1 Memory Management

aka Warehousing!!

Too much stuff ... too little space

- Concerned with organisation of main memory
- Concept of a *Memory Hierarchy*
- Multiple programs belonging to different users active in memory at the same time - hence *need for protection*
- Dynamic vs. Static partitioning of memory
- Placement of programs within the available memory, where ? which to move out ?
- Ability to share code and data
- Memory sharing transparent - can't tell if it is shared (illusion of an infinite memory space)

Possible to have *multiple levels* of caching

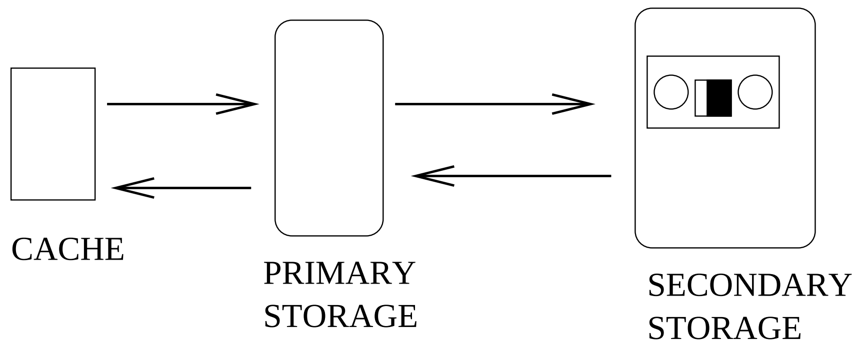

Figure 1: Memory Hierarchy

- There is **never** enough memory (RAM or otherwise)
- Parkinson's law : *"Programs expand to fill the memory available to hold them"*

Idea : Manage the storage available efficiently between the available programs.

2 Single User - Contiguous Storage

- Single user program in memory - and size of program fits memory
- Part of the memory held the operating system, the remainder the user program
- OS protected from user programs (to prevent user program overwriting OS code)
- OS protection by the user of a single *boundary* register built into the CPU
- Boundary register checked on each storage request
- However, if the user needs to use certain OS features (which it definitely will), use OS features through a *kernel* or *supervisor* call
- The size of programs in memory is restricted by the space available

What if multiple jobs active in memory simultaneously ?
Need for protection from **each other** and protection of **OS**

3 Multiprogramming with fixed partitions

Consider the warehouse example again, multiple jobs of different types (perhaps size) entering storage in different **partitions**

- Several users simultaneously compete for system resources
- switch between I/O jobs and calculation jobs for instance
- To take advantage of this sharing of CPU, important for many jobs to be present in main memory
- Allowing **Relocation** and **Transfers** between partitions
- **Protection** implemented by the use of several *boundary registers*: *low and high* boundary registers, or *base* register with *length*
- **Fragmentation** occurs if user programs cannot completely fill a partition - wasteful.

