TANDEMCOMPUTERS

A NonStop Kernel

Joel F. Bartlett

Technical Report 81.4
June 1981
PN87603

A NonStop Kernel

Joel F. Bartlett

June 1981

Tandem Technical Report 81.4

Tandem TR 81.4

A NonStop™ Kernel

Joel F. Bartlett
Tandem Computers Incorporated
19333 Vallco Parkway, Cupertino, CA. 95014

June 1981

ABSTRACT: The Tandem NonStop System is a fault-tolerant [1], expandable, and distributed computer system designed expressly for online transaction processing. This paper describes the key primitives of the kernel of the operating system. The first section describes the basic hardware building blocks and introduces their software analogs: processes and messages. Using these primitives, a mechanism that allows fault-tolerant resource access, the process-pair, is described. The paper concludes with some observations on this type of system structure and on actual use of the system.

TABLE OF CONTENTS

INTRODUCTION

Fault-tolerant computing systems have been built over the last two decades in a number of places to satisfy a variety of goals. The results of these differing approaches have been summarized in references 1, 3, and 11 (see page 12). In the past, many of these systems have been designed for specific tasks, such as telephone switching, where the costs of failure are significant. In addition, the designers of most of the systems did not intend to provide general purpose hardware and software modules which the end user would customize to form a reliable system.

An increasing number of online applications in commercial data processing has created a demand for fault-tolerant general purpose computing. These applications are also characterized by their high rate of growth, which requires that the computing system be significantly expanded over its lifetime. The Tandem system is intended to fit these requirements.

HARDWARE ORGANIZATION

A network consists of up to 255 nodes. Each node is composed of multiple processor and i/o controller modules interconnected by redundant buses [2,3] as shown in PMS [3] notation in Figure 1. A node consists of two to sixteen processors, where each processor (Pcentral) has its own power supply, memory, backup battery, and i/o channel (Sio). All processors are interconnected by redundant interprocessor buses (Sipb). Each i/o controller (Kdisc, Ksync, etc.) is connected to two i/o channels and is powered from two different power supplies using a diode ORing scheme. Finally, dual-ported i/o devices such as discs (Tdisc) may be connected to a second i/o controller. The contents of a disc may be "mirrored" on a second volume, but this function is supported primarily by software rather than by hardware. I/o devices other than discs are normally single ported and are connected to one i/o controller.

The processors are 16 bits wide with up to 2mb of memory per processor. The internal clock speed of the processor is 100ns, resulting in a register-to-register add in .6 microseconds and a load of a word from memory in 1.1 microseconds.

The two interprocessor buses (Sipb) provide each processor with two point-to-point paths to each other processor and to itself. Data transfers to and from the buses are buffered in high speed 16-word packet buffers in each processor, allowing data transfers at a rate of 13.5 megabytes/sec. This is more than three times faster than the processor's memory, which assures that interprocessor messages are not delayed by the bus bandwidth.

Hardware Fault Model

The system design goal is to provide continuous operation in the presence of a single fault. This requires that all single faults be detectable, diagnosable, and repairable online. In addition, the software must allow reintegration of the repaired module into the system.

Error recovery analysis needs some assumptions to be made about the types of faults that the system can tolerate. First, a fault in either a processor, its memory, or its power supply must be contained in and therefore at most disable that processor. Second, a fault in either an interprocessor bus or an i/o channel will at most disable that bus. Third, a fault in an i/o controller will at most disable that i/o controller. With these assumptions, it can be seen that a single fault will at most make i/o devices attached only to one controller unavailable.

Physical events affecting the system's hardware can be divided into three classes [1]. The first class, permanent hardware failures, will be detected either on the initial instance of the failure or shortly thereafter by background tests. These give rise to two kinds of problems: error in recovery algorithms and contamination of data bases that takes place before failure detection.

