```
-- CheckPoint.Mesa
-- Edited by:
--              Sandman on Jul 25, 1978 9:23 AM

DIRECTORY
  AllocDefs: FROM "allocdefs" USING [
    AddSwapStrategy, RemoveSwapStrategy, SwappingProcedure, SwapStrategy],
  AltoDefs: FROM "altodefs" USING [
    BytesPerPage, PageCount, PageNumber, PageSize],
  AltoFileDefs: FROM "altofiledefs" USING [CFA, FA, fillinDA, FP, TIME, vDA],
  BcdDefs: FROM "bcddefs" USING [VersionStamp],
  BFSDefs: FROM "bfsdefs" USING [ActOnPages, GetNextDA],
  ControlDefs: FROM "controldefs" USING [
    Alloc, AllocationVector, AllocationVectorSize, ATPreg, AV, ControlLink,
    EntryVectorItem, FrameHandle, FrameVec, Free, GetReturnLink, GFT,
    GFTIndex, GlobalFrameHandle, Greg, Lreg, MaxAllocSlot, OTPreg, ProcDesc,
    SD, StateVector, SVPointer, WDCreg, XTSreg],
  CoreSwapDefs: FROM "coreswapdefs" USING [PuntInfo, GetLevel, SetLevel],
  DirectoryDefs: FROM "directorydefs" USING [EnumerateDirectory],
  DiskDefs: FROM "diskdefs" USING [DA, DiskRequest, RealDA],
  DiskKDDefs: FROM "diskkddefs" USING [CloseDiskKD],
  FrameDefs: FROM "framedefs" USING [MakeCodeResident, SwapInCode, SwapOutCode],
  ImageDefs: FROM "imagedefs" USING [
    FileRequest, FirstImageDataPage, HeaderPages, ImageHeader, ImagePrefix,
    ImageVersion, MapItem, PuntMesa, UserCleanupProc, VersionID],
  InlineDefs: FROM "inlinedefs" USING [BITAND, COPY],
  LoadStateDefs: FROM "loadstatedefs" USING [
    ConfigIndex, GetInitialLoadState, GetLoadState, InputLoadState,
    ReleaseLoadState],
  MiscDefs: FROM "miscdefs" USING [DAYTIME, GetNetworkNumber, SetBlock, Zero],
  OsStaticDefs: FROM "osstaticdefs" USING [OsStatics],
  ProcessDefs: FROM "processdefs" USING [
    ActiveWord, CurrentPSB, CurrentState, CV, DisableInterrupts, DIW,
    EnableInterrupts, ProcessHandle, Queue, ReadyList, SDC, WakeupsWaiting],
  SDDefs: FROM "sddefs" USING [sAllocTrap, sSwapTrap, sXferTrap],
  SegmentDefs: FROM "segmentdefs" USING [
    AddressFromPage, Append, CloseFile, DataSegmentAddress, DataSegmentHandle,
    DefaultBase, DefaultVersion, DeleteDataSegment, EnumerateDataSegments,
    EnumerateFiles, EnumerateFileSegments, FileError, FileHandle,
    FileSegmentAddress, FileSegmentHandle, GetFileSegmentDA, JumpToPage,
    MapFileSegment, NewDataSegment, NewFile, Read, SetEndOfFile, SwapIn,
    SwapOut, Unlock, Write],
  StreamDefs: FROM "streamdefs" USING [
    CreateWordStream, ReadBlock, StreamHandle],
  StringDefs: FROM "stringdefs" USING [EquivalentString],
  SystemDefs: FROM "systemdefs" USING [AllocatePages, FreePages, PruneHeap],
  TimeDefs: FROM "timedefs" USING [PackedTime];

DEFINITIONS FROM
  LoadStateDefs, DiskDefs, ImageDefs, ControlDefs, SegmentDefs;

CheckPoint: PROGRAM
  IMPORTS AllocDefs, BFSDefs, CoreSwapDefs,
    DirectoryDefs, DiskDefs, DiskKDDefs, FrameDefs, ImageDefs,
    LoadStateDefs, MiscDefs, SegmentDefs, StreamDefs, StringDefs, SystemDefs
  EXPORTS ImageDefs
  SHARES ProcessDefs, DiskDefs, SegmentDefs, ControlDefs, ImageDefs =
  BEGIN

  CFA: TYPE = AltoFileDefs.CFA;
  DataSegmentHandle: TYPE = SegmentDefs.DataSegmentHandle;
