Operating Systems

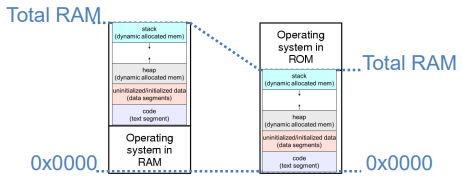
Memory Management

Goals

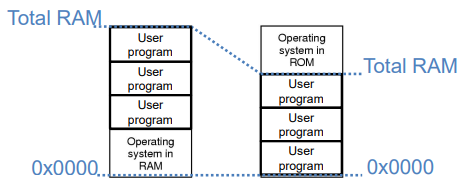
The goal of memory management is to allocate memory resources among processes and the OS maximising memory utilisation and system throughput. It also provides convenient abstractions for processes and OS programmers simplifying utilisation and addressability. Finally it also provides isolation between processes and the OS addressability and protection orthogonal problems.

Background

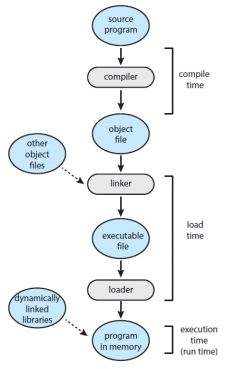
A program must be brought (from disk) into memory and placed within a process context to be run. Main memory and CPU registers are the only storage CPU can access directly (e.g. with CPU instruction). The memory unit only sees Physical memory address + read request or Physical memory address + data and write requests. Registers are generally accessible in one CPU clock cycle while main memory takes multiple CPU cycles causing the CPU to stall, cache which sits between main memory and the CPU registers can reduce the number of CPU cycles needed to access memory and is transparent to the programmer.

One Program With No Memory Abstraction

The program can see (access) the physical memory, the physical memory is shared with the OS (and even the BIOS). This can result in the program being able to mess with the OS (and BIOS).

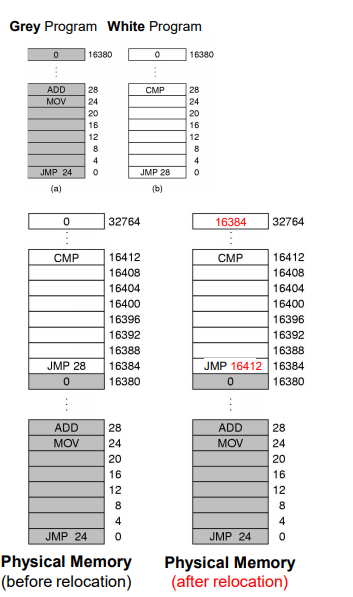
Multiple Program With No Memory Abstraction

Every program sees the physical memory and can access each other’s memory with the total number of programs depending on their size and the size of the memory. These will again share memory with the OS and BIOS again allowing them to mess with them.

Binding

Addresses in a program are usually symbolic (think variables), binding is the process of mapping these symbols to actual memory locations.

This binding can be done at different points by different systems:

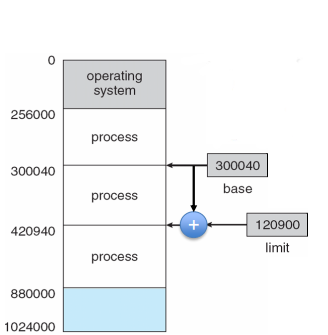
* Compiler (source to object file), this binds the symbols to relocatable addresses at compile time (e.g. base + offset)
* Linker (object file to executable file), This binds the relocatable addresses to absolute addresses if the final memory location is known and these addresses cannot be later moved without hardware support.
* Loader (Executable file image moved to memory), the relocatable addresses are bound to absolute addresses again, after the final memory location is decided they again cannot be moved without hardware support.
* Execution

Using Relocation

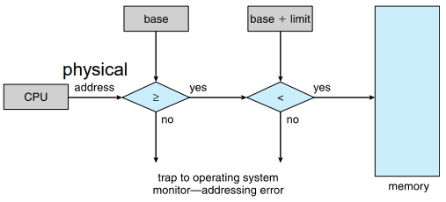
Programs must be relocatable, written to be placed and run at any memory address. This may require extra information in the executable. The loader can then decide where to place them based on the available physical memory, the loader is what performs relocation (this will increase the load time though).

Multiple Programs on Physical Memory

All physical memory will be exposed to all the processes meaning user processes may interfere with the OS and each other, processes may access the OS’s and each others ‘secrets’ and programs must be relocated when loaded.

This provides no protection AND relocation is expensive.

