```
-- file Allocator.Mesa
-- last modified by Sandman, May 11, 1978  4:22 PM

DIRECTORY
  InlineDefs: FROM "inlinedefs" USING [COPY, LongDiv, LongMult],
  SystemDefs: FROM "systemdefs" USING [AllocateHeapNode, FreeHeapNode],
  TableDefs: FROM "tabledefs" USING [
    chunktype, OrderedTableIndex, Region, TableBase, TableIndex, TableLimit,
     TableNotifier, TableSelector];

Allocator: PROGRAM IMPORTS SystemDefs EXPORTS TableDefs =
  BEGIN OPEN TableDefs;

  tableRegion: Region;
  nTables: CARDINAL;

  base: DESCRIPTOR FOR ARRAY --TableSelector-- OF TableBase;
  limit: DESCRIPTOR FOR ARRAY --TableSelector-- OF [0..TableLimit];
  sharedSpace: DESCRIPTOR FOR ARRAY --TableSelector-- OF CARDINAL;
  top, oldTop: DESCRIPTOR FOR ARRAY --TableSelector-- OF CARDINAL;

  tableOpen: BOOLEAN ← FALSE;

  TableOverflow: PUBLIC SIGNAL RETURNS [Region] = CODE;
  TableFailure: PUBLIC ERROR [TableSelector] = CODE;


  -- stack allocation from subzones

  Allocate: PUBLIC PROCEDURE [table: TableSelector, size: CARDINAL]
      RETURNS [OrderedTableIndex] =
    BEGIN
    index: CARDINAL = top[table];
    newTop: CARDINAL = index + size;
    IF newTop <= limit[table]
      THEN  top[table] ← newTop
      ELSE
        IF newTop <= TableLimit
          THEN  BEGIN  top[table] ← newTop;  Repack[]  END
          ELSE  ERROR TableFailure[table];
    RETURN [LOOPHOLE[index, OrderedTableIndex]]
    END;

  TableBounds: PUBLIC PROCEDURE [table: TableSelector] RETURNS [TableBase, CARDINAL] =
    BEGIN
    RETURN [base[table], top[table]]
    END;

  TrimTable: PUBLIC PROCEDURE [table: TableSelector, size: CARDINAL] =
    BEGIN
    IF size > top[table] THEN ERROR TableFailure[table];
    top[table] ← size;
    RETURN
    END;


  Repack: PROCEDURE =
    BEGIN
        -- Garwick's Repacking algorithm (Knuth, Vol. 1, p. 245)
        -- note that d, newBase, oldTop are overlaid (on sharedSpace)
    i: CARDINAL;
    j, k, m: CARDINAL;
    sum, inc, delta, remainder: INTEGER;
    d: DESCRIPTOR FOR ARRAY --TableSelector-- OF INTEGER;
    newBase: DESCRIPTOR FOR ARRAY --TableSelector-- OF TableBase;
    sb, db: POINTER;
    newRegion: Region;
    sum ← tableRegion.size;  inc ← 0;
    d ← DESCRIPTOR[BASE[sharedSpace]+1, nTables];
    FOR j DECREASING IN [0 .. nTables)
      DO
      sum ← sum - top[j];
      inc ← inc + (d[j] ← IF top[j]>oldTop[j] THEN top[j]-oldTop[j] ELSE 0);
      ENDLOOP;
    UNTIL sum >= MIN[tableRegion.size/32, 100B]
      DO
```

```
    newRegion ← SetDescriptors[SIGNAL TableOverflow[]];
    d ← DESCRIPTOR[BASE[sharedSpace]+1, nTables];
    FOR j IN [0 .. nTables)
      DO base[j] ← newRegion.origin + (base[j]-tableRegion.origin) ENDLOOP;
    sum ← sum + (newRegion.size-tableRegion.size);
    tableRegion ← newRegion;
    ENDLOOP;
  delta ← LOOPHOLE[sum, CARDINAL]/(10*nTables);
  remainder ← sum - delta*nTables;
  newBase ← LOOPHOLE[sharedSpace];
  newBase[0] ← base[0];
  FOR j IN (0 .. nTables)
    DO
