Chapter 1 + 2

Introduction

Main Topics:

The study of OS includes mainly three basic topics which are related to the three basic hardware components of the machine (CPU , MEMORY , I-O DEVICES) :

(1) **Processor** Management (CPU management).

(2) **Memory** Management.

(3) **File System** Management.

What is an operating system?

Operating system - a program (a set of programs) that acts as an **intermediary** (**interface**) between a user (running computer program) and the computer hardware.

Operating system goals:

- Overall goal :**Execute user programs** and make solving user problems easier.

- Primary goal :Make the computer system ***convenient*** to use.

- Secondary goal :Use the computer hardware in an ***efficient*** manner.

Computer System Components

1. **Hardware** - provides basic computing resources (CPU, memory, I/O devices).
   Physical devices : Wires, chips, power supplies, … etc
   Microprogram – a primitive software layer which acts as interface between
   The bare hardware and the machine lang. Program.
   An **interpreter** that fetches and executes the machine (assembly) lang.
   Instruction in very little steps.
   Machine language : The set of instructions the Microprogram interprets.

2. **Operating system** - controls and coordinates the use of the hardware among the various application programs for the various users.

3. Applications programs - define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business pro-grams).

4. Users programs.

Operating System Definitions

- Can't give precise definition, like t he government, not useful by itself.

- Instead of defining the OS , what it is. We will state what it can do.

- It depends how we **view** the OS :

* **Control program** : controls the execution of user programs and operation of I/O devices.
(Overall objective: Executes user programs)

* **Extended machine** : It hides all the complexity of system programming.
(Primary goal :Conveniency)

* **Resource (manager)** allocator - manages and allocates resources.
(Secondary goal : Efficiency)

* **Kernel** - the one program running at all times (Anything else is just an application programs).

History of Operating System:

(#) Early Systems - bare machine (early 1950s)

Structure:
- Large machines run from console
- Single user system
- Programmer/User as operator
- Paper tape or punched cards

Early Software:
- Machine language
- Assemblers
- Loaders
- Linkers
- Compilers

Inefficient use of expensive resources:
- LowCPU utilization

- significant amount of setup time

(#) A simple Batch systems

* Automatic job sequencing – automatically transfers control from one job to another. First operating system, which is called the **Resident Monitor**

Problem:

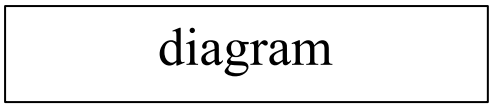
Poor Performance - since I/O and CPU **could not overlap**, and card reader very slow. (Slow I/O devices relative to CPU speed ,
fast card reader 1200 cards/minute ,CPU processes 300 cards/second)

Solution:

* ***Off-line operation*** : speed up computation by loading jobs into memory from tapes and card reading and line printing done off-line.

* ***Bufferring*** : Using buffers by which the I-O of one job is overlapped with its execution.

* ***Spooling*** : Overlap the I-O of one job with the execution of another job



* **While executing one job, the operating system**:

- reads the next job from the card reader into a storage area on the disk (job queue).

- outputs the printout of previous job from disk to the line printer.

* ***Job pool*** - data structure that allows the operating system to select which job to run next, in order to increase CPU utilization.

Multiprogrammed Batch Systems (Multiprogramming)

Several jobs are kept in main memory at the same time, and the CPU is fluctuates among them.
i.e. , the CPU switches to another job when that job **needs to wait, typically when it needs I-O**. In addition to: (**Finishes execution**, **System Interrupt**)

OS Features Needed for Multiprogramming

(-) I/O routine supplied by the system.
(-) Memory management - the system must allocate the memory to several jobs.
(-) CPU scheduling - the system must choose among several jobs ready to run.
(-) Allocation of devices.

Time-Sharing Systems

(*) Several jobs are kept in main memory at the same time, and the CPU is fluctuates among them.

(*) Every job is assigned a **slice** of time (**quantum Q**)

(*) The CPU switches to another job when that job the **Q** for that is finished.
In addition to: (**Needs I-O** ,**Finishes execution**, **System Interrupt**)

(*) A job is swapped in and out of memory to the disk.

(#) Parallel Systems

Multiprocessor systems with more than one CPU in close communication.

(A) ***Tightly coupled*** system - processors share memory and a clock; communication usually takes place through the shared memory.

Advantages of parallel systems:
- Increased *throughput*
- Economical
- Increased reliability

(*) ***Symmetric multiprocessing***
- Each processor runs an identical copy of the operating system.
- Many processes can run at once without performance deterioration.

(*) ***Asymmetric multiprocessing***
1. Each processor is assigned a specific task; master processor schedules and allocates work to slave processors. (**Master/Slave** Relationship)
2. More common in extremely large systems.

(B) Distributed Systems

Distribute the computation among several physical processors.

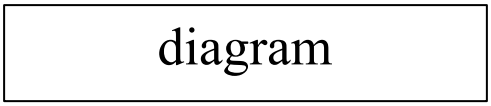
(*) ***Loosely coupled*** system - each processor has its own local memory; processors communicate with one another through various communication lines, such as high-speed buses or telephone lines.

Advantages of distributed systems:
- Resource sharing
- Computation speed up - load sharing
- Communication

(#) Real-Time Systems

(*) Special purpose Os Often used as a control device in a dedicated application such as controlling scientific experiments, medical imaging systems, industrial control systems, and some display systems.

(*) Well-defined fixed-time constraints.



Computer-System Operation

(*) I/O devices and the CPU can execute concurrently.

(*) Each device controller is in charge of a particular device type.

(*) Each device controller has a local buffer.

(*) CPU moves data from/to main memory to/from the local buffers.

(*) I/O is from the device to local buffer of controller.

(*) Device controller informs CPU that it has finished its operation by causing an interrupt.

Bootstrap program

(*) The initial program that runs when the power is on.

(*) It initializes all aspects of the computer, CPU registers, Device Controllers, Memory contents) .

(*) Loads the Kernel of the OS into memory.

(*) The OS executes the first process **init** and waits for an event to occur (interrupt).

Interrupts

A signal sent to the CPU by a **Hardware** or a **Software** (System call).

Events that may trigger interrupts:
- Completion of an I-O.
- Division by zero.
- Invalid memory access.
- Request for OS service.

($$$) Each interrupt has a special service routine for handling the interrupt.

(*) Interrupt transfers control to the interrupt service routine, generally, through the **interrupt vector**, which contains the addresses of all the service routines.

(*) Interrupt architecture must **save** the address of the interrupted instruction.

(*) Incoming interrupts are **disabled** while another interrupt is being processed to prevent a lost interrupt.

(*) A **trap** (or an exception) is a software generated interrupt caused either by :
2. An error : Division by zero , invalid memory access.
3. A user program request for a service by the OS.

(*) An operating system is **interrupt driven** ,idle if there is no activity or interrupt. .

Interrupt Handling

(1) The operating system preserves the state of the CPU by storing registers and the program counter.

(2) Determines which type of interrupt has occurred:
- ***polling*** (querying of all I-O devices which requested service.
- By the vectored interrupt system.

(3) A correct action should be taken for each type of interrupt by executing the appropriate segment of code for that interrupt.

I-O Interrupts Structure

To start an I-O operation:

- CPU loads the appropriate register within the device controller with instruction.
- Device controllers examines the contents of the register to determine the action.
- Once the I-O is complete the device controller informs the CPU that I-O is through an interrupt.

There are two types of I-O.

(1) ***SYNCHRONOUS*** : After I/O starts, control returns to user program only upon I/O completion which is accomplished by :
- **wait** instruction idles the CPU until the next interrupt.
- **loop** (LOOP : jmp LOOP)

Advantage : at most one I/O request is outstanding at a time; no simultaneous I/O processing.

(2) **A*SYNCHRONOUS :*** After I/O starts, control returns to user program without waiting for I/O completion.

4. **System call**: request to the operating system to allow user to wait for I/O completion.

Direct Memory Access (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 the one interrupt per byte.

Storage Structure

Primary Storage :

Main memory

5. The Only large storage media that the CPU can access directly.
6. Array of words, each is addressable.

7. Activity is a sequence of ***load*** and ***store*** instructions.

Load instruction : moves a word(s) from memory to an internal register
Generally known as Instruction Register (IR).
store instruction : moves a word(s) from the register into a memory location.

Instruction Cycle:

8. Fetch instruction from memory into instruction register (IR)
9. Decode the instruction. Fetch operands and operations.
10. Execute the operands with the operations.
11. Store the result back into memory.

Secondary storage

extension of main memory that provides large nonvolatile storage capacity.

Storage Hierarchy

Storage systems organized in hierarchy:

12. speed
13. cost
14. volatility

diagram

Caching

copying information into faster storage system.

15. Registers are considered as a fast cache to memory.
16. Main memory can be viewed as a fast cache for secondary memory.

Example: Instruction cache.
A cache register which contains the next instruction to be executed instead of waiting for fetching the next instruction from memory.

Hardware Protection

(*) Dual-Mode Operation
(*) I/O Protection
(*) Memory Protection

(*) CPU Protection

Dual-Mode Operation

- Sharing system resources requires operating system to ensure that an incorrect program cannot cause other programs to execute incorrectly.

- Provide hardware support to differentiate between at least two modes of operations.

  1. **User mode** - execution done on behalf of a user.
  2. **Monitor mode** (also supervisor mode or system mode) - execution done on behalf of operating system.

- Mode bit added to computer hardware to indicate the current mode:
  0: monitor
  1: user
- When an interrupt or fault occurs hardware switches to monitor mode.

Privileged instructions can be issued only in monitor mode.

I/O Protection

- All I/O instructions are privileged instructions.

- Must ensure that a user program could never gain control of the computer in monitor mode (i.e., a user program that, as part of its execution, stores a new address in the interrupt vector).

Memory Protection

- Must provide memory protection at least for the interrupt vector and the interrupt service routines.

- In order to have memory protection, add two registers that determine the range of legal addresses a program may access:

1- **base** register - holds the smallest legal physical memory address.
2- **limit** register - contains the size of the range.

- Memory outside the defined range is protected.

diagram

CPU Protection

(*) Timer - interrupts computer after specified period to ensure operating system maintains control.
- Timer is decremented every clock tick.
- When timer reaches the value 0, an interrupt occurs.

(*) Timer commonly used to implement time sharing.

(*) Timer also used to compute the current time.

(*) Load-timer is a privileged instruction.

OPERATING-SYSTEM STRUCTURES

Most operating systems support the following types of **system components**:

- Process Management
- Main-Memory Management
- Secondary-Storage Management
- I/O System Management
- File Management

Process Management

(*) A process is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

(*) The operating system is responsible for the following activities in connection with process management:

- process creation and deletion.
- process suspension and resumption.
- provision of mechanisms for:
    # process synchronization
    # process communication

Main-Memory Management

(*) Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

(*) Main memory is a volatile storage device. It loses its contents in the case of system failure.

(*) The operating system is responsible for the following activities in connection with memory management:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes to load when memory space becomes available.
- Allocate and deallocate memory space as needed.

Secondary-Storage Management

(*) Since main memory (primary storage) is volatile and too small to accommodate all data and pro-grams permanently, the computer system must provide secondary storage to back up main memory.

(*) Most modern computer systems use disks as the principle on-line storage medium, for both pro-grams and data.

(*) The operating system is responsible for the following activities in connection with disk management:

- Free-space management
- Storage allocation
- Disk scheduling

I/O System Management

The I/O system consists of:
- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices

File Management

(*) A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

(*) The operating system is responsible for the following activities in connection with file management:

- File creation and deletion.
- Directory creation and deletion.
- Support of primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- File backup on stable (nonvolatile) storage media.

($$) The program that reads and interprets control statements is called variously:

- control-card interpreter
- command-line interpreter
- shell (in UNIX)

Its function is to get and **execute** the next command statement.

Operating-System Services

(*) Program execution: system capability to load a program into memory and to run it. I/O operations - since user programs cannot execute I/O operations directly, the operating system must provide some means to perform I/O.

(*) File-system manipulation - program capability to read, write, create, and delete files.

(*) Communications - exchange of information between processes executing either on the same computer or on different systems tied together by a network. Implemented via shared memory or message passing.

(*) Error detection - ensure correct computing by detecting errors in the CPU and Memory hardware, in I/O devices, or in user programs.

Additional operating-system functions exist not for helping the user, but rather for ensuring efficient system operation.

(*) Resource allocation - allocating resources to multiple users or multiple jobs running at the same time.

(*) Accounting - keep track of and record which users use how much and what kinds of computer resources for account billing or for accumulating usage statistics.

(*) Protection - ensuring that all access to system resources is controlled.

System Calls

(*) System calls provide the interface between a running program and the operating system.

- Generally available as assembly-language instructions.
- Languages defined to replace assembly language for systems programming languages defined to replace assembly language for systems programming allow system calls to be made directly (e.g., C, Bliss, PL/360).

(*) Three general methods are used to pass parameters between a running program and the operating system:

- Pass parameters in registers.
- Store the parameters in a table in memory, and the table address is passed as a parameter in a register.
- Push (store) the parameters onto the stack by the program, and pop off the stack by the operating system.

System Programs

(*) System programs provide a convenient environment for program development and execution.

They can be divided into:
- File manipulation
- Status information
- File modification
- Programming-language support
- Program loading and execution
- Communications
- Application programs

(*) Most users' view of the operation system is defined by system programs, not the actual system calls.

Chapter 3 + 4

Processes + Threads

Process Concept

(*) An operating system executes a variety of programs:
- Batch system - jobs
- Time-shared systems - user programs or tasks

(*) Textbook uses the terms **job** and **process** almost interchangeably.

(*) ***Process***; a program in execution; process execution must progress in a sequential fashion.

(*) A process includes:
- program counter
- stack
- data section

Process States

(*) 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:

diagram

Process Control Block (PCB) - Information associated with each process.

it is a data structure which contains the information or data about the process, mostly this data structure is a table.

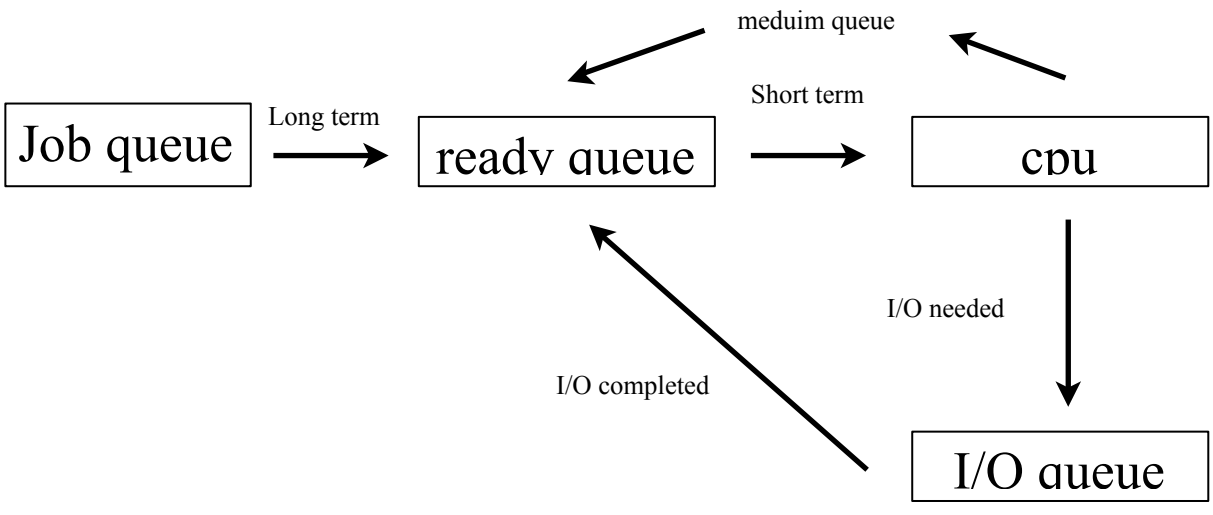
- **Process ID (number)**
- **Process state**
- **Program counter**
- **CPU registers(accumulator, index, stack pointer, …)**
- **CPU scheduling information(process priority, pointers to queues, …)**
- **Memory-management information(base & limit registers, page table, …)**
- **Accounting information**
  - **I/O status information(I-o devices allocated, opened files, …)**

Note: keep in mind that the process is a sequence of cpu execution(burst) and I-O waits.

Process scheduling queues

- **job queue** - set of all processes in the system.
- **ready queue** - set of all processes residing in main memory, ready and waiting to execute.
- **device queues** - set of processes waiting for a particular I/O device.

(*) Process migration between the various queue.



Schedulers

- **Long-term** scheduler (job scheduler) – selects which processes should be brought into the ready queue.

which process will be selected from the job pool to enter memory for execution

- **Short-term** scheduler (CPU scheduler) - selects which process should be executed next and allocates CPU.

- **Medium-term** scheduler – jobs preempted from memory for some reason.

diagram

(*) Short-term scheduler is invoked very frequently (milliseconds)➔(must be fast).

(*) **Long-term scheduler** is invoked very infrequently (seconds, minutes)➔(may be slow).

(*) The long-term scheduler controls the ***degree of multiprogramming***.

degree of multiprogramming: *is the number of jobs in memory (ready queue) for execution*

(*) Processes can be described as either:

I/O-bound process - spends more time doing I/O than computations; many short CPU bursts.

4. **CPU-bound** process - spends more time doing computations; few very long CPU bursts.

note: the OS keep the system on balance of loading, means it tries the best to use all resources of the computer, if the a job in ready queue is mostly IO bound job and IO queue is not busy, then the os chooses it for execution even if there is a CPU bound job ahead of it, same story goes the opposite direction.

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.

(*) Context-switch time is overhead; the system does no useful work while switching.

(*) Time dependent on hardware support.

Job1 CPU Job2

Process Creation

(*) Parent process creates children processes, which, in turn create other processes, forming a tree of processes.

(*) Resource sharing
- Parent and children share all resources.
- Children share subset of parent's resources.
- Parent and child share no resources.

note: the processes in the system compete (fight) for the computer resources (IO, CPU and Memory)

(*) Execution
- Parent and children execute concurrently.
- Parent waits until children terminate.