  FP: TYPE = AltoFileDefs.FP;
  FileHandle: TYPE = SegmentDefs.FileHandle;
  FileSegmentHandle: TYPE = SegmentDefs.FileSegmentHandle;
  PageSize: CARDINAL = AltoDefs.PageSize;
  PageCount: TYPE = AltoDefs.PageCount;
  PageNumber: TYPE = AltoDefs.PageNumber;
  shortFileRequest: TYPE = short ImageDefs.FileRequest;
  vDA: TYPE = AltoFileDefs.vDA;
  GlobalFrameHandle: TYPE = ControlDefs.GlobalFrameHandle;
  ConfigIndex: TYPE = LoadStateDefs.ConfigIndex;
  StreamHandle: TYPE = StreamDefs.StreamHandle;
  ProcDesc: TYPE = ControlDefs.ProcDesc;

DisplayHeader: POINTER TO WORD = LOOPHOLE[420B];
```

```
SwapTrapDuringMakeCheck: PUBLIC SIGNAL = CODE;
SwapErrorDuringMakeCheck: PUBLIC SIGNAL = CODE;
SwapOutDuringMakeCheck: PUBLIC SIGNAL = CODE;
NoRoomInCheckMap: PUBLIC SIGNAL = CODE;

SwapTrapError: PROCEDURE =
  BEGIN
  dest: ControlDefs.ControlLink;
  s: ControlDefs.StateVector;
  ProcessDefs.DisableInterrupts[];
  s ← STATE;
  dest ← LOOPHOLE[REGISTER[ControlDefs.OTPreg]];
  ProcessDefs.EnableInterrupts[];
  SIGNAL SwapTrapDuringMakeCheck;
  RETURN WITH s;
  END;

SwapOutError: AllocDefs.SwappingProcedure =
  BEGIN
  SIGNAL SwapOutDuringMakeCheck;
  RETURN[TRUE];
  END;

-- File Segment Transfer Routines

LockCodeSegment: PROCEDURE [p: ProcDesc] =
  BEGIN
  frame: ControlDefs.GlobalFrameHandle = ControlDefs.GFT[p.gfi].frame;
  FrameDefs.MakeCodeResident[frame];
  FrameDefs.SwapInCode[frame];
  SegmentDefs.Unlock[frame.codesegment];
  END;

UnlockCodeSegment: PROCEDURE [p: ProcDesc] =
  BEGIN
  SegmentDefs.Unlock[ControlDefs.GFT[p.gfi].frame.codesegment];
  END;

DAofPage: PROCEDURE [file: FileHandle, page: PageNumber] RETURNS [next: vDA] =
  BEGIN
  cfa: CFA;
  buf: POINTER = SystemDefs.AllocatePages[1];
  cfa.fp ← file.fp;
  cfa.fa ← AltoFileDefs.FA[file.fp.leaderDA,0,0];
  next ← SegmentDefs.JumpToPage[@cfa,page-1,buf].next;
  SystemDefs.FreePages[buf];
  RETURN
  END;

FillInCAs: PROCEDURE [
  Image: POINTER TO ImageHeader, mapindex: CARDINAL, ca: POINTER] =
  BEGIN
  i: CARDINAL;
  map: POINTER TO ARRAY [0..0) OF normal MapItem = LOOPHOLE[@Image.map];
  addr: POINTER;
  FOR i IN [0..mapindex) DO
    addr ← SegmentDefs.AddressFromPage[map[i].page];
    THROUGH [0..map[i].count) DO
      ca↑ ← addr;
      ca ← ca + 1;
      addr ← addr + AltoDefs.PageSize;
      ENDLOOP;
    ENDLOOP;
  END;

EnumerateNeededModules: PROCEDURE [proc: PROCEDURE [ProcDesc]] =
  BEGIN
  proc[LOOPHOLE[EnumerateNeededModules]];
  proc[LOOPHOLE[BFSDefs.ActOnPages]];
  proc[LOOPHOLE[SegmentDefs.MapFileSegment]];
  proc[LOOPHOLE[SegmentDefs.CloseFile]];
  proc[LOOPHOLE[DiskKDDefs.CloseDiskKD]];
  proc[LOOPHOLE[ImageDefs.UserCleanupProc]];
  proc[LOOPHOLE[DirectoryDefs.EnumerateDirectory]];
  proc[LOOPHOLE[StreamDefs.ReadBlock]];
  proc[LOOPHOLE[StreamDefs.CreateWordStream]];
  proc[LOOPHOLE[StringDefs.EquivalentString]];
  proc[LOOPHOLE[LoadStateDefs.InputLoadState]];
  END;

InstallCheck: PROCEDURE [name: STRING] =
  BEGIN OPEN DiskDefs, AltoFileDefs;
  wdc: CARDINAL;
  diskrequest: DiskRequest;
