```
-- BcdAllocator.Mesa  Edited by Sandman on August 23, 1977  10:36 PM

DIRECTORY
  InlineDefs: FROM "inlinedefs",
  SystemDefs: FROM "systemdefs",
  BcdTableDefs: FROM "bcdtabledefs";

DEFINITIONS FROM BcdTableDefs;

BcdAllocator: PROGRAM
  IMPORTS SystemDefs
  EXPORTS BcdTableDefs
  SHARES BcdTableDefs =
  BEGIN

  tbase: ARRAY TableSelector OF TableBase;
  limit: ARRAY TableSelector OF [0..TableLimit];
  top, oldTop: ARRAY TableSelector OF CARDINAL;

  tableOpen: BOOLEAN ← FALSE;
  tableOrigin: CARDINAL;
  tableLimit: [0..TableLimit];

  TableOverflow: PUBLIC SIGNAL RETURNS [origin, limit: CARDINAL] = CODE;
  TableFailure: PUBLIC ERROR [TableSelector] = CODE;
  StackAllocateError: PUBLIC SIGNAL [TableSelector] = CODE;


  -- stack allocation from subzones

  Allocate: PUBLIC PROCEDURE [table: TableSelector, size: CARDINAL] RETURNS [TableIndex] =
    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, TableIndex]]
    END;

  ResetTable: PUBLIC PROCEDURE [table: TableSelector] =
    BEGIN
    top[table] ← oldTop[table] ← 0;
    RETURN
    END;

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


  Repack: PROCEDURE =
    BEGIN  -- Garwick's Repacking algorithm (Knuth, Vol. 1, p. 245)
    i: CARDINAL;
    j, k, m: [FIRST[TableSelector]..LAST[TableSelector]+1];
    nTables: CARDINAL = (LAST[TableSelector]-FIRST[TableSelector]+1);
    sum, inc, delta, remainder: INTEGER;
    d: ARRAY TableSelector OF INTEGER;
    newBase: ARRAY TableSelector OF TableBase;
    sb, db: POINTER;
    newOrigin, newLimit: CARDINAL;
    sum ← tableLimit;  inc ← 0;
    FOR j IN TableSelector
      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[tableLimit/20, 100B]
      DO
      [origin:newOrigin, limit:newLimit] ← SIGNAL TableOverflow;
  FOR j IN TableSelector
   DO
   tbase[j] ← tbase[j] + (newOrigin-tableOrigin);
   ENDLOOP;
  sum ← sum + (newLimit-tableLimit);
  tableOrigin ← newOrigin;  tableLimit ← newLimit;
  ENDLOOP;
 delta ← sum/(10*nTables);
 remainder ← sum - delta*nTables;
 newBase[FIRST[TableSelector]] ← tbase[FIRST[TableSelector]];
 FOR j IN (FIRST[TableSelector] .. LAST[TableSelector]]
  DO
  newBase[j] ← newBase[j-1] + top[j-1] + delta +
   InlineDefs.LongDiv[
    num: InlineDefs.LongMult[d[j-1], remainder],
    den:inc];
  ENDLOOP;
 j ← FIRST[TableSelector]+1;
 UNTIL j > LAST[TableSelector]
  DO
  SELECT newBase[j] FROM
   < tbase[j] =>
    BEGIN
    InlineDefs.COPY[
     from: LOOPHOLE[tbase[j]],
     to: LOOPHOLE[newBase[j]],
     nwords: MIN[top[j], limit[j]]];
    tbase[j] ← newBase[j];
    j ← j+1;
    END;
   > tbase[j] =>
    BEGIN  k ← j+1;
    UNTIL k > LAST[TableSelector] OR newBase[k] <= tbase[k]
     DO
     k ← k+1;
     ENDLOOP;
    FOR m DECREASING IN [j .. k)
     DO
     sb ← LOOPHOLE[tbase[m]];  db ← LOOPHOLE[newBase[m]];
     FOR i DECREASING IN [0 .. MIN[top[m], limit[m]])
      DO
      (db+i)↑ ← (sb+i)↑;
      ENDLOOP;
     tbase[m] ← newBase[m];
     ENDLOOP;
    j ← k;
    END;
   ENDCASE =>  j ← j+1;
  ENDLOOP;
 oldTop ← top;
 sum ← tableLimit;
 FOR j IN [FIRST[TableSelector] .. LAST[TableSelector])
  DO
  limit[j] ← MIN[LOOPHOLE[tbase[j+1]-tbase[j], CARDINAL], TableLimit];
  sum ← sum - limit[j];
  ENDLOOP;
 limit[LAST[TableSelector]] ← sum;
 UpdateBases[];  RETURN
 END;



-- linked list allocation (first subzone)

Chunk: TYPE = MACHINE DEPENDENT RECORD [
 free, fill1: BOOLEAN,
 size: [0..TableLimit),
 fill2: [0..3],
 flink: CIndex,
 fill3: [0..3],
 bLink: CIndex];

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

NullChunkIndex: CIndex = LAST[CIndex];
```

```
chunkRover: CIndex;

GetChunk: PUBLIC PROCEDURE [size: CARDINAL] RETURNS [TableIndex] =
  BEGIN
  cb: TableBase = tbase[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;
  (tbase[chunktype]+q).free ← FALSE;  RETURN [q]
  END;

FreeChunk: PUBLIC PROCEDURE [i: TableIndex, size: CARDINAL] =
  BEGIN
  cb: TableBase = tbase[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[DESCRIPTOR[tbase]];  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[DESCRIPTOR[tbase]];
    ENDLOOP;
  RETURN
  END;


-- initialization, expansion and termination

InitializeTable: PUBLIC PROCEDURE [origin, size: CARDINAL] =
  BEGIN
  d: CARDINAL;
  i: TableSelector;
  IF tableOpen THEN EraseTable[];
  tableOrigin ← origin;  tableLimit ← size;
  d ← tableLimit/(LAST[TableSelector]-FIRST[TableSelector]+1);
  FOR i IN TableSelector
    DO
    tbase[i] ← origin;   origin ← origin + d;
    limit[i] ← d;  top[i] ← oldTop[i] ← 0;
    ENDLOOP;
  chunkRover ← NullChunkIndex;
  notifyList ← NIL;
  tableOpen ← TRUE;   RETURN
  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 ...
```