Internally, the kernel uses this block allocation mechanism through calls that operate like the familiar *malloc*( ) and *free*( ) routines. User programs written in Limbo don't directly allocate memory like UNIX processes do using the *brk*( ) system call or the *malloc*( ) library call. Consequently, Inferno does not provide any form of memory allocation system call for user processes. There are, however, a number of Limbo constructs that do implicitly allocate memory. The **spawn** statement implicitly allocates memory when creating a new process. Similarly, the **load** statement must allocate space to load a new module into memory. As with most systems, stack space grows automatically with usage. Finally, there are two Dis instructions, `new` and `newa`, which are used when a program dynamically creates a new data structure or a new array. Limbo programs do not directly free any memory they allocate. Instead, Inferno uses a combination of reference counts and mark-and-sweep garbage collection to identify and free memory no longer used by user processes.

11.2 Memory Layouts

Most of our discussions of memory layouts make a distinction between virtual memory space and physical memory space. Because Inferno doesn't use any address translation, these two spaces are the same, and we refer to it simply as the memory space.

The pool/arena allocation strategy has significant implications for the overall memory layout. Aside from the kernel image itself, memory is simply sliced up into arenas according to the sizes of the requests as they are issued. There are no fixed assignments of components in the memory space.

As in most systems, processes in Inferno have separate areas of memory for code and for data. However, unlike most systems, Inferno does not place these areas in particular locations in the address space. Each of the modules used by a process has its own code that occupies the memory block where it was loaded. Similarly, each module has a global data block. All dynamic data items and stack also occupy the blocks that are allocated for them. Because no address translation is done, a process cannot depend on these various memory areas being at known locations. Furthermore, because Limbo does not provide the capability of manipulating arbitrary pointers, a user process doesn't have any way to use that information anyway.

The remainder of this chapter presents a detailed examination of the implementation of Inferno memory management. We begin with the data structures used to represent memory and move on to the functions that handle block allocation and deallocation.

11.3 Memory Management Data Structures

We turn now to the details of how the general memory management techniques in Inferno are implemented. There are two main data structures representing memory in Inferno. The first of these represents pools. There is an array of these, one for each pool. The second is used to represent both arenas and blocks. Blocks are represented by a pair of structures that form a header and a tail. Arenas don't have their own data structure to describe them. Instead, they are treated much like large blocks that are then subdivided into smaller blocks.

11.3.1 Memory Pools

As explained earlier, blocks of memory are allocated from the **main pool**, the **heap pool**, and the **image pool**. For allocations made within the kernel, as well as allocations in some of the built-in modules, the main pool is used. The image pool is used by the code in libdraw for storing image data. The heap pool is used by the code in libinterp for some allocations needed in interpreting Dis bytecode.

Each pool is represented by the following data structure defined in emu/port/alloc.c for hosted builds and os/port/alloc.c for native builds:

```
struct Pool {
  char *name;
  int pnum;
  ulong maxsize;
  int quanta;
  int chunk;
  int monitor;
  ulong ressize;      /* restricted size */
  ulong cursize;
  ulong arenasize;
  ulong hw;
  Lock l;
  Bhdr *root;
  Bhdr *chain;
  ulong nalloc;
  ulong nfree;
  int nbrk;
  int lastfree;
  void (*move)(void *, void *);
};
```

Before looking at each structure member, we examine how the instances of this structure are initialized:

```
struct {
  int n;
  Pool pool[MAXPOOL];
      /* Lock l; */
} table = {
  3,
  {
    {"main", 0, 32*1024*1024, 31, 512*1024, 0, 31*1024*1024},
    {"heap", 1, 32*1024*1024, 31, 512*1024, 0, 31*1024*1024},
    {"image", 2, 32*1024*1024+256, 31, 4*1024*1024, 1,
      31*1024*1024},
```

```
    }
};
Pool *mainmem = &table.pool[0];
Pool *heapmem = &table.pool[1];
Pool *imagmem = &table.pool[2];
```

In this structure named *table*, the member n is the number of pools. As indicated previously, there are three pools. The remainder of *table* is an array of three pool structures that are partially initialized.

