Lecture Overview

- Memory management
  - Address binding
  - Multiprogramming and CPU utilization
  - Contiguous memory management
  - Noncontiguous memory management
    - Paging

Memory

- Ideally programmers want memory that is
  - Large
  - Fast
  - Nonvolatile
- Memory hierarchy
  - Small amount of fast, expensive cache
  - Some medium-speed, medium price main memory
  - Gigabytes of slow, cheap disk storage
- Memory manager handles the memory hierarchy

Process Memory Address Binding

- Program instructions and data must be bound to memory addresses before it can be executed, this can happen at three different stages
  - *Compile time*: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes
  - *Load time*: Must generate *relocatable* code if memory location is not known at compile time
  - *Execution time*: Binding delayed until run time if the process can be moved during its execution from one memory segment to another; need hardware support for address maps (e.g., base and limit registers)

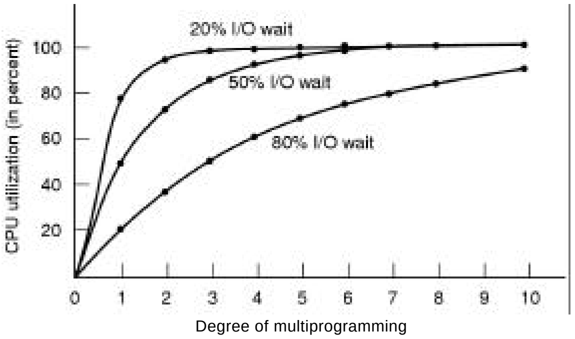
Memory Management

- The simplest approach for managing memory is to execute a process and give it all the memory
  - Every now and then the process can be saved to the disk and another process can be loaded from the disk and be given all the memory
- Just like we want to share the CPU to get better utilization, we also want to share memory to get better utilization
  - A process might not need all the memory, so it would be a waste to give it all the memory

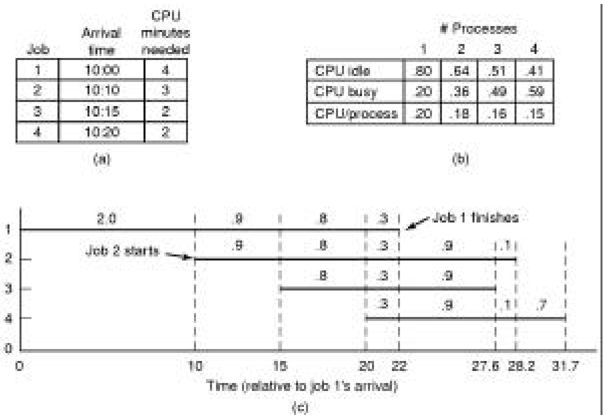
Multiprogramming and CPU Utilization

CPU utilization is a function of number of processes in memory

- CPU utilization = $1 - p^n$
  where p is percentage of time a process is waiting for I/O and n is the number of processes in memory (this is a simplistic equation)
- It is common for processes to exhibit 80% I/O wait time or more



Multiprogramming and CPU Utilization



Job	Arrival time	CPU minutes needed
1	10:00	4
2	10:10	3
3	10:15	2
4	10:20	2

(a)

	# Processes 1	2	3	4
CPU idle	.80	.64	.51	.41
CPU busy	.20	.36	.49	.59
CPU/process	.20	.18	.16	.15

(b)

(c)

- Arrival and work requirements of 4 jobs
- CPU utilization for 1 – 4 jobs with 80% I/O wait
- Sequence of events as jobs arrive and finish
  - Numbers show amount of CPU time jobs get in each interval

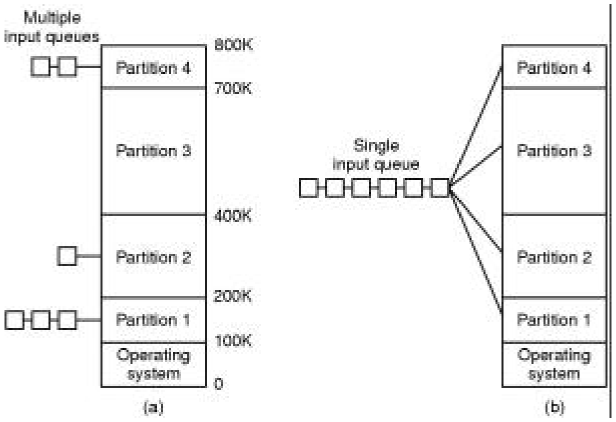
Swapping

- In a multiprogrammed OS, not all processes can be in main memory at the same time
- A process can be *swapped* temporarily out of memory to a *backing store* and then brought back into memory for continued execution
- Backing store is a fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped.

