```
-- Swapper.Mesa  Edited by Sandman on October 6, 1977  7:19 AM

DIRECTORY
  AltoDefs: FROM "altodefs",
  AltoFileDefs: FROM "altofiledefs",
  BootDefs: FROM "bootdefs",
  ControlDefs: FROM "controldefs",
  DiskDefs: FROM "diskdefs",
  FrameDefs: FROM "framedefs",
  InlineDefs: FROM "inlinedefs",
  MiscDefs: FROM "miscdefs",
  ProcessDefs: FROM "processdefs",
  SegmentDefs: FROM "segmentdefs";

DEFINITIONS FROM AltoDefs, AltoFileDefs, BootDefs, DiskDefs, SegmentDefs;

Swapper: PROGRAM [ffvmp, lfvmp: PageNumber]
  IMPORTS BootDefs, FrameDefs, SegmentDefs
  EXPORTS BootDefs, DiskDefs, FrameDefs, MiscDefs, SegmentDefs
  SHARES DiskDefs, SegmentDefs = BEGIN

  nil: POINTER = LOOPHOLE[0];
  driveNumber: PUBLIC [0..1] ← 0;
  sysdisk: DISK ← DISK[nDisks,nTracks,nHeads,nSectors];
  disk: POINTER TO DISK = @sysdisk;

  Zero: PUBLIC PROCEDURE [
    p:POINTER, l:CARDINAL] =
    BEGIN
    IF l=0 THEN RETURN;  p↑ ← 0;
    InlineDefs.COPY [
      from:p, to:p+1, nwords:l-1];
    RETURN
    END;

  SetDisk: PUBLIC PROCEDURE [
    d:POINTER TO DISK] =
    BEGIN
    disk↑ ← d↑;
    RETURN
    END;

  GetDisk: PUBLIC PROCEDURE
    RETURNS [POINTER TO DISK] =
    BEGIN
    RETURN[disk]
    END;

  ResetDisk: PUBLIC PROCEDURE
    RETURNS [POINTER TO DISK] =
    BEGIN
    disk↑ ← DISK[nDisks,nTracks,nHeads,nSectors];
    RETURN[disk]
    END;

  VirtualDA: PUBLIC PROCEDURE [da:DA] RETURNS [vDA] =
    BEGIN
    RETURN[IF da = DA[0,0,0,0,0] THEN eofDA ELSE vDA [
      ((da.disk*disk.tracks+da.track)*disk.heads+
       da.head)*disk.sectors+da.sector]];
    END;

  RealDA: PUBLIC PROCEDURE [v:vDA] RETURNS [da:DA] =
    BEGIN
    i: CARDINAL ← v;
    da ← DA[0,0,0,0,0];
    IF v # eofDA THEN
      BEGIN
      [i,da.sector] ← InlineDefs.DIVMOD[i,disk.sectors];
      [i,da.head] ← InlineDefs.DIVMOD[i,disk.heads];
      [i,da.track] ← InlineDefs.DIVMOD[i,disk.tracks];
      [i,da.disk] ← InlineDefs.DIVMOD[i,disk.disks];
      IF i # 0 THEN da ← InvalidDA;
      END;
    RETURN
    END;
```

```
  -- Disk transfer "process"

  DCseal: BYTE = 110B;

  DCs: ARRAY vDC OF DC = [
    DC[DCseal,DiskRead, DiskRead, DiskRead, 0,0],  -- ReadHLD
    DC[DCseal,DiskCheck,DiskRead, DiskRead, 0,0],  -- ReadLD
    DC[DCseal,DiskCheck,DiskCheck,DiskRead, 0,0],  -- ReadD
    DC[DCseal,DiskWrite,DiskWrite,DiskWrite,0,0],  -- WriteHLD
    DC[DCseal,DiskCheck,DiskWrite,DiskWrite,0,0],  -- WriteLD
    DC[DCseal,DiskCheck,DiskCheck,DiskWrite,0,0],  -- WriteD
    DC[DCseal,DiskCheck,DiskCheck,DiskCheck,1,0],  -- SeekOnly
    DC[DCseal,DiskCheck,DiskCheck,DiskCheck,0,0]]; -- DoNothing

  nextDiskCommand: POINTER TO CBptr = LOOPHOLE[521B];
  diskStatus: POINTER TO DS = LOOPHOLE[522B];
  lastDiskAddress: POINTER TO DA = LOOPHOLE[523B];
  sectorInterrupts: POINTER TO CARDINAL = LOOPHOLE[524B];

  -- DoDiskCommand assumes that the version number in a FID (and
  -- in an FP) will never be used (is always one).  It further
  -- assumes that if fp is nil (zero), a FreePageFID was meant;
  -- this allows the rest of the world to use short (3 word) FPs.

