```
-- file SymbolCache.Mesa
-- last modified by Satterthwaite on August 26, 1977  9:23 PM

DIRECTORY
  AltoDefs: FROM "altodefs",
  ControlDefs: FROM "controldefs",
  SegmentDefs: FROM "segmentdefs",
  SymDefs: FROM "symdefs",
  SymbolTable: FROM "symboltable",
  TableDefs: FROM "tabledefs",
  SymbolTableDefs: FROM "symboltabledefs";

DEFINITIONS FROM SymbolTableDefs;

SymbolCache: PROGRAM
    IMPORTS
      s0: SymbolTable, s1: SymbolTable, s2: SymbolTable, s3: SymbolTable,
      SegmentDefs
    EXPORTS SymbolTableDefs
    SHARES ControlDefs, SymbolTableDefs =
  BEGIN
  OPEN SegmentDefs;

  -- public interface

  NoSymbolTable: PUBLIC SIGNAL [FileSegmentHandle] = CODE;

  TableForFrame: PUBLIC PROCEDURE [frame: ControlDefs.FrameHandle] RETURNS [SymbolTableHandle] =
    BEGIN
    symbolseg: FileSegmentHandle = frame.accesslink.symbolsegment;
    IF symbolseg = NIL THEN ERROR NoSymbolTable[frame.accesslink.codesegment];
    IF symbolseg.class # symbols THEN ERROR;
    RETURN [SymbolTableHandle[symbolseg]]
    END;

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

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


  TooManySymbolTables: PUBLIC SIGNAL [handle: SymbolTableHandle] = CODE;
  SymbolBuffersFull: PUBLIC SIGNAL = CODE;
  IllegalSymbolBase: PUBLIC SIGNAL [base: SymbolTableBase] = CODE;

  AcquireSymbolTable: PUBLIC PROCEDURE [handle: SymbolTableHandle] RETURNS [base: SymbolTableBase] =
    BEGIN
    i: STMapIndex;
    -- repeat on failure
      DO
      i ← strover;
        DO
        IF stmap[i].stlink = NIL
          THEN
            BEGIN  strover ← i;
            stmap[i].stlink ← MakeCacheEntry[handle];
            base ← stmap[i].stbase;
            InstallTable[base, stmap[i].stlink.symheader];
            RETURN
            END;
        i ← IF i=LAST[STMapIndex] THEN FIRST[STMapIndex] ELSE i+1;
        IF i = strover THEN EXIT;
        ENDLOOP;
      SIGNAL TooManySymbolTables[handle];
      ENDLOOP;
    END;

  ReleaseSymbolTable: PUBLIC PROCEDURE [base: SymbolTableBase] =
    BEGIN
    i: STMapIndex ← strover;
```

```
      DO
      IF stmap[i].stbase = base AND stmap[i].stlink # NIL
        THEN
          BEGIN  strover ← i;
          FreeCacheEntry[stmap[i].stlink];  stmap[i].stlink ← NIL;
          RETURN
          END;
      i ← IF i=FIRST[STMapIndex] THEN LAST[STMapIndex] ELSE i-1;
      IF i = strover THEN EXIT;
      ENDLOOP;
    SIGNAL IllegalSymbolBase[base];  RETURN
    END;


  cachepagelimit: INTEGER ← 0;

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

  SetSymbolCacheSize: PUBLIC PROCEDURE [pages: INTEGER] =
    BEGIN
    cachepagelimit ← MAX[pages, 0];
    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: CachePointer;
    i: STMapIndex;
    IF ~suspended THEN ERROR;
    FOR node ← header.next, node.next UNTIL node = free
      DO
      SwapIn[node.table];
      node.symheader ← FileSegmentAddress[node.table];
      ENDLOOP;
    FOR i IN STMapIndex
      DO
      IF stmap[i].stlink # NIL
        THEN
          BEGIN
          SetBases[stmap[i].stbase, stmap[i].stlink.symheader];
          stmap[i].stbase.notifier[stmap[i].stbase];
          END;
      ENDLOOP;
    suspended ← FALSE;
    RETURN
    END;


  -- internal cache management

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

