# CZ2005 Operating Systems

Embedded Operating Systems (Nanyang Technological University)

# CZ2005 Operating Systems

## Introduction

### Operating Systems

An OS is a program that acts as a intermediary between a user and computer hardware.
An OS has 2 major goals:

1. User Convenience
2. Efficient hardware utilisation
    a. Hide hardware complexity
    b. Use hardware efficiently with smart resource allocation across CPU, I/O, memory

These 2 goals can be contradictory.

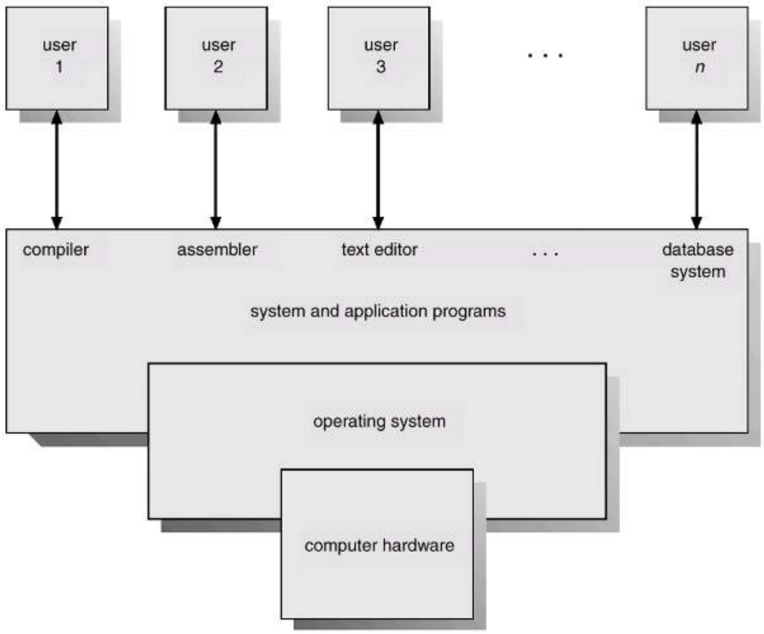
#### A computer system's components

1. Hardware
2. Operating System
3. Application programs
4. Users

#### Different Definitions of Operating Systems

1. Resource allocator
2. Control Program
    a. Controls execution of user programs and operations of I/O devices
3. Kernel
    a. The 'core' program always ready to accept new commands from users as well as hardware and application programs

## An abstract view of system components




# Types of computing Systems

1. **Batch systems**
2. **Multiprogrammed and Time-sharing system**s
    a. E.g: Desktop Systems
3. **Embedded and Cyber-physical systems**
    a. Real-time systems
    b. Handheld systems

## Simple Batch Systems

- Reduce setup time by batching similar jobs
- Automatic job sequencing, transfers control from one job to another automatically
- Simple memory layout
    - One one user job in memory at a time
- Not very efficient
    - I/O waiting results in idle CPU

## Multiprogrammed (Time-sharing) Systems

- Several jobs kept in main memory at the same time, CPU is multiplexed
- Job is swapped in and out of memory
- Highly interactive system that supports multiple online users
  - Desktops, servers

**Features needed for Multiprogramming**
- Memory Management
  - Allocation across several jobs
- CPU Scheduling
  - To choose among several jobs ready to run
- I/O Device scheduling
  - Allocation of I/O devices to jobs

**Desktop Systems**
- Personal computers with several I/O devices
  - Keyboard, mouse, display, screen, printers, etc
- Focus on **user convenience** and **responsiveness**
- May run several types of OS

## Embedded and Cyber-physical systems

- Physical systems whose operations are monitored and controlled by reliable computing and communication core
- Resource constrained: low power, small memory, low bandwidth etc
- Domain-specific OSes: Real-time, handheld, automotve, avionics, industrial controls, sensor networks, etc

## Real-time systems

- Used as a control device in dedicated application in industrial controls, automotive, avionics, medical devices, etc
- Well-defined fixed-time constraints
  - Jobs have a deadline, e.g:airbag control in cars
  - LynxOS, RTLinux, VxWorks Wind River

## Handheld Systems

- Mobile phones and tablets
- Issues
  - Limited memory
  - Slow(er) processors
  - Small display screens
  - E.g: iOS, Android, Windows Phone

## Multiprocessor Systems

- Systems with more than one CPU, or CPU with multiple cores
- Tightly coupled systems: communications usually through shared memory
- Advantages:
  - Increased system throughput
  - Economical due to sharing of memory and I/O devices compared to single CPU
  - Increased reliability due to redundancy

# Computer System Architecture

## Computer system operation

- I/O Devices and CPU and execute concurrently
- Device controller with local buffer
- Device controller moves data between buffer and memory
- Informs CPU that it has finished operation causing interrupt

## Functions of interrupts

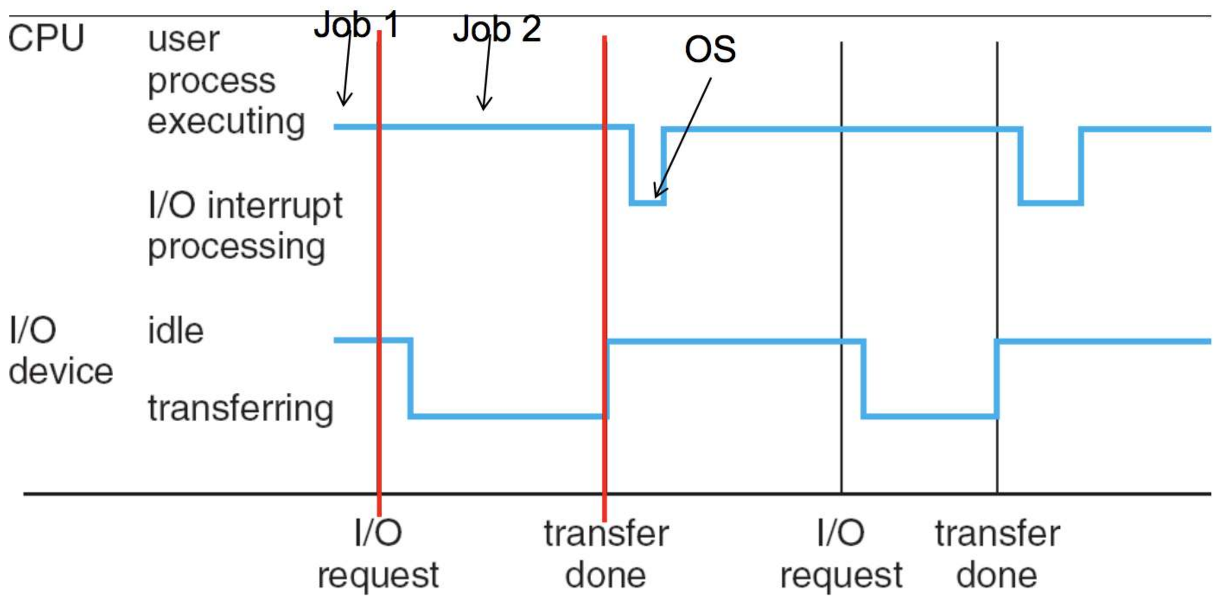
- Interrupt transfers control to **interrupt service routine** through the **interrupt vector**
- Incoming interrupts are disabled while another interrupt is being processed to prevent any loss of interrupts
- A **trap** is a software-generated interrupt caused by error/user request

**An operating system is typically interrupt driven**
An OS shouldn't poll for tasks/event completions (overhead is high)

## Interrupt Handling

- OS preserves state of CPU by storing registers and program counter (context switch)
- OS determines which type of interrupt has occurred
- Based on type of interrupt, identifies the appropriate **interrupt service routine** to execute from the **interrupt vector table**

## Interrupt-driven I/O Timeline




## Direct Memory Access

- Used for high-speed I/O devices
- OS sets up memory blocks, counters etc
- Device controller transfers data blocks from buffer to main memory without CPU intervention
- One interrupt generated per block rather than per byte

## Hardware Protection

**Dual Mode Protection**

- User mode
- Monitor mode (supervisor/system/kernel mode)
    - Executes OS processes
- Mode bit added to computer hardware to indicate current mode
- Interrupt or fault causes hardware to default to monitor bit
- Privileged instructions only allowed in monitor mode

**I/O Protection**

- User program may issue illegal I/O operation such as:
    - Read a file that doesn't exist
    - Unauthorised access to device
- ALL I/O instructions are privileged
- All I/O operations must go through OS to ensure correctness and legality
    - Trap is generated for I/O operations trying to bypass OS

**Memory Protection**

- Interrupt vector and ISR must be protected
- Registers determine range of legal address a program may access
    - Base register: First legal physical memory address
    - Limit register: Size of legal range
- Memory outside defined range protected and inaccesible
- Load instructions of base and limit registers are privileged instructions
- A Load/Store instruction to address:
    - Address checked against base register
    - Address checked against base+limit
    - Trap issued to OS if checks fail

## Operating System Services

**System Calls**

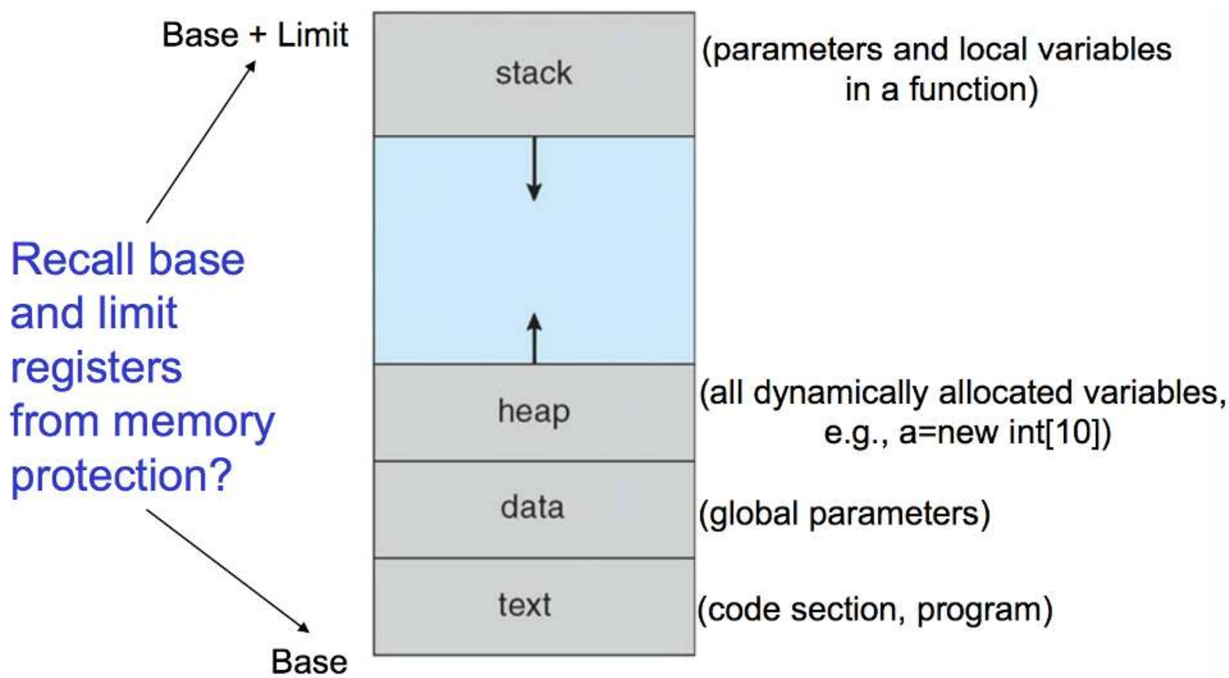
- Interface between user program and operating system (included as standard libraries)
    - Standard libraries act as API for system calls to OS
- Available as assembly-language instructions
- Can be replaced with systems programming to allow system calls made directly (C/C++)
- System call requires switch from user to kernel mode