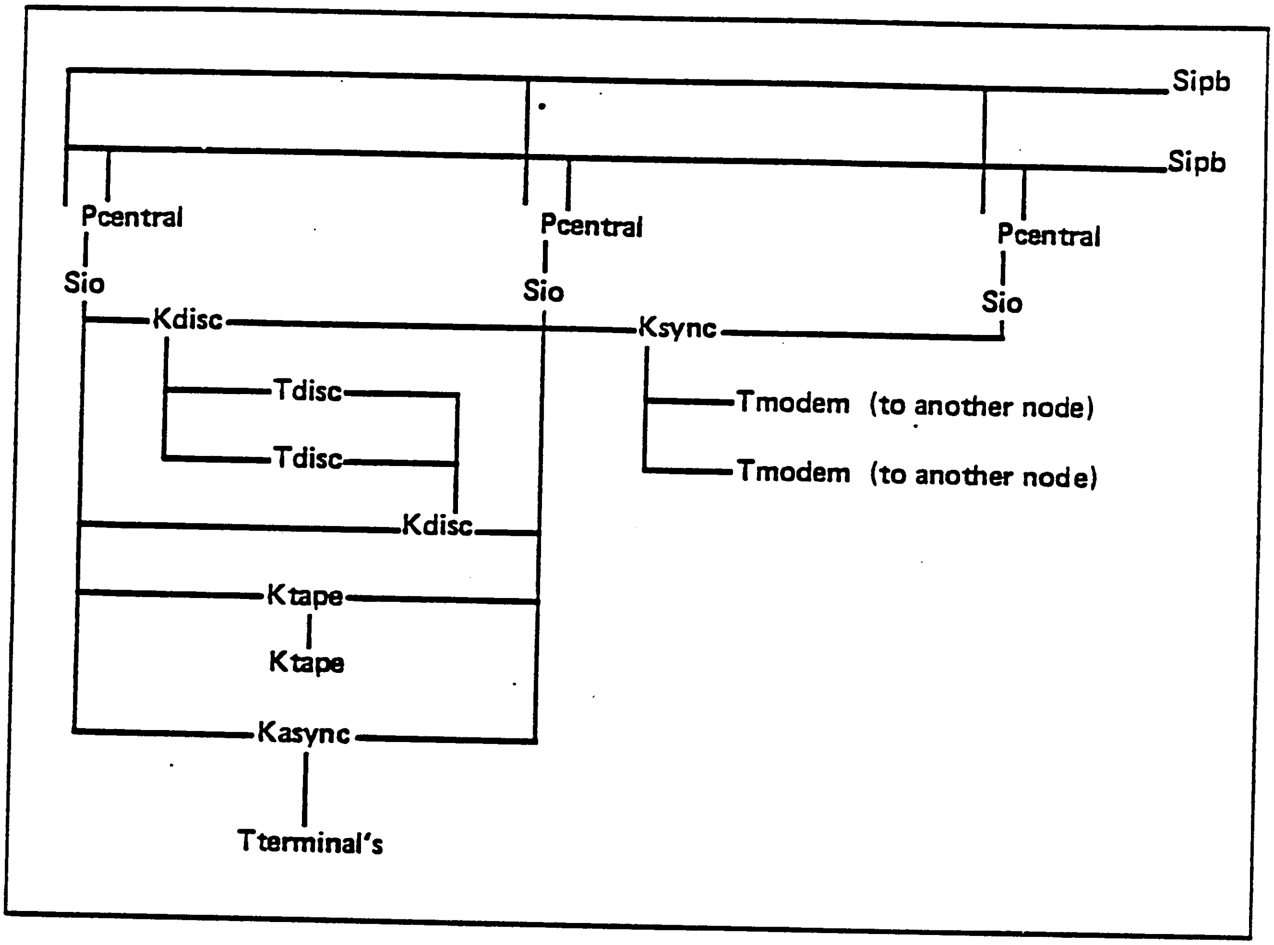

Figure 1. PMS Diagram of a Node.

