

CSC 262

Memory Management

1 Memory Management

Multiprogramming (having more than one program running at the same time) is extremely convenient. And the safest/simplest way to implement multiprogramming is through the address-space construct in a process. Each process can act as though it has the entire memory of the machine all to itself, and it falls upon the hardware and the OS to ensure that this illusion is maintained. There are two interrelated issues:

1. Sharing physical memory between processes
2. Protecting process's address spaces

2 Sharing

With CPU sharing, the OS multiplexes processes across time. Time-slicing and timer-interrupts are the software and hardware mechanisms to accomplish time-division multiplexing on the CPU. In memory, the first step is to **partition** physical RAM between processes. For a process to run, it needs to be loaded into actual RAM, and because we want to have many processes running at the same time, we'll need to have many processes 'living' in main memory. It seems logical that the thing to do is to chop up physical memory into separate address spaces for each process.

The second step is to be able to **swap** out inactive partitions to make room for active ones. This is what the swap partition is for on UNIX systems. Most OSes 'backup' portions of physical memory onto the HD to make more room for running processes. So, when a process is inactive, its memory contents are written out to a swap file on disk to make more room for active processes. When an inactive process wakes up, the OS will need to read out the contents of the swap file back into main memory before the newly awakened process can run.

The third step is to be able to **protect** one process' memory from the actions of another process. Because main memory is shared between several running processes, physically, each process has access to other process' data. Because the OS is trying to maintain the illusion of separate address spaces, it must prevent a broken process running on the CPU from corrupting the memory of other processes waiting to be run.

- **Partitioning** Sharing physical memory between different processes.
- **Swapping** Using the same physical memory for different processes over time.
- **Protection** Preventing processes from reading or corrupting each others' data in main memory.

2.1 Fixed Partitions

The first, and simplest form of partitioning is to just chop main memory up into a series of fixed-size chunks. Each process gets its own chunk to run in. So, if there are 8 processes running on a system with 128MB of ram, each process gets 16MB of address space to play around in. This scheme has the advantage of simplicity, but several disadvantages:

- Limits the number of concurrent processes (determined by $\frac{RAMsize}{partitionsize}$).
- Causes internal fragmentation.

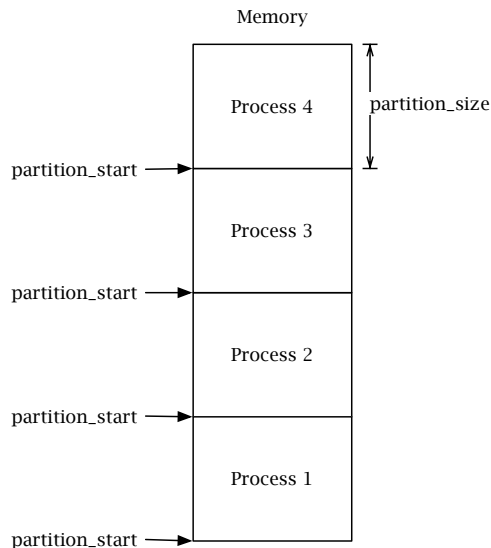


Figure 1: A four-way partitioning of main memory

How would this actually work? Remember that user code is generally running on the processor directly (although virtualization may eventually change this). The OS can't possibly intervene with code at every memory access. Remember that each new instruction fetched is a memory access, and the processor will have to perform loads and stores for each chunk of data not currently in the registers. Obviously, having to context switch every few instructions is just too costly. The answer to these kinds of problems has historically been: let the hardware do it. Of course, the hardware guys have their own concerns, so the hardware solution itself must also be fairly cheap and fast. For fixed partitions, the hardware support is fairly minimal. All you would need is one global register, and one register per process. The global register would hold the size of the fixed partitions (lets call this one *partition_size*, which would be manipulated by the OS). Each process would then have a register loaded with the start of its partition (lets call that one *partition_start*). With these two new registers, the hardware could then check for accesses outside of a partition thus (assuming that *addr* is the address the process is trying to load or store):

```

if((addr >= (partition_start + partition_size)) ||
    (addr < partition_start)){
    //addr not ok, trap to OS
    illegal_access_trap(addr);
}else{
    //addr ok
}

```

So, the hardware just has to do one addition and two compares. Since the code is unchanging, these functions can actually be hardwired in, making them even faster. Even in the days of yore, when memory was almost as fast as the CPU, the cycle or two extra overhead per memory access was an acceptable price to pay for memory protection.

