Date: ___/___/___

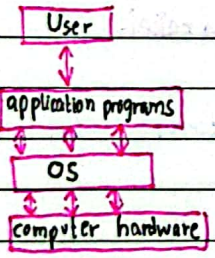
Chapter 1: Introduction

What is an Operating System?

- A software that manages a computer's hardware
- A program that acts as an intermediary between a user of the computer and the computer hardware.
- provides an environment for other programs..

Four Components of a computer system:

1- Hardware - provides basic computing resources for the system e.g: CPU, I/O, memory.
2- Operating System
3- Application programs - e.g: word processors, web browsers etc
4- Users



Viewpoints

User

- OS designed mostly for ease of use
- less attention to performance & none to resource utilization

e.g: mobile phones, PCs, embedded systems have little or no user view

System

1- OS is a resource allocator:
- manages all resources
- Decides between conflicting requests for efficient & fair use.

2- OS is a control program:
- controls execution of program to prevent errors and improper use of computers. especially concerned with I/O.

Purpose of an operating system:

- They offer a reasonable way to solve the problem of creating a useable computing system.
- OS manages resources needed for a computer e.g: hardware & application programs.

* ~~An OS is a~~ A program running all the time is called Kernel, it is a system software part of the OS.

OS	Kernel
· provides interface b/w user & hardware	· provides interface b/w applications & hardware

Date: ___/___/___

* Middle-ware: set of software frameworks that provide additional services to application developers.

Computer System Organization:

- Device Driver: software program → execute and operate systems that interact with devices.
- Device Controller: hardware program → to connect a computers OS and functions by connecting device & device driver.

How it works?

- Device drives loads registers into device controller
- All device controllers are connected through a common bus.
- Each device controller has a local buffer, it moves data b/w devices & local buffer.
- Device controller informs a device driver that an operation has been completed via an interrupt.
- Device driver then gives control to different parts of the OS or returns status info, for each device controller there is a device driver.
- Memory controller synchronizes access to the memory.
- CPU & device controller compete for memory cycles & can execute in parallel.

Interrupts:

- An interrupt is a mechanism by which other modules may interrupt the "normal sequencing of the processor".
- temporarily stops or terminates a current process.
- signal emitted by hardware to CPU.

When CPU is interrupted:

- stops the operation & transfers execution to a fixed location where the starting address of 'interrupt service routine' is located, when 'interrupt service routine executes, the CPU resumes the computation.

Working of Interrupts:

- Interrupt vectors contains pointers to the interrupt service routines holding their starting addresses.
- Interrupt specific handler locates the address of the interrupt service routine in the vector.
- An OS is interrupt driven
- Interrupt is transferred through a wire called interrupt request line to a CPU which invokes the interrupt handler, that does the processing and saves the & restores the state of the CPU.

for unrecoverable memory events ← unmaskable

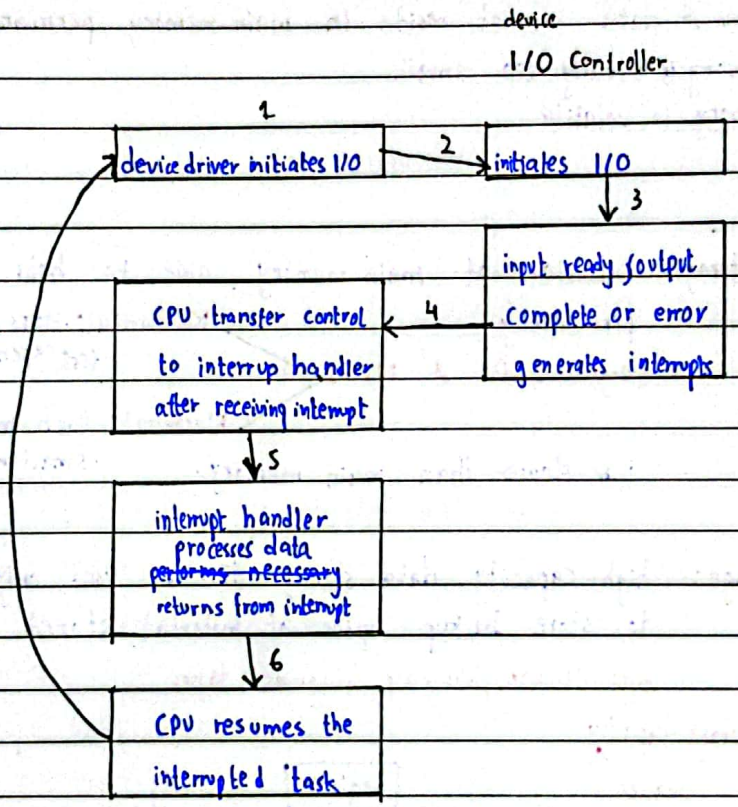
nonmaskable → that can be turned off for high priority processes.