(*) Address space
- Child duplicate of parent.
- Child has a program loaded into it.

(*) UNIX examples
- **fork** system call creates new process.
- **execve** system call used after a fork to replace the process' memory space with a new pro-gram.

Process Termination

(*) Process executes last statement and asks the operating system to delete it (**exit**).
- Output data from child to parent (via fork).
- Process' resources are deallocated by operating system.

Note: the OS does not allow any child process to continue beyond it's parent.

(*) Parent may terminate execution of children processes (abort).

-Child has exceeded allocated resources.
- Task assigned to child is no longer required.
- Parent is exiting.
  # Operating system does not allow child to continue if its parent terminates.
  # Cascading termination. :if the parent exits then all children and descendant must exit by force.

Cooperating Processes

Concurrent processes are either : (runs in parallel)

(1) ***Independent process*** cannot affect or be affected by the execution of another process.

(2) ***Cooperating process*** can affect or be affected by the execution of another process.

(*) Advantages of process cooperation:
- Information sharing
- Computation speed-up
- Modularity
- Convenience

Producer-Consumer Problem

(*) Talking about concurrency requires :
- Cooperating among processes (i.e. communication between processes)
- Synchronization of processes action.

(*) To illustrate the idea of cooperating process consider the producer-consumer problem ,i.e. Paradigm for cooperating processes;
- ***producer*** process produces information.
- ***consumer*** process consumes this information.

Examples:

* Print program produces characters consumed by the printer.
*Compiler produces assembly code consumed by the assembler.
*Assembler produce object module consumed by the loader.

(*) There must be a buffer of items to be filled by the producer, and then consumed by the consumer.

- unbounded-buffer places no practical limit on the size of the buffer.

- bounded-buffer assumes that there is a fixed buffer size.

Implementation of the producer - consumer problem:
we will use a circular buffer (queue) in the implementation.

empty queue means in pointer points at the same location that out pointer

in: pointer (subscript) where data is entered in the queue
out: pointer (subscript) where data is taken from the queue
n: is the size of the buffer
nextP, nextC: item

6	D
5	C
4	B
3	A (out pointer)
2	EMPTY (in pointer)
1	F
0	E

Full buffer case example n=7

full buffer means in+1 mod n = out

$$$ Shared data

```
const n;
type item = ... ;
var buffer: array [0..n-1] of item;
    in, out: 0..n-1;

in := 0;
out := 0;
```

- Producer process

```
repeat
    ...
    produce an item in nextp
    ...
    while in+1 mod n = out do no-op;
    buffer[in]:=nextp;
    in := in+1 mod n;
until false;
```

- Consumer process

```
repeat
    while in = out do no-op;
```

```
    nextc := buffer[out];
    out := out+1 mod n;
     ...
    consume the item in nextc
    ...
until false;
```

- Solution is correct, but can only fill up n –1 buffer.

Threads

A normal process (heavy weight process) contains:
1. code section
2. data section
3. stack
thread contains
- thread id
- registers
- stack
- pc

Advantages of threads:
- resources sharing
- economy
- system utilization
- parallel processing

(*) A ***thread*** (or lightweight process LWP) is a basic unit of CPU utilization; it consists of:

- program counter
- register set
- stack space

(*) A thread shares with its peer threads its:

- code section
- data section
- operating-system resources

collectively known as a **task**.

(*) A traditional or **heavyweight** process is equal to a task with one thread.

Check book for diagram

There are two kinds of threads:

1. User level: this kind of threads is managed totaly by the programer i.e. the programer or user programs (write the parallel code) for this kind of implementation (DIFFICULT TO PROGRAM)
2. Kernal level: OS level
    a. this kind of thread is managed by the os
    b. most operating system has this kind of support (Linux, windows, ..)

There 3 kind of relationship between user & kernal threads:

1. Many-to-one: (diagram)
    a. It keeps many user threads to one Kernal Thread
    b. There is no parallel processing
    c. If one htread blocks the Kernal thread the system collapses and all other threads are blocked

2. One-to-One: (diagram)
    a. Every user thread is mapped into one kernal thread
    b. parallel processing is allowed
    c. If one kernal blocked, other threads continue working
3. Many-to-many:
    a. It maps(multiplexes) many user threads to equal or less number of kernal threads
    b. parallel processing is allowed
    c. if one thread is blocked, other threads continue working

Chapter 5

Basic Concepts

each process is a sequence of CPU bursts and I/O waits

(*) Maximum CPU utilization obtained with multi-programming.

(*) ***CPU-I/O Burst Cycle*** - Process execution consists of a cycle of CPU execution and I/O wait.

(*) CPU burst distribution (diagram from book)

(*) **Short-term scheduler** -selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.

(*) CPU scheduling decisions may take place when a process:

1. switches from running to waiting state.
2. switches from running to ready state.
3. switches from waiting to ready.
4. terminates.

(*) Scheduling under 1 and 4 is ***nonpreemptive***.

(*) All other scheduling is ***preemptive***.

Dispatcher It is the algorithm which is in charge of the switching process

(*) Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
- switching context
- switching to user mode
- jumping to the proper location in the user pro-gram to restart that program

(*) ***Dispatch latency*** - the time it takes for the dispatcher to stop one process and start another running. it should be as minimum as possible

Scheduling Criteria

(*) CPU utilization - keep the CPU as busy as possible (we need is as **max** as possible)

(*) Throughput - # of processes that complete their execution per time unit (we need is as **max** as possible)

(*) Turnaround time - amount of time to execute a particular process it is the time for submitting your job until it finishes execution (we need is as **min** as possible)

(*) Waiting time - amount of time a process has been waiting in the ready queue (we need is as **min** as possible)

(*) Response time - amount of time it takes from when a request was submitted until the first response is produced, **not** output (for time-sharing environment) (we need is as **min** as possible)

Optimization

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

(1) First-Come, First-Served (FCFS) Scheduling

Example:

Process	Burst time
P_1	24
P_2	3
P_3	3

(*) Suppose that the processes arrive in the order: P_1 , P_2 , P_3 .
Compute Average **waiting time & turnaround time**.

Turnaround time (ATT) = [(24-0) + (3-0) + (3-0)] /3
waiting time = [(24-24) + (27-3) + (30-3)] /3
waiting time = ATT - CPU burst

(*) Suppose that the processes arrive in the order: P_2 , P_3 , P_1 .
Compute Average **waiting time & turnaround time**.

(-) Much better than previous case.
(-) ***Convoy effect***: short process behind long process

(2) Shortest-Job-First (SJF) Scheduling

(*) Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time.

(*) Two schemes:

1. **Non-preemptive** - once CPU given to the process it cannot be preempted until it completes its CPU burst.
2. **Preemptive** - if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is known as the Shortest-Remaining-Time-First (**SRTF**).

(*) SJF is optimal - gives minimum average waiting time for a given set of processes.

Example :

Process	Arrival time	CPU time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

(*) **SJF (non-preemptive)**

Average waiting time = (0 + 6 + 3 + 7)/4 = 4

(*) SRTF (preemptive)

Average waiting time = (9 + 1 + 0 + 2)/4 = 3

Note: if SJF gives the minimum waiting time, why don't we use it?
Problem is starvation!!!
In an SJF if a process has long cpu burst other processes will starve for a long time
Solution: Aging

Major problem is how the OS decides the duration (length) of the next cpu burst for the job?
solution: the OS only estimates the length of the next CPU burst.

How do we know the length of the next CPU burst?

- Can only estimate the length.

- Can be done by using the length of previous CPU bursts, using exponential averaging.

1. T_n = actual length of n th CPU burst
2. Y_n = predicted value of n th CPU burst
3. $1 \geq W \geq 0$
4. Define: $Y_{n+1} = W * T_n + (1 - W) Y_n$

Examples:

(*) W = 0
$Y_{n+1} = Y_n$
Recent history does not count.

(*) W = 1
$Y_{n+1} = T_n$ (better)
Only the actual last CPU burst counts.

(*) If we expand the formula, we get:
$Y_n + 1 = W * T_n + (1 - W) * W * T_{n-1} +$
$(1 - W)^2 * W * T_{n-2} + \ldots + (1 - W)^q * W * T_{n-q}$

So if W = 1/2 ➔ each successive term has less and less weight.

(3) Priority Scheduling

(*) A priority number (integer) is associated with each process.

(*) The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority).
a) preemptive
b) nonpreemptive

(*) ***SJN*** is a priority scheduling where priority is the predicted next CPU burst time.

(*) Problem ≡ ***Starvation*** - low priority processes may never execute.
Solution ≡ ***Aging*** - as time progresses increase the priority of the process.

(4) Round Robin (RR)

(*) Each process gets a small unit of CPU time (***time quantum***), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.

(*) If there are n processes in the ready queue and the time quantum is q , then each process gets 1/n of the CPU time in chunks of at most q time units at once. No

process waits more than (n -1)q time units.

(*) Performance

q large ➔ FIFO
q small ➔ q must be large with respect to context switch, otherwise overhead is too high.

Example of RR with time quantum = 20 :

Process	CPU times
P_1	53
P_2	17
P_3	68
P_4	24

Compute the **average turnaround** and waiting times
ATT= 113.5
AWT= 73

(*) Typically, higher average turnaround than SRT, but better ***response***. Also no starvation!

(5) Multilevel Queue

(*) Ready queue is partitioned into separate queues.
Example: foreground (interactive)
background (batch)

These are just examples, means could be there more than two queues , as much as I want

(*) Each queue has its own scheduling algorithm.
Example: foreground - RR
background - FCFS

(*) Scheduling must be done between the queues.

- Fixed priority scheduling
Example: serve all from foreground then from background.
Possibility of ***starvation***.

- Time slice - each queue gets a certain amount of CPU time which it can schedule amongst its processes.
Example:
80% to foreground in RR
20% to background in FCFS

(*) Two schemes:

1. Preemptive
2. Non-preemptive

(6) Multilevel Feedback Queue

(*) A process can move between the various queues; aging can be implemented this way.

(*) Multilevel-feedback-queue scheduler defined by the following parameters:

- number of queues
- scheduling algorithm for each queue
- method used to determine when to upgrade a process
- method used to determine when to demote a process
- method used to determine which queue a process will enter when that process needs service

Example of multilevel feedback queue:

(*) **Three queues**:
- Q_0 - time quantum 8 milliseconds
- Q_1 - time quantum 16 milliseconds
- Q_2 - FCFS

(*) **Scheduling**

A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q 1 .At Q_1 , job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q 2 .

Algorithm Evaluation

(*) Deterministic modeling(Analytic Evaluation) - takes a particular predetermined workload and defines the performance of each algorithm for that workload.

(*) Queuing models

(*) **Simulation**

(*) Implementation

Chapter 6

Concurrent Processes and Process Synchronization

Concurrent Processes

- Concurrent process and either independent or cooperating
- Independent process : can't affect or be affected by the processors

Precedence Graph:

Given the following statements:

(1) a = x + y
(2) b = z + 1
(3) c = a – b
(4) w = c +1

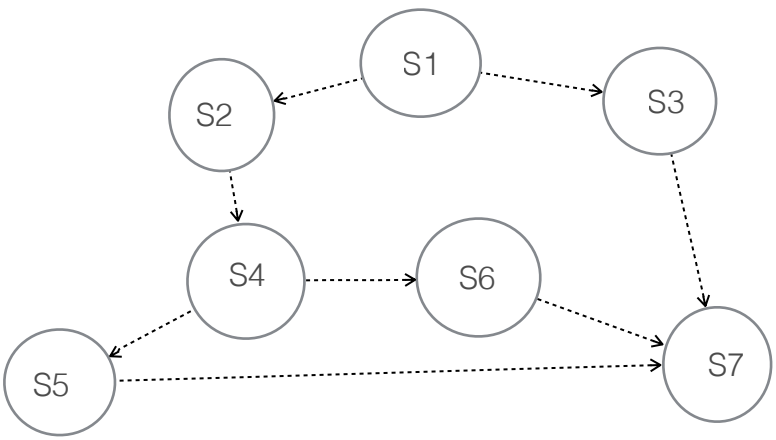
Clearly,

statements (3) & { (1) or (2) }can't executed concurrently.
(4) & (3) can't executed concurrently.
(4) & { (1) or (2) or (3) } can't executed concurrently.

- But statements (1) & (2) can be executed concurrently.

- So if we have multiple functional units in our CPU such as adders or we have multiprocessor system then statements (1) & (2) can be executed concurrently (in parallel).

Definition: A precedence graph is a directed graph whose nodes correspond to statements. An edge from node S_i to node S_j means that S_j is only executed after S_i .



In the given graph:

- S_2 & S_3 can be executed only after S_1 completes
- S_4 can be executed only after S_2 completes.
- S_5 & S_6 can be executed only after S_4 completes.
- S_7 can be executed only after S_5, S_6, S_3 completes.
- S_3 can be executed concurrently with S_2 , S_4 , S_5 , S_6 .

Concurrency Condition

- How do we know if two statements can be executed concurrently and produce the same result?
- **Define:**
  $R(S_i) = \{a_1, a_2, \ldots, a_m\}$ be the **READ** set for statement S_i, which is the set of all variables whose values are **referenced** by statement S_i during execution.
  $W(S_i) = \{b_1, b_2, \ldots, b_n\}$ be the **WRITE** set for statement S_i, which is the set of all variables whose values are **changed** (written) by the execution of statement S_i

  Examples : Given the statements:

- S : c = a – b
  R(S) = {a, b}
  W(S) = {c}

- S : w = c + 1
  R(S) = {c}
  W(S) = {w}

- S: x = x +2
  R(S) = {x}
  W(S) = {x}

- S: read(a)
  R(S) ={a}
  W(S) ={a}

The **Bernstein's** conditions for concurrent statements are:

Given the statements S_1 & S_2 , then S_1 & S_2 can be executed concurrently if:

$R(S_1) \cap W(S_2) = \varnothing$

$W(S_1) \cap R(S_2) = \varnothing$

$W(S_1) \cap W(S_2) = \varnothing$

Example:

Given, $S_1 : a = x + y$

$S_2 : b = z + 1$

$S_3 : c = a + b$

$S_4 : w = c + 1$

$R(S_1) = \{x,y\}$

$W(S_1) = \{a\}$

$R(S_2) = \{z\}$

$W(S_2) = \{b\}$

$\{x,y\} \cap \{b\} = \varnothing$

$\{z\} \cap \{b\} = \varnothing$

$\{a\} \cap \{b\} = \varnothing$

Example:

Given,

$S_3 : c = a - b$

$R(S_3) \cap W(S_2) = \{a, b\} \cap \{b\} \neq \varnothing$

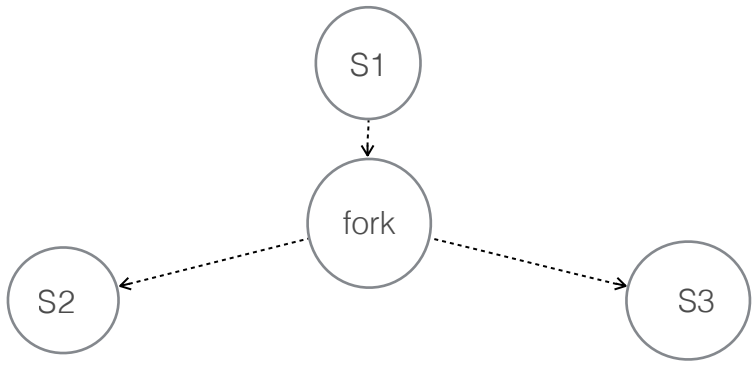
Fork & Join Constructs:

- Precedence graph is difficult to use in Programming Languages, so other means must be provided to specify precedence relation.

- The **Fork L** instruction produces two concurrent executions.
  - One starts at statement labeled L.
  - Other, the continuation of the statement following the fork instruction

Example: The programming. segment corresponds to the precedence graph is:

S_1;
Fork L;
S_2 ; (cpu1)
.
:
L: S_3 ; (cpu2)



(*) When the fork L statement is executed, a new computation is started at S_3 which is executed concurrently with the old computation, which continues at S_2. That is, the fork statement splits one single corporation into two independent computation, hence the name Fork

- The join instruction recombine two concurrent computation. Each computation must ask to be joined.

  Since the two computations executes at different speeds, the statement which executes the join **first** is terminated first, while the second in allowed to continue.

- For 3 computations, two in terminated while the third continues.

