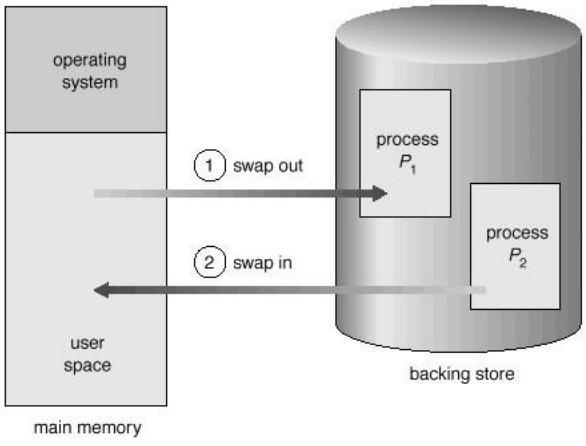
Swapping

- What : temporarily move inactive process to *backing store* (e.g., fast disk). At some later time, return it to main memory for continued execution.
- Why : permit other processes to use memory resources (hence each process can be bigger)
- Who : decision of what process to swap made by medium-term scheduler

Schematic view of Swapping



Swapping

Some possibilities of when to swap

- — if you have 3 processes, start to swap one out when its quantum expires while two is executing. Goal is to have third process in place when twos quantum expires (i.e., overlap computation with disk i/o)
- — context switch time is very high if you cant achieve this

Another option: *roll out* lower priority process in favor of higher priority process. *Roll in* the lower priority process

- when the higher priority one finishes

Swapping

- If you have static address binding (i.e., compile or load time binding) have to swap process back into same memory space. *Why?*
- If you have execution-time address binding, then you can swap the process back into a different memory space.
- Disk is slow and the transfer time needed is proportional to the size of the process, so it is useful if processes can specify the parts of allocated memory that are unused to avoid having to transfer.

Swapping

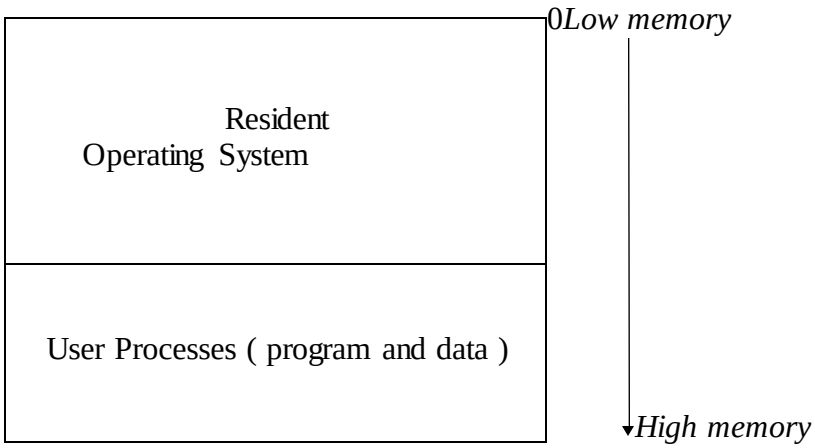
Process cannot be swapped until completely idle. Example of a problem: overlapped DMA input/output. (This requires that you have buffer space allocated in memory when the i/o request comes back)

Note that in general swapping in this form (i.e., with this large sized granularity) is not very common now.

Contiguous Allocation

- Divide memory into *partitions*. Initially consider two partitions--one for the resident operating system and one for a user process.
- Where should the operating system go--low memory or high memory?
- Frequently put the operating system in low memory because this is where the interrupt vector is located. Also this permits the user partition to be expanded without running into the operating system (a factor when we have more than one partition or if we run the same binaries on different system configurations).

Memory Partitions



Single Partition Allocation

- Initial location of the users process in memory is *not* 0
  - The relocation register (base register) points to the first location in the users partition. Users logical addresses are adjusted by the hardware to produce the physical address. (Address binding delayed until execution time.)
  - Relocation register value is static during program execution, hence all of the OS must be present (it might be used). Otherwise have to relocate user code/data on the fly! In other words we cannot have transient OS code.

