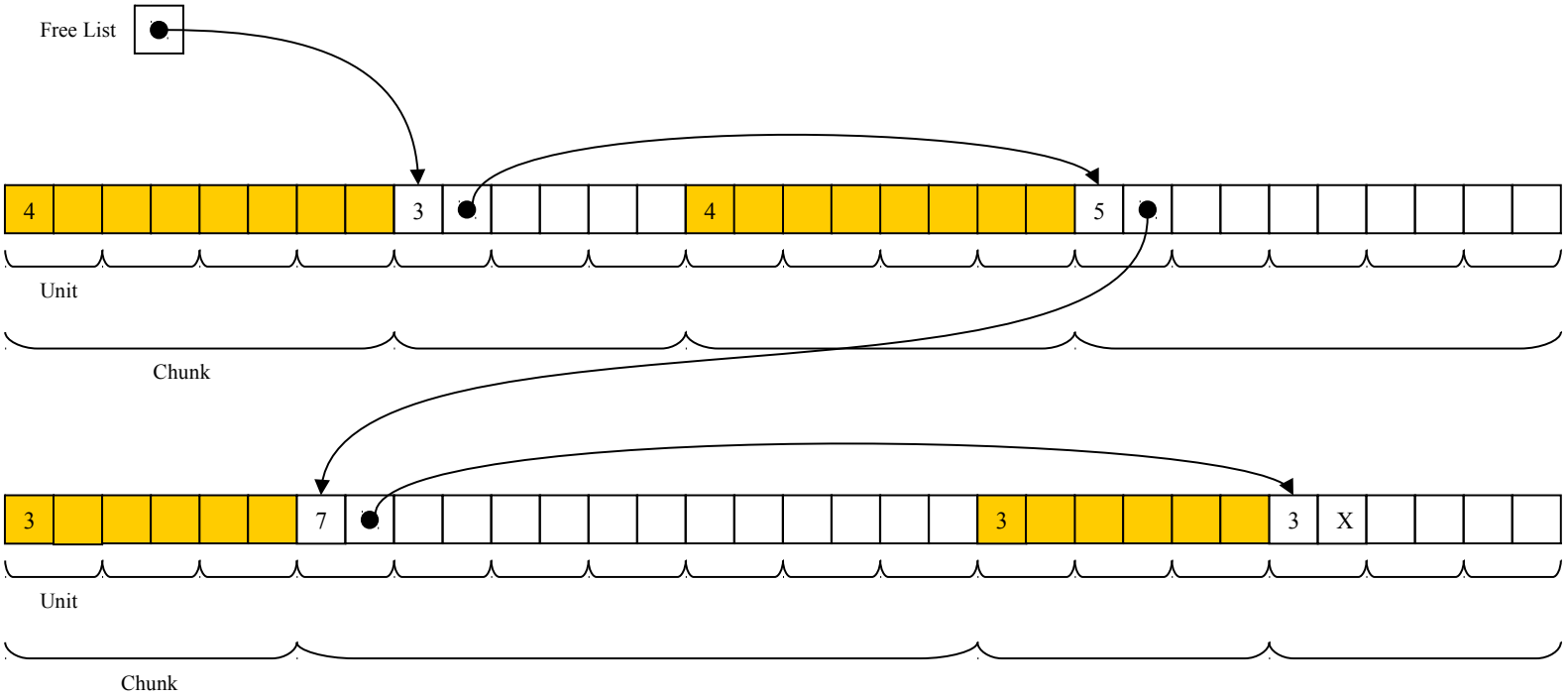
Princeton University
COS 217: Introduction to Programming Systems
HeapMgr3 Data Structures



Each box consists of 8 bytes.
Each Chunk's header Unit contains a length and, if the Chunk is free, a pointer to the next Chunk in the Free List.
The Chunks in the Free List are sorted in increasing order by memory address.
A global variable points to the first Chunk in the Free List.

Princeton University
COS 217: Introduction to Programming Systems
HeapMgr3 Algorithms

void *HeapMgr_malloc(size_t uiBytes)

(1) If this is the first call of HeapMgr_malloc(), then initialize the heap manager.

(2) Determine the number of units the new chunk should contain.

(3) For each chunk in the free list...

    If the current free list chunk is big enough, then...