# Processes and Threads

## Process Concept

- A process is a program in execution, progressing in sequential fashion
- OS executes variety of process
  - Batch jobs
  - Time-sharing systems (user programs & commands)
- Job = Process

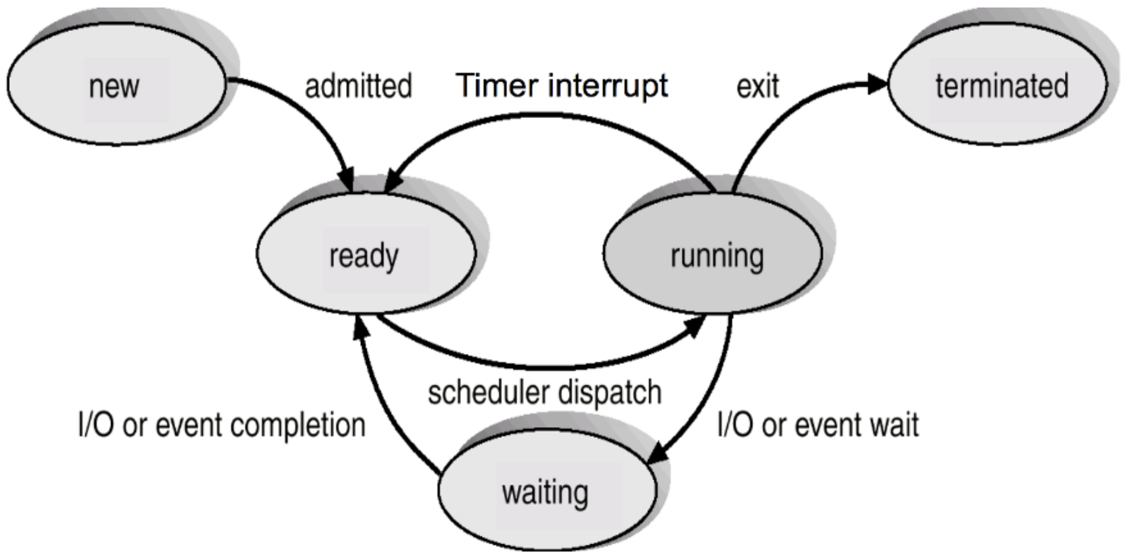
### Process in memory




- Global parameters and the program are nearer to base address
- Dynamically allocated variables are on the heap
- Parameters and local variables in a function are in the stack (starting from the limit address of the memory allocated

## Process states

1. New (Process is created)
2. Running (Instruction is executed)
3. Waiting (Process is waiting for an I/O or event)
4. Ready(Process is ready to run and waiting to be assigned to CPU)
5. Terminated (Process has completed)




Timer Interrupt is used to switch between ready processes

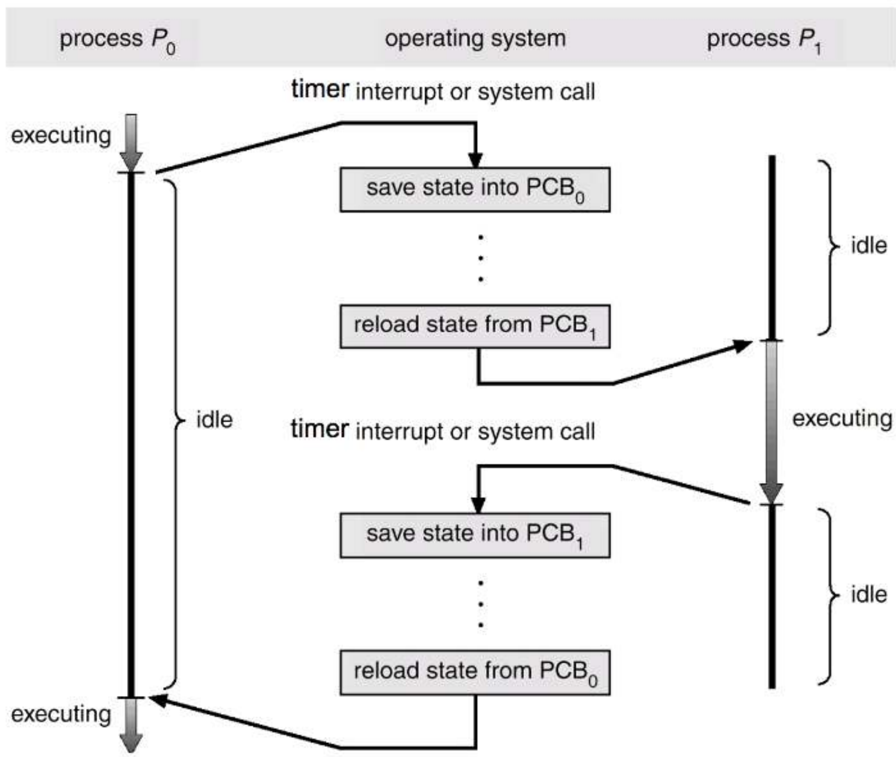
## Process Control Block (PCB)

This data structure keeps information associated with each process and stored in kernel memory space. It consists of:

1. Pointer to other PCBs (PCBs maintained in a queue/list structure
2. Process state
3. Process number (Pid)
4. Program counter (pointer to next instruction)
5. CPU registers
6. Process priority (for scheduling)
7. Memory management information (base and limit register values)
8. Information regarding files (list of open files)
9. Pointer to next PCB

## Process scheduling

### Context Switch between processes




When a CPU switches to another process, the system must save the context of the old process and load the saved context for the new process.
**Context switch time is overhead.** System does no useful work while switching.

### Process scheduling queues

Job queues: Set of all processes with the same state
Ready queue
Device queue (processes in waiting state waiting for specific I/O device)

All queues are stored in main memory (kernel memory space)
Processes migrate between queues when state changes

## Process schedulers

1. Long-term scheduler (job scheduler)
    a. Selects process from disk and loads them into main memory for execution
2. Short-term scheduler (CPU scheduler)
    a. Selects from processes that are ready to execute and allocates CPU to one of them
3. Medium-term scheduler
    a. When system load is heavy, swaps out partially executed process from memory to hard disk
    b. When system load is lighter, these processes can be swapped back into main memory

**2)** is invoked frequently (100ms) in multiprogrammed system for responsiveness and efficiency purposes
**1)** is invoked infrequently (seconds/minutes)

Degree of multiprogramming is initially controlled by long-term scheduler and *thereafter* controlled by medium-term scheduler.

## Process creation

Parent processes create children processes and so on, forming a tree of processes
This can lead to two possible execution orders:

1. Parent and child execute concurrently and independently
2. Parent waits until all children terminate (`wait()`, `join()`)
    a. E.g: Web browsers nowadays fork a new process when making new tab
    b. OS creates background processes for monitoring and maintenance

## Process Termination

1. Exit: process executes last statement and asks OS to delete it
    a. Child output return data to parent
    b. Process resources are de-allocated by OS
2. Abort: Parent terminate execution of children processes at any time
    a. Child exceeded allocated resources
    b. Task assigned to child is no longer required
    c. Parent is exiting

## Cooperating Processes

- Independent process cannot affect or be affected by execution of other process
- Cooperating processes can affect or be affected
    - Such processes have to communicate with each other to share data
    - Inter-Process Communication
        - Message Passing
        - Shared Memory
- **Message passing:**
    - No shared variables
    - 2 system calls needed:
        - send(message, recipient/mailbox)
        - receive(message, sender/mailbox)
    - Mailboxes are used for indirect message passing (AKA ports)
    - Communication requires a communication link and use system calls

**Producer-Consumer Process Paradigm**
Classical paradigm for cooperating processes
- Producer produces information for consumer that uses it
- Shared buffer used for storing information
    - Unbounded or bounded buffer size
- Producer waits while mailbox is full. Consumer waits while mailbox is empty.

# Process Scheduling

## Basic Concepts

**CPU, I/O Bursts:** Process execution alternates between CPU and I/O operations
CPU Burst: Duration of one CPU execution cycle (multiple ticks of a CPU)
I/O Burst: Duration of one I/O operation (wait time)

**CPU Scheduling objective:** Keep CPU busy as much as possible (multiprogramming)

**Focus: short-term (ready queue) scheduler on single CPU**

CPU Scheduler types:
1. Nonpreemptive: Once CPU has been allocated a process, process will keep CPU until it voluntarily releases CPU by termination or requesting I/O or event wait.
2. Preemptive: CPU can be taken away from running process at any time by scheduler
    a. Through interrupt, during process creation, or after I/O or event completion

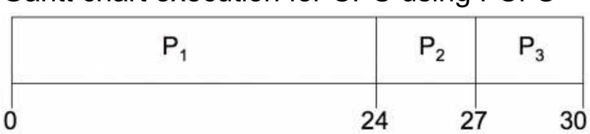
## Scheduling Objectives

1. Max CPU Utilisation (CPU busy as possible)
2. Max throughput (Number of processes that complete their execution per unit time
3. Min turnaround time (time of creation to termination)
4. Min waiting time (amount of time process waits in ready queue)
    a. Waiting time = turnaround time - CPU burst (if process have single CPU burst with no I/O)
5. Min response time (Amount of time from request submission to create process until first response is produced

## Scheduling algorithms

1. First-come, first-served (FCFS)
2. Shortest Job First
3. Priority-based Scheduling
4. Round-robin
5. Multilevel queue scheduling

Gantt chart execution for CPU using FCFS




Waiting time: 0, 24, 27
Average waiting time: 17 seconds

If arrival order is changed to P2, P3, P1, then average waiting time is reduced to 3 seconds

**FCFS Properties:**
1. Nonpreemptive
2. Suffers from convoy effect due to earlier arrived long process

**SJF Scheduling:** Prioritize processes based on CPU burst lengths
SJF/SRTF is OPTIMAL for minimising average waiting time.

When SRTF preempts a process out of the CPU, the waiting time still increases (since it is placed back into the ready queue).