Single Partition Allocation

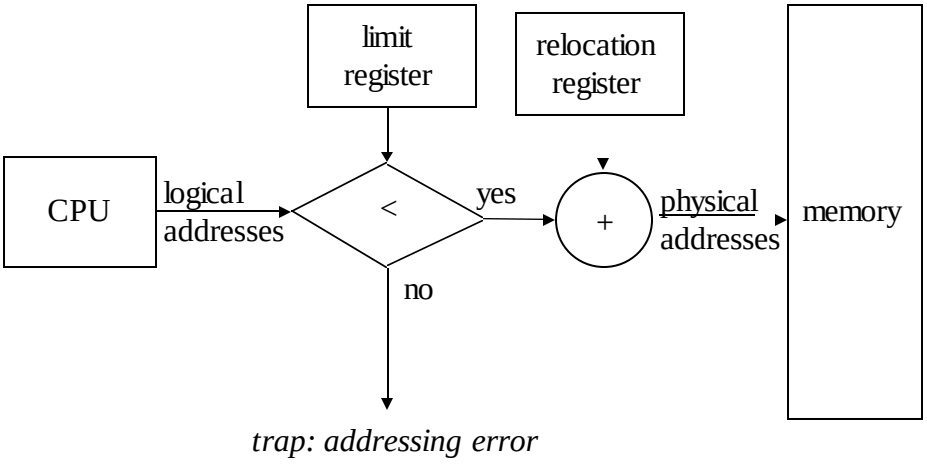
- How about memory references passed from the user process to the OS (for example, blocks of memory passed as an argument to a I/O routine)? The address must be translated from users logical address space to the physical address space. Other arguments dont get translated (e.g., counts). Hence OS software has to handle these translations.

Limit Register

- How do we protect the OS from accidental or intentional interference from user processes?

¥ Add a *limit register* to the address mapping scheme

Limit Register



Multiple-Partition Allocation

Goal: allocate memory to *multiple* processes (which permits rapid switches, for example)

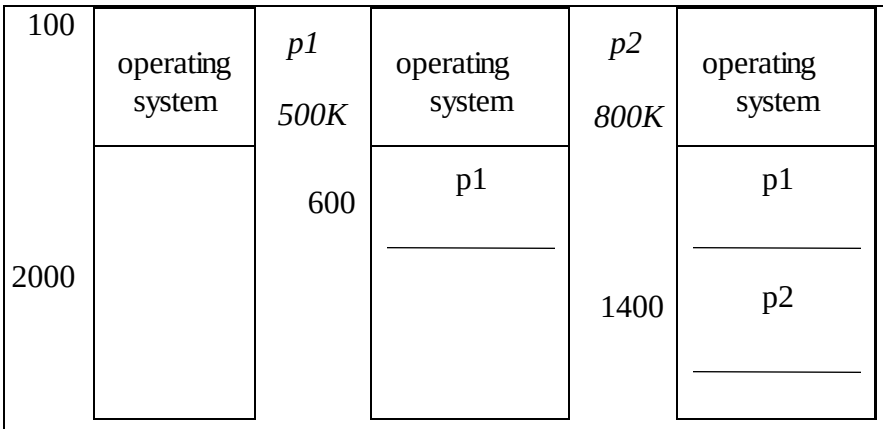
Simple scheme: fixed-size partition

— memory divided into several partitions of fixed size
— each partition holds one process
— partition becomes free when process terminates; another process picked from the ready queue gets the free partition
— number of partitions bounds the degree of multiprogramming
— originally used in the IBM OS/360 operating system (MFT)
— No longer used

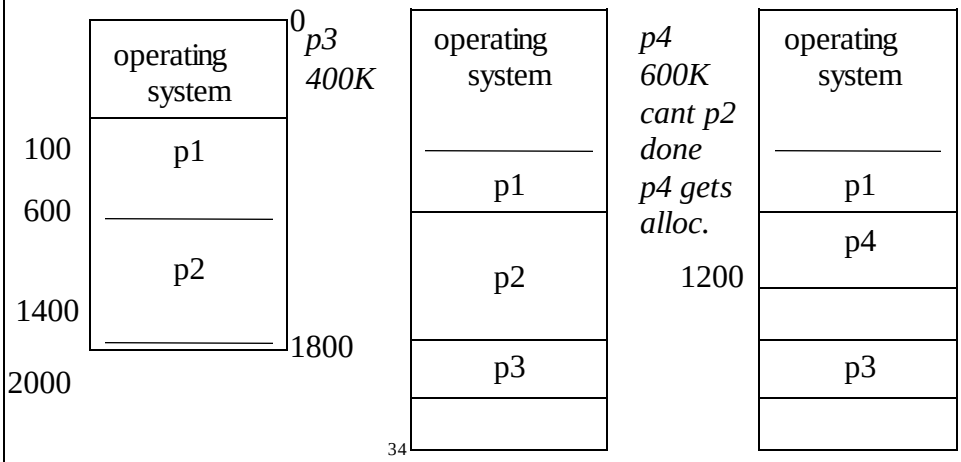
Multiple-Partition Allocation Dynamic Partition