        If the current free list chunk is close to the requested size, then remove it from the free list and return it. If the current free list chunk is too big, then split the chunk and return the tail end of it. In the latter case, the free list need not be altered.

(4) Ask the OS for more memory — enough for the new chunk. Return NULL if the OS refuses. Create a new chunk using that memory. Insert the new chunk at the end of the free list. If appropriate, coalesce the new chunk and the previous one. Let the current free list chunk be the last one.

(5) If the current free list chunk is close to the requested size, then remove it from the free list and return it. If the current free list chunk is too big, then split the chunk and return the tail end of it. In the latter case, the free list need not be altered.

void HeapMgr_free(void *pvBytes)

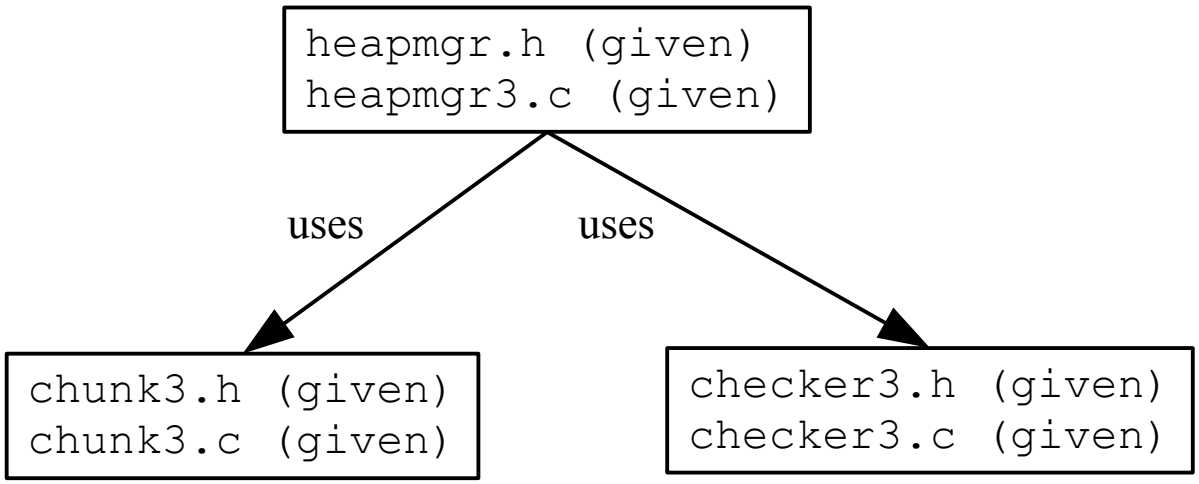
(1) Traverse the free list to find the correct spot for the given chunk.

(2) Insert the given chunk into the free list at the correct spot.

(3) If appropriate, coalesce the given chunk and the next one.

(4) If appropriate, coalesce the given chunk and the previous one.

Princeton University
COS 217: Introduction to Programming Systems
HeapMgr3 Code



chunk3.h (Page 1 of 2)

```
 1: /*-------------------------------------------------------------------*/
 2: /* chunk3.h                                                          */
 3: /* Author: Bob Dondero                                               */
 4: /*-------------------------------------------------------------------*/
 5:
 6: #ifndef CHUNK3_INCLUDED
 7: #define CHUNK3_INCLUDED
 8:
 9: #include <stddef.h>
10:
11: /* A Chunk is a sequence of units. The first unit is a header that
12:    contains the number of units in the Chunk and, if the Chunk is
13:    free, the address of the next Chunk in the free list. The
14:    subsequent units are the payload. */
15:
16: typedef struct Chunk *Chunk_T;
17:
18: /*-------------------------------------------------------------------*/
19:
20: /* The minimum number of units that a Chunk can contain. */
21:
22: static const size_t MIN_UNITS_PER_CHUNK = 2;
23:
24: /*-------------------------------------------------------------------*/
25:
26: /* Translate uBytes, a number of bytes, to units. Return the result. */
27:
28: size_t Chunk_bytesToUnits(size_t uBytes);
29:
30: /*-------------------------------------------------------------------*/
31:
32: /* Translate uUnits, a number of units, to bytes. Return the result. */
33:
34: size_t Chunk_unitsToBytes(size_t uUnits);
35:
36: /*-------------------------------------------------------------------*/
37:
38: /* Return the address of the payload of oChunk. */
39:
40: void *Chunk_toPayload(Chunk_T oChunk);
41:
42: /*-------------------------------------------------------------------*/
43:
44: /* Return the Chunk whose payload is pointed to by pv. */
45:
46: Chunk_T Chunk_fromPayload(void *pv);
47:
48: /*-------------------------------------------------------------------*/
49:
50: /* Return oChunk's number of units. */
51:
52: size_t Chunk_getUnits(Chunk_T oChunk);
53:
54: /*-------------------------------------------------------------------*/
55:
56: /* Set oChunk's number of units to uUnits. */
57:
58: void Chunk_setUnits(Chunk_T oChunk, size_t uUnits);
59:
60: /*-------------------------------------------------------------------*/
61:
62: /* Return oChunk's next Chunk in the free list, or NULL if there
63:    is no next Chunk. */
```