Contiguous Memory Management

- Another simple approach to multiprogramming is to divide memory into a fixed number of partitions
  - Partitions may be of equal or different sizes
- Processes wait on an input queue for a particular memory partition
- Processes execute for some period of time and then are swapped out to give another process a chance to run (if no more partitions are available)

Contiguous Memory Management



Fixed memory partitions can be implemented with

- Separate input queues for each partition
- Single input queue

Contiguous Memory Management

- Given a memory partition scheme, it is clear that we cannot be sure where program will be loaded in memory
  - Address locations of variables, code routines cannot be absolute
  - Must keep a program out of other processes' partitions
- Must use base and limit values
  - Address locations added to base value to map to physical address
  - Address locations larger than limit value is an error

Logical and Physical Addresses

- The concept of a *logical address space* that is bound to a separate *physical address space* is central to memory management
  - *Logical address* are generated by the CPU; also referred to as *virtual address*
  - *Physical address* is generated by the memory unit
- Logical and physical addresses are the same in compile-time and load-time address-binding schemes
- Logical and physical addresses differ in execution-time address-binding scheme

Memory Management Unit (MMU)

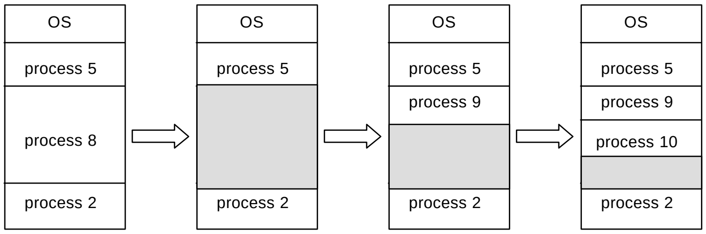
- Hardware device that maps virtual to physical address
- In MMU scheme, every address generated by a user process is manipulated by the MMU to calculate the physical address at the time it is sent to memory
  - For example, add base register and check address against limit register
- The user programs deal with virtual addresses only; they never see the physical addresses

Contiguous Memory Management

- A more complex memory management approach is a variable partitioned approach
  - A *hole* is a block of available memory; holes of various size are scattered throughout memory
  - When a process arrives, it is allocated memory from a hole large enough to accommodate it
  - Operating system maintains information about
    - Allocated partitions
    - Free partitions (i.e., holes)

Contiguous Memory Management

- As processes arrive, they are loaded into a hole that is big enough to accommodate them and the excess space is cut off to create the remaining hole



Memory Partition Allocation Algorithm

- How to satisfy request of size *n* from a list of holes?
  - *First-fit*
    - Allocate the *first* hole that is big enough
  - *Best-fit*
    - Allocate the *smallest* hole that is big enough
    - Must search entire list, unless ordered by size
    - Produces the smallest leftover hole
  - *Worst-fit*
    - Allocate the *largest* hole
    - Must also search entire list, unless ordered by size
    - Produces the largest leftover hole

First-fit and best-fit better than worst-fit in terms storage utilization

Memory Fragmentation

- *External fragmentation*
  - When allocating a hole, the remaining free space is cut off creating a small/smaller hole
  - Over time there will be many non-contiguous holes all over the memory space
  - It may not be possible to satisfy a request for memory even if the memory is available because it is not contiguous
- *Internal fragmentation*
  - Creating arbitrarily small holes in memory (i.e., a couple bytes) is inefficient, so we might choose a minimum partition size
  - In such a scenario, allocated memory may be slightly larger than requested memory
  - This internal size difference is then wasted memory

Memory Fragmentation

- Reduce external fragmentation by *compaction*
  - Shuffle memory contents to place all free memory together in one large block
  - Compaction is possible *only* if relocation is dynamic, and is done at execution time
  - I/O problem
    - Latch job in memory while it is involved in I/O
    - Do I/O only into OS buffers