- Memory is partitioned dynamically
  - **Hole**: block of available memory
  - Holes of various size are scattered throughout memory Process still must occupy contiguous memory
- OS keeps a table listing which parts of memory are available
  - Allocated partitions
  - Free partitions (hole)
- When a process arrives, the OS searches for a part of memory that is large enough to hold the process. Allocates only the amount of needed memory.

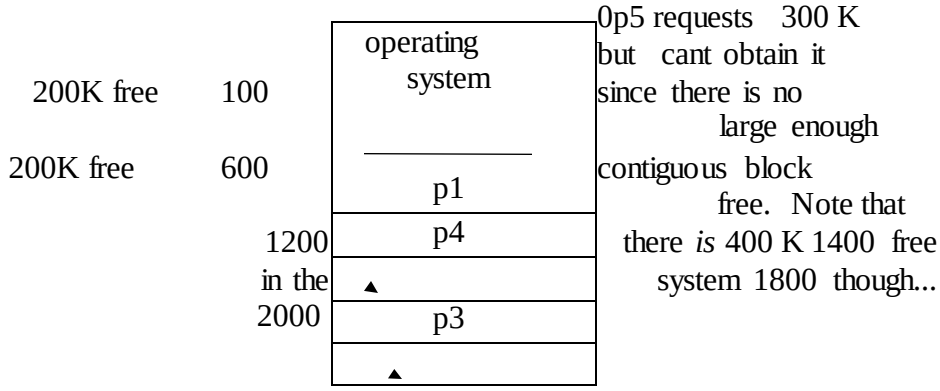
Multiple-Partition Allocation Dynamic Partition



Multiple-Partition Allocation
Dynamic Partition



Multiple-Partition Allocation Dynamic Partition



Multiple-Partition Allocation Dynamic Partition

- This is an example of *external fragmentation*--sufficient amount of free memory to satisfy request but not in a contiguous block.
- We used a *first fit* algorithm this time to decide where to allocate space--what are some strategies for finding a free hole to fill?

Multiple-Partition Allocation Dynamic Partition

- *first fit* algorithm: allocate the *first* hole that is big enough. Searching can start either at beginning of set of holes or where the previous first-fit search ended. We quit when we find a free hole that is large enough. *best fit*: allocate the *smallest* hole that is big enough. Must search entire list to find it if you dont keep free list ordered by size.
- *worst fit*: allocate the *largest* hole. Again may need to search entire free list if not ordered. Produces the largest leftover hole, which may be less likely to create external
- fragmentation.

Multiple-Partition Allocation Dynamic Partition

Simulation shows that first-fit and best-fit are better than worst-fit for time and storage use.

First-fit is faster than best-fit

First-fit and best-fit are similar in storage use.

50% rule--up to 1/3 of memory is lost to external fragmentation in first-fit (*N* allocated, 1/2 *N* lost)

Multiple-Partition Allocation Dynamic Partition

- General comments:
  - — memory protection is necessary to prevent state interactions. This is effected by the limit register.
  - — base registers are required to point to the current partition
- In general, blocks are allocated in some quantum (e.g., power or 2). No point in leaving space free if you cant address it or if it is too small to be of any use at all. Also there is an expense in keeping track of free space (free list; traversing list; etc.).
- This results in lost space--allocated but not required by process
- **Internal fragmentation**: difference between required memory and allocated memory.
- Internal fragmentation also results from estimation error and management overhead.

External Fragmentation

External fragmentation can be controlled with compaction.

- — requires dynamic address binding (have to move pieces around)
- — can be quite expensive in time
- — some schemes try to control expense by only doing certain kinds of coalescing--e.g., on power of 2 boundary. (Topic of a data structures class.)
- — OS approach can also be to roll out/roll in all processes, returning processes to new addresses--no additional code required!