    newBase[j] ← newBase[j-1] + top[j-1] + delta +
     InlineDefs.LongDiv[
      num: InlineDefs.LongMult[d[j-1], remainder],
      den: inc];
    ENDLOOP;
  j ← 1;
  WHILE j < nTables
    DO
    SELECT newBase[j] FROM
      < base[j] =>
        BEGIN
        InlineDefs.COPY[
          from: LOOPHOLE[base[j]],
          to: LOOPHOLE[newBase[j]],
          nwords: MIN[top[j], limit[j]]];
        base[j] ← newBase[j];
        j ← j+1;
        END;
      > base[j] =>
        BEGIN
        k ← j+1;
        UNTIL k = nTables OR newBase[k] <= base[k] DO  k ← k+1  ENDLOOP;
        FOR m DECREASING IN [j .. k)
          DO
          sb ← LOOPHOLE[base[m]];  db ← LOOPHOLE[newBase[m]];
          FOR i DECREASING IN [0 .. MIN[top[m], limit[m]])
            DO  (db+i)↑ ← (sb+i)↑  ENDLOOP;
          base[m] ← newBase[m];
          ENDLOOP;
        j ← k;
        END;
      ENDCASE =>  j ← j+1;
    ENDLOOP;
  FOR j IN [0 .. nTables)
    DO
    oldTop[j] ← top[j];
    delta ← (IF j = nTables-1
      THEN tableRegion.origin + tableRegion.size
      ELSE base[j+1]) - base[j];
    limit[j] ← MIN[delta, TableLimit];
    ENDLOOP;
  UpdateBases[];  RETURN
  END;


-- linked list allocation (first subzone)

Chunk: TYPE = MACHINE DEPENDENT RECORD [
  free: BOOLEAN,              f1: [0..1],     -- fill fields
  size: [0..TableLimit),      f2: [0..3],
  fLink: CIndex,              f3: [0..3],
  bLink: CIndex];

CIndex: TYPE = POINTER [0..TableLimit) TO Chunk;

NullChunkIndex: CIndex = FIRST[CIndex];

chunkRover: CIndex;

GetChunk: PUBLIC PROCEDURE [size: CARDINAL] RETURNS [TableIndex] =
  BEGIN
  cb: TableBase = base[chunktype];
```

```
    p, q, next: CIndex;
    nodeSize: CARDINAL;
    n: INTEGER;
    size ← MAX[size, SIZE[Chunk]];
      BEGIN
      IF (p ← chunkRover) = NullChunkIndex THEN GO TO notFound;
      -- search for a chunk to allocate
        DO
        nodeSize ← (cb+p).size;
        WHILE (next←p+nodeSize) # LOOPHOLE[top[chunktype], CIndex] AND (cb+next).free
          DO
          (cb+(cb+next).bLink).fLink ← (cb+next).fLink;
          (cb+(cb+next).fLink).bLink ← (cb+next).bLink;
          (cb+p).size ← nodeSize ← nodeSize + (cb+next).size;
          chunkRover ← p;       -- in case next = chunkRover
          ENDLOOP;
        SELECT n ← nodeSize-size FROM
          = 0 =>
            BEGIN
            IF (cb+p).fLink = p
              THEN chunkRover ← NullChunkIndex
              ELSE
                BEGIN
                chunkRover ← (cb+(cb+p).bLink).fLink ← (cb+p).fLink;
                (cb+(cb+p).fLink).bLink ← (cb+p).bLink;
                END;
            q ← p;  GO TO found
            END;
          >= SIZE[Chunk] =>
            BEGIN
            (cb+p).size ← n;  chunkRover ← p;  q ← p + n;  GO TO found
            END;
          ENDCASE;
        IF (p ← (cb+p).fLink) = chunkRover THEN GO TO notFound;
        ENDLOOP;
      EXITS
        found =>  NULL;
        notFound =>  q ← Allocate[chunktype, size];
