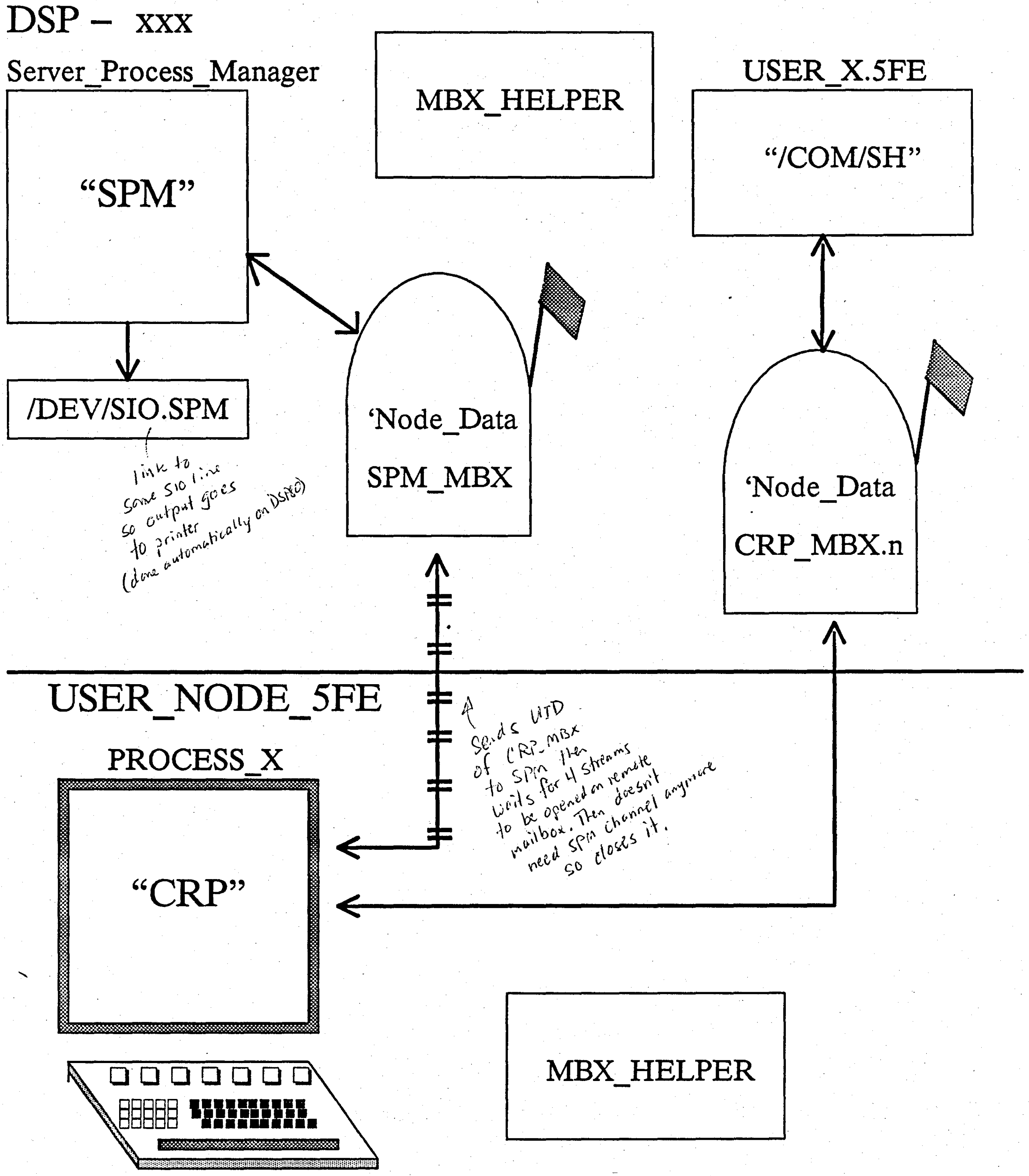
      create read link

      ’NODE_DATA = CRL for directories and DWRX for files
    - * on client node

      ’NODE_DATA = R

- SHUTDOWN Event
  (SPM = PROCESS_1)
  - * Kills All Processes
  - * Closes SPM Mailbox
  - * Calls OS_$SHUTDOWN
- Can Run in Window, Logs Events

CRP -CP



link to
same SIO line
so output goes
to printer
(done automatically on DSPEO)

Sends UID
of CRP_MBX
to SPM then
waits for 4 streams
to be opened on remote
mailbox. Then doesn't
need SPM channel anymore
so closes it.

EVEN MORE SPM DETAILS

SPM REQUEST :

VERSION NUMBER
OPERATION [CP, CPO, CPS]
MBX_UID (for CP)
LOGIN INFO (for CP)
COMMAND LINE for
INVOKED PROCESS

SPM RESPONSE :

VERSION NUMBER
STATUS
PROCESS_UID
ERROR_NAME

SIOMONIT

– Supports successive logins over SIO lines, independent of local node use.

  * Invokes SIO line watchers

    SIOLOGIN

  * Gets instructions from a file

  * Logs its activities

  * Should run as a server

SIOLOGIN

- Watches a single SIO line
- Runs the SHELL FILE

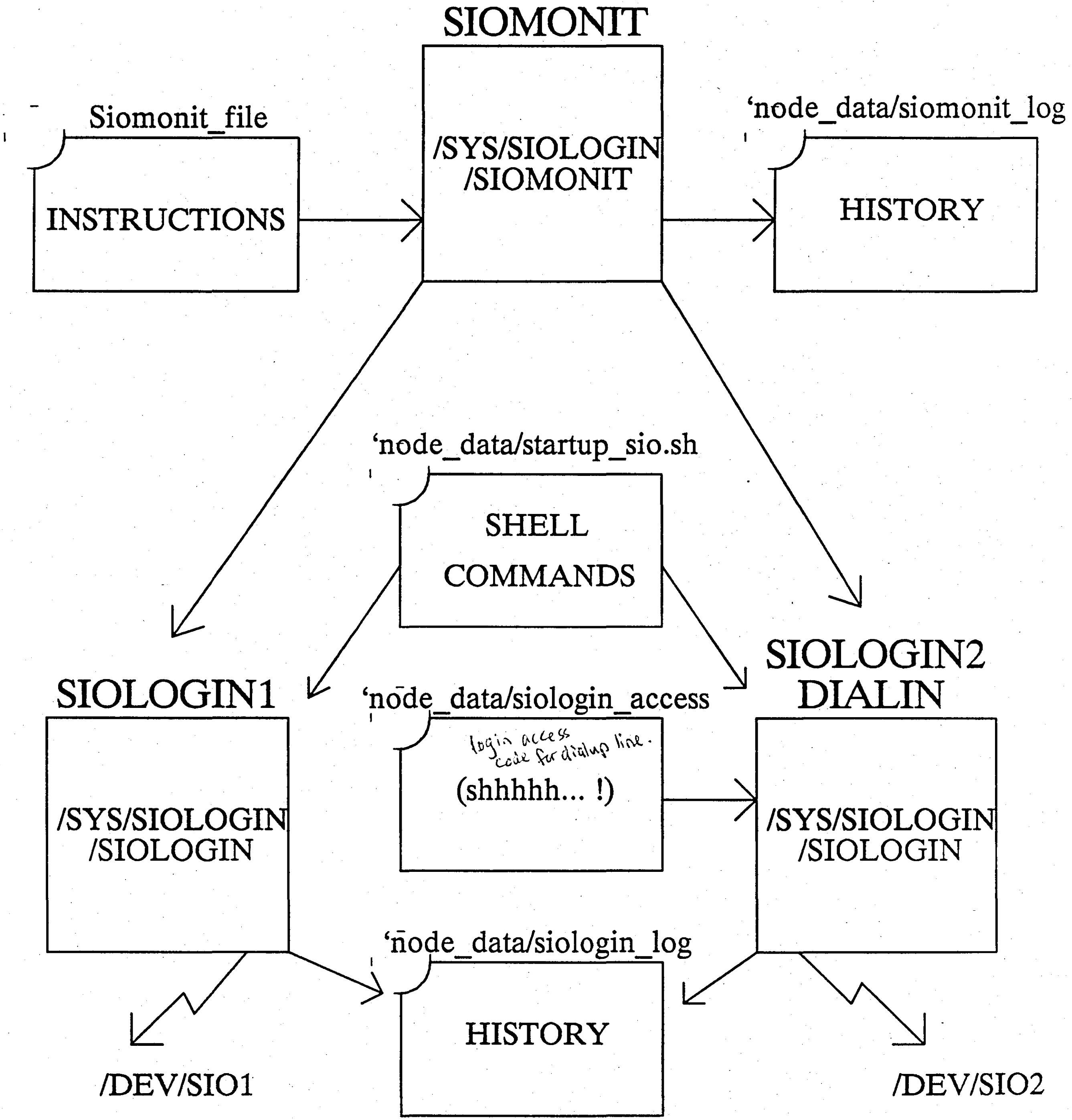
  'NODE_DATA/STARTUP_SIO.SH
- Performs login sequence
- Invokes specified program

DTR dropped on hangup

DTR never dropped

- Supports DIALIN and DIRECT connect
- Additional password on DIALIN
- One login per invocation
- Must be stamped in LOGIN subsystem

SIOMONIT and PROGENY



Other Things to Know

- SIOMONIT

  * Reads SIOMONIT_FILE

  At Startup

  At Child Death if
  –RESTART option

  When ‘QUIT’ Fault Received

  Every 15 minutes if there is
  Child Death

- You can change SIOMONIT_FILE and “SIGP” to kick it off.

- “SIGP –STOP” will stop SIOMONIT.

- Waits 15 seconds to be sure child stays alive.

- SIOLOGIN
  - * Must be stamped in the LOGIN subsystem
  - * Hangs up phone line if –DIALIN option
  - * Can use STARTUP_SIO.SH to force unlock

"ULKOB /DEV/SIOx –F"

described in help files

ALARM_SERVER

- Brings to user's attention certain asynchronous events
- Events currently supported
  - MAIL
  - DSEE TASKLISTS
  - Disk is full for "/"
  - Ring hardware failures
  - NETMAIN observations
- Requires MBX_HELPER

ALARM_SERVER: How It Works

- Internal Scheduler plus Array of Procedures
- Schedules by Time and Certain Event Counts
- Opens Mailboxes in ’NODE_DATA and ~USER_DATA for SEND_ALARMS
- Diddles ACL on ~USER_DATA MBX for MBX_HELPER
- Requires Binding with Initialization and Service Procedures
- Cost
  - * once/minute = 1.5% CPU

Font file format (Version 1 only used):
sys/ins/smdu.ins.pas

Store-and-Forward helper

Clob process UID
in ’node_data/proc_dir

PROC2_$SET_MY_NAME

Store and Forward

- IPC from X to Y when Y may not be available
- Contrast to MBX
- Stuffs messages in SF_QUEUES
- Requires at least one SF_HELPER on ring
- Supports routing & notification
- Special Queue : /SYS/SF/LOCAL_Q
- Used by DSEE
- Interface NOT released

S+F looks at everything in // even if the nodes aren't there anymore

Solution:

uncfobj all in // except self

then ctnode -update

SF—How it Works

- Program calls SF_$PUT
  - * "Enqueue this message over there."

    "OK—done?" or "Couldn't. I put it in the LOCAL_Q."
- Some time later
  - * SF_HELPER wakes up
  - * Looks at his queues
  - * Moves message 'over there'
  - * Can look at all LOCAL_Q's
  - * Uses // directory for ALL_LOCAL
  - * Runs as USER.SERVER.NONE
- Notification Support
  - * A process may register at a queue and receive fault notice

AEGIS

PERFORMANCE

ANALYSIS

Performance Analysis

- Proactive

  * Cost: X

    design performance into application

  * Benefit: 10X

- Reactive,

  * Cost: 10X

  * Benefit: X

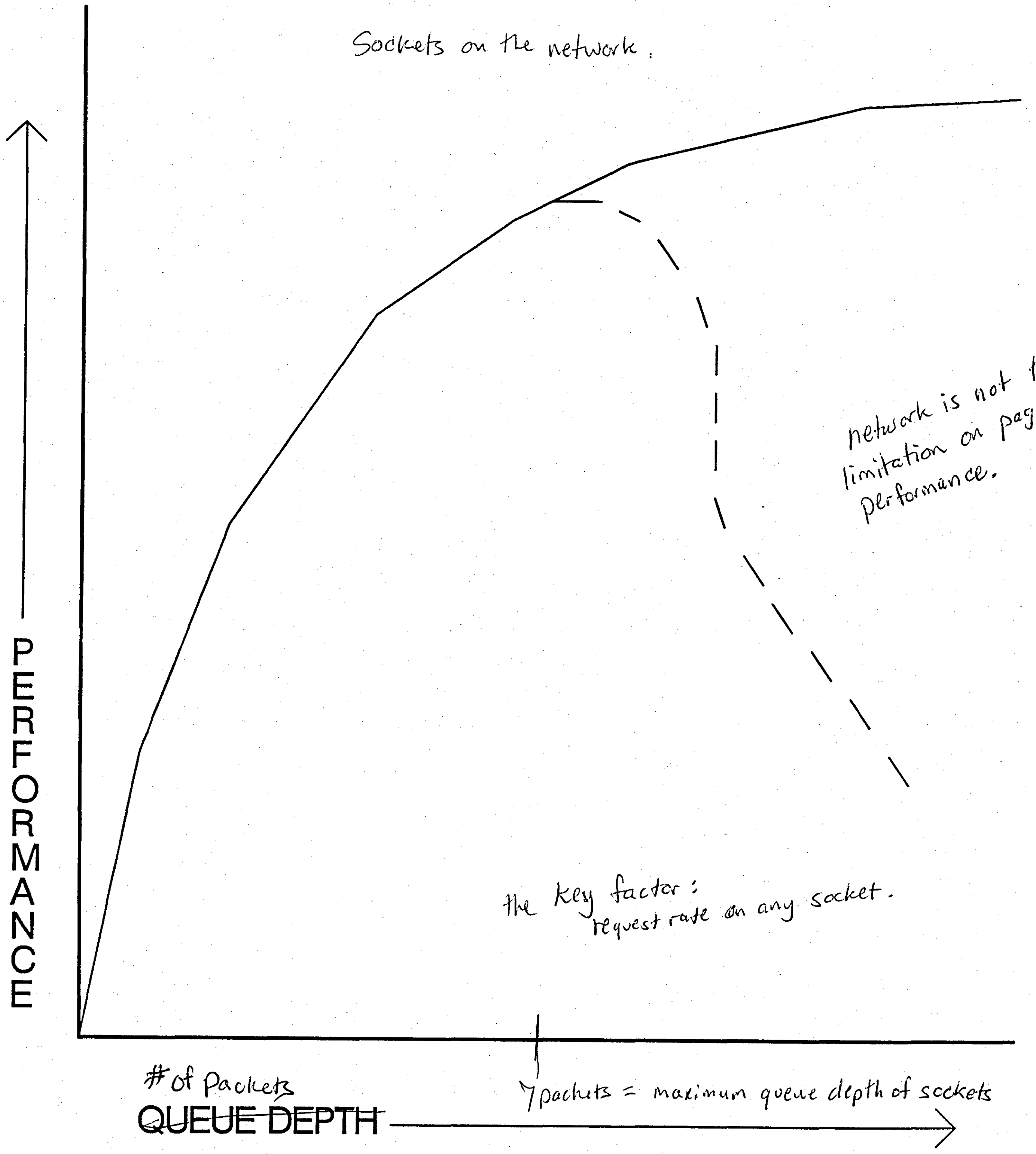
Important Nonlinear Effects

* Queueing

* Caching

* Tuning

QUEUEING



Sockets on the network.

network is not the limitation on paging performance.

the key factor: request rate on any socket.

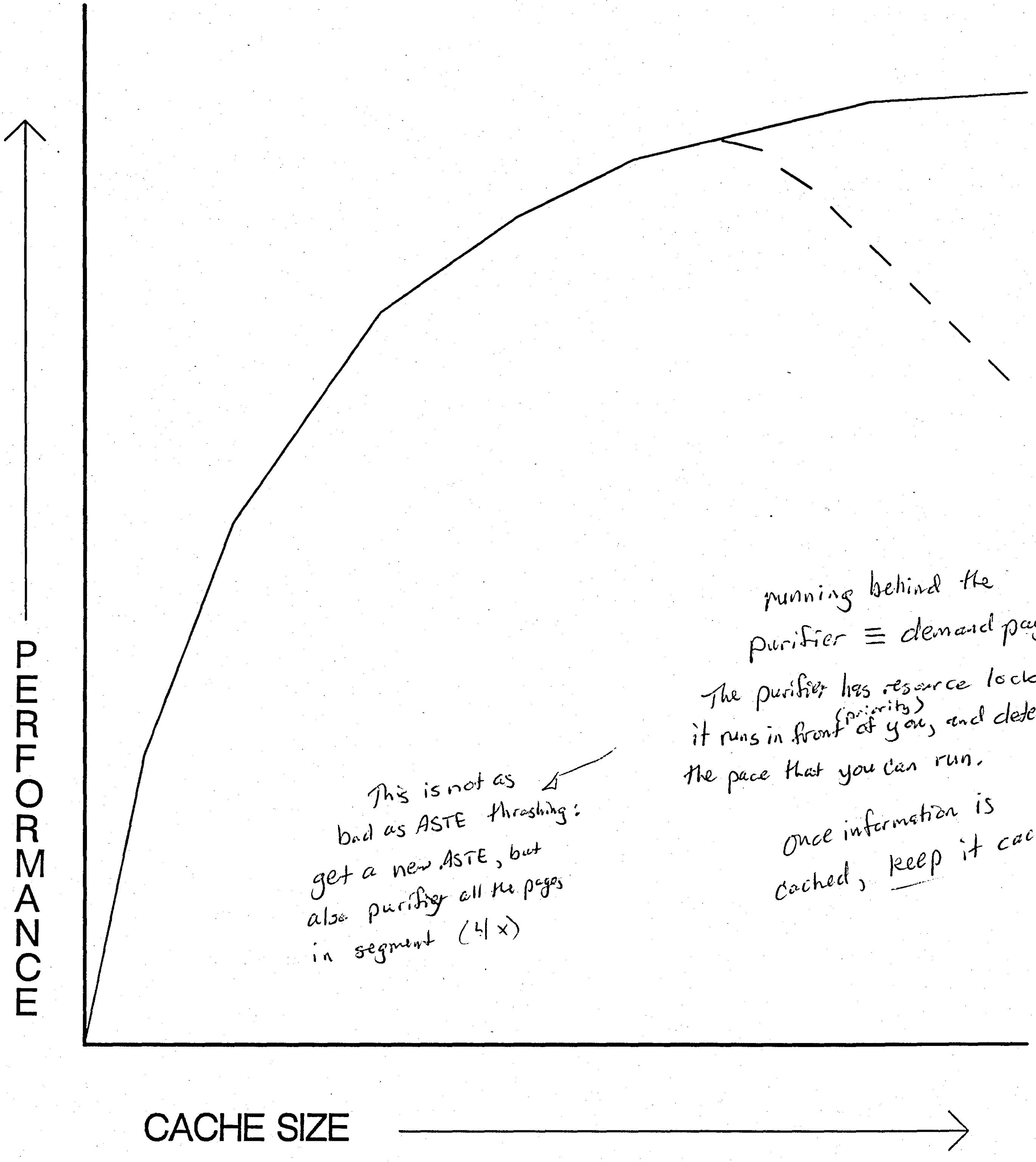
7 packets = maximum queue depth of sockets

Worst cases: a tight loop that reads and writes file; and sparse references.

Solve problems by not demanding pages fast. By waiting a little longer you actually run faster.

Future possibility: Distribute storage over several file servers: Interleave it for efficiency. ("Mega objects" composed of several Apollo objects)

CACHING



This is not as bad as ASTE thrashing: get a new ASTE, but also purifier all the pages in segment (4x)

running behind the purifier ≡ demand paging,

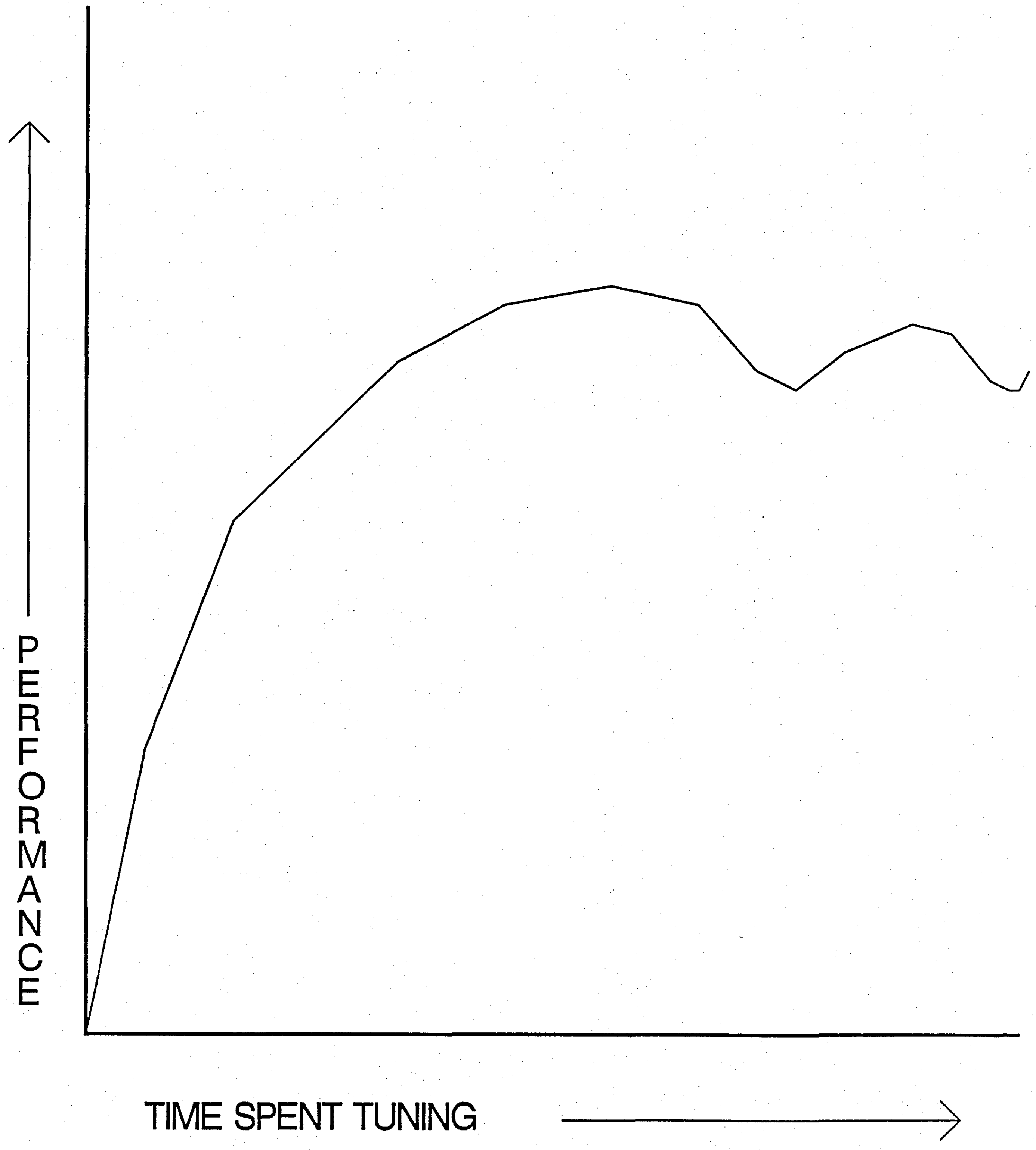
The purifier has resource locks, so it runs in front (priority) of you, and determines the pace that you can run.

Once information is cached, keep it cached.

When you get something where you want to use it, keep it there (by continually touching the page).

Take advantage of the caching.

TUNING



Tuning

- * Start with a known baseline

  write simple test programs to see limitations.

- * Define performance requirements

- * Go for "smoking gun(s)"

  things that have possible dramatic effects.

- * Measure effects at each point in the tuning process.

Put files that a program uses in same directory to take advantage of that directory info already in cache.

Benchmarks

1. NETSVC –L (if possible) See how netsvc effect performance

-all (all systems on network)

2. BLDT (make sure you don't have different revs on the network)

3. /SYSTEST/COM/CALIBRATE — tight for loop measures CPU power.

4. NETSTAT –L –CONFIG (before and after)

5. PST –PA –L1 (before and after)

6. Run benchmark

7. Save pad and a LD –A –SI -D of all important files

/SYSTEST/COM/CALIBRATE

– pure CPU cycle time

- CPU "benchmark"
  - * no I/O or paging
  - * single memory reference
  - * extremely consistent
  - * can be affected by "loading"

- Typical Values (calibration ratios)

DN400:	1.04
DN300:	0.70
DN420 (w/ PEB):	0.70
DSP80:	0.80
DN550:	0.82
DN460:	0.19

the whole loop fits in the I-cache

The Complete Application Debugger's Toolbox

- DEBUG
- PROGRAM
  - * Self–Monitoring
- TB (Traceback)
- FST (Fault Status) the diagnostic frame
- PST (Process Status)
- LAS (List Address Space) -u
- LLKOB (List Locked Objects) -u
- DB (MD–style Debugger)

- DEBUG
  - * Use

    don't use; this defeats all optimization

    Symbol   all

    PAS   –DBS or –DBA

  - * REGS
  - * FPREGS

    -opt all Switch to Fortran

  - * DB
- PROGRAM self–monitoring
  - * Use

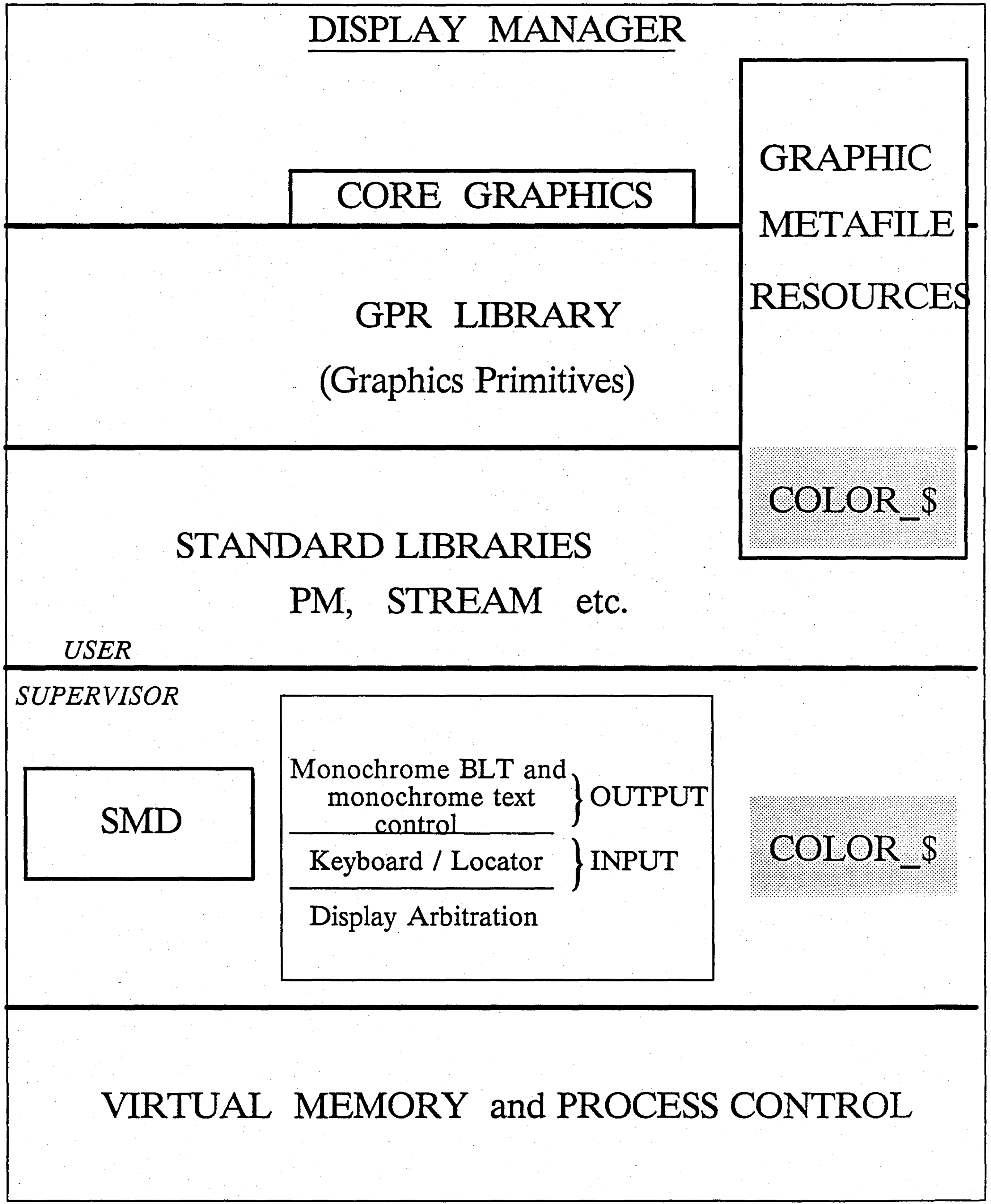
    PAS   –COND
    {% DEBUG} VFMT_$ ...

  - * Switches

    –DB

    –MONIT   (eg. EMT)
- PST

  –L1 (Level one processes)
  –TYPE (aegis/user/server)



DISPLAY MANAGER

CORE GRAPHICS

GRAPHIC METAFILE RESOURCES

GPR LIBRARY

(Graphics Primitives)

COLOR_$

STANDARD LIBRARIES

PM, STREAM etc.

