Memory management

- internal to process
  - segments
  - static allocation
  - dynamic allocation
    - allocation methods: stack and heap
    - reclaiming: garbage collection
- external
  - static relocation
  - dynamic relocation
    - swapping
    - compaction
  - paging (next lecture but one)

Types of information information stored in memory

- by role in program:
  - program instructions
  - constants:
    - pi, maxnum, strings used by printf/scanf
  - variables:
    - locals, globals, function parameters, dynamic storage (from malloc or new)
    - initialized or uninitialized
- by protection status (important for sharing data and code):
  - readable and writable: variables
  - read-only: code, constants
- addresses vs. data:
  - must modify addresses if program is moved (relocation, garbage collection)

Segments

- process' memory is divided into logical *segments* (text, data, bss, heap, stack)
  - some are read-only, others read-write
  - some are known at compile time, others grow dynamically as program runs
- who assigns memory to segments?
  - *compiler* and *assembler* generate an *object file* (containing code and data segments) from each *source file*
  - *linker* combines all the object files for a program into a single executable object file, which is complete and self-sufficient
  - *loader* (part of OS) loads an executable object file into memory at location(s) determined by the operating system
  - *program* (as it runs) uses **new** and **malloc** to dynamically allocate memory, gets space on stack during function calls

Internal memory allocation

- memory allocation
  - static – done before run-time
  - dynamic – done at run-time
- static allocation does not satisfy for all programming needs
  - programmer may not know how much memory will be needed when program runs
  - OS doesn't know in advance which procedures will be called (would be wasteful to allocate space for every variable in every procedure in advance)
  - OS must be able to handle allocation for recursive procedures (same name varables in different invocations)
- dynamic allocation requires two fundamental operations:
  - allocate dynamic storage
  - free memory when it's no longer needed

Dynamic memory allocation methods

- stack
  - good when allocation and freeing are somewhat predictable
  - typically used:
    - to pass parameters to procedures
    - for allocating space for local variables inside a procedure
    - for tree traversal, expression evaluation, parsing, etc.
  - use stack operations: **push** and **pop**
  - simple and efficient, but restrictive
  - keeps all free space together in one place
- heap
  - used when allocation and freeing are not predictable
  - used for arbitrary list structures, complex data organization, etc.
  - more general, less efficient, more difficult to implement
  - system memory consists of allocated areas and free areas (holes)

Fragmentation

- *fragmentation* – free memory is spread in multiple places of various sizes (fragments) which makes further allocation difficult/impossible and thus wastes memory
  - internal – free memory fragments are inside the allocation units
  - external – free memory fragments are outside the allocation units
- arbitrary dynamic allocation/deallocation in heap leads to fragmentation
  - is this fragmentation internal or external?
  - does stack allocation lead to fragmentation?
  - how to reuse the fragments?

Heap memory allocation

- heap-based dynamic memory allocation techniques typically maintain a *free list*, which keeps track of all the fragments (holes)
- algorithms to manage the free list:
  - first fit
    - scan the list for the first hole that is large enough, choose that hole
  - best fit
    - search free 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
    - how to implement this efficiently?
  - worst fit
    - scan for the largest hole (hoping that the remaining hole will be large enough to be useful)
  - which is better?

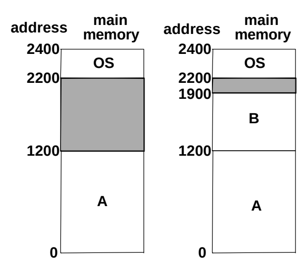
Reclaiming memory

- when can memory be made available for reuse?
  - when the programmer explicitly frees the memory
  - any way to reclaim memory automatically?
    - difficult if that item is shared (i.e., if there are multiple pointers to it)
- implementing automatic reclamation: reference counts
  - OS keeps track of number of outstanding pointers to each memory item
  - when count goes to zero, free the memory
  - potential problems
    - dangling pointers — have to make sure that <u>everyone</u> is finished using the memory when it's shared
    - memory leak — must not "lose" memory by forgetting to free it when appropriate (the pointer still points to it yet the memory is not used)

Garbage collection

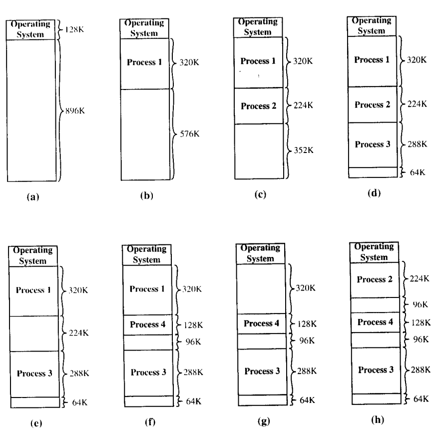
- storage isn't explicitly freed by a free operation; programmer just deletes (or reassigns) the pointers
- when OS needs more storage space, it recursively searches through all the active pointers and reclaims memory that no one is using
- simplifies memory management for the application programmer, but it 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
- used in LISP, Java

Static external 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 external relocation



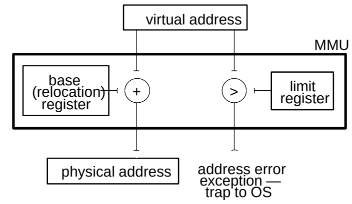
Static external relocation (cont.)

- problems with static relocation:
  - safety: one process can access/corrupt another's memory, can even corrupt OS's memory
  - process cannot request more memory (why?)
  - processes can not move after beginning to run (why would theny need to?)
  - used by MS-DOS, Windows, Mac OS
- alternative: dynamic relocation
  - the basic idea is to change each memory address dynamically at run-time
  - this translation is aided by hardware — between the CPU and the memory is a *memory management unit* (MMU) (also called a *translation unit*) that converts the programs addresses to actual addresses
    - this translation happens for every memory reference the process makes

Dynamic relocation

- there are now two different views of the address space:
  - *physical address space* (used by the OS only)
    - it is as large as there is physical memory on the machine
  - *virtual (logical) address space (used 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 has its own virtual address space
- The OS and hardware must now manage two different addresses spaces for each process

Implementing dynamic relocation



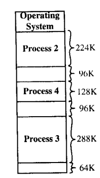
- MMU protects address space, and translates virtual addresses
  - *Base register* holds lowest virtual address of process, *limit register* holds highest
  - Translation:
    physical address = virtual address + base
  - Protection:
    if virtual address > limit, then trap to the OS with an address exception

Swapping

- processes can be swapped out to make room
  - OS *swaps a process out* by storing its complete state to disk
  - OS can reclaim space used by ready or blocked processes
- When process becomes active again, OS must *swap* it back *in* (into memory)
  - With static relocation, the process must be replaced in the same location
  - With dynamic relocation, OS can place the process in any free partition (must update the relocation and limit registers)
- Swapping and dynamic relocation make it easy to increase the size of a process and to compact memory (although slow!)

Compaction

- Dynamic relocation leads to *external fragmentation* - unused space is left between processes
- *compaction* - overcomes this problem by moving the processes so that memory allocation is contiguous
- in previous example we can compact the processes to free up 256K of contiguous memory space (enough to load additional process) by moving the total of 416K of memory
- how is it done?
- can compaction be used with static relocation?
- is compaction efficient?



(h)