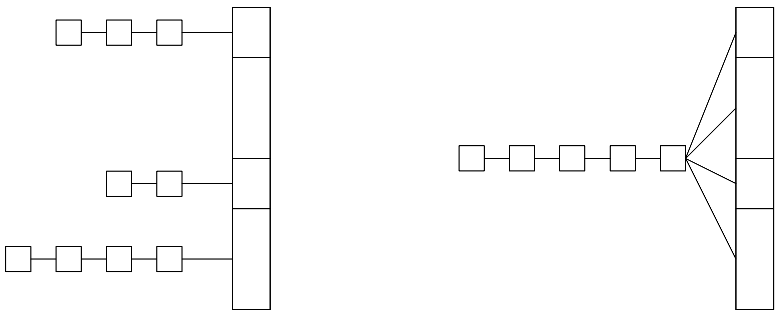
Memory Allocation and Virtual Memory

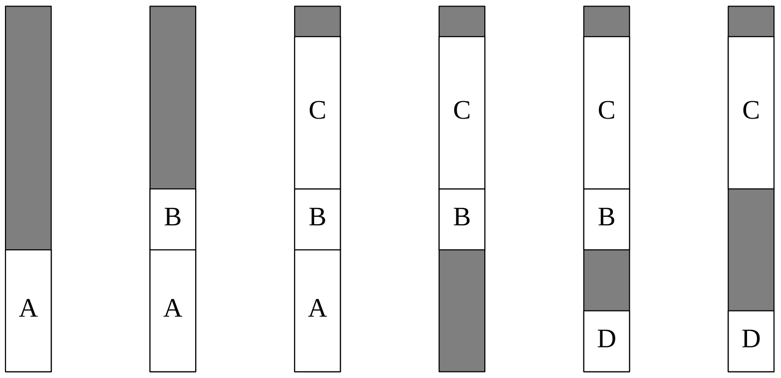
The simplest memory allocation is to have several fixed memory partitions and allocate a process to each one. Different sized partitions can be allocated to accomodate different processes memory needs.



Unless the workload is very well understood, fixed pertitions are seldom used. There is generally wasted space. Variable partitions are more common.

Variable Partitions

Note that algorithms for variable partition allocation are also effective for non-OS allocators. Even systems that use other allocation systems in the OS may use variable size allocators inside processes. (An example is the malloc library call.)



Variable partitions change the wasted space problem. Rather than picking one set of partition sizes and hoping that processes use them wisely, we have to solve the problem of allocating memory well. The goal is to avoid *compaction*, the wholesale reorganization of memory.

At this point we introduce the idea of *swapping.* Swapping is moving a running process from memory to secondary storage to allow another higher process to use its space. Systems may swap processes for a variety of reasons, from higher priority processes becoming available, to making system-wide usage decisions. For example, large batch jobs may run freely when the system is under-utilized, but be swapped out when a lot of interactive jobs are being submitted.

Both fixed and variable partition systems (and paged VM or segmentation systems) can swap. The same variable memory allocation algorithms we discuss here can be used to allocate swap space.

Data Structures

Free memory has to be tracked to be allocated to processes. We can keep track of either holes (unallocated regions of memory) or allocations. Whatever memory isn't in one set is in the other. Some portion of memory is set aside as allocable to processes, and data structures in the address space permanently allocated to the OS are used to keep track of it. used:

Bitmaps

One way to keep track of allocable memory is to divide it into fixed size allocation units and use a bitmap to keep track of the allocated and unallocated areas. This can be slow to search for the next hole - bit operations are slow. This is still sometimes used (nachos uses this to track the used physical memory).

Linked Lists:

A linked list is created of all the allocated memory and holes. Notice that this allows true variable sized partitions.

When a new process is created (or allocated memory) a hole shrinks (or disappears) and a new allocated node appears. On process termination (or return of memory) either a new hole is created, or a hole enlarged, or two holes merge.

There are a couple ways to optimize linked lists. The PCB can point to a processes current allocation, to aid in finding the allocations to release. Double linking the lists can speed merging. Allocating the list nodes in holes can reduce the memory required.