**Priority-based Scheduling:**
Priority number is assigned to process with highest priority

- Smallest integer = highest priority
- Two schemes: Preemptive and nonpreemptive
- FCFS = priority based on arrival order
- SJF = priority based on CPU burst length
- Starvation issue: Lower priority processes may never execute in heavily loaded system
    - Solution: ageing. Increase priority of processes that are not able to execute

**Round-robin Scheduling**
Fixed time quantum for scheduling (q): Process is allocated CPU for q time units, preempted thereafter and inserted at the end of the ready queue. q is usually around 10-100ms

Performance:

- N processes in ready queue: waiting time no more than (n-1)q time units
- Large q = degenerates to FCFS
- Small q = Too many context switches, high overhead
- Generally higher waiting/turnaround time than SFJ
- Better response time

**Multilevel Queue scheduling:**

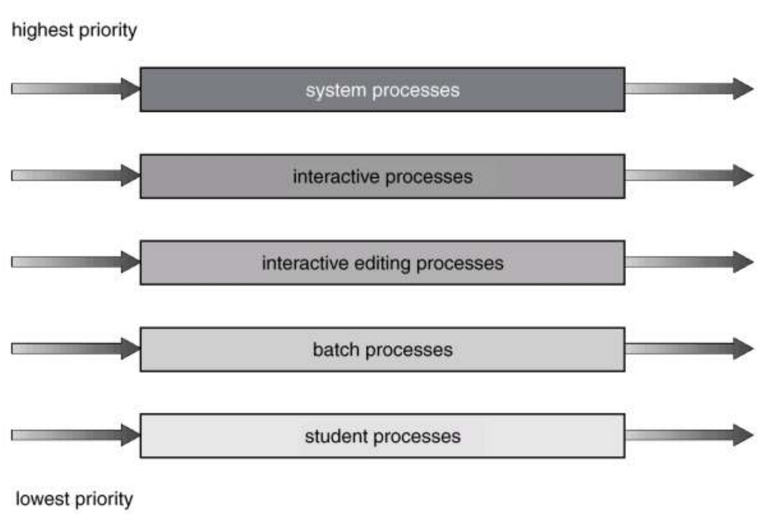
Different processes have different requirements

- Foreground processes like I/O handlers need to be interactive (RR preferred)
- Background processes need not be interactive, scheduling overhead low (FCFS)

In MLQ, Ready queue is partitioned into several queues with its own scheduling algorithm

Two schemes allowing for inter-queue scheduling:

1. Fixed priority scheduling: queues served in priority order (foreground before background)
    a. Starvation issue
2. Time-slice based scheduling: Each queue gets fixed time quantum on CPU
    a. 80ms foreground, 20ms background



# Process synchronisation

## Race Condition & Critical Section

N processes competing to use shared data that requires a proper order of execution for concurrent processes. Causal ordering of reads and writes.

Critical section problem:
Each process has a code segment called a critical section, in which shared data is accessed. At least one process modifies the shared data.

A protocol is required to ensure that when one process is executing in critical section, no other process is allowed to execute in critical section.

Assumption of critical and remainder sections:

1. Each process is guaranteed to make progress over time in these sections
2. Execution speed may be different

Desired Properties of entry and exit sections that bound the critical section:

1. **Mutual exclusion**:
   If a process is executing in its critical section, then no other process can be executing in it's own critical section at the same time.
2. **Progress**:
   If no process is executing in its critical section and there exists processes that wish to enter their critical section, then the selection of the next process to enter the critical section cannot be postponed indefinitely.
3. **Bounded waiting**:
   After a process has requested to enter critical section, and before that request is granted, other processes are allowed to enter their critical section only a bounded number of time.

## User-level software approaches w/o OS support

| Property Analysis | Algorithm 1 (turn) | Algorithm 2(flag) | Algorithm 3 (flag & turn) |
|---|---|---|---|
| Shared variables | `int turn = 0;`<br><br>turn = i means $P_i$ can enter critical section | `boolean flag[2];`<br>`flag[0]=flag[1]=false;`<br><br>Flag[i] = true means $P_i$ ready to enter critical section | `int turn = 0;`<br>`boolean flag[2];`<br>`flag[0]=flag[1]=false;`<br>Flag[i] = true means $P_i$ ready to enter critical section<br>turn = i means $P_i$ can enter critical section |
| Entry section | `while(turn != i);` | `flag[i] = true;`<br>`while (flag[k]);` | `flag [ i ] = true; turn = k; while (flag [ k ] and turn = k);` |
| Exit section | `turn = k;` | `flag[i] = false;` | `flag [ i ] = false;` |
| Mutual Exclusion | Yes | Yes | Yes |
| Progress | No<br>Turn = 0 and $P_0$ is in a long remainder section, and $P_1$ wants to enter critical section<br>Turn is not returned to $P_1$ | No<br>$P_0$ executes flag [ 0 ] = true;<br>**Context switch to $P_1$**<br>$P_1$ executes flag [ 1 ] = true;<br>Both are stuck in infinite while loop | Yes |
| Bounded Waiting | Yes | Yes | Yes |

## OS-level solutions

Software approaches are difficult to implement for more than 2 processes

OS support has 3 kinds:

1. Synchronisation hardware
2. Semaphore
3. Monitor

Modern processors provide special atomic hardware instructions (non-interruptible)

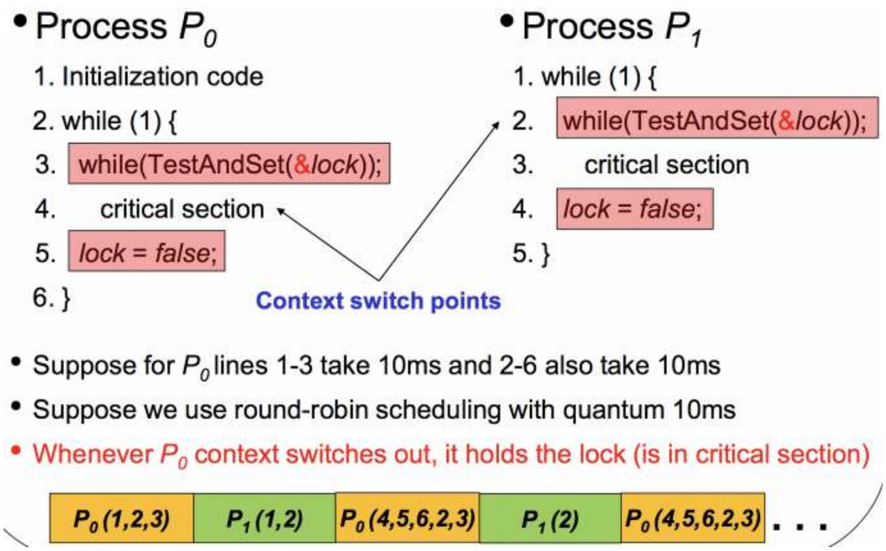
**TestAndSet:** Test and modify content of main memory word atomically

```
boolean TestAndSet(boolean *target){
      boolean rv= *target;
      *target = true;
      return rv;
}
```

**Property Analysis of TestAndSet:**

1. **Mutual Exclusion:** Yes
2. **Progress:** Yes
3. **Bounded Waiting:** No.

Proof:




## Semaphore

Shared integer variable.
Accessible by two atomic operations:

1. `Wait(S): if S>0, S--; else, yield();`
    a. `yield()`: enter waiting state or busy loop waiting
2. `Signal(S): S++;`

**Two kinds of semaphores:**

1) Counting semaphore: Integer value unrestricted
2) Binary semaphore: Value can never be greater than 1

**Classical Semaphore Implementation**

```
Wait(S): while(S<=0); S--;
Signal(S): S++;
```

Pros: No context switch overhead (busy waiting)
Cons: Inefficient if critical sections are long, busy waiting wastes CPU cycles.

Mutual Exclusion: Wait(S) must be atomic. No context switch allowed between while and S--;
If not, mutual exclusion fails.

**Current Semaphore Implementation:**

```
typedef struct {
      int value;
      struct process *L;
} semaphore;
```

L is a queue that stores processes waiting on this semaphore in a form of a PCB list

`block()`: Dequeue current process from ready queue
Enqueue process to L
Change state of process to waiting

`wakeup()`: Dequeue current process from L
Enqueue process to ready queue
Change state of process to ready

```
Wait(S): S.value--;
      If (S.value <0){
            block();
      }

Signal(S): S.value(++)
      If (S.value <=0){
            wakeup(P);
      }
```

## Best practices for semaphores

1. Find shared variables and data/shared resources
2. Identify critical section
3. Protect shared variables
    a. Use binary semaphore for mutex
    b. Apply wait and signal
4. Protect shared resource
    a. Use one counting semaphore per resource
    b. Initial value = number of resource instances
    c. Apply wait and signal

## Starvation

Priority based or LIFO policy queuing can result in starvation, as a process under these policies may never have a chance to be removed from semaphore queue.

# Classical Synchronisation Problems

**Bounded-buffer (Producer-Consumer) Problem**
Multiple consumer/producer environment

Buffer is a variable: apply binary semaphore (mutex)
Full and empty are used as resource counting semaphores
Full initially 0, empty initially n

First wait on empty/full (producer/consumer)
Then request for mutex
After critical section
Release mutex
Signal full/empty (producer/consumer)

**Dining Philosophers Problem**
Five resources, Five processes, Each needs 2

Solutions: Allow at most 4 philosophers to be hungry simultaneously
Allow a philosopher to pick up his chopsticks only if both chopsticks are available
Use an asymmetric solution: Odd philosopher picks up left chopstick first then his right chopstick
Even philosopher picks up right chopstick then left chopstick

**Readers-Writers Problem**
Writer requires exclusive access
Multiple readers can concurrently access

First reader: Block writers
Last reader: After reading, can allow writers

Shared data: readcount=0;
Semaphores: readCounter = 1; databaseAccess = 1;

Writer:
```
wait(databaseAccess);
signal(databaseAccess);
```

Reader:
```
wait(readCounter);readcount++;
If (readcount == 1) wait(databaseAccess);
signal(readCounter);

wait(readCounter);
readcount--;
if(readcount = 0) signal(databaseAccess);
signal(readCounter);
```