Hardware support can make memory protection feasible, but it doesn't address fragmentation. The following picture illustrates the fragmentation problem with fixed-size partitions:

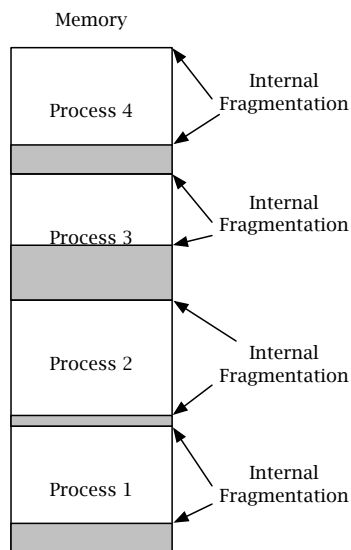


Figure 2: Internal Fragmentation, the bugbear of fixed-size partitioning

If the partitions are too small, then the process can never be run. If the partitions are too big, then the leftover space is basically wasted (internal fragmentation). It cannot be allocated to another process (that would interfere with memory protection). So fixed-size partitioning methods must be conservative. Because most user programs have no easy way of determining exactly how much memory they'll need before they're actually running, the OS will just have to overcommit memory (to be on the safe side). This waste just makes memory-management people cringe. So, what do you do when fixed-size partitions waste space? You make the partitions' size variable, of course.

2.2 Segments/Variable-sized Partitions

Instead of just arbitrarily chopping up memory into a few overly large partitions, segmentation allows for partitions to have variable size. Therefore, a partition can start small and grow as the process requires. Simple, right? No overcommitment of space means no internal fragmentation, which means no space waste! However, the memory protection scheme used above will fail. Since each segment can have its own size, *partition_size* can no longer be a global variable. That means that each process will have to have a segment size register as well as a segment start register. Then code to implement memory protection (per-address, per process) would resemble:

```
if((addr < segment_start) ||
    (addr >= (segment_start + segment_size))) {
    //error, trap to OS
    illegal_access_trap(addr);
} else {
    //addr ok
}
```

Ok, that means we just need two registers for doing HW memory protection with a segmented system. Additionally, we can grow and shrink segments as needed. Therefore, no internal fragmentation.

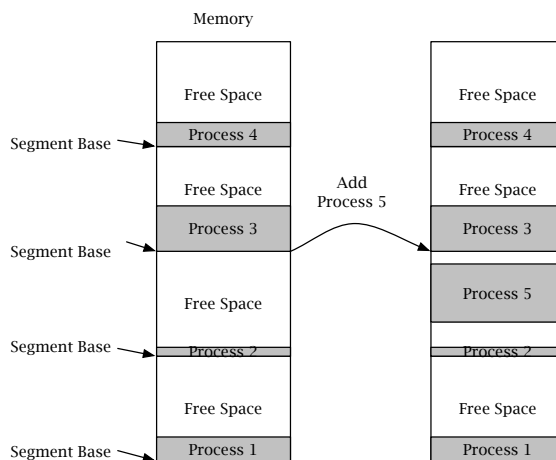


Figure 3: A segmented system

As long as the OS keeps track of the free memory in the system, as long as there's a gap large enough to accommodate a new process, we can place it in memory. This, however, leads to another problem. After an OS has been running for a while, and created and destroyed many processes, memory can become littered with active process segments. These segments are most likely non-contiguous, so there's going to be free space between most segments. This is a situation known

as external fragmentation, because memory becomes chopped up into difficult to use fragments outside of segments. The following figure illustrates this situation:

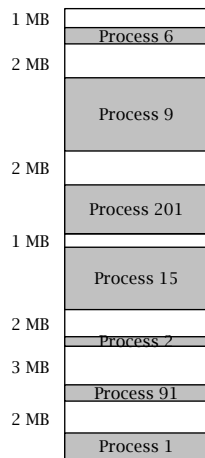


Figure 4: External Fragmentation

In this figure, there are 7 active processes in memory. Each of the segments is separated by some free space. In this figure, there are 13MB free although this space is scattered all over main memory. Although there's no difficulty when launching a process needing 1MB of memory (or 2 or 3). What happens when a new process is launched that needs 4MB of memory? Remember, a segment must be contiguous, so the system has to try and find a single solid chunk of 4MB of memory. There isn't any, therefore, most of the 13MB of free memory is effectively unusable because it is not contiguous. These problems can be offset by relocation and swapping, but to make sense of them we must first talk a bit about how code is organized in memory.

