INTRODUCTION TO LINUX

LECTURE (9)

Elzahraa Hasan

Agenda

- File compression
- Directory compression
- Memory
- Process

File compression

- To compress a file we have two techniques:

- gzip →it is fast   0.002 sec
- Bzip2→save more space   0.003 sec 15% better space

Lab…compression

- ls –LR / >myfile …to create a file ^c
- Ls –lh myfile …..to get the file size

<u>The gzip technique:</u>

- Gzip myfile …to compress
- ls –lh myfile.gz … to check what is the size?
- Gunzip myfile.gz ….to expand
- ls –lh … to check

<u>The bzip2 technique:</u>

- bzip2 myfile …to compress
- ls –lh myfile.bz2 … to check what is the size?
- bunzip2 myfile.bz2 ….to expand
- ls –lh … to check

Lab…compression

- To calculate the execution time for both techniques:

<u>The gzip technique:</u>

- Time gzip myfile …to compress
- gunzip myfile.gz ….to expand

<u>The bzip2 technique:</u>

- Time bzip myfile …to compress
- bunzip2 myfile.bz2 ….to expand

Binding of Instructions and Data to Memory

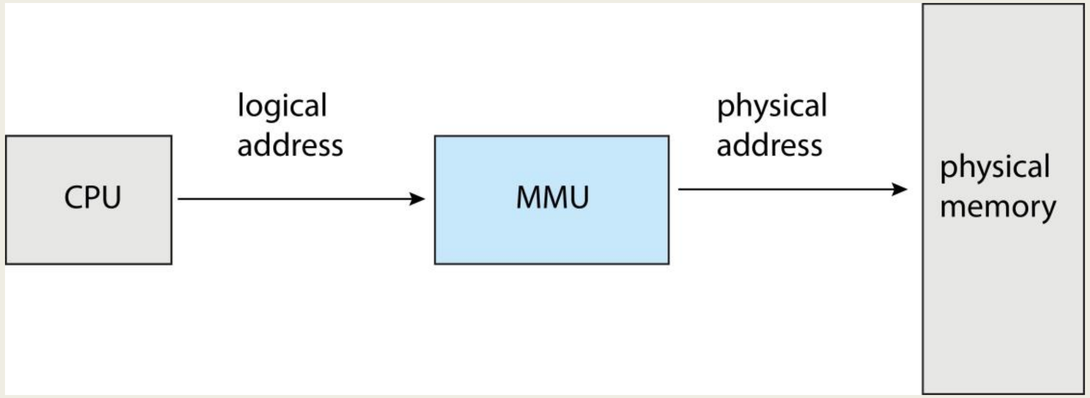
- Address binding of instructions and data to memory addresses can happen at three different stages
  - ***Compile time:*** *If memory location known a priori,* ***absolute code*** *can be generated; must recompile code if starting location changes*
  - ***Load time:*** *Must generate* ***relocatable 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)

Logical vs. Physical Address Space

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

Memory-Management Unit (MMU)

- Hardware device that at run time maps virtual to physical address



Memory-Management Unit (Cont.)

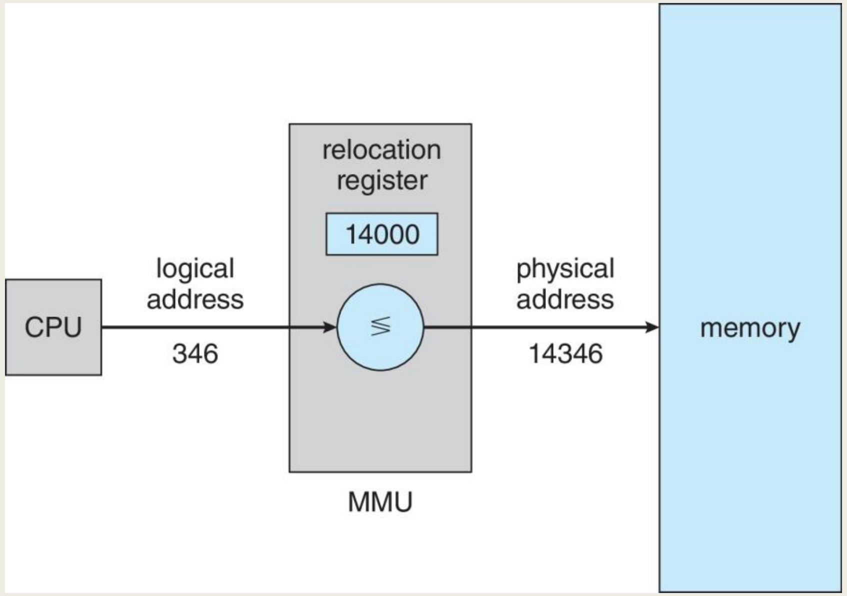
- Consider simple scheme. which is a generalization of the base-register scheme.
- *The base register now called* ***relocation register***
- The value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *real* physical addresses
  - *Execution-time binding occurs when reference is made to location in memory*
  - *Logical address bound to physical addresses*

