

Topics in Memory Management

- Uniprogrammed operating systems
 - Assembling, linking, loading
 - Static memory allocation
 - Dynamic memory allocation
 - Stacks, heaps
 - Managing the free list, memory reclamation
- Multiprogrammed operating systems
 - Includes most of the above topics
 - Static relocation
 - Dynamic relocation
 - Virtual vs. physical address
 - Partitioning (and compaction)
 - Segmentation
 - Paging
 - Swapping
 - Demand paging

Managing the Free List

- Heap-based dynamic memory allocation techniques typically maintain a *free list*, which keeps track of all the holes
- Algorithms to manage the free list:
 - Best fit
 - Keep linked list of free blocks
 - Search the whole list at each allocation
 - Choose the hole that comes the closest to matching the request size
 - Any unused space becomes a new (smaller) hole
 - When freeing memory, combine adjacent holes
 - Any way to do this efficiently?
 - First fit
 - Scan the list for the first hole that is large enough, choose that hole
 - Otherwise, same as best fit
 - Which is better? Why??

Reclaiming Dynamic Memory

- When can memory be freed?
 - Whenever programmer says to
 - Any way to do so automatically?
- Potential problems in reclamation
 - Dangling pointers — have to make sure that everyone is finished using it
 - Memory leak — must not “lose” memory by forgetting to free it when appropriate
- Implementing automatic reclamation:
 - Reference counts
 - Used by file systems
 - OS keeps track of number of outstanding pointers to each memory item
 - When count goes to zero, free the memory

Reclaiming Dynamic Memory (cont.)

- Implementing automatic reclamation:
 - Garbage collection
 - Used in LISP for years, now used in Java
 - Storage isn't explicitly freed by a free operation; programmer just deletes the pointers and doesn't worry about what it's pointing at
 - When OS needs more storage space, it recursively searches through all the active pointers and reclaims memory that no one is using
 - Makes life easier for application programmer, but is difficult to program the garbage collector
 - Often expensive — may use 20% of CPU time in systems that use it
 - May spend as much as 50% of time allocating and automatically freeing memory

Multiprogramming — Goals in Sharing the Memory Space

■ *Transparency:*

- Multiple processes must coexist in memory
- No process should be aware that the memory is shared
- Each process should execute regardless of where it is located in memory

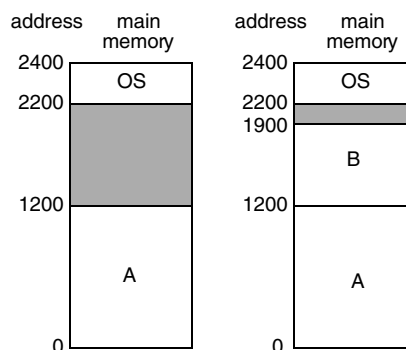
■ *Safety:*

- Processes must not be able to corrupt each other, or the OS
- *Protection* mechanisms are used to enforce safety

■ *Efficiency:*

- The performance of the CPU and memory should not degrade very much as a result of sharing

Static Relocation



- Put the OS in the highest memory
- Compiler and linker assume each process starts at address 0
- At load time, the OS:
 - Allocates the process a segment of memory in which it fits completely
 - Adjusts the addresses in the processes to reflect its assigned location in memory

Static vs. Dynamic Relocation

■ Problems with static relocation:

- Safety — not satisfied — one process can access / corrupt another's memory, can even corrupt OS's memory
- Processes can not change size (why...?)
- Processes can not move after beginning to run (why would they want to?)
- Used by MS-DOS, and early versions of Windows and Mac OS

■ An alternative: dynamic relocation

- The basic idea is to change each memory address dynamically as the process runs
- Translation done by hardware — between the CPU and the memory is a *memory management unit* (MMU) that converts virtual addresses to physical addresses
 - This translation happens for every memory reference the process makes

Dynamic Relocation

- There are now two different views of the address space:
 - The *physical address space* — seen only by the OS — is as large as there is physical memory on the machine
 - The *virtual (logical) address space* — seen by the process — can be as large as the instruction set architecture allows
 - For now, we'll assume it's much smaller than the physical address space
 - Multiple processes share the physical memory, but each can see only its own virtual address space
- The OS and hardware must now manage two different addresses:
 - *Virtual address* — seen by the process
 - *Physical address* — address in physical memory (seen by OS)