In the **Pool** structure, the *name* member holds a descriptive string for that pool. Here, we have the pools named `main`, `heap`, and `image`. The *pnum* member is a numeric identifier and is equal to the index into the array of pool structures. When allocating new arenas for the pool, we are careful never to exceed *maxsize* bytes for the pool. In the declarations given earlier, each of the pools is limited to 32 MB. For a hosted implementation, these maximum sizes can be overridden by command-line arguments. Generally, any memory allocation system will allocate only in multiples of some minimum allocation size. We specify that minimum size with the *quanta* member. However, this value does not give the allocation unit directly. Instead, the value stored here is actually $2^q - 1$, where 2^q is the minimum allocation size. Another way to think about this is that *quanta* is a mask with 1s in the bits that must all be 0 for any valid allocation size. We see exactly how this gets used later. Similarly, when allocating a new arena for the pool, we allocate with a minimum size specified by the *chunk* structure member. The *monitor* flag is used as a switch to enable or disable calling a monitoring function when memory is allocated or freed from that pool. This is enabled only for the image pool. The last structure member initialized at declaration time is *ressize*. As the comment indicates, this is a restricted size for the pool. The effect of these values of restricted size is that only certain processes may allocate the last megabyte of the pool's allowable space.

As we might guess, *cursize* is used to record the amount of memory currently allocated from the pool. On the other hand, *arenasize* gives the total amount of memory allocated to the pool. For bookkeeping purposes, we use *hw* to record the peak amount of memory allocated out of the pool over the history of the system (a high water mark). Because updating a data structure like this is a complex operation, there are inevitably race conditions with allocation and freeing. We use the lock variable l to provide exclusive access. In the next subsection, we discuss how free blocks are stored in a binary tree. A pool's *root* structure member points to the root of the free block tree. The *chain* member points to a linked list of **Bhdr** structures (discussed next) that describe the arenas allocated to the pool. Figure 11-2 shows the relationship between the **Pool** structure and the various **Bhdr** structures. In this figure, the *clink* pointer is part of the **Bhdr** structure. The next three structure members are used for mostly statistical purposes, where *nalloc* records the number of times a block is allocated from the pool, *nfree* does the same for calls to free blocks, and *nbrk* records the number of times a new arena is added to the pool. The *lastfree* member records the number of frees that had occurred as of the last time the pool was compacted. It is used so that we don't go to the trouble of compacting again if there have been no freeing operations in the meantime. The last member of the structure

is a function pointer called *move*. This function is used as part of the pool compacting operation.

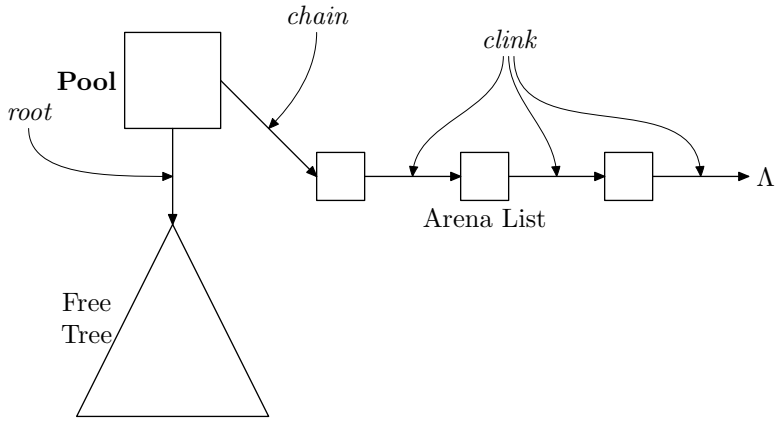

Figure 11-2: Pool Data Structure, Arena List, and Free Tree

11.3.2 Memory Blocks

We now move to the structure of the header at the beginning of each block. This block header is declared in include/pool.h as follows:

```
struct Bhdr {
  ulong magic;
  ulong size;
  union {
    uchar data[1];
    struct {
      Bhdr *bhl;
      Bhdr *bhr;
      Bhdr *bhp;
      Bhdr *bhv;
      Bhdr *bhf;
    } s;
#define clink  u.l.link
#define csize  u.l.size
    struct {
      Bhdr *link;
      int size;
    } l;
```

```
  } u;
};
```