  savealloctrap, saveswaptrap: ControlLink;
  auxtrapFrame: FrameHandle;
  saveAllocationVector: AllocationVector;
  saveXferTrap, saveXferTrapStatus: UNSPECIFIED;
  savePuntData: POINTER;
  datapages: PageCount ← 0;
  SwapOutErrorStrategy: AllocDefs.SwapStrategy ←
    AllocDefs.SwapStrategy[link:,proc:SwapOutError];
  mapindex: CARDINAL ← 0;
  maxFileSegPages: CARDINAL ← 0;
  endofdatamapindex: CARDINAL;
  HeaderSeg: DataSegmentHandle;
  Image: POINTER TO ImageHeader;
  HeaderDA: vDA;
  checkFile: FileHandle;
  saveDIW: WORD;
  savePV: ARRAY [0..15] OF UNSPECIFIED;
  saveSDC: WORD;
  saveReadyList: ProcessDefs.Queue;
  saveCurrentPSB: ProcessDefs.ProcessHandle;
  saveCurrentState: ControlDefs.SVPointer;
  time: AltoFileDefs.TIME ← MiscDefs.DAYTIME[];
  initstateseg: FileSegmentHandle ← LoadStateDefs.GetInitialLoadState[];
  stateseg: FileSegmentHandle ← LoadStateDefs.GetLoadState[];
  net: CARDINAL ← MiscDefs.GetNetworkNumber[];
  segs: DESCRIPTOR FOR ARRAY OF FileSegmentHandle;
  maxnumbersegments: CARDINAL ← 0;
  nextpage: PageNumber;
  level: CARDINAL ← 0;
  restart: BOOLEAN;

  SaveProcesses: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    saveDIW ← DIW↑;
    savePV ← CV↑;
    DIW↑ ← 2;
    WakeupsWaiting↑ ← 0;
    saveSDC ← SDC↑;
    saveReadyList ← ReadyList↑;
    saveCurrentPSB ← CurrentPSB↑;
    saveCurrentState ← CurrentState↑;
    END;
  RestoreProcesses: PROCEDURE =
    BEGIN OPEN ProcessDefs;
    ActiveWord↑ ← 77777B;
    DIW↑ ← saveDIW;
    CV↑ ← savePV;
    SDC↑ ← saveSDC;
    ReadyList↑ ← saveReadyList;
    CurrentPSB↑ ← saveCurrentPSB;
    CurrentState↑ ← saveCurrentState;
    END;
  EnterNormalMapItem: PROCEDURE [vmpage: PageNumber, pages: PageCount] =
    BEGIN
    map: POINTER TO normal MapItem = LOOPHOLE[@Image.map];
    IF pages > 127 THEN SIGNAL SwapErrorDuringMakeCheck;
    IF mapindex >= PageSize*HeaderPages-SIZE[ImagePrefix]-SIZE[normal MapItem] THEN
      SIGNAL NoRoomInCheckMap;
    (map+mapindex)↑ ← MapItem[vmpage, pages, normal[]];
    mapindex ← mapindex + SIZE[normal MapItem];
    END;
  CountDataSegments: PROCEDURE [s: DataSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    datapages ← datapages + s.pages;
    RETURN[FALSE];
    END;
  MapDataSegments: PROCEDURE [s: DataSegmentHandle] RETURNS [BOOLEAN] =
    BEGIN
    IF s # HeaderSeg THEN
      BEGIN
      EnterNormalMapItem[s.VMpage, s.pages];
      nextpage ← nextpage + s.pages;
      END;
    RETURN[FALSE];
    END;
  CountMaxSegmentsPerFile: PROCEDURE [f: FileHandle] RETURNS [BOOLEAN] =
    BEGIN
    maxnumbersegments ← MAX[maxnumbersegments, f.swapcount];
    RETURN[FALSE];
    END;
  EnterSwappedInPerFile: PROCEDURE [f: FileHandle] RETURNS [BOOLEAN] =
    BEGIN
    nsegs: CARDINAL ← 0;
    next: PageNumber ← DefaultBase;
    i: CARDINAL;
    OrganizeSegments: PROCEDURE [s: FileSegmentHandle] RETURNS [BOOLEAN] =
      BEGIN
      i, j: CARDINAL;
      IF ~s.swappedin OR s.file # f THEN RETURN[FALSE];
      FOR i IN [0..nsegs) DO
        IF segs[i].base > s.base THEN GOTO insert;
        REPEAT