  FreePageFID: FID = FID[-1,SN[1,1,1,17777B,-1]];

  NonZeroWaitCell: WORD ← 1;
  waitCell: POINTER TO WORD ← @NonZeroWaitCell;

  ResetWaitCell: PUBLIC PROCEDURE =
    BEGIN
    waitCell ← @NonZeroWaitCell;
    END;

  SetWaitCell: PUBLIC PROCEDURE [p: POINTER TO WORD] RETURNS [preval: POINTER TO WORD] =
    BEGIN
    ProcessDefs.DisableInterrupts[];
    preval ← waitCell;
    waitCell ← p;
    ProcessDefs.EnableInterrupts[];
    RETURN;
    END;

  DoDiskCommand: PUBLIC PROCEDURE [arg:POINTER TO DDC] =
    BEGIN OPEN arg;
    ptr, next, prev: CBptr;
    la: POINTER TO DL;
    zone: CBZptr = cb.zone;
    cb.headerAddress ← @cb.header;
    IF (la ← cb.labelAddress) = nil THEN
      cb.labelAddress ← la ← @cb.label;
    cb.dataAddress ← ca;
    IF cb.normalWakeups = 0 THEN cb.normalWakeups ← zone.normalWakeups;
    IF cb.errorWakeups = 0 THEN cb.errorWakeups ← zone.errorWakeups;
    IF fp = nil THEN la.fileID ← FreePageFID
    ELSE IF fp # NIL THEN la.fileID ← FID[1,fp.serial];
    la.page ← cb.page ← page;
    IF da # fillinDA THEN cb.header.diskAddress ← RealDA[da];
    IF restore THEN cb.header.diskAddress.restore ← 1;
    cb.command ← DCs[action];
    cb.command.exchange ← driveNumber;
    prev ← PrevCB[zone];
    -- Put the command on the disk controller's queue
    ProcessDefs.DisableInterrupts[];
    UNTIL waitCell↑ # 0 DO NULL ENDLOOP; -- Wait for Trident to finish
    IF (next ← nextDiskCommand↑) # nil THEN
      BEGIN
      DO  ptr ← next;  next ← ptr.nextCB;
        IF next = nil THEN EXIT;
        ENDLOOP;
      ptr.nextCB ← cb;
      END;
    -- Take care of a possible race with disk controller.  The disk
    -- may have gone idle (perhaps due to an error) even as we were
    -- adding a command to the chain.  To make sure there was no
    -- error, we check the status of the previous cb in this zone.
```

```
    IF nextDiskCommand↑ = nil THEN
      SELECT MaskDS[prev.status, DSmaskStatus] FROM
        DSfreeStatus, DSgoodStatus => nextDiskCommand↑ ← cb;
        ENDCASE;
    ProcessDefs.EnableInterrupts[];
    EnqueueCB[zone,cb];
    RETURN
    END;
```

```
-- Disk command block queue

InitializeCBstorage: PUBLIC PROCEDURE [
  zone:CBZptr, nCBs:CARDINAL, page:PageNumber, init:CBinit] =
  BEGIN
  cb: CBptr;
  i: CARDINAL;
  nq: CARDINAL = nCBs+1;
  length: CARDINAL = SIZE[CBZ]+nCBs*(SIZE[CB]+SIZE[CBptr]);
  queue: DESCRIPTOR FOR ARRAY OF CBptr ←
    DESCRIPTOR[@zone.queueVec,nq];
  cbVector: DESCRIPTOR FOR ARRAY OF CB ←
    DESCRIPTOR[@zone.queueVec+SIZE[CBptr]*nq,nCBs];
  IF init = clear THEN Zero[zone,length];
  zone.currentPage ← page;  zone.cbQueue ← queue;
  zone.qTail ← 0;  zone.qHead ← 1;
  queue[0] ← NIL; -- end of queue;
  FOR i IN [1..nCBs] DO
    queue[i] ← cb ← @cbVector[i-1];
    cb.zone ← zone;  cb.status ← DSfreeStatus;
    ENDLOOP;
  RETURN
  END;

NumCBs: PROCEDURE [zone:CBZptr] RETURNS [CARDINAL] =
  BEGIN
  RETURN[LENGTH[zone.cbQueue]-1]
  END;

ClearCB: PROCEDURE [cb:CBptr] =
  BEGIN
  zone: CBZptr = cb.zone;
  Zero[cb,SIZE[CB]];
  cb.zone ← zone;
  RETURN
  END;

EnqueueCB: PROCEDURE [zone:CBZptr, cb:CBptr] =
  BEGIN  i: CARDINAL ← zone.qTail;
  IF zone.cbQueue[i] # NIL THEN ERROR;
  zone.cbQueue[i] ← cb;
  IF (i ← i+1) = LENGTH[zone.cbQueue] THEN i ← 0;
  zone.qTail ← i;
  RETURN
  END;

DequeueCB: PROCEDURE [zone:CBZptr] RETURNS [cb:CBptr] =
  BEGIN  i: CARDINAL ← zone.qHead;
  IF (cb ← zone.cbQueue[i]) = NIL THEN ERROR;
  zone.cbQueue[i] ← NIL;
  IF (i ← i+1) = LENGTH[zone.cbQueue] THEN i ← 0;
  zone.qHead ← i;
  RETURN
  END;

PrevCB: PROCEDURE [zone:CBZptr] RETURNS [cb:CBptr] =
  BEGIN  i: CARDINAL ← zone.qTail;
  i ← (IF i=0 THEN LENGTH[zone.cbQueue] ELSE i) - 1;
  IF (cb ← zone.cbQueue[i]) = NIL THEN ERROR;
  RETURN
  END;

CleanupCBqueue: PUBLIC PROCEDURE [zone:CBZptr] =
  BEGIN
  cb: CBptr;
  UNTIL zone.cbQueue[zone.qHead] = NIL DO
    cb ← GetCB[zone,dontClear];
    ENDLOOP;
  RETURN
  END;
```

```
-- Removing CBs from the queue.  If for some reason the disk has
-- gone idle without executing the command, we fake an error
-- in it so that the entire zone of CBs will get retryed.