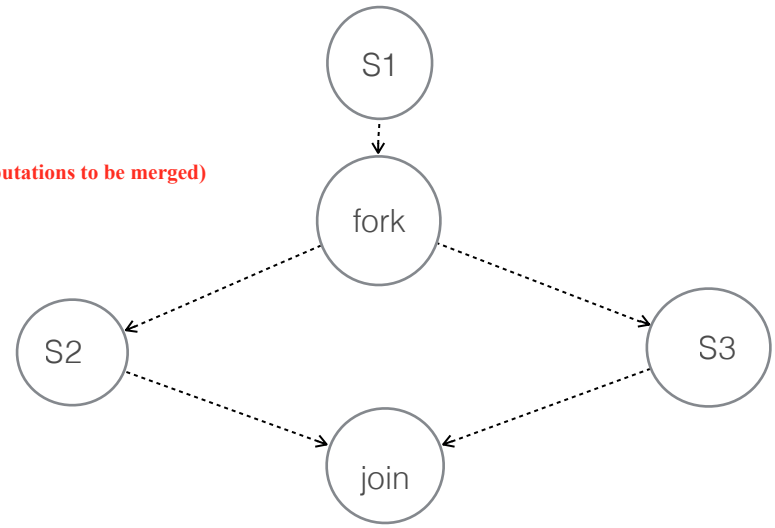
- If **count** is number of computations to join, then the execution of the **join** has the effect

  count = count – 1;
  If count ≠ 0 then quit (quit this computation**)**

The join statement for two computations is executed atomically, i.e. can't be executed concurrently but in a sequential manner, because this might affect **count** giving a wrong result.
For example, if both decrement **count** at same time then count = 0, and the computation **dues not quit.**

- For two processes:

Count =2 (Number of computations to be merged)
Fork L_1;
.
:
S_1 ;
goto L_2;
L_1 : S_2
L_2 : join count



- Let us go back to out four statements in the beginning of this chapter. Using fork & join, this will look lila:

count =2;
Fork L_1;
a = x+y;
goto L_2;
L_1 : b=z+1;
L_2 : joint count;
c =a-b;
w =c+1;

- For the precedence graph earlier:

```
        S1 ;
        count  = 3
        Fork L1;
        S2;
        S4
        Fork  L2;
        S5;
        goto L3;
    L2 : S6;
        goto L3;
    L1 : S3;
    L3 : join count ;
        S7;
```

- Another example is to copy a sequential file **f** to **g** using double buffers **r & s**.
- The program can read from **f** & write to **g** concurrently

```
T = some –record-type;
f , g : file of T;
r, s: T
Begin
  reset (f)
  read (f,r);
  while (not eof (f)) do
        begin
          count = 2;
          s: = r;
          Fork L1;
          Write (g, s);
          goto L2;
       L1: read (f,r);
       L2: join count;
         End;
       Write (g,r);
End;
```

```
Another Way to copy:
T = some –record-type;
f , g : file of T;
r, s: T
Begin
  reset (f)
  read (f,r);
  while (not eof (f)) do
        begin
          s: = r;
         Write (g, s);
         read (f,r);
       End;
      Write (g,r);
End;
```

The concurrent statement:

- The fork & join instructions are powerful means of writing concurrent programs, unfortunately, it is clumsy and very difficult to keep track, because the fork is similar to goto statements.
- A higher–level language constructs for specifying concurrency due to Dijkstra using the notations: parbegin / parend

Example:

```
S0;
Parbegin
    S1;
    S2;
     :
    Sn;
Parend;
Sn+1;
```

- All statements enclosed between parbegin and parend can be executed concurrently

(*) In our pervious example,

```
parbegin
    a  = x+y;
    b  = z+1;
parend ;
   c =a-b;
```

```
   w =c+1;
```

(*) **In the example:**

```
S1;
parbegin
    S3;
     begin
        S2;
        S4;
        parbegin
           S5;
           S6;
        parend;
   end;
parend;
S7;
```

(*) For the files copying files :

```
begin
    reset (f);
    read (f, r);
    while (not eof (f)) do
       begin
         S = r;
         parbegin
              write (g, s);
              read (f, r);
          parend;
       end;
      write (g,r);
end;
```

Process Synchronization

Background

- **Process Cooperation**
  - Information Sharing
  - Computation Speedup
  - Modularity
  - Convenience

Example : **Producer-Consumer** problem , the bounded buffer problem:

Data Structure used:

```
item . . ; //can be of any data type
item buffer[n], nextp , nextc;
int in = 0, out = 0;
```

Producer:	**Consumer:**

do
{ ...
produce an item in nextp
...
while ((in+1)%n ==out)
no-op; // full buffer
buffer[in] = nextp;
in = (in + 1) % n;
}
while true;
	do
	{ while (in == out)
	no-op; // empty buffer
	nextc = buffer[out];
	out = (out + 1)% n;
	...
	consume the item in nextc
	...
	}
	while true;

- Shared memory solution to bounded buffer problem discussed before allows at most **n - 1** items in buffer at the same time.

- Suppose that we modify the producer consumer code by adding a variable ***counter***, initialized to 0 and incremented each time a new item is added to the buffer, and decremented each time an item is taken from the buffer.

Bounded-Buffer

Data Structure used:

```
item . . ; //can be of any data type
item buffer[n], nextp , nextc;
int in = 0, out = 0;
int counter = 0;
```

Producer:	**Consumer:**

do
{ ...
produce an item in nextp
...
while (counter == n)
no-op;
buffer[in] = nextp;
in = (in + 1) % n;
counter = counter + 1;
}
while true;
	do
	{ while (counter == 0)
	no-op;
	nextc = buffer[out];
	out = (out + 1)% n;
	counter = counter - 1;
	...
	consume the item in nextc
	...
	}
	while true;

- Counter = counter + 1; could be implemented as

```
register1 = counter
```

```
register1 = register1 + 1
counter = register1
```

- Counter = counter – 1; could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving:

```
S0: producer execute register1 = counter   {register1 = 5}
S1: producer execute register1 = register1 + 1   {register1 =
6}
S2: consumer execute register2 = counter   {register2 = 5}
S3: consumer execute register2 = register2 - 1   {register2 =
4}
S4: producer execute counter = register1   {count = 6 }
S5: consumer execute counter = register2   {count = 4}
```

  - No problems if there is a strict alternation of the **consumer** and **producer** processes

Problems with Bounded-Buffer with Counter

- Concurrent access to shared data may result in data inconsistency.

- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

- The statements:

- **counter = counter +1;**
- **counter = counter 1;**

must be executed ***atomically***.

Atomically: If one process is modifying counter the other process must wait, that is, as if this is executed sequentially.

The Critical Section Problem

The Problem with Concurrent Execution

- ***Concurrent processes (or threads) often need access to shared data and shared resources.***
- ***If there is no controlled access to shared data, it is possible to obtain an inconsistent view of this data.***
- ***Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.***

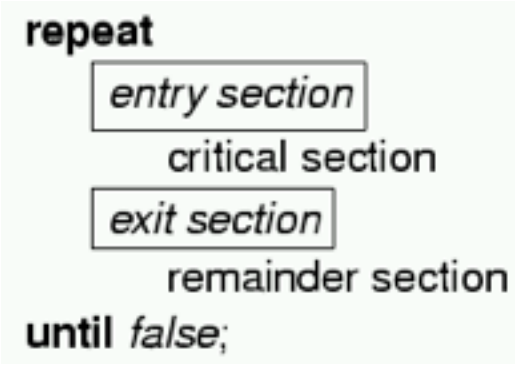
Race Condition: A situation in where several processes access and manipulate data concurrently and the outcome of execution depends on the particular order in which the access takes place.

- *n* processes all competing to use some shared data

- Each process has a code segment, called *critical section*, in which the shared data is accessed.

- Problem - ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

Structure of process P_i



Solution 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 exist 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.

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

Solution to Critical Section Problem

Types of Solutions

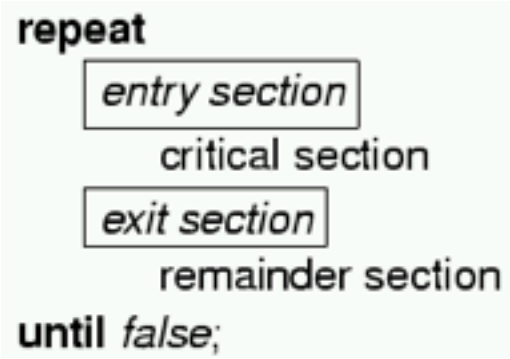
- ***Software solutions***
  - Algorithms whose correctness does not rely on any assumptions other than positive processing speed (that may mean no failure).
  - Busy waiting.

- ***Hardware solutions***
  - Rely on some special machine instructions.

- ***Operating system solutions***
  - Extending hardware solutions to provide some functions and data structure support to the programmer.

SOFTWARE SOLUTION

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)



- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables: -

```
   int turn;   //turn can have a value of either 0 or 1
               //if turn = i, P(i) can enter it's critical
section
```

Process P_i

```
do
{
   while (turn != i) /*do nothing*/ ;

   critical section

   turn = j;

   remainder section
}
while (true)
```

Process P_j

```
do
{
   while (turn != j) /*do nothing*/ ;

   critical section

   turn = i;

   remainder section
}
while (true)
```

- Mutual exclusion ok.

- **Bounded waiting** ok - each only waits at most 1 go.

- Progress not good - each has to wait 1 go. P_0 gone into its (long) remainder, P_1 executes critical and finishes its (short) remainder long before P_0, but still has to wait for P_0 to finish and do critical before it can again.
Strict alternation not necessarily good - Buffer is actually pointless, since never used! Only ever use 1 space of it.

Algorithm 2

- Shared variables

```
boolean flag[2];
flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to  enter its  critical
section
```

Process P_i

```
do
 {  flag[i]= true;
      while (flag[j]) /*do nothing*/ ;

    critical section

   flag[i]=false;

    remainder section
 }
 while (true)
```

Process P_j

```
do
 {  flag[j]= true;
      while (flag[i]) /*do nothing*/ ;

    critical section

   flag[j]=false;

    remainder section
 }
 while (true)
```

- Doesn't work at all. Both flags set to true at start. "After you." "No, after you." "I insist." etc.

- Infinite loop.

Algorithm 3

Combined shared variables of algorithms 1 and 2.

```
int turn;     //turn can have a value of either 0 or 1
boolean flag[2];     flag[0] = flag[1] = false;
// if flag[i] == true, P(i) ready to  enter its  critical      section
```

Process P_i

```
do
 { flag[i]= true;
   turn = j;
   while (flag[j] && turn==j) /*do nothing*/ ;

   critical section

  flag[i]=false;

   remainder section
 }
 while (true)
```

Process P_0

```
do
{ flag[0]= true;
  turn = 1;
  while (flag[1] && turn==1)
     /*do nothing*/ ;

  critical section

  flag[0]=false;

  remainder section

 } while (true)
```

Process P_1

```
do
{flag[1]= true;
 turn = 0;
 while flag[0] && turn==0)
    /*do nothing*/ ;

 critical section

 flag[1]=false;

remainder section

} while (true)
```

- Meets all three requirements; solves the critical section problem for two processes.
- "flag" maintains a truth about the world - that I am at start/end of critical. "turn" is not *actually* whose turn it is. It is just a variable for solving conflict if two processes are ready to go into critical. They all give up their turns so that one will win and go ahead.
- e.g. flags both true, turn=1, turn=0 lasts, P_0 runs into critical, P_1 waits.
  Eventually P_0 finishes critical, flag =false, P_1 now runs critical, even though turn is still 0.
  Doesn't matter what turn is, each can run critical so long as other flag is false. Can run at different speeds.
- If other flag is true, then other one is either *in* critical (in which case it will exit, you wait until then) or at start of critical (in which case, you both resolve conflict with turn).

Note the problem with software solution is the busy waiting, the cpu is used just to make each process wait for nothing and make the cpu busy while the process is not doing anything.

Bakery Algorithm(READ IT-ONLY)

Introduction

This algorithm solves the critical section problem for ***n*** processes in software. The basic idea is that of a bakery; customers take numbers, and whoever has the lowest number gets service next. Here, of course, "service" means entry to the critical section.

Critical section for *n* processes

- Generalization for n processes.
- Each process has an id. Ids are ordered.
- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5...
- Notation <= lexicographical order (ticket #, process id #)
    - (a,b) < (c,d) if a < c or if a = c and b < d
    - $\max(a_0, \ldots, a_{n-1})$ is a number, k , such that $k >= a_i$ for i = 0, . . . , n - 1
- Shared data

```
1   boolean choosing[n]; //initialise all to false
2   int number[n];  //initialise all to 0

3   do
4   { choosing[i] = true;
5   number[i] = max(number[0], number[1], ...,number[n-1]) + 1;
6   choosing[i] = false;
7   for(int j = 0; j < n; j++)
8     { while (choosing[j]== true)
9         /*do nothing*/
10      while ((number[j]!=0) && (number[j],j)< (number[i],i))
11          /*do nothing*/
12    }

13        critical section

14  number[i] = 0;

15        remainder section

    } while (true)
```

Comments

lines 1-2: Here, *choosing*[*i*] is true if P_i is choosing a number. The number that P_i will use to enter the critical section is in *number*[*i*]; it is 0 if P_i is not trying to enter its critical section.

lines 4-6: These three lines first indicate that the process is choosing a number (line 4), then try to assign a unique number to the process P_i (line 5); however, that does not always happen. Afterwards, P_i indicates it is done (line 6).

lines 7-12: Now we select which process goes into the critical section. Pi waits until it has the lowest number of all the processes waiting to enter the critical section. If

two processes have the same number, the one with the smaller name - the value of the subscript - goes in; the notation "(a,b) < (c,d)" means true if a < c or if both a = c and b < d (lines 9-10). Note that if a process is not trying to enter the critical section, its number is 0. Also, if a process is choosing a number when P_i tries to look at it, P_i waits until it has done so before looking (line 8).

line 14: Now P_i is no longer interested in entering its critical section, so it sets *number*[*i*] to 0.

Drawbacks of Software Solutions

- **Complicated to program**
- **Busy waiting (wasted CPU cycles**
- **It would be more efficient to *block* processes that are waiting (just as if they had requested I/O).**

HARDWARE SOLUTION

Hardware Solution ***Disable Interrupts***

On a uni-processor, you can get mutual exclusion by locking out interrupts. Observations:

- You can only afford to do this for a little while, so you don't lose any interrupts (of course in general you don't want to protect expensive things with spin locks).
- Nothing else works if you're sharing memory with a device you sure can't use a spin lock! (DEADLOCK).
- Correct solution for a uni-processor machine, but this doesn't work on multiprocessors, the solution is not correct.
- During critical section multiprogramming is not utilized - performance penalty.

Repeat
disable interrupts
critical section
enable interrupts
remainder section
Forever

Hardware Solution ***Test and Set***

Use better (more powerful) atomic operations:

- **Test and modify the content of a word atomically.**

```
boolean Test_and_Set(Boolean & target)

{boolean test = target;
  target = true;
 return test;
}
```

- Shared data: `boolean lock = false;`

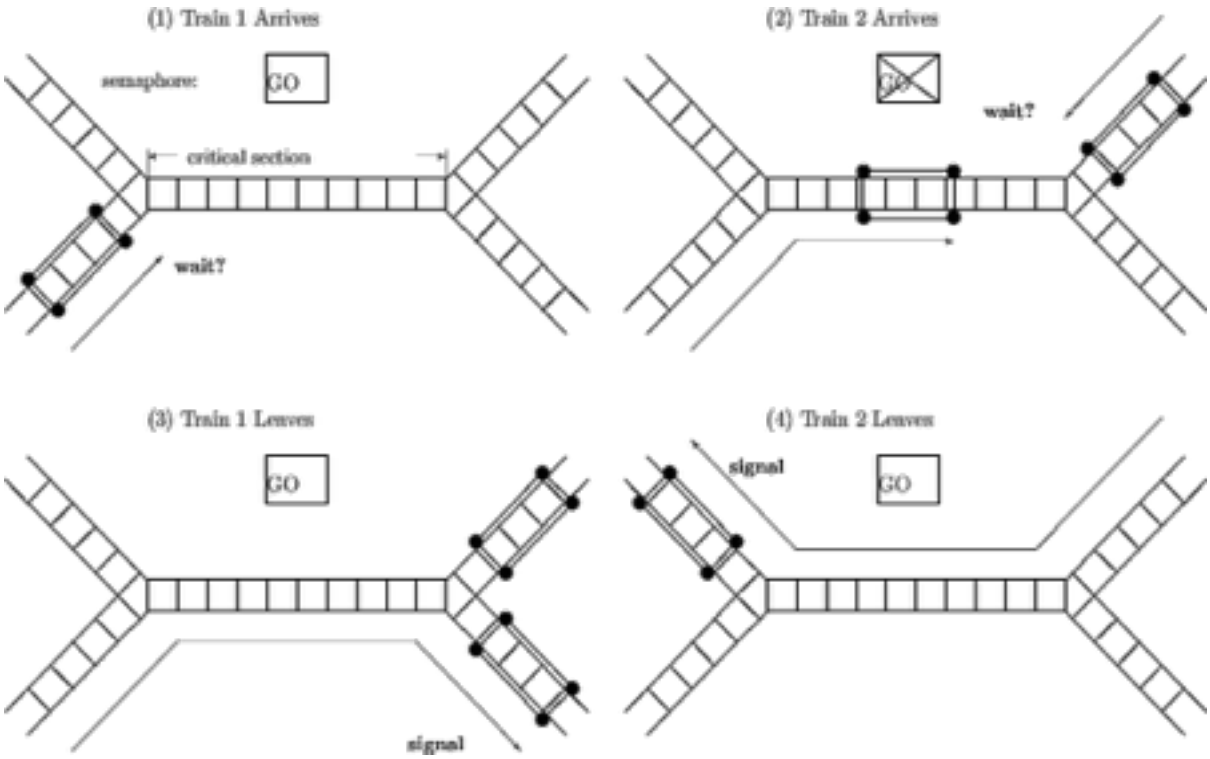
Process P_i

```
do
{ while (Test-and-Set(lock))
       /*do nothing*/ ;
       critical section
  lock = false;
     remainder section
}while (true)
```

OPERATING SYSTEM SOLUTION

Semaphores

Semaphore: wait and signal



Semaphore S - integer variable

- can only be accessed via two indivisible (**atomic**) operations

```
                S =1
wait(s)    : while (S<=0)   /*do nothing*/   ;
             S = S-1;

signal(S)  : S = S + 1;
```

```
mutex : semaphore =1;
Repeat
  wait( mutex );
       critical section
  signal( mutex );
       remainder section
Forever
```

Semaphore Implementation

- Define a semaphore as a record/structure

```
struct semaphore
{ int value;
  List *L;   //a list of processes
}
```

- Assume two simple operations:
    - ***block*** suspends the process that invokes it.
    - ***wakeup(P)*** resumes the execution of a blocked process *P*.
- Semaphore operations now defined as

```
wait(S)
{ S.value = S.value -1;
  if (S.value <0)
    { add this process to S.L;
      block;
    }
}

signal(S)
{ S.value = S.value + 1;
  if (S.value <= 0)
    { remove a process P from S.L;
      wakeup(P);
     }
 }
```

Classical Problems of Synchronization

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

Bounded Buffer Problem

- Shared data

```
char item;                // could be any data type
char buffer[n];
semaphore full = 0;       // counting semaphore
semaphore empty = n;      // counting semaphore
semaphore mutex = 1;      // binary semaphore
char nextp, nextc;
```