Memory-Management Unit (Cont.)

Consider simple scheme. which is a generalization of the base-register scheme.

- *The base register now called* ***relocation register***

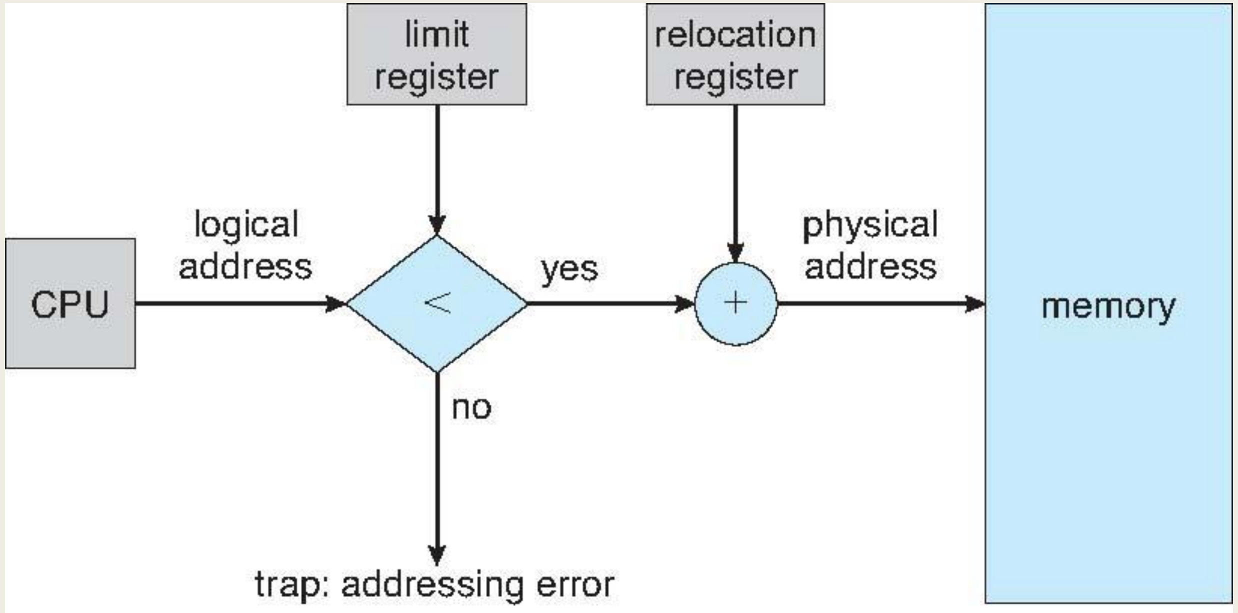
The value in the relocation register is added to every address generated by a user process at the time it is sent to memory



Contiguous Allocation (Cont.)