RetryableDiskError: PUBLIC SIGNAL [cb:CBptr] = CODE;
UnrecoverableDiskError: PUBLIC SIGNAL [cb:CBptr] = CODE;

MaskDS: MACHINE CODE [DS, DS] RETURNS [DS] = LOOPHOLE[InlineDefs.BITAND];

GetCB: PUBLIC PROCEDURE [zone:CBZptr, init:CBinit] RETURNS [cb:CBptr] =
  BEGIN  s:DS;  da:vDA;  ddc:DDC;  ec:CARDINAL;
  cb ← DequeueCB[zone];
  UNTIL cb.status.done # 0 DO
    -- not zero means done or fake or free
    IF nextDiskCommand↑ = nil
    AND cb.status.done = 0 THEN
      cb.status ← DSfakeStatus;
    ENDLOOP;
  cb.command.seal ← 0; -- remove command seal
  s ← MaskDS[cb.status, DSmaskStatus];
  SELECT s FROM
    DSgoodStatus =>
      BEGIN
      IF cb.header.diskAddress.restore=0 THEN
        BEGIN
        zone.errorCount ← 0;
        zone.currentBytes ← cb.labelAddress.bytes;
        IF zone.cleanup # LOOPHOLE[0] THEN zone.cleanup[cb];
        END;
      IF init = clear THEN ClearCB[cb];
      END;
    DSfreeStatus =>
      ClearCB[cb]; -- really means DSneverBeenUsed
    ENDCASE =>
      BEGIN -- some error occurred
      -- busy wait until disk controller is idle
      UNTIL nextDiskCommand↑ = nil DO NULL ENDLOOP;
      ec ← zone.errorCount ← zone.errorCount+1;
      IF ec >= RetryCount THEN ERROR UnrecoverableDiskError[cb];
      da ← zone.errorDA ← VirtualDA[cb.header.diskAddress];
      IF cb.status.finalStatus = CheckError THEN zone.checkError ← TRUE;
      InitializeCBstorage [
        zone,NumCBs[zone],cb.page,dontClear];
      IF ec > RetryCount/2 THEN
        BEGIN -- start a restore before signalling the error
        lastDiskAddress↑ ← InvalidDA;
        ddc ← DDC[GetCB[zone,clear],nil,da,0,NIL,TRUE,SeekOnly];
        DoDiskCommand[@ddc];
        END;
      ERROR RetryableDiskError[cb];
      END;
  RETURN
  END;

-- Don't all Cleanup procedures need to be locked?
```

```
-- Segment swapper

-- Note that each CB is used twice: first to hold the disk label
-- for page i-1, and then to hold the DCB for page i.  It isn't
-- reused until the DCB for page i-1 is correctly done, which
-- is guaranteed to be after the disk label for page i-1 is no
-- longer needed, since things are done strictly sequentially by
-- page number.

-- Currently, DiskRequest.lastAction is not used by SwapPages.