- Producer process

```
do
{ produce an item in nextp
  wait (empty);
  wait (mutex);
  add nextp to buffer //Critical section
  signal (mutex);
  signal (full);
}
while (true)
```

- Consumer process

```
do
{ wait( full );
  wait( mutex );
  remove an item from buffer to nextc
  signal( mutex );
  signal( empty );
  consume the item in nextc;
}
```

Readers-Writers Problem

diagram

- Shared data

```
semaphore mutex = 1;
semaphore wrt = 1;
int readcount = 0;
```

- Writer process

```
wait(wrt);
  writing is performed
signal (wrt);
```

- Reader process

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

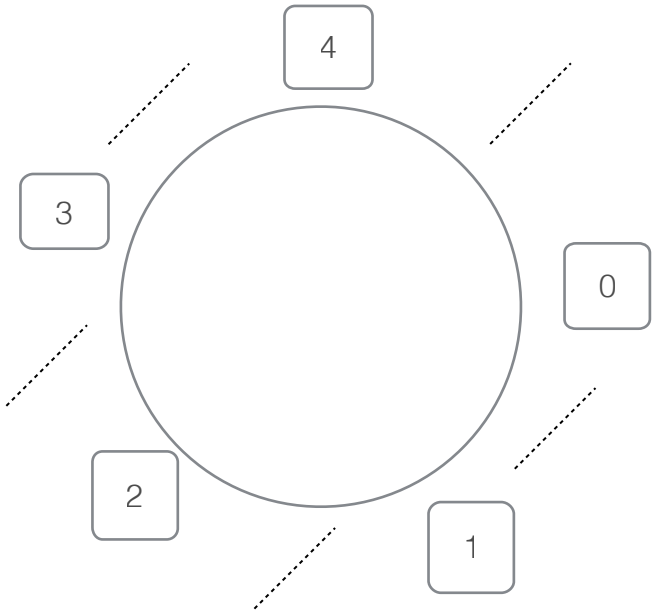
Dining Philosopher Problem

- Shared data

```
semaphore chopstick[5];
chopstick[] = 1;
```

- Philosopher *i*:

```
do
{ wait (chopstick[i]);
  wait (chopstick[i+1 mod 5]);
    eat;
  signal (chopstick [i]);
  signal (chopstick [i+1 mod 5]);
    think;
}
while (true)
```



Problem: DeadLock

Chapter 7

DEADLOCKS

- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock
- Combined Approach to Deadlock Handling

The Deadlock Problem

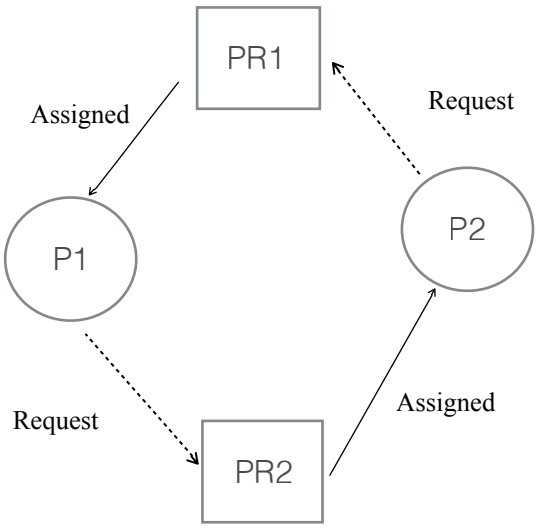
* A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

A deadlock state is simply a set of blocked processes each process is holding (reserving) some resources and waiting for other processes to release their resources and so on.

*** Example**

- System has 2 tape drives.
- *P* 1 and *P* 2 each hold one tape drive and each needs another one.

Full cycle



Example: bridge crossing

- Traffic only in one direction.

- Each section of a bridge can be viewed as a resource.

- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).

- Several cars may have to be backed up if a deadlock occurs.

- Starvation is possible.

Solution: give priority to one direction, but this will make starvation possible
Another solution: Alternating direction priority, but still not 100% solution.

System Model

* Assume Resource types $R_1, R_2, \ldots, R_{\mu-1}$:
CPU cycles, memory space, I/O devices

* Each resource type R_i has W_i instances (W_i number of instances from resource type R_i).
R7 = Hard Disk
W7 = 25
We have 25 Hard Disk

* Each process utilizes a resource as follows:
  - request
  - use
  - release

Deadlock Characterisation–deadlock can arise if four conditions hold **simultaneously**.
Necessary Conditions for a deadlock to occur:
- **Mutual exclusion:** only one process at a time can use a resource.(No sharing)

- **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 $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_n -1 is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P

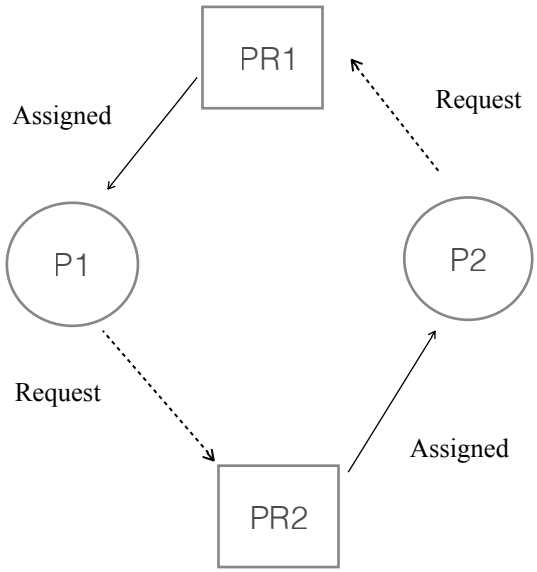
Resource-Allocation Graph – a set of vertices V and a set of edges E.

(*) V is partitioned into two types:
- $P = \{P\,1, P\,2, ..., P\,n\}$, the set consisting of all the processes in the system.

- $R = \{R\,1, R\,2, ..., R\,m\}$, the set consisting of all resource types in the system.

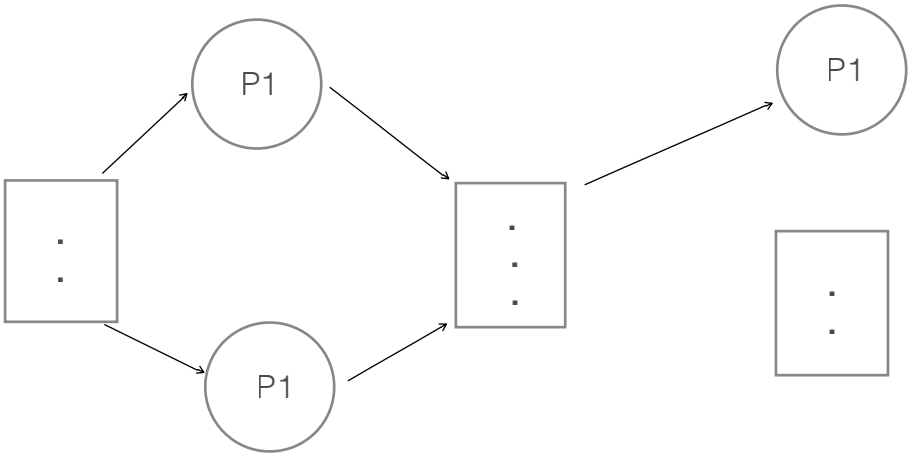
(*) *request edge* – directed edge $P\,i \rightarrow R\,j$

(*) *assignment edge* – directed edge $R\,j \rightarrow P\,i$



Example

- Process



- Resource type with 4 instances

- P_i requests instance of R_j

- P_i is holding an instance of R_j

Example of a resource-allocation graph with no **cycles.**

Example of a resource-allocation graph with a cycle.

(*) If graph contains no cycles ⇒ no deadlock.

(*) If graph contains a cycle ⇒

- if only one instance per resource type, then deadlock.

- if several instances per resource type, possibility of deadlock**.**

Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state.

- Allow the system to enter a deadlock state and then recover. (preemption)

- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX.

Deadlock Prevention – restrain the ways resource requests can be made.
the idea is to make sure at least **one** of the 4 necessary conditions must not hold.

* ***Mutual Exclusion*** – not required for sharable resources; must hold for nonsharable resources.

* ***Hold and Wait*** – must guarantee that whenever a process requests a resource, it does not hold any other resources.

  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none.

  - Low resource utilization; starvation possible.

* **No Preemption** –

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.

- Preempted resources are added to the list of resources for which the process is waiting.

- Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

* **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration**.**

Deadlock prevention tries to ensure one of the 4 necessary conditions do not hold in order to prevent a deadlock state. A side effect of a deadlock prevention:

1. low device utilization
2. reduce system throughput.

Deadlock Avoidance – requires that the system has some additional *a priori* information available.

The operating makes sure the system will never enters a deadlock state

Def: A safe sequence is a sequence of processes {P_0,P_1,P_2,P_3,P_4,P_{n-1}} such that process Pi request oaf resources can be satisfied by:

The currently available resources + the resource released

by process j such that j<i

Def: If there such safe sequence, there no deadlock, otherwise the sequence is unsafe and a deadlock might occur.

(*) Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.

(*) The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.

(*) Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

Safe State – when a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state.

(*) System is in safe state if there exists a *safe sequence* of all processes.

(*) Sequence $<P_1, P_2, ..., P_n>$ is safe if for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus the resources held by all the P_j, with $j < i$.

- If P_i resource needs are not immediately avail-able, then P_i can wait until all P_j have finished.

- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate.

- When P_i terminates, P_{i+1} can obtain its needed resources, and so on.

(**) If a system is in safe state ⇒ no deadlocks.

(**) If a system is in unsafe state ⇒ possibility of deadlock.

(**) Avoidance ⇒ ensure that a system will never enter an unsafe state.

example:
A system with 12 tape drivers and 3 processes

Process	Max needs	Allocated now	Current needs
P0	10	5	5
P1	4	2	2
P2	9	2	7

the available tape driver w=3
solution
at this time the sequence <P1,P0,P2> is a safe sequence

example 2:
assume p2 requested an additional tape driver OS granted

Process	Max needs	Allocated now	Current needs
P0	10	5	5
P1	4	2	2
P2	9	~~2~~ 3	~~7~~ 6

is the sequence safe
solution
no it is not

Resource-Allocation Graph Algorithm

(*) Claim *edge* $P\,i \rightarrow R\,j$ indicates that process $P\,i$ may request resource $R\,j$; represented by a dashed line.

(*) Claim edge converts to request edge when a process requests a resource.

(*) When a resource is released by a process, assignment edge reconverts to a claim edge.

(*) Resources must be claimed *a priori* in the system.

Banker's Algorithm (Avoidance Algorithm)

The algorithm takes the flowing in consideration:
1. for simplicity we will consider only one resource type, but this can be generalised to more than one resource type
2. each process must declare the max needs from the beginning
3. when a process requests a resource it may have to wait
4. when a process gets all it's resources it must return them in a finite amount of time

(*) Multiple resource types.

(*) Each process must *a priori* claim maximum use.

(*) When a process requests a resource it may have to wait.

(*) When a process gets all its resources it must return them in a finite amount of time.

Data Structures for the Banker's algorithm where:

n = number of processes, and m = number of resource types.

Array MAX has the max numbers of each process

MAX[i] = k means process Pi needs a maximum k instances of the resources

Array Allocaation

Allocation[i] = k means process Pi is currently allocated k instances of the resources

Array Need

Need[i] = k means process Pi needs k more instances of the resources

Available = w is how many recourses

- ***Available***: Vector of length m. I*f Available*[j] = *k,* there are *k* instances of resource type *R j* available.

- ***Max***: *n* × *m* matrix. If *Max*[*i,*j]=*k,* then process *P i* may request at most *k* instances of resource type *R j* .

- ***Allocation***: *n* × *m* matrix. If *Allocation*[*i,*j]=*k,* then *P i* is currently allocated *k* instances of *R j* .

- ***Need***: *n* × *m* matrix. If *Need*[*i,*j]=*k,* then *P I* may need *k* more instances of *R j* to complete its task. *Need*[*i,*j]=*Max*[*i,*j] – *Allocation*[*i,*j].

Safety Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively.
   Initialize:
   Work := *Available*
   Finish[i]:=*false* for *i* = 1, 2, ..., *n.*

2. Find an *i* such that both:
   a. *Finish*[i]=*false*
   b. *Need i* ≤ *Work*

If no such *i* exists, go to step 4.

3. *Work* := *Work* + *Allocation i*
Finish[i]:=*true*
go to step 2.

4. If *Finish*[i]=*true* for all *i,* then the system is in safe state.

May require an order of $m \times n^2$ operations to decide whether a state is safe.

1. k array of integers.
Initialize:
w = *Available*
k[i]:=1 for *i* = 1, 2, ..., *n.*

2. Find an *i* such that both:
a. *k*[i]=1
b. *Need[i]*≤ *w*
If no such *i* exists, go to step 4.

3. *w*= w + *Allocation [i]*
k[i]:=0
go to step 2.

4. If *k*[i]=0 for all *i,* then the system is in safe state.

Resource-Request Algorithm for process *P i*

Request i = request vector for process *P i* . f *Request i* [*j*] = *k* , then process *P i* wants *K* nstances of resource type *R j* .

1. If *Request i* ≤ *Need i* , go to step 2. Otherwise, aise error condition, since process has exceeded its maximum claim.

2. If *Request i* ≤ *Available,* go to step 3. Other-wise, *P i* must wait, since resources are not available.

3. Pretend to allocate requested resources to *P I* by modifying the state as follows:

Available := *Available* – *Request i* ;
Allocation i := *Allocation i* + *Request i* ;
Need i := *Need i* – *Request i* ;

- If safe ⇒the resources are allocated to *P i* .
- If unsafe ⇒ *P i* must wait, and the old resource-allocation state is restored.

Example of Banker's algorithm

(*) 5 processes *P* 0 through *P* 4 ; 3 resource types *A* (10 instances), *B* (5 instances), and *C* (7 instances).

(*) Snapshot at time *T* 0 :

	Allocation	*Max*	*Available*	*Need*
	A B C	*A B C*	*A B C*	*AB C*
P 0	0 1 0	7 5 3	3 3 2	7 4 3
P 1	2 0 0	3 2 2		1 2 2
P 2	3 0 2	9 0 2		6 0 0
P 3	2 1 1	2 2 2		0 1 1
P 4	0 0 2	4 3 3		4 3 1

(*) Sequence <*P* 1 , *P* 3 , *P* 4 , *P* 2 , *P* 0 > satisfies safety criteria.

(*) *P* 1 now requests resources.
Request 1 = (1,0,2).

- Check that *Request* 1 ≤ *Available* (that is,
(1,0,2) ≤ (3,3,2)) ==> true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P 0	0 1 0	7 4 3	2 3 0
P 1	3 0 2	0 2 0	
P 2	3 0 2	6 0 0	
P 3	2 1 1	0 1 1	
P 4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence <*P* 1 , *P* 3 , *P* 4 , *P* 0 , *P* 2 > satisfies safety requirement.

(*) Can request for (3,3,0) by *P* 4 be granted?

(*) Can request for (0,2,0) by *P* 0 be granted?

//HE DID NOT EXPLAIN(START)

Deadlock Detection

(*) Allow system to enter deadlock state

(*) Detection algorithm

(*) Recovery scheme

Single Instance of Each Resource Type

(*) Maintain *wait-for* graph

- Nodes are processes.
- P_i ---> P_j if P_i is waiting for P_j .

(*) Periodically invoke an algorithm that searches for a cycle in the graph.

(*) An algorithm to detect a cycle in a graph requires n order of n 2 operations, where n is the number of vertices in the graph.

Several Instances of a Resource Type

(*) Data structures

- *Available:* A vector of length m indicates the number of available resources of each type.

- *Allocation:* An $n \times m$ matrix defines the number of resources of each type currently allo

diagram

- *Request:* An $n \times m$ matrix indicates the current request of each process. If *Request*[*i*,j]=*k,* then process P_i is requesting k more instances of resource type R_j .

Detection Algorithm

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize:

- *Work* := *Available.*
- For *i* = 1, 2, ..., *n,*
  if *Allocation i* ,
  then *Finis*h[i]:=*false;*
  otherwise, *Finis*h[i]:= *true.*

2. Find an index *i* such that both:

a. *Finis*h[i]=*false.*
b. *Request i* ≤ *Work.*

If no such *i* exists, go to step 4.

3. *Work* := *Work* + *Allocation i*
*Finis*h[i]:=*true*

go to step 2.

4. If *Finis*h[i] = false, for some *i,* 1 ≤ *i* ≤ *n,* then the system is in a deadlock state.
Moreover, if *Finis*h[i]=*false,* then *P i* is deadlocked.

(*) Algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

diagram

Example of Detection algorithm

(*) Five processes *P* 0 through *P* 4 ; three resource types *A* (7 instances), *B* (2 instances), and *C* (6 instances).

(*) Snapshot at time *T* 0 :

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P 0	0 1 0	0 0 0	0 0 0
P 1	2 0 0	2 0 2	
P 2	3 0 3	0 0 0	
P 3	2 1 1	1 0 0	

P 4 0 0 2 0 0 2

(*) Sequence <*P* 0 , *P* 2 , *P* 3 , *P* 1 , *P* 4 > will result in *Finis*h[i] = true for all *i.*

(*) *P* 2 requests an additional instance of type *C.*

	Request

	A B C
P 0	0 0 0
P 1	2 0 2
P 2	0 0 1
P 3	1 0 0
P 4	0 0 2

(*) State of system?

- Can reclaim resources held by process *P* 0 , but insufficient resources to fulfill other processes' requests.

- Deadlock exists, consisting of processes *P* 1 , *P* 2 , *P* 3 , and *P* 4 .

diagram

Detection-Algorithm Usage

(*) When, and how often, to invoke depends on:

- How often a deadlock is likely to occur?

- How many processes will need to be rolled back?
  one for each disjoint cycle

(*) If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes"caused" the deadlock.

Recovery from Deadlock

(*) Process termination

- Abort all deadlocked processes.

- Abort one process at a time until the deadlock cycle is eliminated.