USER

SUPERVISOR

SMD

Monochrome BLT and monochrome text control } OUTPUT

Keyboard / Locator } INPUT

Display Arbitration

COLOR_$

VIRTUAL MEMORY and PROCESS CONTROL

apollo

DOMAIN

Integrated Local Network of Workstations

Workstation (node)

- virtual memory
- bit–map graphics / pointing device
- 12 megabit / sec token passing ring

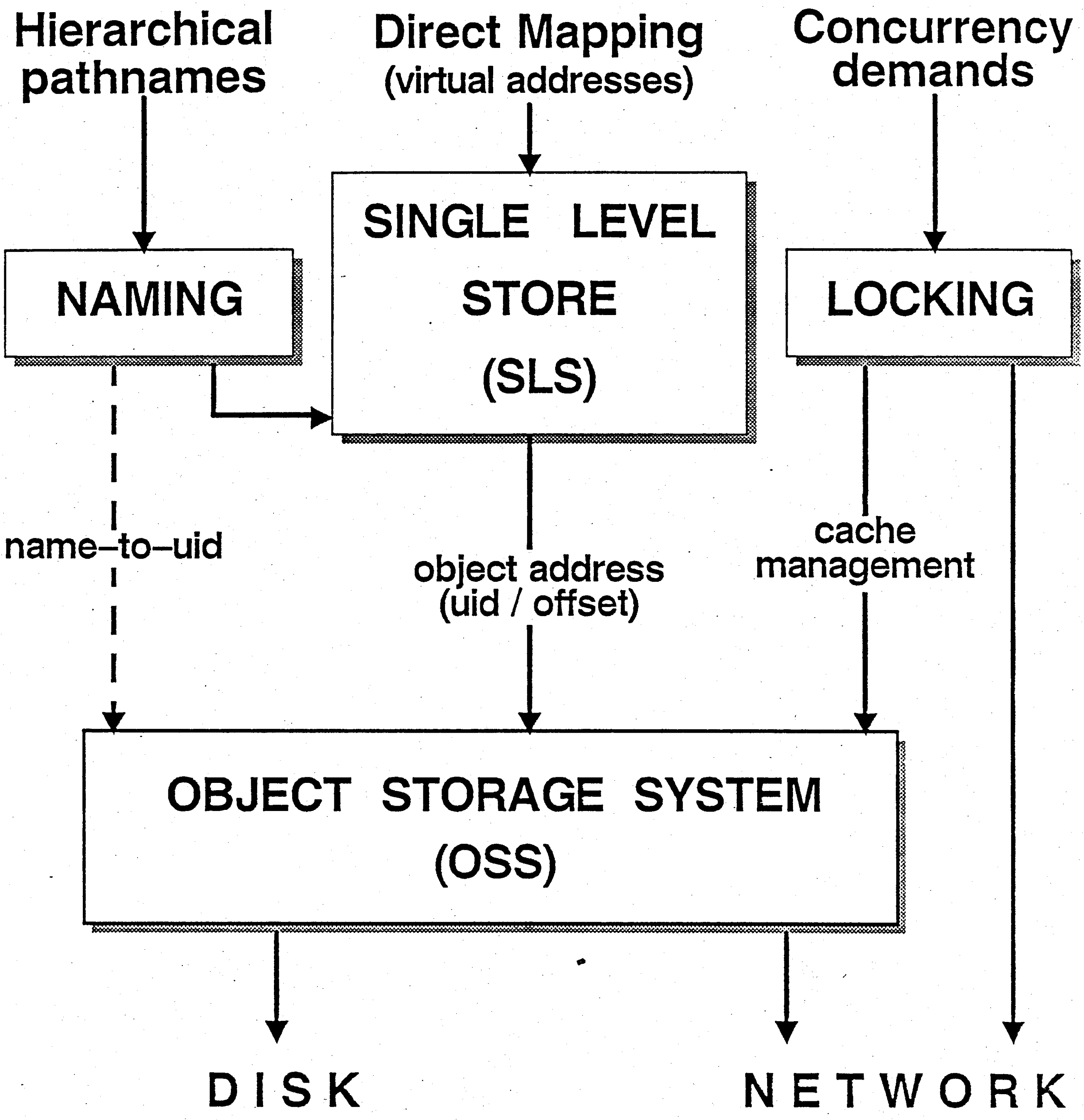
Operating system (AEGIS)

- network–wide flat file system

  typed containers identified by UIDs
- network–wide hierarchical name–space
- network transparency for object access
- single–level–store (SLS)

  objects are "mapped" into the
  process virtual address space and
  operated on with machine instructions

AEGIS SYSTEM MODEL



Hierarchical pathnames

Direct Mapping (virtual addresses)

Concurrency demands

NAMING

SINGLE LEVEL STORE (SLS)

LOCKING

name–to–uid

object address (uid / offset)

cache management

OBJECT STORAGE SYSTEM (OSS)

DISK

NETWORK

SINGLE LEVEL STORE
(SLS)

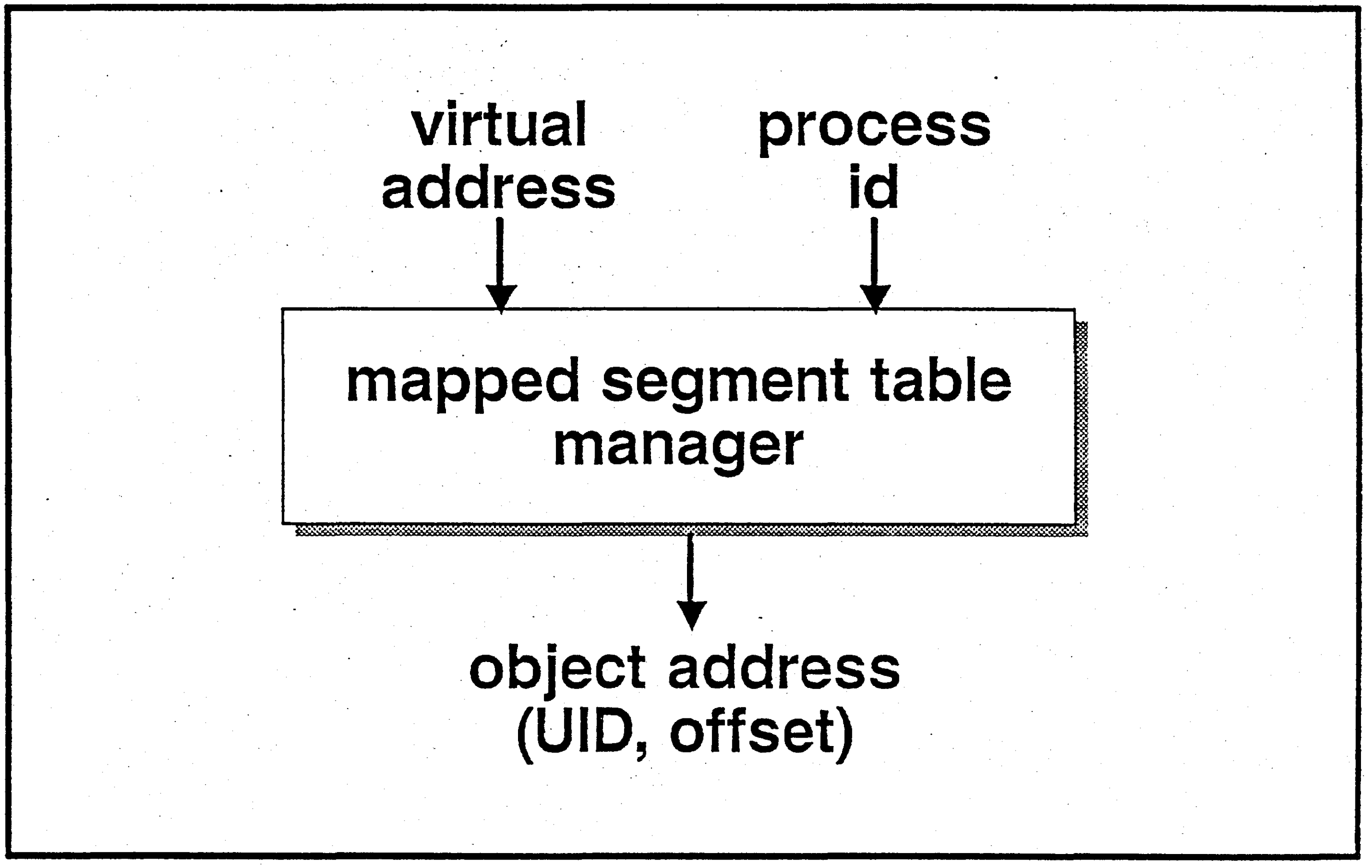
Mapping objects

manage per–process virtual address space

segmented — address space and objects

virtual address —> object address

NO KNOWLEDGE OF OBJECT LOCATION



virtual address

process id

mapped segment table manager

object address (UID, offset)

OBJECT STORAGE SYSTEM (OSS)

Object locating

UID —> location in the network

Location independent object management

create, delete, attributes control

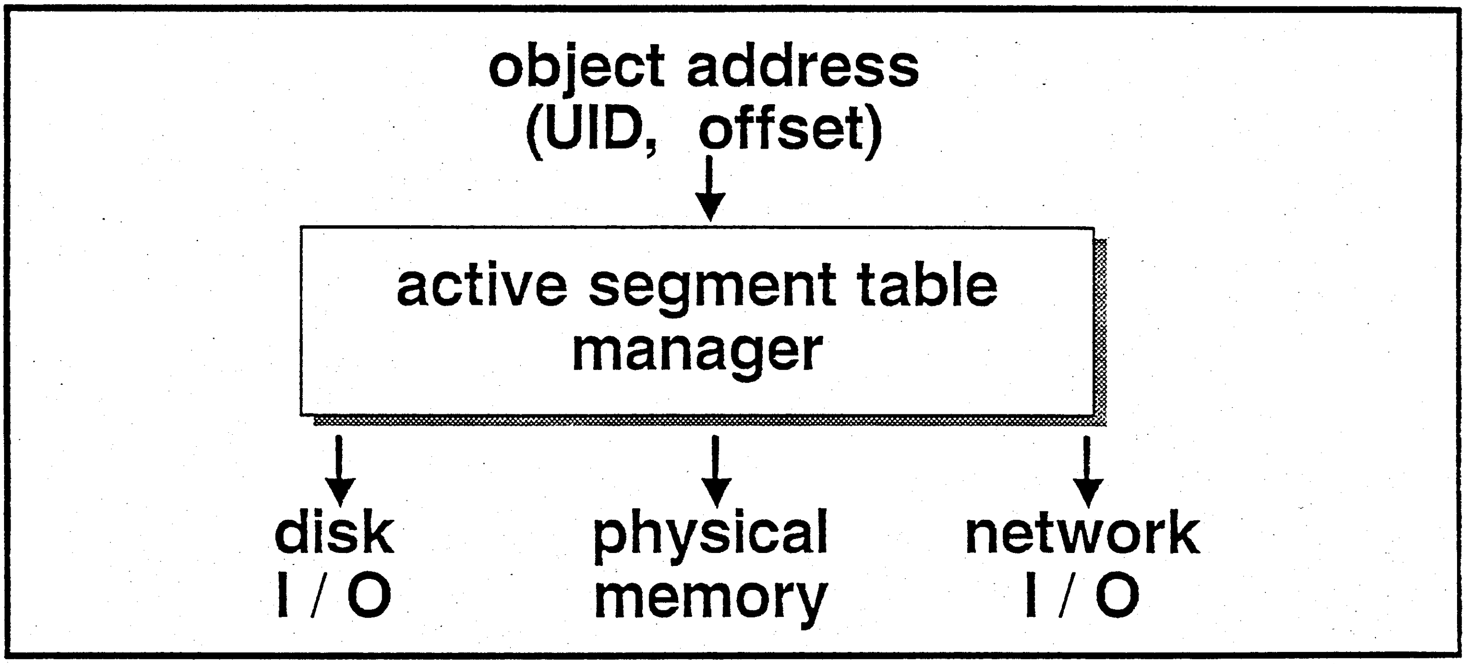
Demand paging

(UID, offset) —> physical memory page #

physical memory page cache management

"active" object table management

disk storage management



```
$ netstat -l -config


     The node ID of this node is 1797.

**** Node 1797 ****   "//slash"
Time 1985/03/05.17:12:12  Up since 1985/03/05.17:10:57

Net I/O:          total=     18   rcvs =     10   xmits =      8

  0 page-in  requests issued.
  0 page-out requests issued.
  0 page-in  requests serviced.
  0 page-out requests serviced.
Detected concurrency violations -- read: 0    write: 0

 Xmit count         8        Rcv eor            0
 NACKs              0        Rcv crc            0
 WACKs              0        Rcv timout         0
 Token inserted     1        Rcv buserr         0
 Xmit overrun       0        Rcv overrun        0

 Xmit Ack par       0        Rcv xmit-err       0
 Xmit Bus error     0        Rcv Modem err      0
 Xmit timout        0        Rcv Pkt error      0
 Xmit Modem err     0        Rcv hdr chksum     0
 Xmit Pkt error     0        Rcv Ack par        0

    Delay switched OUT.

Winchester I/O:  total=   1540   reads=   1149   writes=    391

 Not ready          0        Contrlr busy       0
 Seek error         0        Equip check        0
 Drive time out     0        Overrun            0
 CRC error percentage: 0.00%

No ring hardware failure report.
System configured with 1.5 mb of memory.
A total of 0 parity errors were detected.

  NODE CONFIGURATION
    Node Type:  DN300/DN320
    Display type:  17/19 inch landscape display
    Disk type:  MSD-34M
```

```
$ pst -il -pa -ty -r 30
-------------------------------------------------------------------------------------------------------------
Processor   ¦ PRIORITY ¦  Program ¦ State ¦ Private ¦  Global ¦ D I S K ¦  N E T  ¦  Type  ¦     Process
Time (sec)  ¦ mn/cu/mx ¦  Counter ¦       ¦  Faults ¦  Faults ¦ Page IO ¦ Page IO ¦  UID   ¦      Name
-------------------------------------------------------------------------------------------------------------
   147.752    1/ 0/16            0  Ready         0         0         0         0   aegis   <Null Process>
     0.767    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Clock Process>
     2.037    1/16/16     C9CC00E0   Wait         0         0       478         0   aegis   <Page Purifier>
     0.368    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Terminal Server>
     0.001    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Receive Server>
     0.001    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Paging Server>
     0.026    1/16/16     C9CC00E0   Wait         0         1         3         0   aegis   <Net Request Server>
    16.786   16/16/16        1A686   Wait       545       689       981         0    user   display_manager
     2.181    1/16/16        1A496   Wait        76        52       118         0  server   print_server
    .0.483    1/16/16        1A21E   Wait        29        11        39         0  server   mbx_helper
     1.538    1/14/16        1A5AE   Wait        55        25        46         1    user   process_3
     0.776    1/14/16     <active>  Ready        56         5        17         0    user   process_4
 -----------                                --------  --------  --------  --------
   174.723                                      761       783      1682         1
```

meaningless for aegis processes

address range in private portion of MST

always 0 because no private MST

input

output

level 1 only

```
-------------------------------------------------------------------------------------------------------------
Processor   ¦ PRIORITY ¦  Program ¦ State ¦ Private ¦  Global ¦ D I S K ¦  N E T  ¦  Type  ¦     Process
Time (sec)  ¦ mn/cu/mx ¦  Counter ¦       ¦  Faults ¦  Faults ¦ Page IO ¦ Page IO ¦        ¦      Name
-------------------------------------------------------------------------------------------------------------
    26.138    1/ 0/16            0  Ready         0         0         0         0   aegis   <Null Process>
     0.099    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Clock Process>
     0.099    1/16/16     C9CC00E0   Wait         0         0         9         0   aegis   <Page Purifier>
     0.129    1/16/16     C9CC00E0   Wait         0         2         2         0   aegis   <Terminal Server>
     0.000    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Receive Server>
     0.000    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Paging Server>
     0.001    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Request Server>
     2.447   16/16/16        1A686  Ready         7         7        22         0    user   display_manager
     0.016    1/16/16        1A496   Wait         0         0         0         0  server   print_server
     0.000    1/16/16        1A21E   Wait         0         0         0         0  server   mbx_helper
     0.276    1/15/16        3885E  Ready        10         4        19         0    user   process_3
     0.655    1/16/16     <active>  Ready         3         2         2         0    user   process_4
 -----------                                --------  --------  --------  --------
    29.864                                       20        15        54         0

-------------------------------------------------------------------------------------------------------------
Processor   ¦ PRIORITY ¦  Program ¦ State ¦ Private ¦  Global ¦ D I S K ¦  N E T  ¦  Type  ¦     Process
Time (sec)  ¦ mn/cu/mx ¦  Counter ¦       ¦  Faults ¦  Faults ¦ Page IO ¦ Page IO ¦        ¦      Name
-------------------------------------------------------------------------------------------------------------
    16.701    1/ 0/16            0  Ready         0         0         0         0   aegis   <Null Process>
     0.097    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Clock Process>
     0.066    1/15/16     C9CC00E0   Wait         0         0         6         0   aegis   <Page Purifier>
     0.064    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Terminal Server>
     0.000    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Receive Server>
     0.000    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Paging Server>
     0.000    1/16/16     C9CC00E0   Wait         0         0         0         0   aegis   <Net Request Server>
     1.189   16/16/16        1A686  Ready         0         0         0         0    user   display_manager
     0.016    1/16/16        1A496   Wait         0         0         0         0  server   print_server
     0.000    1/16/16        1A21E   Wait         0         0         0         0  server   mbx_helper
    11.209    1/ 1/16      2B0076  Ready        35        21        31         0    user   process_3
     0.605    1/16/16     <active>  Ready         2         0         0         0    user   process_4
 -----------                                --------  --------  --------  --------
    29.969                                       37        21        37         0
```

/systest/ssr_util/ringlog -start
-stop

```
$ ringlog -start
Ringlog [3.2]
$ lcnode

 The node ID of this node is 2246.
 2 other nodes responded.

Node ID      Boot time             Current time          Entry Directory

 2246   1985/03/05 10:49:54   1985/03/05 10:55:33   //sr8.1
 2EF6   1985/03/05 10:41:55   1985/03/05 10:49:23   //node_2ef6
 146C   1985/03/05 10:11:25   1985/03/05 10:49:23   *** DISKLESS ***  partner node: 2EF6


$ ld //node_2ef6

Directory "//node_2ef6":

bscom            com              dev              domain_examples
ftu              install          lib              preserve
registry         sau2             sau4             sse_035
sys              sys.delete       sysboot          systest

16 entries.
$ ld //node_2ef6/com

Directory "//node_2ef6/com":

acl              arcf             args             bind             bldt
calendar         catf             chhdir           chn              chpass
chpat            chuvol           clstr            cmf              cmsrf
cmt              cpboot           cpf              cpfx25           cpl
cpscr            cpt              crd              crddf            crefpas
crefs            crf              crl              crp              crpad
crrgy            crsubs           crucr            ctnode           ctob
cvt_rec_uasc     date             db               dcalc            debug
dldupl           dlf              dll              dlt              dmtvol
dsee             ed               edacct           edacl            edfont
edmtdesc         edppo            edstr            em3270.icci      em3270.kmw
em3270.pci       emhasp           emrje            emt              emtx25
ensubs           esa              exfld            find_orphans     flen
fmc              fmt              fpat             fpatb            fppmask
fserr            fst              ftn              ftp              haspsvr
help             host             hpc              invol            lamf
las              lbr              lcnode           ld               lkob
llkob            login            lopstr           lrgy             lusr
lvolfs           macro            mtvol            mvf              nd
net              netmain          netmain_chklog   netmain_note     netstat
netsvc           obty             oed              os               pagf
pas              ppri             prf              probenet         prsvr
pst              rbak             revl             rjesvr           rwmt
salacl           sald             salrgy           salvol           scrto
sh               sigp             siorf            siotf            srf
stcode           subs             tb               tcpstat          tctl
tee              telnet           tlc              tpm              tugs
tugs_author      tz               uctnode          uctob            ulkob
vctl             vsize            vt100            wbak             wd
wl               wlist            xdmc             xsubs

149 entries.
```

```
$ llkob //node_2ef6/com

                     Home  Locking
 Use  Constraint     Node   Node        Pathname

  W   nR_xor_1W      2246   2246        /sys/node_data/stack
  W   Cowriters      2246   2246        /sys/node_data/shell
  W   nR_xor_1W      2246   2246        /sys/node_data/hint_file
  W   nR_xor_1W      2246   2246        /sys/node_data/sys_error_log
  W   nR_xor_1W      2246   2246        /sys/node_data/data$
  R   nR_xor_1W      2246   2246        /sys/env
  W   nR_xor_1W      2246   2246        /sys/node_data/global_data
  R   nR_xor_1W      2246   2246        /lib/pmlib
  R   nR_xor_1W      2246   2246        /lib/syslib
  R   nR_xor_1W      2246   2246        /lib/streams
  R   nR_xor_1W      2246   2246        /lib/vfmt_streams
  R   nR_xor_1W      2246   2246        /lib/error
  R   nR_xor_1W      2246   2246        /lib/swtlib
  R   nR_xor_1W      2246   2246        /lib/ftnlib
  R   nR_xor_1W      2246   2246        /lib/pbulib
  R   nR_xor_1W      2246   2246        /lib/gprlib
  R   nR_xor_1W      2246   2246        /lib/clib
  R   nR_xor_1W      2246   2246        /lib/shlib
  R   nR_xor_1W      2246   2246        /lib/tfp
  W   Cowriters      2246   2246        /sys/node_data/acl_cache
  W   nR_xor_1W      2246   2246        /sys/node_data/stream_$sfcbs
  R   nR_xor_1W      2246   2246        /sys/dm/dm
  W   Cowriters      2246   2246        /sys/node_data/dm_mbx
  W   nR_xor_1W      2246   2246        /sys/node_data/pdb
  W   Cowriters      2246   2246        -- temporary file --
  R   nR_xor_1W      2246   2246        /sys/dm/fonts/f5x9
  R   nR_xor_1W      2246   2246        /sys/dm/fonts/legend.191
  R   nR_xor_1W      2246   2246        /sys/dm/fonts/icons
  W   nR_xor_1W      2246   2246        /sys/node_data/paste_buffers/all_group
  W   nR_xor_1W      2246   2246        /sys/node_data/paste_buffers/invis_group
  W   nR_xor_1W      2246   2246        /sys/node_data/paste_buffers/icon_group
  W   Cowriters      2246   2246        /sys/node_data/sysmbx
  R   nR_xor_1W      2246   2246        /com/sh
  R   nR_xor_1W      2246   2246        /sys/mbx/mbx_helper
  R   nR_xor_1W      2246   2246        /com/prsvr
  W   Cowriters      2246   2246        /sys/node_data/dm_mbx
  W   nR_xor_1W      2246   2246        /sys/node_data/dev/sio2
  W   nR_xor_1W      2246   2246        -- Display Manager PAD --
  W   Cowriters      2246   2246        /sys/node_data/dm_mbx
  W   nR_xor_1W      2246   2246        -- Display Manager PAD --
  R   nR_xor_1W      2246   2246        /com/sh
  R   nR_xor_1W      2246   2246        /com/sh
  R   nR_xor_1W      2246   2246        /com/pst
  W   nR_xor_1W      2246   2246        /sys/node_data/paste_buffers/again
  W   nR_xor_1W      2246   2246        -- Display Manager PAD --
  W   Cowriters      2246   2246        /sys/node_data/dm_mbx
  R   nR_xor_1W      2246   2246        /com/sh
  R   nR_xor_1W      2246   2246        /systest/com/calibrate
  R   nR_xor_1W      2246   2246        /com/llkob

  49 files locked.
```

```
♦ ringlog -stop
Ringlog [3.2]
odata.index = 53

              From  TO
    NODE TID  Sock Sock  RQST/RPLY
    ==== ==== ==== ==== ==========
xmt 0002   1C WHO  INFO    2    0    0 2246  3E7   E0 3BEB   E2 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   1C INFO WHO     2    1    0    0    0 2EF6 B1FF  3E7    C    2   20   13   F1
rcv 2EF6   1C WHO  INFO    2    0    0 2246  3E6   E0 3BEB   E2 2020 2020 2020 2020 2020
rcv 146C   1C INFO WHO     2    1    0    0    0 146C B1FF  3E6 F6C0    0    0   E2 E55C
rcv 146C   1C WHO  INFO    2    0    0 2246  3E5   E0 3BEB   E2 2020 2020 2020 2020 2020
xmt 2EF6   1D   12 INFO    2    2    0    0  109    0 7EF7    0 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   1D INFO   12    2    3    0    0 2524 5E2C 2524 64D7    C    2   20   13   F1
xmt 2EF6   1E   12 INFO    2    C    0    0   F0  FBA    3 FF00 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   1E INFO   12    2    D    0    0    0 2EF6   24 64D7    C    2   20   13   F1
xmt 2EF6   1F   12 INFO    2    4    0    0   F0  FBA    3 FF00 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   1F INFO   12    2    5    0    0 24FB C042 5000 2EF6    C    2   20   13   F1
xmt 2EF6   20   12 PAGE info   rqst: 24FBC042.50002EF6 type=8  SS=8000
rcv 2EF6   20 PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
xmt 146C   21   12 INFO    2    2    0    0  148    0 7EB8    0 1000 2246    1   E0 8000  SS=8000
rcv 146C   21 INFO   12    2    3    0    0 2524 42E5 2524 64D7 F6C0    0    0   E2 E55C
xmt 146C   22   12 INFO    2    C    0    0  14F    0 7EB1    0 1000 2246    1   E0 8000  SS=8000
rcv 146C   22 INFO   12    2    D    0    0    0 2EF6 FF24 64D7 F6C0    0    0   E2 E55C
xmt 0002   23 WHO  INFO    2    0    0 2246  3E7   E0 3BEB   E2 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   23 INFO WHO     2    1    0    0    0 2EF6 B1FF  3E7    C    2   20   13   F1
rcv 2EF6   23 WHO  INFO    2    0    0 2246  3E6   E0 3BEB   E2   81    0    0    1 118F
rcv 146C   23 INFO WHO     2    1    0    0    0 146C B1FF  3E6 F6C0    0    0   E2 E55C
rcv 146C   23 WHO  INFO    2    0    0 2246  3E5   E0 3BEB   E2    C    2   20   13   F1
xmt 2EF6   24   12 INFO    2    2    0    0  2BB    0 7D45    0 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   24 INFO   12    2    3    0    0 2524 5E2C 2524 657A    C    2   20   13   F1
xmt 2EF6   25   12 INFO    2    C    0    0  2BB    0 7D45    0 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   25 INFO   12    2    D    0    0    0 2EF6   24 657A    C    2   20   13   F1
xmt 2EF6   26   12 INFO    2    4    0    0  2BB    0 7D45    0 1000 2246    1   E0 8000  SS=8000
rcv 2EF6   26 INFO   12    2    5    0    0 24FB C042 5000 2EF6    C    2   20   13   F1
xmt 2EF6   27   12 PAGE info   rqst: 24FBC042.50002EF6 type=8  SS=8000
rcv 2EF6   27 PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
xmt 146C   28   12 INFO    2    2    0    0  2FA    0 7D08    0 1000 2246    1   E0 8000  SS=8000
rcv 146C   28 INFO   12    2    3    0    0 2524 42E5 2524 657A F6C0    0    0   E2 E55C
xmt 146C   29   12 INFO    2    C    0    0  301    0 7CFF    0 1000 2246    1   E0 8000  SS=8000
rcv 146C   29 INFO   12    2    D    0    0    0 2EF6 FF24 657A F6C0    0    0   E2 E55C
xmt 2EF6   2A   12 PAGE info   rqst: 24FBC042.50002EF6 type=8  SS=8000
rcv 2EF6   2A PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
xmt 2EF6   2B   12 FILE lock   rqst: 24FBC042.50002EF6  --read lock --  SS=8000
rcv 2EF6   2B FILE   12 lock   rply: dtm=25245E31.18 st=0
xmt 2EF6   2C   12 PAGE info   rqst: 24FBC042.50002EF6 type=6  SS=8000
rcv 2EF6   2C PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
xmt 2EF6   2D   12 PAGE multpg rqst: 24FBC042.50002EF6 page=   0 (4 pages) dtm=   22524  SS=8000
rcv 2EF6   2D PAGE   12 multpg rply: 24FBC042.50002EF6 page=   0 (1 of 2) dtmh=2524 st=0
rcv 2EF6   2D PAGE   12 multpg rply: 24FBC042.50002EF6 page=   0 (2 of 2) dtmh=2524 st=0
xmt 2EF6   2E   12 FILE unlock rqst: 24FBC042.50002EF6  SS=8000
rcv 2EF6   2E FILE   12 unlock rply: st=0
xmt 2EF6   2F   12 PAGE info   rqst: 24FBC042.50002EF6 type=8  SS=8000
rcv 2EF6   2F PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
```

```
xmt 2EF6   30   12 FILE nrslve rqst: 24FBC042.50002EF6 "COME..."  SS=8000
rcv 2EF6   30 FILE   12 nrslve rply: "COM"   st=0
xmt 2EF6   31   12 PAGE info   rqst: 24FBC7A4.90002EF6 type=8  SS=8000
rcv 2EF6   31 PAGE   12 info   rply: 24FBC7A4.90002EF6 info= perm dir (nil) st=0
xmt 2EF6   32   12 FILE nrslve rqst: 24FBC042.50002EF6 "COME..."  SS=8000
rcv 2EF6   32 FILE   12 nrslve rply: "COM"   st=0
xmt 2EF6   33   12 FILE lock   rqst: 24FBC7A4.90002EF6  --read lock --  SS=8000
rcv 2EF6   33 FILE   12 lock   rply: dtm=2501C959.88 st=0
xmt 2EF6   34   12 PAGE info   rqst: 24FBC7A4.90002EF6 type=6  SS=8000
rcv 2EF6   34 PAGE   12 info   rply: 24FBC7A4.90002EF6 info= perm dir (nil) st=0
xmt 2EF6   35   12 PAGE multpg rqst: 24FBC7A4.90002EF6 page=  0 (4 pages) dtm=   B2524  SS=8000
rcv 2EF6   35 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  0 (1 of 1) dtmh=2501 st=0
xmt 2EF6   36   12 PAGE multpg rqst: 24FBC7A4.90002EF6 page=  1 (4 pages) dtm=52160598  SS=8000
rcv 2EF6   36 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  1 (1 of 3) dtmh=2501 st=0
rcv 2EF6   36 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  1 (2 of 3) dtmh=2501 st=0
rcv 2EF6   36 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  1 (3 of 3) dtmh=2501 st=0
xmt 2EF6   37   12 PAGE multpg rqst: 24FBC7A4.90002EF6 page=  4 (4 pages) dtm=1AFEC959  SS=8000
rcv 2EF6   37 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  4 (1 of 2) dtmh=2501 st=0
rcv 2EF6   37 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  4 (2 of 2) dtmh=2501 st=0
xmt 2EF6   38   12 PAGE multpg rqst: 24FBC7A4.90002EF6 page=  6 (4 pages) dtm=1AFEC959  SS=8000
rcv 2EF6   38 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  6 (1 of 3) dtmh=2501 st=0
rcv 2EF6   38 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  6 (2 of 3) dtmh=2501 st=0
rcv 2EF6   38 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  6 (3 of 3) dtmh=2501 st=0
xmt 2EF6   39   12 PAGE multpg rqst: 24FBC7A4.90002EF6 page=  9 (4 pages) dtm=1AFEC959  SS=8000
rcv 2EF6   39 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  9 (1 of 2) dtmh=2501 st=0
rcv 2EF6   39 PAGE   12 multpg rply: 24FBC7A4.90002EF6 page=  9 (2 of 2) dtmh=2501 st=0
xmt 2EF6   3A   12 FILE unlock rqst: 24FBC7A4.90002EF6  SS=8000
rcv 2EF6   3A FILE   12 unlock rply: st=0
xmt 2EF6   3B   12 PAGE info   rqst: 24FBC7A4.90002EF6 type=8  SS=8000
rcv 2EF6   3B PAGE   12 info   rply: 24FBC7A4.90002EF6 info= perm dir (nil) st=0
xmt 2EF6   3C   12 FILE lock   rqst: 24FBC042.50002EF6  --read lock --  SS=8000
rcv 2EF6   3C FILE   12 lock   rply: dtm=25245E31.18 st=0
xmt 2EF6   3D   12 FILE unlock rqst: 24FBC042.50002EF6  SS=8000
rcv 2EF6   3D FILE   12 unlock rply: st=0
xmt 2EF6   3E   12 PAGE info   rqst: 24FBC042.50002EF6 type=8  SS=8000
rcv 2EF6   3E PAGE   12 info   rply: 24FBC042.50002EF6 info= perm sysdir (nil) st=0
```

Changes for global libraries: 82/05/26

1) Global I Address Space
   a) global space (8000-200000, or roughly 2 MB)
      1) pure KGT
      2) pure code & data

   b) available private space (200000-BC0000, or roughly 9+ MB)
      1) 200000 (1) - process creation record
      2) 208000 (5) - impure library data
      3) 230000 (1) - guard segment
      4) 238000 (8) - stack
      5) 278000 (1) - guard segment
      6) 280000 (2) - private kgt, rws scratch space
      7) 290000 - available

   c) you'll see "guard fault" on stack overflow - only once per process

2) Global Library Changes
   a) all read-only sections, plus data$ are shared, ergo...
   b) data$ section must be pure (ecb's, ac's, constants only!)
   c) all other data must be placed in other sections (sugg. name: module_data$) use new VAR statement syntax in Pascal, common in Fortran
   d) impure externs must be handled specially (assembler module is required)
   e) all uninitialized pure and impure data are guaranteed to be set = 0, generally eliminating the need for library initialization procedures
   f) 2 new libraries: pmlib (process manager) and shlib (shell)

3) Global Library Installation
   a) installed by process manager when ENV or DMENV is loaded
   b) to install new global library:
      1) rename old library (use change_name's -D option)
      2) copy new library into /lib
      3) exit and re-start the display manager (it's unnecessary to restart OS)
      4) delete the old library (when you're confident of the new one!)

   c) library initialization procedures are still called at process creation
   d) streams is initialized at DMENV load time, by calling stream_$process_init (a misnomer); no per-process streams initialization is currently required
   e) libraries are not unmapped upon return to boot shell. They are re-mapped by env or dmenv

4) Debugging Libraries in User Space
   a) use db's install command, as presently done
   b) 2e doesn't apply, so a main program or init procedure may be required to zero-fill data
   c) names are inserted into private kgt, which is searched prior to global (pure) kgt
   d) just a reminder that mark/release is still not called (this is unchanged)
   e) special handling for streams: to use shared stream sfcb's, don't bind stream_pure_data.bin (omission of this will cause the global space definitions to be used

5) What SSR's and certain customers should know:

a) can't mix and match SR4 libraries and OS with previous releases

b) customers may no longer bind their libraries with FTNLIB

c) customers using mst_$map_at and mst_$seg_guard must also be sensitive to these changes

d) customers may now install a private library by creating an object file named "/lib/userlib.private". The uid of this file is captured at system startup time (i.e. the time at which env or dmenv is loaded) This mechanism is not supported

e) customers may install a global library by creating an object file named "/lib/userlib.global". These global libraries must adhere to the rules outlined above. Apollo is NOT releasing or supporting customer global libraries

Additional information on installed libraries.

