```
-- Fsp.Mesa  Edited by Sandman on May 12, 1978  9:18 AM

DIRECTORY
  AltoDefs: FROM "altodefs" USING [PageSize],
  FSPDefs: FROM "fspdefs" USING [
    BlockSize, Deallocator, FreeNodePointer, NodeHeader, NodePointer,
    ZoneHeader, ZoneOverhead, ZonePointer],
  ProcessDefs: FROM "processdefs" USING [InitializeMonitor],
  StringDefs: FROM "stringdefs" USING [WordsForString],
  SystemDefs: FROM "systemdefs" USING [
    AllocateResidentPages, AllocateResidentSegment, FreePages, PagesForWords];

DEFINITIONS FROM FSPDefs;

FSP: MONITOR LOCKS z.lock USING z: ZonePointer
  IMPORTS ProcessDefs, StringDefs, SystemDefs EXPORTS FSPDefs, SystemDefs
  SHARES FSPDefs = PUBLIC
  BEGIN   -- Mesa Free Storage Package --

    -- A set of procedures to manage allocation within a zone.
    -- Coalescing of free nodes occurs during allocation; all
    -- free nodes following a candidate node are merged before
    -- any space is allocated.  The logic is derived from a
    -- BCPL program by E. M. McCreight and was suggested by an
    -- exercise in Knuth Volume I, p. 453 #19

  UsedNodeSize: PRIVATE BlockSize = SIZE [inuse NodeHeader];
  FreeNodeSize: PRIVATE BlockSize = SIZE [free NodeHeader];
  ZoneHeaderSize: PRIVATE BlockSize = SIZE [ZoneHeader];

  ZoneTooSmall: ERROR [POINTER] = CODE;

  DoNothingDeallocate: Deallocator = BEGIN NULL END;

  MakeZone: PROCEDURE [base: POINTER, length: BlockSize] RETURNS [z: ZonePointer] =
    BEGIN
    RETURN[MakeNewZone[base, length, DoNothingDeallocate]];
    END;

  MakeNewZone: PROCEDURE [base: POINTER, length: BlockSize, deallocate: Deallocator]
  RETURNS [z: ZonePointer] =
    BEGIN
    fn: FreeNodePointer;
    an: NodePointer;
    IF length < ZoneHeaderSize+FreeNodeSize+UsedNodeSize
      THEN ERROR ZoneTooSmall[base];
    z ← base;
    -- set up the bulk of the zone as a large free block.
    fn ← base + ZoneHeaderSize;
    fn↑ ← NodeHeader[length: length-(ZoneHeaderSize+UsedNodeSize),
      extension: free[fwdp: @z.node, backp: @z.node]];
    -- set up allocated node (smallest possible) at end of block.
    an ← base + (length-UsedNodeSize);
    an↑ ← NodeHeader[length: UsedNodeSize, extension: inuse[]];
    -- set up the zone header
    z↑ ← ZoneHeader[rover: fn, lock:, restOfZone: NIL, length: length,
      deallocate: deallocate, threshold: FreeNodeSize, checking: FALSE,
      node: NodeHeader[length: 0, extension: free[fwdp: fn, backp: fn]]];
    ProcessDefs.InitializeMonitor[@z.lock];
    RETURN
    END;

  AddToZone: PROCEDURE [z: ZonePointer, base: POINTER, length: BlockSize] =
    BEGIN
    AddToNewZone[z, base, length, DoNothingDeallocate];
    END;

  AddToNewZone: ENTRY PROCEDURE [z: ZonePointer, base: POINTER,
    length: BlockSize, deallocate: Deallocator] =
    BEGIN
    newz: ZonePointer;
    firstnew, lastnew: FreeNodePointer;
    ValidateZone[z ! UNWIND => NULL];
    newz ← MakeNewZone[base, length, deallocate];
    -- splice the zones together
    firstnew ← newz.node.fwdp;  lastnew ← newz.node.backp;
```

```
    z.node.backp.fwdp ← firstnew;
    firstnew.backp ← z.node.backp;
    lastnew.fwdp ← @z.node;
    z.node.backp ← lastnew;
    -- make newz head an empty list
    newz.node.fwdp ← newz.node.backp ← @newz.node;
    newz.restOfZone ← z.restOfZone;
    z.restOfZone ← newz;
    IF z.checking THEN CheckZone[z ! UNWIND => NULL];
    RETURN
    END;

  InvalidZone: ERROR [POINTER] = CODE;  -- zone header looks fishy

  ValidateZone: PRIVATE PROCEDURE [z: ZonePointer] =
    BEGIN
    SELECT TRUE FROM
      (z.node.length # 0),
      ((@z.node+z.length-UsedNodeSize).length # UsedNodeSize),
      (z.node.fwdp.backp # @z.node OR z.node.backp.fwdp # @z.node) =>