chunk3.h (Page 2 of 2)

```
64:
65: Chunk_T Chunk_getNextInList(Chunk_T oChunk);
66:
67: /*--------------------------------------------------------------------*/
68:
69: /* Set oChunk's next Chunk in the free list to oNextChunk. */
70:
71: void Chunk_setNextInList(Chunk_T oChunk, Chunk_T oNextChunk);
72:
73: /*--------------------------------------------------------------------*/
74:
75: /* Return oChunk's next Chunk in memory, or NULL if there is no
76:    next Chunk. Use oHeapEnd to determine if there is no next
77:    Chunk. oChunk's number of units must be set properly for this
78:    function to work. */
79:
80: Chunk_T Chunk_getNextInMem(Chunk_T oChunk, Chunk_T oHeapEnd);
81:
82: /*--------------------------------------------------------------------*/
83:
84: /* Return 1 (TRUE) if oChunk is valid, notably with respect to
85:    oHeapStart and oHeapEnd, or 0 (FALSE) otherwise. */
86:
87: int Chunk_isValid(Chunk_T oChunk,
88:                   Chunk_T oHeapStart, Chunk_T oHeapEnd);
89:
90: #endif
91:
```

chunk3.c (Page 1 of 3)

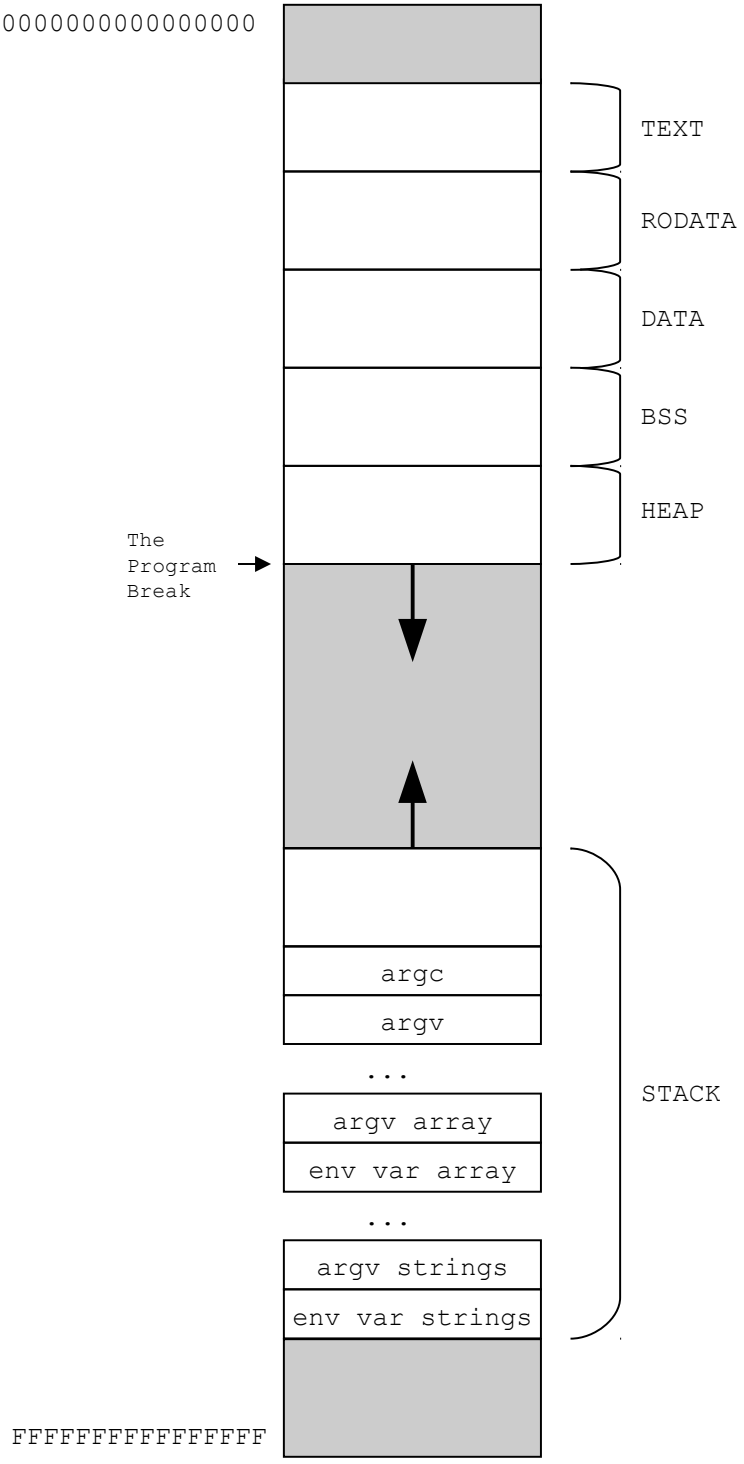
```
 1: /*--------------------------------------------------------------------*/
 2: /* chunk3.c                                                           */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include "chunk3.h"
 7: #include <stddef.h>
 8: #include <stdio.h>
 9: #include <assert.h>
10:
11: /*--------------------------------------------------------------------*/
12:
13: /* Physically a Chunk is a structure consisting of a number of units
14:    and an address. Logically a Chunk consists of multiple such
15:    structures. */
16:
17: struct Chunk
18: {
19:    /* The number of units in the Chunk. */
20:    size_t uUnits;
21:
22:    /* The address of the next Chunk. */
23:    Chunk_T oNextChunk;
24: };
25:
26: /*--------------------------------------------------------------------*/
27:
28: size_t Chunk_bytesToUnits(size_t uBytes)
29: {
30:    size_t uUnits;
31:    uUnits = ((uBytes - 1) / sizeof(struct Chunk)) + 1;
32:    uUnits++;  /* Allow room for a header. */
33:    return uUnits;
34: }
35:
36: /*--------------------------------------------------------------------*/
37:
38: size_t Chunk_unitsToBytes(size_t uUnits)
39: {
40:    return uUnits * sizeof(struct Chunk);
41: }
42:
43: /*--------------------------------------------------------------------*/