- In which order should we choose to abort?

  * Priority of the process.
  * How long process has computed, and how much longer to completion.
  * Resources the process has used.
  * Resources process needs to complete.
  * How many processes will need to be terminate
  * Is process interactive or batch?

(*) Resource Preemption

- Selecting a victim – minimize cost.

- Rollback – return to some safe state, restart process from that state.

- Starvation – same process may always be picked as victim; include number of rollback in cost factor.

Combined Approach to Deadlock Handling

(*) Combine the three basic approaches (prevention, avoidance, and detection), allowing the use of the optimal approach for each class of resources in the system.

(*) Partition resources into hierarchically ordered classes.

(*) Use most appropriate technique for handling deadlocks within each class.

//HE DID NOT EXPLAIN(END)

Chapter 8

Memory Management

Introduction & Background

(*) Recall: Memory is an array of words, each with an address. CPU fetches instructions and store results from/to memory

(*) Program must be brought into memory (limited) as a process for execution. (We assume all the program must be brought into memory before execution starts)

(*) Job queue - collection of processes on the disk that are waiting to be brought into memory for execution.

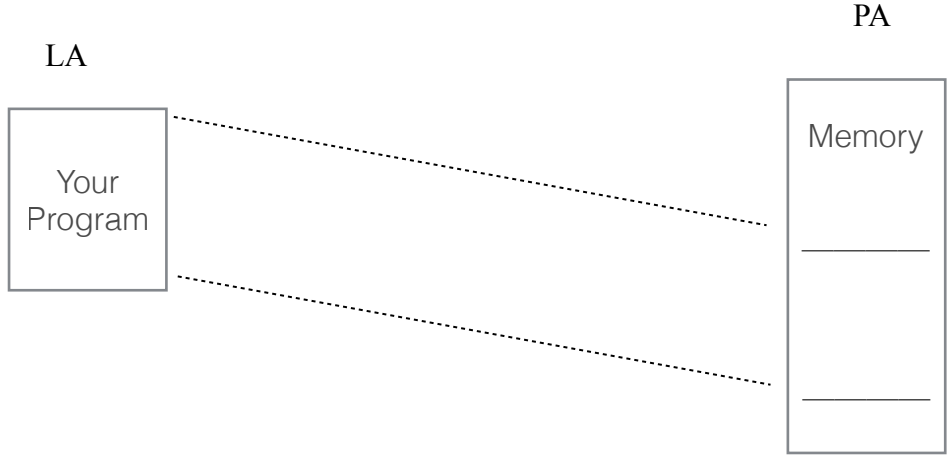
(*) In this chapter, we assume the whole program must be in memory for execution.

(*) Except ***paging***, we assume the program must reside contiguously in memory.

Logical-Virtual-(LA) versus Physical(PA) 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.



Memory-Management Unit (**MMU**)hardware device that maps virtual to physical address.

(*) In **MMU** scheme, 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.

(*) **Address binding** of instructions and data to memory addresses can happen at three stages: (At what time of execution PAs are assigned to LAs)

Compile time: If memory location known a priori, absolute code can be generated; must recompile code if starting location changes.
Not practical in general purpose OS

Load time: Must generate ***relocatable*** code if memory location is not known at compile time. (relocatable for example address 14 from the beginning of the program).
This ok, but theOS can't move the program during execution.

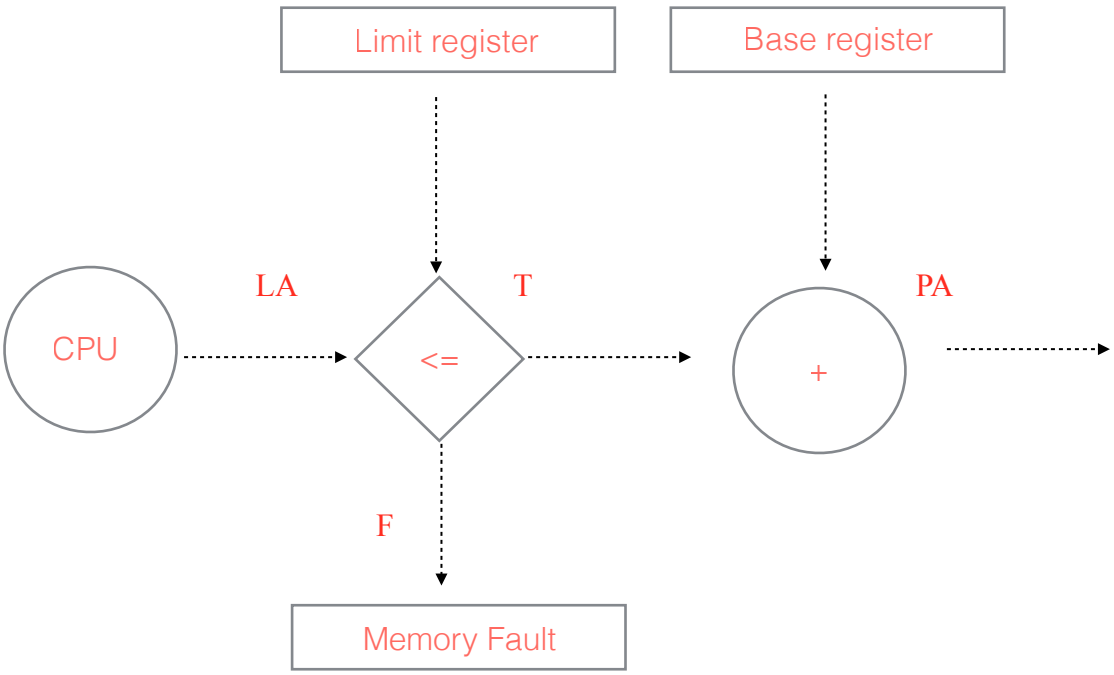
Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., ***base*** and ***limit*** registers).
Which the program can change it's location during execution

Contiguous Allocation – Multiple Partitions

- Program resides contiguously in memory
- Memory is divided into a number of regions(partitions). (have not to be equaled in size)
- When a region becomes free, a process is loaded into it.
- Each program os loaded into one region only
- Hardware: ***base*** and ***limit*** registers.

OS
Region 1
Region 2
Region 3
Region 4

Base register : the address of the beginning of the region
Limit register : the number of the size of the region



PA=LA+Base Register

Fixed Regions

(*) Memory is divided into a fixed number of regions(partitions), with fixed - not necessary equally- sizes.
(*) Degree of multiprogramming(number of executing programs in the memory) is bounded by the number of regions.
(*) Job scheduling:

(a) Each region has a separate queue , usually FCFS.
(b) Only one queue of waiting jobs:
- **FCFS** with or without skip.
- Best fit only
- Best available fit.

(*) Problems:

- Selection of regions sizes.
- What if one job is very big.
- Internal fragmentation. the remaining unused space in the region
- External fragmentation. the regions that are not used (usually with small regions).

(*) Example: IBM OS/360
(called **MFT** : Multiprogramming with Fixed number of Tasks)

Variable(Dynamic) Regions MVT - Multiprogramming with Variable number of Tasks)

(*) Memory is a set of:

(a) ***Allocated*** regions .
(b) ***Holes*** : 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.

Example: Assume the following system load:
Given the following job queue:

Process	Memory	Time
P1	600MB	10
P2	1000MB	5
P3	300MB	20
P4	700MB	8
P5	500MB	15

Assume we have 2560MB and the OS takes 400MB
At T=0

OS 400
P1 1000
P2 2000
P3 2300
EMPTY 260MB

At T=5

OS 400
P1 1000 (Allocated region)
P4 1700 (Allocated region)
EMPTY 2000 (external fragmentation)
P3 2300 (Allocated region)
EMPTY 256 (external fragmentation)

(*) Operating system maintains information about:
- **allocated** partitions
- **free** partitions (hole)

(*) Hardware Needed: ***base*** and ***limit*** registers.

(*) Question: 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.

(*) Problems:

- **External fragmentation** - total memory space exists to satisfy a request, but it is not contiguous.

Solution :**Compaction** , possible only if relocation is dynamic, and is done at execution time.

- **Internal fragmentation** - allocated memory may be slightly larger than requested memory.

Noncontiguous Allocation – Paging

(*) Logical address space of a process can be noncontiguous; process is allocated physical memory wherever the latter is available.

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

(*) Divide logical memory into blocks of same size called pages. page size = frame size

(*) Keep track of all free frames.

(*) To run a program of size **n** pages, need to find **n** free frames and load program.

(*) Set up a page table to translate logical to physical addresses.

(*) Internal fragmentation.

(*) 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 Size(s)

PROGRAM (PAGES)

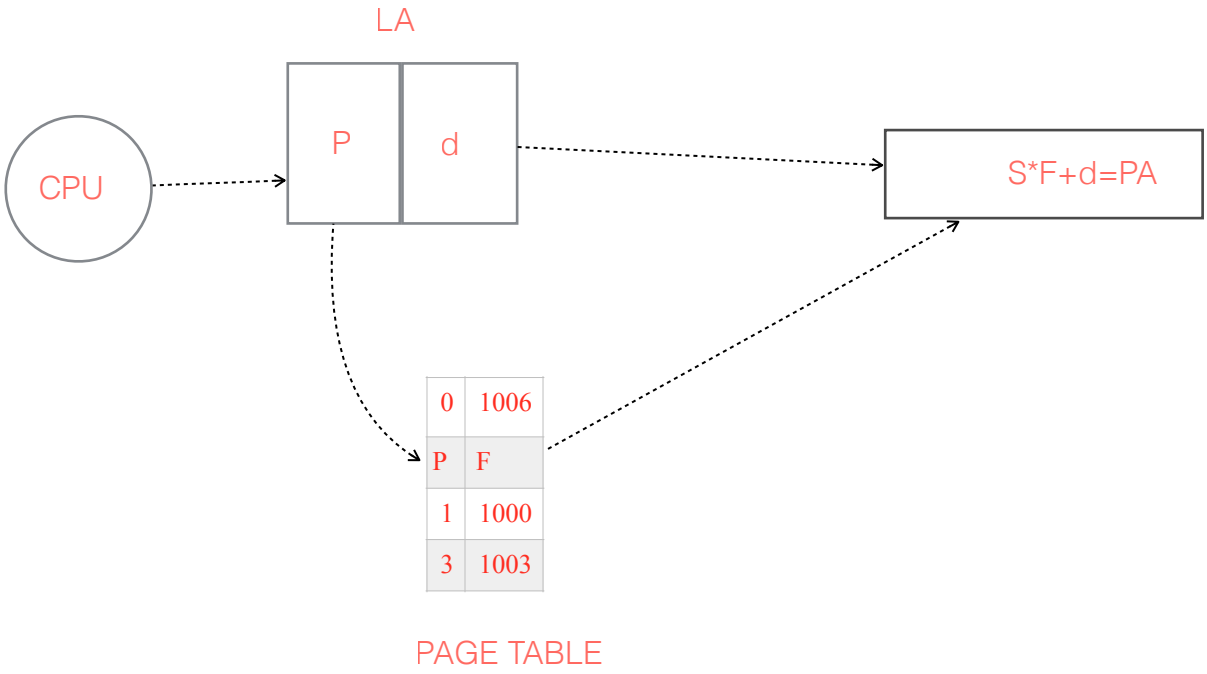
0	A
1	B
2	C
3	D

PAGE TABLE

0	1006
1	25677
2	1000
3	1003

Memory

	OS
1000	C
1001	
1002	
1003	D
1004	
1005	
1006	A
1007	
…	
25675	
25676	
25677	B
25678	
25679	
25680	



(*) Calculation of physical address:

LA = logical address , S = page size , then ,

p = LA div S

d = LA mod S

(*) In practice, the OS takes advantage of the page size being 2^n ,
the low-order **n** bits equal **d** and the remaining bits equal **p**.

Assume page size s = 256 = 2^8
given LA=800
p = 800 / 256 = 3 = 0011
d = 800 % 256 = 32 = 00100000
Assume LA length in bits 12

0	0	1	1	0	0	1	0	0	0	0	0

p (4 bits)	d (8 bits)

(*) Examples :

DEC-10 : LA 20 bits ; page size is 512 ; page no. bits 11 ; page offset 9 bits.

IBM-370: LA 24 bits ; page size is 2048 ; page no. bits 13 ; page offset 11 bits.

(*) Disadvantage :Separation between user's view of memory and actual physical memory reconciled by address-translation hardware; logical addresses are translated into physical addresses.

(*) Advantages: Sharing pages.

Active page table has the page table for the process is currently running (has the CPU)
In context switch:
the active page table is stored back to the page in the PCB for the interrupted process, and the page table for the new process is reloaded into the active page table
that is the active page table is flushed

Implementation of page table(Active page table)

(*) Each job has its own page table which is usually kept in the PCB.

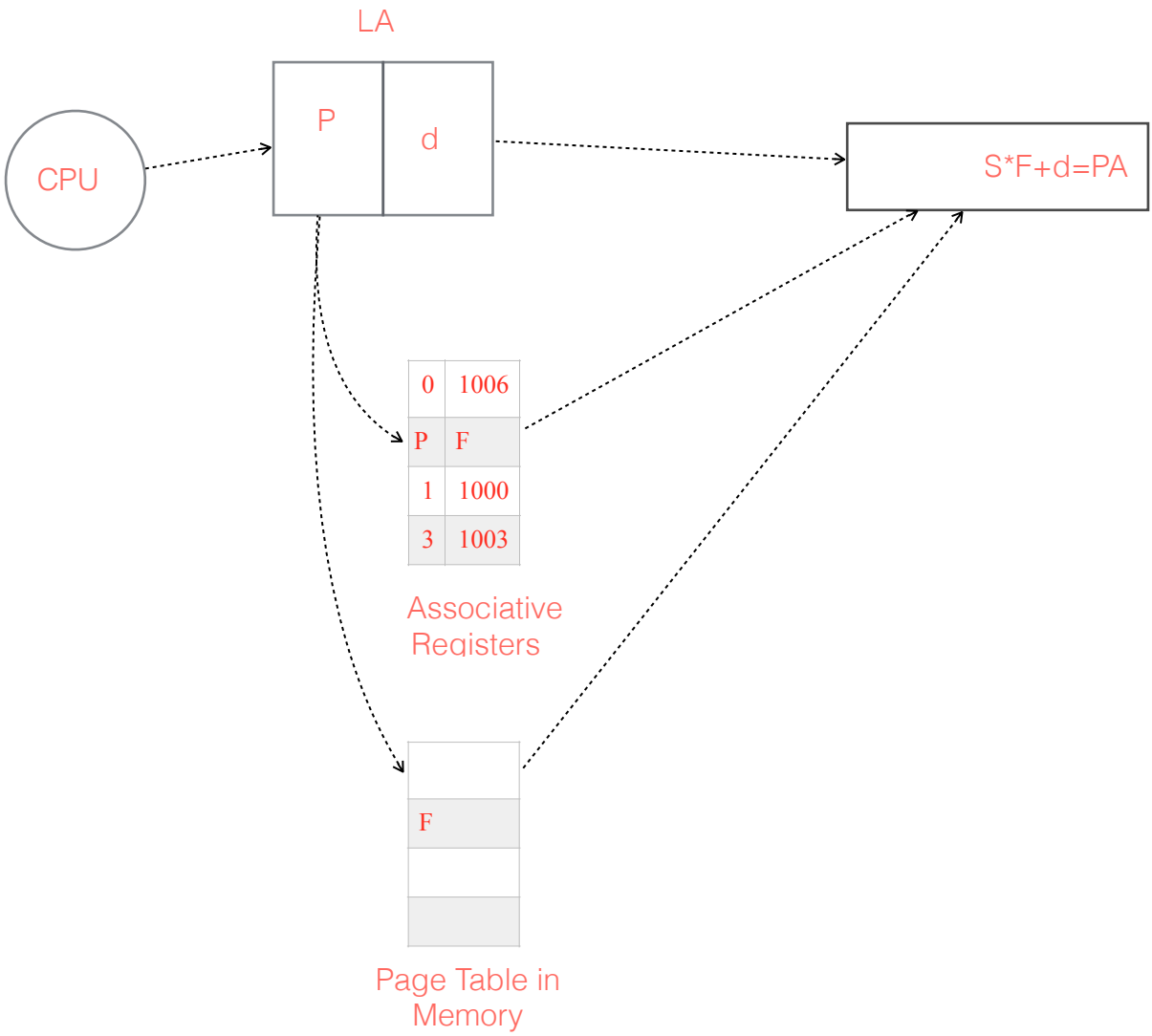
(*) Hardware implementation (Where the active table stored/located):

1. ***A set of dedicated registers***, that is loaded and stored by the CPU dispatcher during program execution like program counter(pc).
   Example: PDP-11 : LA = 16 bits ; page size = 8192 = 2^{13} ; leaving 3 bits or 8 Entries for the page table.
   Suitable if the page table is small (up-to 256 entries) (Only small size tables - There is no register big enough)

2- Page table is kept in main memory.

- Page-table base register (**PTBR**) points to the page table.
- Page-table length register (**PTLR**) indicates size of the page table.
- In this scheme every data/instruction access requires two memory accesses. One for the **page table** and one for the **data/instruction**.

3- The **two** memory access problem is solved using special fast hardware cache, called ***associative registers*** or ***Translation Look-aside Buffers*** (**TLBs**).
(Mixing between the previous two)



- A set of high speed registers are used***(associative registers***).
- Each entry contains (page # , frame #).
- Associative registers contain only **few** page-table entries.
- When CPU generates an address, **p** is first checked in the associative registers. If found, then its frame **f** is used immediately. Otherwise, the page table in memory is used, and the (page # , frame #) is added to the associative registers table.
- Every **context-switch**, associative registers are ***flushed*** for the new process.
- ***Hit ratio*** - percentage of times that a page number is found in the associative registers; ratio related to number of associative registers.
- ***Effective Access Time (EAT) :***
  associative registers lookup = **t** time unit
  assume memory cycle time - **100** nanoseconds
  hit ratio = **h**

EAT= (100 + **t**) **h** +(200 +**t**)(1 -**h**)

t = search (look up) time in the associative registers
m = memory access
h = hit ratio(probability the desired page in the associative register) 0<= h <= 1

EAT= h*(t+m)+(1-h)*(t+2m)

Example:
t = 20 nanoseconds , **h** = 90 %

EAT = (100+20)*0.9 + (200+20)*0.1 = 120*0.9 + 220 * 0.1
= 108 + 22 = 130 nanoseconds.

