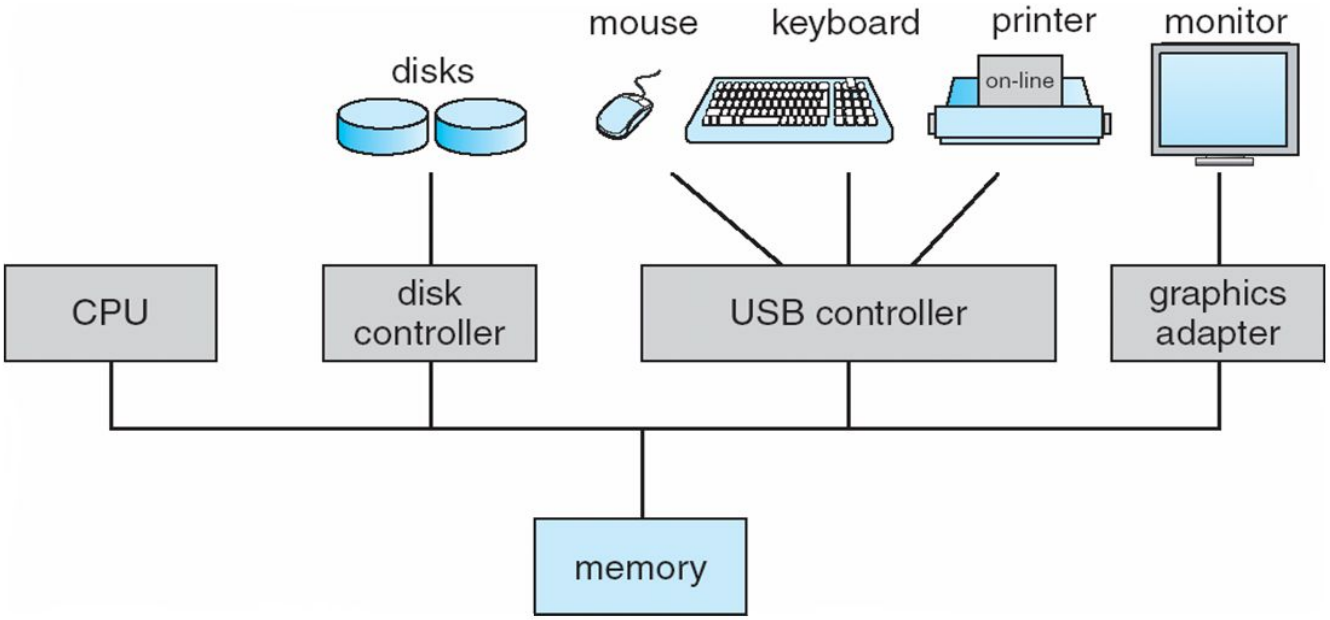
ICS 143 - Principles of Operating Systems

Operating Systems - Review

Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

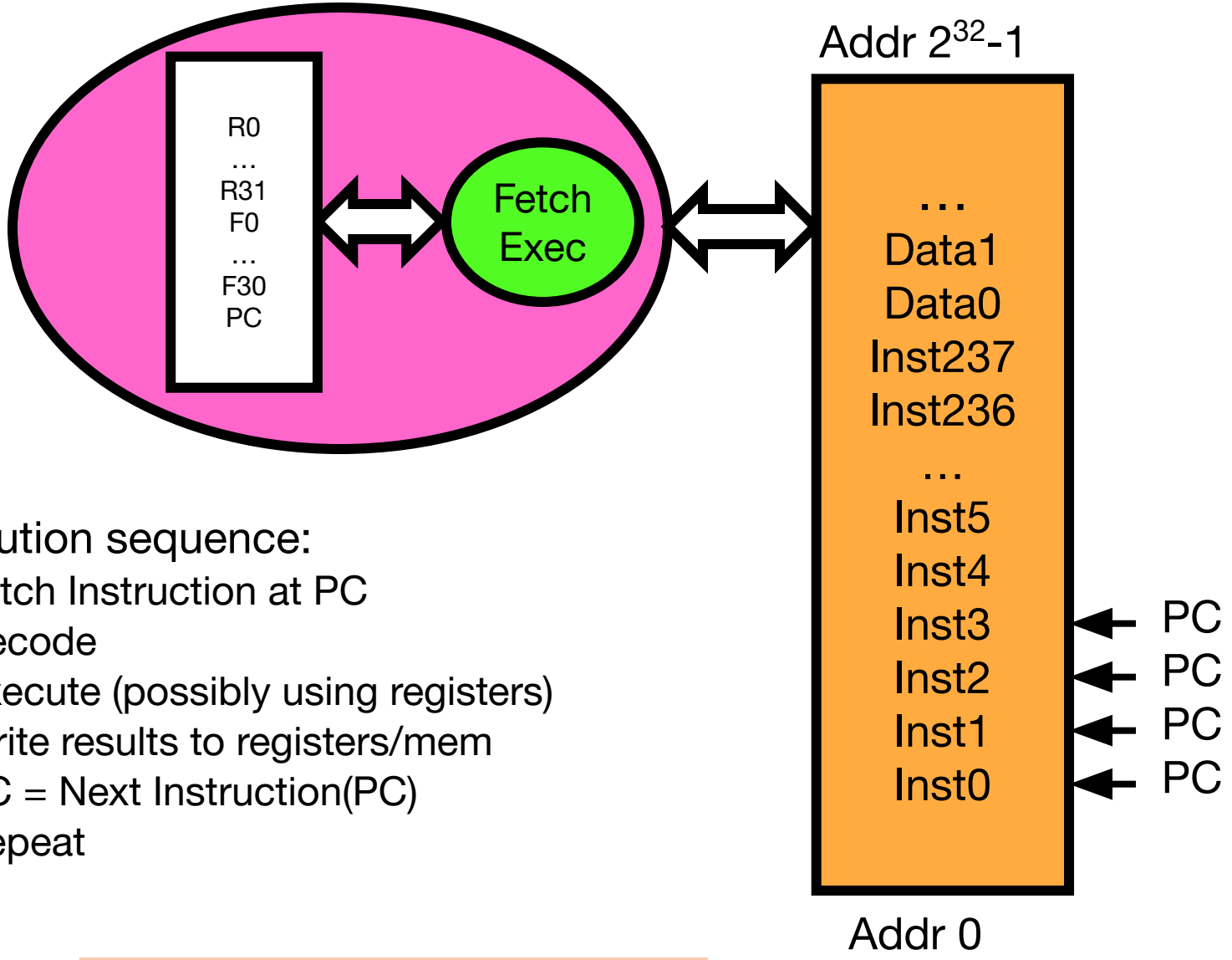
What is an Operating System?