The *size* member of the **Bhdr** structure gives the size of the block in bytes. The *magic* member of the structure is used to record the current status of this block. It may take on one of the following values:

```
enum {
  MAGIC_A = #a110c,      /* Allocated block */
  MAGIC_F = #badc0c0a,       /* Free block */
  MAGIC_E = #deadbabe,       /* End of arena */
  MAGIC_I = #abba      /* Block is immutable (hidden from gc) */
};
```

As the comments imply, `MAGIC_A` is used to mark a block that is assigned to a particular use, whether it be internal to the kernel or as part of a user process. Blocks that are available to be assigned are marked with `MAGIC_F`. The `MAGIC_E` value is used as a marker to identify the end of an arena we subdivide for allocation. Finally, blocks marked with the value `MAGIC_I` are allocated only within the system rather than in response to process requests for memory. The difference between an immutable block and a regular allocated one is that we skip the immutable ones when scanning for garbage collection. In other words, we don't need to have any explicit references to such a block to keep it allocated.

These values are similar to the hexadecimal "dead beef" used in Chapter 7 to mark terminated processes. Here, we see that allocated blocks are identified with the word *alloc* if we accept the abuse of a 1 (one) for the letter "l" and a 0 (zero) for the letter "o". Similarly, the free blocks are identified as *bad cocoa,* the end of the arena is identified as a *dead babe*, and a block that we do not allow to be garbage collected is an homage to the 1970s musical group AЯBA.

The rest of the data structure deserves a little more explanation. The basic idea is pretty straightforward. When a block is allocated, we have a minimal header (the size and magic number) followed by the data itself. However, when a block is unallocated, we are free to use the data space of the block for our free list bookkeeping. So at times, the bytes that follow the size are used for data whose type is determined by the process to which the block is allocated, and at other times, we use those same bytes as administrative values. This type of multiplicity of purposes is exactly the role for which the **union** type in C was created. Here, we have a union called u with three elements. The first element, *data*, is used when the block is allocated. This array of one element serves to provide us with a pointer to the data area of an allocated block. For example, we use it in the macro:

```
#define B2D(bp)  ((void *) bp→u.data)
```

to provide a mapping from a pointer to a block to a pointer to its data area.

The structure, s, is used for those blocks marked with the `MAGIC_F` magic number—namely those that are normal free blocks. The pointers that make up this structure are

used to maintain a tree of free blocks, where each node in the tree is a doubly linked list of blocks of equal size. If the blocks attached to a given node have $size \equiv n$, then the blocks in the left subtree have $size < n$, and the blocks in the right subtree have $size > n$. With the following definitions found in emu/port/alloc.c:

#define *left* *u.s.bhl*
#define *right* *u.s.bhr*
#define *fwd* *u.s.bhf*
#define *prev* *u.s.bhv*
#define *parent* *u.s.bhp*

we can use *left* and *right* for a node's children in the tree. The *parent* macro identifies a pointer to the node's parent in the tree. Similarly, the *fwd* and *prev* macros are pointers that make up the doubly linked list of equal-sized blocks. The net effect of this arrangement is a binary search tree, where each node represents a given size of free block. Multiple free blocks of the same size are kept in a doubly linked list where only the head of the list is part of the tree structure. Figure 11-3 illustrates this design with a small example of a free tree (or a portion of a larger tree). In this figure, the number in a box is the size of free block represented by that box. We label all the pointers to show the mapping from the figure to the structure definition, but the linked list structure is shown only for the central node. Notice that all free blocks in the right subtree below the block of 128 are larger. Also notice that all the blocks in the linked list in the middle of the figure are the same size.

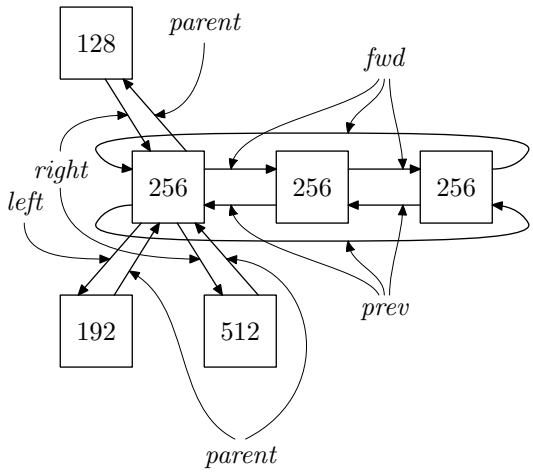

Figure 11-3: Free Blocks

The last structure in the union, called l, is used in those block headers marked with the `MAGIC_E` magic number. These block headers are used as part of the bookkeeping when

a new arena is added to the pool. In particular, the number of bytes in the arena is kept in *size*, and *link* points to the block header structure of the next arena in the list.