Date: ___/___/___

Trap	Interrupt
· software generated interrupt triggered by a user program to invoke OS functionality.	· triggered by a hardware device

Interrupt handling features:

1. ability to defer interrupt handling during critical processing
2. efficient way to dispatch to interrupt handler
3. multilevel interrupts



each elements of interrupt vector pointing to a list of interrupt handles.

* Interrupt chaining is a compromise between overhead of a large interrupt table & the inefficiency of dispatching a single ~~process~~ interrupt handler.

* The interrupt mechanism ~~also~~ implements a system of interrupt priority levels. enable cpu to defer the handling of low and high priority interrupts without masking all interrupts.

Date: ___/___/___

Storage Structure

- CPU loads instructions from memory.
- Most programs are run from Main Memory / RAM, which is volatile (contents lost when powered off).
- Boot-Strap program: - first program to run when computer is powered on.
  - loads the OS. kernel.
  - ↳ loaded into EEPROM, as it is non-volatile
- Electrically Erasable programmable ROM (EEPROM): low speed, for loading static programs not used frequently.

Why programs & data cannot reside in main-memory permanently?

1- Main-memory is usually too small.
2- Main-memory is volatile

Secondary Storage: - extension of main memory, able to hold large quantities of data permanently.

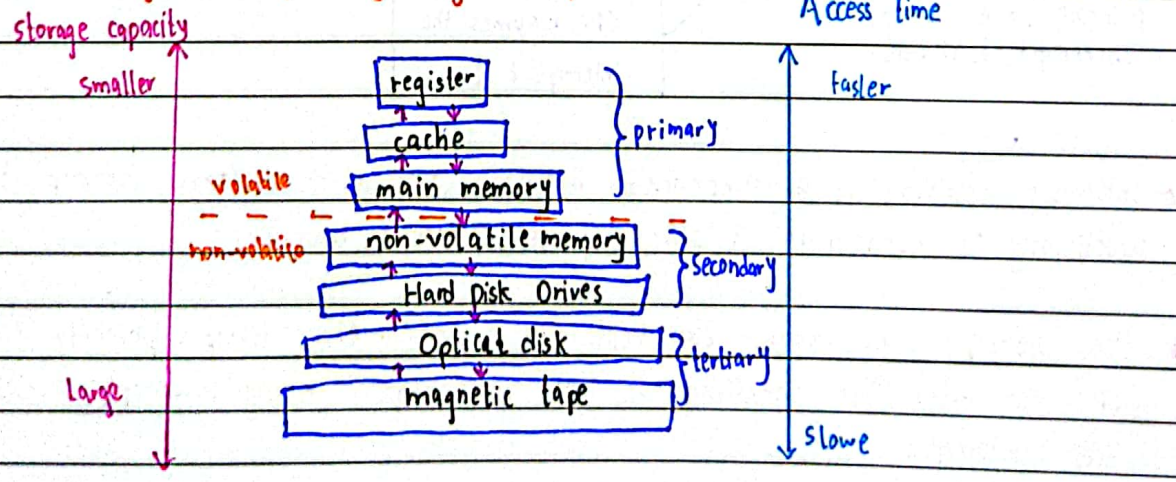
e.g: HDDs & NVMs

- Mechanical: - HDDs, optical disks, magnetic tapes
  - less expensive and per byte & larger
- Electrical: - flash memory, FRAM, NRAM, SSD.
  - smaller, costly & faster.

- ∴ slower than main-memory.