- OS is the software that acts an intermediary between the user applications and computer hardware.



Computer System Organization

CPU execution



- Execution sequence:
  - Fetch Instruction at PC
  - Decode
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

From Berkeley OS course

Interrupts

- Interrupt transfers control to the interrupt service routine
  - Interrupt Service Routine: Segments of code that determine action to be taken for interrupt.
- Interrupt Vector Table
- different interrupt handlers will be executed for different interrupts

Interrupt Number	Address
0	0003h
1	000Bh
2	0013h
3	001Bh
4	0023h
5	002Bh
6	0033h
7	003Bh
8	0043h
9	004Bh
10	0053h
11	005Bh
12	0063h
13	006Bh
14	0073h
15	007Bh

Interrupt Number	Address
16	0083h
17	008Bh
18	0093h
19	009Bh
20	00A3h
21	00ABh
22	00B3h
23	00BBh
24	00C3h
25	00CBh
26	00D3h
27	00DBh
28	00E3h
29	00EBh
30	00F3h
31	00FBh

- Interrupt Handling
- OS preserves the state of the CPU
  - stores registers and the program counter (address of interrupted instruction).
  - Incoming interrupts are disabled while another interrupt is being processed to prevent a *lost interrupt*.

I/O **processing**

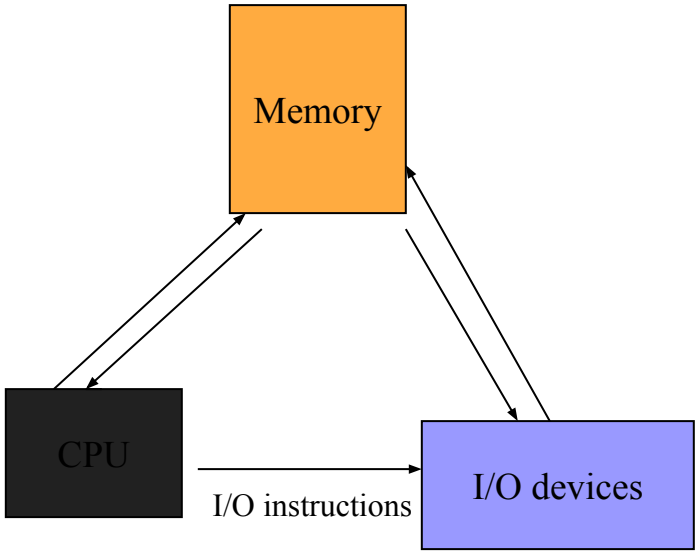
- Synchronous I/O
  - After I/O is requested, *wait* until I/O is done. Program will be idle.
- Asynchronous I/O
  - After I/O is requested, control returns to user program without waiting for I/O completion.

DMA

- Used for high speed I/O devices able to transmit information at close to memory speeds.

- Device controller transfers blocks of data from buffer storage directly to main memory without CPU intervention.

- Only one interrupt is generated per block, rather than one per byte (or word)

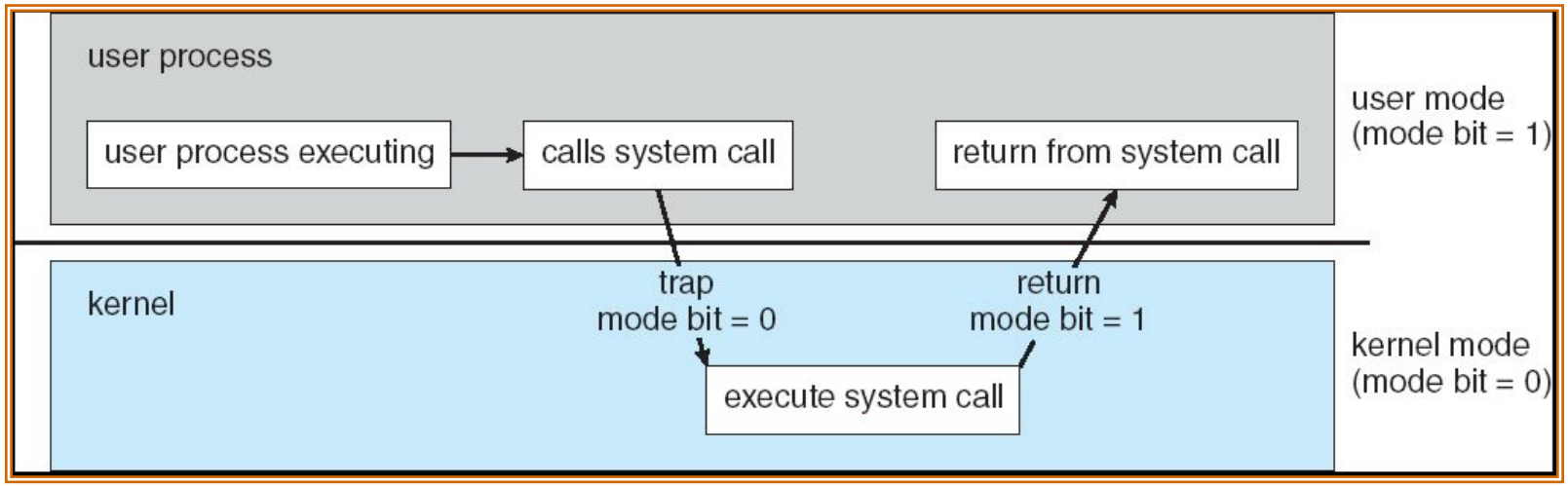


Dual-mode operation

- Provide hardware support to differentiate between at least two modes of operation:
  1. User mode -- execution done on behalf of a user.
  2. Kernel mode (monitor/supervisor/system mode) -- execution done on behalf of operating system.
- "Privileged" instructions are only executable in the kernel mode
- Executing privileged instructions in the user mode "traps" into the kernel mode
- Trap is a software generated interrupt caused either by an error or a user request