      END;
    (base[chunktype]+q).free ← FALSE;  RETURN [q]
    END;

  FreeChunk: PUBLIC PROCEDURE [i: TableIndex, size: CARDINAL] =
    BEGIN
    cb: TableBase = base[chunktype];
    p: CIndex = LOOPHOLE[i];
    (cb+p).size ← MAX[size, SIZE[Chunk]];
    IF chunkRover = NullChunkIndex
      THEN  chunkRover ← (cb+p).fLink ← (cb+p).bLink ← p
      ELSE
        BEGIN
        (cb+p).fLink ← (cb+chunkRover).fLink;
        (cb+(cb+p).fLink).bLink ← p;
        (cb+p).bLink ← chunkRover;
        (cb+chunkRover).fLink ← p;
        END;
    (cb+p).free ← TRUE;  RETURN
    END;


  -- communication

  NotifyNode: TYPE = RECORD [
    notifier: TableNotifier,
    link: POINTER TO NotifyNode];

  notifyList: POINTER TO NotifyNode;

  AddNotify: PUBLIC PROCEDURE [proc: TableNotifier] =
    BEGIN
    p: POINTER TO NotifyNode = SystemDefs.AllocateHeapNode[SIZE[NotifyNode]];
    p↑ ← [notifier:proc, link:notifyList];
    notifyList ← p;
    proc[base];  RETURN
    END;
```

```
DropNotify: PUBLIC PROCEDURE [proc: TableNotifier] =
  BEGIN
  p, q: POINTER TO NotifyNode;
  IF notifyList = NIL THEN RETURN;
  p ← notifyList;
  IF p.notifier = proc
    THEN notifyList ← p.link
    ELSE
      BEGIN
        DO
        q ← p;  p ← p.link;
        IF p = NIL THEN RETURN;
        IF p.notifier = proc THEN EXIT
        ENDLOOP;
      q.link ← p.link;
      END;
  SystemDefs.FreeHeapNode[p];  RETURN
  END;

UpdateBases: PROCEDURE =
  BEGIN
  p: POINTER TO NotifyNode;
  FOR p ← notifyList, p.link UNTIL p = NIL DO  p.notifier[base]  ENDLOOP;
  RETURN
  END;


-- initialization, expansion and termination

InitializeTable: PUBLIC PROCEDURE [region: Region, divisions: CARDINAL] =
  BEGIN
  origin, d: CARDINAL;
  i: TableSelector;
  IF tableOpen THEN EraseTable[];
  nTables ← divisions;  tableRegion ← SetDescriptors[region];
  origin ← tableRegion.origin;  d ← tableRegion.size/nTables;
  FOR i IN [0 .. nTables)
    DO
    base[i] ← origin;
    limit[i] ← IF i = 0 THEN d + (tableRegion.size - d*nTables) ELSE d;
    origin ← origin + limit[i];  top[i] ← oldTop[i] ← 0;
    ENDLOOP;
  chunkRover ← NullChunkIndex;
  notifyList ← NIL;
  tableOpen ← TRUE;   RETURN
  END;

SetDescriptors: PROCEDURE [region: Region] RETURNS [update: Region] =
  BEGIN
  arraySize: CARDINAL = nTables*SIZE[CARDINAL];
  prefixSize: CARDINAL = 4*arraySize + SIZE[CARDINAL];
  IF prefixSize > region.size THEN ERROR TableFailure[0];
  base ← DESCRIPTOR [region.origin, nTables];
  limit ← DESCRIPTOR[BASE[base] + arraySize, nTables];
  top ← DESCRIPTOR[BASE[limit] + arraySize, nTables];
  oldTop ← sharedSpace ← DESCRIPTOR[BASE[top] + arraySize, nTables];
  RETURN [[origin: region.origin+prefixSize, size: region.size-prefixSize]]
  END;

EraseTable: PUBLIC PROCEDURE =
  BEGIN
  p, q: POINTER TO NotifyNode;
  FOR p ← notifyList, q UNTIL p = NIL
    DO  q ← p.link;  SystemDefs.FreeHeapNode[p]  ENDLOOP;
  tableOpen ← FALSE;
  RETURN
  END;

END.
```