Tertiary Storage: high-capacity data storage devices used only for special purposes i.e: to store backup copies of material stored on other devices.

e.g: CD-ROM, blu-ray, magnetic tapes



Date: ___/___/___

I/O Structure:

Why DMA is used?

- interrupt-driven I/O produces high overhead when used for bulk data movements such as NVS I/O.

- In DMA, there is direct data transfer from/to the device and main memory without CPU intervention.
- Only one interrupt per block to inform device driver that the operation is completed.
- As a result, CPU is available for other processes.
- In DMA, switches are used instead of a bus, as a result multiple components can talk to each other concurrently without competing for memory cycles on a shared bus.

Computer System Architecture:

1- Single Processor Systems:

- One CPU with a single core → responsible for executing instructions and registers.
- limited instruction set and do not run processes
- can have special purpose processors that run a

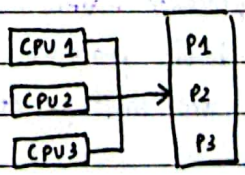
2- Multi-processor Systems: ~~parallel~~ (parallel / tightly coupled systems)

- multiple processors, each with a single-core CPU.

Two types:

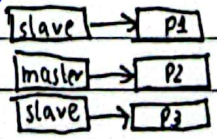
1- Symmetric multiprocessing:

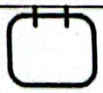
- shared memory, CPUs can work alternatively as per the demands of the processes.
- CPUs won't remain idle
- ~~each system can have a different OS.~~
- single OS is used



2. Asymmetric multiprocessing:

- non-shared, CPU will remain idle if a process is in waiting
- ~~single OS is used~~ each CPU might have a different OS





Disadvantage:

★ When multiple processors ~~perform~~ cooperate on a task, a certain amount of overhead is incurred in keeping all the parts working correctly, as a result the expected gain is lowered.

Date: ___/___/___

Advantages:

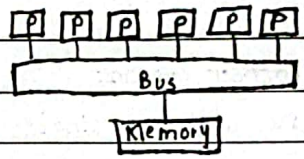
1- Increased throughput:
- more tasks completed in a certain time

2- Economies of scale:
- easier to scale up & down, e.g: increasing RAM/hardware in a computer.

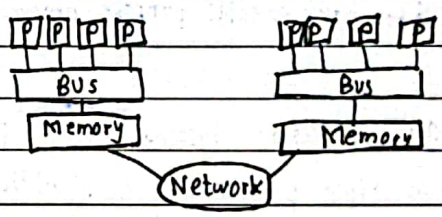
3- Increased reliability:
- fault tolerance, if CPU breaks down, the work load is shifted onto the other ones.

Dual-Core:
- UMA and NUMA architectures

- Uniform Memory Access (UMA):
  - In multiprocessor systems (symmetric)
  - Each processor has uniform access to memory.



- Non-Uniform Memory Access (NUMA):
  - easier to scale than UMA
  - local access is faster than non-local
  - each CPU has a local memory
  - different time of memory access for different levels of data.

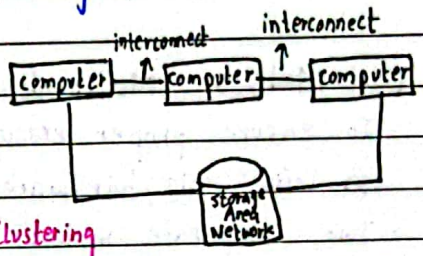


Blade-Servers:
- multi-processor boards, I/O boards & networking boards placed in the same chassis.
- each blade processor boots independently & runs its own OS
- some are multi-processors as well.

Date: ___/___/___

Clustered Systems:

- loosely coupled
- Composed of two or more individual systems, typically a multi-core system.
- share storage and connected via LAN.
- Provides High-availability service: i.e: if a monitored machine fails, the monitoring system takes its storage and restarts its programs, resulting in a brief interruption only. This provides increased reliability



Asymmetric Clustering
- one machine in hot standby mode (monitor)

Symmetric Clustering
- multiple nodes (systems) running applications & monitoring each other
- no system must remain idle.