DiskCheckError: PUBLIC SIGNAL [page:PageNumber] = CODE;

SwapPages: PUBLIC PROCEDURE [arg:POINTER TO swap DiskRequest]
  RETURNS [PageNumber, CARDINAL] =
  BEGIN OPEN arg;
  i: PageNumber;
  cb, nextcb: CBptr;
  cbzone: ARRAY [0..1CBZ) OF UNSPECIFIED;
  zone: CBZptr = @cbzone[0];
  ddc: DDC ← DDC[,ca,da↑,,fp,FALSE,action];
  InitializeCBstorage[zone,nCB,firstPage,clear];
  IF desc # NIL THEN
    BEGIN zone.info ← desc;
    zone.cleanup ← GetDiskPageDesc;
    END;
  BEGIN
    ENABLE RetryableDiskError --[cb]-- =>
      BEGIN
      ddc.da ← zone.errorDA;
      ddc.ca ← cb.dataAddress;
      RETRY END;
    cb ← GetCB[zone,clear];
    FOR i ← zone.currentPage, i+1 UNTIL i=lastPage+1 DO
      IF ddc.da = eofDA THEN EXIT;
      IF signalCheckError AND zone.errorCount = RetryCount/2
        THEN SIGNAL DiskCheckError[i];
      nextcb ← GetCB[zone,clear];
      cb.labelAddress ← LOOPHOLE[@nextcb.header.diskAddress];
      ddc.cb ← cb;  ddc.page ← i;
      IF i # zone.currentPage THEN ddc.da ← fillinDA;
      DoDiskCommand[@ddc];
      IF ~fixedCA THEN ddc.ca ← ddc.ca+PageSize;
      cb ← nextcb;
      ENDLOOP;
    CleanupCBqueue[zone];
    END; -- of enable block
  RETURN[i-1,zone.currentBytes]
  END;

  GetDiskPageDesc: PROCEDURE [cb:CBptr] =
    BEGIN
    la: POINTER TO DL = cb.labelAddress;
    desc: POINTER TO DiskPageDesc ← cb.zone.info;
    desc↑ ← DiskPageDesc [
      VirtualDA[la.prev],
      VirtualDA[cb.header.diskAddress],
      VirtualDA[la.next],
      la.page, la.bytes];
    RETURN
    END;
```

```
-- Routines for peering into machine addresses

PAGEDISP: TYPE = MACHINE DEPENDENT RECORD [
  page: [0..MaxVMPage],
  disp: [0..PageSize)];

PageFromAddress: PUBLIC PROCEDURE [a:POINTER] RETURNS [PageNumber] =
  BEGIN
  RETURN[LOOPHOLE[a,PAGEDISP].page]
  END;

AddressFromPage: PUBLIC PROCEDURE [p:PageNumber] RETURNS [POINTER] =
  BEGIN
  RETURN[LOOPHOLE[PAGEDISP[p,0]]]
  END;

PagePointer: PUBLIC PROCEDURE [a:POINTER] RETURNS [POINTER] =
  BEGIN
  LOOPHOLE[a,PAGEDISP].disp ← 0;
  RETURN[a]
  END;


-- Data Segments

VMNotFree: PUBLIC SIGNAL [base:PageNumber, pages:PageCount] = CODE;

NewDataSegment: PUBLIC PROCEDURE [base:PageNumber, pages:PageCount]
  RETURNS [seg:DataSegmentHandle] = BEGIN
  IF pages ~IN (0..MaxVMPage+1]
    THEN ERROR InvalidSegmentSize[pages];
  seg ← AllocateDataSegment[DataSegmentTable];
    BEGIN ENABLE UNWIND => LiberateDataSegment[DataSegmentTable,seg];
    IF base=DefaultBase THEN
      base ← AllocVM[pages,top,TrySwapping]
    ELSE BEGIN
      DO  -- until pages free
        ProcessDefs.DisableInterrupts[];
        IF PagesFree[base,pages] THEN EXIT;
        ProcessDefs.EnableInterrupts[];
        SIGNAL VMNotFree[base,pages];
        ENDLOOP;
      UpdateVM[base,pages,busy];
      ProcessDefs.EnableInterrupts[];
      END;
    END;
  seg.VMpage ← base;
  seg.pages ← pages;
  RETURN
  END;

BootDataSegment: PUBLIC PROCEDURE [base:PageNumber, pages:PageCount]
  RETURNS [seg:DataSegmentHandle] = BEGIN
  i: PageNumber;
  FOR i IN [base..base+pages) DO
    IF PageFree[i] THEN ERROR;
    ENDLOOP;
  seg ← AllocateDataSegment[DataSegmentTable];
  seg.VMpage ← base;
  seg.pages ← pages;
  RETURN
  END;

DeleteDataSegment: PUBLIC PROCEDURE [seg:DataSegmentHandle] =
  BEGIN
  base: PageNumber; pages: PageCount;
  ValidateDataSegment[DataSegmentTable,seg];
  base ← seg.VMpage; pages ← seg.pages;
  LiberateDataSegment[DataSegmentTable,seg];
  UpdateVM[base,pages,free];
  RETURN
  END;

DataSegmentAddress: PUBLIC PROCEDURE [seg:DataSegmentHandle] RETURNS [POINTER] =
  BEGIN
  RETURN[LOOPHOLE[PAGEDISP[seg.VMpage,0]]]
```

```
    END;
```

```
-- Swapping Segments

SwapError: PUBLIC SIGNAL [seg:FileSegmentHandle] = CODE;

SwapIn: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  IF lock = MaxLocks THEN SIGNAL SwapError[seg];
  ProcessDefs.DisableInterrupts[];
  IF ~swappedin THEN
    BEGIN
    -- make it busy here.