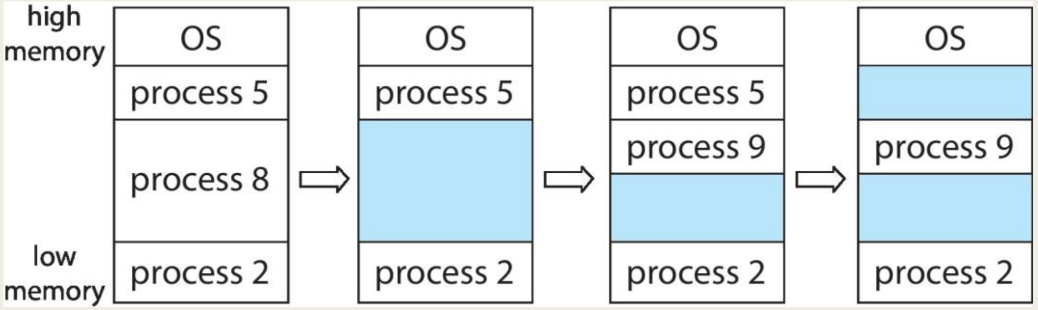
- Relocation registers used to protect user processes from each other, and from changing operating-system code and data
  - *Base register contains value of smallest physical address*
  - *Limit register contains range of logical addresses – each logical address must be less than the limit register*
  - *MMU maps logical address dynamically*
  - *Can then allow actions such as kernel code being **transient** and kernel changing size*

Hardware Support for Relocation and Limit Registers



Variable Partition

- Multiple-partition allocation
  - *Degree of multiprogramming limited by number of partitions*
  - ***Variable-partition*** *sizes for efficiency (sized to a given process' needs)*
  - ***Hole*** *– 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*
  - *Process exiting frees its partition, adjacent free partitions combined*
  - *Operating system maintains information about:*
    a) allocated partitions   b) free partitions (hole)



Dynamic Storage-Allocation Problem

How to satisfy a request of size ***n*** from a list of free 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
  - *Produces the largest leftover hole*

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

example

1. Given five memory partitions of 100Kb, 500Kb, 200Kb, 300Kb, 600Kb (in order), how would the first-fit, best-fit, and worst-fit algorithms place processes of 212 Kb, 417 Kb, 112 Kb, and 426 Kb (in order)? Which algorithm makes the most efficient use of memory?

First-fit:
212K is put in 500K partition
417K is put in 600K partition
112K is put in 288K partition (new partition 288K = 500K - 212K)
426K must wait

Best-fit:
212K is put in 300K partition
417K is put in 500K partition
112K is put in 200K partition
426K is put in 600K partition

Worst-fit:
212K is put in 600K partition
417K is put in 500K partition
112K is put in 388K partition
426K must wait

In this example, best-fit turns out to be the best.

Fragmentation

- External Fragmentation – total memory space exists to satisfy a request, but it is not contiguous
- Internal Fragmentation – allocated memory may be slightly larger than requested memory; this size difference is memory internal to a partition, but not being used
- First fit analysis reveals that given *N* blocks allocated, 0.5 *N* blocks lost to fragmentation
  - *1/3 may be unusable -> 50-percent rule*

Fragmentation (Cont.)

- 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*
    - Do I/O only into OS buffers
- Now consider that backing store has same fragmentation problems