Each block also contains a **Btail** structure at the end. The only member of this structure is a pointer, *hdr*, which points back to the **Bhdr** structure at the beginning of the block. This block trailer allows us to quickly find the header that belongs to the block immediately preceding the one we are using.

11.4 Memory Management Implementation

With the data structures representing memory defined, we now turn our attention to the code that manipulates those structures to allocate and free memory. In a very real sense, all memory management operations boil down to adding and removing free blocks from a tree.

11.4.1 Allocating Memory

Most allocations of memory inside the Inferno kernel are made through variations on the familiar *malloc*( ) call, but all allocations are ultimately handled by the function *dopoolalloc*( ), defined in emu/port/alloc.c. It is in this function where we concentrate our study. The basic strategy in *dopoolalloc*( ) can be summarized in the following steps:

1. Traverse the tree to find the best-fit block.
2. If the size of the block we found is close enough to the requested size, we return the block.
3. If we found a block that is much larger, then we split it into an allocated block to return and a free block to put back in the tree.
4. If we didn't find any block at least as large as the request, then we try to allocate a new arena for this pool and repeat the allocation attempt.

The function is declared as follows:

```
static void *dopoolalloc(Pool *p, ulong asize, ulong pc)
{
```

where *p* points to the **Pool** structure describing the pool from which we are allocating, and *asize* gives the size of the allocation request in bytes. The third argument, *pc*, is the program counter of the caller. It is used only when tracing the behavior of the memory manager. Upon successful allocation, *dopoolalloc*( ) returns a pointer to the newly allocated block. On failure, it returns *nil*. Next, we have some typical local variables that we use as we manage the free space.

```
    Bhdr *q, *t;
    int alloc, ldr, ns, frag;
    int osize, size;
```

Good programming practice always requires that we be careful to keep from acting on unreasonable parameters. In this case, we check to make sure that *asize* is not outside the realm of reason.

```
if (asize ≥ 1024 * 1024 * 1024)      /* for sanity and to avoid overflow */
  return nil;
```

11.4.1.1 Adjusting Request Size

Now comes the time to adjust the size of the request. We need to round the request up to the nearest greater quantum, being careful not to forget to make room for the block header structure.

```
size = asize;
osize = size;
size = (size + BHDRSIZE + p→quanta) & ~(p→quanta);
```

The next two lines represent some administrative overhead. First, we have to gain the mutual exclusion lock to prevent any other thread of control from interfering with us as we work on the data structure. Then, we record the fact that our allocation count has now gone up by one.

```
lock(&p→l);
p→nalloc++;
```

11.4.1.2 Searching for the Best Fit

The basic allocation technique in Inferno follows a best-fit strategy. This first loop searches for the case where we have a block in our pool that is an exact fit. If we ignore the large **if**( ) statement inside the loop for now, then the loop is a pretty typical search through a tree that is kept in sorted order. Along the way, we keep the variable q pointing to the smallest block that is larger than the one we're seeking.

```
t = p→root;
q = nil;
while (t) {
  if (t→size ≡ size) {
    t = t→fwd;
    pooldel(p, t);
    t→magic = MAGIC_A;
    p→cursize += t→size;
    if (p→cursize > p→hw)
      p→hw = p→cursize;
    unlock(&p→l);
    if (p→monitor)
      MM(p→pnum, pc, (ulong) B2D(t), size);
```

```
        return B2D(t);
    }
    if (size < t→size) {
        q = t;
        t = t→left;
    }
    else
        t = t→right;
}
```

Now we turn our attention to the **if**( ) statement that we skipped previously. This is the case where we actually find an exact match. In this case, we only need to remove the block from the pool and update *cursize*. It is worth noting here that we advance t to the second element in the list if there is one. (If there's only one element, t→*fwd* leaves t unchanged.) This reduces the amount of work we have to do in taking the block out of the data structure, because only the head of the list is part of the tree structure. Finally, before we return we must release the mutual exclusion lock.