# Deadlocks and Starvation

## The Deadlock Problem

Deadlock: A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

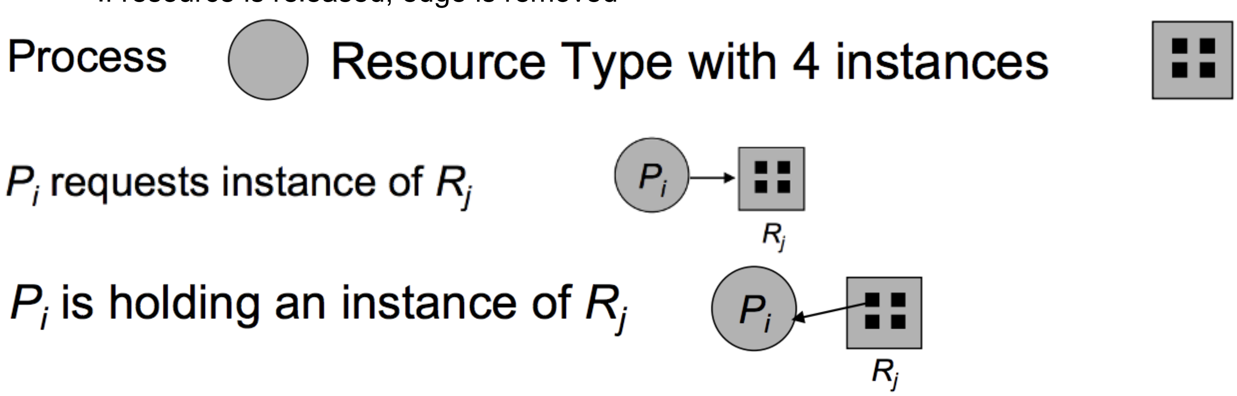
The shared resource deadlock can be resolved if one car backs up (release resource and rollback)

Several processes may need to rollback if a deadlock occurs

## Resource Allocation Graph

Graph of vertices V is partitioned into two types:
P={$P_1$, $P_2$,...$P_n$), consisting of all processes in system
R={$R_1$, $R_2$,...$R_m$), consisting of all resource types in the system

Edges E is partitioned into two types:
Request edge: Directed edge $P_i \rightarrow R_j$
    If request is granted, edge is removed
Assignment edge: Directed edge $R_j \rightarrow P_i$
    If resource is released, edge is removed




If the graph has no cycles, there is no deadlock
If the graph contains cycles, and there is only one instance per resource type, then there is deadlock. If several instances exist per resource type, there is a possibility of deadlock.

## Deadlock conditions

**Mutual exclusion:** Only one process can use a resource at a time.
**Hold and wait:** A process holding at least one resource is waiting to acquire additional resources held by other processes
**No preemption:** A resource can be released only voluntarily by the process holding it, after that process has completed its task
**Circular wait:** There exists a set of processes of processes waiting for resources such that there is a cycle in waiting for resources within the set.

## Handling deadlocks

Ensure system will never enter deadlock state
Allow system to enter deadlock and then recover
Ignore problem and pretend deadlocks never occur
Prevent at least one of the four deadlock conditions

## Deadlock Avoidance Algorithm

Algorithm checks for safe state of resource allocation.
Safe state is when there exists safe completion sequence of all processes without deadlock

1) Find a process $P_1$ where Request <=Available
2) Find a process $P_2$ where Request <=Available + $P_1$.Hold
3) Find a process $P_3$ where Request <= Available + $P_1$.Hold + $P_2$.Hold
4) ….
5) If all processes can be included, the sequence $P_1$, $P_2$, $P_3$ …. is safe.

System in unsafe state means there is a ***possibility of deadlock***. If a process releases resources before its completion then deadlock may not occur

## Banker's algorithm

Algorithm to check whether satisfying a resource request would lead to unsafe state.
Assumptions:

1. Each process must declare the maximum instances of each resource type that it needs
2. When a process gets all its resources it must return them in a finite amount of time

Data structures, where m= number of resource types, n is number of processes

1. available (vector of length m) where if available[j] == k, there are k instances of resource
2. Max (n x m matrix): if Max[i,j] == k, process $P_i$ can request at most k of $R_j$
3. Allocation (n x m matrix): similar to above but about current allocation
4. Need (n x m matrix): $P_i$ may need k more instances of $R_j$ to complete its task

## Banker's algorithm part 1: Safety algorithm

1. Work and finish are vectors of length m and n
2. Work = available, finish[i] = false for all i
3. Find i such that Finish[i] == false and Need[i,*] < work
4. If no such i exists, go to step 6
5. Work = work + allocation[i, *]; Finish[i] = true
6. If finish[i] == true for all i, system is safe. Else, system is unsafe.

## Part 2: Resource request algorithm

Request vector from $P_i$

1) If Request < = $Need_i$ go to 2). Else, raise error condition
2) If Request < = Available go to 3). Else, wait
3) Pretend to allocate requested resources to $P_i$
   a) Available = Available - request
   b) Allocation = Allocation + request
   c) Need = need - request
4) Run safety algorithm to check if safe.
5) If safe, allocate resource. Else, unsafe, $P_i$ must wait and state is restored to pre 3).

# Memory organisation

## Binding code and data to memory

To run a program, a process image must be created and loaded into memory

Instructions in memory must be fetched to the CPU. Instruction may require another access to memory.

Address binding of instructions and data to memory address can happen at 3 stages:

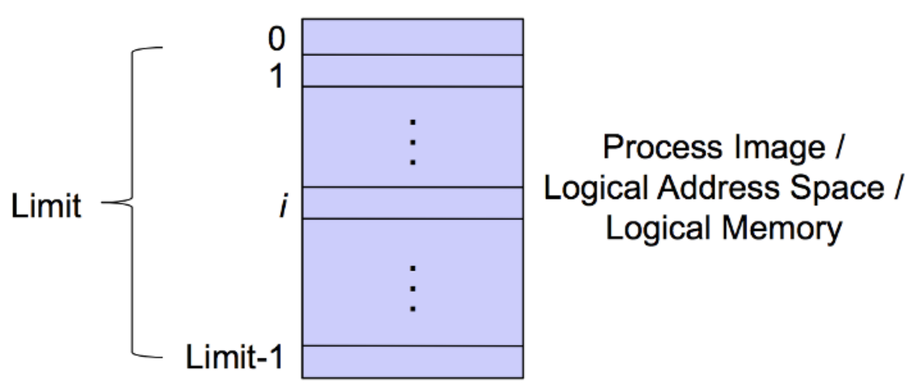
1. Compile time: If memory location is prior known, absolute code can be generated. Recompilation required if starting location changes
2. Load time: Compiler generates relocatable code. Binding performed by image loader.
3. Execution time. If process can be moved during execution from one memory segment to another, binding is delayed until runtime.

In compile- or load-time binding, absolute address format is used. If address generated by CPU < base or >= base+limit → error
Otherwise, memory address = address generated by CPU

In execution-time binding, relative address format is used. If address generated >= limit → error.
Otherwise, memory address = address generated by CPU + base

## Logical vs. Physical Address Space

- Logical address: generated by CPU; also referred to as virtual address
- Physical address: address seen by memory unit
- Address space: Address accessible by process



## Memory allocation among processes

- Contiguous allocation: Logical address space of process remains contiguous in physical memory
    - Fixed partitioning
    - Dynamic partitioning
- Non-contiguous allocation:A process logical address space is scattered over different regions in physical memory

**Fixed Partitioning**
Memory is partitioned into regions with fixed boundaries

**Contiguous Allocation**
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.

Operating system maintains information about allocated partitions and free partitions (holes)

**Dynamic Storage Allocation Problem**
How to satisfy a request of size n from list of free holes?

- First-fit: First hole big enough
- Best-fit: Smallest hole big enough. Must search entire list unless order by size. Produces smallest leftover hole.
- Worst-fit: Allocate largest hole; must also search entire list. Produces largest leftover hole.

**Fragmentation:**

- External fragmentation: Total memory space exists to satisfy a request but it is not contiguous. Happens outside partitions
    - Reduce via compaction: Shuffle memory contents to place all free memory together in one large block.
    - Possible only if relocatable address format is used in process image and binding is done during execution time.
- Internal fragmentation: Allocated memory may be slightly larger than requested memory, size difference is memory internal to partition, but not being used.

## Paging

Memory space allocated to process can be noncontiguous, process is allocated physical memory whenever latter is available.

Divide physical memory into fixed-size blocks called frames(size is power of 2, 512-8192 bytes)

Divide logical memory of same size called pages.

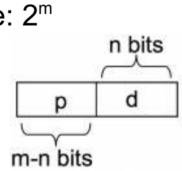
- Keep track of all free frames
- Setup page table to translate logical to physical addresses
- External fragmentation is eliminated
- Internal fragmentation is still possible, on average half of a page is unused.

## Address Translation Scheme

Logical address contains:

- Page number (p) used as index into page table entry containing frame number in physical memory
- Page offset (d) combined with frame number to define physical memory address that is sent to memory unit

Page size: $2^n$
Logical address space: $2^m$
Number of pages:$2^{m-n}$
Page table entry size:

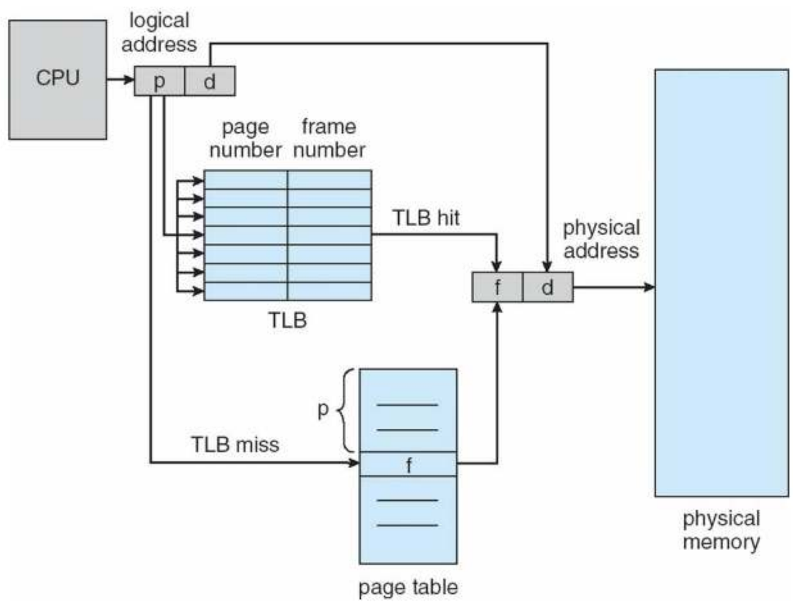



Logical address: p, d: E.g: 0111
Since we know that the logical address' last n bits are the page size (i.e page offset)
So the other bits are the index for the page table to find the frame number in physical memory.
Thus, p = 01, d = 11.
P corresponds to index 1 (page 1) on page table.
Page table indicates frame 6 in main memory.
6 = 110, including offset d, the physical address will be 11011 → 27 in main memory

## Implementation of page table

- Kept in main memory
- Has 2 registers assigned: Page-table base register and Page-table length register
- Every data/instruction access requires two memory accesses: one for page table and one for data/instruction - effective memory access time: 2t with t being memory cycle time unit
- Memory access time can be reduced by use of fast-lookup hardware cache
  - Associative registers (think register array)
  - translation look-aside buffers