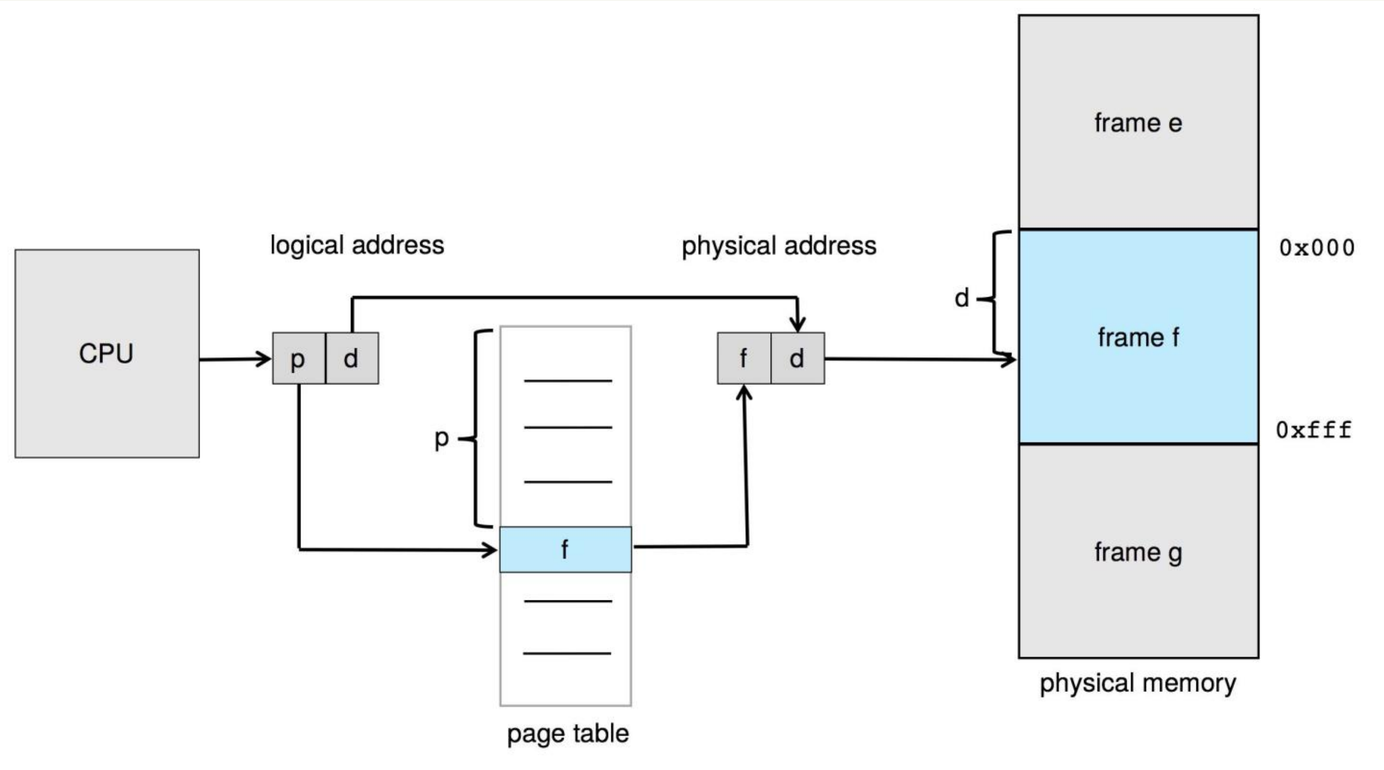
Address Translation Scheme

- Address generated by CPU is divided into:
  - ***Page number** (**p**) – used as an index into a **page table** which contains base address of each page in physical memory*
  - ***Page offset** (**d**) – combined with base address to define the physical memory address that is sent to the memory unit*