- provides High performance computing: ~~Organizati~~ i.e: dividing programs using parallelization on individual cores in a cluster and combining the results from all the nodes for a final solution.
- Distributed Lock Manager (DLM): organises and serialise (synchronizes) access to resources.

Operating System Structure:

- System Daemons: · runs the entire time the kernel is running
  · provides services outside the kernel. loaded into memory at boot time.
- System Call: computer invoking/requesting an OS service using a user program.

Multi-Programming:

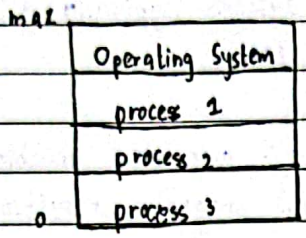
- increases CPU utilization:
- CPU always has a program to execute (never idle)
- If a process goes for its I/O instruction, the CPU looks for another process in the memory, when the first process return, it is executed first.

Multi-tasking: (Time Sharing)

- logical extension of multiprogramming
- Fast switching between processes, user doesn't notice.
- < 1 second response time (efficiency)
- swapping if processes don't fit in memory.

Date: ___/___/___

→ Virtual memory: enables multitasking system to ensure reasonable response time & run programs larger than physical memory.



Memory layout for a multi-programming system.

Dual-Mode & Multimode Operation:

- To ensure proper execution of OS → distinguish b/w execution of OS-code & user-defined code, hardware support is provided as a result.
- Two separate modes:
  1- User Mode
  2- Kernel Mode (privileged / supervisor / system mode).

- modebit: a bit in ~~do~~ hardware
  - (0) for kernel, (1) for user-mode

* At boot time, hardware starts in kernel mode.
* Trap or interrupt causes the hardware to switch from user to kernel mode
* privileged instructions can only be executed in kernel mode, if in user mode → illegal & trapped to OS.

machine instructions that may cause harm e.g: switch to kernel mode, I/O control, timer management etc.

Timer:

- To ensure OS maintains control over CPU
- To prevent a user program from being stuck in an infinite loop.
- set interrupts after specific period, & when transferring control.
- OS decrements counter after each clock tick, when counter = 0, interrupts.
- privileged instructions may modify timer.

Date: ___/___/___

Resource Management:

Process Management:

- A single-threaded process has one program counter (contains address of the instruction to be fetched).
- multi-threaded process has one pc per thread.

:Process Management activities:

1. creating & deleting both user & system processes.
2. Suspending & resuming processes.
3. process synchronization
4. process communication
5. deadlock handling.

Memory Management:

Memory Management activities:

- keeping track of which parts of memory are being used and who is using them
- deciding which processes (or parts thereof) and data to move into and out of memory
- Allocating & deallocating memory space.

File-System Management:

Activities:

- creation & deletion of files & directories
- manipulating files and directories
- Mapping files onto secondary storage
- Backup files onto NVMs

Mass Storage ~~system~~ Management:

- mount & unmount devices
- manage free space
- allocate storage
- schedule disk accesses
- perform partitioning
- protection.

Date: ___/___/___

Cache - Management:

→ Type of memory used to increase the speed of data access.

- Data available in cache memory → cache hit, otherwise cache miss → data is obtained from main-memory & copies it into cache

I/O System Management:

I/O subsystem:

- memory management component that includes buffering, caching & spol spooling.
- device drive interface
- Drivers for specific hardware devices

Virtualization:

- host OS → function simultaneously as different computers.

Distributed Systems:

- group of computers connected by a network allowing users to get services from all the individual computers.
- Some are transparent → only have be ≠ aware of one computer in front of you.

Network OS	Distributed System
. connected, but the computer is independent of other computers in the network.	· illusion that a single OS controls the network
. autonomous	· less autonomous

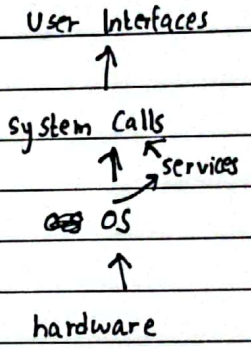
Date: ___/___/___

CH2: Operating System Structures

Operating System Services:

Services:

- User Interface:
  - GUI, touch-screen, CLI
- Program Execution:
  - Loading a program into memory & running it.
- I/O Operations: a means to do
  - provides I/O to a program
- File System Manipulation:
  - OS allows or denies access to file for processes.
- Communications:
  - shared memory, message passing etc.
- Error Detection:
- Resource Allocation
- Accounting
- Protection & security.