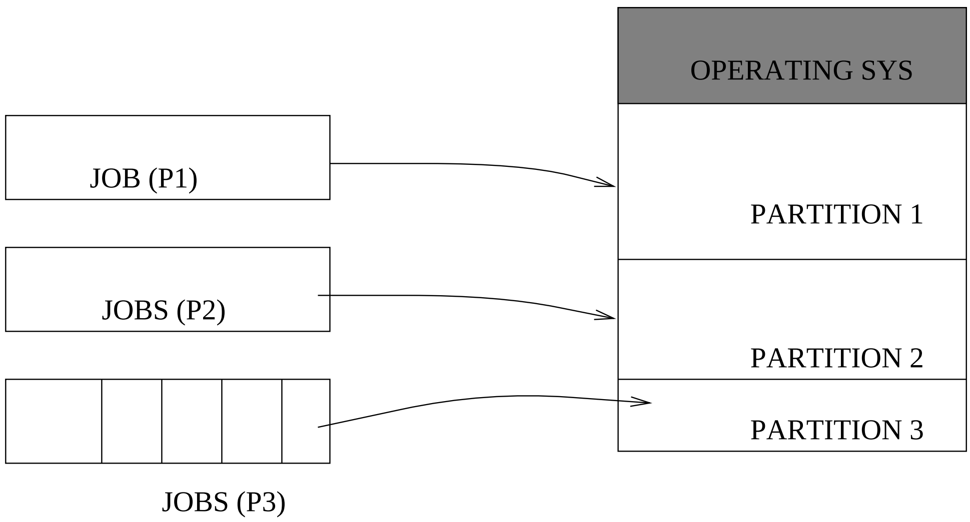

Figure 2: Multiprogramming - fixed partitions

4 Multiprogramming with variable partitions

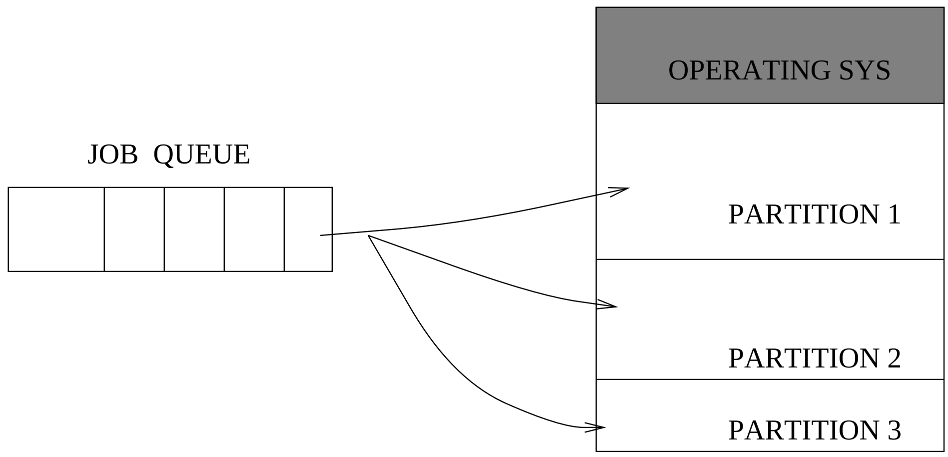

Figure 3: Multiprogramming - variable partitions

- Variable partitions - allowing jobs to use as much space as they needed (limit being the complete memory space)
- No need to divide jobs into *types* - reduce waste if jobs could not fill partition

However, complete wastage is still not reduced

- *Consider what happens when multiple jobs in memory, and jobs start completing activity* - this leads to the creation of **holes** in main storage that must be filled.
- **Question :** What happens if no job in queue can meet the size of the slot left by the departing job ?