11.4.1.3 Splitting a Large Free Block

However, what happens if we don't find an exact match but have at least one block larger than the one we're seeking? In that case, we may return the block as it is, or we may split the block into two pieces, one of which is the size of the request, and one of which is the remaining free space. First, we remove the block from the pool and calculate how much would be left if we split.

```
if (q ≠ nil) {
    pooldel(p, q);
    q→magic = MAGIC_A;
    frag = q→size − size;
```

There's no need to bother splitting the block if the remaining free space would be too small to be useful. We define this condition by saying that if the fragment would be less than 32 KB and also less than one-quarter the size of the allocation, it's too small to be useful. The hexadecimal value 8000 is 2^{15}, which is 32 K. In this case, we just return the whole thing.

```
    if (frag < (size ≫ 2) ∧ frag < #8000) {
        p→cursize += q→size;
        if (p→cursize > p→hw)
            p→hw = p→cursize;
        unlock(&p→l);
        if (p→monitor)
            MM(p→pnum, pc, (ulong) B2D(q), size);
        return B2D(q);
    }
```

If the fragment would be useful, then we split the block into two parts. We do this by constructing new tail and header structures in the middle of the block so that we now have two adjacent blocks. The allocated one is returned, and the remainder is placed back in the pool's free block structure.

```
    ns = q→size − size;
    q→size = size;
    B2T(q)→hdr = q;
    t = B2NB(q);
    t→size = ns;
    B2T(t)→hdr = t;
    pooladd(p, t);
    p→cursize += q→size;
    if (p→cursize > p→hw)
      p→hw = p→cursize;
    unlock(&p→l);
    if (p→monitor)
      MM(p→pnum, pc, (ulong) B2D(q), size);
    return B2D(q);
  }
```

11.4.1.4 Allocating a New Arena

Now comes the most complex of the scenarios we could encounter. Namely, we didn't find any block as large as the one requested. This means we need to get more memory allocated to the pool. We begin by calculating how big the arena needs to be. We want the larger of *chunk* (from the **Pool** structure) and *size* (the adjusted request size).

```
ns = p→chunk;
if (size > ns)
  ns = size;
ldr = p→quanta + 1;
alloc = ns + ldr + ldr;
p→arenasize += alloc;
```

It's possible that adding enough to this pool will cause us to exceed the *maxsize* limit on the pool size. In that case, we try compacting the pool with the function *poolcompact*( ). If we made some progress compacting, then we recursively attempt to allocate again. If not, then there's nothing we can do except deny the allocation request.

```
if (p→arenasize > p→maxsize) {
  p→arenasize −= alloc;
  ns = p→maxsize − p→arenasize − ldr − ldr;
  ns &= ∼p→quanta;
```

```
    if (ns < size) {
        if (poolcompact(p)) {
            unlock(&p→l);
            return poolalloc(p, osize);
        }
        unlock(&p→l);
        print("arena␣%s␣too␣large:␣size␣%d␣cursize␣%lud
            ␣arenasize␣%lud␣maxsize␣%lud\n", p→name, size,
            p→cursize, p→arenasize, p→maxsize);
        return nil;
    }
    alloc = ns + ldr + ldr;
    p→arenasize += alloc;
}
```

Finally, we come to the point where we attempt to add more memory to the pool. In native Inferno, we call a function, *xalloc*( ), which is specific to the hardware platform and knows how to permanently allocate large memory blocks to pools. In hosted Inferno, we request that the hosting OS give us more memory. We do this through the traditional UNIX call *sbrk*( ), which incrementally moves the border between the data and stack spaces. For host operating systems that do not directly support *sbrk*( ), a function is provided to emulate it.

```
p→nbrk++;
t = (Bhdr *) sbrk(alloc);
if (t ≡ (void *) −1) {
    p→nbrk−−;
    unlock(&p→l);
    return nil;
}
t = (Bhdr *) (((ulong) t + 7) & ∼7);
```