2.3 Code Organization and Loading

Up to this point, we've been dealing with programs that use raw, physical addresses; e.g. when the program uses the address `0xDEADBEEF` it means read/write a value to address `0xDEADBEEF` in main memory. This makes sense, and it's the simplest scheme for locating code in memory. However, there are problems. For example, if the addresses are hard-coded into the instructions themselves, that means that that code must always be loaded at exactly the same address in memory. This means that if any other code just happens to use that address range, then the two programs cannot be run simultaneously. Bad idea.

In reality, code is written (or generated by a compiler) to be relocatable. What this means is that the addresses used by the code are either offsets, or stored in a register/memory location. Offsets are good, because the value of an offset doesn't change if you change the physical location of the code in memory. Registers/Memory locations are good, because then the OS can load these values for a process. Then, the process will be addressing things through its address space

registers/locations. If all process code is guaranteed to be written in a relocatable way, then the OS will be able to move processes around in main memory.

Consider if the code from the previous section was relocatable. Then the external fragmentation problem could be solved by suspending and moving some processes around. For example, process 91 can be suspended, and its address space copied (by the OS) byte-by-byte into the space right after process 1. This would then merge the 3MB and 2MB fragments into one 5MB fragment (this is known as coalescing two fragments).

This solves the basic external fragmentation problem, but at some cost. To relocate a segment re-

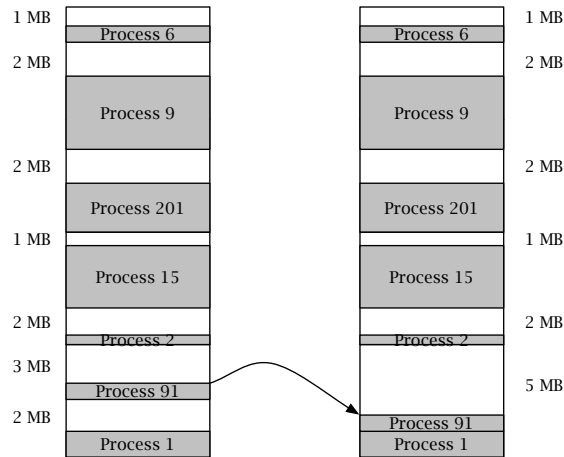


Figure 5: Fragment Coalescing

quires copying the entire segment (which is slow). And, in a heavily fragmented system allocations are highly likely to force relocations. A perfect solution would have no fragmentation and would require only relocating as much of another process' address space as is needed to accommodate a new allocation (when memory was full).

Virtual Addresses: A chunk of relocatable code is relocatable because it no longer depends on being loaded at a specific physical address. All relocatable code needs is a way of mapping a logical, program-local address into a physical address. In the case of an offset, the program just adds (or subtracts) the offset from the current address to derive the physical address of the code/data. The offset, in this case is an example of a *virtual address*. This is an address that means something to the program, but to preserve protection and relocation, may need to be translated into a physical address. Consider segments. The most direct way of doing virtual addresses with segmentation is to treat each program address as an offset from the segment base. This is known as $base + offset$ addressing. So, to the program, the address 0 refers to the first byte in the program's address space. But if the OS happened to load the program at address $0xCAFÉ0000$, then the program's address 0 would be translated to $0xCAFÉ0000 + 0 = 0xCAFÉ0000$. Systems that had segmented memory (like the 80286), would have segment registers that the OS would load (per process). The segment registers would contain the base address for the segment. Therefore any address requested by user code would first be added to the value in the segment register to

generate a physical address.