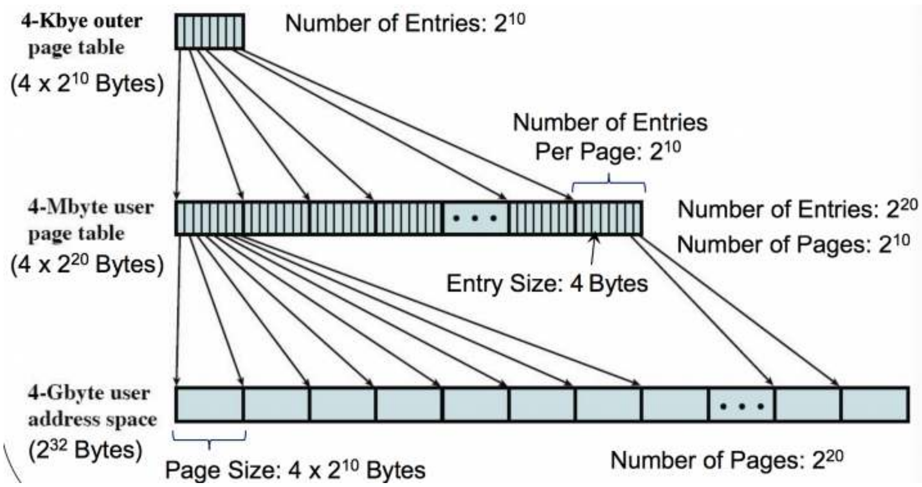
## Paging using TLBs




TLB Lookup = l time unit
Memory cycle time = t time unit
Hit ratio: percentage of times page number is found in TLB: r
Effective access time = (t + l)r + (2t + l)(1-r)
= (2-r)t + l

## Two-level Page-Table Scheme

A logical address (32-bit machine, 4K page size)
$2^{20}$ entries in a page table
If each entry is 4 bytes, each table occupies 4 megabytes of memory
A large page table is divided up to be easier to allocate in main memory, with small increase in effective access time




Logical address on 32-bit 4K page size machine has:

- Page number of 20 bits
- Page offset of 12 bits

Page table is paged: Page number divided into:

- A 10-bit $p_1$: outer page table contains $(4 \times 2^{20})/4K = 2^{10}$ entries
- A 10-bit $p_2$: each page in page table contains $(4 \times 2^{10}) / 4 = 2^{10}$ entries

## Inverted Page Table

Usually each process has its own page table. System could have many page tables consuing substantial memory space

Page table size is proportional to logical address space

**Alternative:** have single table with one entry for each physical frame, as <process-id, page-no>. This is an Inverted Page Table

Logical address: <process-id,page-no,offset>

Increases search time: table sorted by physical address but lookups occur on logical address

Inverted as the table is sorted by frame number not page number.

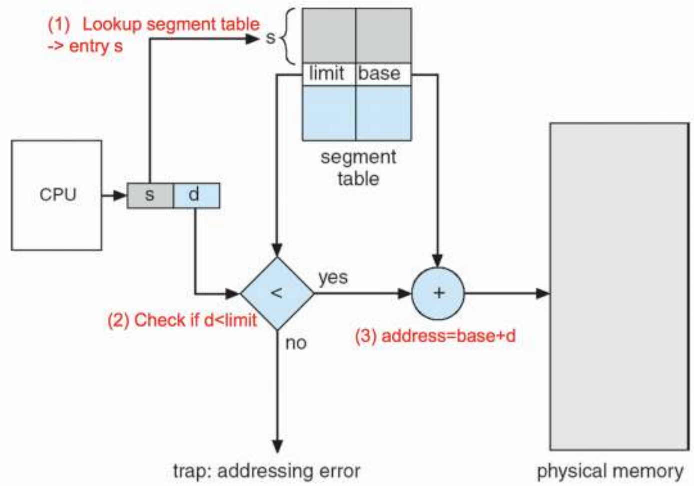
When searching, use pid+page number to find which frame it is in. Then physical address is the frame number + page offset

## Segmentation

- Memory-management scheme that breaks program up into logical segments and allocates space for these segments into memory separately
- Unlike pages, segments are variable size
- A process has collection of segments
- Like pages, segments may not be allocated contiguously

## Address Translation

- Each segment has segment no. & offset.
- Segment table. Each table entry has:
  - Base - starting physical address
  - Limit - length of segment
- Segment-table base register points to segment table's location in memory
- Segment-table length register indicates number of segments used by program




## Fragmentation in segmentation

- With variable segment length, external fragmentation is a problem. Usually use best-fit or first-fit. Dynamic storage-allocation problem.

# Virtual memory

## Swapping

- A process can be swapped out temporarily out of memory to backing store
- Backing store: Fast disk large enough to accomodate 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 amount of memory swapped
- Swapping used to
  - make space available for processes that require more memory
  - Increase degree of multiprogramming (medium-term scheduler)

Assumption was that whole program was loaded into memory at some point of time.
This restricts program size to size of main memory.
To remove this restriction, logical size requirement must be decoupled from size of physical memory space

## Demand paging

- Implementation of virtual memory is implemented using demand paging
- With each page table entry there is a valid-invalid bit
- Initially all invalid
- During address translation, if bit is invalid, there is page fault
- Page is brought into memory

1. CPU reference to page 6
2. Page table checked for page 6: invalid
   a. State of process is saved when trap to OS activated
3. Page is on backing store
4. Page is brought into memory from disk
5. Page is placed in physical memory in free frame
6. Page table is reset
   a. Valid-invalid bit set to valid
   b. Page table entry updated
7. Restart instruction 1)

## Performance of Demand Paging

p = probability of page fault

EAT = (1-p) * memory access time + p * page fault time

Memory access time << page fault time

Page fault time:
1. Servicing page fault interrupt
2. Read new page (longest time)
3. Restart process

## Page Replacement

If there's no free frame, a page not in use must be paged out
Page transfer may be necessary (one out one in) and increases page fault time

1. Find victim page using place replacement algorithm
    a. Check dirty/modified bit, if true, page out
2. Change page table entry of page table to invalid
3. Bring desired page into freed frame
4. Update the page table entry for the new page

## Page-replacement algorithm

Objective: Algorithm with lowest page-fault rate
Evaluation is through a string of memory references called a reference string
Count page faults

As number of available frames increase, number of page faults decrease

# Algorithms

## FIFO Algorithm

Replace page that has been loaded in memory for longest time
Iterate over array and wrap around

**Belady's Anomaly**

Increasing number of frames lead to more page faults

| | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F1 | 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| F2 | | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |
| F3 | | | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 4 | 4 |
| | P | P | P | P | P | P | P | | | P | P | |

| | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| F1 | 1 | 1 | 1 | 1 | 1 | 1 | 5 | 5 | 5 | 5 | 4 | 4 |
| F2 | | 2 | 2 | 2 | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 5 |
| F3 | | | 3 | 3 | 3 | 3 | 3 | 3 | 2 | 2 | 2 | 2 |
| F4 | | | | 4 | 4 | 4 | 4 | 4 | 4 | 3 | 3 | 3 |
| | P | P | P | P | | | P | P | P | P | P | P |

9 page faults vs 10 page faults

## Optimal Algorithm

Replace page that will be used for the longest period of time:
Used as theoretical benchmark.
Inclusion property: Pages loaded in n frames is always a subset of pages in n+1 frames

## Least Recently Used (LRU) Algorithm

Replace page that has not been used for the longest period of time

1. Counter implementation
    a. Each page table entry has last-used entry
    b. CPU increments logical clock for each memory reference
    c. Whenever a page is referenced, clock value to copied to last-used field
    d. During page fault, page with smallest last-used value is kicked out
2. Stack implementation
    a. Keep a stack of page numbers in doubly linked list
    b. Referenced page is moved to the top
    c. Bottom of stack is page to page out
    d. List update is expensive, but no search needed for replacement

## LRU Approximation algorithms

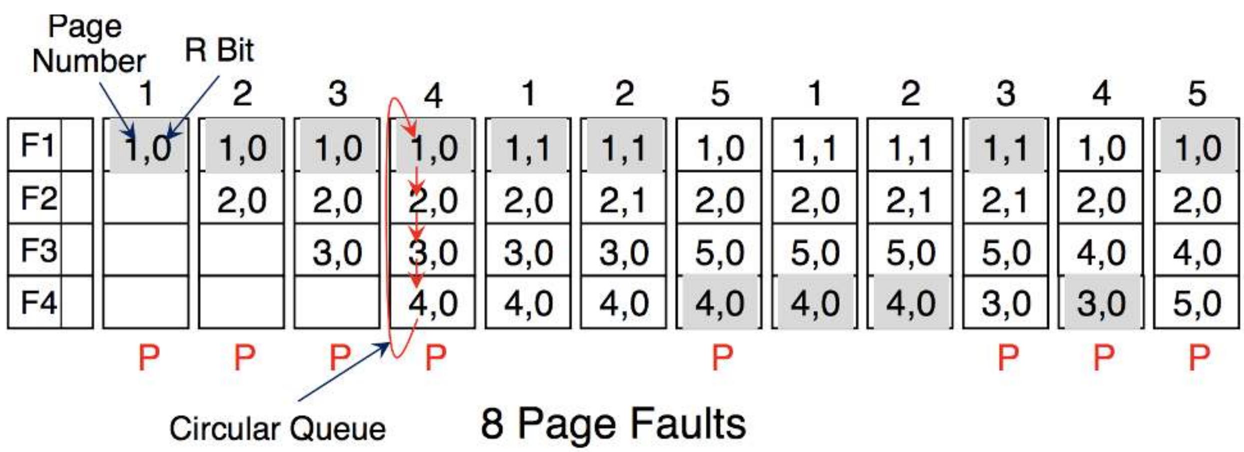
Hardware support necessary for exact algorithms are expensive and generally unavailable
Hardware settable reference bit is used

1. Each page has a bit R initially 0
2. On page reference, R is set to 1
3. Replace page with R = 0 if exists.

## Second Chance Algorithm

Clock Algorithm:

1. Clock hand points to oldest page
2. During page fault, page being pointed by hand is inspected. If R bit is set, it is cleared and the hand is advanced to the next page
3. Otherwise page is evicted and new page is inserted into its place, hand is advanced one position
4. Worst case: when all bits are set, hand cycles through whole queue giving each page a second chance. Degenerates to FIFO replacement