System Calls

- User code can issue a syscall, which causes a trap
- Kernel handles the syscall

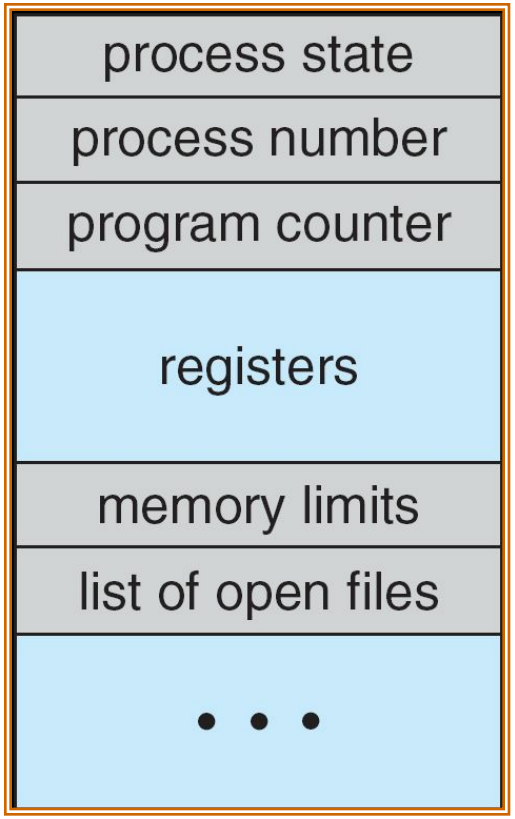


Process Abstraction

- Process: an *instance* of a program, running with limited rights
  - Thread: a sequence of instructions within a process
    - Potentially many threads per process (for now 1:1)
  - Address space: set of rights of a process
    - Memory that the process can access
    - Other permissions the process has (e.g., which system calls it can make, what files it can access)

Enabling Concurrency and Protection: Multiplex processes

- Only one process (PCB) active at a time
  - Current state of process held in PCB:
    - "snapshot" of the execution and protection environment
  - Process needs CPU, resources
- Give out CPU time to different processes (Scheduling):
  - Only one process "running" at a time
  - Give more time to important processes
- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
    - E.g. Memory Mapping: Give each process their own address space



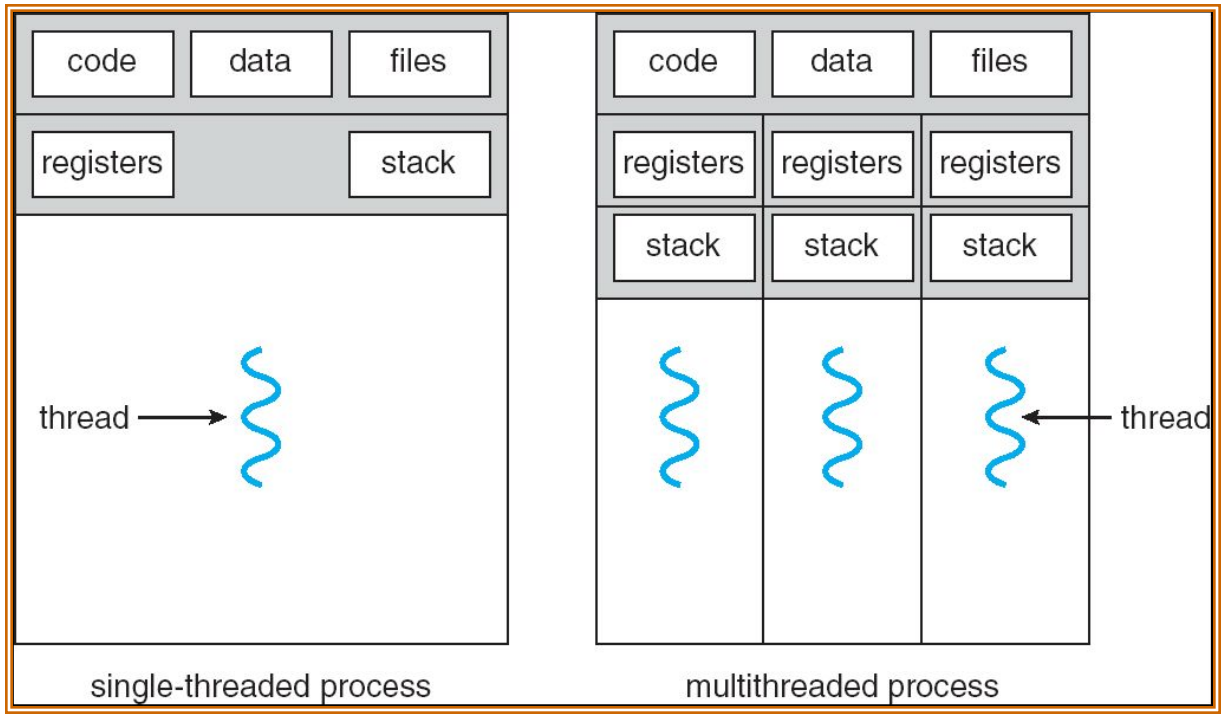
Process Control Block

Threads

- Processes do not share resources well
  - high context switching overhead
- Idea: Separate concurrency from protection

- Multithreading: *a single program made up of a number of different concurrent activities*
- A thread (or lightweight process)
  - basic unit of CPU utilization; it consists of:
    - program counter, register set and stack space
  - A thread shares the following with peer threads:
    - code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
- Heavyweight process is a task with one thread.

Single and Multithreaded Processes



- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system

Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)
- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including program counter)
  - Execution stack
    - Parameters, Temporary variables
    - return PCs are kept while called procedures are executing

Threads (cont.)

- Thread context switch still requires a register set switch, but no memory management related work!
- Thread states -
  - *ready, blocked, running, terminated*
- Threads share CPU and only one thread can run at a time.
- No protection among threads.

Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).