44:
45: void *Chunk_toPayload(Chunk_T oChunk)
46: {
47:    assert(oChunk != NULL);
48:
49:    return (void*)(oChunk + 1);
50: }
51:
52: /*--------------------------------------------------------------------*/
53:
54: Chunk_T Chunk_fromPayload(void *pv)
55: {
56:    assert(pv != NULL);
57:
58:    return (Chunk_T)pv - 1;
59: }
60:
61: /*--------------------------------------------------------------------*/
62:
63: size_t Chunk_getUnits(Chunk_T oChunk)
```

```
   64: {
   65:    assert(oChunk != NULL);
   66:
   67:    return oChunk->uUnits;
   68: }
   69:
   70: /*-------------------------------------------------------------------*/
   71:
   72: void Chunk_setUnits(Chunk_T oChunk, size_t uUnits)
   73: {
   74:    assert(oChunk != NULL);
   75:    assert(uUnits >= MIN_UNITS_PER_CHUNK);
   76:
   77:    oChunk->uUnits = uUnits;
   78: }
   79:
   80: /*-------------------------------------------------------------------*/
   81:
   82: Chunk_T Chunk_getNextInList(Chunk_T oChunk)
   83: {
   84:    assert(oChunk != NULL);
   85:
   86:    return oChunk->oNextChunk;
   87: }
   88:
   89: /*-------------------------------------------------------------------*/
   90:
   91: void Chunk_setNextInList(Chunk_T oChunk, Chunk_T oNextChunk)
   92: {
   93:    assert(oChunk != NULL);
   94:
   95:    oChunk->oNextChunk = oNextChunk;
   96: }
   97:
   98: /*-------------------------------------------------------------------*/
   99:
  100: Chunk_T Chunk_getNextInMem(Chunk_T oChunk, Chunk_T oHeapEnd)
  101: {
  102:    Chunk_T oNextChunk;
  103:
  104:    assert(oChunk != NULL);
  105:    assert(oHeapEnd != NULL);
  106:    assert(oChunk < oHeapEnd);
  107:
  108:    oNextChunk = oChunk + Chunk_getUnits(oChunk);
  109:    assert(oNextChunk <= oHeapEnd);
  110:
  111:    if (oNextChunk == oHeapEnd)
  112:       return NULL;
  113:    return oNextChunk;
  114: }
  115:
  116: /*-------------------------------------------------------------------*/
  117:
  118: int Chunk_isValid(Chunk_T oChunk,
  119:                   Chunk_T oHeapStart, Chunk_T oHeapEnd)
  120: {
  121:    assert(oChunk != NULL);
  122:    assert(oHeapStart != NULL);
  123:    assert(oHeapEnd != NULL);
  124:
  125:    if (oChunk < oHeapStart)
  126:    {  fprintf(stderr, "A chunk starts before the heap start\n");
```

