CS 537
Lecture Notes Part 6
Memory Management

Previous Implementation of Processes
Next Paging
Contents

Contents

Allocating Main Memory

We first consider how to manage main ("core") memory (also called *random-access memory* (RAM)). In general, a memory manager provides two operations:

```
Address allocate(int size);
void deallocate(Address block);
```

The procedure `allocate` receives a request for a contiguous block of `size` bytes of memory and returns a pointer to such a block. The procedure `deallocate` releases the indicated block, returning it to the *free pool* for reuse. Sometimes a third procedure is also provided,

```
Address reallocate(Address block, int new_size);
```

which takes an allocated block and changes its size, either returning part of it to the free pool or extending it to a larger block. It may not always be possible to grow the block without copying it to a new location, so `reallocate` returns the new address of the block.

Memory allocators are used in a variety of situations. In Unix, each process has a *data segment*. There is a system call to make the data segment bigger, but no system call to make it smaller. Also, the system call is quite expensive. Therefore, there are library procedures (called `malloc`, `free`, and `realloc`) to manage this space. Only when `malloc` or `realloc` runs out of space is it necessary to make the system call. The C++ operators `new` and `delete` are just dressed-up versions of `malloc` and `free`. The Java operator `new` also uses `malloc`, and the Java runtime system calls `free` when an object is no found to be inaccessible during *garbage collection* (described below).

The operating system also uses a memory allocator to manage space used for OS data structures and given to "user" processes for their own use. As we saw before, there are several reasons why we might want multiple processes, such as serving multiple interactive users or controlling multiple devices. There is also a "selfish" reason why the OS wants to have multiple processes in memory at the same time: to keep the CPU busy. Suppose there are n processes in memory (this is called the *level of multiprogramming*) and each process is blocked (waiting for I/O) a fraction p of the time. In the best case, when they "take turns" being blocked, the CPU will be 100% busy provided $n(1-p) >= 1$. For example, if each process is ready 20% of the time, $p = 0.8$ and the CPU could be kept completely busy with five processes. Of course, real processes aren't so cooperative. In the worst case, they could all decide to block at the same time, in which case, the CPU *utilization* (fraction of the time the CPU is busy) would be only $1 - p$ (20% in our example). If each processes decides randomly and independently when to block, the chance that all n processes are blocked at the same time is only p^n, so CPU utilization is $1 - p^n$. Continuing our example in which $n = 5$ and $p = 0.8$, the expected utilization would be $1 - .8^5 = 1 - .32768 = 0.67232$. In other words, the CPU would be busy about 67% of the time on the average. [See also Tanenbaum, Section 4.1.3.]

Algorithms for Memory Management

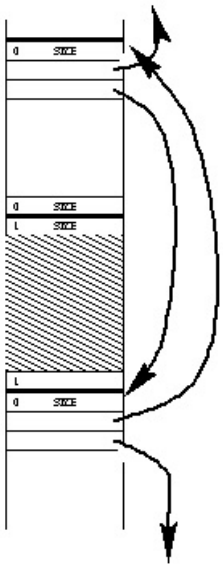
[Tanenbaum, Section 4.2.]

Clients of the memory manager keep track of allocated blocks (for now, we will not worry about what happens when a client "forgets" about a block). The memory manager needs to keep track of the "holes" between them. The most common data structure is doubly linked list of holes. This data structure is called the *free list*. This free list doesn't actually consume any space (other than the head and tail pointers), since the links between holes can be stored in the holes themselves (provided each hole is at least as large as two pointers. To satisfy an `allocate(n)` request, the memory manager finds a hole of size at least n and removes it from the list. If the hole is bigger than n

bytes, it can split off the tail of the hole, making a smaller hole, which it returns to the list. To satisfy a `deallocate` request, the memory manager turns the returned block into a “hole” data structure and inserts it into the free list. If the new hole is immediately preceded or followed by a hole, the holes can be *coalesced* into a bigger hole, as explained below.

How does the memory manager know how big the returned block is? The usual trick is to put a small *header* in the allocated block, containing the size of the block and perhaps some other information. The `allocate` routine returns a pointer to the body of the block, not the header, so the client doesn't need to know about it. The `deallocate` routine subtracts the header size from its argument to get the address of the header. The client thinks the block is a little smaller than it really is. So long as the client “colors inside the lines” there is no problem, but if the client has bugs and scribbles on the header, the memory manager can get completely confused. This is a frequent problem with malloc in Unix programs written in C or C++. The Java system uses a variety of runtime checks to prevent this kind of bug.

To make it easier to coalesce adjacent holes, the memory manager also adds a flag (called a “boundary tag”) to the beginning and end of each hole or allocated block, and it records the size of a hole at *both* ends of the hole.



When the block is deallocated, the memory manager adds the size of the block (which is stored in its header) to the address of the beginning of the block to find the address of the first word following the block. It looks at the tag there to see if the following space is a hole or another allocated block. If it is a hole, it is removed from the free list and merged with the block being freed, to make a bigger hole. Similarly, if the boundary tag preceding the block being freed indicates that the preceding space is a hole, we can find the start of that hole by subtracting its size from the address of the block being freed (that's why the size is stored at both ends), remove it from the free list, and merge it with the block being freed. Finally, we add the new hole back to the free list. Holes are kept in a doubly-linked list to make it easy to remove holes from the list when they are being coalesced with blocks being freed.