Memory Protection

(*) Memory protection implemented by associating protection **bits** with each frame.

(1) **Legal - Illegal** bit attached to each entry in the page table:
- "**legal**" indicates that the associated page is in the process' logical address space, and is thus a legal page.
- "**illegal**" indicates that the page is not in the process' logical address space.

Frame #	Legal/Illegal bit
F	1
	0
	0

(2) **Read-write** bit attached to each entry in the page table:
It indicates whether the page is a **read only** or **read/write** page to protect the page from modifications and rewritten if it is a read only page.

Frame #	Legal/Illegal bit	R/W bit
F	1	
EMPTY	0	
EMPTY	0	

Multilevel Paging

(*) Modern Computers support a very large logical address space (2^{32} – 2^{64}) and the page table becomes very large.

(*) Partitioning the page table allows the operating system to leave partitions unused until a process needs them.

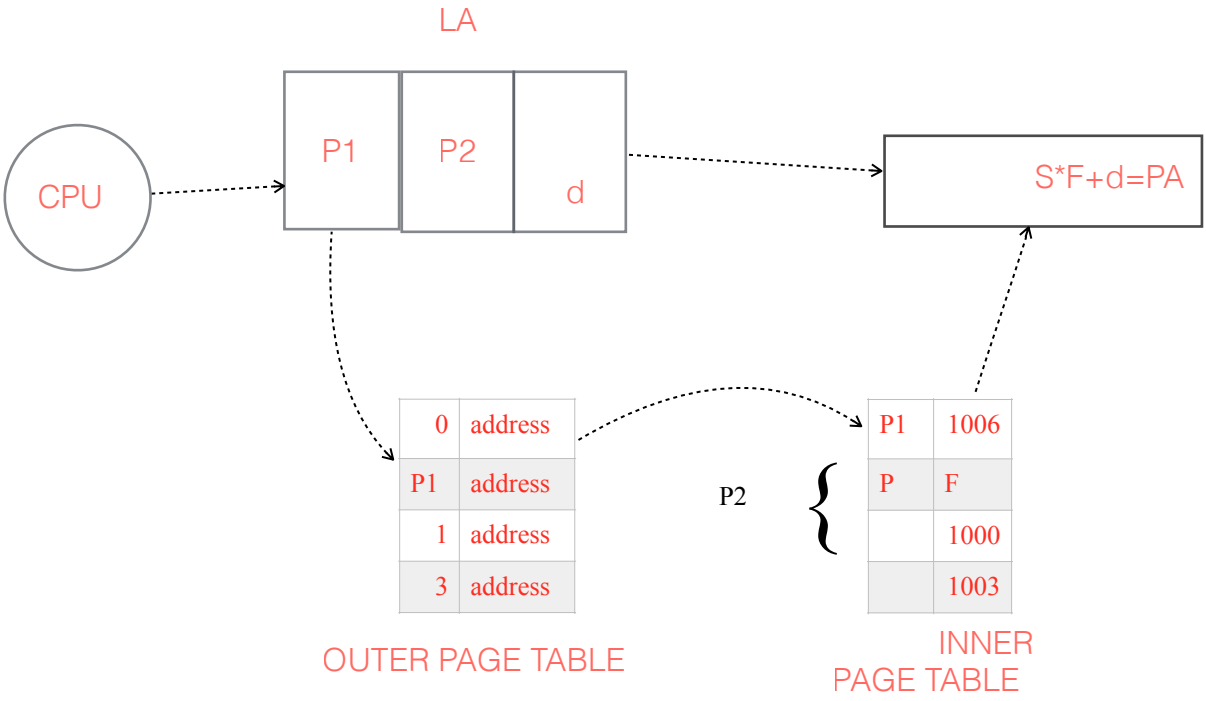
A two-level page-table scheme

Example 1:

32 bits *logical address* ; 4k=2^{12} *page size* ; 20 bits for *page no*.
If every page table entry is 4 bytes , then page table **size** = 4 * 2^{20} = 4 MB and this is **too large** to store contiguously in memory.

Solution: use two level of page table.

* Since the page table is paged, the page number is further divided into:
  - a 10-bit page number.
  - a 10-bit page offset.
* Thus, a logical address is as follows:



(*) Multilevel paging and performance

Since each level is stored as a separate table in memory, converting a logical address to a physical one may take four memory accesses.

- One level requires two memory access.
- Two levels require three memory access.
- Three level requires four memory access.

Even though five memory access time are needed , caching permits Performance to remain reasonable.

Cache hit rate of 98 percent yields:

effective access time = 0.98*120 + 0.02*520 = 128 nanoseconds

which is only a 28 percent slowdown in memory access time.

Example: Assume 2 level page table,

t = 10 millsec

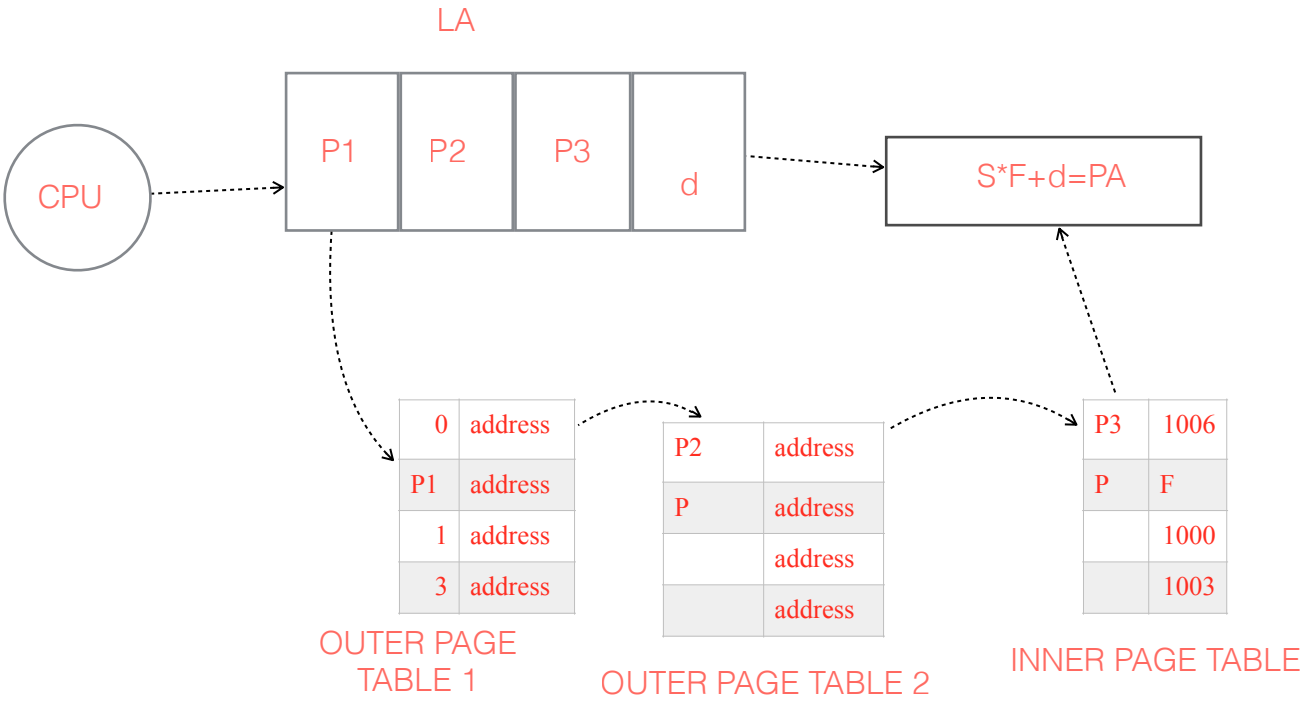
m = memory acces 100 millsec

an associative registers are used in this case and the hit ratio, h=0.95

EAT = h*(t+m)+(1-h)*(t+3m)

EAT = 0.95*(10+100)+(1-0.95)*310 = 120 millsec

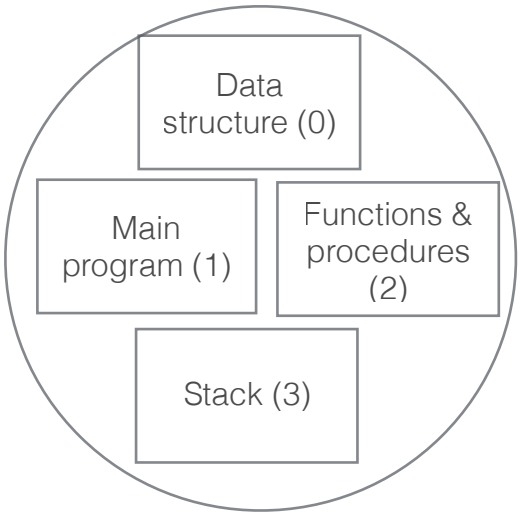
3 level page table



Inverted Page Table (Read by your self)

(*) Instead of using a page table for each process, the system uses only one page table for all frames in memory..
(*) Each page table entry consists of 3 components
[**process-id** , **page-no** , **offset**]
(*) Hashing table used using *process-is & page-no.*
(*) EX : IMB 38 ; IBM *RISC* 6000 ; IBM *RT*

Noncontiguous Allocation –Segmentation

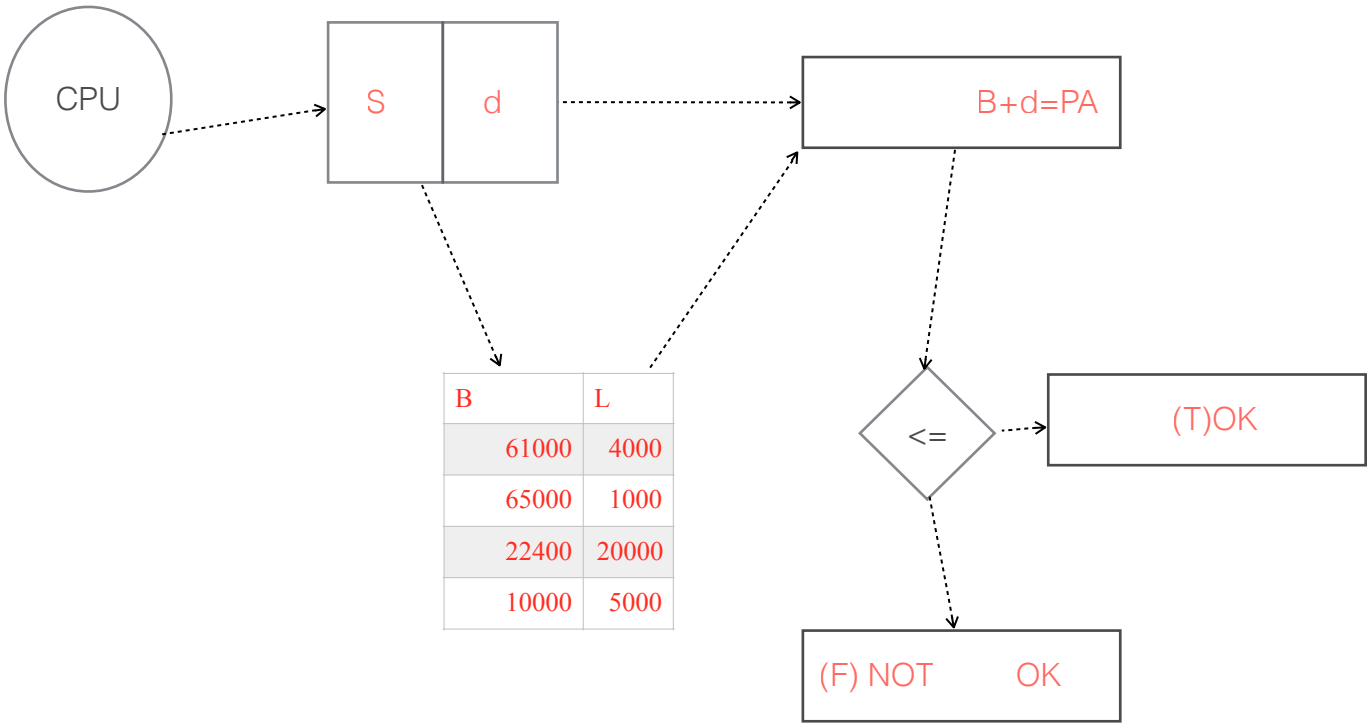


Segment Table

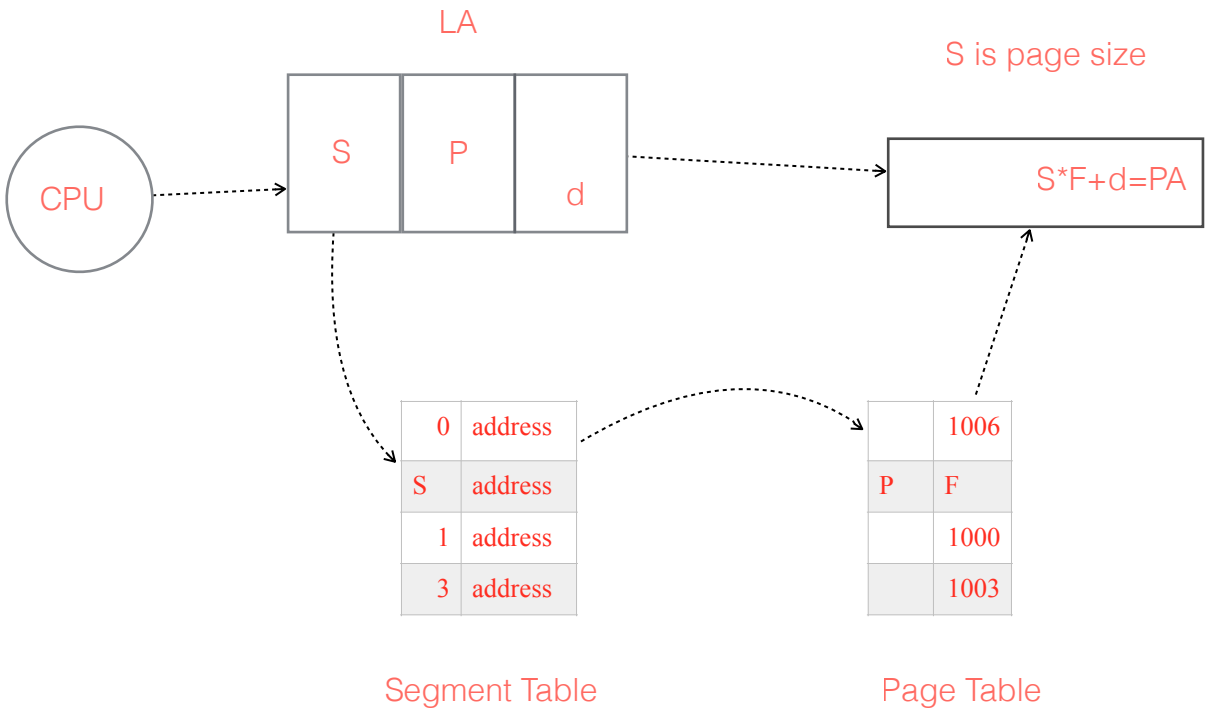
Base	Limit
61000	4000
65000	1000
22400	20000
10000	5000

Memory

	OS
10000	(3)
15000	
1002	
22400	(2)
42400	
1005	
61000	0
65000	
…	
25675	
25676	
65000	(1)
66000	
25679	
25680	



Paging separates user's view of memory from actual memory. But the **segmentation** Scheme supports user view of memory.



S = segment number

(*) A program is a collection of segments. A segment is a logical unit such as:

main program
(subroutines) procedure & function
global variables
stack
symbol table, arrays

(*) Logical address consists of a pair:
(segment-number , offset).

(*) Segment table - maps two-dimensional user-defined addresses into one-dimensional physical addresses; each entry of table has:
- ***base*** - contains the starting physical address where the segments reside in memory.
- ***limit*** - specifies the length of the segment.

(*) Segment-table base register (***STBR***) points to the segment table's location in memory.

(*)Segment-table length register (***STLR***) indicates number of segments used by a program; segment number s is legal if s < STLR.

Noncontiguous Allocation –Segmentation with Paging

Problem with segmentation:

- External fragmentation.
- Search time to allocate a segment using **first-fit** or **best-fit**.

Solution:
Paging the segments.

QUESTION:

What if the program is too big to fit in memory, what to do ?

Dynamic Loading - routine is not loaded until it is called.

- Better memory-space utilization; unused routine is never loaded.
- Useful when large amounts of code are needed to handle infrequently occurring cases.

- No special support from the operating system is required; implemented through program design.

Dynamic Linking – linking is postponed until execution time.

- Small piece of code, ***stub***, used to locate the appropriate memory-resident library routine.
- **Stub** replaces itself with the address of the routine, and executes the routine.
- Operating system needed to check if routine is in processes' memory address.

Overlays - keep in memory only those instructions and data that are needed at any given time.

- Needed when process is larger than amount of memory allocated to it.
- Implemented by user, no special support needed from operating system; Programming design of overlay structure is complex.

Chapter 9

Virtual Memory Management

Background

(*) Virtual memory : allows a very large programs to be executed using limited memory, i.e. , the logical address space can therefore be much larger than physical address space. Dynamic Loading , Dynamic Linking , and Overlays are not good an powerful enough to solve this problem.

(*) Virtual memory : separation of user logical memory from physical memory.
- Only part of the program needs to be in memory for execution.
- Need to allow pages to be swapped in and out.

(*) Virtual memory : can be implemented via:
- Demand paging
- Demand segmentation

Demand Paging

When a page is needed, it is brought into memory,
Hardware support we add a bit to the page table called valid/invalid (v/i) bit

(*) Demand paging is paging with swapping.