1. Installing a library adds the entry points to a per-process database called the "known global table". This table is later used by the loader to resolve globals that were left unresolved by the compiler or the binder.

2. If the object module is processed by the binder, all entry points which are to be added to the known global table must be "marked" using either the -mark or the -allmark binder commands.

3. The main program in an installed library:

   When a library is installed using the inlib command, its main program is called only once, during execution of the inlib command, right after the library is loaded.

   When a library is installed as a global library (/lib/userlib.private), its main program is called once in each process, when the process is being created. Since the DM (or SPM) process is created when the node is booted, the main program is invoked then, before the DM (or SPM) is running. A library need not have a main program, and for global libraries, it is recommended that they NOT have a main program, since this impacts the performance of process creation. Initialization will be discussed further, below.

4. Multiple uses of library procedures:

   Since a library's static data is initialized only once, when it is loaded, and since the library may be used multiple times by different programs, it will in general be necessary for a library to clean up its static data when programs terminate execution. In many cases, the library will have a termination entry that should be called by application programs before they return to the shell. If the application program gets a fault, or neglects to call the termination entry, the library should call it automatically. (For example, any streams which are left open by an application program are closed automatically by the stream manager (which is a global library), when the program terminates. In order gain control at program termination, a library may use the pfm_$static_cleanup. See the programmer's reference manual for further information (actually, I'm not sure this is documented right now). The ideal time to make this call (i.e. to establish the static cleanup handler) is in the first call made to a library procedure by the application program.

5. Initialization of static data:

When a library is installed using the inlib command, its static data are loaded and initialized normally, just as if it were bound with the calling program.

When a library is installed as a global library (/lib/userlib.private), its static data is initialized in a special way:

1) The section named DATA$, which by default contains all static data, is initialized normally at load time (when the node is booted), but is READ-ONLY when the library code is actually executed. This is done to save the overhead of re-initializing the static data in each new process.

2) Other impure sections are allocated address space when the library is loaded, but any static initialization specified in the object module is ignored. Instead, these sections are always initialized to zero in each new process. This is inexpensive, because all newly referenced pages of virtual memory are set to zero by the OS. These pages always occupy the the same range of addresses in each process, but are private to the process. Because they are guaranteed to be zero, the library can determine whether further initialization is needed by declaring a boolean variable which will be guaranteed to be false on the first use of the library in a new process. Note that this variable should also be given a static initial value at compile time, since the static data of a library that is INLIB'ed is NOT initialized to zero. This way, the library will work whether it is a global library or is INLIB'ed.

   The way you get a static data section in Pascal is to follow the VAR keyword by the section name in parenthesis:

```
VAR (my_static_data)
    init_done: boolean := false;
    other_stuff: ...
```

   The way you get a static data section in Fortran is to use named common.

   In C, each global variable is placed in its own static data section.

To summarize, when a library is INLIB'ed, its static data is loaded and initialized normally, and uninitialized data will have random values. When a library is global, its DATA$ section is initialized, but is global, shared, and read-only, whereas its named data sections are read-write, private, initialized to zero, and always occupy the same address range in each process.

6. Multiply defined names. If an external symbol defined by a library is already in the Known Global Table at the time a library is installed (either via INLIB, or global) the new definition will override the old one as long as the library remains installed. In the case of INLIB, the overridden names will be re-instated when the shell that executed the inlib command returns to its caller (e.g. a lower level shell). It is thus possible to redefine system entry points using this mechanism, but this is not generally recommended, because there is no way to reach the real entries while the library is installed -- even from the library itself.

7. Dynamic linking. A limited form of dynamic linking is available. When a library is loaded, any external references which are still unresolved after looking in the known global table are left unresolved, and no message is given. This is true of ordinary programs as well as libraries. If an attempt is made to call one of these entries, the attempt will be trapped, and the symbol will be looked up in the known global table again. If it is now found, the trap will be removed, and the linkage will be established permanently. Thus, a library can reference another library which is loaded later. Note that this works only for procedure and function calls -- it does not work for data references. (When we release the system call that installs libraries, possibly at SR9, this feature will be more useful).

INTRODUCTION

Async fault handling is broken down into two related operations within the kernel: post and delivery.

An async fault is posted by calling PROC2_$TRACE_FAULT with a target process's p2_uid and a fault code (status_$t) to be sent. The post is most frequently made by a user space process; the display manager requesting a quit fault is most common. Less frequently, the kernel posts an async fault be sent to a process; sio line quits and floating point (peb) faults are examples. All kernel-generated async faults that I know about are generated by the terminal helper process. (They can't be generated by interrupt routines or cpu-B-eligible code because the user process OS stack may not be valid and PROC2_$TRACE_FAULT is unwired.)

Async fault delivery is done by FIM_UNWIRED. When an async fault is posted, FIM_UNWIRED is entered with a trace fault. (Implementation details follow.) The trace fault code pushes a diagnostic frame onto the stack containing the status code passed to PROC2_$TRACE_FAULT. It then enters the user space FIM (usually the process fault manager) to perform user space fault handling.

A process that has received an async fault must acknowledge it by calling FIM_$ACKNOWLEDGE. This must be done before any more async faults are accepted by PROC2_$TRACE_FAULT for posting. FIM_$ACKNOWLEDGE is usually called by the user space FIM.

IMPLEMENTATION

N.B.: The term "quit" or "quit fault" used in the variable names and the code is an anachronistic reference to the days when the model of async faults was simpler. When you see "quit", read "async".

The kernel data structures used by the async fault mechanism are indexed by the address space id of the target process. They are:

fim_$trace_sts: ARRAY [asid_t] OF status_$t
the status code to be delivered to the process when a trace fault occurs.

fim_$quit_inh: ARRAY [asid_t] OF char
a flag that indicates the state of async fault handling. A false (00) value indicates that an async fault may be posted for the process; a true value (FF) indicates that the process has an outstanding (unacknowledged) async fault.

fim_$quit_ec: ARRAY [asid_t] OF eventcount_t
a level 1 eventcount that can be used to trigger a process wake up in the event of an async fault. Kernel code that desires to be woken up on an async fault includes this eventcount in the ec_$wait call.

fim_$quit_value: ARRAY [asid_t] OF linteger
the fim_$quit_ec value for the last acknowledged async fault. Kernel code that waits on fim_$quit_ec uses fim_$quit_value+1 as the wake up value.

fim_$deliv_ec: ARRAY [asid_t] OF eventcount_t
an eventcount on which a posting process may wait for the target process to acknowledge a previously posted fault. These ec's are exported to user space via PROC2_$GET_EC.

PROC2_$TRACE_FAULT operates with the proc2 mutex lock held, thereby avoiding problems when 2 processes try to post a fault to the same target at the same time. (It also avoids posting a fault to a target process that deletes itself before the post is complete.)

PROC2_$TRACE_FAULT determines if an async fault is outstanding for the target process. If so, it refuses to post another one and instead returns with the PROC2_$FAULT_PENDING status. If no async fault is outstanding, it sets the status code, the async fault inhibit flag (to say that an async fault is now outstanding), and the trace bit in the process's OS stack SR. It then advances the fim_$quit_ec to wake up the process if its waiting on a quittable event inside the kernel.

When the target process returns to user space, the trace fault occurs after one user space instruction is executed. The trace fault causes entry to FIM_UNWIRED trace fault code. The trace fault code is distinguished from the common FIM code only in that the status code placed in the diagnostic frame is that stored in fim_$trace_sts.

Running in the kernel FIM does not cause the fault to be acknowledged. This means that PROC2_$TRACE_FAULT will not yet allow another async fault to be posted for the target process. Also, the fim_$quit_value is not set to the fim_$quit_ec.value; this allows process-blocking calls such as ec2_$wait_svc to return with a fault-while-waiting status instead of blocking.

The user space fim is responsible for acknowledging the fault when it is capable of accepting another. The user space PM does this when the fault is dispatched. (Dispatching occurs immediately if not pfm_$inhibited, or when the PM's async inhibit counter reaches zero.)

When the fault is acknowledged, FIM_$ACKNOWLEDGE sets the fim_$quit_value to the fim_$quit_ec.value, clears the way for another async fault by setting fim_$quit_inh to false, and advances the fim_$deliv_ec.

USING "FIM_$QUIT_EC"

Fim_$quit_ec is used in various places within the kernel to allow blocking process to wake up on an asynchronous faults. Code that wakes up on the fim_$quit_ec must set the fim_$quit_value to the fim_$quit_ec.value. This is required to prevent spurious wake ups that could occur between the time the fault is posted (eventcount is advanced) and the time the fault is acknowledged.

This requirement is NEW as of 83/09/08. Existing kernel code that used fim_$quit_ec prior to this date has been updated to follow the prescribed protocol.

OS module codes:

NOTE: this list is not "official"

BAT	1	BAT manager
VTOC	2	VTOC manager
AST	3	AST manager
MST	4	MST manager
PMAP	5	PMAP manager
MMAP	6	MMAP manager
MMU	7	MMU manager
DISK	8	DISK manager
EC	9	level 1 eventcounts
PROC1	A	level 1 process manager
TERM	B	(sio line) terminal manager
DBUF	C	disk-buffer manager
TIME	D	time manager
NAME	E	naming server
FILE	F	file manager
IO	10	I/O manager
NETWORK	11	networks
FAULT	12	M68000 and MMU detected faults
SMD	13	screen manager display driver
VOLX	14	volume manager
CAL	15	calendar maint. manager
	16	
	17	
EC2	18	level two eventcounts
PROC2	19	level two process mgr
IMEX	1A	logical volume import/export mgr
OS	1B	os startup/shutdown
VFMT	1C	vfmt input & decode routines
CBUF	1D	circular buffer manager
PBU	1E	peripheral bus unit module
LPR	1F	line printer module
OSINFO	20	OS info supplier
	21	available
MT	22	magtape routines
ACL	23	access control list manager
PEB	24	PEB debugging module
NETLOG	25	network logging mechanism
COLOR	26	color display system
VME	27	vme errors

Notes on the MBX helper process 5/83

1. This is what a mailbox file looks like:

```
 --------------------------------------------
 ¦            MBX FILE  HEADER              ¦
 +------------------------------------------+
 --------------------------------------------CHANNEL 1
 ¦           Channel 1 header               ¦
 ¦------------------------------------------¦
 ¦  Channel 1 client to server data buffer  ¦
 ¦------------------------------------------¦
 ¦  Channel 1 server to client data buffer  ¦
 +------------------------------------------+
 --------------------------------------------CHANNEL2
 ¦           Channel 2 header               ¦
 ¦------------------------------------------¦
 ¦  Channel 2 client to server data buffer  ¦
 ¦------------------------------------------¦
 ¦  Channel 2 server to client data buffer  ¦
 +------------------------------------------+
 --------------------------------------------
 ¦             ....                         ¦
 --------------------------------------------
```

(The size of the buffers are specified by the creator of the mailbox.)

2. The Model

Each Mailbox supports a Server-with-multiple clients model. The mailbox is used to pass messages between the server and his clients (never between two clients directly). The server 'owns' the mailbox and must open it first before any clients can use it.

If the client and the server processes are in the SAME node, they use shared memory to communicate through the file (both map for CO-WRITERS). (Note that the MBX file doesn't have to exist on the same node, just the processes do.) If the client and the server processes are in DIFFERENT nodes, they must use MBX HELPERS to communicate, since two processes on different nodes can't map the same file for CO-WRITERS. (Note that the client needs a helper process even if the MBX file is on the same node as the client.)

3. Here is a picture of server-client communication through a mailbox when the processes are co-resident:

```
                              MBX File
 ----------          +-----------------------+           -----------
 ¦        ¦ put-rec  ¦ client-to-server data¦ get-rec   ¦         ¦
 ¦        ¦-------->  ¦                      ¦-------->  ¦         ¦
 ¦ CLIENT ¦          ¦  ------------------   ¦           ¦ SERVER  ¦
 ¦        ¦          ¦                       ¦           ¦         ¦
 ¦        ¦ get-rec  ¦ server-to-client data¦ put-rec   ¦         ¦
 ¦        ¦ <--------¦                       ¦ <---------¦         ¦
 ----------          +-----------------------+           -----------
```

4. When the Server and Client are not co-resident, each needs a mailbox helper to deliver messages to the other. Here is what happens when a client opens a mailbox to a server:

   a. The client MBX routines get information about the file lock on the MBX file. It must be locked for co-writers (server has opened the mailbox). If it is locked locally, see figure 3 above. If it is not locked in the client's node, continue below.

   b. A channel is opened for the client on his local mailbox, SYSMBX, (which is serviced by his local MBX-helper (let's call him 'MH-C')) and a message is sent to the remote MBX-helper (we'll call him 'MH-S') at his well-known socket in the server's node. The client process then waits on the SYSMBX channel for the open response.

   c. 'MH-S' in the serving node 'helps' the client by doing an open to the target mailbox on behalf of the requestor. He then records information in the channel header about the remote client.

   d. The server in turn reads his mailbox normally (get_rec), sees the open request and (eventually) does a put_rec to his MBX file accepting the open. The MBX library routines, used by the server, 'see' that the addressed channel is really remote and so 'bounce' the msg over over the network to the remote MBX-helper. Note that the server application NEVER KNOWS that the client is remote.

   e. MH-C receives the open response and delivers it through the SYSMBX channel to the waiting client process. The open response is then delivered to the client application as if the open on the target file occurred locally. Actually, what the client has is an open channel that is partly on his local SYSMBX (for reading) and partly in the target file (for writing). Note that the client application NEVER KNOWS that the server is remote and that his mailbox is sort of schizophrenic.

   f. Communication between the client and server now procedes apace, with the client reading from his channel (in SYSMBX) normally (get_rec), while his put_rec's bounce off his SYSMBX mailbox to the remote MH-S. MH-S puts the msgs in the target mailbox, which the server process reads normally, while the server's put_recs bounce off the target mailbox to the client's MH-C which stuffs them in SYSMBX.

   g. Note that all get_recs are local for both the client and server. The MBX-helper is needed only for put_recs.

h. A picture is worth a thousand words:

```
      NODE A

                                SYSMBX file
      ----------          +----------------------+          -----------
      :        : get-rec  : server-to-client data: put-rec  :         :
      :        :<---------:                      :<---------:  MBX    :
      : CLIENT :          :  ------------------  :          : HELPER  :
      :        :          :                      :          :         :
      :        : put-rec  :                      :          :   MH-C  :
      :        :-------->/:                      :          :         :
      ----------        / +----------------------+          :----------
                       /                                     /\
----------------------/--------------------------------------/-------------------
                 /\  /                                /\    /
     NODE B     /  \/                                /  \  /
               /                                    /    \/
              V                 MBX File           /
      ----------          +----------------------+/         -----------
      :        :          :                      /<---------:         :
      :  MBX   :          :                      : put_rec  :         :
      : HELPER :          :  ------------------  :          : SERVER  :
      :        :          :                      :          :         :
      :  MH-S  :put_rec   :client_to_server data : get_rec  :         :
      :        :--------->:                      :--------->:         :
      ----------          +----------------------+          -----------
```

DIRECTORY STRUCTURE

```
    |--------------------------|
    |         header           |      directory configuration information
    |--------------------------|
    |       linear list        |      sequentially used directory entries
    |--------------------------|
    |       info block         |      ACL manager's intial ACL description block
    |--------------------------|
    |      hash threads        |      Pointers to linked lists of hashed entries
    |--------------------------|
    |         entry            |      Holding blocks for hashed entries
    |         blocks           |              and/or link text
    |--------------------------|

       Directory Overview
            (dir_t)
    total length - 2 full segments
        (name.pvt.pas)
```

```
      |--------------------------|
 0    |  version  |    M B Z     |      info block version number
      |--------------------------|
 2    |    info block length     |      total length of info block
      |--------------------------|
 4    |  info block hdr length   |      length of the info block header  (8)
      |--------------------------|
 6    |          M B Z           |      reserved for future use
      |--------------------------|
 8    |    default acl uid       |      uid of acl to be applied to directories
 A    |    for directories       |         catalogued in this directory
      |--------------------------|
 C    |     default acl uid      |      uid of acl to be applied to files
 E    |       for files          |         catalogued in this directory
      |--------------------------|
 10   |    24 unused bytes       |      reserved for future use
      |--------------------------|

        Directory "info block"
            infoblk_hdr_t
       total length - 48 bytes
          (name.pvt.pas)
```

0	entry name	32 bytes of entry name
20	unused	reserved
22	unused	reserved
24	unused	reserved
26	name len | entry type	name len - # of useful characters in entry name entry type - 0 = not in use 1 = name/uid pair 3 = name/link-data pair
28	4 words of entry data (either UID or link text description)	if entry type = 1, this is the UID entry type = 3, this describes the link text: link text len block that holds link text chars 1-144 block that holds link text chars 145-256 reserved for future use

Directory "entry"
dir_entry_t
total length - 48 bytes
(name.pvt.pas)

0	next block number	forward thread for doubly linked list
2	prev block number	backward thread for doubly linked list
4	use count | block type	use count - # of used entries in this block block type- 0 = not in use - 1 = hash block with 3 dir entries - 3 = link text holding block
	entry block data	either 3 dir entries or up to 144 chars of link text

Directory "entry block"
entry_block_t
total length - 150 bytes
(name.pvt.pas)

```
    |------------------------|
 0  |        version         |    version number of this directory (1)
    |------------------------|
 2  |       hash value       |    # of hash threads used for entry name hashing
    |------------------------|
 4  |       list size        |    # of entries configured into linear list (18)
    |------------------------|
 6  |       pool size        |    # of entry blocks in this directory (429)
    |------------------------|
 8  |   entries per block    |    # of entries that fit in an entry block (3)
    |------------------------|
 A  |   high block number    |    # of the highes entry block used so far
    |------------------------|
 C  |   free block thread    |    # of the first block on the free block list
    |------------------------|
 E  |         unused         |    reserved for future use
    |------------------------|
10  |         unused         |    reserved for future use
    |------------------------|
12  |         unused         |    reserved for future use
    |------------------------|
14  |         unused         |    reserved for future use
    |------------------------|
16  |      entry count       |    # of entries currently catalogued in this dir
    |------------------------|
18  |     maximum count      |    # of entries this directory CAN hold (1300)
    |------------------------|
```

Directory "header"
first part of dir_t
total length - 26 bytes
(name.pvt.pas)

Notes on directories:

```
   1. To add an entry to a directory:
          (a) Look for an unused entry in the linear list.
              If you find one, use it and you're done.
          (b) Hash the name you want to add.
          (c) Get the hash thread for the specified hash value
              and call that value the found block.
          (d) If the found block number is 0 then we need a new entry block, so:
                (i)   See if there are any blocks threaded through the
                      free block list and if so, take one of those.
                      Otherwise, bump the high block number and use that.
                (ii)  Initialize the newly obtained block, add it to the
                      end of the apprpriate hash chain, add the new entry
                      as the first entry in the new entry block and you're done.
          (e) If there is an unused entry in the found block,
              use it and you're done.
          (f) Change the found block value to the number in the current
              found block's NETX BLOCK field and goto step (d).
```

2. The searching rule for a directory is:
   (a) look in the linear list.
   (b) hash the name you're searching for.
   (c) follow the hash thread for the specified hash value to the first entry block with that hash synonym.
   (d) search all (3) of the entries in the found entry block
   (e) follow the "next block number" in the found entry block to get a NEW found entry block. If the next block number is zero, then return NOT FOUND.
   (f) goto step (d) with the newly found block.

CRASH ANALYSIS COMMANDS

Here are the first three things you will do. The "ma" (map) command maps the dump and gives its length and starting location. (The dump is mapped for read/write access, no extend.). The "da", "am", and "st" commands are described below. You may want to start by reading their descriptions.

```
  $ db
  ! ma dump.425.04.07
  134000 bytes mapped at 2F8000

  ! da
  System built on Tuesday, March 22, 1983   3:13:09 pm (EST)

  ! am map.425.04.07
     ———               System built at 1983/03/22 15:14:02 EST (Tue)
  mapped mode entered
  Current asid = 1

  ! st
   ...
```

---

a7 [<value>]   set SP at time of dump

A7 must always be saved or remembered before taking a dump, since it gets clobbered. This command will set the SP displayed by the DR command to the given value. If no value is entered, the contents of 0E003FC (physical 1003FC) are used. (This is where crash_system saves a7 before entering the prom.)

---

a{b|w|l}[e] <sym>   access via symbol name

These are special flavors of db's 'a' command that take a symbol name rather than a hex address. The suffixes 'b', 'w', 'l' stand for byte, word, long. 'e' can also be appended if you specify a procedure name and want its ecb instead of its entry point.

```
  ! al os_stack_base
  E31CEC:      0
  E31CF0:   E4D400
  E31CF4:      0
  E31CF8:   EA8800
  E31CFC:   EA9400
  E31D00:   EA9C00
  E31D04:   EAA400
  E31D08:   EAB000 /

  ! ale ast_$touch
  E29DC4: 4EF900E0
  E29DC8: 182400E2 /
  !
```

am <path>   load Aegis Map

This tells db to load a map of aegis as produced by bind_aegis. Example:

```
  ! am //hifi/sau/aegis.map
    -------               System built at 1983/03/24 13:17:08 EST (Thu)
  mapped mode entered
  Current asid = 2
```

The first line printed indicates when the system was built (this is the first line of the map file); the second line is printed if a dump (or, actually, anything) has been previously mapped with db's map command; the third line indicates the current address space (procl_$as_id).

If you are looking at a dump, the map should, of course, correspond to the version of aegis in the dump. To determine this, compare the build time printed by the 'am' command (see below) with the build time shown by the 'st' command. These times should be within 15-20 seconds of each other; if they are not, you've got the wrong map. If the 'st' command says "Build time not available", which it will for any aegis built before 02/28/83, then you should perform some reasonability checks if you have any doubts as to whether or not you have the correct map.

Note 1

In systems built after 02/18/83 the clockh of the build time is stored in BUILD_$TIME, which is at 0E00800, wired, and should always be in the dump.)

Note 2

The 'am' command can be used even if you haven't mapped a dump. The 'wh' command can then be used to look up symbols in the map. This is useful, for example, if you have crashed node next to one on which the map can be examined.

as [<asid>]   set/display current asid

This command is useful only if you have to look in the private address space of a process other than the current process. For example, if process 9 (user process 1) is current but you want to look at the stack of user process 2, you will need to set the asid to 3. (His stack, of course, may not be in the dump.) If you don't know the asid of a process, dump its pcb with the 'dp' command.

```
  ! as
  current asid = 1

  ! as 2

  !
```

aste <addr>|<astex> print contents of aste

The 'aste' command dumps an aste (active segment table entry) identified either by astex (aste index, starting at 1) or by an address. Example:

```
! aste 2

aste 2 at EDC0B0: //HIFI/SYS/NET/PAGING_FILE.4BA
fsegno = 1, link = 1 (= EDC000), con_ctrl = 0 (none)
permanent, not immutable, no file_trouble, not in_trans, hold_count = 1
vtoce_addr = 8000039F, fm_addr =        0, sys_type = 0
file map not modified, blocks_delta = 0, cur_len = 8001
gtms = false, dtm_flag = true, grace_flag = false, volx = 15, npr = 28
dtm= Monday, April 4, 1983   7:27:32 pm (EST)
type= uid_$nil, acl= acl_$nil

 0: wired=1 resident, ppn=442              14: wired=0 resident, ppn=6C5
 1: wired=1 resident, ppn=443              15: wired=0 resident, ppn=78C
 2: wired=1 resident, ppn=444              16: wired=0 resident, ppn=6CA
 3: wired=1 resident, ppn=445              17: wired=0 resident, ppn=4ED
 4: wired=1 resident, ppn=446              18: wired=0 resident, ppn=6EB
 5: wired=1 resident, ppn=447              19: wired=0 resident, ppn=7CA
 6: wired=1 resident, ppn=448              20: wired=0 resident, ppn=788
 7: wired=1 resident, ppn=449              21: wired=0 resident, ppn=457
 8: wired=1 resident, ppn=44A              22: wired=0 resident, ppn=458
 9: wired=0 resident, ppn=6D1              23: wired=0 resident, ppn=45B
10: wired=0 resident, ppn=6D9              24: wired=0 resident, ppn=45C
11: wired=0 resident, ppn=6DA              25: wired=0 resident, ppn=45D
12: wired=0 resident, ppn=4F2              26: wired=0 resident, ppn=6E6
13: wired=0 resident, ppn=6C7              27: wired=0 resident, ppn=6DD

Next (cr), link (l) or done (q)?q
```

If you type return to the above prompt, the next sequential aste is displayed. If the aste has a non-zero hash thread, you can display the next aste on the hash thread by typing "l". The aste command will bitch if you give it an unreasonable astex or an address outside the ast.

d460

This prints hardware information unique to DNx60 processors:

```
! f460
This dump was taken by CPIO (not CPU)
  Current hardware region registers:
   RAR(00-07):  C0200C00  80272C00        0        0        0        0        0
   RAR(08-0F):         0         0        0        0        0        0        0
   RAR(10-17):         0         0        0        0        0        0        0
   RAR(18-1F):         0         0        0        0        0        0 8029F800  C02
  CPU state as saved by CPIO:
    CPU PC:      3256,  CPU SR:  82A2700,  CPU USP:   875258
    D0-D7:    82A0004  FFFFFFFF     13AA      190  2020000C  F9257464      400    20A0C
    A0-A7:     20A852    20A852     BC00     9090      BC00      8401   200130    20A83
```

da [<clockh>] display date

The long word entered is interpreted as a clockh_t and displayed. If you do not enter a time, the build time of the system in the dump is displayed.

```
 ! al build_$time
 E0082A: 171E81FD /

 ! da 171e81fd
 Tuesday, March 22, 1983   3:13:09 pm (EST)

 ! da
 System built on Tuesday, March 22, 1983   3:13:09 pm (EST)
```

Note 1 This command can be used even if a map of aegis has not been loaded. It can thus be used when deciding what map to load.

db   enter/leave debug mode

This command (which won't appear in the help list) toggles an internal variable that controls the display of certain debugging information, particularly during the process of converting mapped addresses into their dump-relative equivalents. You should normally have no need of this command, but if you are getting strange results or unexpected vtop misses or access violations, turning on debug mode may help isolate the problem.

dct [<index>]   display dcte(s)

One or all (if <index> is omitted) of the dctes are displayed. Each dcte contains information about a particular disk or ring controller on the system. Example:

```
 ! dct 0

 DCTE for ctype 0 (winchester) at E2F4A8  (cnum=0):
    ctlr status = 0
    lock_no=0015, iomap_base=0040, vector_ptr=240, csrs_ptr=FF9C00
    blk_hdr_ptr = E2F400   PAGE_INIT
    int_entry   = E2F584   DCTE4 + 0
    int_routine = E3469A   WIN_$INT<e>
    int ec at 274EBA:     114502 E2F4BC E2F4BC     DCTE.WIN + 14
```

df <address>   display fault diagnostic record

Just like an "fst -a", except you have to supply the address of the fault record. Usually, you won't know where a fault diagnostic record is. One technique is to enter physical mode and search the mapped dump for occurences of DFDF:

```
$ db

! ma dump.144b.01.17
200400 bytes mapped at 2F8000

! s 2f8000 2f8000+2003fe 0dfdf:w

3066A0: DFDF
338D32: DFDF
392420: DFDF
424804: DFDF
 ...

! df 424804
Fault Diagnostic Information
Fault Status    = 9B450000:
status 9B450000
Fault occured in supervisor due to user program error.
Access Addr     = FFF0246E
IR              = 0014
Acc. Info       = 4E56
User Fault PC   = 488148C1
D0-D7:  00000000 64BA2000 00000000 00000000 00000000 00000001 00020000 388E0000
A0-A7:  00200000 388E0000 55480000 64900000 64940000 649A2F0D 42A72F08 2A680006
Supervisor ECB = 2803242E
Supervisor SR  = FFF4
Supervisor PC  = 264A528A
```

Most of the DFDF's you find will not be real diagnostic records, and df will display junk. The one above, for example, has very few reasonable numbers and should be ignored.

---

dpt   disable PTT (remove from address space)

The PTT, mapped at 700000, is removed from the address space. Subsequent references to virtual addresses in the range 700000-7FFFFF will reference user space addresses.

---

dp [<pid>]   display pcb (first ten if no pid entered)

The 'dp' command displays the contents of a pcb (process control block) in nice easy to digest format. If "pid" is not specified, the pcb's of all bound processes are dumped. Example:

```
! dp 9
E2FB82:  PID = 9, ASID = 2     *** USER PROCESS 1 ***
   LOCKS HELD: none
   STATE: bound waiting on 3 eventcounts:
      E32890:             4 EBEF4A EBEF4A    SOCK_$SOCKET<d> + 80
      E33396:  392138772 EA9304 EBEF5A    TIME_$CLOCKH_EC<d>
```

```
        E30550:           0 EBEF6A EBEF6A   FIM_$QUIT_EC<d> + 18
      REMAINING TIMESLICE = 764      NEXT = E2FA6A, PREV = E2FA6A     STACK PTR = EBEF36
      CLOCKH_T AT START OF LAST WAIT = 175F58D5     PRIORITY = 3      SP's=277B04/EBEF90
  !
```

Note 1

If a lock is displayed as:

```
  LOCKS HELD: win_$lock(W)
```

it means that the the process is waiting to acquire the lock; someone else is actually holding the lock. (db notices that the process is waiting on an eventcount in LOCK_$EVENT_LISTS.)

Note 2

"STACK PTR" is a pointer to where the USP and SP were saved on the process's stack. The saved USP and SP are displayed following "SP's". For the current process, all three of these fields should be ignored; the current SP is in the registers saved by MD (if you're lucky).

Note 3

Examination of "CLOCKH_T AT START OF LAST WAIT" is sometimes useful in determining which processes have run recently.

Note 4

In the interpretation of the eventcounts a process is waiting on, the first field (the count) is in decimal.

Note 5

One of the first things you should do in analyzing dumps, particularly those of obscure cause, is dump all the pcbs. This will tell you who was running (current), who was ready to run, who ran recently, and who was blocked and why. After looking at a few dumps, you will recognize which processes are in their normal quiescent states and which have had their cages rattled. See also the RL command.

---

dr   display registers at crash

This command dumps the last set of registers saved by MD. Note that this is NOT a shorthand for "d d0 a7 8:1", which will show meaningless information.

```
  ! dr
  d0:          0  FFFFFFFF       13        0       10        0        1     8000
  a0:        7D8    E00294   E002E2   E2FA10   E00242   FFB001   E00200   140000
  !
```

Note 1

The A6 and A7 shown above are typical of the registers saved following a reset command; they should be ignored. (Usually only A7 has been clobbered.)

ds   display disk statistics

The "ds" command dumps WIN_$CNT, SM_$CNT (if the system has a storage module), and DISK_$ERROR_INFO -- information about the most recent disk error.

```
! ds

Winchester I/O:  total=  18441   reads=  10338   writes=    8103
   Not ready          0         Contrlr busy       0
   Seek error         0         Equip check        0
   Drive time out     0         Overrun            0
   CRC errors     0

No disk error info has been recorded.
!
```

dv <addr>   convert db address to virtual address

If you have had to go into physical mode (see "p" command) to look at something, the "dv" command can be used to translate physical addresses back into their virtual equivalents (if one exists). Examples:

```
! dv 32c188
32C188 = 0/E2F988   PCBS<d>

! dv 69
addr not part of dump
```

The number preceeding the "/" is the asid of the address.

dvt   print disk volume table

The "dvt" command dumps the entire disk volume table. Use this to see what volumes were mounted at the time of the dump, the state of the volumes, etc.

```
! dvt

DVTE for lvolx 1 at E33F4E: mounted
   unit = 0, dtype = 0, dcte ptr = E2F0A8   DCTE.WIN + 0
   b_per_vol = EB67 (60263), b_per_trk = 12, t_per_cyl = 3, curr_cyl = 1D3
   lv_base = 1, owner pid = 1, volume uid = 11EA304C.10000105

DVTE for pvolx 2 at E33F72: free

DVTE for pvolx 3 at E33F96: free

DVTE for pvolx 4 at E33FBA: free

DVTE for pvolx 5 at E33FDE: free
```

```
  DVTE for pvolx 6 at E34002: mounted
     unit = 0, dtype = 0, dcte ptr = E2F0A8   DCTE.WIN + 0
     b_per_vol = EB68 (60264), b_per_trk = 12, t_per_cyl = 3, curr_cyl = 0
     lv_base = 0, owner pid = 1, volume uid = 11EA2E85.00000105
  !
```

---

ept   enable PTT into the address space

The PTT is mapped into the address space at 700000. This also enables the PT command.