For fixed size partitions, virtual addresses are slightly different. Because the size of the partition is known, then an address can be chopped into two pieces. The high-order bits (most-significant) will refer to the partition and the low-order bits will be an offset in that partition. For example, assume that we have 8MB fixed partitions on a machine with 32-bit addresses. $8\text{MB} = 2^3 * 2^{20} = 2^{23}$ bytes. Therefore the low 23 bits of an address will be some byte offset within an 8MB partition. The upper 9 bits will then specify which partition. That means that there's at most 512 partitions. Lets look at a simple example. If the virtual address is `0xFF00BEEF`, then the upper 9 bits are `111111110`, which is 510. The lower 23 bits are `0x00BEEF`, which specifies byte number 48879 in the partition. Therefore, if we directly map partitions to main memory (and we have a full 4GB to work with), this address maps to byte 48879 in partition 510.

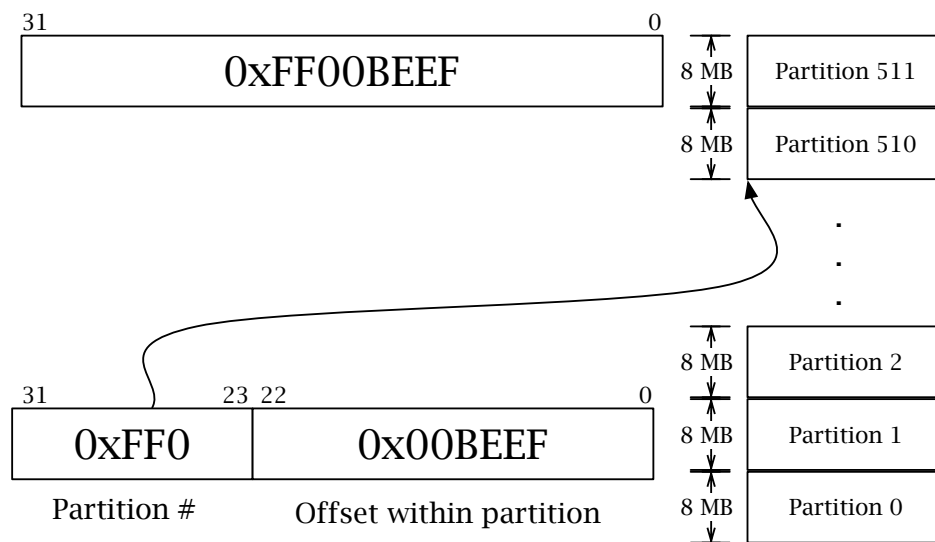


Figure 6: A Virtual Address for a fixed partition

Multiple Segments: You may have noticed that I said that segmented processors had multiple segment registers. In real segmented systems, a process will generally have multiple segments associated with its address space. One for the stack, one for the heap, one for the static data, and one for the code (there may be more). In the bad old days of computing, when instruction sets were huge and far too many people wrote entire applications in assembly; the instruction itself would indicate which segment to index off of (e.g. `push` used the stack segment registers (`esp`, `ebp` on intel)).

2.4 Allocation Strategies

Relocation makes it possible to move a process' memory around, but the OS still has to find free space to place the memory in the first place. The choice of an allocator is a major design issue. A good allocator will avoid fragmentation and make coalescing easy. A bad allocator can impact performance by poorly distributing memory and forcing expensive relocations. Given several processes in main memory and spaces between them, the first obvious strategy would be to just grab the first available chunk that's large enough to accommodate the process you're allocating. This is called **first fit**. In practice, it isn't actually as horrible as you may suspect. By grabbing the first suitable chunk, first fit minimizes search time, however small holes tend to accumulate near the start of the search.

Best fit: This is the most obvious "optimal" solution. Because fragmentation is bad, trying to waste as little space as possible makes sense. Best fit is a strategy that attempts to find the smallest space that is large enough. Of course, this may require searching through all the available spaces. In fact, unless there's a free chunk exactly the right size, best fit will inspect all available spaces. Long search times are bad, generally speaking. Additionally, best fit tends to result in lots of tiny fragments scattered all over.

Worst fit: If scattered tiny fragments are bad, then maybe the OS should allocate processes in the largest available space. This would leave behind the largest possible fragment. Of course, this has the same problem as best fit, in that you may have to inspect all spaces before you can decide which is the worst. In practice, this seems to result in more fragmentation than best fit.

Next fit: This is a heuristic improvement to first fit. First fit tends to cluster teeny fragments near the start of memory. This is often because first fit always starts at the beginning of memory. With next fit, the idea is to do first fit, but start from the last place you allocated. That way you evenly distribute fragments around main memory.