(*) Bring a page into memory only when it is needed.
- Less I/O needed
- Less memory needed
- Faster response
- More users

(*) Page is needed ➔ reference to it
- invalid reference ➔ abort
- not in memory ➔ bring to memory

Valid-Invalid bit

(*) With each page table entry a valid-invalid bit is associated
(1 ➔ in-memory, 0 ➔ not-in-memory)

(*) Initially valid-invalid bit is set to 0 on all entries.

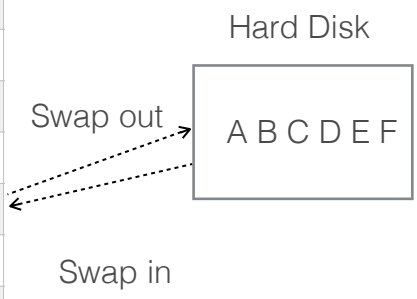
(*) During address translation, if valid-invalid bit in page table entry is
0 ➔ page fault.

0	A
1	B
2	C
3	D
4	E
5	F

Logical Program

Page	FRAME	(V/I) bit
0	2009	1
1		0
2		0
3	2001	1
4		0
5		0

	OS
2000	
2001	D
2002	
2003	
2004	
2005	
2006	
2007	
2008	
2009	A
2010	
2011	
2012	
2013	
2014	



Page Fault

1. The LA address is checked through the page table if it is a valid address , if so, then reference it , otherwise, trap to OS ➔ **page fault**. (software interrupt)
2. OS looks at another table copy (in PCB) to decide:
   a) Just an invalid reference ➔ abort the process.
   b) Just not in memory.
3. Get empty frame.
4. Swap page into frame.
5. Reset tables, validation bit = 1. (Update tables)
6. Restart instruction: (Continue execution)

What happens if there is no free frame?

(*) Page replacement - find some page (***victim***) in memory, but not really in use, swap it out.

➔ **algorithm**
➔ **performance** - want an algorithm which will result in minimum number of page faults.

(*) Same page may be brought into memory several times.

(*) We could start execution with 0 pages in memory, ***pure demand paging***.

Performance of Demand Paging

(*) Page Fault Rate (probability) $0 \le p \le 1.0$
if p = 0, no page faults (Ideal)
if p = 1, every reference is a fault (Worst)

(*) Effective Access Time (**EAT**)
EAT = (1 - p) * memory access
+ p * (page-fault-handling overhead (* updating page table *)
+ [swap page out] (* […] 0 or 1 occurrence *)
+ swap page in)

EAT= (1-p)*m + p*[swap-in page from HD to memory + maybe swap-out from memory to HD +m(is very small so we can ignore it)]

(*) **Example**:
- memory access time = 5 mics
- 50% of the time the page that is being replaced has been modified and therefore needs to be swapped out.
- Swap Page Time = 10 mils = 10,000 mics
- EAT = (1-p)*memory-access-time + p*(page-fault-handling-time)
≅ (1-p)*5 + p (swap-out + swap-in)
≅ (1-p)*5 + p (0.5*10000 + 10000)
≅ (1-p)*5 + p (15000)
≅ 5 + 14995p (in mics)
≅ 15000p mics

(*) This means the memory access time is almost negligible compared to the swap time is the major time a

EAT is totally depends on P
Our objective is to min the page fault P
we should elect the victim page to min the swap out rate

Page Replacement

(*) we want optimum page replacement time.

(*) Use modify (***dirty***) bit to reduce overhead of page transfers - only modified pages are written to disk.

Page	FRAME	(V/I) bit	Dirty bit
0	2009	1	
1		0	
2		0	
3	2001	1	
4		0	
5		0	

Page-Replacement Algorithms

(*) Want lowest page-fault rate. - Our objective is to min the number of page faults happening

(*) Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.

(*) In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

First-In-First-Out (FIFO) Algorithm

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

- 3 frames (3 pages can be in memory at a time per process)

1	1	1	4	4	4	5	5	5
	2	2	2	1	1	1	3	3
		3	3	3	2	2	2	4

9 page faults

- 4 frames (4 pages can be in memory at a time per process)

10 page faults

FIFO Replacement - Belady's Anomaly more frames does not ➔ less page faults

Optimal replacement Algorithm

(*) Replace the page that will not be used for the longest period of time.

Example: 3 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

7 page faults

Example: 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

6 page faults

(*) Problem: How do you know this? Estimate previous history.
(*) Used for measuring how well your algorithm performs.

Least Recently Used (LRU) Algorithm

Replace the page which **has not** been used for the longest period of time.

Example: 4 frames

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	1	1	1	1	1	1	5
	2	2	2	2	2	2	2
		3	3	5	5	4	4
			4	4	3	3	3

8 page faults

(*) FIFO : Uses the time when the page has brought into memory.
OPTR : Uses the time when the page will be used.
LRU : Uses the recent past history. Replace the page which has not been used For the longest period of time.

Implementation:

(*) Counter implementation

- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter. counter stores the time the page has been accessed lately.
- When a page needs to be changed, look at the counters to determine which are to change

(*) Stack implementation

When the page is referenced, the page is pushed on top of all stack.
In this case
Top stack : most recently used
Bottom of stack: least recently used

-keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
  requires 6 pointers to be changed
- No search for replacement

LRU Approximation Algorithms

keep in mind:

(*) Reference bit - is added to the page table to indicate if the page is referenced

Page	FRAME	(V/I) bit	Dirty bit	Reference bit
0	2009	1		
1		0		
2		0		
3	2001	1		
4		0		
5		0		

- With each page associate a bit, initially = 0.
- When page is referenced bit set to 1.
- Replace the one which is 0 (if one exists). We do not know the order, however.

(*) Second chance - enhanced reference bit

- Need reference bit.
- Clock replacement.
- If page to be replaced (in clock order) has reference bit = 1, then:

  a) set reference bit 0.
  b) leave page in memory.
  c) replace next page (in clock order), subject to same rules.

(*)Enhanced second chance
use also the dirty bit in the selection.
(reference bit, dirty bit)
(0,0) (not reference, not dirty) (1) BEST TO CHOOSE (no swap out)
(0,1) (not reference, dirty) (3)
(1,0) (reference, not dirty) (2)
(1,1) (reference, dirty) (4) WORST TO CHOOSE

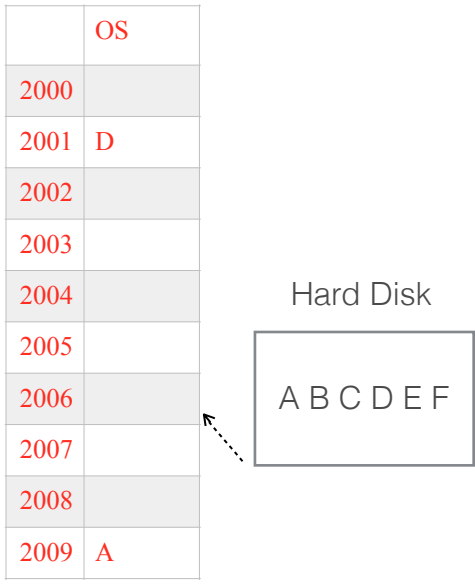
Counting Algorithms

(*) keep a counter of the number of references that have been made to each page.
- **LFU** Algorithm: replaces page with smallest count.
- **MFU** Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

(*) Page-Buffering Algorithm

desired page is read into a free frame from the pool before the victim is written out.
Always there is a free frame in memory

1. Swap in the desired page in the frame
2. Resume execution
3. Find a victim which will the next free frame



(*) Other algorithms : Random Replacement Algorithm

Allocation of Frames

Theoretically, the job can start execution with 0 pages in memory, this called “pure Demand Paging”
(*) Each process needs minimum number of pages.

Example: IBM 370 - 6 pages to handle

(*) Two major allocation schemes:
- **fixed** allocation
- **priority** allocation

Fixed Allocation

- Equal allocation

If 100 frames and 5 processes, give each 20 pages.

- Proportional allocation

Allocate according to the size of process.

s_i = size of process p_i

$S = \sum s_i$

m= total number of frames

a_i = allocation for p_i = $(s_i/S)*m$

Example : m =64

s_1 =10

s_2 = 127

a_1 = (10/137)*64 ≅ 5

a_2 = (127/137)*64 ≅ 59

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

example: assume 100 frames of memory

Process	Priority
P1	5
P2	2
P3	1

p1 gets (5/8)*100

p2 gets (2/8)*100

p3 gets (1/8)*100

- If process **P_i** generates a page fault,

select for replacement one of its frames.

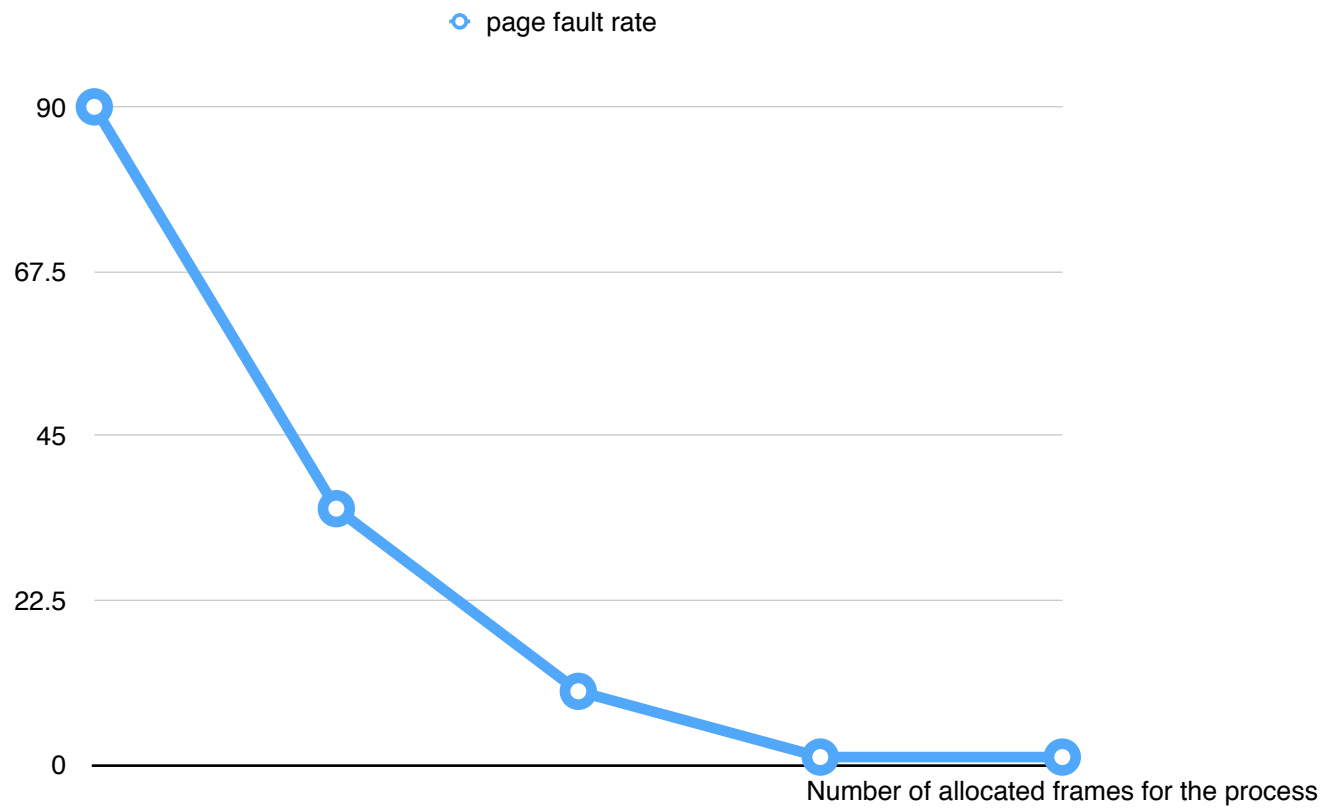
select for replacement a frame from a process with lower priority number.

Global versus local allocation - Two types of replacement algorithms

(*) Global replacement - process selects a replacement frame from the set of all frames; one process can take a frame from another.

(*) Local replacement - each process selects from only its own set of allocated frames. (interrupted process)

Thrashing



OS is busy in swapping due to the low number of frames allocated for the process

(*) If a process does not have ‘‘enough’’ pages, the page-fault rate is very high:

- ➔ low CPU utilization.
- ➔ operating system thinks that it needs to increase the degree of multiprogramming.
- ➔ another process added to the system.

(*) ***Thrashing*** ≡ a process is busy swapping pages in and out.

Demand Segmentation

used when insufficient hardware to implement demand paging.

(*) OS/2 allocates memory in segments, which it keeps track of through ***segment descriptors***.

(*) Segment descriptor contains a valid bit to indicate whether the segment is currently in memory.

- If segment is in main memory, access continues,
- If not in memory, segment fault.

Chapter 10 +11+12

File Concept

(*) File is: Contiguous logical address space

(*) Types:
- Data
  - numeric
  - character
  - binary
- Program
  - source
  - object (load image)
- Documents

File Structure

(*) None - sequence bytes, words
(*) Simple record structure
- Lines
- Fixed length
- Variable length

(*) Complex Structures
- Formatted document
- Relocatable load file

Can simulate **last two** with **first** method by inserting appropriate control characters.

File Attributes

- **Name** - only information kept in human-readable form.
- **Type** - needed for systems that support different types.
- **Location** - pointer to file location on device.
- **Size** - current file size.
- **Protection** - controls who can do reading, writing, executing.
- **Time, date, and user identification** - data for protection, security, and usage monitoring.

(*) Information about files are kept in the directory structure, which is maintained on the disk, which is called generally, ***device directory***.

Hard Disk

Sector 0 is boot sector
Sector 1 and above are File allocator Table FAT (Device Directory)

In the case of tapes each **label** is the device directory.

File Operations

- create
- write
- read
- reposition within file - file seek
- delete
- open(F_i) - search the directory structure on disk for entry F_i, and move the content of entry to memory.
- close(F_i) - move the content of entry F_i in memory to directory structure on disk.

Access Methods

- Sequential Access
  - *read next*
  - *write next*
  - *reset*
  - **no** *read* after last *write*
  - *generally* ***no*** *(rewrit*e)

- Direct Access
    read n
    write n
    position to n
    read next
    write next
    rewrite n

n = relative block number

Directory Structures

(*) The general information kept about directory system are :

- Name
- Type
- Address(location) the address of the first block in the device directory
- Current length
- Maximum length
- Current position (File Pointer – FP) the address of the current address of the file
- Date last accessed (for archival)
- Date last updated (for dump)
- Owner ID (who pays)
- Protection information (discuss later)

(*) Operations performed on directory:

- Search for a file
- Create a file
- Delete a file
- List a directory
- Rename a file
- Traverse the file system g Date last accessed (for archival)

(*) There are two directory systems:

- **Device Directory :** The directory where the **physical information** generally is kept about the files in that device , such as, name, location, position, … etc.
- **User File directory (UFD)** : The directory where the **logical information** generally is kept about the user files, such as, name, size, date, … etc.

Note: Searching the directory is performed more often (frequently) the data structure user for the directory must provide quick search in the directory

(*) Both the directory structure and the files reside on .

(*) Backups of these two structures are kept on tapes.

(*) Device Directory Implementation:

(1) Linear list :

- simple to program
- time-consuming to execute (search time) (Sequential search)
- Not practical

Name	
Sam	
xx	
AB	
comp	

(2) Sorted Linear list:

- Binary Search
- time-consuming to execute
- what if a file added or deleted ?
- Not a solution

Name	
AB	
comp	
Sam	
xx	

(3) Hash Table - linear list with hash data structure.

- decreases directory search time
- *collisions* – situations where two file names hash to the same location

(4) Tree Structure. complex to implement

(*) User File Directory:

Organize the directory (logically) to obtain: The Criteria we need in UFD structure:

(*) Efficiency - locating a file quickly. quick search
(*) Naming - convenient to users.
- Two users can have same name for different files.
- The same file can have several different names (*Aliases*).

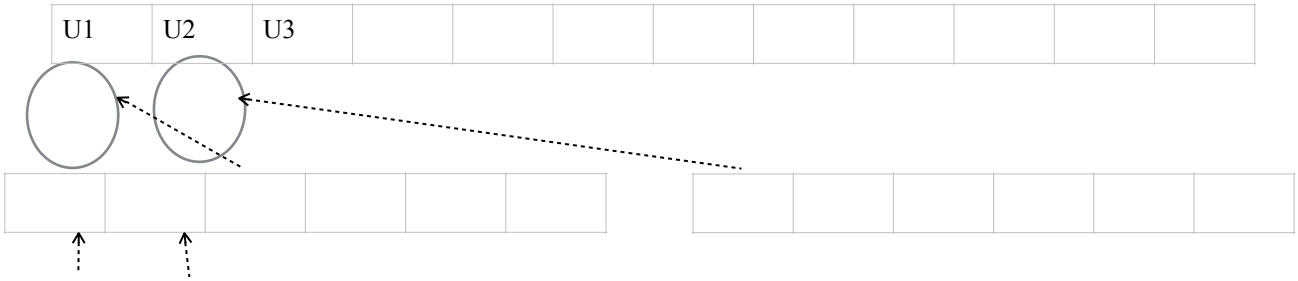
(*) Grouping - logical grouping of files by properties, e.g., all Pascal programs, all games,…

The Structures of the UFD

(1) Single-Level Directory - a single directory for all users.