Kernel Threads

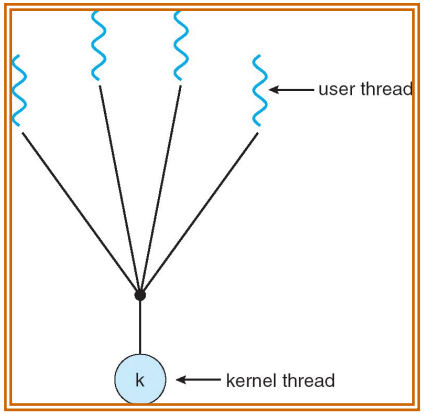
- Supported by the Kernel
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things
- Downside of kernel threads: a bit expensive
  - Need to make a crossing into kernel mode to schedule
- Examples
  - Windows XP/2000, Solaris, Linux,Tru64 UNIX, Mac OS X, Mach, OS/2

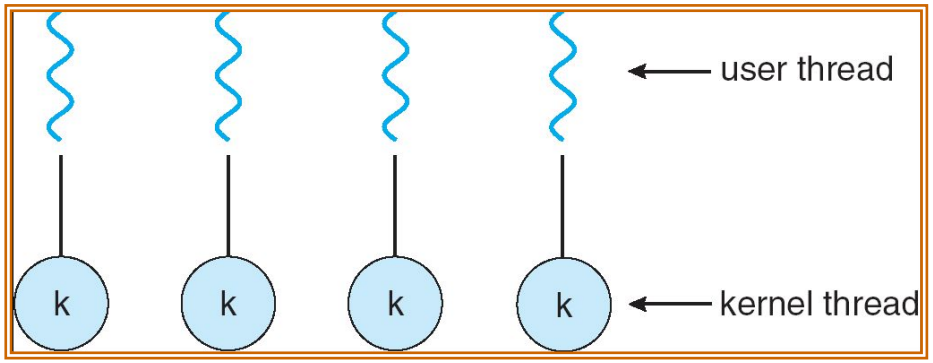
User Threads

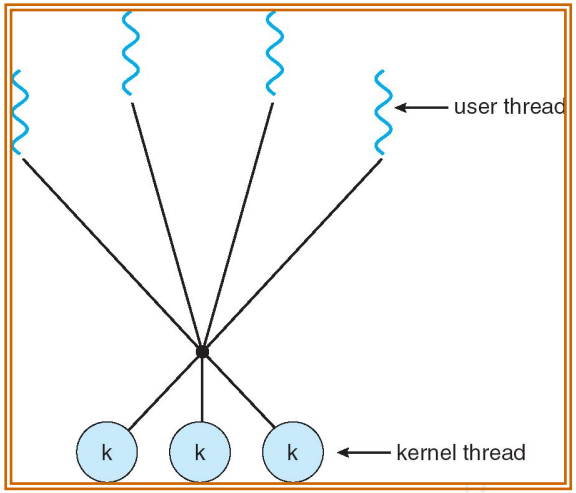
- Supported above the kernel, via a set of library calls at the user level.
  - Thread management done by user-level threads library
    - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())
- Advantages
  - Cheap, Fast
    - Threads do not need to call OS and cause interrupts to kernel
- Disadv: If kernel is single threaded, system call from any thread can block the entire task.
- Example thread libraries:
  - POSIX Pthreads, Win32 threads, Java threads

Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

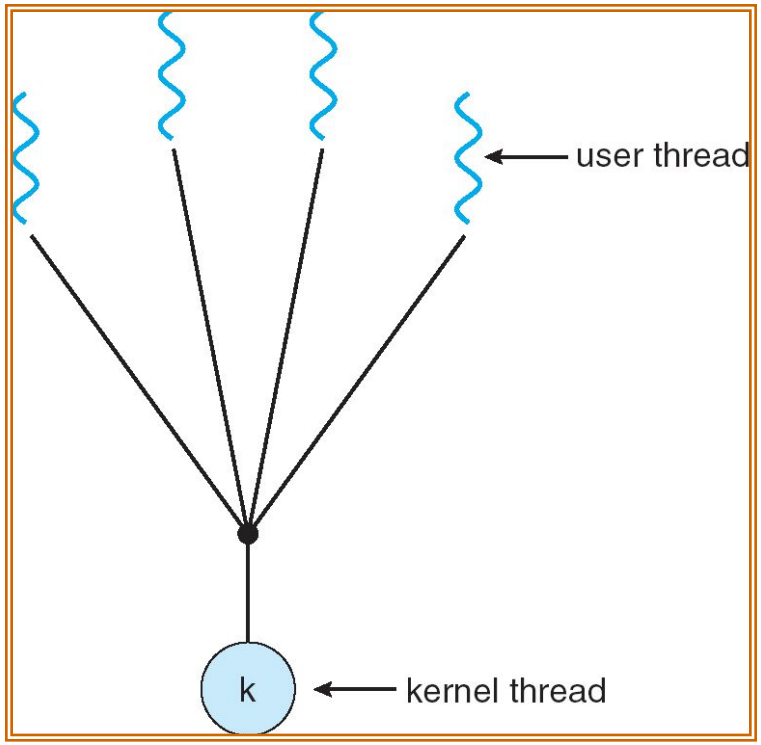






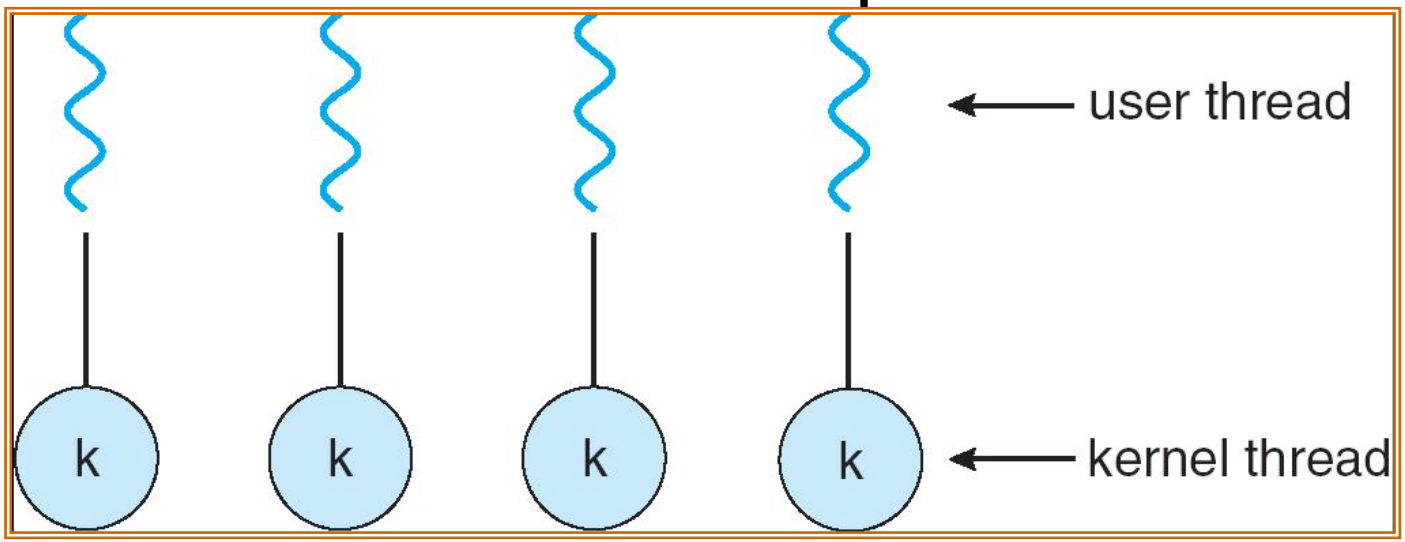
Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One-to-One

- Each user-level thread maps to kernel thread

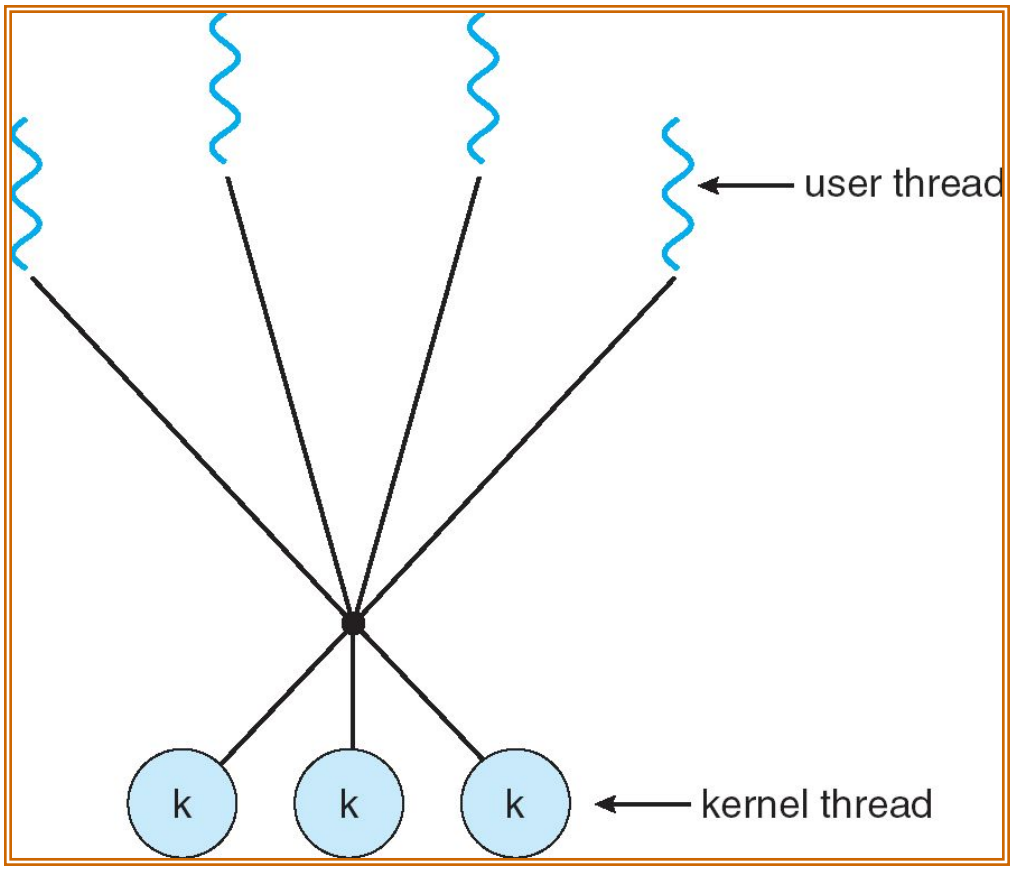


Examples

- Windows NT/XP/2000; Linux; Solaris 9 and later

Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in Chapter 5.

Interprocess Communication – Message Passing

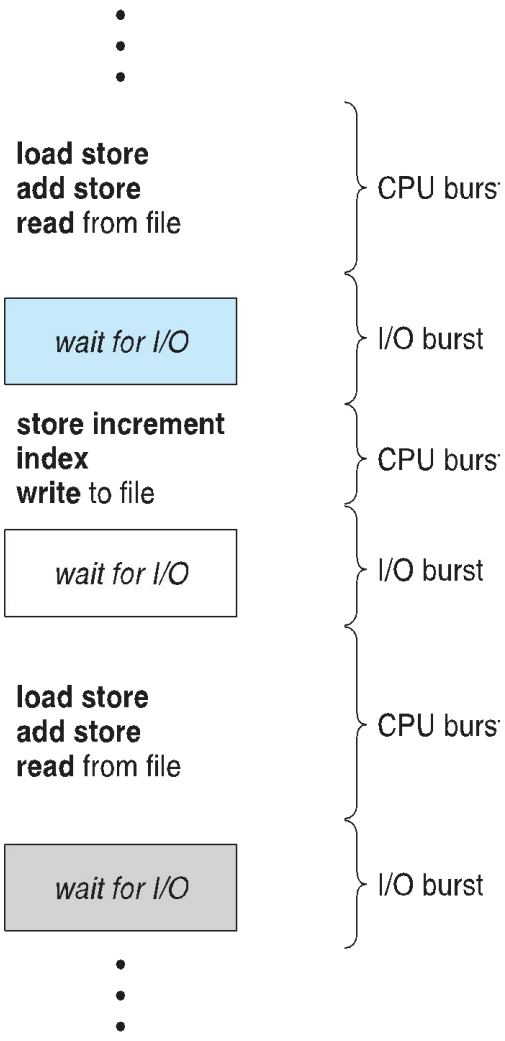
- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **send**(*message*)
  - **receive**(*message*)
- The *message* size is either fixed or variable

Schedulers

- Long-term scheduler (or job scheduler) -
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes); may be slow.
  - controls the degree of multiprogramming
- Short term scheduler (or CPU scheduler) -
  - selects which process should execute next and allocates CPU.
  - invoked very frequently (milliseconds) - must be very fast
- Medium Term Scheduler
  - swaps out process temporarily
  - balances load for better throughput

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming.
- CPU-I/O Burst Cycle
  - Process execution consists of a cycle of CPU execution and I/O wait.



CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.
  - Non-preemptive Scheduling
    - Once CPU has been allocated to a process, the process keeps the CPU until
      - Process exits OR
      - Process switches to waiting state
  - Preemptive Scheduling
    - Process can be interrupted and must release the CPU.
      - Need to coordinate access to shared data

Scheduling Criteria

- CPU Utilization
  - Keep the CPU and other resources as busy as possible
- Throughput
  - # of processes that complete their execution per time unit.
- Turnaround time
  - amount of time to execute a particular process from its entry time.
- Waiting time

  amount of time a process has been waiting in the ready queue.
- Response Time (in a time-sharing environment)

  amount of time it takes from when a request was submitted until the first response is produced, NOT output.

Optimization Criteria

- Max CPU Utilization
- Max Throughput
- Min Turnaround time
- Min Waiting time
- Min response time

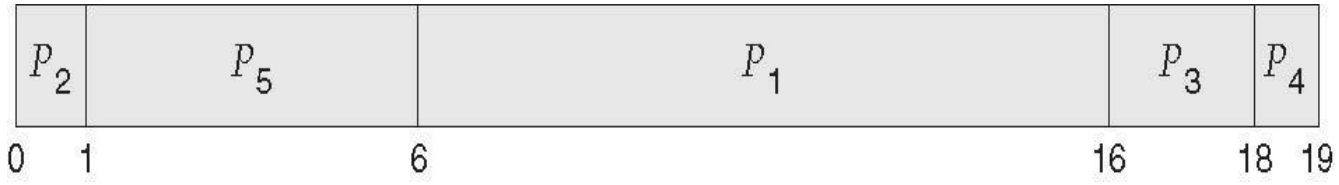
CPU Scheduling Algorithms

- **First Come First Serve or FIFO**
  - **Convoy Effect**
- **Shortest Job First (Optimal)**
  - **Non - Preemptive**
  - **Shortest Remaining Time First - SRTF (Preemptive)**
- **Priority**
- **Round Robin**
- **Multilevel Queue**
- **Multilevel Feedback Queue**

Priority Scheduling - Non-preemptive

ProcessA	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

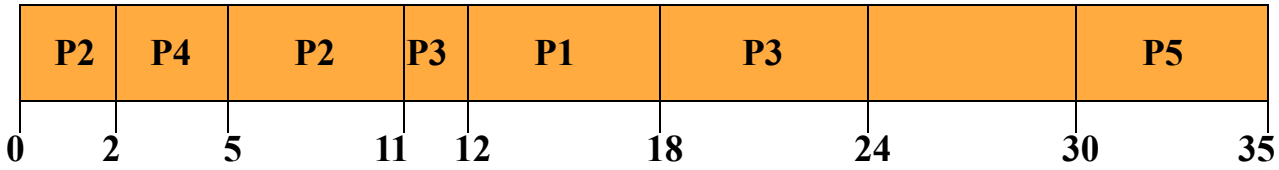


- Average waiting time = 8.2 msec

Priority Scheduling - Preemptive

ProcessA	Burst Time	Priority	Arrival Time
P_1	6	3	12
P_2	8	2	0
P_3	7	4	4
P_4	3	1	2
P_5	5	5	30

- Gantt Chart



- Average waiting time = [0+3+(7+6)+0+0)]/5 = 16/5 = 3.2 msec
- Average turnaround time = (6 + 11 + 20 + 3 + 5)/5 = 45/5 = 9 msec
- Average response time (assuming immediate response by a process when executed) = (0 + 0 + 7 + 0 + 0) / 5 = 1.4 msec
- CPU utilization = 29 / 35 = 0.83 = 83%
- Throughput = 5 / 35 = 0.14 #process/msec

Critical Section Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$} competing to access shared data
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Solution: Critical Section Problem - Requirements

- **Mutual Exclusion**
  - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
- **Progress**
  - If no process is executing in its critical section and there exists some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.
- **Bounded Waiting**
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Solution: Critical Section Problem - Requirements

- Assume that each process executes at a nonzero speed.
- No assumption concerning relative speed of the n processes.

Algorithm 1 (Similarly Algos 2,3)

- Shared Variables:
  - **int turn = 0**;
  - (turn == i) means that P_i can enter its critical section
- Process P_i

```
do {
    while (turn == j);

        critical section

    turn = j;

        remainder section

} while (true);
```

Satisfies mutual exclusion, but not progress.

Algorithm 4

- Combined Shared Variables of algorithms 1 and 2
- Process Pi

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);
        critical section
    flag[i] = false;
        remainder section
} while (true);
```

YES!!! Meets all three requirements, solves the critical section problem for 2 processes.

This is called the “Peterson’s solution”.

Bakery Algorithm

- Critical section for n processes
  - Before entering its critical section, process receives a number. Holder of the smallest number enters critical section.
  - If processes Pi and Pj receive the same number,
    - if i <= j, then Pi is served first; else Pj is served first.
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,3,4,4,5,5

Bakery Algorithm (cont.)

- Notation -
  - Lexicographic order(ticket#, process id#)
    - $(a,b) < (c,d)$ if $(a<c)$ or if $((a=c)$ and $(b < d))$
    - max(a_0,....a_{n-1}) is a number, k, such that $k >= a_i$ for $i = 0,\ldots,n-1$
- Shared Data

  var *choosing*: **array**[0..*n*-1] **of** *boolean*;(initialized to *false*)

  number: **array**[0..*n*-1] **of** *integer*; (initialized to 0)

Bakery Algorithm (cont.)

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1],…,number[n-1]) +1;
    choosing[i] := false;
    for j := 0 to n-1
       do begin
       while choosing[j] do no-op;
             while number[j] <> 0
                   and (number[j] ,j) < (number[i],i) do no-op;
       end;
        critical section
     number[i]:= 0;
        remainder section
until false;
```