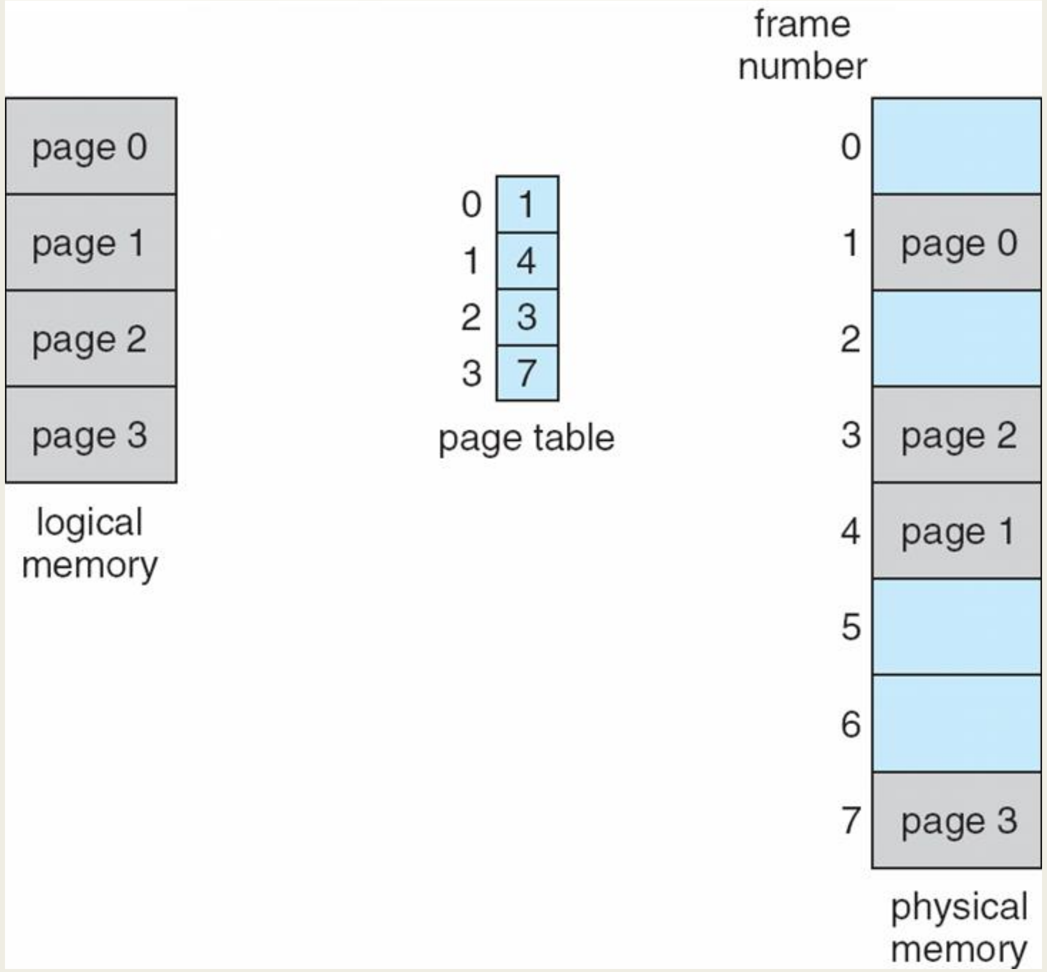
page number	page offset
p	d
m −n	n

  - *For given logical address space 2^m and page size 2^n*

Paging Hardware



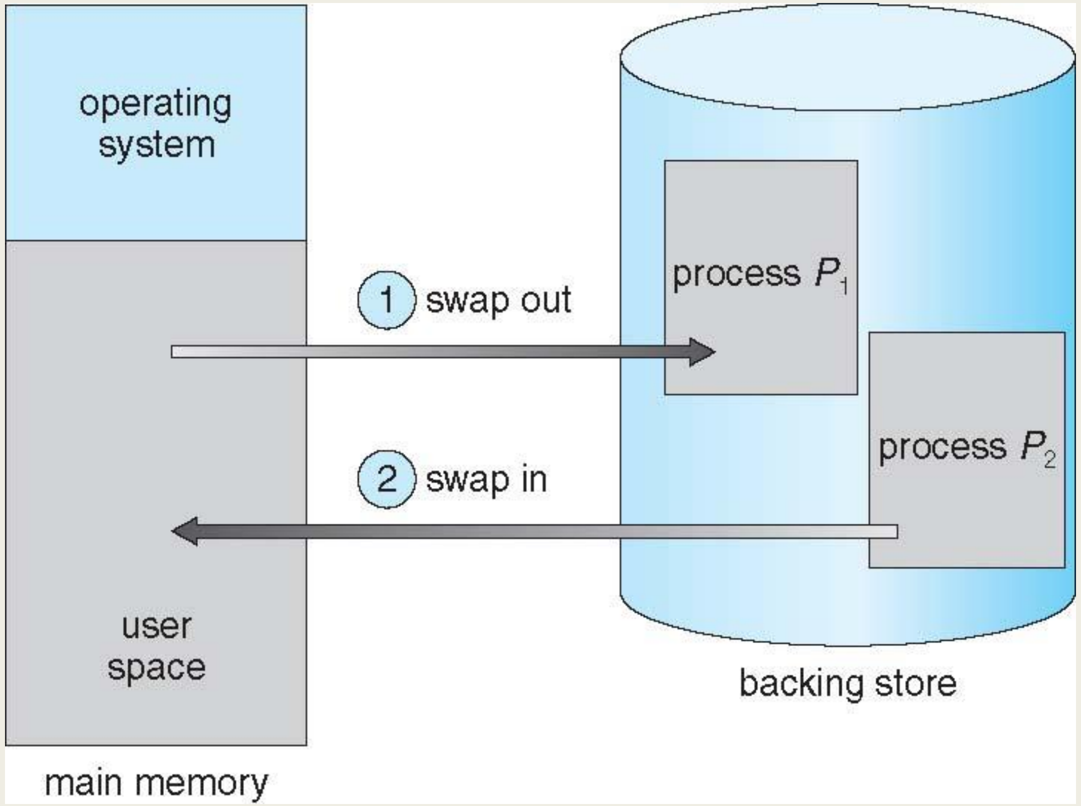
Paging Model of Logical and Physical Memory