---

ff [<addr>]   try to find stack frame in addr - addr+1024

This command attempts to find a reasonable looking stack frame in 1K bytes starting with the specified address. If it finds one, it then calls the trace stack command to display the stack from that point. If you don't like the resulting chain of stack frames, type "ff" again with no argument. The search will be restarted just after (above) the first frame found.

```
  ! ff 0ea9000

  stack frame at: EA9006...
     previous frame: EA906C   PROCESS 4 STACK - 394
     ecb           : E31CC8   EC_$WAITN<e>
     unit list     : 0
     caller' db    : E340B8   WIN_$RD_WRT<e> + C
     pc for return : E2FE6C   EC_$WAIT<d> + 24
     argument 1    : EA9028   PROCESS 4 STACK - 3D8
     argument 2    : EA9034   PROCESS 4 STACK - 3CC
     argument 3    : 200E1
  Continue trace back? n
  !
```

Note 1

If you hit on an old chain of stack frames, the trace back will mostly likely end up a garbagey stack frame, access violation, etc. Several "ff" commands are usually needed before finding a reasonable chain that reaches all the way back to top of the processs'es stack.

---

gd [<unit>]   get (pbu) dcte

This command will dump the current state of a PBU dcte (not to be confused with disk/net dcte's). This command is only useful on systems that have a pbu; particular dcte's of interest are those of the tape (3) and storage module (4). If no unit number is specified, all the PBU dctes are dumped.

```
  ! gd 0
```

```
DCTE 0 AT E3B946:


int_addr:     E3BC00              Unit 0
asid:         0000
pid:          0
flags/eoi:    0060   (ec not advanced, int_addr not set)
base_unit:    0
uint_addr:    000000
ec_addr:      000000
ec:                  0, E3B95A, E3B95A
timer:               0, E3B966, E3B966
usp:          000000
csr_ppn:      0
csr_ptr:      000000
iomap_base:   0
iomap_start:  0000
iomap_end:    0000
mem_ptr:      000000
mem_len:      0000
mem_iova:     0000
!
```

---

ha <hi> <lo> | <addr>   hash uid to astex

The "ha" command will accept a uid or the address of a uid and calculate the index of the start of the ast hash thread for that uid. This is useful when you have the uid of an object and want to examine what the ast says about the current state of the object.

```
! wh network_$paging_file_uid
network_$paging_file_uid at E2BA10

! ha 0e2ba10
hashs to 48, first astex = B

! ha 1790BA98 800003D4
hashs to 40, first astex = 8A
```

---

le   list system error log

If system error logging is turned on, the le command displays the contents of the mapped log file at the time of the crash.

```
! le
Thursday, October 20, 1983
   5:32:15 am (EDT)  system startup
   1:23:28 pm (EDT)  crash on Tuesday, October 20, 1983   1:19:21 am (EDT)
      crash status - manual stop: type G<ret>G *+2<ret> to continue  (OS/terminal manage
   1:23:28 pm (EDT)  system startup
   4:25:34 pm (EDT)  system shutdown
```

```
     4:25:55 pm (EDT)  system startup
     6:19:11 pm (EDT)  system shutdown
     6:19:30 pm (EDT)  system startup

  Error totals:
     system startups     4
     disk errors         0
     eccc errors         0
     parity errors       0
     system shutdowns    2
     system crashes      0
```

lvl <addr>   print logical volume label

This will interpret and display a logical volume label starting at <addr. This command can also be used after rwvol has been used to read the lv label.

m   enter mapped mode

In mapped mode, all addresses that you feed db are interpreted according to the state of the mmu when the dump was taken. In addition to normal virtual addresses, certain (mapped) hardware addresses can be entered. These are:

FFF800-FFF9FE	IOMAP
700000-7FFFFF	PTT
FFB404-FFB407	MMU status register (Apollo_1 only)
FFB40A-FFB40B	MMU bus status register
FFB800-FFF7FF	PFT

Certain other pages (e.g., trap page, debugger page) can be referenced by both their physical and virtual addresses.

Note 1

Mapped mode is automatically entered by the 'am' and 'ma' commands once a dump has been mapped and a map loaded.

Note 2

It is possible for the mmu (ptt, pft, etc.) to be messed up in a dump. This can cause the mapped-to-physical address translation mechanism in db to cause access violations. Since db's fault handler immediately tries to use the same mechanism, an infinite loop can result. To prevent this, db briefly leaves mapped mode when there's a possibility of a fault being generated. If there IS a fault, you will see the fault message and be left back in physical mode. Just type 'm' again to continue. (This hack will be fixed up sometime.)

Note 3

In a dump taken from a floppy, only the first 1K entries of the pft will be present (since only the first 1M of memory will fit on a floppy).

mm <addr>|<ppn>   print mmap entry

The "mm" command shows you the current state of a physical page of memory. Of particular interest is the astex, which will indicate the aste of the object to which the page belongs. Example:

```
 ! mm 500
 E41C00: ppn 500: C4B50117  in_use, astex=B5, daddr_h=0, pttx=117
   avail=true, null=false, rmod=false, usedp=false, usedr=true
 Next (cr) or done (q)?
 E41C04: ppn 501: C430020E  in_use, astex=30, daddr_h=0, pttx=20E
   avail=true, null=false, rmod=false, usedp=false, usedr=true
 Next (cr) or done (q)?q

 !
```

mr   print mem_rec (eccc or parity error log)

The contents of the memory eccc or parity record are displayed. (Info is the same as that displayed at the end of a netstat -l.)

```
 ! mr
 A total of 0 parity errors were detected.
```

ms <args>   mapped search (just like md's 's')

This works just like md's "s" command, except that you specify dump-relative addresses. (There are bugs here.)

mst [<asid>|<msteaddr>] print mst for an asid (0 for gbl, omit for curr)

This command will dump the mst (mapped segment table) for a given asid. If omitted, the current asid is used (see "as" command). The "mst" command will also accept an address that is in some part of the mst. It will figure out which asid corresponds to that address and dump the entire mst for that asid.

```
 ! mst 3
 -- MST is at EC8000 --
 MST for asid 3.  1st MSTE is at: ECBC00

     VA Range     Obj Start   UID/Pathname

 200000 - 28FFFF          0   1784E56D.70000192
 290000 - 297FFF          0   /SYS/NODE_DATA/DM_MBX
```

```
298000 - 29FFFF         0   /COM/SH
2A0000 - 2BFFFF     90000   1784E56D.70000192
2C0000 - 2C7FFF      8000   1784E56B.30000192
2C8000 - 2D7FFF         0   /COM/DB
2D8000 - 2F7FFF     B0000   1784E56D.70000192
BC0000 - BCFFFF         0   /GMS/MEMOS
BD0000 - BDFFFF         0   /NOS
```

---

mste <addr>   print the mste for a particular virtual address

The "mste" command is similar to the "mst" command, but only the mste corresponding to the given virtual address is dumped. The current asid is used. Addresses in the global A or B areas can be specified without switching to asid 0.

```
! mste 298000
mste at ECBD30:
298000 - 29FFFF         0   176930FB.300003D4   fsegno=0, ext_ok=false
access=rx, guard=false, pastex=78, locx=10000001 (ta_cnt=4, lcl, volx = 1)
```

---

p   enter physical (normal) mode

Physical mode (as opposed to mapped mode, which see) is the normal state of affairs in db. Addresses fed to db are interpreted as referring to the address space of the process in which you are running db.

It is occasionally useful to enter physical mode when analyzing a dump in order to search the entire dump for some pattern. For example, if you are looking for all fozzards that have ppn 425 in their back pocket, you could do the following (don't expect such terse output as is shown here!):

```
! p

! s 2f8000 2f8000+134000 425:w      (using the values printed by the 'ma' command)

2FA68A:  425

! m                                 (just so you don't forget)

! dv 2fa68a                         (convert db addr back to virtual addr)
2FA68A = 0/FFBA8A

! wh 0ffba8a
   PFT + 28A                        (as you might expect)

!
```

Physical mode is also useful if a page in the dump has useful information but was not in the mmu at the time of the dump (see next command).

---

pf <ppn>|<addr>   display pft entry

This command displays a pft entry given either a ppn or an address in the pft.

```
  ! pf 500

  pfte for 500 at FFCC00:  06630519  asid=3, access=wr, xsvpn=3
    eoc=false, pmod=false, used=false, global=false, link=519

  Next (cr), link (l) or done (q)?l

  pfte for 519 at FFCC64:  017EF5E7  asid=0, access=swrx, xsvpn=E
    eoc=true, pmod=true, used=true, global=true, link=5E7

  Next (cr), link (l) or done (q)?l

  pfte for 5E7 at FFCF9C:  08636500  asid=4, access=wr, xsvpn=3
    eoc=false, pmod=true, used=true, global=false, link=500

  Next (cr), link (l) or done (q)?q
```

---

pt <pttx>   display ptt entry

The "pt" command displays the ptt entry for a given ptt index (pttx). Example:

```
  ! pt 241
  790400 (2F8682) = FC38

  !
```

The first address is where the entry would appear in a real ptt. The virtual addresses corresponding to the pttx in the above example would be x90400 (90400, 290400, E90400, etc.). To see what the ptt entry is currently pointing to, display the pft entry pointed to by the ptt entry (ignore the top 4 bits, e.g., C38 in the example). The number in parens is where the ptt entry is stored in the dump, in case you want to poke around in physical mode. Note that in physical mode the ptt has only one entry for every 1K entries in the real pft, e.g., the ptt entry at physical location 2F8684, pttx 242, would appear in the real ptt at 790800.

To use this command, you must first "enable" the PTT with the EPT command.

---

pv <ppn>   convert ppn to virtual address

The 'pv' command shows you what virtual address is currently associated with a physical page from the dump. Examples:

```
  ! pv 425
  425 = 0/E08C00   PMAP_$GROW<p> + A4

  ! pv 4be
  ppn 4BE is not in use, but is at 32B800
```

The number preceeding the "/" is the asid of the address.

In the second example, the ppn was not in the mmu at the time of the dump (e.g., maybe someone was doing i/o to or from it). In this case, db prints the address where the page can be found in physical mode (see 'p' command).

---

pvl <addr>   print physical volume label

This will interpret and display a physical volume label starting at <addr. This command can also be used after rwvol has been used to read the pv label.

---

rl [check]   print ready list

This is like the DP (display PCBs) command except that the PCBs are displayed in the order in which they appear in the ready list, starting with the current process. If you give the RL command any argument, the ready list is just checked for correct order.

---

st   display status at crash

This is usually the first thing to do after loading the map of aegis. Example:

```
  ! st

  Crash occurred on Monday, April 4, 1983   1:40:26 pm (EST)       node =   105

  System built on Thursday, February 14, 1980   8:07:18 am (EST).

  Machine id = 0
  System configured with 1024K of memory

  Crash status:  120020: supervisor fault while resource lock(s) set  (OS/fault handler)
  ECB: E2FA6A

  current process: 1

  E2FA42:  PID = 1, ASID = 1     *** DISPLAY MANAGER ***
     LOCKS HELD:  acl_$lock
     STATE: tse_onb bound current
     REMAINING TIMESLICE = 7749       NEXT = E2FAE2, PREV = E2FA6A     STACK PTR = E4DC92
     CLOCKH_T AT START OF LAST WAIT = 175F8FFC      PRIORITY = 16      SP's=FFFFFFFF/E4DCDC

  current mmu status:          BE0000
  bus status: FFB2     cpub_status: 80110007 remote node failed to respond to request  (OS

  last miss handled by cpub: AEBE0000 (miss, sup data read)

  last state saved by md:
```

```
d0:          0  FFFFFFFF        13         0        10         0         1      8000
a0:        7D8    E00294    E002E2    E2FA10    E00242    FFB001    E00200    100100
!
```

ts <pid or addr>   traceback stack

The "ts" command shows you where a process is, given either its pid or a valid SB. If you specify the pid of the current process, the current SB in the registers saved by MD is used. For other processes, the starting SB is taken from what STACK PTR is pointing at (the second address following "SP'S= in a pcb display). Example:

```
! ts 8

stack frame at: EBFF24...  (non-standard stack frame)
   previous frame: EBFF7A   PROCESS 8 STACK - 86
          EBFF28 : E035FE   DISPATCH<p> + 8
          EBFF2C : E31CE0   EC_$READ<e> + C
          EBFF30 : E0AB5A   EC_$WAITN<p> + 10A
          EBFF34 : 986
Continue trace back?

stack frame at: EBFF7A...
   previous frame: EBFFEC   PROCESS 8 STACK - 14
   ecb           : E31CC8   EC_$WAITN<e>
   unit list     : 0
   caller' db    : E30FF8   NETWORK_$LOCATE<e> + C
   pc for return : E2FE6C   EC_$WAIT<d> + 24
   argument 1    : EBFF9C   PROCESS 8 STACK - 64
   argument 2    : EBFFA8   PROCESS 8 STACK - 58
   argument 3    : 300E0
Continue trace back?

stack frame at: EBFFEC...
   previous frame: 0
   ecb           : E30EF4   NETWORK_$MONITOR<e>
   unit list     : 0
   caller' db    : E2F988   PCBS<d>
   pc for return : E036DE   INIT_STACK<p> + 2C
   argument 1    : 0
   argument 2    : 9000
   argument 3    : 16C4929E

!
```

Note 1

The first two stack frames for a waiting process will always be the dispatcher and EC_$WAITN. "non-standard stack frame" is printed when db notices that a non-standard calling sequence was used.

Note 2

If you want to trace a stack back into user space, you should first set the asid appropriately.

Note 3

If you do not have a valid SB, use the "ff" command.

===

uid <hi> <lo> | <addr> interpret uid

The "uid" command will tell you all it can find out about a uid. You can either specify the address of a uid or the uid itself as two hex numbers. Examples:

```
  ! ui 174F38C7 90000192
  /SYS/DM/DM
  ! wh network_$paging_file_uid
  network_$paging_file_uid at E3103E

  ! ui 0e3103e
  11EA3A0D.50000105

  ! ui 0e0cfda
  name_$canned_root_uid
```

Note 1

A name_$gpath is attempted on the uid, so if the network is flakey or down, there will be a significant pause during the Bls Memorial Timeout period. This will also occur during other commands that invoke the "uid" command internally.

===

vd <addr>   convert virtual address to db address

This command will show you where in the mapped dump a certain virtual address is to be found. Example:

```
  ! vd 0e2f988
  E2F988 = 32C188
  !
```

===

ve <addr>   print vtoce at <addr>

This command is useful when investigating disk/vtoc/file related problems and you want to see what dbuf has in its back pocket. Note that the first vtoce will appear 4 bytes beyond the address of one of pages in dbuf_blks. Example:

```
  ! wh dbuf_blks
  dbuf_blks at EC0000

  ! ve 0ec0004

  vtoce 0 at EC0004: version = 0, sys_type = 0
  con_ctrl = 0 (none), permanent, not immutable, no file_trouble,
  object uid= 16C4929E.B0000105
    type uid= object_file_$uid
     acl uid= 16E73FA1.40000105
     dir uid= 167F3ACD.60000105
  cur_len = 296792, blocks_used = 293, ref_cnt = 0
  dtu= Thursday, March 17, 1983   5:13:40 pm (EST)
  dtm= Thursday, March 17, 1983   5:13:40 pm (EST)
    0:      ADF     AE2     AE5     AE8     AEB     AEE     AF1     AF4
    8:      AE0     AE3     AE6     AE9     AEC     AEF     AF2     AF5
   16:      AE1     AE4     AE7     AEA     AED     AF0     AF3     AF6
   24:      AFA     AFD     B01     B04     B07     B13     B16     AFB
  fm2:      AFE    1FBA       0
  Next (cr) or done (q)?
```

Note 1

This command can also be used to look at a blocks read by online rwvol.

vm   verify mmu (against mmap)

The "vm" command steps through the mmap, pft, and ptt in the dump and verifies that they are consistent with one another.

```
  ! vm
  ppn 414: more than one eoc in chain
  ppn 414: mmap 417 wrong pttx is 15 sb 12
  ppn 414: more than one eoc in chain
  ppn 414: mmap E66 wrong pttx is 15 sb 12
  ppn 414: pft has bad chain pointer
  ppn D4F: mmap E8F wrong pttx is 163 sb 15B
  ppn D4F: pft has bad chain pointer
  pttx: 334 mismatch. is DD7 sb EF8
  pttx: 336 mismatch. is D9F sb B6
  pttx: 33F mismatch. is E20 sb 0
```

Note 1

At the current time (SR6.0 and earlier), Aegis does not bother remove the pages of (nonexistent) second display memory from the mmu, although it does release the corresponding mmap pages. For this reason, the "vm" command ignores errors involving ppns 100-180.

vp <addr>   convert virtual address to ppn

The 'vp' command converts a virtual address from the dump into the ppn corresponding to the address when the dump was taken. Examples:

```
  ! vp 0ec0000
  EC0000 = 402
```

```
! vp 200400
mmu_$vtop - mmu miss  (OS/MMU manager)

!
```

In the second example, there was no entry for 200400 in the mmu when the dump was taken.

---

vv <addr> <data>   verify vmtest page

On systems with flakey memory or disk hardware, this command is useful to pinpoint vmtest failures that result in system crashes (e.g., memory parity, disk data checks, etc.) The page at the specified address is scanned using the given starting data and vmtest's increment/decrement values. Note: the page of interest may well not be in the mmu, so you may have to resort to a db-relative starting address (p mode).

```
! vv 348c00 348c00
  offset 0 s/b 0034C000, is 00000000
  offset 4 s/b 0034C004, is 1A98ED9B
!
```

---

wh[p|d|e] <sym or addr> look up [proc|data|ecb] or address in aegis map

The 'wh' command takes either a symbolic name or a virtual address, the latter starting with a numeric, as always. When looking up a procedure, the suffixes "p", "d", "e" can be used to select a particular definition of the symbol: procedure, data, or its ecb. When finding an address, db appends "<p>", "<d>", "<e>" the the symbolic name to indicate where in the map the symbol was found. Examples:

```
! wh pcbs
pcbs at E2F988

! wh 0e12345
   FILE_$SET_LEN<p> + 7

! wh mst_$touch
mst_$touch at E049B4

! whd mst_$touch
mst_$touch at E30C32

! whe vtoc_$allocate
vtoc_$allocate at E3350C

!
```

INTERVAL TIMER IMPLEMENTATION

Existing timer facilities

In aegis there are two mechanisms which provide timer facilities to user processes. One mechanism uses the clock process to implement its timer functions. The corresponding user callable procedures are implemeted in time.pas and include time_$wait, time_$advance and time_$cancel. The second mechanism uses the terminal helper process in conjunction with the eventcount time_$clockh_ec. The user callable procedures using this mechanism are implemented in time_$unwired.pas and include the procedures time_$alarm and time_$free_asid. The first mechanism can handle time specifications in the order of microseconds whereas the second mechanism can handle it only in the order of seconds. The advantage of the second mechanism is that it much more efficient in cpu time consumption.

Background information on the clock process

The timer interrupt handler handles interrupts from three timers and depending on which timer went off it does the following.

o If the interrupt was from the time_of_day clock then it advances time_$clockh_ec. (happens every 1/4 th of a second). The terminal process suspends itself on this eventcount and is awakened to complete the timer related processing required by user processes.

o If the interrupt was from the 8 micro second timer for time slice end it calls procl_$end_time_slice and and procl_$int_exit which reorder the ready list, set the timer and dispatch a new process. procl_$end_time_slice updates the cumulative virtual time used by the process and also assigns a new time slice to the process.

o If the interrupt was from the 32 microsecond real time timer then it advances time_$int_ec. This awakens the clock process which does timer related processing for user processes and sets the next timer value at which it should be awakened. It suspends itself by waiting on time_int_ec.

Interval timers implemented

There are two types of interval timers which have been implemented. They are the real timer which decrements in real time and the virtual timer which decrements in user process virtual time only. The two functions generic to both the timers are getitimer and setitimer which read the current value and set new values for the interval timers. Interval time completion is made known to the user process by posting an appropriate fault.

Real interval timer implementation

The real interval timer has been implemented by enhancing the first mechanism (i.e. the clock process). The second mechanism was not chosen since bsd 4.2 required time intervals in units of the system clock (4 micro seconds). Setting the real interval timer translates into the modification of the timer queue. If the entry is made into the head of the timer queue then a new value is written into the 32 micosecond real time timer. When the clock process is awakened due to an interval time completion it checks if the queue entry belongs to an interval timer. If so it reintroduces the entry back into the queue for the next interval completion. In addition it communicates with the terminal process to actually post the fault to the user process. The clock process cannot directly post the fault to the user process since it is capable of running on the B processor in two processor system. The communication with the terminal process is done in the following manner. The clock process updates a database called the time_$itimer_db and then advances the eventcount called time_$itimer _ec. The terminal process suspends itself on a list of eventcounts one of which is the time_$itimer_ec. When it awakens due to the advancing of this eventcount it looks at the database time_$itimer_db and posts a fault to the proper user process.

Virtual interval timer implementation

The virtual interval timer has been implemented by enhancing the mechanism which keeps track of the cumulative time used by a process. The functions which perform this are the dispatcher, eventcount advance and the time_slice_end. These functions use the 8 microsecond timer. The advance procedure has been modified not to alter the time slice if the virtual timers are being used. This implies that the control for time slice selection will only be done by the time_slice_ end function. The time_slice_end function has been enhanced to check for interval timer completion and also setting the next time slice such that it never exceeds the next interval. If the time_slice_end function recognizes the expiry of an interval time it communicates with the terminal process in the same manner as the clock process. The database in this case is called time_$vitimer_db and the eventcount on which the terminal process sleeps is time_$vitimer_ec. The terminal process then completes the posting of the fault to the user process.

Force writing Files

As of the SR3.0 software release, AEGIS supports two user space calls that force the modified pages of a file to be written to disk. These calls guarantee that any changes to a file are recorded on disk and therefore that such changes will not be lost in the event of a system crash. The services provided are identical for both local and remote files.

There is one caveat to the use of the file force write calls. These calls are intended for use while the file is locked for writing (in the "file_$lock" sense) by their caller. There is no enforcement of this condition, and in fact the force write calls may be safely issued by any process on any node at any time. However, the guarantee is weakened when a force write call is issued by process A and the file is locked for writing by process B. Specifically, the changes made by B will not necessarily be written to disk if (1) A and B are running on different nodes, and (2) B is a remote user of the relevant file. The description of the calls below does not call out this exception explicitly.

FILE_$FW_FILE (uid, status)

The first of these calls is FILE_$FW_FILE. This call takes as its only input argument the UID of the file being force written. Once called, FW_FILE either returns an error code in its status return argument or STATUS_$OK to indicate that all of the file's modified pages have been safely written to disk.

FILE_$FW_PARTIAL (uid, start, length, status)

This call may be used to force write a specified section of a file rather than the whole file. The caller must provide the UID of the file to be force written, the byte offset into the file at which force writing is to begin, and the number of bytes starting at the supplied byte offset to include in the operation. As with FILE_$FW_FILE, this partial file force write returns either an error status code or STATUS_$OK to indicate a successful force write.

Proceedings of the Symposium on Principles of Distributed Computing, Ottawa, Canada, Aug. 1982, pp. 34-41.

UIDs as Internal Names in a Distributed File System

Paul J. Leach, Bernard L. Stumpf,
James A. Hamilton, and Paul H. Levine
Apollo Computer, Inc.
19 Alpha Road, Chelmsford, MA 01824

Abstract

The use of UIDs as internal names in an operating system for a local network is discussed. The use of internal names in other distributed systems is briefly surveyed. For this system, UIDs were chosen because of their intrinsic location independence and because they seemed to lend themselves to a clean structure for the operating system nucleus. The problems created by UIDs were: generating UIDs; locating objects; supporting multiple versions of objects; replicating objects; and losing objects. Some solutions to these problems are presented; for others, no satisfactory solution has yet been implemented.

1. Introduction

Although the area of distributed systems is a relatively new one, there are already many examples of implemented distributed operating systems for local networks and their attendant file systems. Many of these systems have chosen to use internal names for the objects they support, into which user visible text string names are mapped. Among the most popular forms of internal name have been *unique identifiers* (UIDs); however, there has been little in the literature discussing the motivation for choosing one form of name over another, or the consequences of a choice once made. This paper presents the experiences that resulted from using UIDs as internal names in one particular distributed system: the Aegis operating system for the Apollo DOMAIN network [APOL 81], [NELS 81].

1.1. Organization

The rest of this paper is organized as follows. Section 2 discusses internal names as they are used in several other distributed systems. Section 3 presents an overview of the DOMAIN system environment, and of the nature of UIDs and objects in Aegis. Section 4 deals with the motivations and perceived advantages that led us to choose UIDs. Section 5 deals with the problems we foresaw or discovered in the process of implementing the system, and presents some solutions to these problems. Section 6 offers some final observations and conclusions.

2. Internal names in other systems

Given that one decides to use internal names, there seem to be just two fundamental alternatives: to use UIDs or "structured names". UIDs can be thought of as simply large integers or long bit strings, although some other information may be encoded within them. The important characteristic is that they are large enough that the same UID will never refer to two different objects at the same time. *Structured names*, as in [SVOB 79], contain more than one component, some of which are used to indicate the location of, or route to, the object named. However, individual components may be unique for all time only within the context of the other components; some systems with this property have called their internal names UIDs. This section briefly indicates the internal naming schemes used by

(C) ACM 0-89791-081-8/82/008/0034 00.75

several distributed systems or their distributed file system components.

2.1. WFS

The Woodstock File Server (WFS) [SWIN 79] uses "file identifiers" (FIDs) to name files. FIDs are 32 bit unsigned integers, which are unique for all time within a individual WFS server, but may be duplicated across servers. Thus, it is up to each WFS client to remember the server associated with each FID. The combination of server name and FID is a form of structured name. The mapping from FID to physical disk addresses is via a hash table.

2.2. Pilot

Pilot [REDE 80] uses "universal identifiers (UIDs)" to name files; they are 64 bits long and "guaranteed unique in both space and time". UIDs were chosen so that removable volumes could be transported between machines without fear of conflict. A B-tree is used to map UIDs to physical disk addresses.

2.3. DFS

The *distributed file system* (DFS) [STUR 80] also uses UIDs. We suspect that they are really UIDs because the implementors provide "a simple locating service" to help find the server which holds a file, given only its UID; a structured name would not need a locating service. Like Pilot, a B-tree is used to map UIDs to physical disk addresses.

2.4. CFS

The Cambridge File Server (CFS) [DION 80] uses what it calls UIDs to name files. They are 64 bits long; 32 bits are a random number, and 32 bits contain the disk address of the object's descriptor. The use of garbage collection [GARN 80] guarantees that an object will not be deleted while a reference to it exists, and therefore that, within a single server, a UID can never refer to more than one object. However, it seems that UIDs can be duplicated on different servers, although the 32 bit random number makes it highly improbable.

2.5. Felix

The Felix File Server [FRID 81] uses a system generated "File Identifier" (FID) to name files. An FID is a "universal access capability" for the file it names. When the file is deleted, its FID is guaranteed not to be reused for a certain period of time. It also seems that FIDs with the same numerical value can be in use by more than one server at the same time.

2.6. LOCUS

The LOCUS system [POPE 81] uses structured internal names. A name is a pair "<file group number, file descriptor number>". The file group number can be thought of as uniquely identifying a logical volume. The file descriptor number is an index into a per-file-group array of file descriptors; it is unique within a file group as long as any references to the file it identifies exist. The choice of internal name seems to have been motivated by UNIX (TM, Bell Laboratories) compatibility constraints: directory structures are visible to application programs and contain file descriptor numbers, which are relative to the file group containing the directory.

2.7. Others

There are a number of other recent implementations of, or designs for, distributed systems for which descriptions have been published: S/F-UNIX [LUDE 81]; ACCENT [RASH 81]; TRIX [WARD 80], [CLAR 81]; EDEN [LAZO 81]. However, they concentrate on other aspects of distributed systems design, and do not provide much information on their use of internal names.

2.8. Summary

When the design of Aegis began in early 1980, there were fewer examples of distributed systems to study; Pilot and WFS particularly influenced us. Pilot uses UIDs; WFS uses IDs which are unique within a single file server, but which require its clients to remember upon which server files reside. From our studies we got little motivation for either choice; yet upon starting our design it became clear that there were non-trivial problems involved with either choice.

3. DOMAIN system environment

3.1. Hardware

A DOMAIN system consists of a collection of powerful personal computers (nodes) connected together by a high speed (12 megabit/second) local network. Each node has a 'tick' time [LAMP 80] of 1.25 microseconds and can have up to 3.5 megabytes of main memory. Most nodes have 33 megabytes of disk storage and a 1 megabyte floppy disk, but no disk storage is required for a node to operate. A bit mapped display has 800 by 1024 pixels, and a *bit BLT* (block transfer) to move arbitrary rectangular areas at high speed. The display is allocated into windows (called PADs) which are a form of virtual terminal [LANT 79]; multiple concurrent processes, each possessing its own window(s), can be controlled by the user simultaneously. Dynamic address translation hardware allows each process to address 16 megabytes of demand paged virtual memory. The network arbitrates access using a token passing method; each node's network controller provides a unique node ID which is assigned at the factory and contained in the controller's microcode PROMs.

3.2. System usage characteristics

It is expected that the nodes in a network will be owned by many organizations, with each organization owning many nodes. One organization is likely to be chartered to provide computing related services and resources to the entire network community. Within an organization, a high degree of cooperation will be desired; while between organizations, a higher degree of autonomy will be preferred; and the service organization wants resource sharing, protection and (perhaps) accountability. Aegis provides tools to allow a high degree of cooperation, and tools to create policies which can allow a high degree of autonomy. This results in an environment of "policy parameterized autonomy".

3.3. Objects and UIDs

At the highest level, Aegis is an "object-oriented" system, and objects are named by UIDs. Objects are typed and protected: associated with each object is the UID of an access control list, the UID of a type descriptor, as well as a physical storage descriptor, and some other attributes. Supported objects include: alphanumeric text, record structured data, IPC mailboxes, executable modules, directories, access control lists, serial I/O ports, magnetic tape drives, and display bit maps. UIDs are also used to identify persons, projects, and subsystems for protection purposes.

Aegis UIDs are 64 bit structures, containing a 36 bit creation time, a 20 bit node ID, and 8 other bits whose use is described later. UIDs possess the addressing aspects of a capability, but without the protection aspects [FABR 74]. Or, a UID can be thought of as the absolute address of an object in a 64 bit address space. The hardware does not support this form of address, so programs access objects by presenting a UID and asking for the object it names to be "mapped" into the program's hardware processor address space (see [REDE 80] on the desirability of mapping in distributed systems). After that, they are accessed via virtual memory paging: not to create shared memory semantics, but as a form of lazy evaluation, since only the needed portions of objects are actually fetched from disk or over the network.

The system provides a high degree of *network transparency* in accessing objects. The mapping operation is independent of whether the UID is for a remote or local object. As long as programs assume that their objects are not local, and hence operations on them are subject to communication failures, they need not be aware of their location (see [POPE 81] for a discussion).

