```
-- file SymbolCache.mesa
-- last edited by Sandman, May 12, 1978  4:46 PM

DIRECTORY
  AllocDefs: FROM "allocdefs" USING [
    AddSwapStrategy, CantSwap, RemoveSwapStrategy, SwappingProcedure,
    SwapStrategy],
  AltoDefs: FROM "altodefs" USING [PageSize],
  SegmentDefs: FROM "segmentdefs" USING [
    FileSegmentAddress, FileSegmentHandle, InsufficientVM, InvalidFP, SwapIn,
    SwapOut, Unlock],
  SymbolTable: FROM "symboltable" USING [
    bb, cacheInfo, ctxb, extb, fgTable, hashVec, ht, link, ltb, mdb, notifier,
    NullNotifier, seb, sourceFile, ssb, stHandle, tb],
  SymbolTableDefs: FROM "symboltabledefs" USING [
    SymbolTableBase, SymbolTableHandle],
  SymDefs: FROM "symdefs" USING [fgHeader, HTIndex, HTRecord, STHeader],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode],
  TableDefs: FROM "tabledefs" USING [TableBase];

DEFINITIONS FROM SymbolTableDefs;

SymbolCache: PROGRAM
    IMPORTS AllocDefs, initial: SymbolTable, SegmentDefs, SystemDefs
    EXPORTS SymbolTableDefs SHARES SymbolTableDefs =
  BEGIN
  OPEN SegmentDefs;

  -- public interface

  NoSymbolTable: PUBLIC SIGNAL [FileSegmentHandle] = CODE;

  TableForSegment: PUBLIC PROCEDURE [seg: FileSegmentHandle] RETURNS [SymbolTableHandle] =
    BEGIN
    IF seg = NIL THEN ERROR NoSymbolTable[seg];
    RETURN [SymbolTableHandle[seg]]
    END;

  SegmentForTable: PUBLIC PROCEDURE [table: SymbolTableHandle] RETURNS [FileSegmentHandle] =
    BEGIN
    RETURN [table.segment]
    END;


  IllegalSymbolBase: PUBLIC SIGNAL [base: SymbolTableBase] = CODE;

  AcquireSymbolTable: PUBLIC PROCEDURE [handle: SymbolTableHandle]
    RETURNS [base: SymbolTableBase] =
    BEGIN
    stlink: CachePointer;
    IF freeTables = NIL THEN
      BEGIN
      base ← NEW initial;
      START base
      END
    ELSE
      BEGIN base ← freeTables; freeTables ← freeTables.link END;
    stlink ← MakeCacheEntry[handle];
    InstallTable[base, stlink];
    base.link ← inuseTables;
    inuseTables ← base;
    RETURN
    END;

  ReleaseSymbolTable: PUBLIC PROCEDURE [base: SymbolTableBase] =
    BEGIN
    stlink: CachePointer ← header;
    tableCache: CachePointer = base.cacheInfo;
    prev, table: SymbolTableBase ← NIL;
    DO
      IF stlink = tableCache THEN EXIT;
      stlink ← stlink.next;
      IF stlink = header THEN ERROR IllegalSymbolBase[base];
      ENDLOOP;
    FOR table ← inuseTables, table.link UNTIL table = NIL DO
      IF table = base THEN
```

```
        BEGIN
        IF prev # NIL THEN prev.link ← table.link
        ELSE inuseTables ← table.link;
        EXIT
        END;
      prev ← table;
      REPEAT
        FINISHED => ERROR IllegalSymbolBase[base];
      ENDLOOP;
    FreeCacheEntry[tableCache];
    base.link ← freeTables;
    freeTables ← base;
    RETURN
    END;

  inuseTables: SymbolTableBase ← NIL;
  freeTables: SymbolTableBase ← initial;

  cachepagelimit: CARDINAL ← 0;

  SymbolCacheSize: PUBLIC PROCEDURE RETURNS [pages: CARDINAL] =
    BEGIN
    RETURN[cachepagelimit]
    END;

  SetSymbolCacheSize: PUBLIC PROCEDURE [pages: CARDINAL] =
    BEGIN
    cachepagelimit ← pages;
    trimcache[cachepagelimit];
    RETURN
    END;

  suspended: BOOLEAN;

  SuspendSymbolCache: PUBLIC PROCEDURE =
    BEGIN
    node: CachePointer;
    trimcache[0];
    suspended ← TRUE;
    FOR node ← header.next, node.next UNTIL node = free
      DO  Unlock[node.table];  SwapOut[node.table]  ENDLOOP;
    RETURN
    END;

  RestartSymbolCache: PUBLIC PROCEDURE =
    BEGIN
    node, tableInfo: CachePointer;
    base: SymbolTableBase;