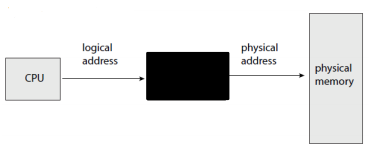
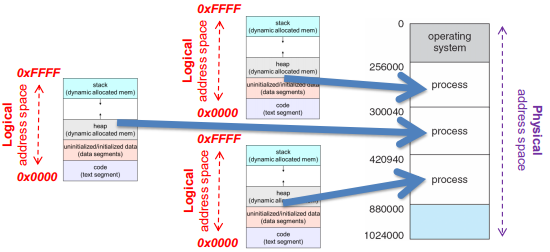
Protection: Base and Limit Registers

Base and limit registers define the valid addresses for a process. The CPU checks every memory access generated by the process to ensure it lies in the range defined by the base and limit, if the address is not valid (outside the range) it traps to the OS.

Memory Abstraction: Address Space

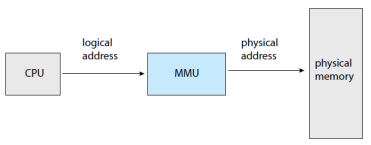
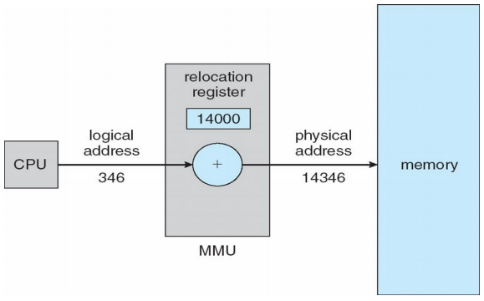
Address spaces are an abstraction from physical memory space. An address space is a set of memory addresses that a process can use, independently from other processes. Note that these addresses don’t have to be physical memory addresses.

Logical Addresses

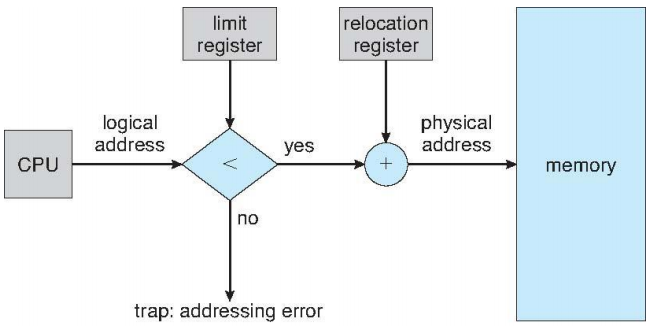
To make it easier to manage the memory of multiple processes we make processes use logical memory addresses. Logical addresses are independent of physical addresses with the data actually living in physical addresses with it’s exact location managed by the OS. Logical addresses will be translated by hardware into physical addresses with the OS configuring this translation.

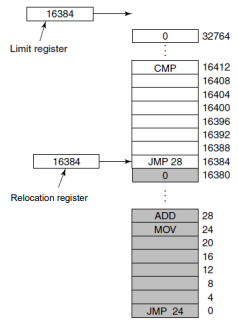
The set of logical addresses a process can reference is its address space. A program issues addresses in a logical address space which are then trnaslated to the physical address space. You can think of the program as having a continguous logical address space starting at 0 and a continguous physical address space that starts somwhere else.

Memory-Management Unit (MMU)

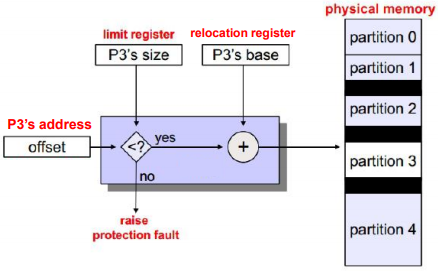
The MMU is the hardware component that translates the CPU generated addresses to physical addresses. This ensures that programs never actually see the physical addresses. There are many implementations of MMUs (Relocation+limit registers, segmentation, paging, etc…) and many names (MMU, MTU, MPU, etc…).

As an example an MMU as a relocation register will provide no protection:

And an MMU as a relocation register AND a limit register would provide some protection:



Multiple Programs: Relocation and Limit Registers

Relocation and limit registers requires hardware support (the actual registers) but we don’t need the programs be be relocatable (this is taken care of) meaning we can use a simple loader and have faster load times. We can hav protection by giving each process its own privtate adress space using these base and limit registers.

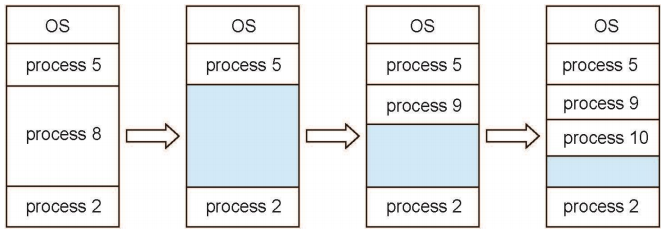
We can also have variable size partitions as the limit register determin the partition sizes and can hold whatever value (size) we like.