chunk3.c (Page 3 of 3)

```
127:       return 0;
128:    }
129:    if (oChunk >= oHeapEnd)
130:    {  fprintf(stderr, "A chunk starts after the heap end\n");
131:       return 0;
132:    }
133:    if (oChunk + Chunk_getUnits(oChunk) > oHeapEnd)
134:    {  fprintf(stderr, "A chunk ends after the heap end\n");
135:       return 0;
136:    }
137:    if (Chunk_getUnits(oChunk) == 0)
138:    {  fprintf(stderr, "A chunk has zero units\n");
139:       return 0;
140:    }
141:    if (Chunk_getUnits(oChunk) < MIN_UNITS_PER_CHUNK)
142:    {  fprintf(stderr, "A chunk has too few units\n");
143:       return 0;
144:    }
145:    return 1;
146: }
147:
```

Princeton University
COS 217: Introduction to Programming Systems
The Memory Layout of a Linux Process



heapmgr3.c (Page 1 of 4)

```
 1: /*-------------------------------------------------------------------------*/
 2: /* heapmgr3.c                                                              */
 3: /* Author: Bob Dondero (similar to K&R version)                            */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #define _GNU_SOURCE
 7:
 8: #include "heapmgr.h"
 9: #include "checker3.h"
10: #include "chunk3.h"
11: #include <stddef.h>
12: #include <assert.h>
13: #include <unistd.h>
14:
15: /*-------------------------------------------------------------------------*/
16:
17: /* The state of the HeapMgr. */
18:
19: /* The address of the start of the heap. */
20: static Chunk_T oHeapStart = NULL;
21:
22: /* The address immediately beyond the end of the heap. */
23: static Chunk_T oHeapEnd = NULL;
24:
25: /* The free list is a list of all free Chunks. It is kept in
26:    ascending order by memory address. */
27: static Chunk_T oFreeList = NULL;
28:
29: /*-------------------------------------------------------------------------*/
30:
31: /* Request more memory from the operating system -- enough to store
32:    uUnits units. Create a new chunk, and either append it to the
33:    free list after oPrevChunk or increase the size of oPrevChunk.
34:    Return the address of the new (or enlarged) chunk. */
35:
36: static Chunk_T HeapMgr_getMoreMemory(Chunk_T oPrevChunk, size_t uUnits)
37: {
38:    const size_t MIN_UNITS_FROM_OS = 512;
39:    Chunk_T oChunk;
40:    Chunk_T oNewHeapEnd;
41:    size_t uBytes;
42:
43:    if (uUnits < MIN_UNITS_FROM_OS)
44:       uUnits = MIN_UNITS_FROM_OS;
45:
46:    /* Move the program break. */
47:    uBytes = Chunk_unitsToBytes(uUnits);
48:    oNewHeapEnd = (Chunk_T)((char*)oHeapEnd + uBytes);
49:    if (oNewHeapEnd < oHeapEnd)  /* Check for overflow */
50:       return NULL;
51:    if (brk(oNewHeapEnd) == -1)
52:       return NULL;
53:    oChunk = oHeapEnd;
54:    oHeapEnd = oNewHeapEnd;
55:
56:    /* Set the fields of the new chunk. */
57:    Chunk_setUnits(oChunk, uUnits);
58:    Chunk_setNextInList(oChunk, NULL);
59:
60:    /* Add the new chunk to the end of the free list. */
61:    if (oPrevChunk == NULL)
62:       oFreeList = oChunk;
63:    else
```

```
   64:       Chunk_setNextInList(oPrevChunk, oChunk);
   65:
   66:    /* Coalesce the new chunk and the previous one if appropriate. */
   67:    if (oPrevChunk != NULL)
   68:       if (Chunk_getNextInMem(oPrevChunk, oHeapEnd) == oChunk)
   69:       {
   70:          Chunk_setUnits(oPrevChunk,
   71:             Chunk_getUnits(oPrevChunk) + uUnits);
   72:          Chunk_setNextInList(oPrevChunk, NULL);
   73:          oChunk = oPrevChunk;
   74:       }
   75:
   76:    return oChunk;
   77: }
   78:
   79: /*-------------------------------------------------------------------*/
   80:
   81: /* If oChunk is close to the right size (as specified by uUnits),