    IF ~suspended THEN ERROR;
    FOR node ← header.next, node.next UNTIL node = free
      DO
      SwapIn[node.table];
      node.symheader ← FileSegmentAddress[node.table];
      ENDLOOP;
    FOR base ← inuseTables, base.link UNTIL base = NIL DO
      tableInfo ← base.cacheInfo;
      SetBases[base, tableInfo];
      base.notifier[base];
      ENDLOOP;
    suspended ← FALSE;
    RETURN
    END;


-- internal cache management

  CacheNode: TYPE = RECORD[
    prev, next: CachePointer,
    table: SymbolTableHandle,
    symheader: POINTER,
    refcount: CARDINAL];

  CachePointer: TYPE = POINTER TO CacheNode;

  header, free, flushed: CachePointer;
  CacheNodes: CARDINAL = 7;
```

```
  cachedpages: CARDINAL;

  IncompatibleSymbolTable: ERROR = CODE;


  --    C A C H E   O R G A N I Z A T I O N

  -- The cache keeps track of segments in CacheNodes.  The CacheNodes are
  -- kept in a doubly-linked list and organized in three groups, separated
  -- by three pointers: header, free, and flushed.  Assuming a clockwise
  -- ordering, the list is organized as follows:
  --   header -> last empty node,
  --   free -> first node with segment but no frame,
  --   flushed -> first empty node,
  --   flushed = free => no nodes with segment but no frame,
  --   header.next = free => no nodes with frame,
  --   header = flushed => no available empty nodes

  -- Initial conditions are header.next = free = flushed


  MakeCacheEntry: PROCEDURE [handle: SymbolTableHandle]
    RETURNS [node: CachePointer] =
    BEGIN
    FOR node ← header.next, node.next UNTIL node = free DO
      IF node.table = handle THEN GO TO allocated;
      REPEAT
        allocated =>  NULL;
        FINISHED =>
          BEGIN
          FOR node ← free, node.next UNTIL node = flushed DO
            IF node.table = handle THEN GO TO unflushed;
            REPEAT
              unflushed =>
                BEGIN
                movenode[node, free];
                IF flushed = free THEN
                  AllocDefs.RemoveSwapStrategy[@flushstrategy];
                END;
              FINISHED =>
                BEGIN
                node ← GetEmptyNode[];
                SwapIn[handle !
                  InvalidFP => ERROR NoSymbolTable[handle];
                  InsufficientVM => IF free # flushed THEN
                    BEGIN  FlushATable[];  RESUME  END];
                cachedpages ← cachedpages + handle.pages;
                node.table ← handle;
                node.symheader ← FileSegmentAddress[handle];
                movenode[node, free];
                END;
            ENDLOOP;
          movenode[node, free];
          END;
      ENDLOOP;
    node.refcount ← node.refcount+1;  RETURN
    END;

  FreeCacheEntry: PROCEDURE [node: CachePointer] =
    BEGIN
    np: CARDINAL;
    slot: CachePointer;
    SELECT (node.refcount ← node.refcount-1) FROM
      =0 =>
        BEGIN
        slot ← free;  np ← node.table.pages;
        IF 3*np > cachepagelimit THEN
          UNTIL slot = flushed OR slot.table.pages > np DO
            slot ← slot.next;
            ENDLOOP;
        IF flushed = free THEN AllocDefs.AddSwapStrategy[@flushstrategy];
        movenode[node, slot];
        IF slot = free THEN free ← node;
        trimcache[cachepagelimit];
        END;
```

```
      >0 => NULL;
      ENDCASE => ERROR;
    RETURN
    END;

  GetEmptyNode: PROCEDURE RETURNS [node: CachePointer] =
    BEGIN
    IF flushed # header THEN
      BEGIN node ← header; header ← header.prev; END
    ELSE
      BEGIN