Contiguous Allocation

Main memory must support both OS and user processes but as memory is limited it must be allocated efficiently. Contingous allocation is an early method of allocation.

Main memory was split into usually two partitions one for the OS held (and interrupt vector) in low memory and the other for processes held in high memory (each contained in a single contiguous section of memory. Relocation and limit registers are used to protect the user process from each other and from changing the OS code and data.

Multiple-Parition Allocation

In-order to have mutliple programms running we will need to split the partition into further partitions (for each program). For efficiency these partitions will need to be variable in size (so as not to waste memory) and the degree of muiltprogramming will be limited by the number of partitions we can have.

A hole is a block of available memory, these can be of various sizes. When a process ‘arrives’ the OS allocates it memory from a hole large enough to accommodate it. When a process exits it returns its parition to the OS and adjacent free partitions are combined to make one large hole again. This means the OS needs to maintain infomration about allocated partitions and holes (free partitions).

Dynamic Storage-Allocation Problem

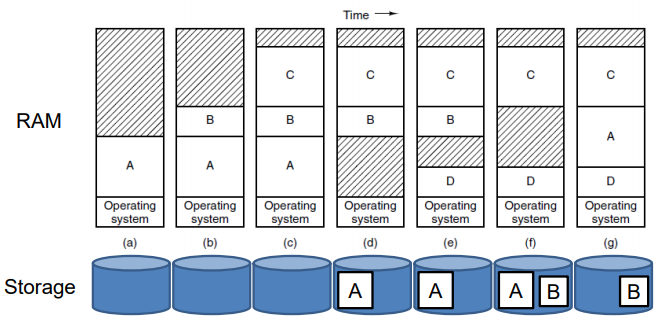
How do we satisfy a request of size n from a list of free holes? There are many ways to do so but here are three basic ones:

* First-fit, allocate the first hole that is big enough for the n
* Best-fit, allocate the smallest hole that is big enough for n (this must search the entire list, unless the list is ordered by size in which case it’s the same as first-fit), this will produce the smallest leftover holes (memory from a chosen hole that wasn’t allocated for the process, this sits between processes)
* Worst-fit, allocate the largest hole (again must search the entire list), this produces the largest leftover hole (which can also be good as we may be able to fit a new process in large left over holes).

First-fit and best-fit are better than worst-fit in terms of speed and storage utilisation.

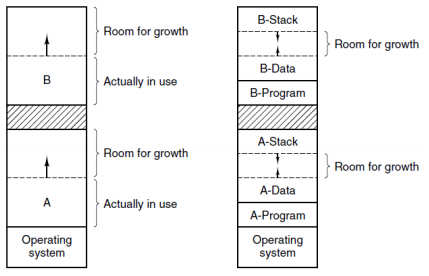
Multiple Programs Swapping

When we can’t fit all the programs we want in memory we run into problems. To solve this we use swapping, in swapping there are three basic actions:

1. Swap in, bring an entire process from disk into memory
2. Run, run a process for a while
3. Swap out, take a process in memory and place it, in its entirety, on the disk

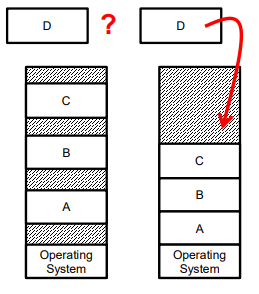
Essentially idle process are stored on disk until they stop idling so they don’t take up memory. Note that this is for running process, not programs.

Multiple Programs: Growing Programs

So far we’ve assumed programs initially ask for the most amount of memory they will use, this isn’t the case in practice however. In reality the memory needed for a process will grow and shrink during its running, we will need a solution to deal with this, there are two simple cases for this:

1. Allocate the new space needed at the end of the physical address space, relocating if no space is available
2. Allocate for a growth inside the current address space (this does imply a maximum size though)

Neither solution is ideal.

Multiple Programs: Memory Fragmentation

Fragmentation occurs due to the constant creation, destruction and swapping of processes, this creates holes between the process that are too small on their own to be used for any new processes but may be able to fit a new process if they were contiguous.

Compaction is the act of relocating all of this memory to be right up against each other either by copying the data or swapping in and out the process. However compaction is computationally expensive and gets in the way of growing programs, what if a process’ stack or heap needs to grow?

There are two main types of fragmentation, external fragmentation where when a process is created it is allocated the exact amout of memory requested but the result is the gaps between process; and internal fragmentation where you allocate more than is requested (the space left over in the last allocated block) so this unused memory is trapped inside processes.

A first-fit analysis reveals that given N blocks of memory to be allocated, 0.5N blocks will be lost to fragmentation.