  CachePointer: TYPE = POINTER TO CacheNode;

  STMapIndex: TYPE = INTEGER [0..4);
```

```
  stmap: ARRAY STMapIndex OF RECORD[
    stbase: SymbolTableBase,
    stlink: CachePointer];
  strover: STMapIndex;

  header, free, flushed: CachePointer;
  CacheNodes: INTEGER = 7;

  cachedpages: INTEGER;

  IncompatibleSymbolTable: ERROR = CODE;


  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 RemoveSwapStrategy[@flushstrategy];
                END;
              FINISHED =>
                BEGIN
                trimcache[cachepagelimit];
                node ← GetFlushedNode[];
                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: INTEGER;
    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 AddSwapStrategy[@flushstrategy];
        movenode[node, slot];
        IF slot = free THEN free ← node;
        trimcache[cachepagelimit];
        END;
      >0 => NULL;
      ENDCASE => ERROR;
    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 RemoveSwapStrategy[@flushstrategy];
    RETURN
    END;

  GetFlushedNode: PROCEDURE RETURNS [CachePointer] =
    BEGIN
    UNTIL flushed # header
      DO
      IF free # flushed THEN FlushATable[] ELSE SIGNAL SymbolBuffersFull;
      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: INTEGER] =
    BEGIN
    WHILE cachedpages > size AND free # flushed DO FlushATable[] ENDLOOP;
    RETURN
    END;

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

  flushtables: SwappingProcedure =
    BEGIN
    changed: BOOLEAN ← (free # flushed);
    trimcache[0];
    RETURN [changed]
    END;


  -- symbol table setup

  InstallTable: PROCEDURE [base: SymbolTableBase, b: POINTER] =
    BEGIN
    SetBases[base, b];  base.ignorecases ← FALSE;
    base.notifier ← base.NullNotifier;
    RETURN
    END;

  SetBases: PROCEDURE [base: SymbolTableBase, b: POINTER] =
    BEGIN
    OPEN base;
    tb: TableDefs.TableBase = LOOPHOLE[b];
    p: POINTER TO SymDefs.STHeader = b;
    q: POINTER TO SymDefs.fgHeader;
    hashvec ← DESCRIPTOR[b+p.hvOffset, p.hvSize/SIZE[SymDefs.HTIndex]];
    ht ← DESCRIPTOR[b+p.htOffset, p.htSize/SIZE[SymDefs.HTRecord]];
    ssb ← b + p.ssOffset;
    seb ← tb + p.seOffset;
    ctxb ← tb + p.ctxOffset;
    mdb ← tb + p.mdOffset;
    bb ← tb + p.bodyOffset;
    stHandle ← p;
    IF p.fgRelPgBase = 0
      THEN  sourcefile ← NIL
      ELSE
        BEGIN
        q ← b + p.fgRelPgBase*AltoDefs.PageSize;
```

```
        sourcefile ← LOOPHOLE[@q.sourcefile];
        fgt ← DESCRIPTOR[
                b + p.fgRelPgBase*AltoDefs.PageSize + q.fgoffset,
                q.fglength];
        END;
    RETURN
    END;

  -- initialization

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

    -- initialization code (must be STARTed)
      i: STMapIndex;
      j: INTEGER [0..CacheNodes];
      -- initialize the symbol map
        FOR i IN STMapIndex DO stmap[i].stlink ← NIL ENDLOOP;
        stmap[0].stbase ← s0;  RESTART s0;
        stmap[1].stbase ← s1;  RESTART s1;
        stmap[2].stbase ← s2;  RESTART s2;
        stmap[3].stbase ← s3;  RESTART s3;
        strover ← FIRST[STMapIndex];
      -- initialize the cache bookkeeping
        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];
        free ← flushed ← header.next;  cachedpages ← 0;
        suspended ← FALSE;

    END.
```