   82:    then splice oChunk out of the free list (using oPrevChunk to do
   83:    so), and return oChunk. If oChunk is too big, split it and return
   84:    the address of the tail end.  */
   85:
   86: static Chunk_T HeapMgr_useChunk(Chunk_T oChunk,
   87:    Chunk_T oPrevChunk, size_t uUnits)
   88: {
   89:    Chunk_T oNewChunk;
   90:    size_t uChunkUnits;
   91:
   92:    assert(Chunk_isValid(oChunk, oHeapStart, oHeapEnd));
   93:
   94:    uChunkUnits = Chunk_getUnits(oChunk);
   95:
   96:    /* If oChunk is close to the right size, then use it. */
   97:    if (uChunkUnits < uUnits + MIN_UNITS_PER_CHUNK)
   98:    {
   99:       if (oPrevChunk == NULL)
  100:          oFreeList = Chunk_getNextInList(oChunk);
  101:       else
  102:          Chunk_setNextInList(oPrevChunk, Chunk_getNextInList(oChunk));
  103:       return oChunk;
  104:    }
  105:
  106:    /* oChunk is too big, so use the tail end of it. */
  107:    Chunk_setUnits(oChunk, uChunkUnits - uUnits);
  108:    oNewChunk = Chunk_getNextInMem(oChunk, oHeapEnd);
  109:    Chunk_setUnits(oNewChunk, uUnits);
  110:    return oNewChunk;
  111: }
  112:
  113: /*-------------------------------------------------------------------*/
  114:
  115: void *HeapMgr_malloc(size_t uBytes)
  116: {
  117:    Chunk_T oChunk;
  118:    Chunk_T oPrevChunk;
  119:    Chunk_T oPrevPrevChunk;
  120:    size_t uUnits;
  121:
  122:    if (uBytes == 0)
  123:       return NULL;
  124:
  125:    /* Step 1: Initialize the heap manager if this is the first call. */
  126:    if (oHeapStart == NULL)
```

```
127:    {
128:       oHeapStart = (Chunk_T)sbrk(0);
129:       oHeapEnd = oHeapStart;
130:    }
131:
132:    assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
133:
134:    /* Step 2: Determine the number of units the new chunk should
135:       contain. */
136:    uUnits = Chunk_bytesToUnits(uBytes);
137:
138:    /* Step 3: For each chunk in the free list... */
139:    oPrevPrevChunk = NULL;
140:    oPrevChunk = NULL;
141:    for (oChunk = oFreeList;
142:         oChunk != NULL;
143:         oChunk = Chunk_getNextInList(oChunk))
144:    {
145:       /* If oChunk is big enough, then use it. */
146:       if (Chunk_getUnits(oChunk) >= uUnits)
147:       {
148:          oChunk = HeapMgr_useChunk(oChunk, oPrevChunk, uUnits);
149:          assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
150:          return Chunk_toPayload(oChunk);
151:       }
152:
153:       oPrevPrevChunk = oPrevChunk;
154:       oPrevChunk = oChunk;
155:    }
156:
157:    /* Step 4: Ask the OS for more memory, and create a new chunk (or
158:       expand the existing chunk) at the end of the free list. */
159:    oChunk =  HeapMgr_getMoreMemory(oPrevChunk, uUnits);
160:    if (oChunk == NULL)
161:    {
162:       assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
163:       return NULL;
164:    }
165:
166:    /* If the new large chunk was coalesced with the previous chunk,
167:       then reset the previous chunk. */
168:    if (oChunk == oPrevChunk)
169:       oPrevChunk = oPrevPrevChunk;
170:
171:    /* Step 5: oChunk is big enough, so use it. */
172:    oChunk = HeapMgr_useChunk(oChunk, oPrevChunk, uUnits);
173:    assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
174:    return Chunk_toPayload(oChunk);
175: }
176:
177: /*--------------------------------------------------------------------*/
178:
179: void HeapMgr_free(void *pv)
180: {
181:    Chunk_T oChunk;
182:    Chunk_T oNextChunk;
183:    Chunk_T oPrevChunk;
184:
185:    assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
186:
187:    if (pv == NULL)
188:       return;
189:
```