-Do an example on the board

Buddy Memory Allocation (aka the buddy system): An interesting memory allocation technique that attempts to minimize both search time and fragmentation is the buddy system. In the buddy system, memory is allocated in powers of two. This allows for a kind of binary search over free space that reduces search time. When using buddy allocation, there is a maximum allocatable size (which is a power of two) and a minimum size (to limit overhead, also a power of two, usually on the order of a few K). Buddy allocation works by basically stepping down the power of two needed to allocate an object until the smallest chunk that is large enough to fit the request is found. It works this way:

- 1) Find the minimal power of two needed (e.g. 64k for a 34k request)
- 2) If such a slot exists, allocate it.
- 3) If not, try to create such a slot,
- 4) Split a larger memory slot into two halves
- 5) If one of the halves is the right size, allocate it, otherwise go back to 1)

When freeing space, the buddy system checks to see if there's a neighboring free chunk with the same size. If so, it merges the two chunks together into one free chunk of twice the size (and then checks that chunk's neighbors). Although splitting free chunks in half may seem somewhat

arbitrary, it means that the buddy system can be backed by a binary tree. And that means that it takes at most $\log(n)$ inspections to find a suitable free chunk. The following diagram illustrates the steps of the buddy system when handling a request to allocate 70K, then 200K, then free the 70K allocation.