User & OS Interface:

Command interpreter (CI)/(CLI):

- multiple CIs are called shells
- function is to get and execute the next user specified command.
- can be internal or external

GUI:

- screen, mouse & keyboard interfaces.
- multiple windows & menus arranged according to desktop metaphor.
- KDE & GNOME are open software that run on Linux

Date: ___/___/___

Touch-screen:

- mobile devices
- tap and swipe / audio based

Which Interface to choose?

- CLI are powerful for doing challenging tasks but take longer time to run.
- GUIs & touch screen interface make it easy to learn to perform common tasks.

System Calls:

- It is a function call, request for an OS service
- Typically written in high-level language.

Application Progamming interface

- specifies a set of instruction to an application progam.
- Three common APIs:
  1- Windows API
  2- POSIX API
  3- JAVA API
- Advantages: (of using APIs over direct system calls)
  - program portability
  - easier to work with than system calls

- Run-time Environment (RTE): full suit of software needed to execute applications written in a given programming language.
  - provides a system call interface → serves as the link to system calls by OS.
  - system call interface intercepts function calls in API & invokes system calls in OS kernel & returns its status
  - system calls hidden by API & managed by RTE

Date: ___/___/___

·Three method & of system call Parameter Passing:

1- pass ~~pass~~ parameters in registers.
2- parameters stored in a block/table/memory & address of the block passed in a register.
3- push & pop from stack by OS.

TYPES OF SYSTEM CALLS:

1- Process Control:

· creation, termination, loading & execution of processes.
· getting & setting process attributes
· wait & signal an event
· allocate & free memory.

2- File Management:

· create, delete, open files etc.

3- Device Management:

· request to device
· release, read, write & reposition a device
· similar to file usage

4- Info maintenance:

· get time, ~~the~~ amount of free memory etc.

5- Communication:

· get host/process ID, open/close connections
· read/write messages etc.

6- Protection:

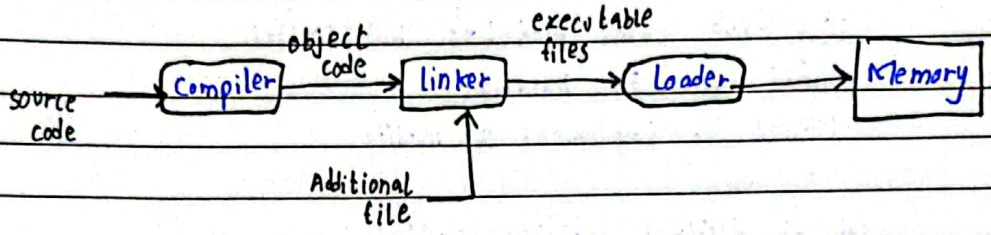
· get/set permission
· control access to resources.

Date: ___/___/___

System Services:

- System service software → layer outside kernel yet close to it.
- OS usually means kernel.
- Background services: launched at boot time, in user context.

Linker & Loader:



Why Applications are Operating-System Specific:

- To run applications, instructions must be written in machine language understood by computer.
- OS have APIs for sys programs to make system calls.
- No way to write one program that will run on dissimilar computer systems.

Operating-system Design & Implementation

- Convenient / reliable, easier to build.

Mechanisms	Policies
• how we do what we do • do not change over time • e.g: how a CPU scheduler works	• what we decide to do • changes over time e.g: FCFS policy

Implementation of OS:

- few parts written in Assembly language
- But nowadays almost everything written in high-level language as it is fast & convenient.
- Using better data structures & algorithms.

Date: ___/___/___

Operating System Structure:

1- Monolithic Structure:

- speed & efficiency in a monolithic kerned → single, static, binary file that runs in a single address space.
- May have some modules but limited.
- Direct access to hardware
- Difficult to implement & modify.