## Allocation of Frames

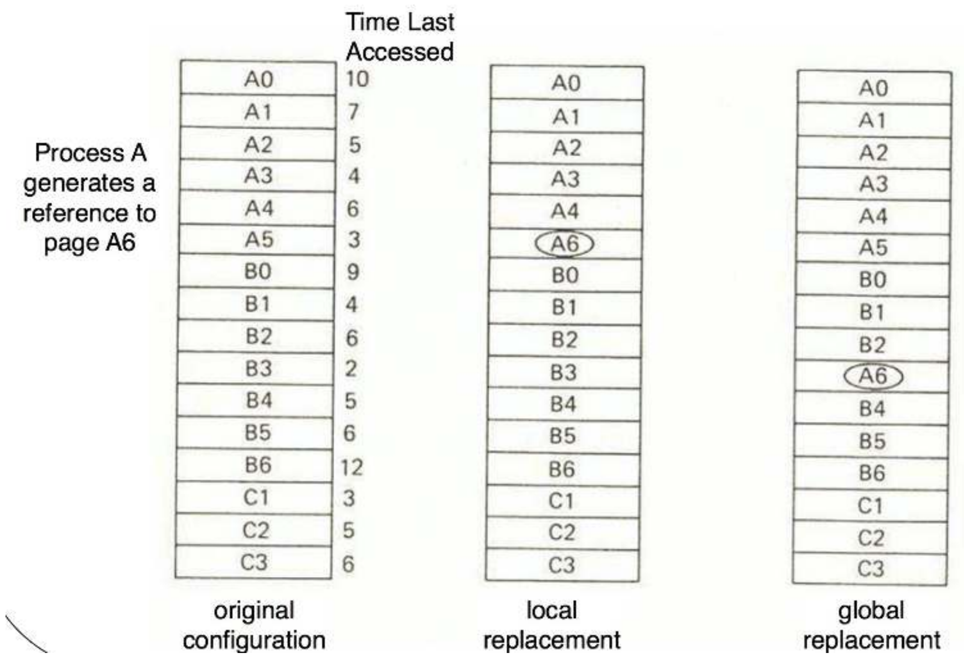
- Smaller memory allocated, more processes residing in memory
- Small number of pages loaded increases page faults
- Past a certain size, further allocation of pages will not affect page fault rate

1. Fixed allocation
    a. Fixed number of frames (equally or proportionally allocated)
    b. During page fault, one of the pages of that process must be replaced
2. Variable allocation
    a. Allows page frame allocated to a process be varied over process lifetime

## Global vs Local Allocation

Global Replacement: Process selects replacement frame from set of all frames; one process can take a frame from another process. Performance dependant on external circumstances

Local Replacement: Each process selects only from its own set of allocated frames. May hinder other processes by not making available its less used pages/frames




| | Local Replacement | Global Replacement |
|---|---|---|
| Fixed Allocation | • Number of frames allocated is fixed<br>• Page to be replaced is chosen from frames allocated to process | Not possible |
| Variable Allocation | • Number of frames allocated variable across process lifetime<br>• Page to be replaced is chosen from frames allocated to process | Page to be replaced chosen from all available frames in main memory |

## Thrashing

If a process does not have enough pages, page-fault rate is high. Leads to:

1. Low CPU utilisation
2. OS thinks degree of multiprogramming needs to increase
3. Another process is added
4. CPU utilisation drops further

## Working-set Model

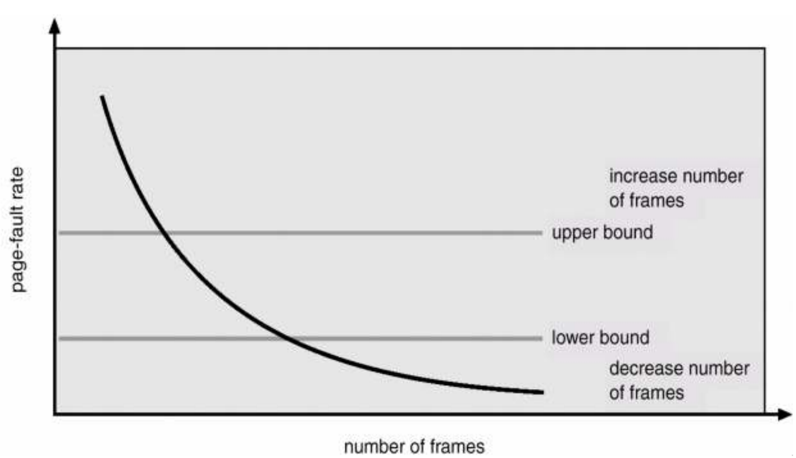
Locality of reference:

- Temporal Locality: Locations referred to just before are likely to be referred to again. \
- Spatial Locality: Code & data are usually clustered physically

Working-set window: fixed number of page references
Working set is considered as the set of unique frame references in the last n references

$WSS_i$ (working set size of Process $P_i$): total number of pages referenced in most recent working set window
$D = \sum WSS_i$: total demand for frames
M: total number of frames
If D>m: Thrashing
Policy: if D>m, then suspend one of the processes

## Page-Fault Frequency Scheme




Establish “acceptable” page-fault rate

- If actual rate is too low, remove a frame from that process
- If actual rate is too high, give process a frame
- May need to suspend process if fault rate increases and no free frames available

# File Systems

## File System Structure

Consists of many levels:

- Logical File System
- File-Organisation Module
- Basic File System
- I/O Control

### File attributes

1. Name
2. Type
3. Location
4. Size
5. Time, Date, User ID
    a. For protection, security, usage monitoring

### File Structure

Unstructured:
A file is a stream of bytes. Each byte is individually addressable from beginning of the file

Used by UNIX and MSDOS

## File access methods

### Sequential Access

Information in the file is processed in order, byte after byte: Operations include:

- Read byte from next position
- Write byte to next position
- Reset to beginning

### Direct Access

- Bytes of file can be read in any order by referencing byte number
- Based on disk model of file
- Operations: read byte at n, write byte at n, position to n

## File Operations

| *Commands* | *Explanation* |
|---|---|
| Create | allocate disk space; create directory entry with file attributes. |
| Delete | delete the corresponding directory entry; deallocate disk space. |
| Open | search the directory structure for file entry; move the content to memory (put in the *open file table*).<br>Information Associated with an open file:<br>- *Current Position Pointer*<br>- *File Open Count*<br>- *Disk Location* |
| Close | move the content of directory entry in memory to directory structure on disk. |
| Write | search *open file table* to find the location of file; write data to the position pointed by *Current Position Pointer.* |
| Read | search *open file table* to find the location of file; read data from the position pointed by *Current Position Pointer.* |

## Directory Structure

Directory contains one entry per file, residing on disk
Structures:

1. Each entry contains file name and other attributes
2. Each entry contains file name and pointer to another data structure where attributes are found

**Implementation**

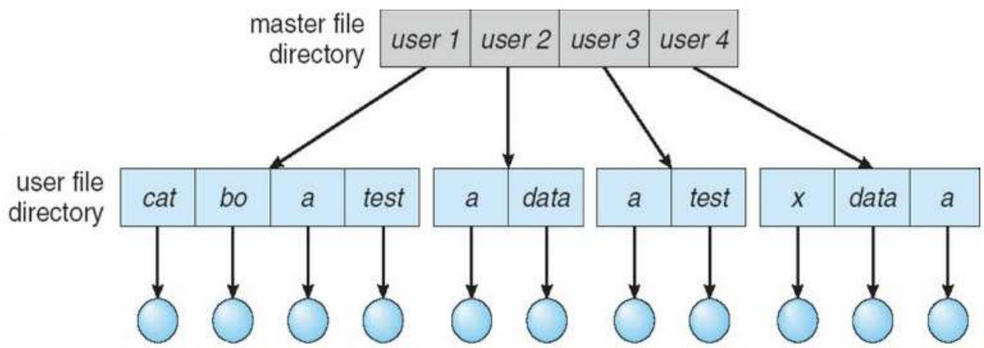
1. Linear list
    a. Simple implementation
    b. Time-consuming to execute, requires linear search
2. Hash table
    a. Decreases directory search time
    b. Collisions: two file names hash to same location, use linked list of entries instead

## Directory organisation

Logical organisation is needed to have:
1. Efficiency of file locating
2. Naming
   a. Different users, same name for different files
   b. Same file, different names
3. Grouping
   a. Logical grouping of files by their properties

## Two Level Directory

One directory for each user:




Efficiency: Pass
Naming: Pass
Grouping: Fail

## Tree-Structured Directories

- Path name:
  - Absolute: Begins at root and follows path down to specific file
  - Relative: Defines path from current directory
- Characteristics
  - Efficiency: Pass
  - Naming: Pass
  - Grouping: Pass

## Acyclic-graph Directory

Graph with no cycles is natural generalisation of the tree scheme
Naming: File can have two different names (aliasing)
Support for File Sharing: Necessary for collaboration

## File Sharing

1. Symbolic LInk
    a. Directory entry is link, containing absolute/relative path name
    b. Resolve link by using path name to locate real file
    c. Slower access than hard linking
2. Hard link
    a. Duplicate all information about file (File Control Block) in multiple directories

**Issues with File Sharing:**

- In traversing the file system, shared files visited more than once
    - Need to ignore the link
- Deleting a shared share leaves dangling pointers to non-existent file
  Solutions:
    - Search for dangling links and remove them
    - Leave dangling links and delete them only when they are used again
    - Preserve the file until all references are deleted

## Directory operations

| *Commands* | *Explanation* |
|---|---|
| Create | create a directory.<br>In UNIX, two entries "." and ".." are automatically added when a directory is created. "." refers to the *current directory*; and ".." refers to its *parent*. |
| Delete | delete a directory.<br>Only empty directory can be deleted (directory containing only "." and ".." is considered empty). |
| List | list all files (directories) and their contents of the directory entry in a directory |
| Search | search directory structure to find the entry for a particular file. |
| Traverse | access every directory and every file within a directory structure. |

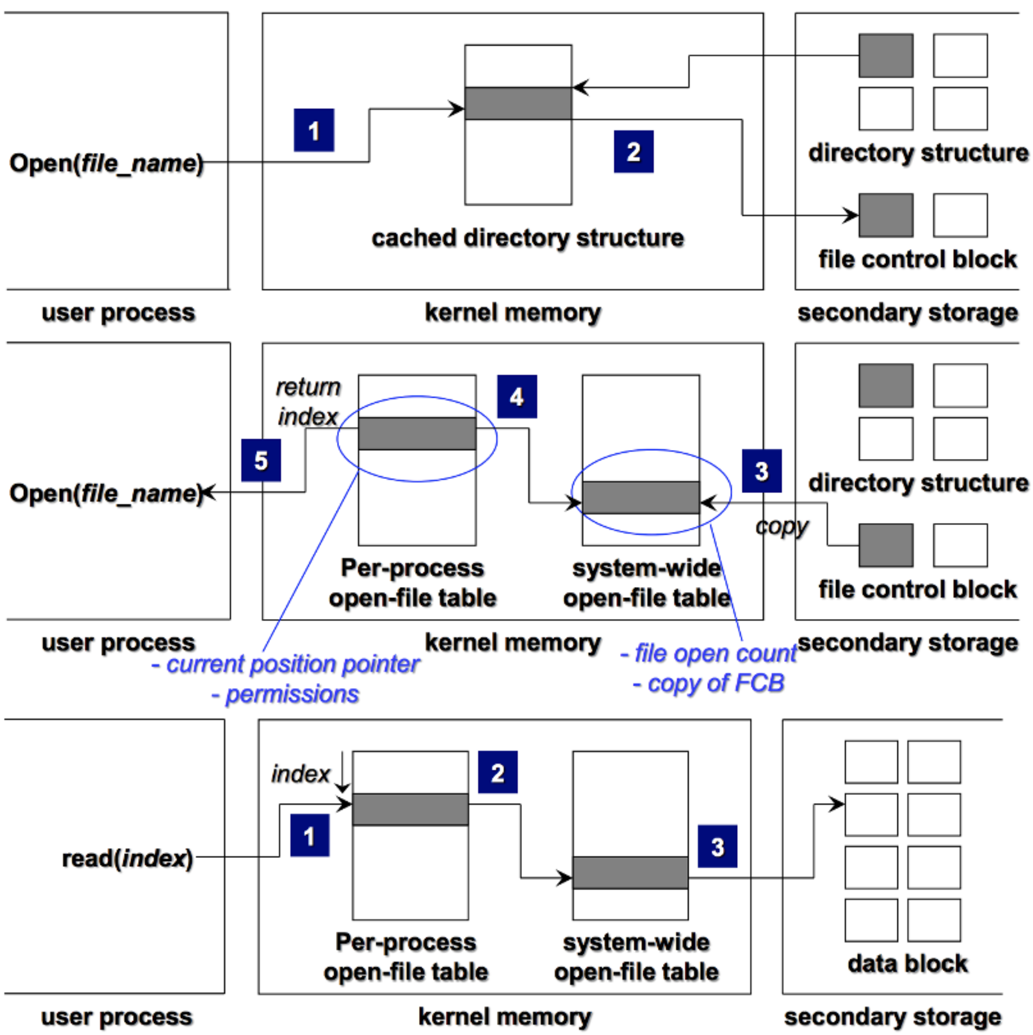
## File protection in UNIX

Model of Access: Read, Write, Execute
Three classes of users: Owner, Group and Public access

