CIS520 – Operating Systems
Handout 8
Introduction to Memory Management

- Point of memory management algorithms - support sharing of main memory. We will focus on having multiple processes sharing the same physical memory. Key issues:
  - Protection. Must allow one process to protect its memory from access by other processes.
  - Naming. How do processes identify shared pieces of memory.
  - Transparency. How transparent is sharing. Does user program have to manage anything explicitly?
  - Efficiency. Any memory management strategy should not impose too much of a performance burden.
- Why share memory between processes? Because want to multiprogram the processor. To time share system, to overlap computation and I/O. So, must provide for multiple processes to be resident in physical memory at the same time. Processes must share the physical memory.
- Historical Development.
  - For first computers, loaded one program onto machine and it executed to completion. No sharing required. OS was just a subroutine library, and there was no protection. What addresses does program generate?
  - Desire to increase processor utilization in the face of long I/O delays drove the adoptation of multiprogramming. So, one process runs until it does I/O, then OS lets another process run. How do processes share memory? Alternatives:
    * Load both processes into memory, then switch between them under OS control. Must relocate program when load it. Big Problem: Protection. A bug in one process can kill the other process. MS-DOS, MS-Windows use this strategy.
    * Copy entire memory of process to disk when it does I/O, then copy back when it restarts. No need to relocate when load. Obvious performance problems. Early version of Unix did this.
    * Do access checking on each memory reference. Give each program a piece of memory that it can access, and on every memory reference check that it stays within its address space. Typical mechanism: base and bounds registers. Where is check done? Answer: in hardware for speed. When OS runs process, loads the base and bounds registers for that process. Cray-1 did this. Note: there is now a translation process. Program generates virtual addresses that get translated into physical addresses. But, no longer have a protection problem: one process cannot access another's memory, because it is outside its address space. If it tries to access it, the hardware will generate an exception.
- End up with a model where physical memory of machine is dynamically allocated to processes as they enter and exit the system. Variety of allocation strategies: best fit, first fit, etc. All suffer from external fragmentation. In worst case, may have enough memory free to load a process, but can't use it because it is fragmented into little pieces.
- What if cannot find a space big enough to run a process? Either because of fragmentation or because physical memory is too small to hold all address spaces. Can compact and relocate processes (easy with base and bounds hardware, not so easy for direct physical address machines). Or, can swap a process out to disk then restore when space becomes available. In both cases incur copying overhead. When move process within memory, must copy between memory locations. When move to disk, must copy back and forth to disk.

- One way to avoid external fragmentation: allocate physical memory to processes in fixed size chunks called page frames. Present abstraction to application of a single linear address space. Inside machine, break address space of application up into fixed size chunks called pages. Pages and page frames are same size. Store pages in page frames. When process generates an address, dynamically translate to the physical page frame which holds data for that page.
- So, a virtual address now consists of two pieces: a page number and an offset within that page. Page sizes are typically powers of 2; this simplifies extraction of page numbers and offsets. To access a piece of data at a given address, system automatically does the following:
  - Extracts page number.
  - Extracts offset.
  - Translate page number to physical page frame id.
  - Accesses data at offset in physical page frame.
- How does system perform translation? Simplest solution: use a page table. Page table is a linear array indexed by virtual page number that gives the physical page frame that contains that page. What is lookup process?
  - Extract page number.
  - Extract offset.
  - Check that page number is within address space of process.
  - Look up page number in page table.
  - Add offset to resulting physical page number
  - Access memory location.
- With paging, still have protection. One process cannot access a piece of physical memory unless its page table points to that physical page. So, if the page tables of two processes point to different physical pages, the processes cannot access each other's physical memory.
- Fixed size allocation of physical memory in page frames dramatically simplifies allocation algorithm. OS can just keep track of free and used pages and allocate free pages when a process needs memory. There is no fragmentation of physical memory into smaller and smaller allocatable chunks.
- But, are still pieces of memory that are unused. What happens if a program's address space does not end on a page boundary? Rest of page goes unused. This kind of memory loss is called internal fragmentation.