Allocation Algorithms

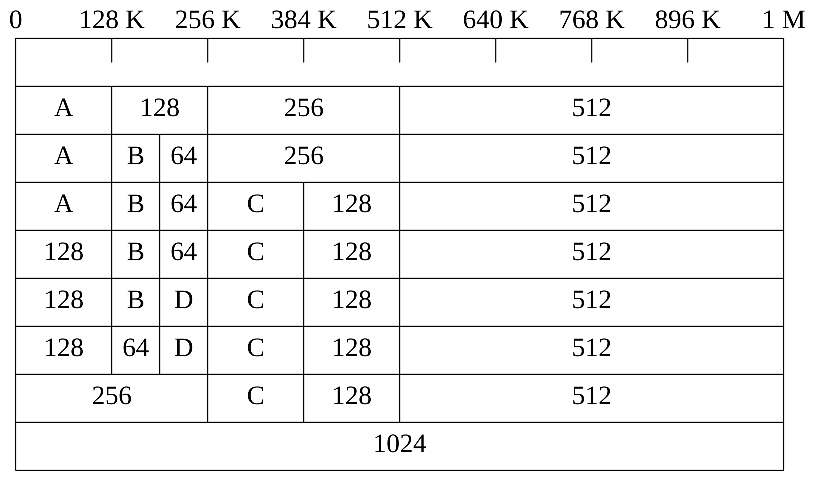
The problem here is finding the appropriate hole to meet a request for memory. We want to be able to grant as many requests as possible, but just like CPU scheduling, our enemy is uncertainty.

These are analogous to CPU scheduling algorithms (in terms of linked lists):

- First Fit - satisfy the request with the first hole that can be used. Always start from the head of the list. (Average time N/2 - O(N))
- Next Fit - satisfy the request with the first hole that can be used. Start where the last search left off. (Average time N/2, but maybe more even memory utilization O(N))
- Best Fit - pick the hole with the closest size to the request. (O(N) - with a sorted list - implies O(N) time to insert a new hole, which we do on every allocation)
- Worst Fit - pick the hole that worst matches the request. Why? best fit can result in a lot of small holes that are useless. Worst fit keeps big holes around. (Avg time is O(N/2) if we keep the list sorted (O(1) to find the hole and O(N) to insert it).
- Quick Fit - keep a list of common sizes and allocate from it.

The Buddy System

The Buddy System is designed to make merges fast when blocks are returned. Blocks are allocated in sizes that are powers of 2. When a block is returned, it can be merged if the block that is the same size and next to it is also available. Search times are very short (even without expensive indexing systems) for that reason.


0
128 K
256 K
384 K
512 K
640 K
768 K
896 K
1 M
A | 128 | 256 | 512
A | B | 64 | 256 | 512
A | B | 64 | C | 128 | 512
128 | B | 64 | C | 128 | 512
128 | B | D | C | 128 | 512
128 | 64 | D | C | 128 | 512
256 | C | 128 | 512
1024

The buddy system is a fast merger, but introduces internal fragmentation.

Swap Management

Swap space is the disk used to store the memory in use by swapped out processes. Swap space management has similar problems to those of variable space allocations, and similar solutions. (Fixed disk block sizes imply that there will be internal fragmentation on disk).

One difference is whether to acquire swap space for a process when it starts, or to hope that space will be available if the process needs to be swapped out. Hoping (called overcommitting) allows more processes to run, but runs the risk that a process will need to be swapped, and there isn't room for it - a situation similar to deadlock that requires killing a process. Requiring swap to be allocated for each process may require the system to reserve significantly more swap space per-process.

This problem really occurs more in terms of backing store for paging systems.

Virtual Memory (VM)

What's a paging system?

More properly, what's this Virtual Memory you keep alluding to?

Virtual memory is designed to solve the problem of running a program that needs more memory than the hardware has. One way of approaching this is to use *overlays*. Overlays are code and data written to memory under system or programmer control to reuse memory for a process. The old memory could be overwritten or saved first to disk for later use as another overlay. Programmers had to create the overlays, which required laying out their code is such a way that it could be overlaid. VM provides the same functionality, and solved the protection and relocation problems in an interesting way.