3.4. Naming Objects

Text string names for objects are provided by a directory subsystem layered on top of the Aegis nucleus. The name space is a hierarchical tree, like Multics [ORGA 72] or UNIX [RITC 74], with directories at the nodes and other objects at the leaves. Each directory is primarily a simple set of associations between *component names* (strings) and UIDs. The *absolute path name* of an object is an ordered list of component names. All but (possibly) the last are names of directories, which, when resolved starting from a network-wide distinguished "root" directory, lead to the UID of the object. Thus, an absolute path name, like a UID, is valid throughout the entire network, and denotes just one object.

4. Motivation for using UIDs

There were several main reasons for choosing UIDs as internal names. First, we wanted location independence: to divorce the internal name of an object from its location in the network. Second, we wanted absolute internal names: ones that could be passed from process to process, and from node to node, without having to be relocated at each step. Third, we wanted to separate text string naming from internal naming, in order to remove string name management from the nucleus. Fourth, we wanted a uniform way of naming all objects in the system. Fifth, we wanted to be able to construct composite objects (objects which refer to other objects) easily, and to allow user programs to do likewise. Sixth, we wanted to allow for typing of objects, and in a potentially extensible and manageable way.

We wanted objects to be able to move without having to find and alter all references to them. The system does not move objects except when explicitly directed to do so. However, users may want to move dismountable volumes from one node to another, or to move a peripheral from a disabled node to a functioning one. Structured names imply locations, which makes moving an object harder, because references to the moved object have to be updated; this in turn mitigates against composite objects. UIDs, because of their location independence, have no such problem.

From an implementation point of view, we wanted to be able to start with simple object locating algorithms, perhaps with restrictions placed on object locations, and work up to better ones, again without changing any stored data. Structured names seemed to freeze this decision too early: the locating scheme is bound into the name. We also wanted to avoid the proliferation of ad hoc internal names by having a single, simple, cheap, uniformly applicable naming scheme available at all but the lowest levels of the system.

Text string names can also be made location independent, but we wanted the nucleus interface to be simpler than string names. Also, string names are too long to be embedded in objects, too expensive to resolve, and therefore can usually be used only at fairly high levels in the system.

So, unlike structured names, UIDs had the right properties to satisfy these requirements. They are intrinsically location independent: they uniquely identify an object no matter where it resides. The node ID contained in our UIDs says where the object was created, but has no *necessary* connection with its current location. They are absolute, and they are (relatively) short and of fixed length. The combination of these attributes means that it is easy to embed UIDs in objects to make composite objects, and that there is little space penalty in using them to name all objects. It also makes it easy to do mapping from text string names to UIDs in a layer above the nucleus. A UID can be used to denote the type of an object. New types (UIDs) can easily be generated without interfering with others doing the same, and can extensibly refer to a type descriptor object containing type data and operations.

There were other, less crucial, advantages that we foresaw. UIDs are good for objects without string names, such as temporary files; objects can even be created as temporaries, then given string names later. Because they are short, they can be easily hashed, and stored in system tables, and passed in IPC messages. Because they are guaranteed to be unique, they can be used as transaction IDs, with the TID also serving to name the commit record object for the transaction. Finally, because UIDs are hard to guess, there are certain capability protection aspects to them: in some cases, it may be acceptable to use possession of a UID as permission to operate on the underlying object.

5. Problems with UIDs

We also quickly discovered that there were problems that needed solution to use UIDs effectively.

1. Generating UIDs and guaranteeing their uniqueness.
2. Locating an object given its UID.
3. Naming different versions of an object
4. Replication of objects
5. Lost objects

5.1. Generating UIDs

We thought that generating UIDs would be easy: concatenate the node ID of the generating node with a reading from its real time clock. The first issue to deal with was choosing the size of the UID. We had a 48 bit 4 microsecond basic system clock, but that, plus a 20 bit node ID, and a few bits for future expansion, seemed to imply a UID that we felt would be a bit long. We settled on a 36 bit creation time, which meant a 16 millisecond resolution. We justified it by noting that, since most objects reside on disk, they can't be created faster than disk speeds; 36 bits allowed a resolution several times higher. To allow for possibly bursty UID generation, the system remembers unused UIDs from the previous minute or so, and uses them before generating new ones.

The second issue is guaranteeing uniqueness. Concatenating a node ID and a real time clock reading guarantees uniqueness as long as one makes sure that the clock always advances. We thought this could be assured by providing a battery operated calendar clock from which to initialize the real time clock. But batteries have a limited shelf life; and since it is important that a UID not be reused, other measures were needed. So the system stores the last shutdown time on the disk, and checks it against the calendar clock during initialization. If the time is too far wrong, either backward, or forward, it requests verification and/or correction from the user. It is clear that the clock cannot be allowed to go backwards; what may not be so instantaneously obvious is that too long a forward jump is also dangerous. Such a jump is likely to be an error, requiring later correction; but if any UIDs are generated from the erroneously advanced clock, they may be duplicated when real time catches up to that point.

Another solution is to use other nodes in the network to corroborate the calendar clock reading; but since it is possible that none will be available, our solution would still need to be resorted to in that case. It seems that no solution is foolproof, but that the probability of failure can be made fairly small. Our experience to date supports this conclusion: with several hundred nodes in use, we know of no problems.

5.2. Locating objects

A direct consequence of the location independence of UIDs is that a locating service is needed to find an object given its UID. This is the fundamental distributed algorithm in Aegis: no global state information is kept about object locations. The complexity of this task depends on the restrictions on object location that higher levels of the system can enforce, and on the desired level of performance. Some examples of the effect of various restrictions that could be imposed are as follows. - One can restrict objects not to move from the node where they are created, in which case node ID part of the UID is certain to be the location of the object. - One can restrict (most) objects to be on same volume as the directory in which they are cataloged. Then, as long as the locations of a few volume root directories can be found, all other objects can be found. - One can restrict object location as in either of the above examples, then relax it by establishing equivalence classes among nodes or volumes, such that if the above rules allowed an object to be on one node or volume of a class, then by these rules, it could be on any node or volume in the class. This would allow multiple physical copies of an object with the same UID to exist and be located. - Of course, it is possible to have no restrictions at all, and still locate objects. After whatever other means exist have failed, a request to return the location of an object can be broadcast, and an answer awaited. Also, in this case, there is absolutely no necessary relation between nodes or volumes and directory hierarchies, making hierarchy backup and crash reconstruction difficult.

We considered all the schemes indicated by the above examples. Because we allow removable volumes, the assumption that objects reside at the node where they were created is not valid. We also convinced ourselves that in a sufficiently large (inter)network, and given the possibility of removable volumes whose node of origin was in a disjoint network, we could not guarantee to find an object even if it were online and accessible. As noted above, even in this case the object could be found if one were willing to make a broadcast to the entire internet, and wait a (possibly) very long time for an answer; but since this had performance implications, as well as the other problems noted above, we were unwilling to base our design on this approach. Thus, we would have to rely on heuristics, and, ultimately, perhaps even help from the user. Our initial goal was to pursue the second approach, as it met our immediate requirements; and it can readily be extended into the third scheme, which we think is sufficiently flexible to eliminate any need for the fourth.

We have already gone through three generations of locating algorithms, and can foresee more. They used two sources of 'hints': the node ID in the UID, and the *hint manager*. The sources for the hint manager's hints can be any program which believes it can guess the whereabouts of an object, or even direct input from a user. In particular, the string name manager guesses that a cataloged object is on the same node as the directory in which it is cataloged (except for special node boundary crossing points).

The first generation algorithm was very simple. To locate an object given a UID, it would first search all local disks. If the local search failed, it would try the node whose ID was contained in the UID. This procedure could always find local objects, objects on dismountable volumes mounted locally, and remote objects that had never moved from where they were created; others, however, could not be located. In particular, remote objects on removable volumes that had been moved from their creation node were unlocatable. Also, for remote objects, time was wasted searching local secondary storage. Note that for remote objects in this scheme, the node ID in the UID was more than just a hint: it had to be right.

The second algorithm added the hint manager. After trying locally, it would consult the hint manager, and if a hint were present, would use the hint. If this failed, it would proceed as in the first case. Therefore, even remote objects on removable volumes could be located, if they were on the same node as the directory in which they were cataloged. This would normally be very likely even if we didn't enforce it (which we currently do).

The time wasted searching locally for remote objects in the previous algorithms was noticeable, so a third was adopted. Before searching locally, the node ID in the UID is examined; if it is not the ID of the local node, then the local search is bypassed. Only if the remote search fails is a local search initiated.

In the future, it is likely that direct input to the hint manager will be added, as will the equivalence class technique. Also, in an internet environment, a second level of hint manager, usually residing on gateway nodes, will probably become necessary. However, its task will be eased considerably because it will only have to store location information for objects that could not be located using the other available hints.

It is significant to note that the object locating service is layered above the nucleus. An object's location is determined when it is mapped into a process' address space, and retained. Thus, it is guaranteed to be known at critical junctures, such as when servicing page faults. It is also cached, so that the location of active objects is likely to be in the cache. The first case is important for clean system structure; the second for good system performance. However, even in the absence of cached or retained information, locating a remote object usually takes only one, and at most two, messages with the current algorithm.

Using UIDs, plus repeated improvement to locating algorithms, has allowed us to benefit from the location independence of UIDs, without paying a serious performance penalty.

5.3. Object versions

If UIDs are allowed to be embedded in objects, the object version problem arises. The object containing the reference may wish not to refer to a particular instance of an object, but to its latest version. A procedure object may contain the UIDs of other programs or of libraries, for example. The fundamental problem is that the same UID can not name two different objects, even if they are just different versions. (For Aegis UIDs, this is true; if they contained an explicit version number, it need not be true.) We see two possible solutions to this problem in our context, both of which involve the use of *indirection objects;* in one case, the indirection object contains a symbolic name; in the other, the UID of the current version of the object. (Indirection objects with symbolic names are also used in the iMAX-432 filing system [POLL 81], where they are called linkage objects.) In the first case, whenever a new version becomes available, the binding of the symbolic name is changed to refer to the new version. In the second case, the indirection object is updated with the new version's UID. In our environment, the second solution is simplest, because it doesn't involve the string name manager to resolve the reference. (The iMAX-432 uses the symbolic solution because it doesn't have real UIDs.)

5.4. Replication

To take advantage of the potential for enhanced reliability that distributed systems offer, it is desirable to be able to redundantly store objects at more than one node. The logical object thus created we call a *replicated object* and each of the redundant copies we call a *replica*. If a replicated object is immutable, this presents no great problem. It is relatively easy for the nucleus to support a replicated immutable object: all the replicas can have the same UID. Even though this results in multiple physical objects with the same UID, since they are all immutable and identical, it never matters which one the nucleus finds and uses; there is only one logical object with that UID. One of the object attributes supported by Aegis' nucleus is immutability.

For mutable objects, however, it is not as easy; updates to the object instances must be coordinated so that all clients see a consistent state. We don't deal with the concurrency management problem here, only the problem of naming the replicated object and its components. ([GIFF 79] and [POPE 81] deal directly with replication; DFS [STUR 80] provides general support for multi-node atomic operations which can be used for replication purposes.) Because it is complex, it is desirable to leave the management of replication out of the nucleus, while still allowing it to be conveniently layered on top. In order to make the new layer transparent to client programs, it is necessary that they be able to refer to a replicated object via one UID. The replication manager, on the other hand, needs to distinguish between the replicas, because internally to it they will have different states, even though the client only sees consistent states. Thus it needs different UIDs for each replica. This leads to essentially the same difficulty as in the object version problem: the same UID needs to refer to more than one object. The replication manager must map a UID presented by a client into the UIDs of the mutable replicas.

One way to accomplish this is to record the UIDs of the replicas in an immutable object, and have clients use its UID to denote the replicated object. A copy of this immutable object is then put at each site holding a replica. When a client refers to the replicated object, its UID is used to locate one of the immutable object copies; if one can be found, then at least the replica at the same site will be available. However, this does not allow the addition of new replicas. To solve this, we use 4 of the 8 'other' bits in the UID to denote particular replicas; let us call it the *replica field*. A replicated object has a UID with a replica field of zero; there is no physical object with this UID. Each of the replicas (up to fifteen of them) has the same UID except for a non-zero replica field. Thus, a client of a replicated object always names it with a UID having a replica field of zero; the replication manager selects and operates on specific replicas via non-zero replica fields.

Contrasting the two solutions, we see that using an immutable object supports an arbitrary mapping from UID of a replicated object to the UIDs of the replicas which constitute its representation; whereas the second scheme causes these UIDs to be easily computable from one another, eliminating the need for the arbitrary map. In addition, the second solution allows replicas to be added and deleted.

5.5. Lost objects

A lost object is one which exists, but for which no references exist; hence it is inaccessible, i.e. lost. Unfortunately, it still takes up disk space. Objects become lost due to crashes, or when objects which contain references to them are deleted. Actually, objects are never completely lost: a scan of a volume's (undamaged) table of contents data structure can find all objects on a volume. However, if an object becomes inaccessible via its text string name, it is often as good as completely lost. The only complete way to recover is garbage collection, but we chose not to implement it. Again, the consideration was nucleus complexity: if internode object references are allowed, a distributed, asynchronous collector is called for, such as [BISH 77]. We knew of no implemented example; the nearest thing is the CFS garbage collector [GARN 80], which is asynchronous, but which doesn't handle internode references. Furthermore, in our current objects, there is no general way to locate all the UIDs, although the implementation of partitioned objects (objects segregated into UID parts and data parts [JONE 80]) would solve this problem. Finally, we felt that most common cases could be handled without it. Most objects are cataloged; and by arranging that an object is not marked *permanent* until it has successfully been cataloged, any newly created but not yet cataloged object will still be temporary if the system crashes, and will be deleted by the file system salvager (see [REDE 80]). Furthermore, all objects have a *father object* attribute, which is the UID of the directory in which they are cataloged, or of the (primary) object which contains its UID. If the father object should cease to exist, the resulting lost object(s) can be deleted. Thus, object tree structures can be handled. We felt that the sum of these techniques would be sufficient.

6. Observations and conclusions

The principal advantages of UIDs are their size, location independence, and the opportunity for layering the nucleus implementation that they provided. Most of the problems involved have been overcome or are understood satisfactorily; the possible exception is the general lost object problem. A feature of UIDs we have taken advantage of is that, because they are location independent, initial implementations of higher layers can impose restrictions on object location, and the restrictions can later be removed without restructuring the lower layers; the same would seem to be hard to accomplish with structured names.

Of course, it is eventually necessary to translate UIDs into structured names, because the knowing the location of an object is a prerequisite to accessing it. We have found it advantageous to delay this binding as long as possible, and to make general and uniform use of the unbound names.

Aegis as currently implemented is missing some of the features described above. Presently, it does not support indirection objects, object replication, partitioned objects, garbage collection, network verified time for UID generation, or extensible types. However, the fundamental groundwork, that of making a design that can be gracefully extended, and anticipating the most likely areas of extension, is essential to any system which is intended to have a long and useful life. We think that we have accomplished that goal.

REFERENCES

[APOL 81] — **Apollo DOMAIN Architecture.** Apollo Computer Inc., Chelmsford, Mass., 1981.

[BIRR 80] Birrel, A. D., Needham, R. M. "A Universal File Server." *IEEE Tranactions on Software Engineering*, SE-6, 5 (September 1980), pp. 450-453

[BIRR 82] Birrel, A. D., Levin, R., Needham, R. M., Schroeder, M. D.
"Grapevine: An Exercise in Distributed Computing." *Communications of the ACM*, **25**, 4 (April 1982), pp. 260-274.

[BISH 77] Bishop, P. B. **Computer Systems with a Very Large Address Space and Garbage Collection.** Technical Report LCS/TR-178, Laboratory for Computer Science, M.I.T., Cambridge, Mass., May 1977.

[CLAR 81] Clark, D., Halstead, B., Keohan, S., Sieber, J., Test, J., Ward, S.
"The TRIX 1.0 Operating System." *Newsletter of IEEE Tech. Comm. on Distributed Processing*, **1**, 2 (December 1981), pp. 3-5.

[DION 80] Dion, J.
"The Cambridge File Server." *Operating Systems Review*, **14**, 4 (October 1980), pp. 26-35.

[FABR 74] Fabry, R.S.,
"Capability-Based Addressing" *Communications of the ACM*, **17**, 7 (July 1974), pp. 403-412.

[FRID 81] Fridrich, M., Older, W.
"The FELIX File Server." *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 37-44.

[GARN 80] Garnett, N. H., Needham, R. M.
"An Asyncronous Garbage Collector for the Cambridge File Server." *Operating Systems Review*, **14**, 4 (October 1980), pp. 36-40.

[GIFF 79] Gifford, D. K.
"Weighted Voting for Replicated Data," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 150-162.

[JONE 80] Jones, A.K.
"Capability Archictecture Revisited." *Operating Systems Review*, **14**, 3 (July 1980), pp. 33-35.

[LAMP 80] Lampson, B. W., and Redell, D. D.
"Experience with Processes and Monitors in Mesa." *Communications of the ACM*, **23**, 2 (February 1980), pp. 105-113.

[LANT 79] Lantz, K. A., Rashid, R. F.
"Virtual Terminal Management in a Multiple Process Environment." *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 86-97.

[LAZO 81] Lozowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S.
"The Architecture of the Eden System." *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 148-159.

[LEVI 79] Levin, R., Cohen, E., Corwin, W., Pollack, F., Wulf, W.
"Policy/Mechanism Seperation in Hydra." *Proceedings of the Fifth Symposium on Operating Systems Principles*, December 1979, pp. 132-140.

[LISK 79] Liskov, B.
"Primitives for Distributed Computing". *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 33-42.

[LUDE 81] Luderer, G. W. R., Che, H., Haggerty, J. P., Kirslis, P. A., Marshall, W. T.
"A Distributed Unix System Based on a Virtual Circuit Switch". *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 160-168.

[NEED 78] Needham, R. M., Schroeder, M. D.
"Using Encryption for Authentication in Large Networks of Computers." *Communications of the ACM*, **21**, 12 (December 1978), pp. 993-999.

[NELS 81] Nelson, D. L.
"Role of Local Network in the Apollo Computer System." *Newsletter of IEEE Tech. Comm. on Distributed Processing*, **1**, 2 (December 1981), pp. 10-13.

[ORGA 72] Organick, E. I. **The Multics System: An Examination of Its Structure** M.I.T. Press, 1972.

[POLL 81] Pollack, F., Kahn, K., Wilkinson, R.
"The iMAX-432 Object Filing System." *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 137-147.

[POPE 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G.
"LOCUS: A Network Transparent, High Reliability Distributed System." *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 169-177.

[RASH 81] Rashid, R. F., Robertson, G. G.
"Accent: A Communications Oriented Network Operating System Kernel," *Proceedings of the*

Eighth Symposium on Operating Systems Principles, December 1981, pp. 64-75.

[REDE 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. C.
"Pilot: an Operating System for a Personal Computer." *Communications of the ACM*, **23**, 2 (February 1980), pp. 81-91.

[RITC 74] Ritchie, D. M., Thompson, K.
"The UNIX time-sharing system" *Communications of the ACM*, **17**, 7 (July 1974), pp. 365-375.

[STUR 80] Sturgis, H., Mitchell, J., Israel, J.
"Issues in the Design and Use of a Distributed File Server." *Operating Systems Review*, **14**, 3 (July 1980), pp. 55-69.

[SVOB 79] Svobodova, L., Liskov, B., Clark, D. **Distributed Computer Systems: Structure and Semantics.** Technical Report LCS/TR-215, Laboratory for Computer Science, M.I.T., Cambridge, Mass., March 1979.

[SWIN 79] Swinehart, D., McDaniel, G., Boggs, D.
"WFS: A Simple Shared File System for a Distributed Environment." *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 9-17.

[WARD 80] Ward, S.
"TRIX: A Network-oriented Operating System." *Proceedings of COMPCON '80*, San Fransisco, Feb. 1980.

[WULF 74] Wulf, W., Cohen, E., Corwin, W., Jones. A., Levin, R., Pollack, F.
"Hydra: The Kernel of a Multiprocessor Operating System." *Communications of the ACM*, **17**, 6 (June 1974), pp. 337-345.

To Appear: ACM Computer Science Conference, New Orleans, LA, March 13-15, 1985.

The File System of an Integrated Local Network

Paul J. Leach, Paul H. Levine,
James A. Hamilton, and Bernard L. Stumpf

Apollo Computer, Inc.
15 Elizabeth Drive, Chelmsford, MA 01824

Abstract

The distributed file system component of the DOMAIN system is described. The DOMAIN system is an architecture for networks of personal workstations and servers which creates an integrated distributed computing environment. The distinctive features of the file system include: objects addressed by unique identifiers (UIDs); transparent access to objects, regardless of their location in the network; the abstraction of a single level store for accessing all objects; and the layering of a network wide hierarchical name space on top of the UID based flat name space. The design of the facilities is described, with emphasis on techniques used to achieve performance for access to objects over the network.

1. Introduction

This paper describes the design of the distributed file system for the Apollo DOMAIN operating system. DOMAIN is an integrated local network of powerful personal workstations and server computers ([APOL 81], [NELS 81]); both of which are called *nodes*. A DOMAIN system is intended to provide a substrate on which to build and execute complex professional, engineering and scientific applications ([NELS 83]). Other systems built following the integrated model of distributed computing include EDEN [LAZO 81] and LOCUS [POPE 81].

Within the DOMAIN system, the network and the distributed file system contribute to this goal by allowing the professional to share programs, data, and expensive peripherals, and to cooperate via electronic mail, with colleagues in much the same manner as on larger shared machines, but without the attendant disadvantage of sharing processing power. Cooperation and sharing are facilitated by being able to name and access all objects in the same way regardless of their location in the network.

Thus, when we say that DOMAIN is an integrated local network, we mean that all users and applications programs have the same view of the system, so that they see it as a single integrated whole, not a collection of individual nodes. However, we do not sacrifice the autonomy of personal workstations to achieve integration: each personal workstation is able to stand alone, but the system provides mechanisms which the user can select that permit a high degree of cooperation and sharing when so desired.

Another reason we say that DOMAIN is an integrated local network is that each machine runs a complete (but highly configurable) set of standard software, which (potentially) provides it with all the facilities it normally needs – file storage, name resolution, and so forth. In contrast are server-based distributed systems, wherein network wide services are provided by designated machines ("servers") which run special purpose software tailored to providing some single service or small number of services (e.g. Grapevine [BIRR 82], WFS [SWIN 79], and DFS [STUR 80]). DOMAIN has server nodes; however, they are created by configuring the standard hardware and software for a special purpose – a "file server" node, say, is created using a machine with several large disks and system software configured with the appropriate device drivers.

1.1. Organization

The rest of this paper is organized as follows. The remainder of this introduction briefly descibes the hardware environment on which the system runs. Section 2 provides an overview of the file system, and breaks it into four major component groups. Section 3 gives a block diagram of the file system structure, and a brief description of each module, locating it within one of the component groups. Sections 4, 5, 6, and 7 each describe one of these component groups. Finally, section 8 focuses on those aspects of the design which we believe have contributed most to the efficiency of the system.

1.2. Hardware Environment

A DOMAIN system consists of a collection of powerful personal workstations and server computers (generically, nodes) interconnected by a high speed local network.

1.2.1. User Interface

Users interact with their personal nodes via a display subsubsystem, which includes a high resolution raster graphics display, a keyboard and a locating device (mouse, touch pad, or tablet). A typical display has 800 by 1024 pixels, and *bit BLT* (bit block transfer) hardware to move arbitrary rectangular areas at high speed. Server nodes have no display, and are controlled over the network. More information on the user environment can be found in [NELS 84].

1.2.2. CPU

There are several models of both personal and sever nodes. Their 'tick' times [LAMP 80] range from .4 to 1.25 microseconds; their maximum main memory ranges from 3.5 megabytes to 8 megabytes. Most personal nodes have 33 to 154 megabytes of disk storage and a 1 megabyte floppy disk, but no disk storage is required for a node to operate. Server nodes configured as file servers can have 300-1000 megabytes or more of disk storage; those configured as peripheral servers can have printers, magnetic tape drives, plotters, and so forth.

All nodes have dynamic address translation (DAT) hardware which supports up to 128 processes, with each process able to to address 16 or 256 megabytes of demand paged virtual memory (depending on CPU model). The DAT hardware on some models uses a reverse mapping scheme, similar to that used in the IBM System/38 [HOUD 78]; it is a large, hardware hash table keyed by virtual address, with the physical address given by the hash table slot number in which a translation entry is stored. Other models use a forward mapping scheme, similar to the VAX [DEC 79] or System/370 [IBM 76]. The DAT also maintains used and modified statistics on a per page basis for the use of page replacement software, and access protection controlling read, write and execute access. The differences between the DATs of the different models are abstracted away by an MMU (memory management unit) module.

1.2.3. Network

The network is a 12 megabit per second baseband token passing ring (other ring implementations are described in [WILK 79], [GORD 79]; and reasons for preferring a ring network in [SALT 79], [SALT 81]). Each node's ring controller provides the node with a unique node ID, which is assigned at the factory and contained in the controller's microcode PROMs. The maximum packet size is 2048 bytes. The controller has a broadcast capability.

We will not discuss the network further here; for purposes of the file system, all that is required is that the it deliver messages with high probability and low CPU overhead. For more information on the ring controller and data link layer protocols see [LEAC 83].

2. File System Overview

The DOMAIN file system is actually made of four distinct components: an *object storage system* (OSS), the *single level store* (SLS), the *lock manager*, and the *naming server*. (See figure 1 for a block diagram.)

The OSS provides a flat space of objects (storage containers) addressed by unique identifiers (UIDs). Objects are typed, protected, abstract information containers: associated with each object is the UID of a type descriptor, the UID of an access control list (ACL) object, a disk storage descriptor, and some other attributes: length; date time created, used and modified; reference count; and so forth. Object types include: alphanumeric text, record structured data, IPC mailboxes, DBMS objects, executable modules, directories, access control lists, serial I/O ports, magnetic tape drives, and display bit maps. (Other objects which are *not* information containers also exist. UIDs are used to identify processes; and to identify persons, projects, organizations, and protected subsystems for authentication and protection purposes.) The distributed OSS makes the objects on each node accessible throughout the network (if the objects' owners so choose by setting the objects' ACLs appropriately). The operations provided by the OSS on storage objects include: creating, deleting, extending, and truncating an object; reading or writing a page of an object; getting and setting attributes of an object such as the ACL UID, type UID, and length; and locating the home node of an object. The OSS automatically uses a node's main memory as a cache of recently used pages, attributes, and locations of objects, including remote ones. It does nothing to guarantee cache consistency between nodes; however, it does provide mechanisms that the lock manager can use to make and enforce such guarantees.

A unique aspect of the DOMAIN system is its network wide single level store (SLS). (Multics [ORGA 72] and the IBM System/38 [FREN 78] are examples of a single level store for centralized systems.) Programs access all objects by presenting their UIDs and asking for them to be "mapped" into the program's address space (see [REDE 80] on the desirability of mapping in distributed systems); subsequently, they are accessed with ordinary machine instructions, utilizing virtual memory demand paging.

The purpose of the single level store is not to create network wide shared memory semantics akin to those of a closely coupled multiprocessor; instead, it is a form of lazy evaluation: only required portions of objects are actually retrieved from disk or over the network. Another purpose is to provide a uniform, *network transparent* way to access objects: the mapping operation is independent of whether the UID is for a remote or local object. As long as programs make the worst case assumption that their objects are not local, and hence that operations on them are subject to communication failures, they need not be aware of their location. (See [POPE 81] on the desirability of network transparency.)

The lock manager serializes multiple simultaneous access to objects by many processes, including ones on different nodes. A process must lock an object prior to its use; the lock manager arbitrates lock requests, and uses the sequence of requests to keep main memory caches consistent.

The naming server allows objects to be referred to by text string names. It manages a collection of directory objects which implements a hierarchical name space much like that of Multics or UNIX[1] [RITC 74]. The result is a uniform, network wide name space, in which objects have a unique canonical text string name as well as a UID. The name space supports convenient sharing, which would be severely hampered without the ability to uniformly name the objects to be shared among the sharing parties.

3. File System Structure

Figure 1 shows a block diagram of the file system. Each of the major component groups is indicated by a different form of shading. The arrows between blocks indicate call dependencies; in addition, all modules above the "pageable" boundary have an implicit dependency on the SLS.

The system is stuctured using a data abstraction approach, sometimes called a "type manager" approach when applied to operating systems ([JANS 76]). Each module has a set of operations and a private database in which to record its state. Thus, in describing the components of the system, we will identify the managers which comprise that component, and then, for for each manager, the essential operations provided by that manager, and an indication of the form of the database and algorithms used to implement the operations. (Note: in the descriptions of calls in this paper, irrelevant details have often been suppressed for ease of exposition; the intent is to capture the semantic flavor of the interfaces, not their precise syntax.)

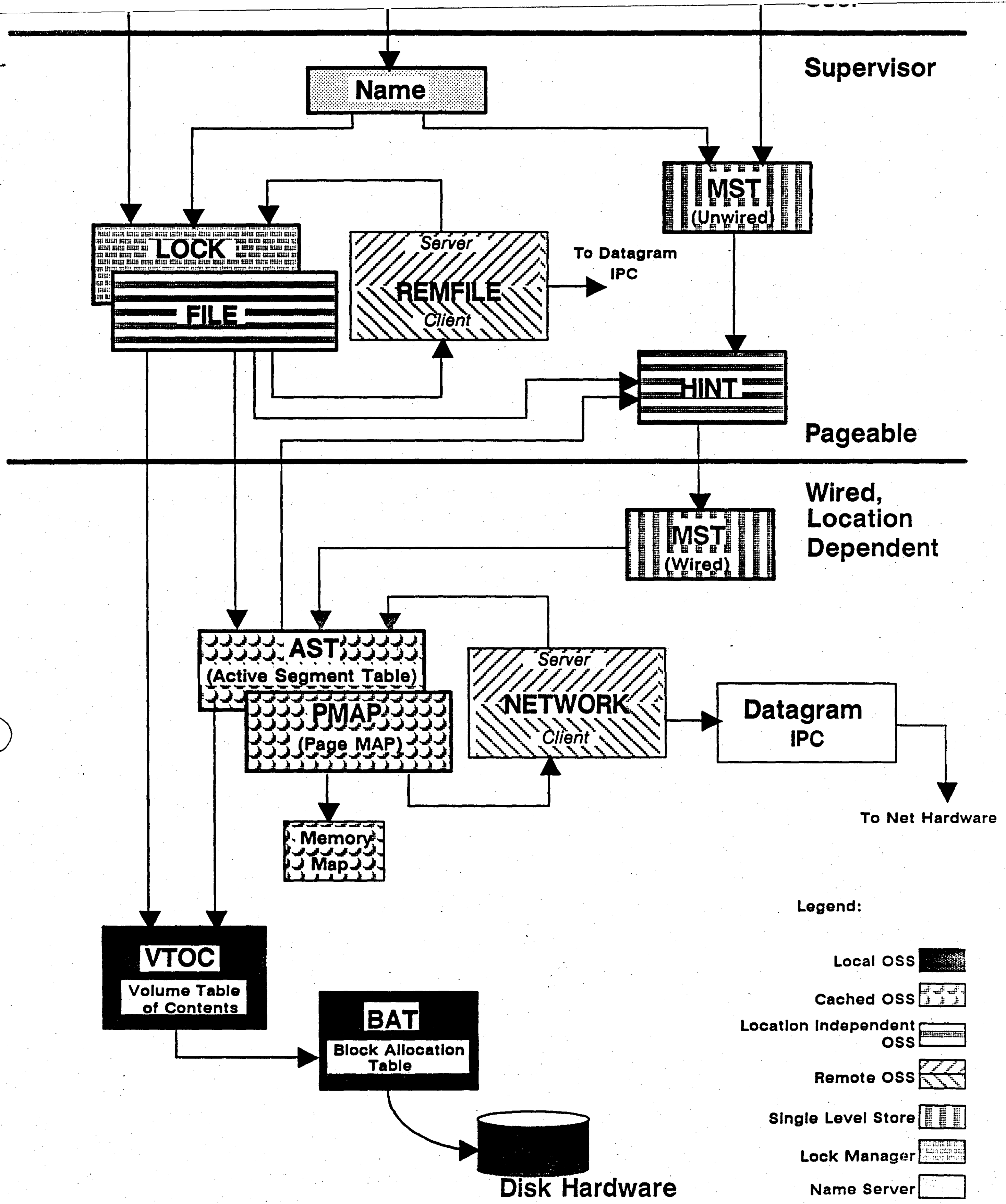
4. Object Storage System

The OSS is the DOMAIN counterpart of distributed file systems such as WFS [SWIN 79] and DFS [STUR 80]. The purpose of the OSS is to provide permanent storage for objects, and to allow objects to be identified by and operated on using UIDs, independent of their location in the network.