How does the memory manager choose a hole to respond to an `allocate` request? At first, it might seem that it should choose the smallest hole that is big enough to satisfy the request. This strategy is called *best fit*. It has two problems. First, it requires an expensive search of the entire free list to find the best hole (although fancier data structures can be used to speed up the search). More importantly, it leads to the creation of lots of little holes that are not big enough to satisfy any requests. This situation is called *fragmentation*, and is a problem for all memory-management strategies, although it is particularly bad for best-fit. One way to avoid making little holes is to give the client a bigger block than it asked for. For example, we might round all requests up to the next larger multiple of 64 bytes. That doesn't make the fragmentation go away, it just hides it. Unusable space in the form of holes is called *external fragmentation*, while unused space inside allocated blocks is called *internal fragmentation*.

Another strategy is *first fit*, which simply scans the free list until a large enough hole is found. Despite the name, first-fit is generally better than best-fit because it leads to less fragmentation. There is still one problem: Small holes tend to accumulate near the beginning of the free list, making the memory allocator search farther and farther each time. This problem is solved with *next fit*, which starts each search where the last one left off, wrapping around to the beginning when the end of the list is reached.

Yet another strategy is to maintain separate lists, each containing holes of a different size. This approach works well at the application level, when only a few different types of objects are created (although there might be lots of instances of each type). It can also be used in a more general setting by rounding all requests up to one of a few pre-determined choices. For example, the memory manager may round all requests up to the next power of two bytes (with a minimum of, say, 64) and then keep lists of holes of size 64, 128, 256, ..., etc. Assuming the largest request possible is 1 megabyte, this requires only 14 lists. This is the approach taken by most implementations of malloc. This approach eliminates external fragmentation entirely, but internal fragmentation may be as bad as 50% in the worst case (which occurs when all requests are one byte more than a power of two).

Another problem with this approach is how to coalesce neighboring holes. One possibility is not to try. The system is initialized by splitting memory up into a fixed set of holes (either all the same size or a variety of sizes). Each request is matched to an “appropriate” hole. If the request is smaller than the hole size, the entire hole is allocated to it anyhow. When the allocate block is released, it is simply returned to the appropriate free list. Most implementations of malloc use a variant of this approach (some implementations split holes, but most never coalesce them).

An interesting trick for coalescing holes with multiple free lists is the *buddy system*. Assume all blocks and holes have sizes which are powers of two (so requests are always rounded up to the next power of two) and each block or hole starts at an address that is an exact multiple of its size. Then each block has a “buddy” of the same size adjacent to it, such that combining a block of size 2^n with its buddy creates a properly aligned block of size 2^{n+1} For example, blocks of size 4 could start at addresses 0, 4, 8, 12, 16, 20, etc. The blocks at 0 and 4 are buddies; combining them gives a block at 0 of length 8. Similarly 8 and 12 are buddies, 16 and 20 are buddies, etc. The blocks at 4 and 8 are not buddies even though they are neighbors: Combining them would give a block of size 8 starting at address 4, which is not a multiple of 8. The address of a block's buddy can be easily calculated by flipping the nth bit from the right in the binary representation of the block's address. For example, the pairs of buddies (0,4), (8,12), (16,20) in binary are (00000,00100), (01000,01100), (10000,10100). In each case, the two addresses in the pair differ only in the third bit from the right. In short, you can find the address of the buddy of a block by taking the exclusive *or* of the address of the block with its size. To allocate a block of a given size, first round the size up to the next power of two and look on the list of blocks of that size. If that list is empty, split a block from the next higher list (if that list is empty, first add two blocks to it by splitting a block from the next higher list, and so on). When deallocating a block, first check to see whether the block's buddy is free. If so, combine the block with its buddy and add the resulting block to the next higher free list. As with allocations, deallocations can cascade to higher and higher lists.

Compaction and Garbage Collection

What do you do when you run out of memory? Any of these methods can fail because all the memory is allocated, or because there is too much fragmentation. Malloc, which is being used to allocate the data segment of a Unix process, just gives up and calls the (expensive) OS call to expand the data segment. A memory manager allocating real physical memory doesn't have that luxury. The allocation attempt simply fails. There are two ways of delaying this catastrophe, compaction and garbage collection.

Compaction attacks the problem of fragmentation by moving all the allocated blocks to one end of memory, thus combining all the holes. Aside from the obvious cost of all that copying, there is an important limitation to compaction: Any pointers to a block need to be updated when the block is moved. Unless it is possible to find all such pointers, compaction is not possible. Pointers can stored in the allocated blocks themselves as well as other places in the client of the memory manager. In some situations, pointers can point not only to the start of blocks but also into their bodies. For example, if a block contains executable code, a branch instruction might be a pointer to another location in the same block. Compaction is performed in three phases. First, the new location of each block is calculated to determine the distance the block will be moved. Then each pointer is updated by adding to it the amount that the block it is pointing (in)to will be moved. Finally, the data is actually moved. There are various clever tricks possible to combine these operations.