          insert =>
            BEGIN
            FOR j DECREASING IN [i..nsegs) DO segs[j+1] ← segs[j]; ENDLOOP;
            segs[i] ← s;
            END;
          FINISHED => segs[nsegs] ← s;
        ENDLOOP;
      RETURN[(nsegs ← nsegs+1) = f.swapcount];
      END;
    IF f = checkFile OR f.swapcount = 0 THEN RETURN[FALSE];
    [] ← EnumerateFileSegments[OrganizeSegments];
    FOR i IN [0..nsegs) DO
      IF segs[i].base # next THEN EnterChangeMapItem[segs[i]]
      ELSE EnterNormalMapItem[segs[i].VMpage, segs[i].pages];
      next ← segs[i].base + segs[i].pages;
      ENDLOOP;
    RETURN[FALSE];
    END;
  EnterChangeMapItem: PROCEDURE [s: FileSegmentHandle] =
    BEGIN
    map: POINTER TO change MapItem = LOOPHOLE[@Image.map];
    da: DiskDefs.DA ← DiskDefs.RealDA[GetFileSegmentDA[s]];
    IF s.pages > 127 THEN SIGNAL SwapErrorDuringMakeCheck;
    IF mapindex >= PageSize*HeaderPages-SIZE[ImagePrefix]-SIZE[change MapItem] THEN
      SIGNAL NoRoomInCheckMap;
    (map+mapindex)↑ ← MapItem[s.VMpage, s.pages, change[da, s.base]];
    mapindex ← mapindex + SIZE[change MapItem];
    END;

  checkFile ← NewFile[name, Read+Write+Append, DefaultVersion];
  ProcessDefs.DisableInterrupts[];
  wdc ← REGISTER[WDCreg];
  level ← CoreSwapDefs.GetLevel[];
  CoreSwapDefs.SetLevel[-1];
  SaveProcesses[];
  ImageDefs.UserCleanupProc[Checkpoint];

  SwapIn[initstateseg];
  [] ← LoadStateDefs.InputLoadState[];  -- bring it in for first time
  [] ← SystemDefs.PruneHeap[];

  SetupAuxStorage[];
  EnumerateNeededModules[LockCodeSegment];
  HeaderDA ← DAofPage[checkFile, 1];

  -- set up private frame allocation trap
  ControlDefs.Free[ControlDefs.Alloc[0]]; -- flush large frames
  savealloctrap ← SD[SDDefs.sAllocTrap];
  SD[SDDefs.sAllocTrap] ← auxtrapFrame ← auxtrap[];
  saveAllocationVector ← AV↑;
  AV↑ ← LOOPHOLE[DataSegmentAddress[AuxSeg], POINTER TO AllocationVector]↑;
  [] ← EnumerateDataSegments[CountDataSegments];
  SetEndOfFile[checkFile, datapages+stateseg.pages*2+FirstImageDataPage-1,
    AltoDefs.BytesPerPage];
  [] ← DiskKDDefs.CloseDiskKD[];

  HeaderSeg ← NewDataSegment[DefaultBase, 1];
  Image ← DataSegmentAddress[HeaderSeg];
  MiscDefs.Zero[Image, AltoDefs.PageSize*HeaderPages];
  Image.prefix.versionident ← ImageDefs.VersionID;
--Image.prefix.options ← 0;
--Image.prefix.state.stk[0] ← Image.prefix.state.stk[1] ← 0;
  Image.prefix.state.stkptr ← 2;
  Image.prefix.state.dest ← REGISTER[Lreg];
  Image.prefix.type ← checkfile;
  Image.prefix.leaderDA ← checkFile.fp.leaderDA;
  Image.prefix.version ← BcdDefs.VersionStamp[
    time: TimeDefs.PackedTime[lowbits: time.low, highbits: time.high],
    zapped: FALSE,
    net: net,
    host: OsStaticDefs.OsStatics.SerialNumber];
  Image.prefix.creator ← ImageDefs.ImageVersion[]; -- version stamp of currently running image

  nextpage ← FirstImageDataPage;
  [] ← EnumerateDataSegments[MapDataSegments];
  IF nextpage # FirstImageDataPage+datapages THEN ERROR;
  endofdatamapindex ← mapindex;

  -- Move LoadStates
  InlineDefs.COPY[
    from: FileSegmentAddress[stateseg],