heapmgr3.c (Page 4 of 4)

```
190:    oChunk = Chunk_fromPayload(pv);
191:
192:    assert(Chunk_isValid(oChunk, oHeapStart, oHeapEnd));
193:
194:    /* Step 1: Traverse the free list to find the correct spot for
195:       oChunk. (The free list is kept in ascending order by memory
196:       address.) */
197:    oPrevChunk = NULL;
198:    oNextChunk = oFreeList;
199:    while ((oNextChunk != NULL) && (oNextChunk < oChunk))
200:    {
201:       oPrevChunk = oNextChunk;
202:       oNextChunk = Chunk_getNextInList(oNextChunk);
203:    }
204:
205:    /* Step 2: Insert oChunk into the free list. */
206:    if (oPrevChunk == NULL)
207:       oFreeList = oChunk;
208:    else
209:       Chunk_setNextInList(oPrevChunk, oChunk);
210:    Chunk_setNextInList(oChunk, oNextChunk);
211:
212:    /* Step 3: If appropriate, coalesce the given chunk and the next
213:       one. */
214:    if (oNextChunk != NULL)
215:       if (Chunk_getNextInMem(oChunk, oHeapEnd) == oNextChunk)
216:       {
217:          Chunk_setUnits(oChunk,
218:             Chunk_getUnits(oChunk) + Chunk_getUnits(oNextChunk));
219:          Chunk_setNextInList(oChunk,
220:             Chunk_getNextInList(oNextChunk));
221:       }
222:
223:    /* Step 4: If appropriate, coalesce the given chunk and the previous
224:       one. */
225:    if (oPrevChunk != NULL)
226:       if (Chunk_getNextInMem(oPrevChunk, oHeapEnd) == oChunk)
227:       {
228:          Chunk_setUnits(oPrevChunk,
229:             Chunk_getUnits(oPrevChunk) + Chunk_getUnits(oChunk));
230:          Chunk_setNextInList(oPrevChunk,
231:             Chunk_getNextInList(oChunk));
232:       }
233:
234:    assert(Checker_isValid(oHeapStart, oHeapEnd, oFreeList));
235: }
```

checker3.h (Page 1 of 1)

```
 1: /*-------------------------------------------------------------------------*/
 2: /* checker3.h                                                              */
 3: /* Author: Bob Dondero                                                     */
 4: /*-------------------------------------------------------------------------*/
 5:
 6: #ifndef CHECKER3_INCLUDED
 7: #define CHECKER3_INCLUDED
 8:
 9: #include "chunk3.h"
10:
11: /* Return 1 (TRUE) if the heap is in a valid state, or 0 (FALSE)
12:    otherwise. The heap is defined by parameters oHeapStart (the address
13:    of the start of the heap), oHeapEnd (the address immediately
14:    beyond the end of the heap), and oFreeList (a list containing free
15:    chunks). */
16:
17: int Checker_isValid(Chunk_T oHeapStart, Chunk_T oHeapEnd,
18:    Chunk_T oFreeList);
19:
20: #endif
```