Garbage collection finds blocks of memory that are inaccessible and returns them to the free list. As with compaction, garbage collection normally assumes we find all pointers to blocks, both within the blocks themselves and “from the outside.” If that is not possible, we can still do “conservative” garbage collection in which every word in memory that contains a value that appears to be a pointer is treated as a pointer. The conservative approach may fail to collect blocks that are garbage, but it will never mistakenly collect accessible blocks. There are three main approaches to garbage collection: reference counting, mark-and-sweep, and generational algorithms.

Reference counting keeps in each block a count of the number of pointers to the block. When the count drops to zero, the block may be freed. This approach is only practical in situations where there is some “higher level” software to keep track of the counts (it's much too hard to do by hand), and even then, it will not detect cyclic structures of garbage: Consider a cycle of blocks, each of which is only pointed to by its predecessor in the cycle. Each block has a reference count of 1, but the entire cycle is garbage.

Mark-and-sweep works in two passes: First we mark all non-garbage blocks by doing a depth-first search starting with each pointer “from outside”:

```
void mark(Address b) {
    mark block b;
    for (each pointer p in block b) {
        if (the block pointed to by p is not marked)
            mark(p);
    }
}
```

The second pass sweeps through all blocks and returns the unmarked ones to the free list. The sweep pass usually also does compaction, as described above.

There are two problems with mark-and-sweep. First, the amount of work in the mark pass is proportional to the

amount of *non*-garbage. Thus if memory is nearly full, it will do a lot of work with very little payoff. Second, the mark phase does a lot of jumping around in memory, which is bad for virtual memory systems, as we will soon see.

The third approach to garbage collection is called *generational* collection. Memory is divided into *spaces*. When a space is chosen for garbage collection, all subsequent references to objects in that space cause the object to be copied to a new space. After a while, the old space either becomes empty and can be returned to the free list all at once, or at least it becomes so sparse that a mark-and-sweep garbage collection on it will be cheap. As an empirical fact, objects tend to be either short-lived or long-lived. In other words, an object that has survived for a while is likely to live a lot longer. By carefully choosing where to move objects when they are referenced, we can arrange to have some spaces filled only with long-lived objects, which are very unlikely to become garbage. We garbage-collect these spaces seldom if ever.

Swapping

[Tanenbaum, Section 4.2]

When all else fails, `allocate` simply fails. In the case of an application program, it may be adequate to simply print an error message and exit. An OS must be able recover more gracefully.

We motivated memory management by the desire to have many processes in memory at once. In a batch system, if the OS cannot allocate memory to start a new job, it can "recover" by simply delaying starting the job. If there is a queue of jobs waiting to be created, the OS might want to go down the list, looking for a smaller job that can be created right away. This approach maximizes utilization of memory, but can starve large jobs. The situation is analogous to short-term CPU scheduling, in which SJF gives optimal CPU utilization but can starve long bursts. The same trick works here: aging. As a job waits longer and longer, increase its priority, until its priority is so high that the OS refuses to skip over it looking for a more recently arrived but smaller job.

An alternative way of avoiding starvation is to use a memory-allocation scheme with fixed partitions (holes are not split or combined). Assuming no job is bigger than the biggest partition, there will be no starvation, provided that each time a partition is freed, we start the first job in line that is smaller than that partition. However, we have another choice analogous to the difference between first-fit and best fit. Of course we want to use the "best" hole for each job (the smallest free partition that is at least as big as the job), but suppose the next job in line is small and all the small partitions are currently in use. We might want to delay starting that job and look through the arrival queue for a job that better uses the partitions currently available. This policy re-introduces the possibility of starvation, which we can combat by aging, as above.

If a disk is available, we can also *swap* blocked jobs out to disk. When a job finishes, we first swap back jobs from disk before allowing new jobs to start. When a job is blocked (either because it wants to do I/O or because our short-term scheduling algorithm says to switch to another job), we have a choice of leaving it in memory or swapping it out. One way of looking at this scheme is that it increases the multiprogramming level (the number of jobs "in memory") at the cost of making it (*much*) more expensive to switch jobs. A variant of the MLFQ (multi-level feedback queues) CPU scheduling algorithm is particularly attractive for this situation. The queues are numbered from 0 up to some maximum. When a job becomes ready, it enters queue zero. The CPU scheduler always runs a job from the lowest-numbered non-empty queue (i.e., the priority is the negative of the queue number). It runs a job from queue i for a maximum of i quanta. If the job does not block or complete within that time limit, it is added to the next higher queue. This algorithm behaves like RR with short quanta in that short bursts get high priority, but does not incur the overhead of frequent swaps between jobs with long bursts. The number of swaps is limited to the logarithm of the burst size.

Previous Implementation of Processes
Next Paging
Contents

---

solomon@cs.wisc.edu
Tue Jan 16 14:33:41 CST 2007