Swapping

- A process can be **swapped** temporarily out of memory to a backing store, and then brought **back** into memory for continued execution
  - *Total physical memory space of processes can exceed physical memory*
- **Backing store** – fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images
- **Roll out, roll in** – swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- System maintains a **ready queue** of ready-to-run processes which have memory images on disk

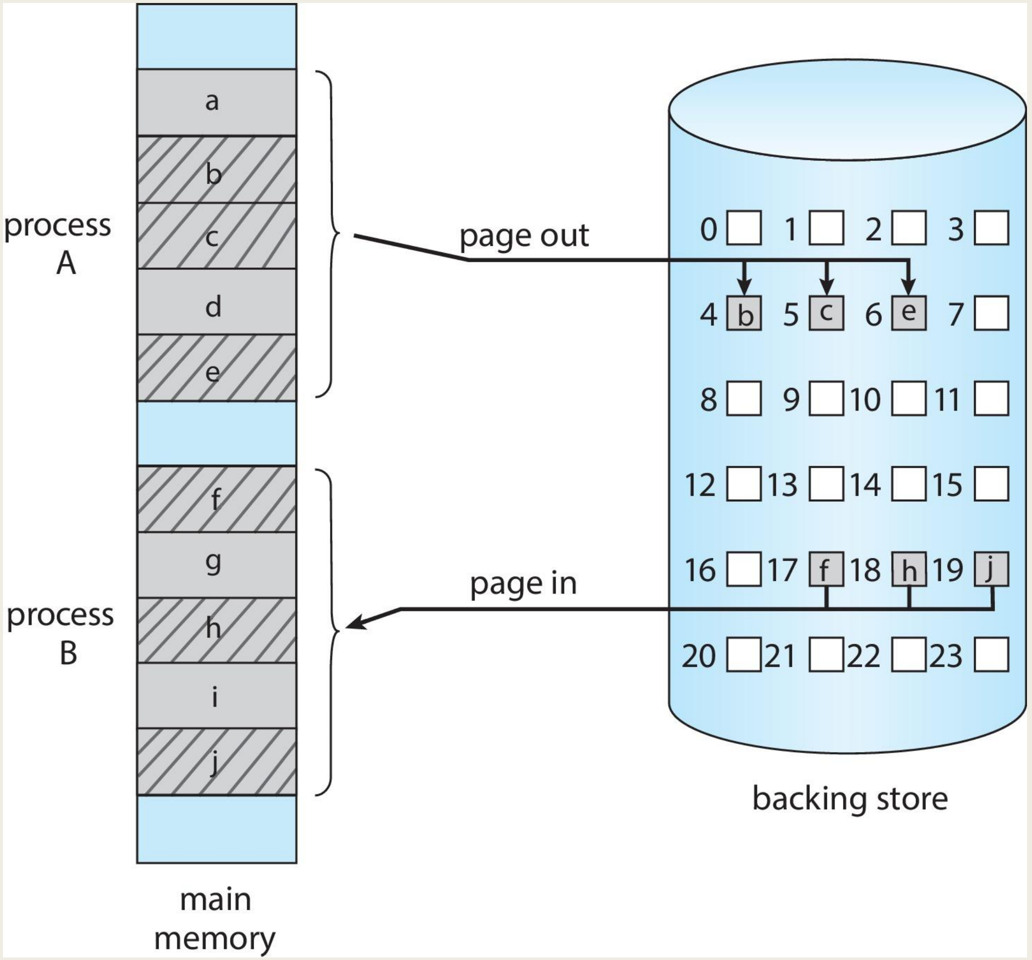
Schematic View of Swapping



Context Switch Time including Swapping

- If next processes to be put on CPU is not in memory, need to swap out a process and swap in target process
- Context switch time can then be very high
- 100MB process swapping to hard disk with transfer rate of 50MB/sec
  - *Swap out time of 2000 ms*
  - *Plus swap in of same sized process*
  - *Total context switch swapping component time of 4000ms (4 seconds)*
- Can reduce if reduce size of memory swapped – by knowing how much memory really being used