Permissions on a directory:

- To access a directory, the execute permission is needed. Without it, one cannot execute any command on that directory or have access to file contained in the file hierarchy rooted at that directory
- No read permission: Cannot list directory
- No write permission: Can't create or delete files in directory

## In-memory File System Data Structure

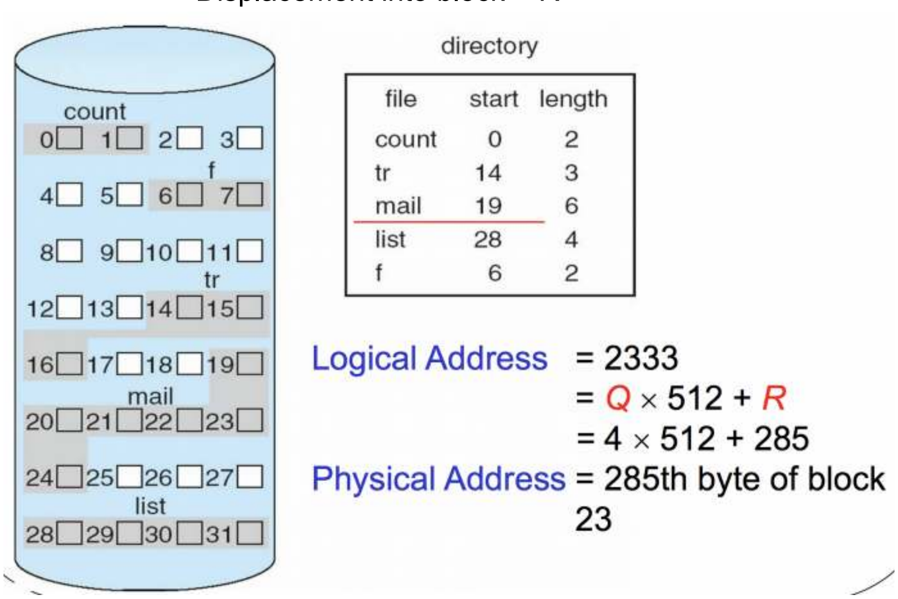


## Allocation Methods

1. Contiguous allocation
2. Linked allocation
3. Indexed Allocation

**Contiguous allocation**
Each file occupies set of contiguous blocks on disk

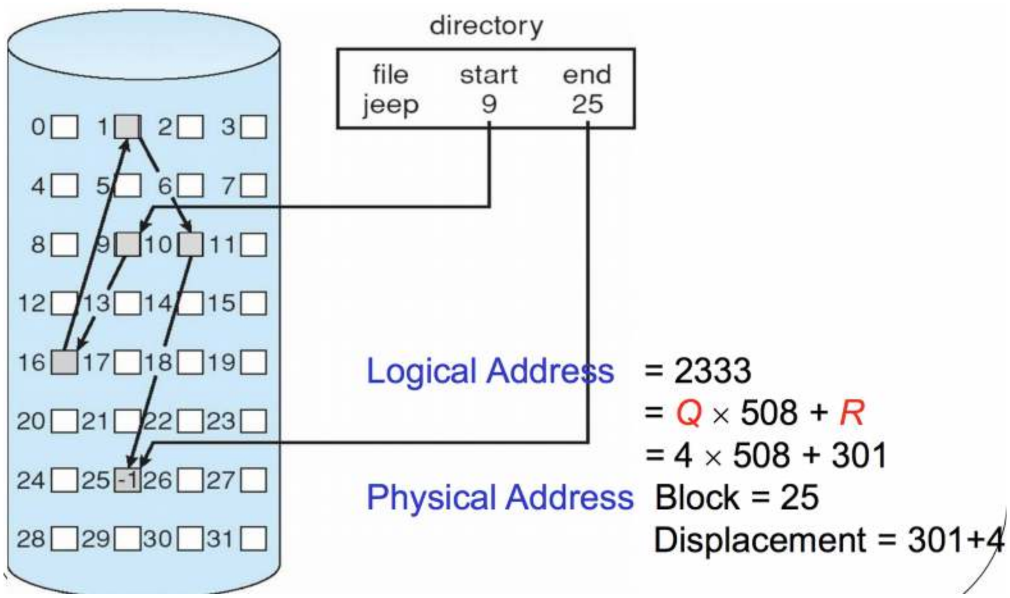
- Simple: starting location (block #) and length (number of blocks) is required
    - Supports random access
- Problems: Waste of space (fragmentation etc)
    - Find space constricted by size of hole, may need to move to bigger hole
    - If needed file space is overestimated: Internal fragmentation
- Logical to physical address mapping:
    - Block size 512 bytes
    - Logical address Q x 512 + R
    - Block to be accessed = Q + starting address
    - Displacement into block = R



**Linked Allocation**

Each file is a linked list of disk blocks, blocks are scattered anywhere on disk

- Simple: need only starting address
  - No waste of space
  - No constraints on file size: blocks allocated as needed
- Problem
  - Random Access not supported
- Logical to Physical Address Mapping:
  - 512 bytes, first 4 bytes reserved for pointer to next block in the list
  - Logical address: Q x 508 + R
  - Block to be accessed is (Q+1)th block in linked chain of blocks representing file
  - Displacement into block = R+4
- Allocate as needed, link together




**Linked Allocation: File Allocation Table**

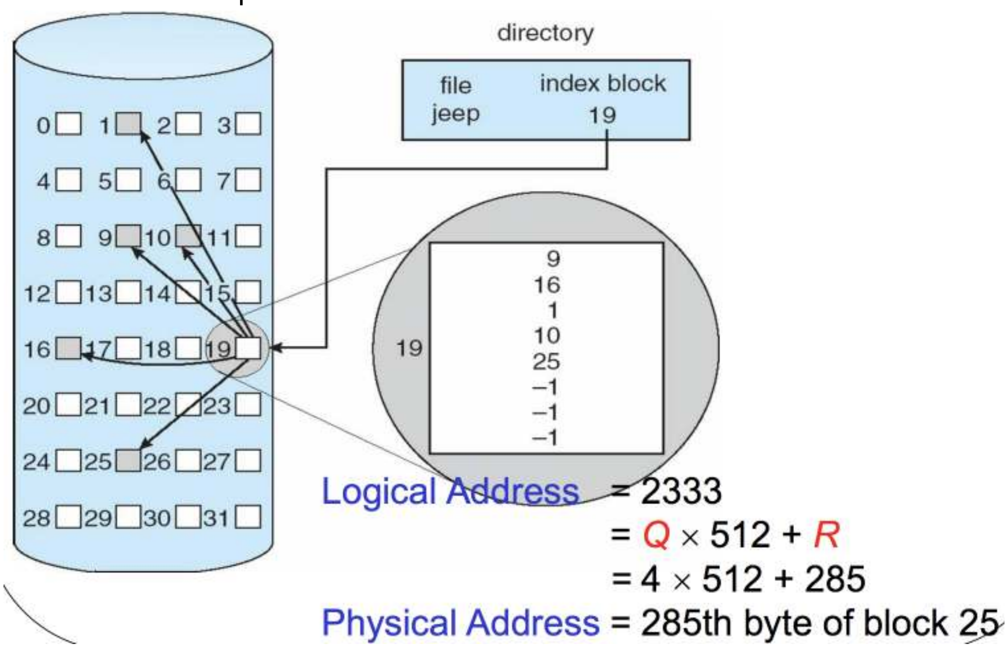
Disk space allocation used by MS-DOS and OS/2

- File Allocation Table has one entry for each disk block, indexed by block number
- The FAT entry contains either a special end-of-file value, or block number of next block in file, or a "free"
- Directory entry contains block number of first block
- Random access can be optimised

**Indexed Allocation**

Each file has index block containing pointer to allocated blocks. Directory entry contains block number of index lock

- Supports random access
- Dynamic storage allocation without external fragmentation
- Problem:
  - Overhead of keeping index blocks and address mapping
- Logical to physical address mapping:
  - Maximum file size is 128K bytes and block size is 512 bytes
  - 2 blocks needed for index table (4 bytes per pointer)
  - Logical Address: Q x 512 + R
  - Displacement into index table: Q
  - Displacement into block: R




**Index Allocation: UNIX inode**

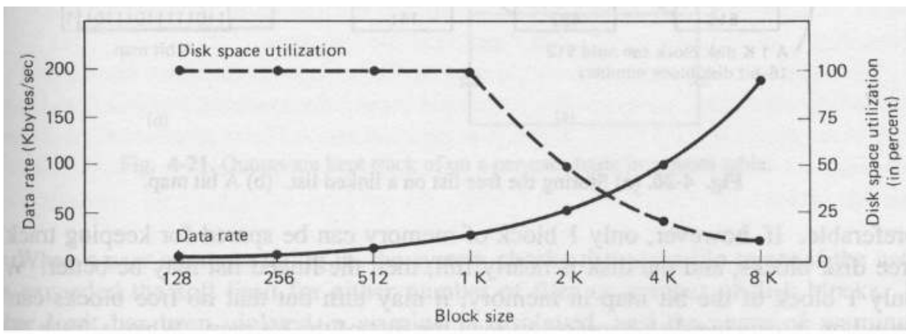
- For each file/directory, there is an inode (index block)
- Inode contains:
  - File attributes
  - 12 pointers to direct data blocks
  - 3 pointers to indirect index blocks
    - Single indirect
    - Double indirect
    - Triple indirect