At the level we will discuss here, an object is just a data container: an array of uninterpreted data bytes, or more precisely, an array of pages (1024 byte units into which objects are divided). Other object attributes, such as it's type descriptor and access control list are not used by the OSS, but are simply stored for the use of higher levels. (Not all objects are represented by storage containers: for example, processes are identified by UIDS, but are not associated with any permanent storage.)

The OSS consists of several component subgroups: a *local OSS*, *remote OSS*, *cached OSS*, and an *object locating service*. The top-level *location independent OSS*

[1] UNIX is a trademark of Bell Laboratories.



File System Structure

abstraction is created utilizing these services.

4.1. Identifying Objects

UIDs of objects are bit strings (64 bits long); they are made unique by concatenating the unique ID of the node generating the UID and a time stamp from the node's timer. (The system does not use a global clock.) UIDs are also *location independent*: the node ID in an object's UID can not be considered as anything more than a hint about the current location of the object. (More detail on the use and implementation of UIDs is presented in [LEAC 82].)

At any point in time, the permanent storage for an object resides entirely at only one node; also, the system never attempts to transparently move it to a different node. So, for every object there is always one distinguished node which is its "home", and which serves as the locus of operations on the object. Above the OSS level, only UIDs are used to address objects; an operation whose UID addresses a remote object is sent to the object's home node to be performed.

4.2. Local OSS

This subgroup provides access to local objects: *i.e.*, those objects stored on disk volumes which are attached to the node accessing them. It provides operations to create and delete local objects, and to access the attributes and contents (pages) of existing objects (see figure 2). There are two managers in this group: the VTOC (volume table of contents) and the BAT (block allocation table).

The VTOC for a volume contains an entry for each object on the volume; an object's VTOC entry contains the object's attributes and the root of its *file map*, which translates page numbers within an object to disk block addresses. (VTOC entries are very similar to UNIX *inodes* [THOM 78].) The VTOC is organized as an associative lookup table keyed by object UID, which permits rapid location of an object's VTOC entry given its UID. (Using a large direct mapped hash table with chained overflow buckets and avoiding high utilization, the average lookup time is just over one disk access.)

To access the contents of an object requires two steps: translate the object reference to disk block address, then read (or write) the disk block. (An object reference is a pair consisting of the object's UID and a page number within the object.) The VTOC only provides operations to do the translation, not the reads or writes, because the translations are then cached and

allocate — allocate a VTOC entry for an empty object and set its attributes

> The object is created on the local disk volume specified by 'vol-index'. The object descriptor contains the object's UID and initial attributes.

FUNCTION allocate(vol-index, obj-decriptor): vtoc-index

lookup — get the VTOC index of an object
FUNCTION lookup(vol-index, obj-uid): vtoc-index

read — get the VTOC entry of an object given its VTOC index

> Attributes in the 'vtoc-entry' include: object UID; type UID; ACL UID; length; time created, used, and modified; reference count, etc.

FUNCTION read(vol-index, vtoc-index): vtoc-entry

write — write the VTOC entry of an object given its VTOC index

> Note: overwriting a VTOC entry for an object with an empty VTOC entry has the effect of deleting the object.

FUNCTION write(vol-index, vtoc-index, vtoc-entry)

read-fm — get the file map for a segment of an object

> Object are divided into 32 page segments; the 'seg-no' indentifies the segment; the 'file-map' is an array of 32 disk block addresses, one for each page in the segment.

FUNCTION read-fm(vol-index, vtoc-index, seg-no): file-map

write-fm — write the file map for a segment of an object
FUNCTION write-fm(vol-index, vtoc-index, seg-no, file-map)

Figure 2: Sample VTOC Operations

used by the cached OSS (see below). The translation is done by reading or writing the file map for 32 page units of the file called *segments*.

The BAT for a volume keeps track of which disk blocks are available for allocation on that volume. The principle operations on the BAT are ones to allocate and free disk blocks. One interesting feature is that the allocation operation aids in creating locality of the pages within an object on the disk. One of the input parameters of the allocation operation is a disk block address; an attempt is made to make the newly allocated block as close as possible to it. When a new page is being added to an object, this parameter is usually set to the disk address of the previous logical page of that object. We observe that this causes much better clustering of objects on the disk than not doing anything at all, except when the disk is nearly full. (We have not analyzed the benefit quantitatively. Also, to get really good locality, it is probably necessary to use the more comprehensive methods of [MCKU 84].)

4.3. Cached OSS

Disk operations and remote operations are both expensive, so it is desirable to avoid them when possible. One means of doing so is to cache recently obtained results of such operations, and reuse them when it can be ascertained that they are still valid.

The cached OSS consists of the AST, PMAP, and MMAP managers. The AST (active segment table) caches locations, pages, and attributes of *active* (recently used) objects, whether local or remote. Each entry in the AST contains the UID, location and attributes of an object, plus the PMAP for one segment of the object. The PMAP (page map) for a segment contains the file map for that segment, plus references to all resident main memory pages. Part of the maintenance of PMAPs is done by the *purifier* process, which periodically writes back modified pages to secondary storage (local or remote, as need be). The MMAP (memory map) is the allocator of main memory pages, and keeps track of their contents.

The AST provides operations to access pages and attributes (including locations) of objects (see figure 3). If the requested information is not in its cache (or PMAP's), then it uses the local or remote OSS to get the necessary information and encache it. The *touch* operation fetches object contents (pages). (There is no write operation; pages are modified via the single level store while in the cache, then written back later by the PMAP purifier process.) The *get-attr* operation fetches

touch — cause several consecutive pages of an object to be cached in main memory

> Cause 'n' pages pages starting with 'page-num' of object with UID 'object-uid' to be cached. The object 'location' is the ID of the remote node or local volume where the object resides.

FUNCTION touch(location, object-uid, page-num, n): phys-page-list

get-attr — get an object's attributes

> Attributes in the 'attr-rec' include: type UID; ACL UID; length; time created, used, and modified; reference count, etc.

FUNCTION get-attr(object-uid): attr-rec

set-attr-X — set attribute X *of an object*

> This is a set of operations, where X can be replaced by any of the attributes above.

PROCEDURE set-attr-X(object-uid, X-value)

cond-flush — remove stale pages of an object from the cache

> The boolean 'flushed' is true if any stale data was flushed.

FUNCTION cond-flush(object-uid, dtm): flushed

purify — send all modified pages of an object back to its 'home' node

> if 'force' is true, write the pages to disk immediately at the home node, else just leave them in the home node's cache.

PROCEDURE purify(object-uid, force)

Figure 3: Sample AST Operations

object attributes, and *set-attr* allows objects' attributes to be individually changed.

The AST also provides operations to manage its cache's consistency with that of other nodes, and which are designed to be used by the lock manager: it only allows access to objects if they are properly locked; it maintains a version number for each object; and it provide operations to control the contents of the cache.

4.3.1. Lock Enforcement

As one of its attributes, each file system object has a *lock key*. The lock key is set to either a network node ID or one of (for now) two special values: **readbyall** and **writebyall**. When an object's lock key is set to **N**, only OSS requests from node N are processed. All other requests are denied with an error indication of concurrency violation. When the lock key is set to **readbyall**, read requests (for pages and attributes) from every node are allowed while all write requests are denied regardless of their source. Finally, a lock key value of **writebyall** completely disables the OSS level concurrency control checking and so all requests are always fulfilled.

4.3.2. Object Versions

A time stamp based version number scheme is used to support the cache validation mechanism. An object's version number is its *date-time modified* (DTM) attribute. (See [KOHL 81] for a survey of distributed concurrency techniques.) Every object has a DTM with 8 millisecond resolution associated with it, which records the time the object was last modified.

The DTM of an object is maintained at its home node. When an object is modified by locally originating memory writes, the page modified bits in the DAT hardware record that fact; periodically, the modified bits are scanned and cause the object's DTM to be updated. If an object is modified by a remote node, eventually the object's modified pages are sent back to the home node; the paging server updates an object's DTM in response to remotely originating OSS requests to write its pages.

In addition, every node also remembers the DTM for all remote objects whose pages it has encached in its main memory. Every time a page of an object is read from or written back to its home node, the latest DTM is sent with the network reply message. Recall that the requests for page level operations are filtered through the lock key based low-level concurrency control.

4.3.3. Content Control

There are several operations explicitly provided by the AST to allow for cache management by higher level synchronization mechanisms.

1. A *conditional flush* operation expunges from the cache all pages of an object that are not from the current version of the object. (This is used by the lock manager when it discovers that the DTM associated with the cached pages of an object is different from the object's real DTM.)
2. A *get-attr* operation returns (among other attributes) the DTM of the current version of an object.
3. A *purification* operation sends copies of all modified pages of an object back to the home node of the object (but leaves the pages encached for possible later use). (This is used by the lock manager at unlock time.)
4. A *force write* variant of the purification operation causes a page to be written to permanent store on its home node; its purpose is to be a minimally sufficient toe hold with which to implement more complex atomic operations.

We shall see that using by using the AST's lock enforcement, object version, and cache content control facilites, the lock manager can effectively guarantee cache consistency for all clients who obey the system locking rules (see section 6).

4.4. Location Independent OSS

Location independent access to objects is provided by the SLS and the location independent OSS. The SLS provides access to the contents of already existing objects, while the location independent OSS provides access to object attributes, and supports object creation and deletion.

The location independent OSS consists of the FILE manager, and the HINT manager. The FILE manager exports the attribute access and cache control operations of the AST to user programs in a location independent way. In addition, it implements a *create* operation to create new objects, a *delete* operation to destroy them, and a *locate* operation to return the node ID of the home node of an object (see figure 4). To create location independence, the FILE manager uses the HINT manager to determine the location of an object, then either does the operation locally (using the local or cached OSS), or uses the services of REMFILE (see below) if it must go remote.

create — create an object

the new object is created on the same node as 'loc-object-uid'

FUNCTION create(loc-object-uid): new-object-uid

delete — delete an object

PROCEDURE delete(object-uid)

locate — return the node address of the home node of an object

FUNCTION locate(object-uid): node-id

Figure 4: Sample FILE Operations

The HINT manager is the backbone of the locating service: given an object's UID, it finds the ID of the node on which an object resides. This is the fundamental distributed algorithm in the system: no global state information is kept about object locations. Instead, a heuristic search is used to locate an object. Complete details are in [LEAC 82], including design considerations and the evolutionary history of the algorithm. To summarize briefly, the current algorithm relies heavily on hints about object location. One source is the node ID in the object's UID, another is the *hint file*. Any time a software component can make a good guess about the location of an object, it can store that guess in the hint file for later use; one particularly good source of hints is the naming server, which guesses that objects are co-located with the directory in which they are catalogued. If all hints fail to locate the object, then the requesting node's local disk is searched for the object. The algorithm works because, although it is possible for objects to do so, they rarely move from the node where they were created; and if they do, then the naming servers hint will nearly always be correct. A last resort, which would be completely sufficient, would be to accept user input into the hint file; this has not yet been implemented, as it hasn't really been needed.

4.5. Remote OSS

The remote OSS is separated into two parts which are at two very different layers of the system: the NETWORK manager, which provides remote access to the attributes and contents of already existing objects; and the REMFILE manager, which provides facilities to remotely create and delete objects. This is in contrast to the local OSS, where one set of managers provides both capabilities; the purpose is to separate the pieces of the remote OSS which are needed to resolve page faults from those which are not. This both minimizes the amount of code and data which must be permanently resident in main memory in order to implement virtual memory, and allows the REMFILE manager to use the virtual memory provided by the SLS. Both NETWORK and REMFILE are location dependent abstractions: in order to access a remote object, its location must already be known. Both of these managers can be thought of as hand-coded stubs for a simple form of remote procedure call (RPC) [BIRR 84].

The NETWORK manager is divided into a *client side* and a *server side*. The client side is used by the cached OSS to access the attributes and contents (pages) of already existing remote objects that are not in the main memory cache. When the client side is called to make a remote access, it is given the request parameters and the node ID of the home node of the object being accessed. (The request parameters always include the UID of an object, and, for a read page request, would include the page number of the object to read, for example). It packages the request parameters into a message, sends it to the given node using the low-level *socket* datagram IPC and waits for a response. Since the requests are all idempotent, it can use a very simple request-response protocol ([SPEC 82]); for more details on sockets and protocols see [LEAC 83].

The server side uses a *remote paging server* process to handle client requests, which services all remotely originating requests to read or write pages and attributes of objects on that node. The paging server has a socket assigned to it, with a well known ID, upon which it receive requests; it uses the local access mechanism to fulfill those requests. Remote paging operations are requested via **(UID, page number)** pairs only, never by disk address, and other remote operations only via UIDs; thus, a node never depends on any other node for the integrity of its object store. (This is one of the reasons the system is truly a collection of autonomous nodes – to which are added mechanisms permitting a high degree of cooperation – as distinguished from, say, a locally dispersed loosely coupled multiprocessing system.)

The REMFILE manager is also divided into client and server sides, and except that the operations are to create and delete objects, its structure is nearly identical to the NETWORK manager. The server side uses a *remote file server* process; it services client requests by calling the FILE manager to service requests. REMFILE also handles remote lock requests for the LOCK manager; see section 6.

5. Single Level Store

The single level store concept means that all memory references are logically references directly to objects. This is in contrast to a multi-level store, which typically has a "primary" store and one (or more) "secondary" store(s); only the primary store is directly accessible by programs, so they have to do explicit "I/O" operations to copy an object's from secondary to primary store before the data can be accessed. To make the distinction between primary and secondary store transparent, a single level store has to manage main memory as a cache over the object store: fetching objects (or portions of objects) from permanent store into main memory as needed, and eventually writing back modified objects (or portions thereof) to the permanent store. SLS is thus a form of virtual memory, since all referenced information need not (indeed could not) be in main memory at any one time.

Our implementation of SLS has many aspects in common with implementations of SLS for a centralized system: main memory is divided into page frames; each page frame holds one object page; main memory is managed as a write-back cache; DAT hardware allows references to encached pages at main memory speeds. If an instruction references a page of an object which is not in main memory, the DAT hardware causes a page fault, and supplies the faulting virtual address and the ID of the faulting process to software. The page fault handler finds a frame for the page; reads the page into the frame; updates the DAT related information to show that the page is main memory resident; and restarts or continues the instruction.

The SLS is implemented by the MST manager, which comes in two modules: one which is permanently resident, called MST-wired; and one which is pageable, called MST-unwired. Both manipulate a per process table, the *Mapped Segment Table* (MST), which translates a virtual address to a **(UID, page number)** pair.

MST-unwired implements a *map* operation, which adds an object to the address space of a process given the object's UID; an *unmap* operation, which removes an object; a *get-uid* operation to inquire about the objects in an address space; and a *set-touch-ahead-cnt* operation to cause read-ahead on page faults. To map

---

map — make an object accessible through a virtual address space range
FUNCTION map(object-uid, protection, grow-ok, out obj-length): virt-addr

---

unmap — remove an object from the address space
PROCEDURE unmap(virt-addr)

---

getuid — get the UID of a mapped object
FUNCTION getuid(virt-addr): object-uid

---

set-touch-ahead-cnt — set demand paging cluster factor for a mapped object

> Causes pages of the object to be read/written in 'cluster-size' units.

PROCEDURE set-touch-ahead-cnt(virt-addr, cluster-size)

---

touch — cause a page to be cached in main memory

> The page refered to by virtual address 'virt-addr' is brought into memory, and the MMU is loaded with the 'virt-addr' <-> 'phys-page-addr' association.

PROCEDURE touch(virt-addr): phys-page-addr

---

wire — cause a page to be cached in main memory and made non-pageable
PROCEDURE wire(virt-addr): phys-page-addr

---

find — find the phyical page address for a virtual address

> Optionally wire the page if 'wire-flag' is true.

PROCEDURE find(virt-addr,wire-flag): phys-page-addr

Figure 5: Sample MST Operations

an object into the address space, an entry defining the **(virtual address, UID)** association is made in the MST; unmapping just removes the appropriate entry. None of these operations are required while servicing a page fault; thus, the module can be pageable.

MST-wired implements a *touch* operation, which for a given virtual address, causes the object page associated with it to be cached in main memory. The *touch* operation is given the virtual address of the faulting page, which it looks up in the MST to get the UID of the object mapped at that address; fetching the page is then just a request to the OSS, even if the page belongs to a remote object (see figure 5). If the touch ahead count is more than one, it will also pre-fetch succeeding pages of the object. Other operations include a *wire* operation, which is similar to *touch*, except that the page is made permanently resident as well; and a *find* operation, which returns the main memory address of a page if it is resident.

What distinguishes our implementation from a centralized one is the necessity of dealing with multiple main memory caches: in fact, one for each node in the network. This leads to the problem of synchronizing the caches in some way: of finding and fetching the most up-to-date copy of an object's page on a page fault, and of avoiding the use of "stale" pages (ones that are still in a node's cache, but have been more recently modified by another node). The objective of synchronization is to give programs a consistent view of the current version of an object in the face of (potentially) many updaters. A second objective is that the synchronization algorithm should be quite simple and need only a small data base, as it would be part of the SLS implementation and hence be permanently resident in main memory.

These objectives appeared, for practical purposes, to be mutually exclusive, so our SLS implementation does *not* guarantee consistency or the use of the current version. Instead, the implementation *does* provide operations and information from which a higher level can build a mechanism that makes the stronger guarantees. In addition, the higher level can use the virtual memory provided by SLS, and thereby be in large measure freed of the constraints mentioned earlier on the size of it and its data base. The system provides a readers/writers locking mechanism at the higher level; however, other clients are free to construct their own synchronization mechanism at this level if they do not wish to use ours.

lock — lock an object

See text for explanation of 'obj-mode'; 'acc-mode' is one of read, write, or read-intend-write. The boolean 'locked' is returned true if the object was locked; the caller never waits.

FUNCTION lock(object-uid, obj-mode, acc-mode): locked

relock — change the access mode of an lock

The boolean 'changed' is returned true if the access mode was changed.

FUNCTION relock(object-uid, acc-mode): changed

unlock — unlock an object

FUNCTION unlock(object-uid, acc-mode)

read-entry — find the lock entry record for an object

the 'lock-rec' contains the object uid, process uid of the locking process, the object and access modes of the lock, and a transaction ID (see text).

FUNCTION read-entry(object-uid): lock-rec

iter-entry — iterate through all locked objects

if 'volume-uid' is non-nil, restrict the iteration to just objects on that volume; 'N' starts at 0, and after each call is the index of the next entry to be returned.

FUNCTION iter-entry(volume-uid, N, object-uid): lock-rec

Figure 6: Sample LOCK Operations

6. Lock Manager

The LOCK manager provides clients of the file system the means to obtain control over an object and to block processes that wish to use the object in an incompatible way. The tools that the lock manager has at its disposal are its own lock data base and the lock key attribute associated with each object.

The *lock* operation supports two locking modes for objects. The more familar is the many readers or single writer lock mode [HOAR 74]. A *co-writers* (co-located writers) lock mode is also provided, which makes no restrictions on the number of readers and writers, but demands that they be co-located at a single network node. This mode allows the use of shared memory semantics, but only among processes located at the same node.

(Guardians [LISK 79] employ this same notion, but at the level of linguistic support for distributed computation.) For either mode, several types of access mode are supported: read, write, read with intent to write later [GIFF 79].

Other operations include: *unlock*, to unlock an object; *relock*, to change one type of lock to another without unlocking; *read-entry*, to inquire whether an object is locked, and if so, how; and *iter-entry*, to list all locked objects on a node.

An instance of the lock manager exists on every network node, and each lock manager keeps its own lock data base. This data structure records all of the objects, local or remote, that are locked by processes running on the local node. The same structure also records locks that remotely running processes are holding over local objects. Lock and unlock requests for remote objects are always sent to the home node of the object involved, and both the requesting node and the home node update their data bases. The LOCK manager uses the REMFILE manager to handle the remote requests.

The lock manager enforces compatible use of an object by not granting conflicting lock requests. However, it guards against accidental or malicious subversion of the locking mechanism by communicating its current intent to the OSS on a per object basis through the lock key. When an object is locked in a way that excludes any writers, the lock manager sets its lock key to the **readbyall** value. When an object is locked for use by a single writer, the lock manager sets its key to the node ID of the writing process. This causes both reads and writes from any other node in the network to be refused as *concurrency violations*. Today's implementation of the lock manager does not use the **writebyall** value for the lock key, however newly created objects have their lock key initialized to this value.

Locks are either granted immediately or refused; processes never wait for locks to become available, so there is no possibility of deadlock (but indefinite postponement is of course possible). This kind of locking is not meant for distributed database types of transactions, or for providing atomicity in the face of node failures, but for human time span locking uses such as file editing. For this same reason, locks are not timed out, since realistic time outs would be unreasonably long.

6.1. Cache Consistency

In a centralized virtual memory system, the main memory is the single cache over the permanent storage of a file system object. Since all of the users (both simultaneous and serial) of an object run on the same system, the memory cache is common to each of them and so no cache validation need ever be done. When the object is "unlocked" by one process, its pages may stay in the main memory cache for awhile, and if another process comes along to use the same file, that second process will always see the latest version of the object.

In the DOMAIN distributed SLS the simultaneous users of a particular file are either all readers (in which case the data they see is identical), or all processes running on the same node (in which case the main memory cache they see is the same as in the case of a single centralized system). All other simultaneous uses of a file system object are unsupported by the DOMAIN file system. However, we would like *serial* users of an object in the DOMAIN file system to each correctly see all changes made to the file by earlier users.

The simplest demonstration of the problem we faced requires two nodes **A** and **B**. Suppose a one page long file system object **O** resides on a disk that is physically connected to node **A**. A process on **B** locks the object **O** and reads its single page. That page moves through the network from **A** to B and ends up in the main memory of system **B**. After studying the page for some time, the process on **B** unlocks the file and goes about its business. A short time later, another process on **B** wants to read the same file **O**. It locks **O** for reading and accesses that page. We wanted the second user of **O** to be able to dependably use (or knowingly discard) the copy of the page cached in **B**'s main memory. It should be able to use that page (without refetching it from the network) if the file **O** has not been modified since the page was fetched, and it must refetch the page if the file has been modified. In this case, we needed to be able to answer the question: Did a process on **A** modify **O** between the time the page was delivered to **B** and the time the second **B** process wanted to use it? The mechanism described below allows us to efficiently answer that question, and to invalidate the cached copy if it was modified by **A**.

The version number (DTM) kept by the AST for each object can be used to synchronize main memory caches, as follows. The remote user of an object can prove the validity of his cached copy by verifying that the current DTM (as kept by the home node of the object) is identical to the DTM his node has remembered for the cached pages. Should they be different, the locally cached pages need to be invalidated. The lock manager performs this validation at lock time for all remote objects: a request to lock a remote object that is granted returns the current version number (DTM) of the object, which is used in a conditional flush operation, thereby removing stale pages of the object from the requesting nodes main memory.

A second version of the caching problem is to insure that if (extending the example above) the first **B** process to use **O** had modified the object, that the change be available to a process on **A** that wants to use the object immediately after the **B** process releases it. To guarantee correctness in this case, copies of all changed pages of remote objects are delivered back to their home node before the object is unlocked. This function is performed by the lock manager as part of the unlock function: a request to unlock a remote object first purifies the object (forces modified pages back to the home node), then frees the lock to make the object available.

Note that concurrency violations can only occur in multi-node situations: if an object is never locked, and is used by only one node, that node is the only source of version number changes, and will hence always see a consistent view of the current version. This is why the LOCK and HINT managers' state can be stored in virtual memory: the objects that store their code and data do not need to be locked because they are only used on one node.

6.2. Discussion

This two-layer approach to concurrency management has several desirable attributes. First is that it allows the (presumably) more complicated and larger higher level protocol to use the services of OSS to maintain its data base. Second is its flexibility. Changes to the higher-level lock manager can be accomplished without affecting the OSS-level implementation at all. Also, because the operations to manage the cache are exported, clients can implement their own schemes, any number of which can coexist as long as they manage disjoint sets of objects. Lastly, the burden of lock key checking assigned to the per-page operations at the OSS level is very slight compared to the lock manager's data base maintenance.

One restriction that it would be desirable to relax is that the concurrency granularity of the current implementation is at the level of entire objects. The lock key as described is insufficient for some forms of concurrency control. However, if the higher-level protocols wanted to take on the entire control task, the lock key could be set to its **writebyall** value to disable concurrency checking by the OSS-level. Note that the per-object techniques described above, but with a version number (DTM) per page, would allow page level concurrency control. We already store the DTM with each page on backing store; thus keeping one DTM per main memory page frame would suffice for this extension.

7. Naming Objects

For users, UIDs are not a very convenient means to refer to objects; for them, text string names are preferable. However, like UIDs, they should be uniform throughout the network, so that the name of an object does not change from node to node. In DOMAIN, text string names for objects are provided by a directory subsystem layered on top of the single level store. The name space is a hierarchical tree, like Multics [ORGA 72] or UNIX [RITC 74], with directories at the nodes and other objects at the leaves. A directory is just an object, with its own UID, containing primarily a simple set of associations between *component names* (strings) and UIDs. (A symbolic link facility, like that of Multics, is the other major feature of directories.) A single component name is *resolved* in the context of a particular directory by finding its associated UID (if any). The *absolute path name* of an object is an ordered list of component names. All but (possibly) the last are names of directories, which, when resolved starting from a network-wide distinguished "root" directory, lead to the UID of the object. Thus, an absolute path name, like a UID, is valid throughout the entire network, and denotes just one object. (There are other forms of path name besides the absolute form; these *relative* path names are mainly for convenience, since absolute path names are potentially very long in a large network with large numbers of objects. They are all expressible as the concatenation of some absolute path name prefix to the relative path name itself.)

8. Lessons

The first implementation of the DOMAIN system was completed in March of 1981. Since then, the system has been tested, used, and measured extensively. At this writing, the largest operational DOMAIN network system is a single token-ring network consisting of over 600 nodes, and DOMAIN installations of over 70 nodes are not uncommon. As a result of this almost four years of experience, we believe we have learned some important practical lessons – some of which validate (and in some cases vindicate) our choices and others that suggest alternative implementations.

8.1. Choosing SLS

The DOMAIN-chosen technique mapping file system objects into process address space and then turning MMU faults into object read requests of the form (UID, pageno) has been very successful. It enjoys the benefits of simplicity of implementation, stateless remote servers and the efficency of demand-paging lazy evaluation. Further, a single main memory cache management mechanism equally manages object pages for local and remote objects. Our original goal for the remote paging system was to have remote sequential file system I/O take no more than two times longer than the file I/O from a local disk. Over the years, this ratio has averaged around 1.8 to 1.

8.2. Seduction by SLS

The characteristics of network location transparency and a low penalty for remote transparent access combine to make the "map-it, use-it, unmap-it" approach to object manipulation terrifically attractive. However, we have learned that there are sometimes compelling pratical reasons for avoiding the allure of network transparency at the SLS level for some object managers that want to provide a higher level of abstraction.

Our naming server, which implements the directory hierarchy and the name-to-UID translation, was originally implemented completely on top of the location transparent SLS level. As a result, it mapped and operated on directories without regard to their location in the network. The naming server, then, did not, in fact could not, distinguish between directories on local disks and those on remote disks. As a result, the server was straightforward to implement, and as soon as it worked on local directories, it worked on remote directories.

The problem with this implementation strategy for the naming server was that the storage system (naturally) provided no layer of abstraction for the notion of directory. The SLS provided access to the raw bits of a directory to each naming server that wanted to manipulate that directory. This was fine as long as each naming server in the network could operate on directories of the same format. In practice, however, the naming servers are *not* the same on every node in the network (generally due to software updates occuring at different times) and the older naming servers are unable to handle constructs added to directories by newer naming servers running on other nodes.

Directories are an important example for a system like DOMAIN. They are permanent (stored on disk), heavily shared by multiple nodes, and most transactions on them take very little time. Also, they are likely candidates for extensions and improvements over time. Because we can never demand simultaneous update of software on every node in a network, and because we want very much to offer cross-release compatibility, we have found ourselves constrained by our original implementation.

As if that were not enough, we have found that the performance of the naming server tree-walk was significantly increased by asking the node that owned the target directory do the lookup work itself, rather than sending pages of the directory over to the requesting node. This change demanded that the naming server learn the difference between local and remote directories, and was an example of when "moving the work to the data" was a win over "moving the data to the computation."

8.3. Use Simple Protocols

The key to the attainment of our remote performance goals has been the use of light-weight problem-oriented protocols. We have taken full advantage of the relatively clean environment provided by our high-speed ring network to avoid often costly protocol supported reliability.

Operations that are idempotent (i.e. for which repeated applications have the same effect as a single application) use a *connectionless* protocol [SWIN 79] and retry often enough to achieve the desired level of reliability. Network operations to read and write attributes and pages are all of this form.

Operations which are not idempotent (i.e. which have side effects), but which naturally have some state associated with them, can often be made idempotent using a transaction ID. Each time a client sends a new request (not a retry) to perform an operation, it chooses a new transaction ID. If an operation was performed once with a particular transaction ID, the receipt of a second request with the same ID should be rejected. File locking, for example, saves the the transaction ID of the operation which set the lock along with the lock state.

The SLS protocols we use are inexpensive because they are end-to-end protocols [SALT 80] and do not rely on the communications substrate to provide any service guarantees. Instead, each remote operation individually implements the least mechanism required by its reliability semantics.

8.4. Obtaining High Performance

Much has been written on this subject lately for distributed systems. (In particular, see [CHER 83] and [LAZO 84].) The DOMAIN file system has evolved over the years to provide as much as six times the performance of its original implementation. Certainly in the case of completely diskless nodes, but also very frequently in the case of disked nodes, the performance-critical information needed is elsewhere in the network. Our performance goals coupled with our aggressive remote-to-local ratio goal has influenced the implementation in several ways.

The disk subsystem implements fairly familiar techniques for performance enhancement including: physical locality optimizing, control structure caching, batched reads, and clustered writes. Physical locality is encouraged by the increasingly clever allocation of successive file blocks and their file maps and VTOC entries. The basic disk control structures (free-block allocation tables and VTOC control blocks) are cached in their own set of control block buffers. File page reads are "batched" at the SLS-level. Recall that in DOMAIN, all file read activity is caused by touching the bytes of the file with normal CPU instructions and thereby page-faulting on the needed page. When the SLS catches the page-fault and determines the need for some **(UID, pageno)**, it may ask the lower levels for up to 31 additional successive object pages. Most disk write operations are instigated by the page purifier process, and it tries to hand the low-levels a large collection of pages to write so that seek-ordering and rotational-ordering can be performed. In addition, for remote file system I/O, DOMAIN implements trans-network batched reads; a single read page request message may result in as many as eight reply pages in anticipation of their need. In this way, the ultimate client receives more of the benefit of disk page touch-ahead.

We have ended up caching more kinds of information than we originally expected and probably in slightly different ways. In cases where the cost of a disk access would have been barely acceptable, the cost of a network message pair in addition encouraged the use of more aggressive caching strategies.

8.5. Indefinite Postponement

In theory, the remote file server running on one node can service requests from any number of clients. In practice, however, a single server can be flooded with requests from ten, twenty, even one hundred hungry clients. Because the communications protocol layer provides no delivery guarantees to the higher layers, it blithely discards messages it receives after its assorted queues and buffers fill up. In theory, the issuer of the discarded message will send a time-out based retry and all will be well. In practice, indefinite postponement is a definite possibility. As networks get larger, and in particular as server nodes get busier, a solution that formally addresses this problem completely is needed (rather than an ad hoc approach that, for example, increases the depth of the queues periodically).

8.6. Conclusion

The essential ingredients to good performance of a distributed file system include all those things required for a good centralized file system: caching, bulk data transfer from the disk, and good object locality on the disk. In addition, the distributed file system needs more: it needs caching of remote data to avoid as many remote operations as possible; cheap, fast protocols; and bulk data transfer over the network, even when the protocols are very cheap.