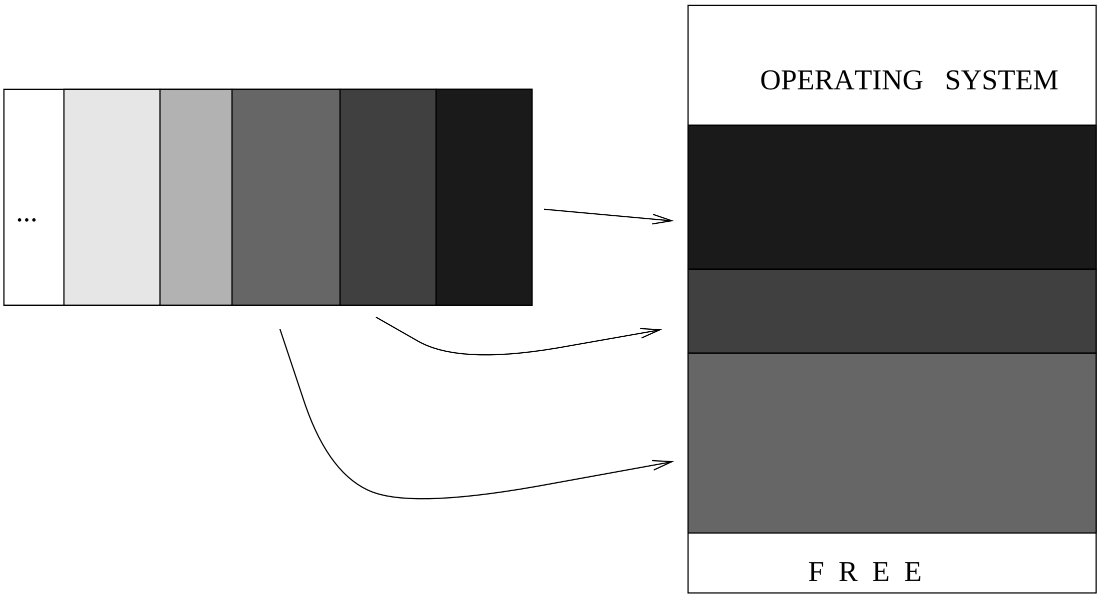

Figure 4: Allocating Memory

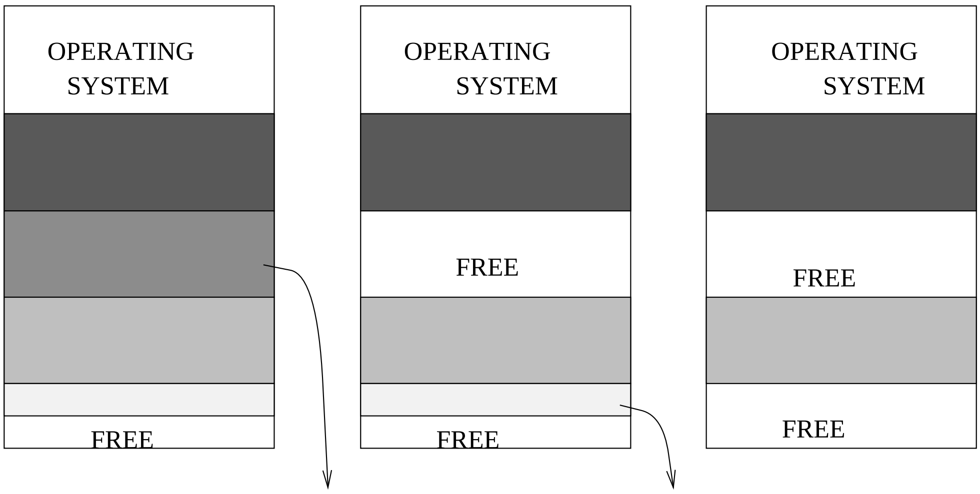

Figure 5: Too much free space - but not enough for task

5 Storage Placement

- Determines where in storage to put incoming programs and data
- Three strategies most frequently mentioned :
- **Best fit strategy** : An incoming job is placed in the hole where it best fits (i.e. the amount of free space left is minimal)
- **First fit strategy** : Placed in the first available slot large enough to hold the job
- **Worst fit strategy** : Place in storage in the largest slot available. The remaining may still be large enough to hold another job.

Many variations to these strategies are possible.

6 Storage Compaction

- Departing jobs leave their space free
- New job (from queue) must be able to fill the space (i.e. must be either of smaller or same size as the departing job).
- *Holes* created in memory
- The idea of combining *holes or slots* together = **Compaction**
- **Compaction** : Move all occupied areas of store to one end - leaving one large free space for incoming jobs, instead of numerous small ones
- Must be performed on *each* new allocation of job to memory or *completion* of job from memory
- System must also maintain relocation information
- Wasteful of CPU ? Can take a long time to just do compaction and not any serious computation
- Also called - **Garbage Collection** - *hours on early computers*