      node ← SystemDefs.AllocateHeapNode[SIZE[CacheNode]];
      node↑ ← CacheNode[NIL, NIL, [NIL], NIL, 0];
      node.prev ← header;
      header.next.prev ← node;
      node.next ← header.next;
      header.next ← node;
      END;
    RETURN
    END;

  FlushATable: PROCEDURE =
    BEGIN
    IF free = flushed THEN RETURN;
    Unlock[flushed.prev.table]; SwapOut[flushed.prev.table];
    cachedpages ← cachedpages - flushed.prev.table.pages;
    flushed ← flushed.prev;
    IF flushed = free THEN AllocDefs.RemoveSwapStrategy[@flushstrategy];
    RETURN
    END;

  GetFlushedNode: PROCEDURE RETURNS [CachePointer] =
    BEGIN
    UNTIL flushed # header DO
      IF free # flushed THEN FlushATable[] ELSE ERROR;
      ENDLOOP;
    RETURN [flushed]
    END;

  movenode: PROCEDURE [node, position: CachePointer] =
    BEGIN
    IF node = free THEN free ← free.next;
    IF node = flushed THEN flushed ← flushed.next;
    IF node # position AND node.next # position THEN
      BEGIN
      node.prev.next ← node.next; node.next.prev ← node.prev;
      node.prev ← position.prev; node.prev.next ← node;
      node.next ← position; position.prev ← node;
      END;
    RETURN
    END;

  trimcache: PROCEDURE [size: CARDINAL] =
    BEGIN
    WHILE cachedpages > size AND free # flushed DO FlushATable[] ENDLOOP;
    RETURN
    END;

  flushstrategy: AllocDefs.SwapStrategy ← [link:, proc: flushtables];

  flushtables: AllocDefs.SwappingProcedure =
    BEGIN
    changed: BOOLEAN ← (free # flushed);
    flushstrategy.proc ← AllocDefs.CantSwap;
    trimcache[IF needed >= cachedpages THEN 0 ELSE cachedpages - needed];
    flushstrategy.proc ← flushtables;
    RETURN [changed]
    END;


-- symbol table setup

  InstallTable: PROCEDURE [base: SymbolTableBase, node: CachePointer] =
    BEGIN
    SetBases[base, node]; base.notifier ← base.NullNotifier; RETURN
    END;
```

```
  SetBases: PROCEDURE [base: SymbolTableBase, node: CachePointer] =
    BEGIN
    b: POINTER = node.symheader;
    tB: TableDefs.TableBase = LOOPHOLE[b];
    p: POINTER TO SymDefs.STHeader = node.symheader;
    q: POINTER TO SymDefs.fgHeader;
    base.cacheInfo ← node;
    base.hashVec ←
    DESCRIPTOR[b+p.hvBlock.offset, p.hvBlock.size/SIZE[SymDefs.HTIndex]];
    base.ht ←
    DESCRIPTOR[b+p.htBlock.offset, p.htBlock.size/SIZE[SymDefs.HTRecord]];
    base.ssb ← b + p.ssBlock.offset;
    base.seb ← tB + p.seBlock.offset;
    base.ctxb ← tB + p.ctxBlock.offset;
    base.mdb ← tB + p.mdBlock.offset;
    base.bb ← tB + p.bodyBlock.offset;
    base.tb ← tB + p.treeBlock.offset;
    base.ltb ← tB + p.litBlock.offset;
    base.extb ← tB + p.extBlock.offset;
    base.stHandle ← p;
    IF p.fgRelPgBase = 0 THEN  base.sourceFile ← NIL
    ELSE
      BEGIN
      q ← b + p.fgRelPgBase*AltoDefs.PageSize;
      base.sourceFile ← LOOPHOLE[@q.sourcefile];
      base.fgTable ← DESCRIPTOR[
        b + p.fgRelPgBase*AltoDefs.PageSize + q.fgoffset, q.fglength];
      END;
    RETURN
    END;

-- initialization

  CacheEntries: ARRAY [0..CacheNodes] OF CacheNode;

  Init: PROCEDURE =
    BEGIN
    j: INTEGER [0..CacheNodes];
    START initial;
    FOR j IN [0..CacheNodes] DO
      CacheEntries[j].refcount ← 0;
      CacheEntries[j].next ← @CacheEntries[IF j=CacheNodes THEN 0 ELSE j+1];
      CacheEntries[j].prev ← @CacheEntries[IF j=0 THEN CacheNodes ELSE j-1];
      ENDLOOP;
    header ← @CacheEntries[0];
    initial.link ← NIL;
    free ← flushed ← header.next;  cachedpages ← 0;
    suspended ← FALSE;
    END;

  Init[];

  END..
```