    ProcessDefs.EnableInterrupts[];
    IF file.swapcount = MaxRefs THEN SIGNAL SwapError[seg];
    IF ~file.open THEN OpenFile[file];
    VMpage ← AllocVM[pages,bottom,TrySwapping];
    BEGIN ENABLE UNWIND => UpdateVM[VMpage,pages,free];
      IF (hint.page # base OR hint.da = eofDA)
      AND PositionSeg[seg,TRUE] AND pages = 1
        THEN NULL ELSE MapVM[seg,ReadD];
      END;
    file.swapcount ← file.swapcount+1;
    ProcessDefs.DisableInterrupts[];
    -- make it unbusy here.
    swappedin ← TRUE;
    END;
  lock ← lock+1;
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

Unlock: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  IF lock = 0 THEN
    ERROR SwapError[seg];
  ProcessDefs.DisableInterrupts[];
  lock ← lock-1;
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

SwapUp: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  IF swappedin AND write THEN
    BEGIN
    IF ~file.open THEN OpenFile[file];
    IF hint.page # base
    OR hint.da = eofDA THEN
      [] ← PositionSeg[seg,FALSE];
    MapVM[seg,WriteD];
    END;
  RETURN
  END;

SwapOut: PUBLIC PROCEDURE [seg:FileSegmentHandle] =
  BEGIN OPEN seg;
  temp: PageNumber;
  IF ~swappedin THEN RETURN;
  IF lock # 0 THEN ERROR SwapError[seg];
  SwapUp[seg];
  ProcessDefs.DisableInterrupts[];
  swappedin ← FALSE;
  temp ← VMpage;  VMpage ← 0;
  file.swapcount ← file.swapcount-1;
  ProcessDefs.EnableInterrupts[];
  -- IF swapcount = 0 THEN CloseFile[]; ???
  UpdateVM[temp,pages,free];
  RETURN
  END;

SegmentFault: PUBLIC SIGNAL [seg:FileSegmentHandle, pages:PageCount] = CODE;

MapVM: PUBLIC PROCEDURE [seg:FileSegmentHandle, dc:vDC] =
  BEGIN OPEN seg;
  page: PageNumber;  byte: CARDINAL;  temp: PageCount;
  arg: swap DiskRequest ← DiskRequest [
    AddressFromPage[VMpage], @hint.da,
```

```
    base, base+pages-1, @file.fp,
    FALSE, dc, dc, FALSE, swap[NIL]];
  IF hint.page # base THEN ERROR SwapError[seg];
  [page,byte] ← SwapPages[@arg];
  temp ← page-base+(IF byte=0 THEN 0 ELSE 1);
  IF temp=0 THEN ERROR SegmentFault[seg,0];
  IF temp # pages THEN
    BEGIN SIGNAL SegmentFault[seg,temp];
    UpdateVM[VMpage+temp,pages-temp,free];
    pages ← temp;
    END;
  RETURN
  END;
```

```
-- Code Swapping and Swap Strategies

TrySwapInProgress: BOOLEAN ← FALSE;

TrySwapping: PROCEDURE RETURNS [did: BOOLEAN] =
  BEGIN  sp, next: POINTER TO SwapStrategy;
  ProcessDefs.DisableInterrupts[];
  IF TrySwapInProgress THEN
    BEGIN
    ProcessDefs.EnableInterrupts[];
    RETURN[TryCodeSwapping[]];
    END;
  TrySwapInProgress ← TRUE;
  ProcessDefs.EnableInterrupts[];
  did ← TRUE;
  FOR sp ← StrategyList, next UNTIL sp = NIL DO
    next ← sp.link;
    IF sp.proc[] THEN EXIT;
    REPEAT FINISHED => did ← FALSE;
    ENDLOOP;
  TrySwapInProgress ← FALSE;
  RETURN
  END;

CantSwap: PUBLIC SwappingProcedure =
  BEGIN
  RETURN[FALSE]
  END;

gftrover: ControlDefs.GFTIndex ← 1;

TryCodeSwapping: PUBLIC SwappingProcedure =
  BEGIN OPEN ControlDefs;
  g: POINTER TO GFTItem;
  passtwo: BOOLEAN ← FALSE;
  start: GFTIndex ← gftrover;
  sd: POINTER TO ARRAY [0..0) OF CARDINAL = REGISTER[SDreg];
  gft: DESCRIPTOR FOR ARRAY OF GFTItem =