2- Layered Approach:

- set of distinct modules called layers.
- each layer is a data structure.
- layers numbered from lowest to highest
- code in one layer can make calls to functions in same or lower layers
- starts with hardware as the first layer
- easier to build
- command in higher layer can cause cascade of calls to lower layers & result in a delay (slow & inefficient).

3- Micro kernels:

- possible for file systems & device drivers to be user level programs.
- contains core process & memory managing functions.
- message passing service.
- modular
- reliable, easy to secure, maintain, modify & port.
- Might suffer from performance problems due to message passing.

4- Modules:

- similar to micro-kernel
- employs dynamically loadable kernel Modules (LKM) → not constrained to communicate with other kernels through core part of kernel.
- LKM may be loaded at boot time or when needed, can be deleted when not needed.
- e.g: device driver support module loaded when a device is plugged, when unplugged, module deleted.

Date: ___/___/___

5- Hybrid :

· combination of different OS structures.

Mac OS and IOS:

· Darwin → hybrid → mix of mach micro-kernel & BSD unix kernel.
· programs can make mach system calls & BSD calls.

Android:

· modified linux kernels.

Windows subsystem for linux:

· subsystem emulating other OSs.
· subsystem for linux → linux applications on windows
· linux translated into windows.

Operating System Debugging:

· OS generates log file containing error info.
· Failure of an app → core dump file capturing process's memory.
· if kernel fails → OS crashes & creates a crash dump file.

Operating System Generation:

· SYSGEN into & maybe incorporated into source code of OS and then the OS is compiled.
· System can also be pre-compiled & SYSGEN info saved in form of tabular info into files accessed by the pre-compiled OS
· Tables determie which dynamically Loadable modules are loaded at boot time.
· SYSGEN is followed by linking but not compiling step to incorporate modules into a permanent executable image of the kernel.

CHAPTER 3: "Processes"

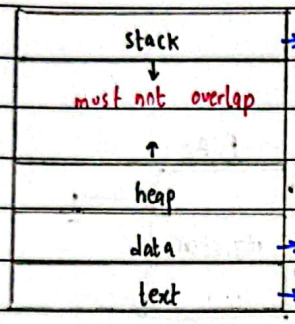
Date: ___/___/___

Process:

- A program in execution.

→ Batch systems: early computers that executed jobs

→ Time Shared systems: ran user programs or tasks



→ temporary data storage (can shrink & grow)

→ dynamic memory allocated at run time

→ global variable (fixed size)

→ executable code

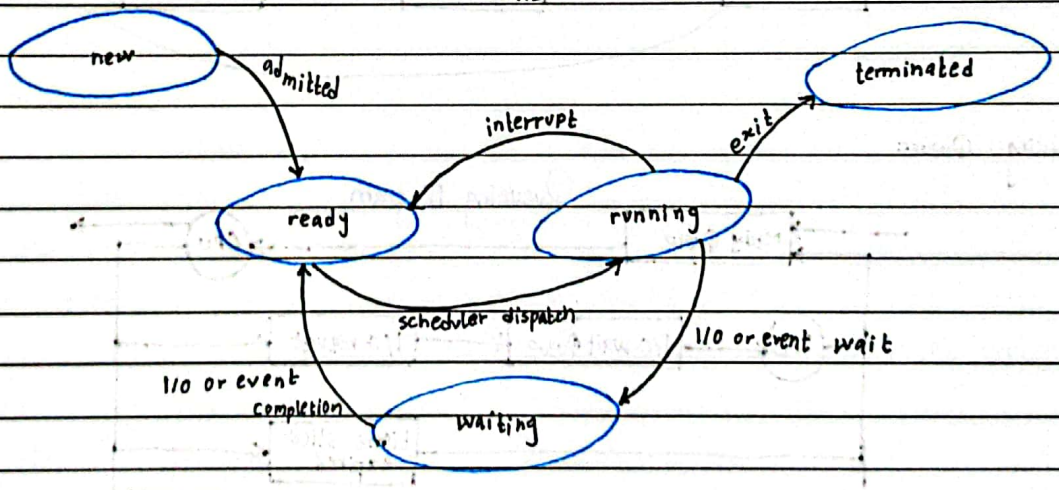
Fig 3.1: Layout of a process in memory

* status of current activity of a process is represented by PC & content of registers.