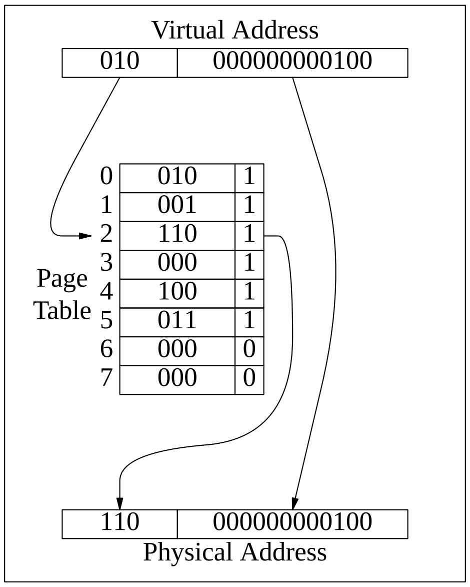
Central to VM is the idea of a *virtual address* and the associated *virtual address space.* Under VM all processes execute code written in terms of virtual addresses that are translated by the memory management hardware into the appropriate physical address. Each process thinks it has access to the whole physical memory of the machine. This solves the relocation problem - no rewriting of addresses is **ever** necessary, and the protection problem because a process can no longer express the idea of accessing another process's memory.[1] The open issues are how the virtual to physical translation is made, and how this all allows automatic overlays.

A virtual address is generally broken up into two parts a page number and an offset.

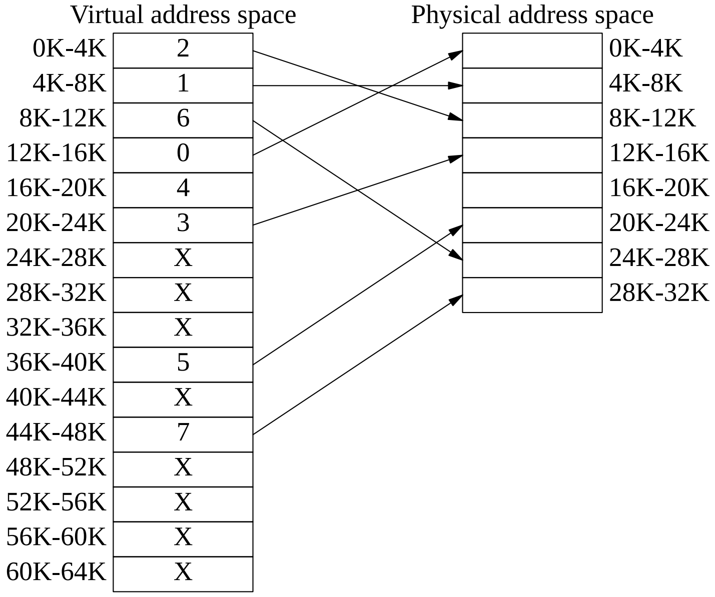
[1] We'll discuss how processes can share memory under VM later.

Page	Offset
0101	110101110101
5	d75

The size of the offset determines the size of the pages, and the size of the page number the number of pages. Address translation is accomplished via a page table which lists the physical address a current page resides in. The physical 1-page units are called *page frames.*



The MMU hardware knows what a page table looks like and its location. Every memory reference is translated from the virtual address of the into the read address needed to access the physcial memory. The result is an address space that appears contiguous to a process, but in reality may be spread across many pages in the physical memory.



Note that not all pages are present in physical memory at once. When the MMU tries to translate a virtual address that is not mapped to a page frame, a *page fault* occurs. This is an interrupt back to the CPU, which must load the page from *backing store* (pages not in use are stored on disk, called backing store) and update the page table entry. In the process, the OS may need to write another page to disk and invalidate its entry. Once the required page is in memory, and the page table is correct, the OS restarts the faulting instruction.

The result is a system that allows a process to address more memory than is physically present in the computer, because memory is automatically moved to and from backing store as needed. Furthermore, the virtual address structure solves the relocation and protection problems that were so thorny without VM. All this is paid for with a higher **per-memory-reference** cost.

Next time we'll talk about how to make this all efficient, and about more details.