Intermittent component failures share the previously mentioned problem areas. In addition, unless there is immediate detection, there is a far higher probability of data base corruption as the background tests are much less likely to see the problem.

A final source of problems, and perhaps the most serious in actual operation, is that caused by external interference with the system. This class includes such items as air conditioning failures, but is primarily composed of operational errors by either the computer operator or service personnel.

Sometimes these occur during normal operations, but often their actions are in response to another fault at which point a single misstep may cause the entire system to fail.

SOFTWARE STRUCTURE

The operating system [4] provides many of the user services usually associated with medium-size systems: multiprogramming, access to a gigabyte of virtual memory per processor, a file system, and extensive communications facilities. However, it offers these services in a fault-tolerant manner on a modular, expandable computer system. To do this, its structure has been designed as an analogue of the hardware structure. As the hardware consists of multiple processors and i/o controllers, the operating system functions are distributed over multiple processes; and as the hardware modules are interconnected via redundant buses, the operating system processes communicate via fault-tolerant messages.

At the time this structure was proposed and development started, December 1974, there were few precedents for it. Our main inspiration was found in the work of Dijkstra [5] and Brinch Hansen [6], whose ideas and examples provided the key kernel primitives and structure for the system design.

Processes

Each processor supports up to 256 concurrent processes. Each process has a private data space, but may share code with other processes in the same processor. A major portion of the cost of a process switch is chargeable to memory mapping which is required to designate the new process's code and data spaces. This results in a cost of approximately .5 milliseconds to switch processes.

All processors contain both a monitor process and a memory manager process. The Monitor's functions include process management within its processor (e.g., process creation and deletion), information return, message system control, and fault recovery. When a process is created, it is given a unique "processid" composed of two parts. The first is its location (node number, processor number, and process number) and the second is either a unique timestamp or a symbolic name.

Process synchronization primitives include counting semaphores and process-local event flags. Semaphores are used within the kernel to control such things as access to shared i/o controllers. Event flags are used to signal a process that events such as device interrupt, message arrival, and message completion have occurred. These primitives were chosen using the author's experience at the time rather than following an exhaustive survey of available methods. They have been more than adequate for the resource control involved in i/o operations and in the implementation of the message system. More complex resource control is handled by requesters sending messages to the process which "owns" the resource.

Messages

Almost all information flow, even within a single processor, is carried in messages rather than through shared storage. Each process has a message queue where all messages sent to it by other processes are placed. Messages are queued in either a FIFO manner or according to the sender's priority, at the receiver process's option. The message system is designed to provide a process-to-process communication mechanism which is independent of the location of the processes and transparent to interprocessor bus transmission errors.

A message consists of a request for a service and a reply by the server. For example, if a process wishes to create another process, it calls the procedure NEWPROCESS which in turn sends a message to the Monitor process in the processor where the process is to be created. The sender will then wait for the message reply. When the Monitor retrieves the request from its message queue, it creates the process, and then replies to the message (which awakes the sender) with either the new processid or an error indication.

This model is similar to that discussed by Cheriton in reference 7 and has the desirable property that it is an analog of an operating system procedure that is called by the application program to perform a specific service and return a result. Inherent in it is a positive acknowledgement for each logical request, which is the first key to providing fault tolerance in this system.

In addition, the application program is not allowed direct access to the message system. This is done by providing a conventional user/privileged mode in the processor. User programs may only enter the operating system (and privileged mode) at certain defined entry points, e.g., NEWPROCESS in the previous example. This restriction provides obvious protection and information hiding benefits and allows the operating system to control error recovery on message failures, a second key to the system's fault tolerance.