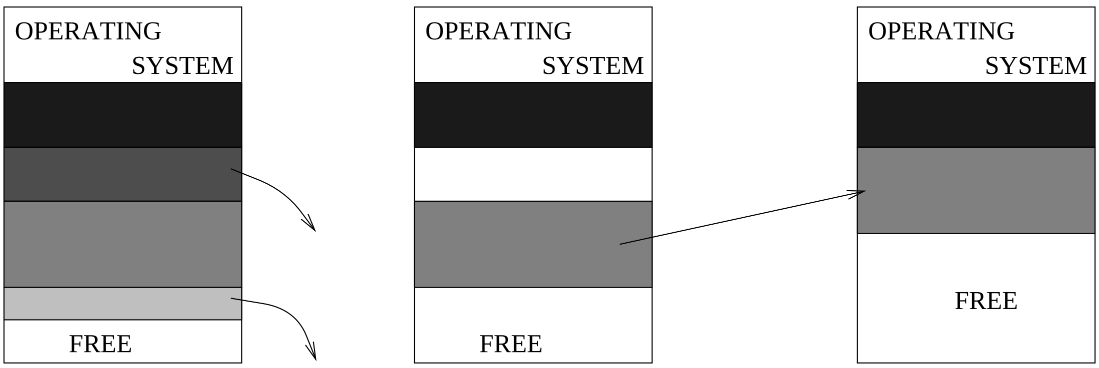

Figure 6: Storage Compaction

7 Swapping

- In previous schemes, user programs remain in memory until completion
- Now - another twist - swap job out of main storage (to secondary storage) if it requires service from some external routine (so another process can use the CPU)
- A job may typically be need to swapping in and out many time before completion
- Early systems - Only one job occupies the main storage at one time
- Now, main store large enough to have many user programs active at the same time (swapping in and out of main memory)
- Other variations of swapping exist, will come to these later
- Allocate a *bit more* than necessary for process to grow during execution
- How do we keep track of memory in implementation : *use of linked lists, buddy system (dividing memory according to a power of 2) - leads to big wastes (checkerboading or external fragmentation) or bitmaps (use a structure, where a 0 indicates if block is free, and 1 otherwise*

- When process in memory, no disk space allocated to it
- However, when a need for swapping occurs, suitable disk space must be found
- Area on disk where this happens : *swap space* (usually the /tmp area in unix)
- Sometimes, swap space automatically allocated on process creation (hence, a fixed swap space per process)
- An *average process* in the middle of memory (after system has reached equilibrium) will encounter half allocations and half deallocations (above and below)
- The **Fifty percent rule** = if *mean* number of processes in memory is n, the mean number of holes is $n/2$
- Thats because adjacent holes are merged, adjacent processes are not (hence a process / hole asymmetry) - just a heuristic
- Numerous others, used for optimising swapping performance

8 Address Space and Virtual Memory

Remember? : We covered it in the context of processes and threads.

- It constitutes all the addresses a program can touch. All the state that a program can affect or be affected by
- There are generally two views of memory :
  One view from the CPU : What the program sees : **Virtual Memory**
  View from the memory : What exists in practise : **Physical Memory**
- The memory management unit (MMU) translates the CPU (program) addresses into real (physical) addresses
- The translation via MMU helps to achieve protection, as no two programs can access the same physical memory
- The translation via MMU can take place by a number of different methods, which we shall now discuss
- Similar to one person giving orders, another person interpreting these orders and performing activity (also somewhat like macros - a simple name can mean many activities combined together).
- Modification of the translation table can only be done by kernel (in supervisor or previledged mode)
- So, *user* program can only affect its *own address space*, *kernel* can affect *all address spaces*

9 Memory Management and Address Translation

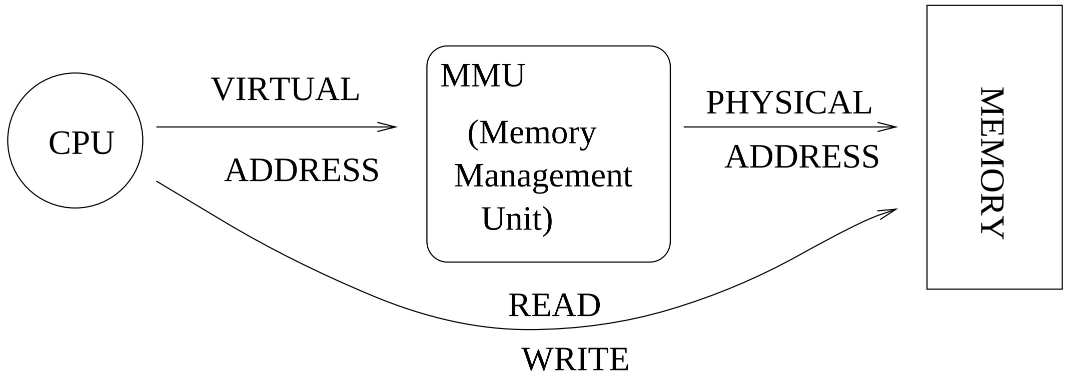

Figure 7: Memory Management Unit (MMU)

10 Linker Loader

Question : Can multiple programs share physical memory without hardware translation ?