    to: FileSegmentAddress[initstateseg],
    nwords: initstateseg.pages*PageSize];
  MapFileSegment[stateseg, checkFile, datapages+FirstImageDataPage];
  EnterNormalMapItem[stateseg.VMpage, stateseg.pages];
  MapFileSegment[
    initstateseg, checkFile, datapages+FirstImageDataPage+stateseg.pages];
  EnterNormalMapItem[initstateseg.VMpage, stateseg.pages];
  Image.prefix.loadStateBase ← stateseg.base;
  Image.prefix.initialLoadStateBase ← initstateseg.base;
  Image.prefix.loadStatePages ← initstateseg.pages;

  -- now disable swapping
  savePuntData ← CoreSwapDefs.PuntInfo↑;
  saveswaptrap ← SD[SDDefs.sSwapTrap];
  SD[SDDefs.sSwapTrap] ← SwapTrapError;
  AllocDefs.AddSwapStrategy[@SwapOutErrorStrategy];
  [] ← EnumerateFiles[CountMaxSegmentsPerFile];
  segs ← DESCRIPTOR[auxalloc[maxnumbersegments], maxnumbersegments];
  [] ← EnumerateFiles[EnterSwappedInPerFile];

  SegmentDefs.CloseFile[checkFile ! SegmentDefs.FileError => RESUME];
  checkFile.write ← checkFile.append ← FALSE;

  diskrequest ← DiskRequest[
    ca: auxalloc[datapages+3],
    da: auxalloc[datapages+3],
    fixedCA: FALSE,
    fp: auxalloc[SIZE[FP]],
    firstPage: FirstImageDataPage-1,
    lastPage: FirstImageDataPage+datapages-1,
    action: WriteD,
    lastAction: WriteD,
    signalCheckError: FALSE,
    option: update[BFSDefs.GetNextDA]];

  diskrequest.fp↑ ← checkFile.fp;
  (diskrequest.ca+1)↑ ← Image;
  FillInCAs[Image, endofdatamapindex, diskrequest.ca+2];
  MiscDefs.SetBlock[diskrequest.da,fillinDA,datapages+3];
  (diskrequest.da+1)↑ ← HeaderDA;

  saveXferTrap ← SD[SDDefs.sXferTrap];
  SD[SDDefs.sXferTrap] ← REGISTER[Lreg];
  saveXferTrapStatus ← REGISTER[XTSreg];

  restart ← BFSDefs.ActOnPages[LOOPHOLE[@diskrequest]].page = 0;
  REGISTER[WDCreg] ← wdc;
  AV↑ ← saveAllocationVector;
  SD[SDDefs.sAllocTrap] ← savealloctrap;
  SD[SDDefs.sXferTrap] ← saveXferTrap;
  REGISTER[XTSreg] ← saveXferTrapStatus;
  Free[auxtrapFrame];
  DeleteDataSegment[HeaderSeg];

  -- turn swapping back on
  AllocDefs.RemoveSwapStrategy[@SwapOutErrorStrategy];
  SD[SDDefs.sSwapTrap] ← saveswaptrap;

  RestoreProcesses[];
  CoreSwapDefs.PuntInfo↑ ← savePuntData;
  IF ~restart THEN CoreSwapDefs.SetLevel[level];
  ProcessDefs.EnableInterrupts[];

  InlineDefs.COPY[
    to: FileSegmentAddress[stateseg],
    from: FileSegmentAddress[initstateseg],
    nwords: initstateseg.pages*PageSize];
  LoadStateDefs.ReleaseLoadState[];
  Unlock[initstateseg];
  SwapOut[initstateseg];
  DeleteDataSegment[AuxSeg];

  EnumerateNeededModules[UnlockCodeSegment];
  ImageDefs.UserCleanupProc[IF restart THEN Restart ELSE Continue];
  RETURN
  END;

-- auxillary storage for frames and non-saved items
AuxSeg: DataSegmentHandle;
freepointer: POINTER;
wordsleft: CARDINAL;

SetupAuxStorage: PROCEDURE =
  BEGIN
  av : POINTER;
  i: CARDINAL;
  AuxSeg ← NewDataSegment[DefaultBase,5];
  av ← freepointer ← DataSegmentAddress[AuxSeg];
  wordsleft ← 10*PageSize;
  [] ← auxalloc[AllocationVectorSize];
  freepointer ← freepointer+3; wordsleft ← wordsleft-3;