    DESCRIPTOR[REGISTER[GFTreg],sd[sGFTLength]];
  DO -- until we've looked at them all twice
    ProcessDefs.DisableInterrupts[];
    IF (gftrover ← gftrover+1) = sd[sGFTLength] THEN gftrover ← 1;
    IF gftrover = start THEN
      IF passtwo THEN EXIT
      ELSE passtwo ← TRUE;
    g ← @gft[gftrover];
    IF g.frame # NULLFrame AND g.epbase = 0
    AND g.frame.codesegment.swappedin
    AND g.frame.codesegment.lock = 0 THEN
      IF g.frame.codebase↑ # 0 THEN g.frame.codebase↑ ← 0
      ELSE
        BEGIN
        ProcessDefs.EnableInterrupts[];
        SwapOutCode[g.frame];
        RETURN[TRUE]
        END;
    ProcessDefs.EnableInterrupts[];
    ENDLOOP;
  ProcessDefs.EnableInterrupts[];
  RETURN[FALSE]
  END;

SwapOutCode: PUBLIC PROCEDURE [f:ControlDefs.GlobalFrameHandle] =
  BEGIN OPEN ControlDefs;  cseg: FileSegmentHandle;
  g: GlobalFrameHandle ← f.accesslink;
  dupfound: BOOLEAN;
  FindDup: PROCEDURE [testf:GlobalFrameHandle]
    RETURNS [BOOLEAN] = BEGIN
    RETURN[cseg = testf.codesegment AND testf # g]
    END;
  MarkFrame: PROCEDURE [f:GlobalFrameHandle]
    RETURNS [BOOLEAN] = BEGIN
    IF cseg = f.codesegment THEN
      f.codebase ← LOOPHOLE[1];
    RETURN[FALSE]
    END;
```

```
    IF g.accesslink # g THEN ERROR SwapError[NIL];
    cseg ← g.codesegment;
    ProcessDefs.DisableInterrupts[];
    IF cseg.swappedin THEN
      BEGIN
      SwapIn[cseg]; -- lock it so it won't go away
      ProcessDefs.EnableInterrupts[];
      dupfound ← FrameDefs.EnumerateGlobalFrames[FindDup] # NULLFrame;
      ProcessDefs.DisableInterrupts[];
      IF dupfound THEN [] ← FrameDefs.EnumerateGlobalFrames[MarkFrame]
      ELSE g.codebase ← LOOPHOLE[1];
      Unlock[cseg]; SwapOut[cseg];
      END;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  LastResort: SwapStrategy ← SwapStrategy[NIL,TryCodeSwapping];
  StrategyList: POINTER TO SwapStrategy ← @LastResort;

  AddSwapStrategy: PUBLIC PROCEDURE [strategy:POINTER TO SwapStrategy] =
    BEGIN
    sp: POINTER TO SwapStrategy;
    ProcessDefs.DisableInterrupts[];
    FOR sp ← StrategyList, sp.link
    UNTIL sp = NIL DO
      IF sp = strategy THEN RETURN;
      ENDLOOP;
    strategy.link ← StrategyList;
    StrategyList ← strategy;
    ProcessDefs.EnableInterrupts[];
    RETURN
    END;

  RemoveSwapStrategy: PUBLIC PROCEDURE [strategy:POINTER TO SwapStrategy] =
    BEGIN
    sp: POINTER TO SwapStrategy;
    prev: POINTER TO SwapStrategy ← NIL;
    ProcessDefs.DisableInterrupts[];
    FOR sp ← StrategyList, sp.link UNTIL sp = NIL DO
      IF sp = strategy THEN
        BEGIN
        IF prev = NIL
          THEN StrategyList ← sp.link
          ELSE prev.link ← sp.link;
        EXIT END;
      prev ← sp;
      ENDLOOP;
    ProcessDefs.EnableInterrupts[];
    strategy.link ← NIL;
    RETURN
    END;
```

```
-- Memory Allocator

PageState: TYPE = {free,busy};

PageMap: ARRAY [0..(MaxVMPage+1)/wordlength) OF WORD;

GetPageMap: PUBLIC PROCEDURE RETURNS [POINTER] =
  BEGIN RETURN[@PageMap[0]] END;

MapAddress: TYPE = MACHINE DEPENDENT RECORD [
  fill: [0..377B],
  word: [0..17B],
  bit: [0..17B]];

-- Warning: the bits in PageMap are numbered backwards (bit zero on the right)!