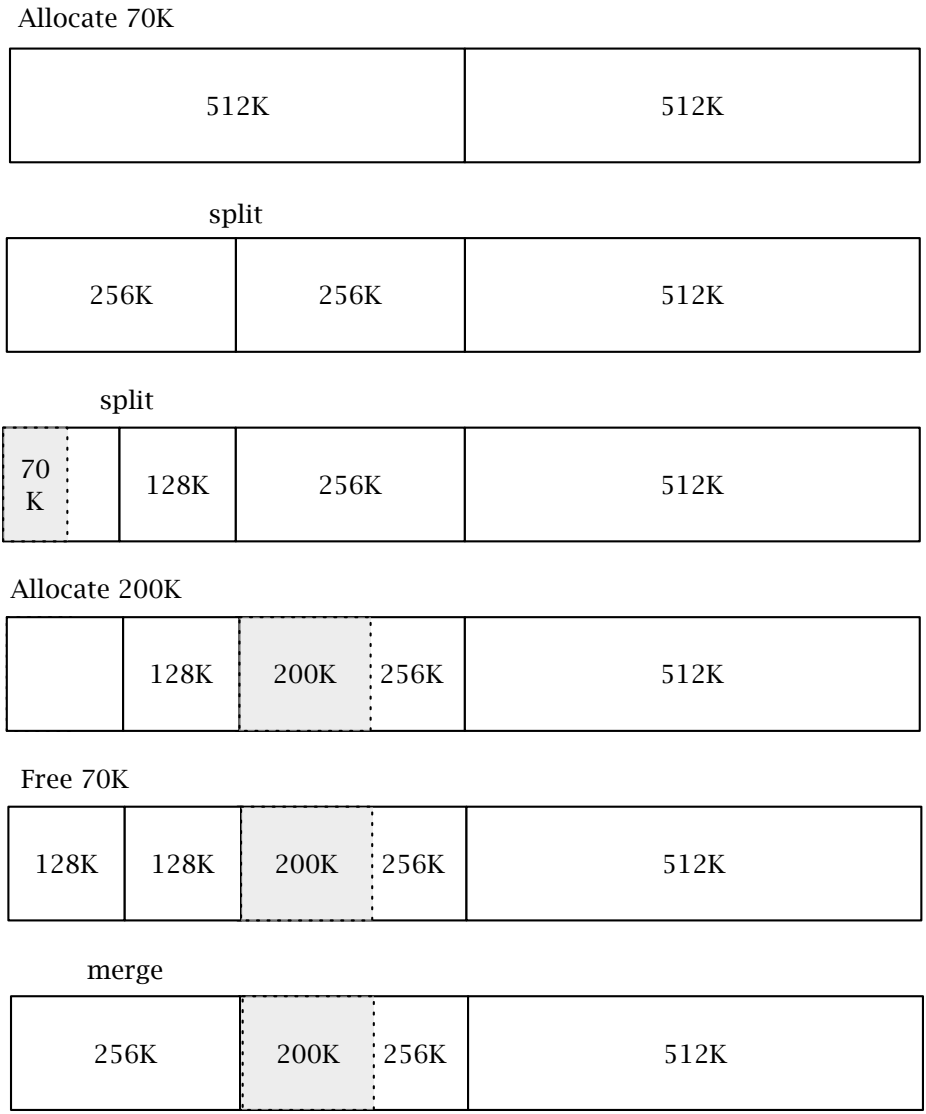


Figure 7: The buddy system in action

2.5 Swapping

If you have relocatable processes, then they can be moved around by the OS (when suspended, of course). Moving around segments to coalesce free memory is one use of relocation, but what happens when you just don't have enough free memory? **Swapping** is a technique for freeing up main memory without having to force kill a process. The OS takes a non-running process, and writes its address space out to disk (usually to a dedicated swap partition or swap file). Then, the memory that was formerly occupied by that process is effectively free for a new process. Then, later on, when you want to run the swapped process, you just read it back in from disk.

Swapping removes the limitations on the number of processes in a system, and it allows the OS to allocate as much physical memory as needed to a process (up to the actual amount of memory in the machine, that is). If a process needs more, the OS can just swap out another (suspended) process. Swapping introduces two new process states into the old 3 state system:

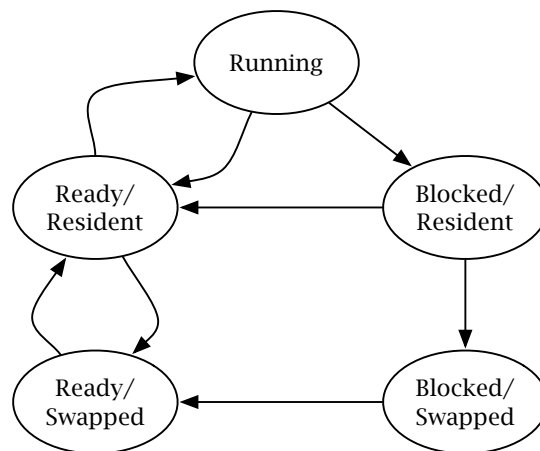


Figure 8: Process States, with swapping supported

In this diagram, resident means that a process is in memory and swapped means that its been swapped out to disk. So, a running process must be in main memory. But the OS now has the option to swap out a blocked process or a ready process. Once a blocked process is swapped out, it doesn't need to be loaded into memory again until it becomes unblocked (hence no edge back to the blocked/resident state).

Of course, there is one giant issue here. Disks are MUCH MUCH slower than main memory. About a million times slower, actually (memory operates in the 10's of nanoseconds and disks operate in the 10's of milliseconds). So, swapping looks like it might greatly slow down a system. Imagine the following case, a system with 256MB of memory, running two processes. Process 1 uses 200MB of memory and Process 2 uses 200MB of memory. Obviously neither process 1 nor process 2 can be entirely in memory all at once. Looking online, a perfectly decent SATA harddrive can support a transfer rate of 3 billion bits per second and a seek time of 4.2 ms. Perfectly

decent DDR2 memory operates with a latency of 3.75ns. To make the analysis easier, let's increase memory latency to 4.2ns (making it exactly one million times faster than disk).

Using the disk stats, we can calculate that it would take at least 9.4ms to write out or read in a 200MB address space from disk. Let's assume that the OS is switching running processes every 100ms (to increase responsiveness). And let's assume that both process 1 and process 2 would normally take 500ms to execute (no I/O). In this case (assuming that we started with process 1 in memory), the OS would execute process 1 for 100ms. Then switch to process 2, which is swapped out, so the OS would have to swap out process 1 (9.4ms) and then swap in process 2 (9.4ms). Then process 2 would execute for 100ms, at the end of which the OS would have to swap out process 2 and swap in process 1. This continues until both processes are finished, resulting in an 159.8ms overhead for swapping (which is roughly 14% of the total running time). Remember that this is also the most generous numerical evaluation of the disk/memory difference. In reality, swaps can take hundreds of milliseconds each. Real OSes use swapping to keep memory pressure down, but they are very tricky about how and which parts of memory get swapped.