- Yes : when program is copied into memory its addresses (loads, stores, branches, jumps) are changed to use the addresses of where program is placed in memory
- The *linker-loader* - which does the resolving
- Unix **ld** works this way : compiler generates a .o file with code that starts at location 0. How is an *actual* executable created from this ?
- Go through each .o file, and change addresses to point to where each module goes in the larger program (compiler oriented)
- With *linker-loader* no protection: one program's bugs can cause others to crash

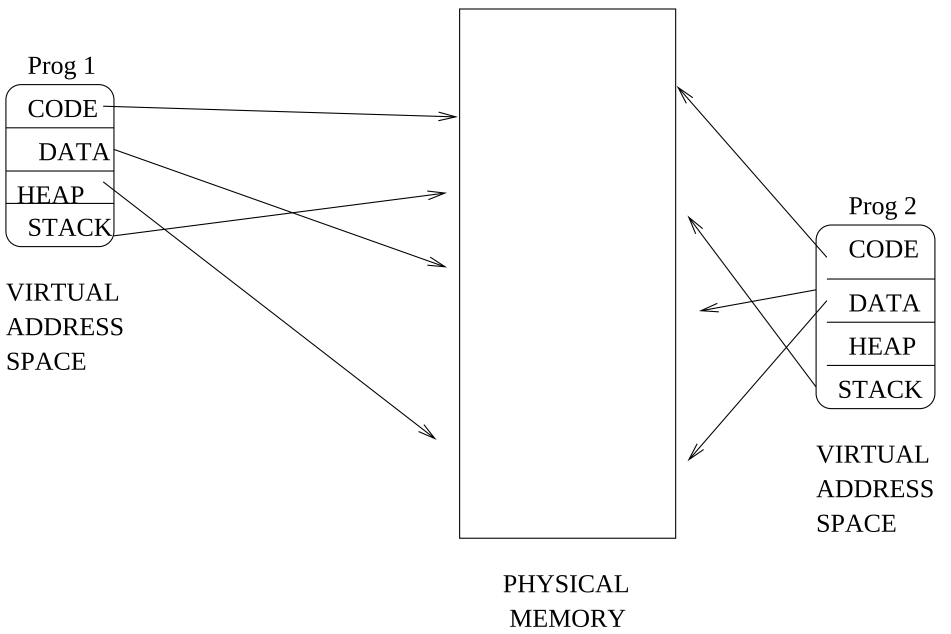

Figure 8: Multiple User Programs in Memory

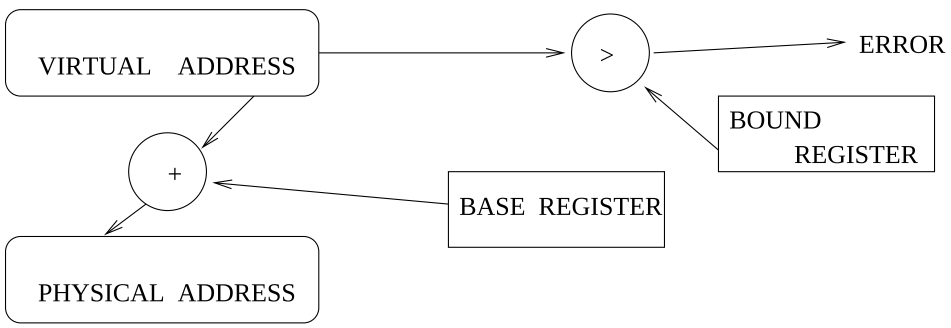

Figure 9: Use of Base and Bound Register

- **Bound Register** keeps check no unauthorised locations are referenced
- **Base Register** helps to determine *virtual address*