With the new arena added to the pool, it's time to add it to the linked list of these arenas. However, if it turns out that this one abuts the last one we added to this pool, we can just combine them. With the original *sbrk*( ), this will happen exactly when there are no intervening allocations for other pools. The actual mechanism of combination is pretty simple. We pretend that the new block was previously allocated, we mark the older block as being bigger by *alloc* bytes, and, finally, we request that the newly allocated block be freed. As with the compaction case, after we've finished the work, we recursively attempt to satisfy the allocation request.

```
if (p→chain ≠ nil ∧ (char *) t − (char *) B2LIMIT(p→chain) − ldr ≡ 0) {
    if (0)
        print("merging␣chains␣%p␣and␣%p␣in␣%s\n", p→chain, t,
            p→name);
```

```
    q = B2LIMIT(p→chain);
    q→magic = MAGIC_A;
    q→size = alloc;
    B2T(q)→hdr = q;
    t = B2NB(q);
    t→magic = MAGIC_E;
    p→chain→csize += alloc;
    p→cursize += alloc;
    unlock(&p→l);
    poolfree(p, B2D(q));        /* for backward merge */
    return poolalloc(p, osize);
}
```

If, on the other hand, the newly added arena is not adjacent to an old one, we add it as a separate arena to the pool. We do this by building the block header structure for this arena and inserting it at the beginning of the pool's list. At the end, we reset t to point to the data area following the newly created header.

```
t→magic = MAGIC_E;      /* Make a leader */
t→size = ldr;
t→csize = ns + ldr;
t→clink = p→chain;
p→chain = t;
B2T(t)→hdr = t;
t = B2NB(t);
```

Now that the arena management is done, we can carve out of the data area the space we need to satisfy the allocation request.

```
t→magic = MAGIC_A;      /* Make the block we are going to return */
t→size = size;
B2T(t)→hdr = t;
q = t;
```

In the case that the request doesn't use the entire new arena, we'll add the remainder to the free tree. Finally, we clean everything up and return.

```
ns -= size;      /* Free the rest */
if (ns > 0) {
    q = B2NB(t);
    q→size = ns;
    B2T(q)→hdr = q;
    pooladd(p, q);
}
B2NB(q)→magic = MAGIC_E;      /* Mark the end of the chunk */
```

```
    p→cursize += t→size;
    if (p→cursize > p→hw)
      p→hw = p→cursize;
    unlock(&p→l);
    if (p→monitor)
      MM(p→pnum, pc, (ulong) B2D(t), size);
    return B2D(t);
  }
```

The result of all the slicing and dicing of the newly allocated arena is shown in Figure 11-4. In the figure, the boxes marked E are the **Bhdr** structures with *magic* ≡ `MAGIC_E`. In other words, these are the markers at the beginning and the end of the arena, with the first one being placed in the linked list of arenas attached to the **Pool** structure. The box marked A is the **Bhdr** structure for the allocated block, and the one marked F is for the free block. The arrows show the pointers from the block trailers back to the headers.

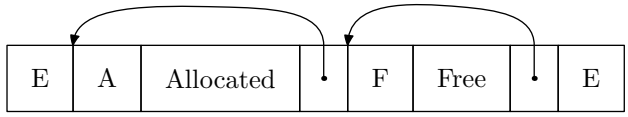

Figure 11-4: A New Arena Divided into Allocated (A) and Free (F) Blocks with Arena Markers (E)

11.4.2 Removing a Free Block from the Tree

In *dopoolalloc*( ), we make use of two functions that update the free block tree. These are *pooldel*( ) and *pooladd*( ). Because satisfying an allocation request implies that we must remove a free block from the tree, we examine *pooldel*( ) here and save *pooladd*( ) for the next section.

The easiest case is the one where the block we want to remove is part of a linked list but is not the head of that list. In determining that this is the case, we have to be careful to treat the root list differently. All other list heads point to their parent. For this case, we just remove it from the linked list and we're done.

```
  void pooldel(Pool *p, Bhdr *t)
  {
    Bhdr *s, *f, *rp, *q;
    if (t→parent ≡ nil ∧ p→root ≠ t) {
      t→prev→fwd = t→fwd;
      t→fwd→prev = t→prev;
```