

1 Avoiding Deadlock

Deadlock avoidance is the middle path. Rather than attempting to totally prevent deadlock in the first place, the threading system makes it much less likely. This is accomplished by relaxing the prevention techniques so that they don't impact programmer convenience or efficiency too much. Deadlock may still occur, but it is now much more unlikely. This means that an avoidance system will still occasionally have to detect deadlock, or rely on programmers to write deadlock-free code.

1.1 2PL and Transactions

An example of a relaxed prevention technique is **2 Phase Locking** (often called 2PL). This technique is particularly popular in databases and transactional systems. In 2PL a thread/transaction can still grab resources 1 at a time (partial allocation), but resources can only be released when all needed resources are acquired. This is why it's called two-phase locking: the first phase is when the thread grabs all the locks it needs. Then the thread processes for a bit. The second phase is when the thread releases all of its held resources (usually in the reverse order it grabbed them). In practice, the first phase is usually lock-grabbing interleaved with some computation (while the thread figures out which lock to grab next), and the second phase happens all in a big chunk at the end. 2PL does not absolutely prevent deadlock, but it establishes an order to acquiring and releasing resources that avoids many common deadlock scenarios. In DB systems, transactions are run in a 2PL fashion, but because deadlock can still happen the DB performs occasional deadlock checks.

This brings us to the second example of a deadlock avoidance system, which is the **transaction**. Transactions are the lifeblood of a database, and they have recently become a hot topic in programming language research. A transaction is an abstraction of an atomic action. A transaction allows the programmer to lump up a bunch of operations and tell the DB/runtime system to execute all these actions atomically. The transaction processing system, then has to ensure that the transactions behave atomically (i.e. either the transaction happens once or not at all). The system also wants to enhance concurrency as much as possible, so its going to try to run as many transactions at once as possible. In general, transactions are run "optimistically". What this means is that the system runs transactions until there's a problem (the technical term is **conflict**), and then resolve the problem.

Transactions are usually run in a 2PL fashion, and thus deadlock can be detected when transactions attempt to grab a lock. When the DB/runtime detects the conflict it needs to pick a transaction to sacrifice. That transaction is suspended and restarted (it releases all its resources). This will let the other transactions proceed. How do you restart a transaction? Well, the DB or runtime system has secretly been logging all of the changes to memory/DB tables that the transaction has been making. When the transaction is **aborted** (stopped and restarted), the DB/runtime can then run

this log backwards in time to restore the previous values. Note that this is ok, because the transaction had to hold a lock on that word/table anyway to change it, so no other transaction could see those values change and then change back.

2 The Banker's Algorithm

Another approach is the banker's algorithm. This was developed by Dijkstra to solve resource allocation deadlock within an OS. It is called the banker's algorithm because the OS in this case acts like the bank, and the processes act like depositors and lenders. The OS (as a bank) has a set of assets, which are the resources. Each type of resource is basically like a separate account. Each process has a credit limit, which is the maximum amount of resources that it can claim. Each process also has a current balance, which is the amount of resources actually claimed. In this case, the OS acts like a hyper-conservative banker. Because the OS would like all processes to terminate, a given resource will be lent out only if it couldn't possibly prevent another process from finishing (from allocating its credit limit). Consider the following example with 3 resources (files, network connections, and memory) and 3 processes (A, B, and C) the amounts are listed as allocated(limit):

	Files	Network Con.	Memory
Initial amounts:	256	256	128 Kpages
Proc. A	128 (130)	200 (256)	16 (32)
Proc. B	8 (10)	0 (0)	48 (64)
Proc. C	50 (64)	0 (56)	8 (32)

The idea here is to keep the system in "safe" states. A state is safe if it is still possible for all processes to terminate. Since we have no foreknowledge of when the process will actually terminate, we have to make the pessimistic assumption that it may allocate up to its credit limit before releasing any resources. The state above is safe because it is possible for any one of the processes to complete. Consider the two scenarios:

1. Process A requests 16 pages. This is safe, there is still enough memory around to satisfy B and C's maximum request size.
2. Process C requests 1 network connection. This is unsafe, because it means that process A will not be able to request its maximum amount. Therefore this request is denied.

Although the banker's algorithm was used on the THE OS, it isn't practical for most situations. First, it is usually impossible for a programmer to actually determine the maximum number of resources needed (unless it's something trivial like 0), so it seems likely that the average programmer would just overcommit the program's limit. This would reduce concurrency, and as such is unacceptable. Second, this system assumes a fairly static view of processes. However, we know that processes are created and destroyed dynamically, which complicates things greatly for the banker's algorithm.

3 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

4 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.

4.1 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.

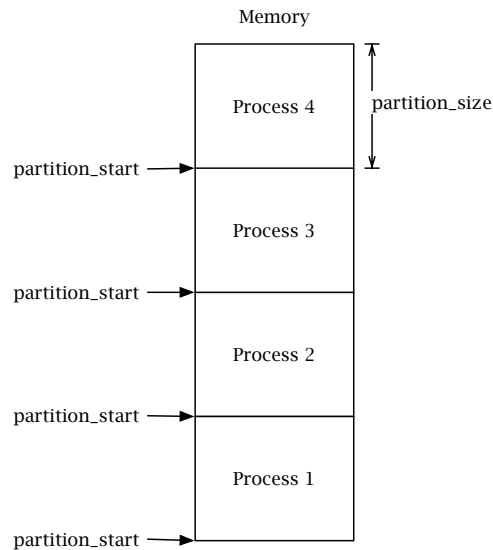


Figure 1: A four-way partitioning of main memory

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:

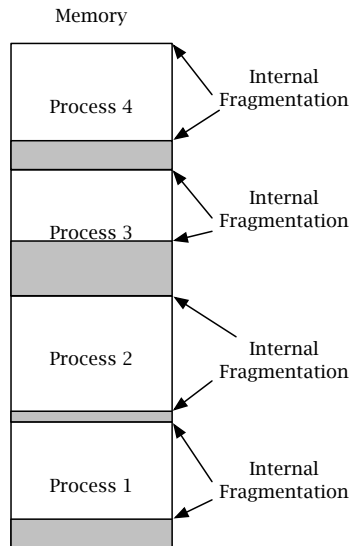


Figure 2: Internal Fragmentation, the bugbear of fixed-size partitioning

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.

4.2 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.

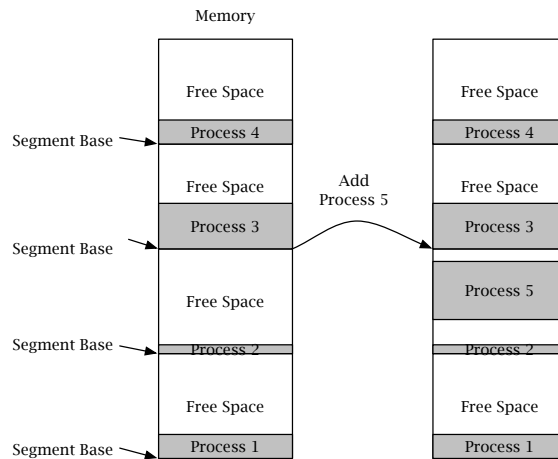


Figure 3: A segmented system

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:

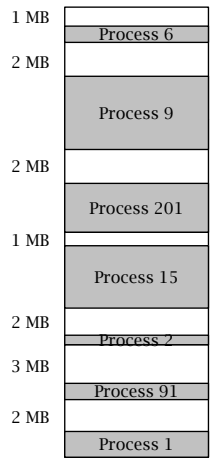


Figure 4: External Fragmentation

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.