Noncontiguous Memory Management

- For N memory blocks, the loss of $0.5N$ blocks is possible due to external fragmentation
  - *50-percent rule*
  - This means that one-third of memory is not usable
- Compaction is too costly to perform regularly
- External fragmentation arises because we are trying to allocate memory contiguously
- We can deal with external fragmentation if we can allow process memory to be noncontiguous

Paging

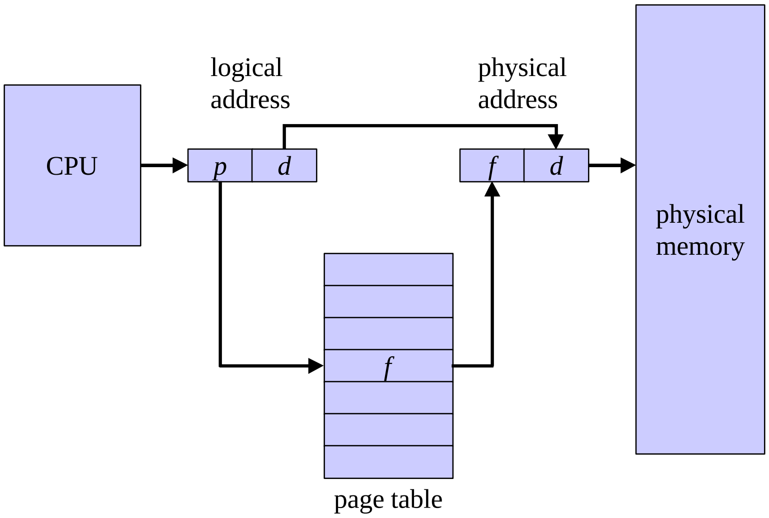
- Paging permits physical address space of a process to be noncontiguous
- Divide physical memory into fixed-sized blocks
  - Called *frames* (size is power of 2, between 512 bytes and 8192 bytes)
- Divide logical memory into fixed-sized blocks
  - Called *pages* (same size as frames)
- Keep track of all free frames
- To run a program of size *n* pages, need to find *n* free frames and load program
- Use a *page table* per process for translating logical to physical addresses
- Use a *frame table* to keep track of physical memory usage

Paging Address Translation Scheme

- Address generated by CPU is divided into
  - *Page number (p)* is an index into page table which contains base address of each frame in physical memory
  - *Page offset (d)* is combined with base address to define the physical memory address that is sent to the memory
- By using a page size that is a power of two, translating a logical address to a page number and offset is easy
  - Assume size of logical address space is 2^m and page size is 2^n
  - Then the low *n* bits of a logical address are the page offset and the high *m* - *n* bits are the page number

p	*d*
page number (m - n bits)	page offset (n bits)

Paging Address Translation Scheme

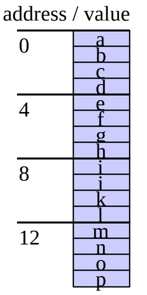


Paging Example

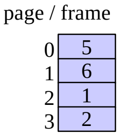
Example for 32-byte memory with 4-byte pages:

32 bytes = 2^5, $m = 5$, therefore we have a 5 bit logical address

4 bytes = 2^2, $n = 2$, the lower 2 bits are the offset



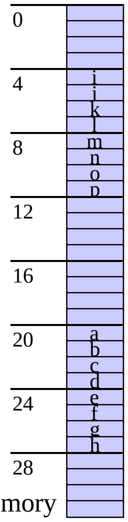
logical memory



page table

Translate logical address 13

- 13 = 01101 in binary
- d = 01 in binary; 1 in decimal
- p = 011 in binary; 3 in decimal (to get p simply right-shift logical address n times)
- physical address equals = page frame * page size + offset $2 * 4 + 1 = 9$



physical memory

Internal Fragmentation in Pages

- Memory cannot be allocated in blocks smaller than the page size
  - This leads to internal fragmentation since the last page frame for a process may not be completely full
  - On average fragmentation is one-half page per process
- This might suggest to use small page sizes
  - However, there is overhead involved in managing the page table and smaller pages means a bigger page table
  - When writing pages to disk, bigger is better too
  - Typical page size is between 2k to 8k