PageFree: PUBLIC PROCEDURE [page:PageNumber] RETURNS [BOOLEAN] =
  BEGIN OPEN InlineDefs;
  ma: MapAddress = LOOPHOLE[page];
  RETURN[ma.fill=0 AND
     BITAND[PageMap[ma.word],BITSHIFT[1,ma.bit]]=0]
  END;

PagesFree: PUBLIC PROCEDURE [base:PageNumber, pages:PageCount] RETURNS [BOOLEAN] =
  BEGIN
  FOR base IN [base..base+pages) DO
    IF ~PageFree[base] THEN RETURN[FALSE];
    ENDLOOP;
  RETURN[TRUE]
  END;

SetPageState: PROCEDURE [page:PageNumber, state:PageState] =
  BEGIN OPEN InlineDefs;
  ptr: POINTER TO WORD = @PageMap[LOOPHOLE[page, MapAddress].word];
  IF LOOPHOLE[page, MapAddress].fill # 0 THEN ERROR;
  ptr↑ ← IF state = free
    THEN BITAND[ptr↑,BITNOT[BITSHIFT[1,LOOPHOLE[page  MapAddress].bit]]]
    ELSE BITOR[ptr↑,BITSHIFT[1,LOOPHOLE[page, MapAddress].bit]];
  RETURN
  END;

AllocType: TYPE = {bottom,top};

InsufficientVM: PUBLIC SIGNAL [needed:PageCount] = CODE;

AllocVM: PROCEDURE [
  pages:PageCount, type:AllocType,
  swap:PROCEDURE RETURNS [BOOLEAN]]
  RETURNS [PageNumber] =
  BEGIN  n: CARDINAL;  base: PageNumber;
  direction: INTEGER = IF type = bottom THEN 1 ELSE -1;
  DO -- repeat if insufficient VM
    ProcessDefs.DisableInterrupts[];
    IF type = bottom THEN
      BEGIN
      -- eliminate prefix of allocated pages
      FOR ffvmp INCREASING IN [ffvmp..lfvmp] DO
        IF PageFree[ffvmp] THEN EXIT;
        ENDLOOP;
      base ← ffvmp;
      END
    ELSE
      BEGIN
      -- eliminate suffix of allocated pages
      FOR lfvmp DECREASING IN [ffvmp..lfvmp] DO
        IF PageFree[lfvmp] THEN EXIT;
        ENDLOOP;
      base ← lfvmp;
      END;
    n ← 0;  -- count of contiguous free pages
    UNTIL (IF direction>0 THEN base>lfvmp ELSE base<ffvmp) DO
      IF ~PageFree[base] THEN n ← 0
      ELSE IF (n ← n+1) = pages THEN
        BEGIN
        IF direction>0 THEN base ← base-n+1;
        FOR n IN [base..base+pages) DO
```

```
            SetPageState[n,busy];
            ENDLOOP;
          ProcessDefs.EnableInterrupts[];
          RETURN[base]
          END;
        base ← base+direction
        ENDLOOP;
      ProcessDefs.EnableInterrupts[];
      IF ~swap[] THEN SIGNAL InsufficientVM[pages];
      ENDLOOP
    END;

  UpdateVM: PROCEDURE [base:PageNumber, pages:PageCount, state:PageState] =
    BEGIN
    IF state = free THEN
      BEGIN
      ProcessDefs.DisableInterrupts[];
      ffvmp ← MIN[ffvmp,base];
      lfvmp ← MAX[lfvmp,base+pages-1];
      ProcessDefs.EnableInterrupts[];
      END;
    FOR base IN [base..base+pages) DO
      SetPageState[base,state];
      ENDLOOP;
    RETURN
    END;
```

```
-- Primative Object Allocation

ObjectSeal: BYTE = 321B;

InvalidObject: PUBLIC SIGNAL [table:TableHandle, object:POINTER] = CODE;

AllocateObject: PUBLIC PROCEDURE [table:TableHandle]
  RETURNS [ObjectHandle] = BEGIN
  frob: FrobHandle;
  subtable: SubTableHandle;
  ProcessDefs.DisableInterrupts[];
  FOR subtable ← table.link, subtable.link
  UNTIL subtable = NIL DO
    IF subtable.free # NIL THEN EXIT;
    REPEAT FINISHED =>
      subtable ← AllocateSubTable[table
        ! UNWIND => ProcessDefs.EnableInterrupts[]];
    ENDLOOP;
  frob ← subtable.free;
  frob.inuse ← TRUE;
  subtable.free ← frob.link;
  subtable.alloc ← subtable.alloc+1;
  ProcessDefs.EnableInterrupts[];
  RETURN[LOOPHOLE[frob]]
  END;

LiberateObject: PUBLIC PROCEDURE [table:TableHandle, object:ObjectHandle] =
  BEGIN
  subtable: SubTableHandle = PagePointer[object];
  frob: FrobHandle = LOOPHOLE[object];
  ProcessDefs.DisableInterrupts[];
  frob↑ ← Object[FALSE,FALSE,Free[0,]];
  frob.link ← subtable.free;
  subtable.free ← frob;
  subtable.alloc ← subtable.alloc-1;
  IF subtable.alloc = 0 THEN
    LiberateSubTable[table,subtable];
  ProcessDefs.EnableInterrupts[];
  RETURN
  END;

AllocateSubTable: PROCEDURE [table:TableHandle]
  RETURNS [subtable:SubTableHandle] = BEGIN
  frob, prev: FrobHandle;
  n: CARDINAL = (PageSize-SIZE[SubTable])/table.size;
  subtable ← AddressFromPage[AllocVM[1,top,TryCodeSwapping]];
  subtable↑ ← SubTable[,,ObjectSeal,0,];
  frob ← subtable.free ←
    LOOPHOLE[subtable+SIZE[SubTable]];
  THROUGH [0..n) DO
    prev ← frob;  frob ← frob+table.size;
    prev↑ ← Object[FALSE,FALSE,Free[0,]];
    prev.link ← frob;
    ENDLOOP;
  prev.link ← NIL;
  subtable.link ← table.link;
  table.link ← subtable;
  subtable.seg ← BootDataSegment [
    PageFromAddress[subtable],1];
  RETURN
  END;

LiberateSubTable: PROCEDURE [table:TableHandle, subtable:SubTableHandle] =
  BEGIN
  current: SubTableHandle;
  prev: SubTableHandle ← NIL;
  FOR current ← table.link, current.link
  UNTIL current = NIL DO
    IF current = subtable THEN
      BEGIN
      IF prev = NIL
        THEN table.link ← current.link
        ELSE prev.link ← current.link;
      -- oops: this had better not recur!