    ERROR InvalidZone[z];
      ENDCASE;
    RETURN
    END;

  NodeLoop: ERROR [ZonePointer] = CODE;

  CheckZone: PRIVATE PROCEDURE [z: ZonePointer] =
    BEGIN
    node: FreeNodePointer;
    count: INTEGER;
    ValidateZone[z];
    count ← (LAST[BlockSize]-FIRST[BlockSize])/FreeNodeSize + 1;
    node ← @z.node;
      DO
      CheckNode[z, node];
      IF (count ← count-1) < 0 THEN ERROR NodeLoop[z];
      IF (node ← node.fwdp) = @z.node THEN EXIT;
      ENDLOOP;
    RETURN
    END;

  InvalidNode: ERROR [POINTER] = CODE;  -- node appears damaged

  CheckNode: PRIVATE PROCEDURE [z: ZonePointer, node: NodePointer] =
    BEGIN
      DO
      WITH node SELECT FROM
        inuse =>
          IF length = UsedNodeSize THEN EXIT;    -- end of zone
        free =>
          BEGIN
          IF fwdp.backp # node OR backp.fwdp # node
            THEN GO TO error;
          IF length=0 AND node # @z.node THEN GO TO error;
          END;
        ENDCASE;
      node ← node + node.length;
      IF node.state # inuse THEN EXIT;
      REPEAT
        error =>
          BEGIN  z.checking ← FALSE;
          ERROR InvalidNode[node+UsedNodeSize];
          END;
      ENDLOOP;
    RETURN
    END;

  NoRoomInZone: SIGNAL [ZonePointer] = CODE;

  MakeNode: PROCEDURE [z: ZonePointer, n: BlockSize] RETURNS [POINTER] =
    BEGIN
    node: NodePointer;
    n ← MAX[n+UsedNodeSize, FreeNodeSize];
    WHILE (node ← GetNode[z, n]) = NIL DO
      SIGNAL NoRoomInZone[z];
```

```
    ENDLOOP;  -- try again if RESUMEd from the signal
  RETURN[node]
  END;

GetNode: PRIVATE ENTRY PROCEDURE [z: ZonePointer, n: BlockSize] RETURNS [POINTER] =
  BEGIN
  rover: FreeNodePointer ← z.rover;
  node, neighbour: NodePointer;
  nodelength, nl: BlockSize;
  IF z.checking THEN CheckZone[z ! UNWIND => NULL];
  DO
    nodelength ← rover.length;
    FOR neighbour ← rover+nodelength, neighbour+nl DO
      WITH neighbour SELECT FROM
        inuse => EXIT;
        free =>
          BEGIN        -- coalesce
          IF (nl ← length) = 0 THEN EXIT;        -- end of zone
          fwdp.backp ← backp;  backp.fwdp ← fwdp;
          z.rover ← rover;        -- in case neighbor was z.rover
          nodelength ← nodelength+nl;
          END;
        ENDCASE;
      ENDLOOP;
    IF nodelength >= n THEN
      BEGIN
      IF (nl ← (nodelength-n)) > MAX[FreeNodeSize, z.threshold] THEN
        BEGIN        -- split the block
        z.rover ← rover;  rover.length ← nl;
        node ← rover+nl;  nodelength ← n;
        END
      ELSE
        BEGIN
        rover.fwdp.backp ← rover.backp;
        z.rover ← rover.backp.fwdp ← rover.fwdp;
        node ← rover;
        END;
      node↑ ← NodeHeader[nodelength, inuse[]];
      RETURN [node + UsedNodeSize]
      END
    ELSE rover.length ← nodelength;
    IF (rover ← rover.fwdp) = z.rover THEN EXIT;
    ENDLOOP;
  RETURN[NIL];
  END;

FreeNode: ENTRY PROCEDURE [z: ZonePointer, p: POINTER] =
  BEGIN
  IF z.checking THEN CheckZone[z ! UNWIND => NULL];
  FreeThisNode[z, p ! UNWIND => NULL];
  RETURN
  END;

FreeThisNode: PRIVATE PROCEDURE [z: ZonePointer, p: POINTER] =
  BEGIN
  node: NodePointer = p-UsedNodeSize;
  WITH node SELECT FROM
    free => ERROR InvalidNode[p];
    ENDCASE =>
      node↑ ← NodeHeader[node.length, free[@z.node, z.node.backp]];
  WITH n:node SELECT FROM
    free => z.node.backp ← n.backp.fwdp ← @n;
    ENDCASE;
  RETURN
  END;

SplitNode: ENTRY PROCEDURE [z: ZonePointer, p: POINTER, n: BlockSize] =
  BEGIN
  node: NodePointer = p-UsedNodeSize;
  lastpart: NodePointer;
  t: INTEGER;
  IF z.checking THEN CheckZone[z ! UNWIND => NULL];