Message primitives

Before the error recovery strategies can be shown, the kernel's message primitives must be introduced. A message exchange using these primitives takes the following form:

A process "R", the requestor, sends a message to process "S", the server, by calling the procedure LINK. The caller supplies a processid, six message parameter words, and an optional resident buffer which may contain additional data related to the message or may be used to return a result. If the message can be sent, then the address of the sender's Link Control Block (LCB) is returned to R, a matching LCB (which contains the six message parameter words, R's processid, S's processid, and the address of R's LCB) is queued on S's message queue, and S is awakened on the event LREQ (request pending).

When S is ready to process a request, it calls the procedure LISTEN, which dequeues the first LCB from its message queue. The returned LCB contains the six message parameters and the size of the data buffer that the caller supplied to LINK. If there is any data that must be retrieved from R before the operation is performed, S calls the procedure READLINK with the addresses of its data buffer and LCB. READLINK copies the data from R's buffer to S's buffer.

S then performs the appropriate operation and returns the results to R by calling the procedure WRITELINK with the appropriate LCB and, optionally, a buffer whose contents are to be returned. Prior to this call, S may alter the message parameter words in its LCB to return status to R. WRITELINK causes S's buffer and message parameter words to be copied into R's buffer and LCB. A "done" flag is set in R's LCB, and then R is awakened on the event LDONE (request complete).

At this point, S is finished with the message and may go on to other things. R can examine the results of the request and then return its LCB and buffer (if any) to the system by calling BREAKLINK.

Message properties

The previous example illustrates several key aspects of the message system. First, all messages are sent by value. By having no shared data structures, the message system looks the same irrespective of the relative locations of the processes R and S.

Since information transfer between R and S only occurs via message system primitives, which in turn only work when the supplied LCB's match (processid's for R and S and R's LCB address are identical in both LCB's), the message is always abortable by either R or S. The requestor, R, can terminate the message by calling BREAKLINK. If the message has not been completed, a cancel flag will be set in S's LCB and S will be awakened on the event LCAN (request cancellation). The server, S, may also terminate the message by setting the cancel flag in its LCB and calling WRITELINK without specifying a buffer. This will result in the requestor being awakened on LDONE with both the completion and the cancel flag set in its LCB.

Completion of a message with the cancel flag set provides a uniform mechanism for signaling failures. It allows outstanding messages to be cleaned up on a process or processor failure by setting the cancel flag, mimicking cancellation by the failed end of the message.

Certain system status messages that need no reply, such as processor failure or reload, are sent by system processes to application processes. It is important that the system process not be blocked if the application process does not pick up the message. Hence the sender calls BREAKLINK immediately after calling LINK, terminating its end of the message. This technique can only be used when the information can fit in the six message parameter words.

A server process is free to pick up multiple requests through LISTEN, queue them internally, and not READLINK or WRITELINK the request until it is ready to process the request or reply to it. For example, a process controlling a disc may need to block certain requests that need to access a locked record until the record is unlocked.

Conversely, a process may have multiple LINK's outstanding at any time. This allows an application process to keep a read request posted for each terminal that it manages, processing the input data as requests complete.

Message system resource control

In a message-based system, allocation of control blocks and buffers is a potential source of resource contention and deadlock. No formal limits on process interaction via messages have been defined. This approach allows flexible application design, but at some risk. The resource allocation strategies that have been devised are not "correct" in a formal sense, but they do minimize this risk.

First, LCB allocation is controlled by providing both "reserved" and "pool" LCB's. A process may reserve some number of LCB's to queue incoming messages and some number of LCB's to send messages. If a process has all of its reserved (possibly none) LCB's in use, then pool LCB's will be allocated if they are available. If an LCB cannot be obtained within 10 seconds, then the call to LINK will fail. System server processes reserve one or more LCB's for incoming messages and a sufficient number (dependent solely on the servers' needs during request processing) for outgoing messages to assure that they can complete any request made of them.

Message buffer allocation is managed by several techniques. First, data buffers for incoming messages are not allocated until the server process is ready to request the data via READLINK. Second, allocation is made from different storage pools on the basis of the type of request. For example, buffers for reads from terminals are separated from buffers for disc requests since these two buffer types are usually held for significantly different periods of time. In addition, certain types of system server processes have permanently allocated buffers so that they may always service requests.