Supporting Synchronization

Programs	*Shared Programs*
Higher-level API	*Locks Semaphores Monitors Send/Receive CCregions*
Hardware	*Load/Store Disable Ints Test&Set Comp&Swap*

- We are going to implement various synchronization primitives using atomic operations
  - Everything is pretty painful if only atomic primitives are load and store
  - Need to provide inherent support for synchronization at the hardware level
  - Need to provide primitives useful at software/user level

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

Semaphore

- Semaphore *S* - integer variable (non-negative)
  - used to represent number of abstract resources
- Can only be accessed via two indivisible (atomic) operations

  wait (*S*): **while** (*S* <= 0);
  S--;

  signal (S): *S*++;
  - *P* or *wait* used to acquire a resource, waits for semaphore to become positive, then decrements it by 1
  - *V* or *signal* releases a resource and increments the semaphore by 1, waking up a waiting P, if any
  - If *P* is performed on a *count* <= 0, process must wait for *V* or the release of a resource.

P():"*proberen*" (to test) ; V() "*verhogen*" (to increment) in Dutch

Example: Critical Section for n Processes

- Shared variables

  semaphore mutex;
  initially mutex = 1

- Process P_i

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (true);
```

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* execute in P_i
  - *As in Homework problem*
- Use semaphore *flag* initialized to 0
- Code:

P_i	P_j
⋮	⋮
A	*wait*(*flag*)
signal(*flag*)	*B*

Problem...

- Locks prevent conflicting actions on shared data
  - Lock before entering critical section and before accessing shared data
  - Unlock when leaving, after accessing shared data
  - Wait if locked
- All Synchronization involves waiting
  - **Busy Waiting,** uses CPU that others could use. This type of semaphore is called a *spinlock*.
- For longer runtimes, need to modify P and V so that processes can *block* and *resume.*

Synchronization Hardware

- Test and modify the content of a word atomically - Test-and-set instruction

```
function Test-and-Set (var target: boolean): boolean;
  begin
     Test-and-Set := target;
      target := true;
  end;
```

- Similarly "SWAP" instruction

Mutual Exclusion with Test-and-Set

- Shared data: var lock: boolean (initially false)
- Process Pi

```
repeat
        while Test-and-Set (lock) do no-op;
                critical section
              lock := false;
                remainder section
    until false;
```

Classical Problems of Synchronization

- Bounded Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded Buffer Problem

- Producer process - creates filled buffers

```
repeat
        ...
   produce an item in nextp
      ...
   wait (empty);
   wait (mutex);
   ...
   add nextp to buffer
   ...
   signal (mutex);
   signal (full);
until false;
```

Consumer process - Empties filled buffers

```
repeat
        wait (full );
     wait (mutex);
        ...
     remove an item
  from buffer  to
  nextc
        ...
     signal (mutex);
     signal (empty);
        ...
        consume the
  next item in nextc
        ...
until false;
```

Bounded Buffer Problem

- Consumer process - Empties filled buffers

```
repeat
        wait (full );
    wait (mutex);
      …
    remove an item from buffer  to  nextc
      …
    signal (mutex);
    signal (empty);
      …
      consume the next item in nextc
      …
until false;
```

Discussion

- ASymmetry?
  - Producer does: `P(empty), V(full)`
  - Consumer does: `P(full), V(empty)`
- Is order of P's important?
  - Yes! Can cause deadlock
- Is order of V's important?
  - No, except that it might affect scheduling efficiency

Readers-Writers Problem

- Shared Data

  var *mutex, wrt*: *semaphore* (=1);
  readcount: *integer* (= 0);

- Writer Process

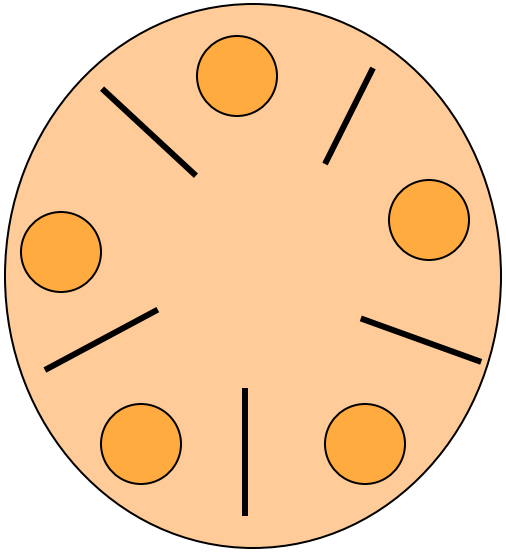
  wait(*wrt*);
  …
  writing is performed
  …
  signal(*wrt*);

Readers-Writers Problem

- Reader process