  n ← n+UsedNodeSize;
  WITH node SELECT FROM
    free => RETURN WITH ERROR InvalidNode[p];
    ENDCASE =>
```

```
      IF (t ← node.length - n) >= MAX[FreeNodeSize,  z.threshold] THEN
    BEGIN
    lastpart ← node+n;
    lastpart↑ ← NodeHeader[t, inuse[]];
    FreeThisNode[z, lastpart+UsedNodeSize];
    END;
    RETURN
    END;

  NodeSize: PROCEDURE [p: POINTER] RETURNS [BlockSize] =
    BEGIN
    node: NodePointer = p-UsedNodeSize;
    WITH node SELECT FROM
      free => ERROR InvalidNode[p];
      ENDCASE => RETURN [length-UsedNodeSize];
    END;

  PruneZone: ENTRY PROCEDURE [z: ZonePointer] RETURNS [BOOLEAN] =
    BEGIN
    didit: BOOLEAN ← FALSE;
    rest: ZonePointer;
    zone: ZonePointer ← z;
    prev: ZonePointer ← z;
    node: NodePointer;
    nl: BlockSize;
    FOR zone ← z.restOfZone, rest UNTIL zone = NIL DO
      rest ← zone.restOfZone;
      IF zone.deallocate # DoNothingDeallocate THEN
        FOR node ← LOOPHOLE[zone+ZoneHeaderSize, NodePointer], node+nl DO
          WITH node SELECT FROM
            inuse =>
              BEGIN
              IF length = UsedNodeSize THEN
                BEGIN  -- end of zone
                FreeZone[zone ! UNWIND => NULL];
                didit ← TRUE;
                prev.restOfZone ← rest;
                END
              ELSE prev ← zone;
              EXIT
              END;
            free => nl ← length;
            ENDCASE;
          ENDLOOP;
      ENDLOOP;
    z.rover ← z.node.fwdp; -- reset rover incase in a freed zone
    RETURN[didit];
    END;

  FreeZone: PRIVATE PROCEDURE [zone: ZonePointer] =
    BEGIN
    node: NodePointer;
    nl: BlockSize;
    FOR node ← LOOPHOLE[zone+ZoneHeaderSize, NodePointer], node+nl DO
      WITH node SELECT FROM
        inuse => EXIT;  -- end of zone
        free =>
          BEGIN
          nl ← length;
          backp.fwdp ← fwdp;
          fwdp.backp ← backp;
          END;
        ENDCASE;
      ENDLOOP;
    zone.deallocate[zone];
    END;


  -- DestroyZone is not an entry procedure since the monitorlock will be
  --   gone when we try to exit the procedure

  DestroyZone: PROCEDURE [z: ZonePointer] =
    BEGIN
    rest: ZonePointer;
    IF z = TheHeap THEN RETURN;
    FOR z ← z, rest UNTIL z = NIL DO
```

```
      rest ← z.restOfZone;
      z.deallocate[z];
      ENDLOOP;
    END;

  -- management of the heap

  TheHeap: PRIVATE ZonePointer;

  HeapZone: PROCEDURE RETURNS [ZonePointer] =
    BEGIN
    RETURN [TheHeap]
    END;

  AllocateHeapNode: PROCEDURE [nwords: CARDINAL] RETURNS [p: POINTER] =
    BEGIN OPEN SystemDefs;
    np: CARDINAL;
    p ← MakeNode[TheHeap, nwords ! NoRoomInZone =>
    BEGIN
    np ← PagesForWords[nwords + ZoneOverhead + UsedNodeSize];
    nwords ← np*AltoDefs.PageSize;
    AddToNewZone[
      TheHeap, AllocateResidentSegment[nwords], nwords, FreePages];
    RESUME
    END];
    RETURN
    END;

  FreeHeapNode: PROCEDURE [p: POINTER] =
    BEGIN
    FreeNode[TheHeap, p];  RETURN
    END;


  AllocateHeapString: PROCEDURE [nchars: CARDINAL] RETURNS [STRING] =
    BEGIN
    OPEN StringDefs;
    p: POINTER TO MACHINE DEPENDENT RECORD[      -- faked string header
      length: CARDINAL,
      maxlength: CARDINAL];
    p ← AllocateHeapNode[WordsForString[nchars]];
    p.length ← 0;  p.maxlength ← nchars;
    RETURN [LOOPHOLE[p, STRING]]
    END;

  FreeHeapString: PROCEDURE [s: STRING] =
   LOOPHOLE[FreeHeapNode];

  PruneHeap: PROCEDURE RETURNS [BOOLEAN] =
    BEGIN
    RETURN[PruneZone[TheHeap]]
    END;

  -- initialization code

    TheHeap ← MakeNewZone[SystemDefs.AllocateResidentPages[1],
      1*AltoDefs.PageSize, SystemDefs.FreePages];

END.
```