  - *System calls to inform OS of memory use via `request_memory()` and `release_memory()`*

Swapping with Paging



Process Concept

- An operating system executes a variety of programs that run as a process.
- **Process** – a program in execution; process execution must progress in sequential fashion. No parallel execution of instructions of a single process
- Multiple parts
  - *The program code, also called **text section***
  - *Current activity including **program counter**, processor registers*
  - ***Stack** containing temporary data*
    - Function parameters, return addresses, local variables
  - ***Data section** containing global variables*
  - ***Heap** containing memory dynamically allocated during run time*

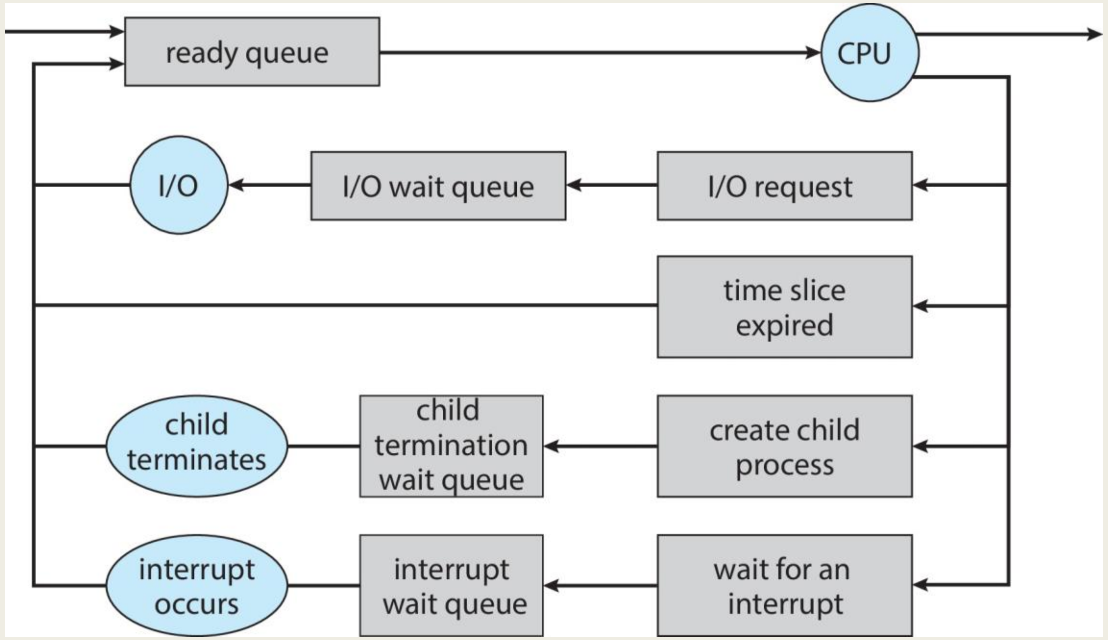
Process Concept

- Program is **passive** entity stored on disk (executable file); process is **active**
  - *Program becomes process when an executable file is loaded into memory*
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can be several processes
  - *Consider multiple users executing the same program*
  - *./filename*