Interprocessor bus protocol

While messages between two processes in the same processor can be sent using standard mutual exclusion primitives and moves, messages between processes in different processors must flow over the interprocessor buses.

In order to make messages useful as an abstraction, it is important that the message primitives recover from bus errors and fail only if the other process does not exist, the other processor is down, or there are no free LCB's. In addition, localized error detection and correction is required for fault isolation and repair. These arguments for local robustness should not be interpreted as arguments against end-to-end checking as encouraged by Saltzer in [7]. On the contrary, the author believes that they are both necessary in fault-tolerant systems.

From an implementation and confidence viewpoint, it is desirable that the error recovery scheme be as simple as possible yet still detect misrouted, inserted, or lost packets. In addition, the two physical buses must support an arbitrary number of logical connections between processes so that transmissions can be multiplexed.

With these requirements in mind, a bus protocol, similar in spirit to HDLC [8], was designed which uses sequenced packets and positive acknowledgements. When the sender processor sends data, it is divided into 16-word packets consisting of the sender and receiver processor numbers, a sequence number, 13 information words, and a checksum. Following transmission, the request waits on the wait acknowledge (WACK) list. If the request is still on the WACK list after one second, it is resent on the other bus. This cycle repeats until either the transmission is acknowledged or the receiving processor is declared down. Repeated failures to acknowledge transmission over a bus to another processor cause that path to be marked as down. The sender processor may send up to three logical transfers ahead of the last acknowledged logical transfer. Additional transfers will be queued until the previous transfers are acknowledged.

Each processor maintains a Bus Receive Table (BRT) entry for incoming data from each processor, which includes a buffer address, a transfer count, and the next expected sequence number. When a packet arrives on either bus, it is checked for correct routing, moved into memory as its checksum is computed, and the checksum and the sequence number verified. A good packet causes the BRT entry to be updated. When the transfer count becomes zero or a packet error occurs, an interrupt is posted. On detecting a bus receive error, the processor need only note the type of error and flush the packet; error recovery is the responsibility of the sending processor.

An area of concern in this error recovery mechanism is the time lost to timeouts or packet flushing. This time is minimal, as the bus error rate is very low: errors are only observed when a hardware fault has occurred. For example, a month's error log for the system that this paper was prepared on contained no bus error messages. Given this error rate, the correctness of the error recovery scheme is far more important than its efficiency.

An example of the low-level protocol is the action taken when a server, S, WRITELINKs data to its requestor, R. First, the request is queued on the Send Data List (SDL), and a sequence number assigned to it. Then S is suspended, the request is added to the WACK list, and a here's data back (PHDB) control packet and the data block packet(s) are sent to R's processor.

R's processor sees a bus receive interrupt for the PHDB packet, sets the BRT buffer address to R's buffer, sees a data completion interrupt after the data block packet(s) have been received, and then queues an acknowledgement for S's processor. The acknowledgement is either sent as an unsequenced control packet or piggy-backed on some control packet that is part of another request. When the packet holding the acknowledgement arrives, S's processor completes the request and activates S.

Every second, each processor sends an unsequenced acknowledgement packet over each bus to every processor. This packet has two purposes: to recover from lost acknowledgements and to tell the other processors that this processor is up. Every two seconds, each processor checks whether it has received an unsequenced packet from each other processor. If not, it considers that processor to be down, and cancels all messages from it as described earlier.