  FOR i IN [0..MaxAllocSlot) DO
    (av+i)↑ ← (i+1)*4+2;
    ENDLOOP;
  (av+6)↑ ← (av+MaxAllocSlot)↑ ← (av+MaxAllocSlot+1)↑ ← 1;
  END;

auxalloc: PROCEDURE [n: CARDINAL] RETURNS [p: POINTER] =
  BEGIN -- allocate in multiples of 4 words
  p ← freepointer;
  n ← InlineDefs.BITAND[n+3,177774B];
  freepointer ← freepointer+n;
  IF wordsleft < n THEN ImageDefs.PuntMesa[];
  wordsleft ← wordsleft-n;
  RETURN
  END;

auxtrap: PROCEDURE RETURNS [myframe: FrameHandle] =
  BEGIN
  state: StateVector;
  newframe: FrameHandle;
  eventry: POINTER TO EntryVectorItem;
  fsize, findex: CARDINAL;
  newG: GlobalFrameHandle;
  dest, tempdest: ControlLink;
  alloc: BOOLEAN;
  gfi: GFTIndex;
  ep: CARDINAL;

  myframe ← LOOPHOLE[REGISTER[Lreg]];
  state.dest ← myframe.returnlink; state.source ← 0;
  state.instbyte←0;
  state.stk[0]←myframe;
  state.stkptr←1;

  ProcessDefs.DisableInterrupts[];

  DO
    ProcessDefs.EnableInterrupts[];
    TRANSFER WITH state;

    ProcessDefs.DisableInterrupts[];
    state ← STATE;

    dest ← LOOPHOLE[REGISTER[ATPreg]];
    myframe.returnlink ← state.source;
    tempdest ← dest;
    DO
      SELECT tempdest.tag FROM
        frame =>
          BEGIN
          alloc ← TRUE;
          findex ← LOOPHOLE[tempdest, CARDINAL]/4;
          EXIT
          END;
        procedure =>
          BEGIN OPEN proc: LOOPHOLE[tempdest, ControlDefs.ProcDesc];
          gfi ← proc.gfi;  ep ← proc.ep;
          [frame: newG, epbase: findex] ← GFT[gfi];
          eventry ← @newG.code.prefix.entry[findex+ep];
          findex ← eventry.framesize;
          alloc ← FALSE;
          EXIT
          END;
        indirect => tempdest ← tempdest.link↑;
        ENDCASE => ImageDefs.PuntMesa[];
      ENDLOOP;

    IF findex >= MaxAllocSlot THEN ImageDefs.PuntMesa[]
    ELSE
      BEGIN
      fsize ← FrameVec[findex]+1;  -- includes overhead word
      newframe ← LOOPHOLE[freepointer+1];
      freepointer↑ ← findex;
      freepointer ← freepointer + fsize;
      IF wordsleft < fsize THEN ImageDefs.PuntMesa[]
      ELSE wordsleft ← wordsleft - fsize;
      END;

    IF alloc THEN
      BEGIN
      state.dest ← myframe.returnlink;
      state.stk[state.stkptr] ← newframe;
      state.stkptr ← state.stkptr+1;
      END
    ELSE
      BEGIN
      IF dest.tag # indirect THEN
        BEGIN
        state.dest ← newframe;
        newframe.accesslink ← newG;
        newframe.pc ← eventry.initialpc;
        newframe.returnlink ← myframe.returnlink;
        END
      ELSE
        BEGIN
        IF findex = MaxAllocSlot THEN ImageDefs.PuntMesa[];
        state.dest ← dest;
        newframe.accesslink ← LOOPHOLE[AV[findex].frame];
        AV[findex].frame ← newframe;
        END;
      state.source ← myframe.returnlink;
      END;

    ENDLOOP;
    END;

-- The driver
```

```
MakeCheckPoint: PUBLIC PROCEDURE [name: STRING] =
  BEGIN
  s: StateVector;
  s ← STATE;
  s.stk[0] ← REGISTER[Greg];
  s.stkptr ← 1;
  s.dest ← FrameDefs.SwapOutCode;
  s.source ← GetReturnLink[];
  InstallCheck[name];
  RETURN WITH s;
  END;

END.
```