      DeleteDataSegment[current.seg];
      RETURN
```

```
        END;
      prev ← current;
      ENDLOOP;
    ERROR InvalidObject[table,subtable];
    END;

  ValidateObject: PUBLIC PROCEDURE [table:TableHandle, object:ObjectHandle] =
    BEGIN
    i, j: CARDINAL;
    subtable: SubTableHandle = PagePointer[object];
    [i,j] ← InlineDefs.DIVMOD [
      object-LOOPHOLE[subtable+SIZE[SubTable],ObjectHandle], table.size];
    IF i ~IN [0..(PageSize-SIZE[SubTable])/table.size)
    OR j # 0 OR ~object.inuse OR subtable.seal # ObjectSeal
      THEN ERROR InvalidObject[table,object];
    RETURN
    END;

  EnumerateObjects: PUBLIC PROCEDURE [table:TableHandle,
    proc:PROCEDURE [ObjectHandle] RETURNS [BOOLEAN]]
    RETURNS [object:ObjectHandle] = BEGIN
    subtable: SubTableHandle;
    n: CARDINAL = (PageSize-SIZE[SubTable])/table.size;
    FOR subtable ← table.link, subtable.link
    UNTIL subtable = NIL DO
      IF subtable.alloc # 0 THEN
        BEGIN
        object ← LOOPHOLE[subtable+SIZE[SubTable]];
        THROUGH [0..n) DO
          IF object.inuse AND proc[object] THEN RETURN;
          object ← object+table.size;
          ENDLOOP;
        END;
      ENDLOOP;
    RETURN[NIL]
    END;
```

```
  -- Managing Data Segment Objects

  DataSegmentObjects: Table ← Table[SIZE[DataSegmentObject],NIL];
  DataSegmentTable: PUBLIC TableHandle ← @DataSegmentObjects;

  GetDataSegmentTable: PUBLIC PROCEDURE RETURNS [TableHandle] =
    BEGIN RETURN[DataSegmentTable] END;

  AllocateDataSegment: PROCEDURE [TableHandle] RETURNS [DataSegmentHandle];
  ValidateDataSegment: PROCEDURE [TableHandle, DataSegmentHandle];
  LiberateDataSegment: PROCEDURE [TableHandle, DataSegmentHandle];

  EnumerateDataSegments: PUBLIC PROCEDURE [
    proc:PROCEDURE [DataSegmentHandle] RETURNS [BOOLEAN]]
    RETURNS [DataSegmentHandle] =
    BEGIN RETURN[LOOPHOLE[
      EnumerateObjects[DataSegmentTable,LOOPHOLE[proc]]]]
    END;

  VMtoDataSegment: PUBLIC PROCEDURE [a:POINTER] RETURNS [DataSegmentHandle] =
    BEGIN
    pg: PageNumber ← PageFromAddress[a];
    MatchPage: PROCEDURE [seg:DataSegmentHandle] RETURNS [BOOLEAN] =
      BEGIN
      RETURN[pg IN [seg.VMpage..seg.VMpage+seg.pages)]
      END;
    RETURN[EnumerateDataSegments[MatchPage]]
    END;


  -- Main body

  i: [0..MaxVMPage];

  FOR i IN [0..MaxVMPage] DO
    SetPageState[i,
      IF i IN [ffvmp..lfvmp] THEN free ELSE busy];
    ENDLOOP;

  AllocateDataSegment ← LOOPHOLE[AllocateObject];
  ValidateDataSegment ← LOOPHOLE[ValidateObject];
  LiberateDataSegment ← LOOPHOLE[LiberateObject];

  END.
```