* program is not a process but a passive entity
  eg: executable file.

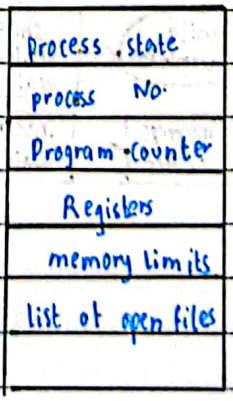
* process is an active entity
  eg: executable file loaded into memory.

PROCESS STATE:



Process Control Block (PCB) / Task Control Block:

- represents a process
- all the data needed to start or restart a process, along with some accounting data.

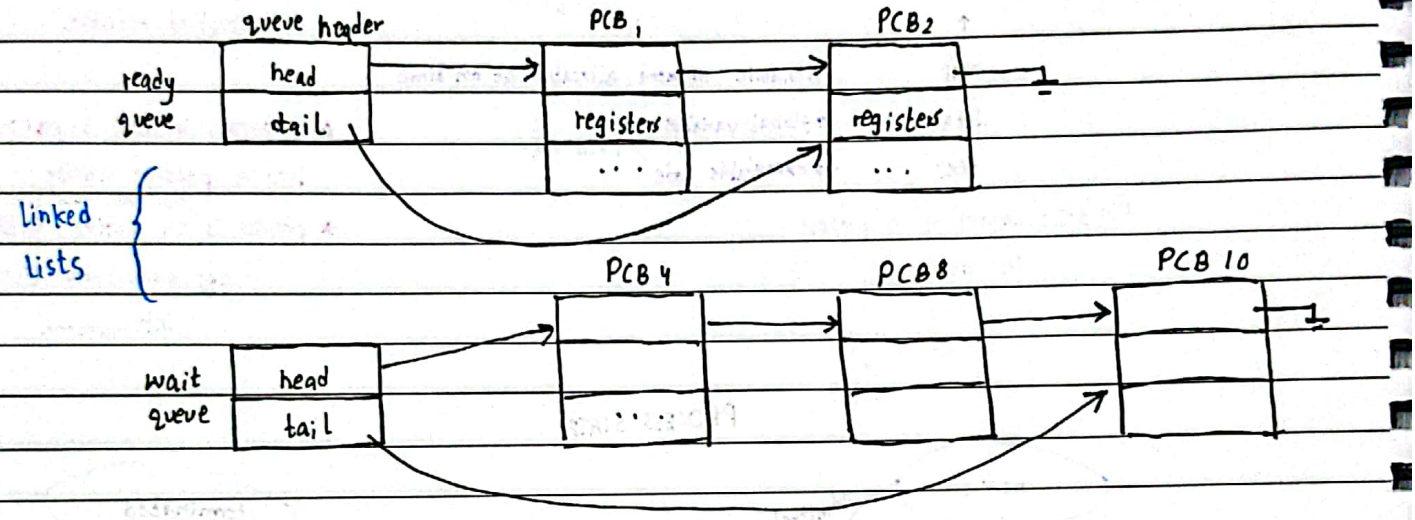


(PCB)