Message system performance

Sample program segments and performance data are shown below. Neither process does any processing on the message, and message buffers are preallocated. The requestor process executes:

```
while true do
  begin
  LINK( ... );
  wait for message completion;
  BREAKLINK( ... );
  end;
```

and the server process executes:

```
while true do
  begin
  wait for a message;
  LISTEN;
  READLINK( ... );
  WRITELINK( ... );
  end;
```

with the results:

READLINK	WRITELINK	Elapsed time/msg (ms)	
(bytes)	(bytes)	intracpu	intercpu
0	0	2.1	2.6
0	200	2.3	2.9
200	0	2.4	4.2
2000	2000	4.6	7.0

The asymmetry between the second and the third example occurs because a WRITELINK must always be performed to complete the message, but a READLINK is only performed when data must be moved from R to S.

PROCESS-PAIRS

Processes and messages provide a method for hiding processor boundaries and inter-processor bus errors. Associating a process with a resource provides a method of addressing and accessing it. The next step is to build a protocol that provides fault-tolerant access to the resource, protecting against information loss due to a processor failure.

A pair of processes and a symbolic name are associated with each i/o device or application server process. Requests are sent to the "primary" process of the "process-pair", which handles the request and controls the resource. When the primary receives a request for an operation such as a file open or close, the primary process "checkpoints" the request to the "backup" process via the message system. These checkpoints ensure that the backup process has all information that it will need to assume control of the device in the event of an i/o channel error or a fault of the primary process's processor. When the primary fails, the backup process "takes over" and becomes the primary.

The message system directs messages to process-pairs as follows. Each processor maintains a name table, associating a symbolic name with two processids in the node. When a message is sent to a named process, the first processid of the pair is used. If that process does not exist or is not currently the primary process, then the message fails; the processids in the table are exchanged, and the message is resent to the other half of the process-pair.

Error Recovery Using Process-Pairs

Unfortunately, not all requests for service are arbitrarily retryable. For example, writing a record into a key-sequenced file causes a duplicate key error to be returned if the record already exists. Hence, it is important that a server process-pair do the actual processing of a non-retryable request exactly once (see [7] for a discussion of this problem of "atomicity"). Requests are assigned sequence numbers and the server is required to track non-retryable requests by saving their sequence numbers and status, so that it may simply return the status on a duplicated request.

For example, let R and R' be primary and backup requestor processes and S and S' be primary and backup server processes. A sequence number is kept for each opener of a file by both the requestor and the server. When the file is opened, the initial state has all sequence numbers equal to zero. When R wishes to write a record into a disc file controlled by S, R sends S a message:

(1) $R'_{seq=0}\ \ R_{seq=0}\ \ -\ \ S_{seq=0}\ \ S'_{seq=0}$

S picks up the message and performs the following check to see if this is a redundant operation:

(2) if requestor seq < my seq then return saved status

The amount of previous status that must be saved here is dependent upon the processes involved. Here, S is the standard disc process so R would have had to specify the number of previous results to save at the time the file was opened.

If the check for a saved result fails, then the operation is performed. S reads the disc block, checks if the record already exists, and then checkpoints the request and the new block to its backup, S'.

(3) $R'_{seq=0}\ \ R_{seq=0}\ \ S_{seq=0}\ \ -\ \ S'_{seq=0}$

The block is written to disc(s), the completion status is saved and checkpointed to S', and both S and S' increment their sequence number.

(4) $R'_{seq=0}$ $R_{seq=0}$ $S_{seq=1}$ — $S'_{seq=1}$

The result is then returned to R, who also increments his sequence number.

(5) $R'_{seq=0}$ $R_{seq=1}$ — $S_{seq=1}$ $S'_{seq=1}$

Finally, R checkpoints the result to R', who increments his sequence number, returning the system to a state indicating that there are no requests in progress.

(6) $R'_{seq=1}$ — $R_{seq=1}$ $S_{seq=1}$ $S'_{seq=1}$

Failure recovery during a request takes the following forms:

First, R' or S' may fail during the request without affecting the operation, as it can be carried out even if the checkpoints fail.

Second, if R fails following step (1), then S performs the operation, but is unable to return the result. When R' becomes R, it repeats the request starting at (1); but since its sequence number is still zero, the test in (2) returns the result that would have been returned had R not failed, and the operation is not repeated.