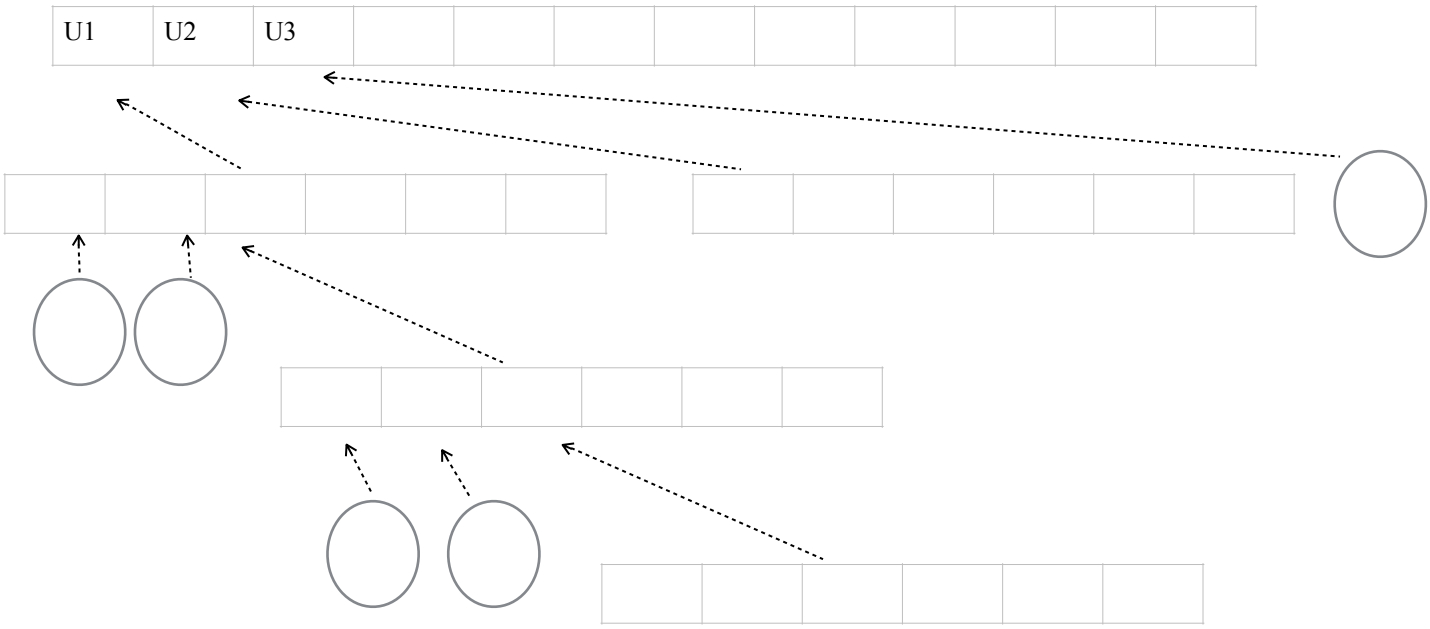
(*) Naming problem
(*) Grouping problem

(2) Two-Level Directory - separate directory for each user.



(*) Path name
(*) Can have the same file name for different user - NO ALIASING
(*) Efficient searching
(*) No grouping capability

(3) Tree-Structured Directories



(*) Efficient searching
(*) Grouping capability
(*) Current directory (working directory)
(*) No aliasing

(4) Acyclic-Graph Directories - have shared subdirectories and files.

(*) Two different names (***aliasing***)
(*) If A deletes D ==> dangling pointer.

Solutions:
- Back pointers, so we can delete all pointers. Variable size records a problem.
- Back pointers using a daisy chain organization.
- Entry-hold-count solution.

Protection

(*) File owner/creator should be able to control:
- what can be done
- by whom

(*) Types of access

- Read
- Write
- Execute
- Delete
- List

Access Lists and Groups

(*) Mode of access***: read, write, execute***
(*) Three classes of users

		RWX
a) owner access	7	1 1 1
b) group access	6	1 1 0
b) group access	5	1 0 1 - Teacher thinks book is mistaken, that it is 5 not 6
b) public access	1	0 0 1

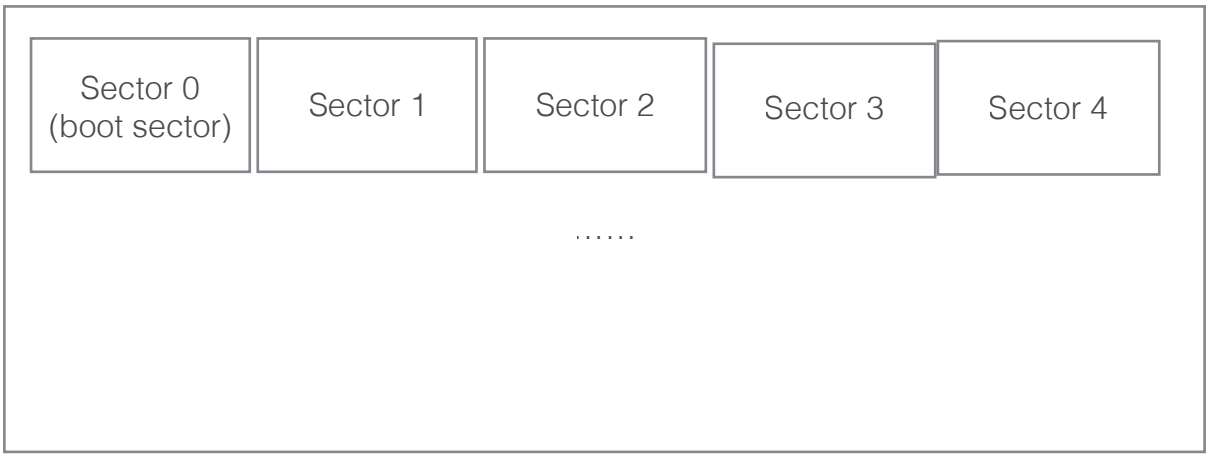
(*) Ask manager to create a group (unique name), say G, and add some users to the group.

(*) For a particular file (say *game*) or subdirectory, define an appropriate access.

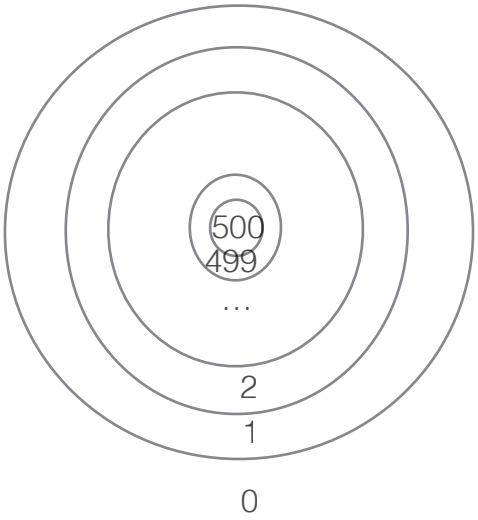
Disk Structure

(*) A disk can be viewed as an array of sectors (blocks).

A sector (block) : smallest addressable unit in the disk. **(track , surface , sector)**
Sector's size 512



Hard Disk



(*)Each sector has 3-dimensional address (i,j,k)
i = #track(cylinder)
j = #surface
k = #sector
(*)HD is considered as an arroyo of sectors, the OS transfer the 30dim address to one-dim address
(*)The addressing used us called cylindrical addressing
(*)The OS maps the 3-dim add. to one-dim
(*)The formula for transforming the 3-dim address (i,j,k) to one-dim

(*) Given the address **(i , j , k)** , then , transformation from 3-dim to one-dim

b = k + s * (j + i * t)

Where,
t = number of surfaces (tracks per cylinder)
s = number of sectors per surface

example: HD has 1000 track(cylinder), 6 surfaces and the number of sectors per track is 50 sectors
1. given the address (100, 5, 37), convert it to one dim
2. if sector size is 512 byte, compute the size of HD
solution
1. HD size = 6*1000*50*512 =~ 150 MB
b=37+50*(100*6+5)

(*) ***Seek time*** : time to move the R/W head to a particular track.

(*) ***Latency time*** : time to rotate the sector under the R/W head.

(*) ***Access time*** = Seek time + latency time + transfer time

File-System Structure

(*) File structure
- Logical storage unit
- Collection of related information

(*) File system resides on secondary storage (disks).

(*) *File control block* - storage structure consisting of information about a file.

(*) The logical file must be mapped into the physical storage media (disk)

Blocking

Assume sector size 512 bytes we have found a DB fit with record size 200 bytes
left bytes 512-200=112 so wasting percentage =112/512 = 22%

Blocking: Packing and unpacking a number of logical records into a physical block.

Blocking factor: The number of logical record packed into a physical block.
Example: the blocking factor is 5
//I did not understand it!

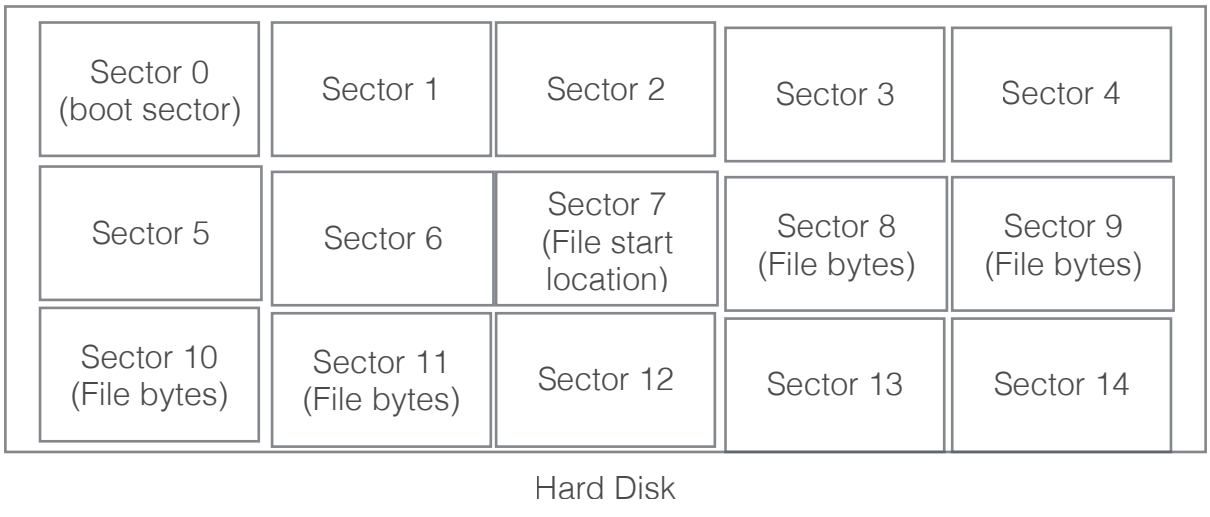
Allocation Methods:

How files are stored in the disk. How disk blocks (sectors) are allocated for a file

Contiguous Allocation

(*) each file occupies a set of contiguous blocks on the disk.

(*) The file is defined by the address of the first block and its length. Location & Length



Hard Disk

Name	Location	Length	File Pointer (FP)
Sam	7	6	10

Advantages:

(*) No seek is required to access block **(b+1)** after block **b** unless **b** is last block in the cylinder.

(*) Random and sequential access are supported easily .

Note: we say the sequential is direct support easy or not depending on how easy or hard to provide the address of the block to be accessed.

Sequential addressing:

If FP at block with address n to access next block, we increment n, the address of next block is n+1 (simple addition operation)

Direct access: means to access (r/w) block n in the file n is relative address (n is the offset) that is, means the block with address n in your file.

Example: read block 17

in our example read block 4

address of block #4 = 7 + 4 - 1 = 10

In general:

address of block #n = location + n - 1

Disadvantages:

(*) Problem : external fragmentation (holes) in the disk.

Solution : Compaction.

(*) ***Major problem*** : Files cannot grow.

(*) How to find hole for the file: **First fit , Best fit , Worst fit**

First fit and best fit have better performance.

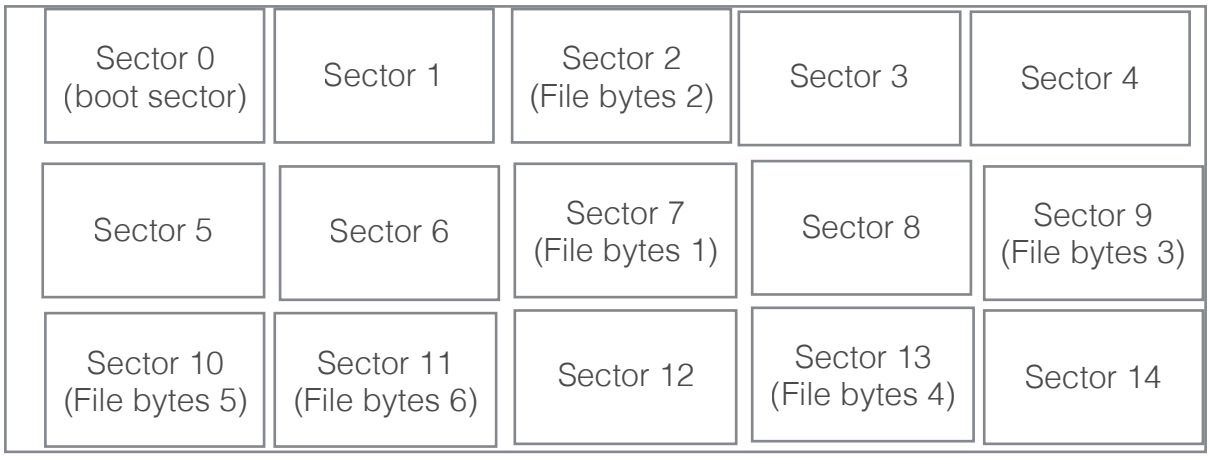
Linked Allocation

(*) each file is a linked list of disk blocks

(*) blocks may be scattered anywhere on the disk.

(*) Allocate as needed, link together.

Example:

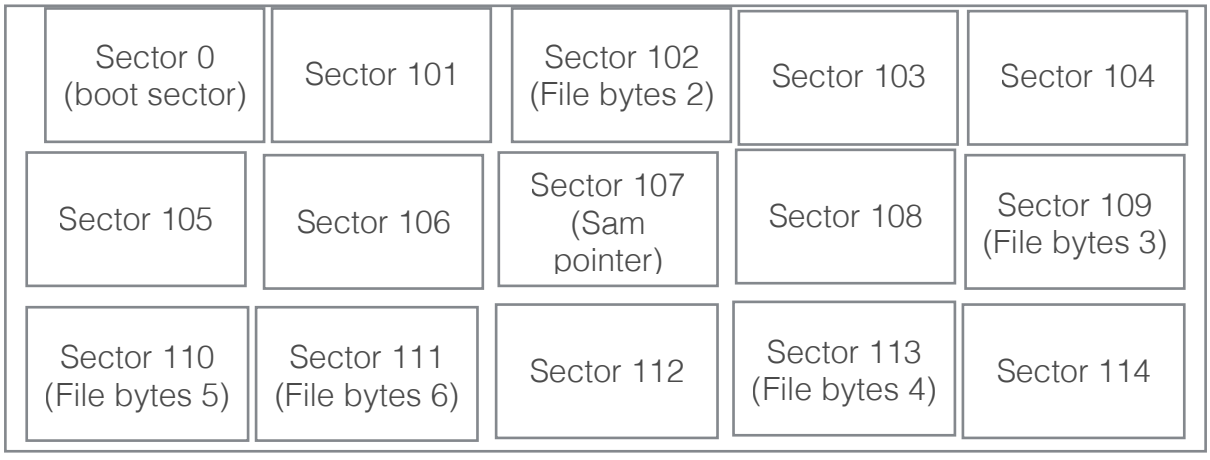


(*) **Advantages**:
- Simple - need only starting address and size.
- Free-space management system - no waste of space. No External fragmentation. (no fragmentation)
- File can grow

(*) **problem**: address pointer waste each block as address of next block
major problem: Supports sequential access only.

Indexed Allocation

- brings all pointers together into the ***index block***.
each file has index block that contains address of data file blocks
(*) Need index table
(*) Random access in addition to sequential access.
(*) Dynamic access without external fragmentation, but have overhead of index block.
(*) Mapping from logical to physical in a file of maximum size of 256K words and block size of 512 words. We need only 1 block for index table.



Sector 107 (Sam pointer)

Has this table of blocks

119
102
103
121
114
113
116
-1
….

T

Name	Index block	Length
Sam	112	7

Device Directory

Sequential access: If FP at index #n, the address of next block is T[n+1]
Direct access: If FP at index [0], address of block #n, T[n-1]
If file is large the last entry of T is the address of another block has the rest of T table
Disadvantages: For every File we waste at least one index block. it is based on ratio wasting, if big file no affect, but if small file wasting block is too much wasting
Most OS now use hybrid system.

Free-Space Management

(*) Bit map - vector (n blocks)
1=full, 0=empty

(*) Linked list (free list)

(*) counting : keep the address of the **first free block** and the number **n** of adjacent free blocks. This is best used with contiguous allocation.

(*) grouping : Store the addresses of **n** free blocks in the first free block. (Same Idea indexed allocation)

Disk Scheduling

Access time = seek time + latency time+ transfer time

(*) Disk Requests - Track/Sector
- Seek (This we can control it)
- Latency (Hardware related we can do nothing with it)
- Transfer (Hardware related we can do nothing with it)

(*) Minimize Seek Time (Our objective)

Example:
Assume HD with 200 tracks (0-199)
The queue of waiting jobs request service in the track
98, 183, 37, 122, 14, 124, 65, 67
The read/write head is currently serving job at track 53, and just finished serving track 40

(*) Seek Time » Seek Distance
(*) A number of different algorithms exist.

We illustrate them with a request queue (0-199).

98, 183, 37, 122, 14, 124, 65, 67

Head is currently serving 53 and just finished 40.

Algorithms:

FCFS
Average head movement = (183-53 + 183-37 + 122-37 + 122-14 + 124-14 + 124-65 + 67-65)/8 = 50
SSF Shortest Seek First
AHM = (67-53 + 67-14 + 128-14)/8 = 29 track/job optimum
Major Problem: Starvation
SCAN (Elevator)
AHM = (199-53 + 199-14) / 8 = 41 track/job
LOOK
AHM = (183-53 + 183-14) /8 = 37 track/job
C-SCAN Circular Scan
AHM = (199-53 + 199 -0 + 37-0) /8 = 47 track/job
C-LOOK
AHM = (183-53 + 183-14 + 37-14) /8 = 40.25 track/job

NOTE : For diagrams and more details see the text book.