With 4-byte file pointer and 4K bytes block.
Max file size:
Direct access 12 x 4K = 48K
Single indirect access: 4K * 4K/4 = $2^{22}$
Double indirect access 4K * 4K/4 * 4K/4 = $2^{32}$
Total file size: > 4GB

## Disk-Space Management

Block size affects both data rate and disk space utilisation

- Big block size: File fits into few blocks: Fast to find and transfer, wastes space if file does not occupy entire last block
- Small block size: File consist of many blocks: Slow data rate
- Trade-off between time and space utilisation




**Keeping track of free blocks**
Bitmap or Bit Vector
Linked List

## Bitmap

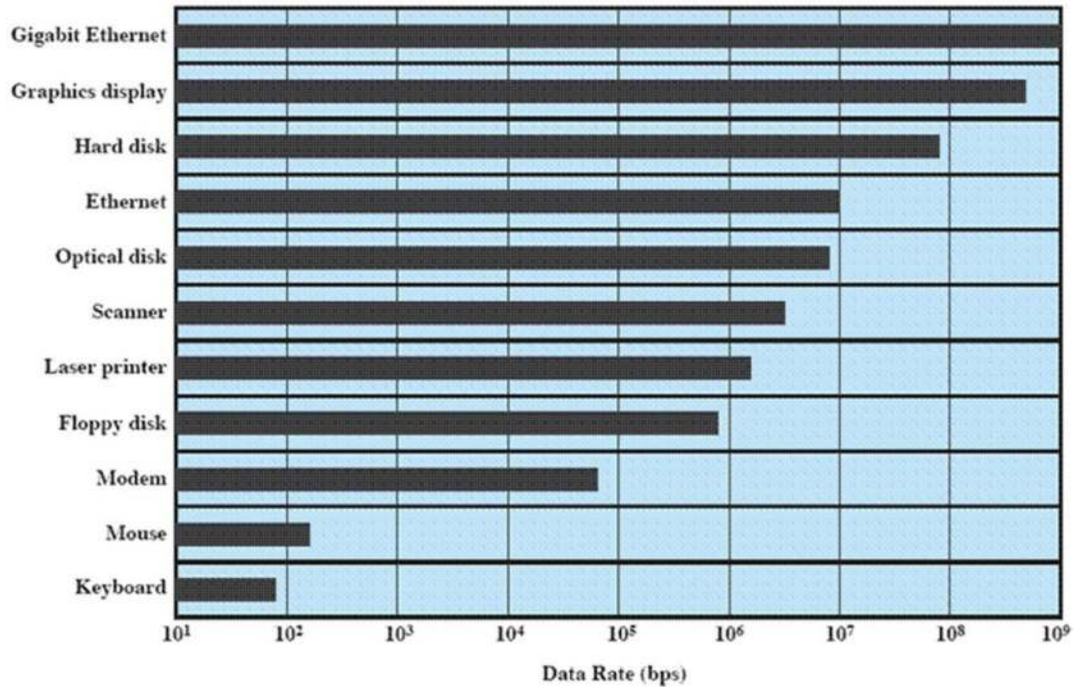
Block size: $2^9$
Disk Size: $2^{34}$ (16 GB)
Bit map size = $(2^{34}/2^9)/8 = 2^{22}$ bytes

- Bit map usually kept in fixed place on disk, brought into memory for efficiency, write back to disk occasionally for consistency and security
- Easy to locate free blocks but inefficient unless entire map is kept in memory

## Linked list

Link all free disk blocks together, keeping pointer to first free block in special location on disk and caching it in memory
No extra space required but not efficient

# I/O System and Disk




## I/O System Design Objectives

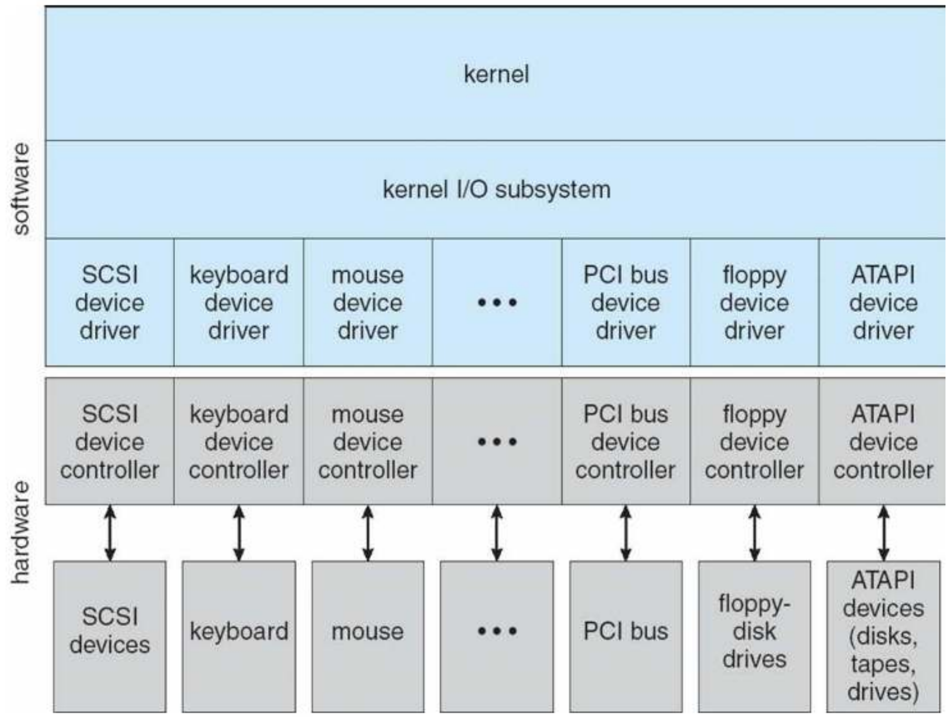
Efficiency

- I/O operations often form bottleneck
- I/O operations are extremely slow compared with main memory and processor
- Area with most attention is disk I/O

Generality

- Desirable to handle all devices in uniform manner
- Applies to the way processes view I/O devices and OS manages I/O devices and operations

## Kernel I/O Structure

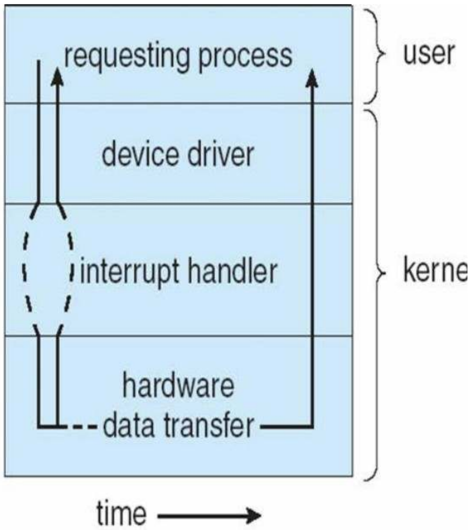


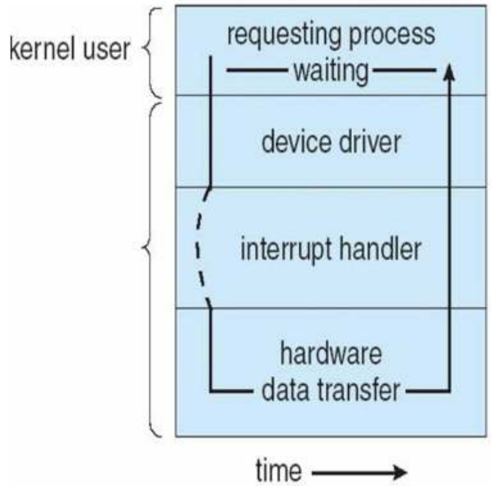

## Kernel I/O Subsystem

- I/O Scheduling
  - Schedules I/O request through rearranging the order of serivce
  - Can improve system efficiency
- Buffering
  - Store data in memory while transferring devices
  - Cope with device speed mismatch
  - Cope with device transfer size mismatch like in networking
- Caching
  - Fast memory holding copy of data
  - Improve I/O efficiency for files written and reread rapidly
- Spooling
  - Hold output for device
  - Used in device serving only one request at a time like a printer

## Performance Consideration

1. Synchronous/ Blocking I/O →
2. Asynchronous/ Non-blocking




I/O ↓ Asynchronous I/O and double-buffer to improve performance: CPU request and granting does not need to wait, i.e, asynchronous. CPU issues request and continues running other processes.

Data copying between I/O controllers and physical memory also affect system performance

## Disk Scheduling

OS is responsible for using hardware efficiently: Disk drives must have fast access time and high disk bandwidth

Access time of a disk:

1. Seek time: Disk to move heads to track containing desired sector (5-25 ms)
2. Rotational latency: Disk rotate desired sector to disk head (8ms)

Aim: Minimise seek time (seek distance)
Disk bandwidth is total number of bytes transferred, divided by total time between 1st request and completion of last transfer

## Disk Scheduling Algorithms

**FCFS**
FCFS may have wild swings back and forth

**Shortest-Seek-Time-First (SSTF)**
Selects request with minimum seek time from current head position

SSTF is a form of SJF: Starvation under heavy load as distant requests may never be serviced

**Scan/Elevator algorithm**
Disk arm starts at one end of disk, moves towards other end, servicing requests until it gets to other end where head movement reversed and servicing continues

Unlikely Starvation but requests may be delayed

**Circular-SCAN (C-SCAN)**
Head moves from low end to high end of disk, servicing requests along
When reaching the high end, head immediately back to beginning of disk without servicing

**C-LOOK**
Moves from low end to high end
Reaches last request, head returns immediately to lowest cylinder # request

Disk servicing influenced by file-allocation method:
Program reading contiguously allocated file generate requests close together
Linked or indexed file generates requests widely apart resulting in greater head movement

## Selecting a disk-scheduling algorithm

The algorithm should be written as a separate module, allowing it to be replaced with different algorithm if necesary.
SSTF or C-LOOK is a reasonable choice.

## Disk Management

Low-level formatting (physical formatting) Dividing a disk into sectors that the disk controller can read and write
To use a disk to hold files, the OS needs to record its own data structure on the disk

1. Partition the disk into one or more groups of cylinders
2. Logical formatting or "making a file system"
3. Bootstrap program initializes system
    a. Tiny bootstrap stored in ROM, loads full bootstrap program (in the disk boot

blocks) into memory and starts execution
b. Finds OS kernel on disk and loads into memory
c. Jumps to initial address to start OS

## Disk Reliability

**Disk striping** uses group of disks as one storage unit
Each block is broken into sub-blocks, each sub-block stored on each disk
Block transfer is faster due to sub-blocks transferred in parallel

**Disk mirroring** keeps duplicate of each disk
Logical disk consisting of 2 physical disks
If one fails, the data can be read from the other

RAID (Redundant-Array-of-Independent-Disks)

RAID 0: Non-redundant striping
RAID 1: Mirror duplicates
Mirror of Strips: RAID 0 + 1
Strips of Mirrors: RAID 1 + 0