Finally, if S fails during the operation, S' becomes S and either does nothing or completes the operation using the checkpointed information, saving the completion status and incrementing its sequence number. When R resends the request (1) to the new S, it either does the operation or returns the saved result.

It should be emphasized that this checkpoint and error recovery process is independent of the location of the processes, the number of processors in the system, and the other message traffic in the system. By confining the recovery actions to the processes directly involved, the mechanism is both simple and arbitrarily expandable.

Process-Pair Maintenance

Process-pairs for i/o processes are created at system configuration time. A memory image for each processor is created with all necessary data structures. At initial load time, a processor is loaded from the disc. Each configured process will test to see if its other half exists, and observing that it does not, it will become the primary. An initial command interpreter will be created by the Monitor process which can then be used to issue RELOAD commands to load the rest of the processors.

When a processor is reloaded at an initial load or following repair, the configured processes will start execution. Each will observe that the other half of its process-pair is up, and therefore it is the backup. While this is going on, the process which reloaded the processor will be notifying all interested processes in the system that the processor is now up. The primary of each process-pair will then checkpoint its current state to the backup.

Application process-pairs are created in a more general manner. An initial primary process will be created in a processor. The process will in turn select some other processor and create its backup there. If the primary fails, then the backup will take over and it may select another processor to create a new backup, or, as is usually done, it will wait for the original processor to be repaired and reloaded before creating a backup. The primary of any process-pair may switch roles with its backup. This is generally done to balance the load on a system following the repair and reintegration of a processor.

OBSERVATIONS

The system does tolerate faults, they are repaired online and reintegrated into running systems. More specific reliability and availability statements are best left for our users to make.

System performance is likewise difficult to comment upon. In the example of section 3.1, the process R might be able to originate ten of these requests per second. However, it must be emphasized that there is no such thing as a "standard transaction" and specific applications must be examined for accurate measurements. The reader is directed to [9] for the description and results of an extensive benchmark involving this and other systems.

A system structured around process-pairs communicating via messages has already faced some of the major problems involved in distributed computing systems: decentralized control, partial system failure, and reintegration of repaired components. Two logical extensions to the system have been done to allow distributed systems to be connected into a network: the inclusion of a node specifier in processids, and the extension of message destinations to include processes in other nodes with communication provided by more conventional data communications equipment.

The advantages and disadvantages of multiple-processor computer systems have been explored in some detail in [11] and need not be restated here. However, there is no question that there is some cost associated with the use of a message system rather than shared memories in either a uni- or a multiprocessor configuration. The additional system resources consumed by a message system are processor time and memory, which are becoming the cheapest part of the system. This "penalty" has to be offset against the system's fault tolerance, the ease of system expansion, and the possibilities for system evolution (as shown by the addition of a network capability) that such a structure provides.

It is reasonable to ask whether users can write application process-pairs. Many do, and if the design has been done carefully, they will recover correctly from single faults. A detailed discussion of the various techniques for checkpointing is presented in [10]. However, the use of higher-level tools for terminal management functions and transaction backout [12] allows users to write server processes in COBOL which do not have backup processes, do not issue checkpoints, nor concern themselves with failure recovery. Thus, the application's fault tolerance need not rely heavily on the user's abiiity to design and implement correct error recovery.

Unfortunately, the hard problem is not the handling of the fault tolerance, but the design of a reasonable online system. Many user's initial reaction to this design problem is to try to create a monolithic application program to do all functions. Such a user often is from a batch data processing shop which is attempting to develop its first online application. This approach may get early results, but the program will be very hard to maintain and it will not scale up as it cannot be split across multiple processors for higher throughput. Another reaction (which sometimes follows the first) is to divide the application into countless processes which all must be used to process the simplest terminal inquiry. This is often combined with a very ornate key-sequenced data base. The result of this can be an application which handles two transactions per minute.