REFERENCES

[APOL 81] Apollo Computer, Inc.
Apollo DOMAIN Architecture, Apollo Computer Inc., Chelmsford, Mass., 1981.

[BIRR 82] Birrel, A. D., Levin, R., Needham, R. M., Schroeder, M. D.
"Grapevine: An Exercise in Distributed Computing," *Communications of the ACM*, 25, 4 (April 1982), pp. 260-274.

[BIRR 84] Birrel, A. D., Nelson, B. J.
"Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, **2**, 1 (February 1984), pp. 39-59.

[CHER 83] Cheriton, D. R., Zwaenepoel, W.
"The Distributed V Kernel and its Performance for Diskless Workstations," *Proceedings of the Ninth Symposium on Operating Systems Principles*, October 1983, pp. 128-139.

[DEC 79] Digital Equipment Corporation.

VAX 11/780 Hardware Handbook, Digital Equipment Corporation, Maynard, MA, 1979.

[FREN 78] French, R. E., Collins, R. W., Loen, L. W. "System/38 Machine Storage Management," *IBM System/38 Technical Developments*, IBM General Systems Division, pp. 63-66, 1978.

[GIFF 79] Gifford, D. K. "Weighted Voting for Replicated Data," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 150-159.

[GORD 79] Gordon, R. L., Farr, W., Levine, P. H. "Ringnet: A Packet Switched Local Network with Decentralized Control," *Computer Networks*, **3**, North Holland, 1980, pp. 373-379.

[HOAR 74] Hoare, C. A. R. "Monitors: an Operating System Structuring Concept," *Communications of the ACM*, **17**, 10 (October 1974), pp. 549-557.

[HOUD 78] Houdek, M. E., Mitchell, G. R. "Translating a Large Virtual Address," *IBM System/38 Technical Developments*, IBM General Systems Division, pp. 22-24, 1978.

[IBM 76] International Business Machines Corporation **IBM System/370 Principles of Operation**, GA22-7000-5, IBM, 1976

[JANS 76] Janson, P. A. **"Using Type Extension to Organize Virtual Memory Mechanisms,"** *Technical Report LCS/TR-167*, Laboratory for Computer Science, M.I.T., Cambridge, Mass., September, 1976.

[KOHL 81] Kohler, W. H. "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *Computing Surveys*, **13**, 2 (June 1981), pp. 149-184.

[LAMP 80] Lampson, B. W., and Redell, D. D. "Experience with Processes and Monitors in Mesa," *Communications of the ACM*, **23**, 2 (February 1980), pp. 105-113.

[LAZO 81] Lazowska, E., Levy, H., Almes, G., Fischer, M., Fowler, R., Vestal, S. "The Architecture of the Eden System," *Proceedings of the Eighth Symposium on Operating Systems Principles*, December 1981, pp. 148-159.

[LAZO 84] Lazowska, E. D., Zahorjan, J., Cheriton, D. R., Zwaenepoel, W. **"File Access Performance of Diskless Workstations"**, *Technical Report 84-06-01*, Department of Computer Science, University of Washington, Seattle, WA, June 1984.

[LEAC 82] Leach, P. J., Stumpf, B. L., Hamilton, J. A., Levine, P. H. "UIDs as Internal names in a Distributed File System," *Proceedings of the 1st Symposium on Principles of Distributed Computing*, Ottawa, Canada, Aug. 1982.

[LEAC 83] Leach, P. J., Levine, P. H., Douros, B. P., Hamilton, J. A., Nelson, D. L., Stumpf, B. L. "The Architecture of an Integrated Local Network," *IEEE Journal on Selected Areas in Communication*, **SAC-1**, 5 (November 1983), pp. 842-857.

[LISK 79] Liskov, B. H. "Primitives for Distributed Computing," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 33-42.

[MCKU 84] McKusick, M. K., Joy, W. N., Leffler, S. J., Fabry, R. S. "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, **2**, 3 (August 1984), pp. 181-197.

[NEED 79] Needham, R. M. "Systems Aspects of the Cambridge Ring," *Proceedings of the Seventh Symposium on Operating Systems Principles*, December 1979, pp. 82-85.

[NELS 81] Nelson, D. L. "Role of Local Network in the Apollo Computer System," *Newsletter of IEEE Tech. Comm. on Distributed Processing*, **1**, 2 (December 1981), pp. 10-13.

[NELS 83] Nelson, D. L. "Distributed Processing in the Apollo DOMAIN," *The CAD Revolution*, Second Chautauqua on Productivity in Engineering and Design, (sponsored by Schaeffer Analysis, Inc., Mont Vernon, New Hampshire). Kiawah Island, South Carolina, November 1983, pp 45-51.

[NELS 84] Nelson, D. L., Leach, P. J. "The Architecture and Applications of the Apollo DOMAIN," *IEEE Computer Graphics and Applications*, **4**, 2 (April 1984), pp. 58-66.

[ORGA 72] Organick, E. I.
The Multics System: An Examination of Its Structure M.I.T. Press, 1972.

[POPE 81] Popek, G., Walker, B., Chow, J., Edwards, D., Kline, C., Rudisin, G., Thiel, G.
"LOCUS: A Network Transparent, High Reliability Distributed System," *Proceedings of the Eighth Symposium on Operating Systems Principles,* December 1981, pp. 169-177.

[REDE 80] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. C.
"Pilot: an Operating System for a Personal Computer," *Communications of the ACM,* **23**, 2 (February 1980), pp. 81-91.

[RITC 74] Ritchie, D. M., Thompson, K.
"The UNIX time-sharing system," *Communications of the ACM,* **17**, 7 (July 1974), pp. 365-375.

[SALT 79] Saltzer, J.H., Pogran, K.T.
"A Star-Shaped Ring Network with High Maintainability," *Proceedings of the Local Area Communications Network Symposium,* Mitre Corp, May 1979, pp. 179-190.

[SALT 80] Saltzer, J. H., Reed, D. P., Clark, D. D.
"End-to-End Arguments in System Design," *Notes from IEEE Workshop on Fundamental Issues in Distributed Systems,* Pala Mesa, Ca., December 15-17, 1980.

[SALT 81] Saltzer, J. H., Clark, D. D., Pogran, K. T.
"Why a Ring," *Proceeding Seventh Data Communications Symposium,* October 27-29, 1981, pp. 211-217.

[SPEC 82] Spector, A. Z.
"Performing Remote Operations Efficiently On a Local Network," *Communications of the ACM,* **25**, 4 (April 1982), pp. 246-260.

[STUR 80] Sturgis, H., Mitchell, J., Israel, J.
"Issues in the Design and Use of a Distributed File Server," *Operating Systems Review,* **14**, 3 (July 1980), pp. 55-69.

[SWIN 79] Swinehart, D., McDaniel, G., Boggs, D.
"WFS: A Simple Shared File System for a Distributed Environment," *Proceedings of the Seventh Symposium on Operating Systems Principles,* December 1979, pp. 9-17.

[THOM 78] Thompson, K.
"UNIX Implementation," *Bell System Technical Journal,* **57**, 6 (July-August 1978), pp. 1931-1946.

[WILK 79] Wilkes, M. V., and Wheeler, D. J.
"The Cambridge Digital Communication Ring," *Proceedings of the Local Area Communications Network Symposium,* May, 1979, pp. 47-61.

ANATOMY OF A PAGE FAULT

3/83

This is the story of how the pages of an object are brought into memory. We will concentrate on objects mapped by segments into a process virtual address space.

The tale begins with the mapping of the object (usually through an mst_$map call) somewhere in the address space. The unit of mapping is a segment, so 32 consecutive pages of the virtual address space are reserved by creating an entry in the mst.

The mst is a two dimensional array whose first indice is a process id and whose second indice is an mst entry for an object in that process's address space.

Each time an entry is added to the mst (representing the mapping of a segment of an object in some process's virtual address space), an entry must also be made for that object segment in the ast. The ast is a table used to keep track of 'active' objects; it relates pages of segments of objects to physical memory; it caches static and dynamic information about objects (e.g. where they live and whether they've been modified). There is one ast for the whole system (it is not per-process); its size determines how many objects can have pages resident at a time and is a function of physical memory size.

Back to the mst. An mst entry (mste) contains information about a segment of a mapped object (e.g. the segment number, access rights, its storage location) and it contains a page map (pmap), a table with 32 entries. Each entry in the pmap is used to describe the status of one page in the segment. A page may be:

wired	not available for page stealing
resident	in memory
in_trans	in some sort of transition state, so hands off

Each pmap entry also contains the physical page number for the page or its disk block address if it is not resident.

Mapping an object does NOT cause any of its pages to be brought into memory. Instead, the first reference to a page within the object causes a page fault to occur. (PAGE FAULT: the result of trying to reference a virtual address that is not currently mapped to a physical address). Briefly, the page fault brings you into code which determines that this is indeed a fault on a non-resident page and calls mst_$touch. Mst_$touch does some checking to be sure the page exists (or can be created (object is writable)) and eventually determines that it should call ast_$touch. If the page does NOT have to be created, mst_$touch includes in its request to ast_$touch a count of the number of consecutive pages within the segment it really would like to have resident (beginning with the referenced page). This is the 'touch-ahead' count for the object; it is settable from user space (mst_$set_touch_ahead_cnt) and is used to get better paging performance.

Ast_$touch does a little checking of its own and then calls pmap_$touch, whose job it is (finally) to get the page(s) into memory.

Pmap_$touch determines how many of the pages requested really can be touched by looking at the page map in the ast for this segment. It will only try to touch consecutive pages, starting at the first page requested and stopping at the point that:

1. the count would cause a segment boundary to be crossed
2. a page is found in transition (remember hands off?)
3. a page is found already resident in memory

or 4. a page is found that has not yet been created

Pmap_$touch puts the pages it is going to read in transition (in the pmap) and then allocates enough physical memory to hold the pages (a local subroutine 'alloc' calls mmap_$alloc - but the mmap is another story for another time). Pmap_$touch also determines if the object is local or remote and calls either disk_$read_ahead or network_$read_ahead to trigger the i/o. If there are any errors in the i/o, one or more of the pages requested will be released from transition. Pmap_$touch then installs each successfully-read page in the mmap (by calling mmap_$install) and, in pmap, marks each page as resident and sets its ppn to the physical page number. It then returns the count of pages touched with each page still marked in transition.

Seeing that the pmap touch was successful, ast_$touch returns (to mst_$touch) which installs all the touched pages in the mmu (mmu_$install), clears the in-transition bit for the pages and returns to the fim code which resumes the faulting process, having successfully resolved the page fault.

Somewhat more than this happens of course if the original page cannot be read in, or if there is a concurrency violation in pages received from the network or if a page needs to be created, etc.

A few more words should be said about the locking involved in all this. Most of this work is done under the page resource lock, 'pag_$lock', which must be held whenever a change is to be made to the state of a page (as reflected in the information in the pmap). However, there is another rule that says the page lock cannot be held during i/o (so someone else can get work done while you wait for the i/o). To prevent a page from being stolen or modified by someone else when you have to give up the page lock, the in-transition bit in the pmap must be set. However, this in itself isn't enough. The mmap (remember?) is a table that describes the state of physical memory. It contains one entry for each physical page. This still isn't the time for the mmap story, but suffice it to say that there is some code that doesn't know about the pmap and the in-transition bit, but only knows about the mmap and the avail bit. Any page in the mmap marked 'avail' is eligible to be taken for use. (Available does not mean 'not used', it means 'may be stolen for another use'.) So, to keep a page from being tampered with when you can't keep the page lock, the in-transition bit in the pmap MUST be set AND the avail bit in the mmap MUST NOT be set (call mmap_$unavail).

Further information (and pictures) for most disk data structures and layouts can be found in the section on the File System in the Engineering Handbook. Pascal type definitions are mostly in ins/vol.ins.pas, with a fewer lower level ones in ins/base.ins.pas. Exceptions are noted. Values for particular disk parameters can be found under Peripheral I/O in the handbook.

ALTERNATE LV LABEL

When INVOL initializes a logical volume, it allocates a block (typically the last block on the logical volume) to hold a copy of the logical volume label. The physical volume label contains an array (alt_lv_list) of the physical disk addresses of the alternate lv labels for all logical volumes on the disk.

If the lv label of a volume gets destroyed, it can be regenerated from the alternate lv label with the following steps:

1. Find the daddr of the alternate lv label by reading the pv label and finding the alt_lv_list. If the pv label has also been destroyed, use rwvol to read the blocks at the end of the logical volume (assume that the volume is the maximum number of blocks) and look for a block whose block header uid is 201.0.

2. Use rwvol to read the alternate lv label.

3. Use DB (or MD, if running offline) to patch page number (3rd long word) and daddr (8th long word) as follows:

   page: ??? —> 0
   daddr: ??? —> 1

4. Use rwvol to write out the block to daddr 1.

ASSIGNED DISK

A physical or logical volume whose "ownership" has been assigned to a user process using either the disk_$pv_assign or disk_$lv_assign call. An assigned disk is not used for file system (virtual memory) operations; all i/o to the disk is performed by user programs using the disk_$as_read and disk_$as_write calls. NOTE: even though the disk is under the control of a user program, the physical block format — 32 byte header and 1024 bytes of data — is unchanged. See also Assigned Disk Routines; contrast with Mounted Disk.

ASSIGNED DISK ROUTINES

There are seven routines that are available to handle assigned disks. These routines and their functions are described below (Calling sequences are defined in /us/ins/disk.ins.pas. Argument types and meanings are as described herein.)

disk_$pv_assign — assigns control of a physical volume to the caller and returns the volx of the volume to use in subsequent assigned calls. The caller must supply controller type, controller number, and drive unit number. If known, the size of the physical volume, blocks/track, and tracks/cylinder can be supplied. If they are unknown, the size of the physical volume (b_per_pvol) should be specified as 0, and the appropriate parameters will be returned by the low-level driver. (If the low-level driver doesn't know the disk parameters, you MUST supply them.)

disk_$lv_assign — assigns control of a logical volume and returns the volx of the volume to use in subsquent calls. The volx of the physical volume, which must have been previously mounted or assigned, must be supplied by the caller. The address of the alternate lv label is also returned. (This is because the online SALVOL needs the address of the alternate lv label, but may not be able to read it from the physical volume label if the volume has been mounted.)

disk_$as_read — reads a block from the assigned volume and returns the block header and data. The data buffer must be page aligned. The read is under the control of the assigned options as described under disk_$as_options. Note: Aegis assumes that the caller doesn't know what the block header should contain, so an assigned read will never generate a block header error.

disk_$as_write — writes a block to the assigned volume. The data buffer must be page aligned. The write is under the control of the assigned options as described under disk_$as_options.

disk_$format — the specified track on the assigned volume is formatted.

disk_$as_options — this allows the override of some of the default behavior of the low-level disk routines. Options are:

write_protect — logically write-protects the assigned volume.

no_crc_retry — if a data check occurs during a read, it is not retried (used by FBS).

use_caller_blkhdr — tells Aegis not to touch the block header, in particular not to fill in the dtm, pad, chksum, or daddr fields (used by FBS).

disk_$unassign — relinquishes control of an assigned volume. Any assigned options that have been specified are reset.

BADSPOT CYLINDER

A cylinder, typically one of the last two on a physical disk (see Engineering Handbook), used by INVOL to hold the physical badspot list. The physical badspot list is written out to each head on the badspot cylinder in an attempt to overcome any badspots that might appear on the badspot cylinder.

BADSPOT LISTS

There are two types of badspot lists — physical and logical. The physical badspot list is constructed by INVOL or a disk diagnostic and written out to the badspot cylinder (which see). There is also a logical badspot list contained in the LV label of each logical volume on the disk. This list describes only those badspots which lie within the confines of the logical volume.

BADSPOT MANAGER

A set of subroutines that contain all knowledge about the format of the physical and logical badspot lists. Programs needing to reference the badspot lists (INVOL, SALVOL, FBS) all call the badspot manager to read, write, and update the badspot lists.

BADSPOT

A media defect on a disk that renders one or more blocks unusable for data storage. Most disks we use come from the manufacturer with a list of badspots. (Some storage module packs are guaranteed defect-free; floppies do not have badspot lists.)

When a disk is initialized, INVOL is used to translate the hard-copy badspot list for permanent storage on the disk (see Badspot Cylinder). In some cases, the badspot information is stored on the disk by the manufacturer, and the appropriate disk diagnostic can be used to automatically read this information and construct the physical badspot list on the disk.

As part of disk initialization, INVOL reads the physical badspot list and removes any bad blocks from the Block Availability Table (which see). Note in particular that Aegis knows nothing about badspots; they just appear to be pre-allocated blocks on the disk.

BAT

See Block Availability Table.

BLOCK

See Disk Block.

BLOCK AVAILABILITY TABLE (BAT)

A bitmap describing the current allocation of blocks in a logical volume. The location and size of the BAT is described by the BAT header, which lives in the logical volume label.

Each bit in the BAT describes the state of one disk block — 0 if the block is free, 1 if the block is in use (or is a badspot). The BAT header contains the disk address of the block represented by the first bit in the map.

The BAT is initialized by INVOL during initialization of a logical volume. When SALVOL is run, the BAT is reconstructed using the current state of the VTOC and the badspot list in the logical volume label.

BLOCK HEADER

See Disk Block Header.

BLOCKS_PER_VOL

A disk parameter giving the total number of blocks on a physical volume that are available for the definition of logical volumes. Typically, blocks_per_vol will equal blocks_per_pvol (which see) minus the number of blocks in the badspot and diagnostic cylinders. On some disks, blocks_per_vol is artificially reduced further so that the primary and secondary sourced disks will be of comparable size.

BLOCKS_PER_PVOL

A disk parameter giving the total number of usable blocks on a physical disk volume (contrast with Blocks_Per_Vol).

BOOT

See SYSBOOT.

CALENDAR

An offline (SAU) or online (/COM) command used to set the calendar clock on a node. The calendar utility will also update the last valid time in the logical volume label.

NOTE: calendar should be run on a node before using the offline INVOL to initialize a disk on the node. If this is not done, INVOL will generate invalid UIDs for the disk. (INVOL will check for this in the future.)

CHECKSUM COMMAND

A command (see /usx/com) used to enable, disable, and display the checksum status of the system. The format of the command is:

```
CS [-e | -d] [winchester | floppy | storage_module | network]
```

"-e" enables checksumming for the specified device; "-d" disables checksumming. Only one device can have checksumming enabled at a time. If neither -e or -d is specified, the checksum status of the system is displayed.

When checksumming is enabled for a device, Aegis performs the following actions whenever a block is read or written:

1. Before writing a block, a software checksum is calculated and stored in the block header. The 16-bit checksum is a simple sum of the 512 words of data in the block.
2. After any block is written to the device, it is immediately reread and checked as in #3.
3. When any block is read from the device, if the checksum in the header is non-zero (meaning that it was previously written with checksumming enabled), a new checksum is calculated and compared with the checksum in the header.

When checksumming is enabled, Aegis will crash on any of the following conditions:

read_after_write	(8001C)	Following a write, the subsequent read incurred an uncorrectable disk error or the block had an incorrect block header.
read_chksum	(8001F)	A read (not a read_after_write) failed the checksum test.
read_after_write_chksum	(80020)	A read_after_write failed the checksum test.

CS

See Checksum Command.

CHUVOL (CHANGE_UIDS_ON_VOLUME)

An offline (SAU) and online (in /INSTALL) utility used to change every UID on a physical volume. The need for this procedure arises when a disk is initialized on a node whose node ID is different from the ID of the node to which the disk is eventually to be attached. (For example, manufacturing initializes, loads, and stockpiles DN300 disks without knowing the eventual destinations of the disks.) When Aegis is running, it expects the node ID part of UIDs for local objects to match the ID of the node on which it is running. If these IDs differ, Aegis performance suffers because the algorithm for finding object in the network generates many needless network transmits (trying to find the node that originally initialized the disk).

To prevent this, once a disk reaches its eventual home, CHUVOL is run to "rename" every object on the disk. This involves reconstructing the VTOC and changing the block header of every block in use.

WARNING: CHUVOL should be run only when you have a high degree of confidence in the disk hardware and the file system on the disk in known to be in a consistent state. If there are user files on the disk (i.e., files not replaceable from master release media), they should be backed up prior to running CHUVOL.

CNUM

See Controller Number.

CONTROLLER NUMBER (CNUM)

A number defining which controller of a given controller type you want to talk about. A controller number can be 0 (first controller) or 1 (second controller). Currently, Aegis and the standalone utilities support only one controller number -- 0.

CONTROLLER TYPE (CTYPE_T)

An ennumerated type defining the names of the various controllers that support file system activity. Possible values are

WINCHESTER	(all flavors of winchester disks)
FLOPPY	
RING_XMIT	(use this, not ring_rcv)
RING_RCV	
STORAGE_MODULE	(includes Intel controller and file server disks)
CTAPE	(cartridge tape)

CPBOOT

A command for copying SYSBOOT onto a disk (and the ONLY way SYSBOOT can be placed on a disk -- see also SYSBOOT). Command format is

```
CPBOOT <source-dir> <target-dir>
```

Note that the source and target are pathnames of the directories containing SYSBOOT; do not specify SYSBOOT as part of the pathnames.

CTYPE

See Controller Type.

CYLINDER

A vertical slice through a physical disk. A cylinder contains one or more heads or tracks.

DADDR

See Disk Address.

DCT

See Device Controller Table.

DEVICE CONTROLLER TABLE (DCT)

(Aegis internal) A table internal to Aegis that describes the controllers, ring and disk in particular, that are or may be part of the hardware configuration of the system. Each DCT entry (DCTE) contains the controller number and type, and a set of parameters that are common to all controllers in the table (interrupt vector address, iomap slots, read/write routine addresses, etc.). The DCTE type definition is in ins/io.ins.pas; actual DCTEs are defined in ker/io_tbls.asm.

DIAGNOSTIC CYLINDER

A cylinder — typically the last or next to the last on a physical disk (see Engineering Handbook) — reserved for diagnostic operations by disk diagnostics (offline diagnostics, controller built-in diagnostics, the online TESTVOL program).

DISK ADDRESS (DADDR)

The address of a block on disk, sometimes represented as cylinder/head/sector numbers, but more typically represented as a single DADDR — the sequence number of the block in a physical or logical volume (starting from 0).

DADDR = (cylinder*tracks/cylinder + track) * sectors/track + sector

("track" is the same as "head".)

Disk addresses can be physical or logical. A physical daddr is the absolute address of a block relative to the start of the physical volume regardless of which (if any) logical volume it may be in. A logical daddr is the address of a block relative to the start of the logical volume to which it belongs. So, for example, the physical daddr of the first logical volume label on a disk is 1; its logical daddr is 0. (In general the logical daddrs of all disk addresses on the first logical volume will be one less than their physical disk addresses.)

ALL disk addresses appearing in a logical volume (except those in block headers) are logical disk addresses.

DISK BLOCK

A sector or record on a disk. A disk block consists of a 32-byte software header (see Disk Block Header) and 1024 bytes of data, so the physical block size on disk is 1056 bytes. (Floppy disk blocks have no headers, so the physical block size is 1024 bytes.) For disk block addressing, see Disk Address.

DISK BLOCK HEADER (BLK_HDR_T)

The first 32-bytes of data in any physical disk block (except for floppies, which have no headers). The block header is used by Aegis to verify that the correct block was read and by SALVOL to verify the consistency of the file system. The block header contains the following information:

UID The UID of the file to which the block belongs;

PAGE The page number of the block within the file (the first block is page 0, the second is page 1, etc.);
The UID and page number are sufficient to uniquely identify any block in use.

DTM The time (as a clockh_t) when the block was last written to the disk.

BLKTYP Identifies the block as data (0) or level 1, 2, 3 filemap

SYSTYP Identifies the type of object (file, dir, sysdir).

CHKSUM A software calculated checksum for the data in the block. (This is used only if read-after-write checksumming is turned on — see Read-After-Write Checksum.)

PAD Unused (0's).

DADDR The physical disk address of the block.

DISK PARAMETERS

A set of numbers that describe the size and "shape" of a physical disk volume. These numbers are stored in the physical volume label (which see) of a disk so that Aegis and the standalone utilities can determine the size of a disk without depending on self-identifying hardware on the disk drive. The parameters describing a disk are

DRIVE TYPE
BLOCKS_PER_PVOL
BLOCKS_PER_TRK
TRACKS_PER_CYL
PHYS_SECTOR_SIZE
PHYS_SECTOR_START
SECTOR_DELTA

DISK VOLUME TABLE (DVT)

(Aegis internal) A table setup and maintained by Aegis to describe the state of all mounted and assigned disks on the system. Each DVT entry (DVTE) contains the state of the volume (being_mounted, mounted, assigned), the disk parameters describing the volume, the identity of the current owner, the UID of the volume, and a pointer to the DCTE for the controller of the drive on which the volume resides. For both mounted and assigned volumes, disks are identified by Aegis by a Volume Index (VOLX), which is the index of the DVTE for the disk in the DVT. The layout of a DVTE is in ins/disk.pvt.pas; the actual DVT lives in nuc/disk_wired.pas.

DISK_ERR

An online utility (in /SYSTEST/SSR_UTIL) that prints out information saved by Aegis on most recent unrecovered disk error. The information includes the disk volx, the time, disk address, and physical page number into which the block was read, the error status, and the requested and actual block headers.

DMTVOL (DISMOUNT_VOLUME)

An online command to dismount a mounted volume.

DRIVE TYPE (DTYPE)

A number, which can be passed in to disk_$pv_assign but is more typically set and returned by the lower controller-specific driver, that identifies a particular drive type for a controller that can support more than one

kind of drive (e.g., 30MB and 70MB winchesters).
(Currently, the only disk driver that takes dtype as an IN argument is the floppy driver, for which the drive type is used to differentiate between single and double density floppies -- coming soon from pjl.)

TYPE

See Drive Type.

DVTE

See Disk Volume Table.

EXTENT

A contiguous set of blocks in the VTOC. Each VTOC extent is described by an entry in the VTOC map (which see).

FBS (FIND_BADSPOTS)

An offline (SAU) utility that can be used to construct a badspot list for a physical volume if the original badspot list has been lost. FBS writes and reads several worst-case data patterns to every block on the disk for a user-specified number of passes. The original contents of the disk are, of course, completely hosed.

FILE MAP

A list of (logical) disk addresses that define the locations of the blocks of an object in a logical volume. There are four levels of file maps, referred to as Level 0, 1, 2, and 3. A Level 0 file map points to the first 32 blocks (pages 0-31) of an object and lives in the VTOC entry for the object. A Level 1 file map is 256 entries long and points to pages 33-287 of the object. A Level 2 file map contains up to 256 pointers to further Level 1 file maps for the object. A Level 3 file map contains up to 256 pointers to Level 2 file maps. The first Level 1, 2, and 3 file maps are pointed to by the VTOC entry. The maximum size of an object is thus

(32 + 256 + 256**2 + 256**3) * 1024 = 17,247,300,000 bytes

Level 1, 2, and 3 file maps are each 1024 bytes long and are allocated as required when a file grows. The UID of the block header for file map blocks is that of the owning object; the block type will identify the level of the filemap.

HEAD

One of the n thingamawidgets that sit on disk surfaces and do reads and writes. Number of heads = number of tracks/cylinder.

INTERLEAVING

The physical layout of logically contiguous pages of an object on disk. Since Aegis (and/or the disk controller) typically isn't fast enough to read consecutive blocks from the disk without losing a revolution of the disk, Aegis, when allocating disk blocks to an object, skips one or more disk blocks between consecutive pages of the object. So, for example, pages 6, 7, 8, 9 of a file might be given disk addresses 100, 103, 106, 109, 10C (assuming an interleave factor or Sector Delta of 3). The optimal interleave factor is a function of the speed of revolution of the disk, the amount of work required by the disk driver, and the pattern of reference by the program using the file. Interleave factors range from 2 for a floppy disk up to 9 or so for a storage module on an Intel controller.

INVOL (INITIALIZE_VOLUME)

An offline (SAU) or online (/COM) utility for initializing disk volumes. INVOL has several options that allow initializing logical volumes, entering badspot information, building an os paging file, and displaying the status of the volume. Complete instructions on usage are in some manual.

LOGICAL DADDR

The address of a disk block relative to the start of the logical volume to which it belongs. All disk addresses (excluding those in block headers) on a logical volume are relative to the start of the logical volume. See also Disk Address.

LOGICAL VOLUME

A self-contained and independently addressable entity on a physical volume. A physical disk volume may contain one or more logical volumes, each of which may be mounted (for file system operations) or assigned (for assigned i/o). Logical volumes are numbered starting at 1.

Logical volumes are created using INVOL. The first block of a logical volume is the Logical Volume Label, which contains the name and UID of the logical volume and information about the other structures on the logical volume.
...

LOGICAL VOLUME LABEL (LV LABEL)

The first block in a logical volume (logical daddr 0), holding information about the size and state of the logical volume, headers for other data structures on the logical volume (the BAT and VTOC), and pointers (VTOCXs) to certain standard objects on the logical volume (network root — //, root directory — /, os paging file, SYSBOOT).
The lv label also contains the date-times of last mount, dismount, and salvage (see SALVOL).
See also Alternate Logical Volume Label.
...

LV LABEL

See Logical Volume Label.

MOUNTED DISK

A physical or logical volume that is available for file system (virtual memory) operations. A volume is mounted using the MTVOL command (an exception being the boot volume, which is automatically mounted by Aegis at system startup). Once mounted, all access to the volume is controlled by Aegis via file system and virtual memory paging operations.
See also Assigned Disk.

MTVOL (MOUNT_VOLUME)

The command used to mount a logical volume and catalog the volume in the file system.

NETWORK ROOT (//)

A directory, //, that is initialized by INVOL as part of any logical volume. A pointer (VTOCX) to the network root directory is stored by INVOL in the logical volume label.

OS PAGING FILE

An uncataloged permanent object that must appear on any logical volume that is to be used as the boot device for Aegis. The os paging file is the backing store for those parts of Aegis that are eligible to be paged out to disk. The paging file is built using INVOL, and a pointer (VTOCX) to the paging file is stored in the logical volume label.

PHYSICAL DADDR

The absolute physical address of a disk block relative to the start of the physical volume; see Disk Address.

PHYSICAL VOLUME

A disk, consisting of a physical volume label (first block on the disk, daddr 0), one or more logical volumes, a badspot cylinder, and a diagnostic cylinder. A physical volume can be mounted or assigned. See also Logical Volume.

PHYSICAL VOLUME LABEL (PV LABEL)

The first block — physical daddr 0 — of a physical disk volume. The pv label contains parameters describing the physical disk (see Disk Parameters) and lists containing the addresses (physical daddrs) of each logical volume and its associated alternate lv label.

Since the pv label is the first record on a disk, it can be read without first knowing the exact parameters of the disk, which are normally required to convert a daddr into cyl-head-sector for the low-level disk driver. Aegis and the standalone utilities make use of this fact when mounting (or assigning) a disk on a drive whose parameters are unknown.

PV LABEL

See Physical Volume Label.

READ-AFTER-WRITE CHECKSUMMING

See Checksum Command.

ROOT DIRECTORY (/)

A directory, /, that is initialized by INVOL as part of any logical volume. A pointer (VTOCX) to the root directory is stored by INVOL in the logical volume label. The root directory is the top level of the directory structure for the file system on the logical volume.

RWVOL (READ/WRITE_VOLUME)

A standalone (SAU) or online (/SYSTEST/SSR_UTIL) utility for reading and writing blocks from a physical disk. (To use the online RWVOL, the physical disk cannot be mounted.) RWVOL is a useful tool for examining and repairing parts of the file system. It can also be used to help diagnose failing controllers or drives.

SALVOL (SALVAGE_VOLUME)

A standalone (SAU) or online (/COM) utility for salvaging a disk after a system crash or other occurrence that may have corrupted the file system on the disk. Since many changes to files, the VTOC, and other parts of the file system are not immediately reflected on the disk, a crash may leave the disk in an inconsistent state. For example, a file may have grown (had new blocks allocated to it), but the Block Availability Table (BAT) may not have been updated on the disk.

A logical volume is identified as needing salvage by examining the last-mounted-time, last-dismounted-time, and last-salvage-time, three fields in the logical volume label. If the last mount predated the last dismount, and the last salvage was not performed after the last mount, then the volume was not correctly dismounted and has not yet been salvaged.