Process Scheduling

- **Process scheduler** selects among available processes for next execution on CPU core
- Goal -- Maximize CPU use, quickly switch processes onto CPU core
- Maintains **scheduling queues** of processes
  - ***Ready queue*** *– set of all processes residing in main memory, ready and waiting to execute*
  - ***Wait queues*** *– set of processes waiting for an event (i.e., I/O)*
  - *Processes migrate among the various queues*

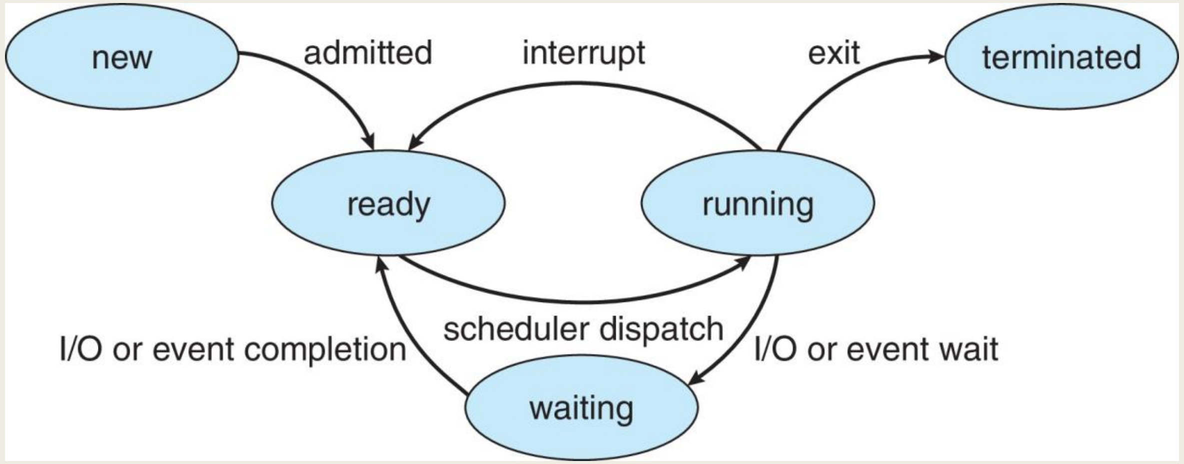
Representation of Process Scheduling



Process State

- As a process executes, it changes **state**
  - ***New:*** *The process is being created*
  - ***Running:*** *Instructions are being executed*
  - ***Waiting:*** *The process is waiting for some event to occur*
  - ***Ready:*** *The process is waiting to be assigned to a processor*
  - ***Terminated:*** *The process has finished execution*

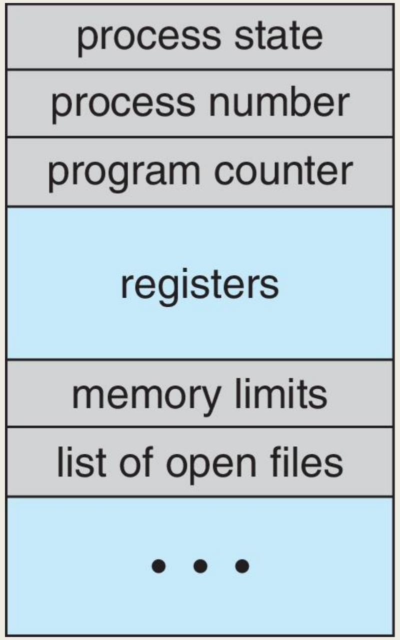
Diagram of Process State



Process Control Block (PCB)

Information associated with each process(also called task control block)

- Process state – running, waiting, etc.
- Program counter – location of instruction to next execute
- CPU registers – contents of all process-centric registers
- CPU scheduling information- priorities, scheduling queue pointers
- Memory-management information – memory allocated to the process
- Accounting information – CPU used, clock time elapsed since start, time limits
- I/O status information – I/O devices allocated to process, list of open files



Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in the PCB
- Context-switch time is pure overhead; the system does no useful work while switching
  - *The more complex the OS and the PCB ➔ the longer the context switch*
- Time dependent on hardware support
  - *Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once*

CPU Switch From Process to Process

A **context switch** occurs when the CPU switches from one process to another.