Application design and system sizing are still art rather than science. We have developed tools and guidelines for application design, for simulation and modelling to aid in system sizing [13], and for measurement of the resultant system [14]. These are extensively used in developing quotes for systems and tend to eliminate disappointment upon system delivery.

Transaction systems have needs that differ somewhat from those of general purpose computing systems. The primary function of a processor in a transaction system is to move data between discs and terminals, with a surprisingly small amount of processing done along the way. The high demands for message and i/o buffers may force the purchase of additional processors to gain buffer address space, rather than to increase processor power.

Therefore, extensions to the system architecture to improve memory access can be more useful than reimplementation of the hardware for greater speed. The former direction has been taken by our NonStop II system, which is a compatible extension of our 16-bit architecture to provide a 32-bit logical address space. This additional addressing capability eliminates many processor limitations that could be traced to small address spaces.

The decision to build a complex system based entirely upon processes interacting via messages has been the proper one for us. It has allowed us to construct reliable transaction systems differing in cost by an order of magnitude which can be constructed with the same modules and run the same software. It has also provided a flexible base on which to do significant software extensions without major redesign of existing system software.

ACKNOWLEDGMENTS

This kernel has evolved over a period of time and it reflects the contributions of many people. In particular, I would like to acknowledge the efforts of Dennis McEvoy, David Hinders, Jerry Held, Mike Green, Harald Sammar and Wendy Bartlett. I would also like to thank Wendy Bartlett, Jim Gray and the referees for their criticism which improved the presentation of this material.

REFERENCES

[1] Avizennis, A., *Architecture of Fault-Tolerant Computing Systems*, FTC-5, IEEE and F.N.I.E., Paris (June 1975), pp 3-16.

[2] Katzman, J. A., *A Fault-Tolerant Computing System*, Eleventh Hawaii International Conference on System Sciences (January 1978), pp 85-102.

[3] Siewiorek, D. P, Bell, C. G., and Newell, A., *Computer Structures: Readings and Examples*, McGraw-Hill, Inc. (1982).

[4] Bartlett, J. F., *A NonStop Operating System*, Eleventh Hawaii International Conference on System Sciences (January 1978), pp 103-117.

[5] Dijkstra, W., *The Structure of the "The Multiprogramming System,"* Comm. ACM 11 (May 1968), pp 341-346.

[6] Brinch Hansen, P., *The Nucleus of a Multi-programming System*, Comm. ACM 13 (April 1970), pp 238-241, 250.

[7] Liskov, B., *Report on the Workshop on Fundamental Issues in Distributed Computing*, Operating Systems Review (July 1981), pp 9-38.

[8] CCITT, *Recommendation X.25, Level 2*, Geneva, (1976).

[9] Gleser, M. A., Bayard, J., and Lang, D. D., *Benchmarking for the Best*, Datamation (May 1981).

[10] Tom, G. F., *Checkpointing Techniques for Fault-Tolerant Process-Pairs*, BS Thesis at MIT (June 1981).

[11] Enslow, P. H. Jr., *Multiprocessor Organization—A Survey*, Computing Surveys, Vol 9, Number 1 (March 1977), pp 103-129.

[12] Borr, A. J., *Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Processing*, 7th International Conference on Very Large Data Bases (September 1981).

[13] Blake, R., *Tailor: A Simple Model That Works*, Proc. of Conf. on Simulation, Measurement, and Modelling of Computer Systems, ACM SIGMETRICS, Boulder (August 1979).

[14] Blake, R., *XRAY: Instrumentation for Multiple Computers*, Proc. Int'l. Symp. on Computer Performance, Modelling, Measurement, and Evaluation, ACM SIGMETRICS and IFIP WG7.3, Toronto (May 1980).

Distributed by
TANDEM
Corporate Information Center
10400 N. Tantau Ave., LOC 248-07
Cupertino, CA 95014-0726