The chief operation performed by SALVOL is to scan the entire VTOC on a logical volume and reconstruct the BAT so as to be consistent with the contents of the VTOC. In the process, SALVOL will detect and attempt to fix many other file system errors, for example, multiply allocated blocks (blocks that claim to belong to two or more objects), bad chain pointers in VTOC blocks, and incorrect ACL reference counts.

When booting a node in normal mode, SYSBOOT checks to see if the boot volume needs salvaging. If it does, SALVOL is automatically run before bringing up Aegis.

SECTOR

Same as Disk Block (which see).

SECTOR DELTA

See Interleaving.

STANDALONE UTILITIES (SAUs)

A set of programs that live in the SAUn directory and perform various disk maintainence and diagnostic functions. The standalone utilities are CALENDAR, CHUVOL, INVOL, FBS, RWVOL, and SALVOL (all of which see). Most of these utilities have online versions that can be run under Aegis on an assigned disk (a disk which is not the boot volume and has not been mounted for file system use). Online versions of CALENDAR, INVOL, and SALVOL live in /COM; the online CHUVOL lives in /INSTALL; the online RWVOL lives in /SYSTEST/SSR_UTIL.

SYSBOOT

A program that lives in (physical) disk blocks 02-0B on any physical volume that is to be used as a boot device. SYSBOOT is read from the selected boot device by MD whenever an EX, EY, LO, or LD command is issued. SYSBOOT knows just enough about the file system to be able to find the SAUn directory and read in the requested file. SYSBOOT can also recognize a volume in need of salvaging and, when asked to load Aegis in normal mode, will first execute SALVOL.

Records 02-0B are also the first 10 data blocks of the first logical volume on the disk. These blocks are set aside (marked in use in the BAT) by INVOL when the first logical volume is initialized. INVOL also catalogs SYSBOOT in the root directory of the first logical volume, but DOES NOT copy SYSBOOT onto the logical volume. To do this, the CPBOOT command (which see) must be used. Also, since SYSBOOT occupies a particular physical position on the disk, it CANNOT be replaced by normal file system operations (e.g., CPF).

TESTVOL (TEST_VOLUME)

An online disk diagnostic that lives in /SYSTEST.
...

TRACKS_PER_CYL

A disk parameter defining the number of tracks (heads) per cylinder on a physical disk.

UID

Unique identifier. A 64-bit number that is the unique "name" of any object (file, physical or logical volume, acl, directory, etc.) that lives in or is part of the Apollo file system. Certain objects, since their UIDs must be known a priori, are given "canned" UIDs. In particular the following parts of a disk have canned UIDs:

Physical volume label	200.0
Logical volume label	201.0
VTOC blocks	202.0
BAT blocks	203.0

UNIT

The number of a particular disk drive controlled by a given disk controller. Unit numbers range from 0 to 3, 0 being the number of the first (or only) drive on a controller.

VOLUME INDEX (VOLX)

The number returned by the disk_$pv_assign and disk_$lv_assign calls that is used to identify the assigned volume in subsequent calls for assigned i/o (read, write, format, etc.). (Internally, the VOLX is the index of the assigned volume in the Disk Volume Table, which see.)

VOLUME TABLE OF CONTENTS (VTOC)

A table describing the current contents of a logical volume. The VTOC is an area allocated near the center of a logical volume by INVOL during the initialization of a logical volume. The size of the VTOC is a function of the size of the logical volume and the average file size as specified by the user.

The VTOC is allocated in from 1 to 8 extents, each extent being a contiguous set of blocks. Each extent is described by an entry in the VTOC map, a table in the VTOC header (which is in turn part of the lv label). INVOL allocates the VTOC in such a way as to minimize conflicts with badspots and thus keep the number of VTOC extents to a minimum.

Each block in the VTOC contains up to 5 VTOC entries (which see). Each VTOC entry contains information about an object stored on the disk. The VTOC entry for a particular object is found by hashing the UID of the object (using a hash modulus stored in the VTOC header) to obtain the index of the VTOC block in which the VTOC entry for the object is to be found. (This calculation produces the daddr portion of a VTOC Index, which see.)

If an object is being created, and its UID hashs to a VTOC block that already contains 5 entries, a VTOC extension block (hash bucket) is allocated and chained to the full VTOC block.

VOLX

See Volume Index.

VTOC

See Volume Table of Contents.

VTOC ENTRY (VTOCE)

An entry in a VTOC block describing the attributes and location of an object on a logical volume. A VTOC entry contains the UID of the object, the date/times last used and modified, the current length and the UIDs of the ACL, TYPE, and containing directory for the object (the latter only if the object is cataloged).

A VTOC entry also contains pointers to the first 32 blocks of the object and pointers to the Level 1, 2, and 3 file maps (if any) for the object.

VTOC INDEX (VTOCX_T)

A pointer to the VTOC entry for an object of the form DDDDDX, where DDDDD is the logical daddr of the VTOC block for the VTOC entry of the object and X is the index (0-4) of the VTOC entry in the block.

For example, the pointer to root directory in the VTOC header is a VTOCX. If it has a value of 734D0, then the VTOC entry for "/" is the first entry in physical disk block 734E, assuming the logical volume starts at daddr 1.

VTOC MAP

An array in the VTOC header (in the lv label) describing the location and size of up to 8 VTOC extents. See VTOC.

VTOCE

See VTOC Entry.

VTOCX

See VTOC Index.

WRITE PROTECTED

The state of a mounted or assigned volume that inhibits any writes to the volume. Of the disks supported by Aegis, only floppies and some storage modules have hardware write protect mechanisms. When a volume is write protected (by the -protect option of MTVOL or by disk_$as_options), the protected state is recorded by Aegis (in the DVTE for the volume) and prevents Aegis from attempting writes.

DISPLAY MANAGER

DISPLAY MANAGER

- Definitions
- Data Structures
- IPC Mechanisms
- The Big Picture
- Window Display Fundamentals
- Obscure Windows
- Some typical Sequences
  - Ordinary output to a transcript
  - Carriage return in an input pad
  - Creating a process
  - Opening an edit pad

DEFINITIONS

Display Manager: the program called /sys/dm/dm, that runs in user process 1, together with the program interface in streams. Does not include graphics or device drivers.

Pad: a sequence of elements which are either lines or "frames". Often just a simple ascii text file.

Frame: a two dimensional element of a pad in which x-y positioning and DM mediated graphics are possible.

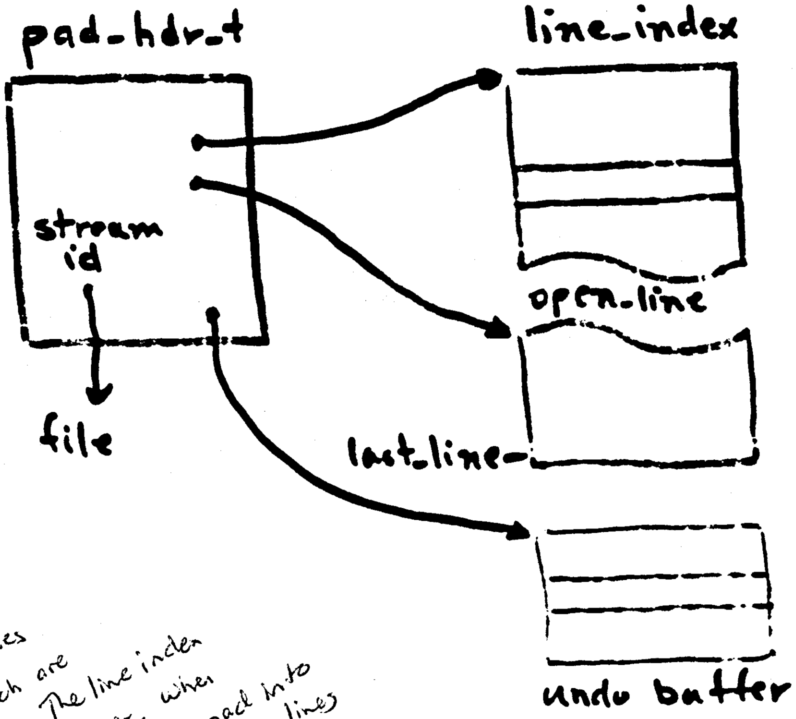
Window: a rectangular region of the screen, containing a banner (a.k.a. legend), a border, and contents which view some part of a pad. Multiple windows may view a single pad.

Pane: a sub-window of a larger window. Panes have borders, but not banners. They behave just like full windows in terms of viewing a pad.

You can get rid of header, banner, and borders

Pad Internals

Display manager uses virtual terminal stuff to read input.



line entry

seek key | heap ptr
byte offset | length
font index
flags: in_heap
has_frame
has_ff

pad is a series of lines which are out of order. The line index table gives the order. when you make a pad into a unix file the lines are ordered.

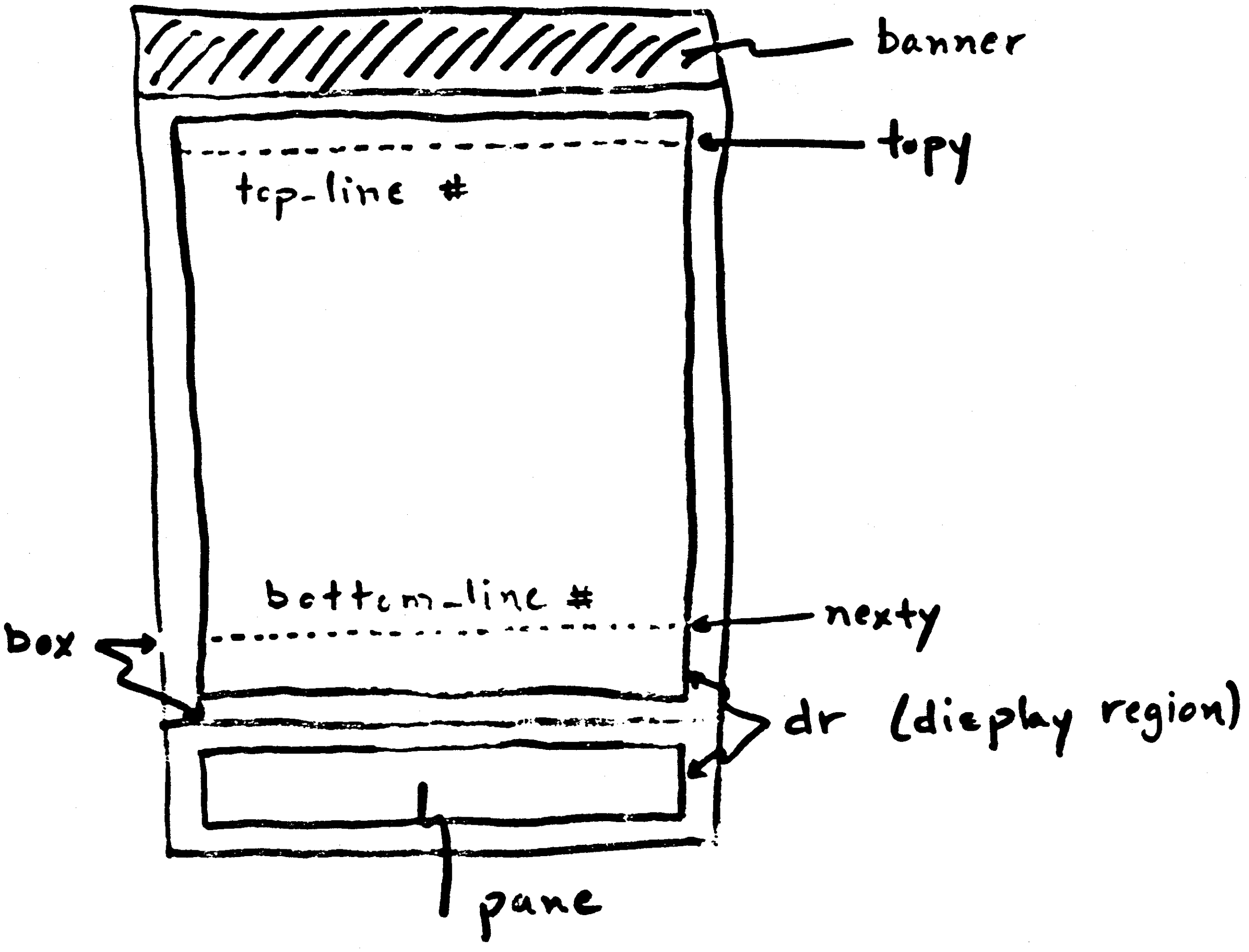
- Line index is an unnamed, temporary file, one per transcript/edit pad (input pads have line index in heap)
- Input pads have mbx channel # instead of stream id. All lines are in heap
- Line index grows by remapping file at ends
- UNDO buffer is a circular array of former line index entries. Freeing heap storage is deferred until entry leaves UNDO buffer.

* Use the Undo command for fast scroll back to previous command that occured before a long transcript output.

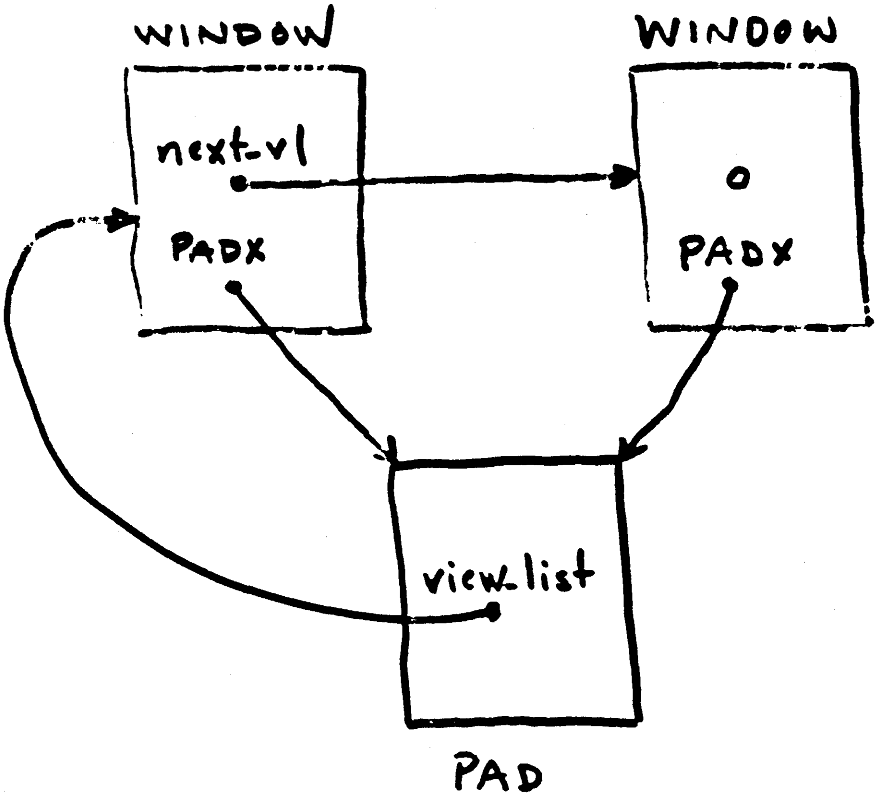
one undo queue for each pad.

a single frame is treated as one line.

WINDOW INTERNALS



WINDOW- PAD RELATIONSHIPS



OPENING AN EDIT PAD (read key)

- Keystroke goes through usual path to parse_cmd with "cv &'Read file:'"

- Discover "&" and call read_sm_input

- Whole DM runs recursively until EN command enqueues pathname response in command window, and inloop discovers it there and returns from read_sm_input

- name$resolve pathname, and look up UID in currently open pads. If found, just call create_window on that pad.

- call stream_$open, and then call create_pad to make a new pad. Mark it read-only (the pad)

- create_pad calls read_more to read the first 100 lines of the file. Read_more calls scan_line for each buffer full, so the same processing gets done as for transcripts

- call create_window

Some size measures

- /sys/dm/dm

  24 modules (1 asm, 23 pascal)
  21124 lines of code
  139K bytes of procedure text
  47K bytes of static data

rev6

no pad or virtual terminal calls in 17

- PAD_$ & VT_$ calls in streams

  15 modules (1 asm, 14 pascal)
  4300 lines of code
  24K bytes of procedure text
  8K bytes of static data

CREATING A PROCESS (shell key)

- Inloop recieves keystroke and passes it to inchar. Inchar discovers definition, and passes "cp /com/sh" to parse-cmd

- Dispatch to CP command

- Read pathname (/com/sh), read arguments (none in this case), build argument list

- Read or build process name ("Process_N" in this case) and check for name conflicts.

- Call pad_sdm_create to create the file for the transcript, and return a stream to it. This file is always temporary + unnamed, until PN command is given.

- Call create_pad to allocate & initialize a pad record.

- Call pad-sdm_create for the input pad. This uses mbx_sopen_by_server. Call create_pad for this too.

CP - continued

- Call pgm_$invoke, passing the input stream_id twice, and the transcript stream_id twice. The streams import/export mechanism gets the streams open in the new process.

- Close the input pad stream. Only needed for export.

- Set the process name and make it an orphan

- Call create_window for the transcript, and relate for the input pane. These use region information or defaults established prior to dispatching to CP command.

CARRIAGE RETURN IN AN INPUT PAD

- Initial conditions: one incomplete line of text is in the input pad, and the user process has written an unterminated output line to the transcript as a prompt. This prompt is not yet displayed anywhere on the screen

- User program calls stream_$get_rec. vt_$get_rec sends input request via smd_$signal, and waits for input in the mailbox.

- Signal processing in inloop:
  - call do.input to check for input already to go
  - since there is none, record the request in the input pad, and call prompt to extract the unterminated line from the transcript and display it in the input window. The entire input window contents are re-displayed. SHOW.WINDOW understands prompts.

- CR keystroke arrives in inloop:
  - call in.char to handle keystroke
  - in.char discovers that the key is defined, and calls PARSE.CMD with the string "EN"
  - parse.cmd calls insert.nl in response to the EN command.

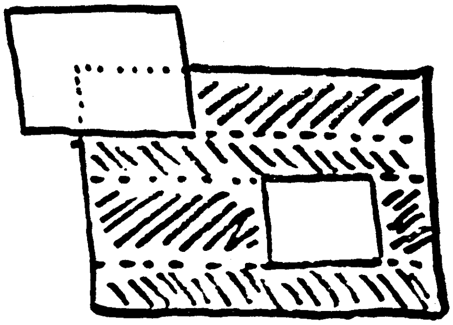
CR - continued

- insert_nl notices that a process is waiting for input on this pad, and that this line can satisfy the request. It removes the line from the pad (ins_lines(-1)), deletes the prompt, and redisplays the (now empty) input window.

- It then writes the line to the transcript, and calls append-pad to update the transcript window display.

Obscure Windows

- Root (non-pane) windows are marked obscure, and a list of visible sub-windows is computed.



- Window contents (pads) can always be redisplayed observing the visible regions, but full redisplay is necessary - the bit blt is not used

- Window borders + banners don't fully observe obscurity, so ctrl-F or screen configuration changes require bottom-up redraw

- Minimum redraw after configuration change would be just those windows overlapped by the window being moved/grown/pushed. Bottom-up redraw requires that any window overlapped by a window being redrawn also be redrawn.

ORDINARY ASCII OUTPUT TO A TRANSCRIPT

- User program calls stream_$put_rec

- vt_$put_rec calls d_file2_$put_rec to append the line to the file, then sends an output request via smd_$signal.

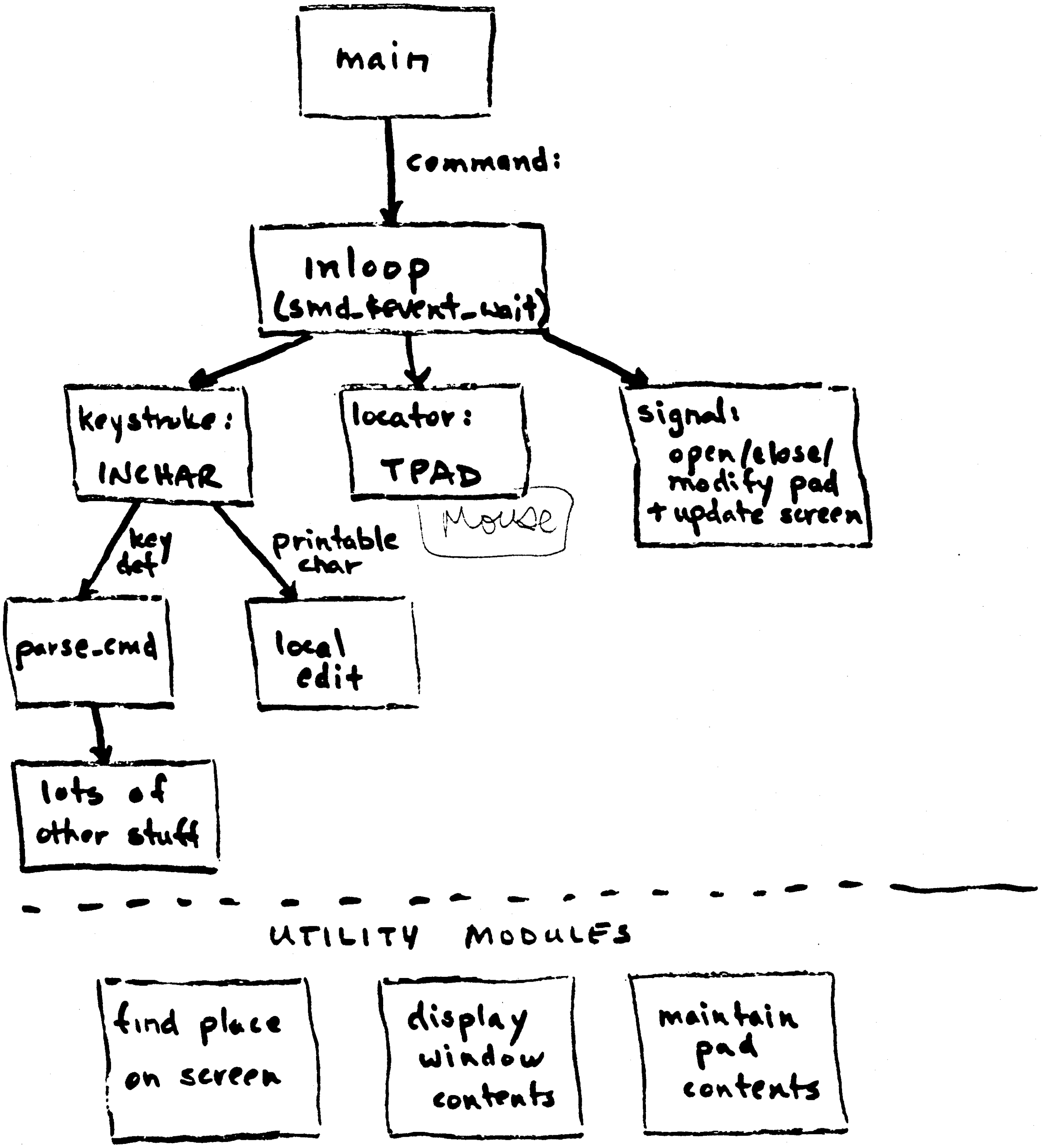
- Inloop receives the signal and calls append_pad

- Append_pad reads the line from the file (via stream_$get_buf, in force_locate mode) and checks for a request sequence.

- Finding none, it calls scan_line to update the line index. Scan_line examines each character and processes newlines, form feeds, bells, and load_font and create_frame requests. It calls ins_lines (+1) as necessary

- Finally, show_window is called, subject to the settings of hold, autohold, scroll, etc. There are some optimizations that avoid calls to show_window for common special cases.

THE BIG PICTURE



Window Display Fundamentals

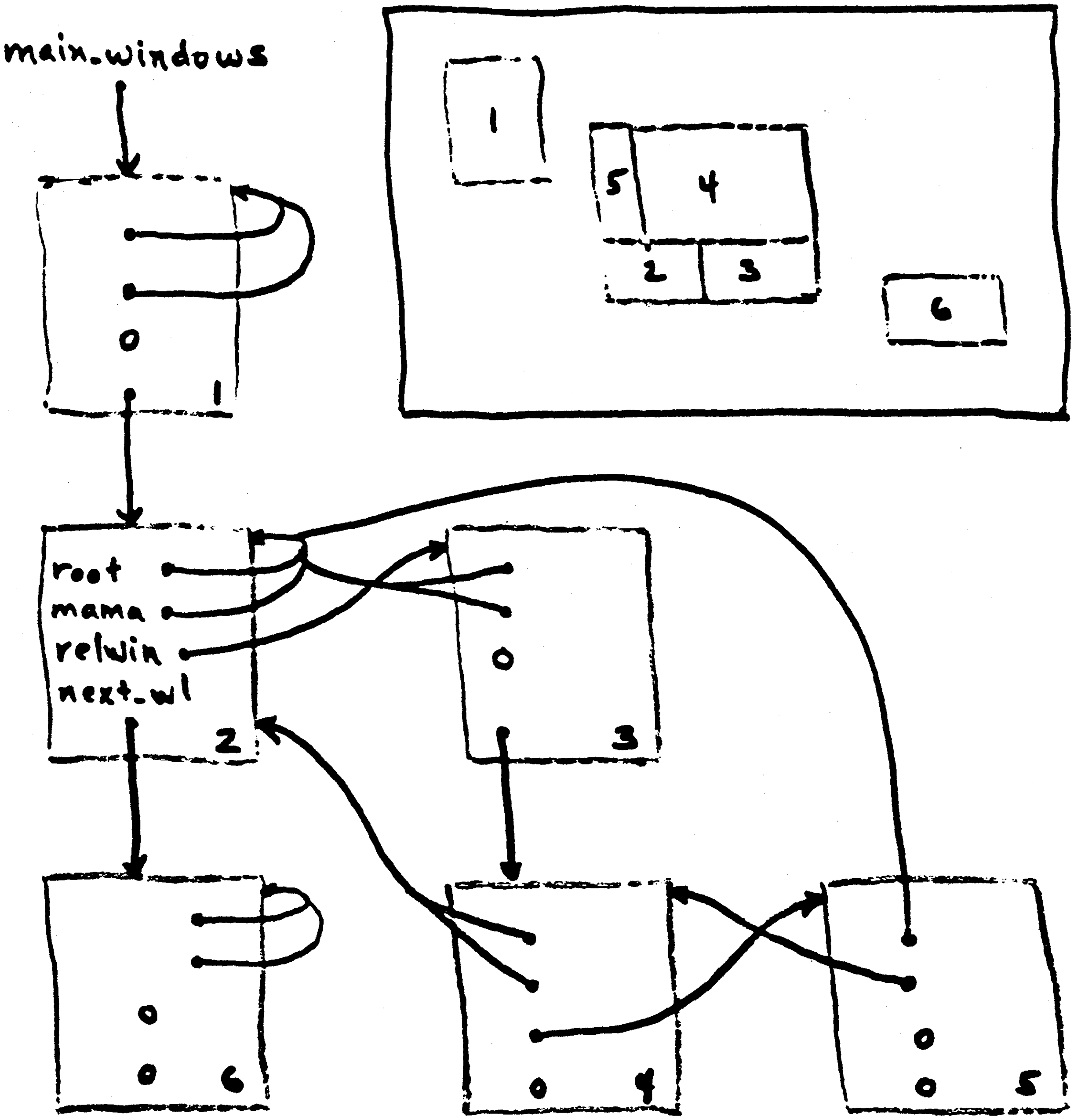
- Typical screen editor approach:
  - make changes to file
  - call general screen update procedure to re-discover changes and modify display

- Complexities of bit-map display, multiple fonts, bit-blt efficiency make this approach more difficult

- General update procedure in the DM:
  - if current image is not up-to-date, we redraw entire window contents
  - otherwise only repositioning is necessary – figure out how to use bit-blt to optimize

- Local changes made inline, by special purpose code
  - insertion/deletion of characters/lines
  - substitute, cut/paste within a single line – for multiple lines, do full redisplay
  - these optimizations require that the cursor be on the line in question & that it be visible
  - local changes must leave window record consistent with actual display

WINDOW - WINDOW RELATIONSHIPS



FILES USED BY THE DM

- Process 1 stack (`node_data/stack)
  - ordinary procedure call stack, etc.
  - 60 pre-allocated window records
  - other DM static variables

- `node_data/PDB
  - heap file UID
  - pad records

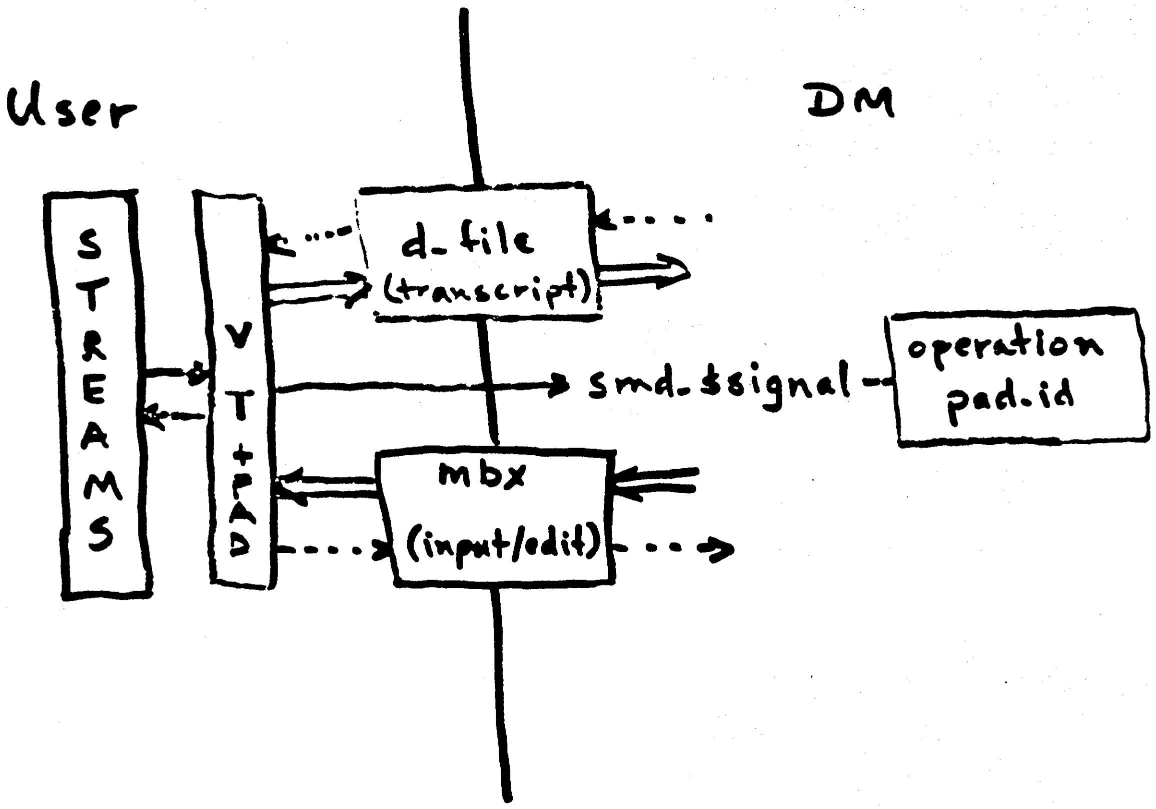
- Heap (unnamed temp file – max size = 2MB)
  - modified lines of edit & input pads
  - line indexes for input pads
  - UNDO buffers
  - key definitions
  - miscellaneous queues & things

- /sys/dm/output + input
  - empty files whose UID's identify the internal DM input + output pads

- startup, startup-login, std_keys, user_data/key-defs, fonts

- `node_data/dm_error_log

IPC Mechanisms



PAD_$ CALLS

- Escape Sequences (simple output operations)
  - pad_$use_font
  - pad_$move
  - pad_$gpr_call (supports most GPR ops in frames)
  - ansi escape sequences
  - these are just put in transcript • no synchronization or reply required. Regular ascii escape is used.

- Request Sequences
  - require a reply, or removal of request from transcript
  - use a different escape character (16#1D)
  - vt_$put_rec rejects ordinary user output beginning with the request character

- All request and escape sequences consist of the request/escape character followed by a one byte printable ascii opcode, followed by arguments in binary (except ansi escapes)