checker3.c (Page 1 of 2)

```
 1: /*--------------------------------------------------------------------*/
 2: /* checker3.c                                                         */
 3: /* Author: Bob Dondero                                                */
 4: /*--------------------------------------------------------------------*/
 5:
 6: #include "checker3.h"
 7: #include <stdio.h>
 8:
 9: /* In lieu of a boolean data type. */
10: enum {FALSE, TRUE};
11:
12: /*--------------------------------------------------------------------*/
13:
14: int Checker_isValid(Chunk_T oHeapStart, Chunk_T oHeapEnd,
15:    Chunk_T oFreeList)
16: {
17:    Chunk_T oChunk;
18:    Chunk_T oPrevChunk;
19:    Chunk_T oTortoiseChunk;
20:    Chunk_T oHareChunk;
21:
22:    /* Do oHeapStart and oHeapEnd have non-NULL values? */
23:    if (oHeapStart == NULL)
24:    {
25:       fprintf(stderr, "The heap start is uninitialized\n");
26:       return FALSE;
27:    }
28:    if (oHeapEnd == NULL)
29:    {
30:       fprintf(stderr, "The heap end is uninitialized\n");
31:       return FALSE;
32:    }
33:
34:    /* If the heap is empty, is the free list empty too? */
35:    if (oHeapStart == oHeapEnd)
36:    {
37:       if (oFreeList == NULL)
38:          return TRUE;
39:       else
40:       {
41:          fprintf(stderr, "The heap is empty, but the list is not.\n");
42:          return FALSE;
43:       }
44:    }
45:
46:    /* Traverse memory. */
47:
48:    for (oChunk = oHeapStart;
49:         oChunk != NULL;
50:         oChunk = Chunk_getNextInMem(oChunk, oHeapEnd))
51:
52:       /* Is the chunk valid? */
53:       if (! Chunk_isValid(oChunk, oHeapStart, oHeapEnd))
54:       {
55:          fprintf(stderr, "Traversing memory detected a bad chunk\n");
56:          return FALSE;
57:       }
58:
59:    /* Is the list devoid of cycles? Use Floyd's algorithm to find out.
60:       See the Wikipedia "Cycle detection" page for a description. */
61:
62:    oTortoiseChunk = oFreeList;
63:    oHareChunk = oFreeList;
```

checker3.c (Page 2 of 2)

```
64:    if (oHareChunk != NULL)
65:       oHareChunk = Chunk_getNextInList(oHareChunk);
66:    while (oHareChunk != NULL)
67:    {
68:       if (oTortoiseChunk == oHareChunk)
69:       {
70:          fprintf(stderr, "The list has a cycle\n");
71:          return FALSE;
72:       }
73:       /* Move oTortoiseChunk one step. */
74:       oTortoiseChunk = Chunk_getNextInList(oTortoiseChunk);
75:       /* Move oHareChunk two steps, if possible. */
76:       oHareChunk = Chunk_getNextInList(oHareChunk);
77:       if (oHareChunk != NULL)
78:          oHareChunk = Chunk_getNextInList(oHareChunk);
79:    }
80:
81:    /* Traverse the free list. */
82:
83:    oPrevChunk = NULL;
84:    for (oChunk = oFreeList;
85:         oChunk != NULL;
86:         oChunk = Chunk_getNextInList(oChunk))
87:    {
88:       /* Is the chunk valid? */
89:       if (! Chunk_isValid(oChunk, oHeapStart, oHeapEnd))
90:       {
91:          fprintf(stderr, "Traversing the list detected a bad chunk\n");
92:          return FALSE;
93:       }
94:
95:       /* Is the chunk in the proper place in the list? */
96:       if ((oPrevChunk != NULL) && (oPrevChunk >= oChunk))
97:       {
98:          fprintf(stderr, "The list is unordered\n");
99:          return FALSE;
100:      }
101:
102:      /* Is the previous chunk in memory in use? */
103:      if ((oPrevChunk != NULL) &&
104:          (Chunk_getNextInMem(oPrevChunk, oHeapEnd) == oChunk))
105:      {
106:         fprintf(stderr, "The heap contains contiguous free chunks\n");
107:         return FALSE;
108:      }
109:
110:      oPrevChunk = oChunk;
111:   }
112:
113:   return TRUE;
114: }
```