```
wait(mutex);
   readcount := readcount +1;
   if readcount = 1 then wait(wrt);
signal(mutex);
            ...
              reading is performed
            ...
wait(mutex);
   readcount := readcount - 1;
   if readcount = 0 then signal(wrt);
signal(mutex);
```

Dining-Philosophers Problem



Shared Data

var *chopstick*: **array** [0..4] **of** *semaphore* (=1 initially);

Deadlocks

- System Model
  - Resource allocation graph, claim graph (for avoidance)
- Deadlock Characterization
  - Conditions for deadlock - mutual exclusion, hold and wait, no preemption, circular wait.
- Methods for handling deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance
  - Deadlock Detection
  - Recovery from Deadlock
  - Combined Approach to Deadlock Handling

Deadlock Prevention

- If any one of the conditions for deadlock (with reusable resources) is denied, deadlock is impossible.
- Restrain ways in which requests can be made
  - Mutual Exclusion - cannot deny (important)
  - Hold and Wait - guarantee that when a process requests a resource, it does not hold other resources.
  - No Preemption
    - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process releases the resources currently being held.
  - Circular Wait
    - Impose a total ordering of all resource types.

Deadlock Avoidance

- Requires that the system has some additional *a priori* information available.
  - Simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need.
- Computation of Safe State
  - When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state. Sequence <P1, P2, …Pn> is safe, if for each Pi, the resources that Pi can still request can be satisfied by currently available resources + resources held by Pj with j<i.
  - Safe state - no deadlocks, unsafe state - possibility of deadlocks
  - Avoidance - system will never reach unsafe state.

Algorithms for Deadlock Avoidance

- Resource allocation graph algorithm
  - only one instance of each resource type
- Banker's algorithm
  - Used for multiple instances of each resource type.
  - Data structures required
    - Available, Max, Allocation, Need
  - Safety algorithm
  - resource request algorithm for a process.

Memory Management

- Main Memory is an array of addressable words or bytes that is quickly accessible.
- Main Memory is volatile.
- OS is responsible for:
    - Allocate and deallocate memory to processes.
    - Managing multiple processes within memory - keep track of which parts of memory are used by which processes. Manage the sharing of memory between processes.
    - Determining which processes to load when memory becomes available.

Binding of instructions and data to memory

- Address binding of instructions and data to memory addresses can happen at three different stages.
  - Compile time, Load time, Execution time
- MMU - Memory Management Unit
  - Hardware device that maps virtual to physical address.

Contiguous Allocation

- Divides Main memory usually into two partitions
  - Resident Operating System, usually held in low memory with interrupt vector and User processes held in high memory.
- Single partition allocation
  - Relocation register scheme used to protect user processes from each other, and from changing OS code and data
- Multiple partition allocation
  - holes of various sizes are scattered throughout memory. When a process arrives, it is allocated memory from a hole large enough to accommodate it.
  - Variation: Fixed partition allocation

Dynamic Storage Allocation Problem

- How to satisfy a request of size n from a list of free holes.
  - First-fit
  - Best-fit
  - Worst-fit
- 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.
  - Reduce external fragmentation by compaction

Paging

- Logical address space of a process can be non-contiguous;
  - process is allocated physical memory wherever the latter is available.
  - Divide physical memory into fixed size blocks called ***frames***
    - size is power of 2, 512 bytes - 8K
  - Divide logical memory into same size blocks called ***pages.***
    - Keep track of all free frames.
    - To run a program of size n pages, find n free frames and load program.
  - Set up a page table to translate logical to physical addresses.
  - Note:: Internal Fragmentation possible!!

Page Table Implementation

- Page table is kept in main memory
  - Page-table base register (PTBR) points to the page table.
  - Page-table length register (PTLR) indicates the size of page table.
- Every data/instruction access requires 2 memory accesses.
  - One for page table, one for data/instruction
  - Two-memory access problem solved by use of special fast-lookup hardware cache (i.e. cache page table in registers)
    - associative registers or translation look-aside buffers (TLBs)

Paging Methods

- Multilevel Paging
  - Each level is a separate table in memory
  - converting a logical address to a physical one may take 4 or more memory accesses.
  - Caching can help performance remain reasonable.
- Inverted Page Tables
  - One entry for each real page of memory. Entry consists of virtual address of page in real memory with information about process that owns page.
- Shared Pages
  - Code and data can be shared among processes. Reentrant (non self-modifying) code can be shared. Map them into pages with common page frame mappings

Segmentation

- Memory Management Scheme that supports user view of memory.
- A program is a collection of segments.
- A segment is a logical unit such as
    - main program, procedure, function
    - local variables, global variables,common block
    - stack, symbol table, arrays
  - Protect each entity independently
  - Allow each segment to grow independently
  - Share each segment independently

Segmented Paged Memory

- Segment-table entry contains not the base address of the segment, but the base address of a page table for this segment.
  - Overcomes external fragmentation problem of segmented memory.
  - Paging also makes allocation simpler; time to search for a suitable segment (using best-fit etc.) reduced.
  - Introduces some internal fragmentation and table space overhead.
- Multics - single level page table
- IBM OS/2 - OS on top of Intel 386
  - uses a two level paging scheme

Virtual Memory

- Virtual Memory
  - Separation of user logical memory from physical memory.
  - Only *PART* of the program needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Need to allow pages to be swapped in and out.
- Virtual Memory can be implemented via
  - Paging
  - Segmentation

Demand Paging

- Bring a page into memory only when it is needed.
  - Less I/O needed
  - Less Memory needed
  - Faster response
  - More users
- The first reference to a page will trap to OS with a page fault.
- OS looks at another table to decide
  - Invalid reference - abort
  - Just not in memory.

Page Replacement

- Prevent over-allocation of memory by modifying page fault service routine to include page replacement.
- Use modify(dirty) bit to reduce overhead of page transfers - only modified pages are written to disk.
- Page replacement
    - large virtual memory can be provided on a smaller physical memory.

Page Replacement Strategies

- The Principle of Optimality
  - Replace the page that will not be used again the farthest time into the future.
- Random Page Replacement
  - Choose a page randomly
- FIFO - First in First Out
  - Replace the page that has been in memory the longest.
- LRU - Least Recently Used
  - Replace the page that has not been used for the longest time.
  - LRU Approximation Algorithms - reference bit, second-chance etc.
- LFU/MFU - Least/Most Frequently Used
  - Replace the page that is used least/most often.

Allocation of Frames

- Single user case is simple
  - User is allocated any free frame
- Problem: Demand paging + multiprogramming
  - Each process needs minimum number of pages based on instruction set architecture.
  - Two major allocation schemes:
    - Fixed allocation - (1) equal allocation (2) Proportional allocation.
    - Priority allocation - May want to give high priority process more memory than low priority process.

Thrashing

- If a process does not have enough pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - OS thinks that it needs to increase the degree of multiprogramming
  - Another process is added to the system.
  - System throughput plunges...
- *Thrashing*
  - A process is busy swapping pages in and out.
  - In other words, a process is spending more time paging than executing.

Working Set Model

- Δ ≡ working-set window
  - a fixed number of page references, e.g. 10,000 instructions
  - WSSj (working set size of process Pj) - total number of pages referenced in the most recent Δ (varies in time)
    - If Δ too small, will not encompass entire locality.
    - If Δ too large, will encompass several localities.
    - If Δ = ∞, will encompass entire program.
  - D = ∑ WSSj ≡ total demand frames
    - If D > m (number of available frames) ⇒thrashing
  - Policy: If D > m, then suspend one of the processes.