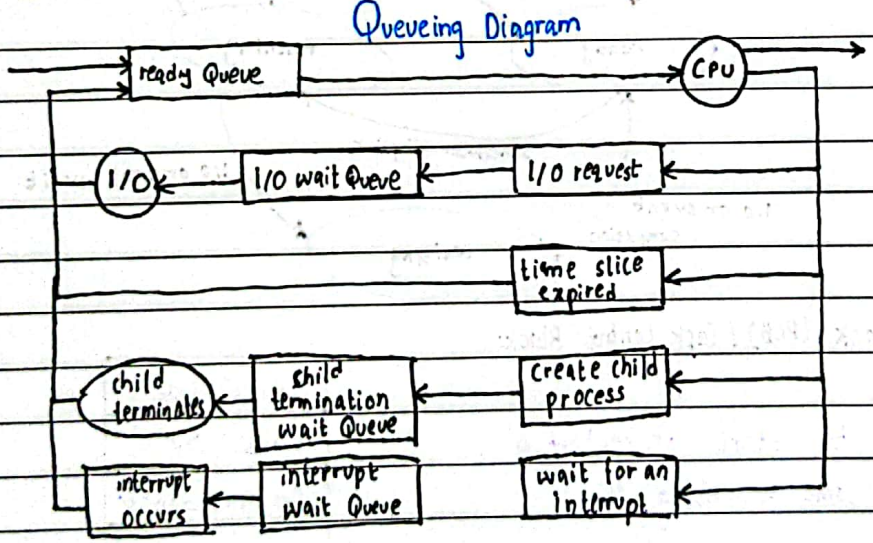
Date: ___/___/___

Process Scheduling:

- selecting an available process for program execution on a core.
- each core can run one process at a time
- Degree of multi-programming:
  - No. of processes currently in memory.
- Processes
  - ~~I/O Bound: more time doing I/O~~
  - CPU Bound: more computations



Scheduling Queues:



Date: ___/___/___

CPU Scheduling:

- CPU Scheduler: selects a process from ready queue & assign it to the CPU core
- likely to allocate a process with shorter execution time
- Might remove longer process from the CPU.
- Executes at least once every 100 milliseconds.

- Swapping: - a way of scheduling
  - swaps a process from memory to a disk
  - Used when memory is overcommited
  - reduces the degree of multi-programming.

Context Switch: (Context)

- saving the state of the process when an interrupt occurs.
- Context is represented on the PCB.
- Context switch: state save of the current process and state restore of a different process.
- switching creates an overhead
- usually several microseconds, depends varies from system to system due to it's hardware.

Process Creation:

- Parent Process: creating process
- Child Process: created process
- PID: Identification of processes by OS
- systemd: first process created when system boots (Parent Process)
  ps -el → lists all processes currently in the system.

- Child process obtains resources from OS OR the parent process.
- To prevent overloading, a child process is restricted to a subset of the parent's resources. (Too many child processe)

- When a process creates a new process:

sibilities for execution: 1- parent may continue to execute concurrently with its children

OR

2- parent may wait until some or all children have terminated.

Date: ___/___/___

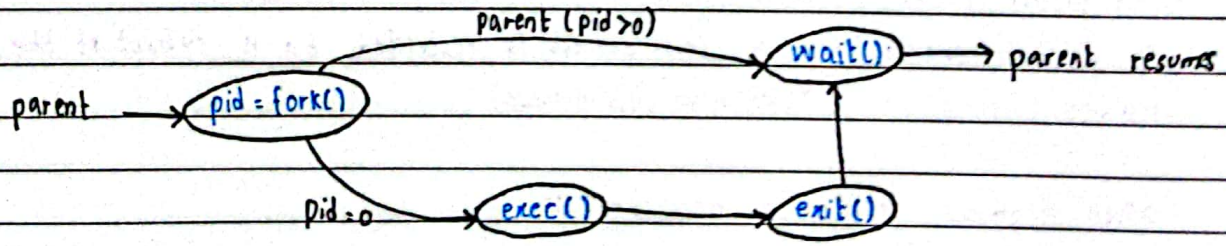
Possibilities for address space:

1- Child process is a duplicate of parent
2- child process is new.

- UNIX:
  - process creation using fork()
  - new process contains copy of the original process
  - execution is continued after fork()
  - return code:
    - 0 for child
    - Non-zero for parent

- exec() → - replaces process's memory space with a new program
  - loads a binary file into memory
  - ~~pro Ac~~ Any of the two process may use this system call
  - Both processes go separate ways.
  - If a child process calls it:
    - parent either waits using wait() or creates other child processes.
  - Does not return call unless an error occurs.

- Windows:
  - CreateProcess() is a combination of fork() and exec().



Date: ___/___/___

Process Termination:

- using exit(), process returns a status value to the waiting parent process.
- Resources of the process are reclaimed by the OS.
- A process can terminate another process.

Reasons for execution of children by parent process:

- child exceeds usage of the resources
- task assigned to child not longer required

Cascading termination → OS doesn't allow child to continue if the parent is exiting.

- A process's entry remains in the entry table until the parent calls wait()

- Zombie Process: process terminated but no wait() call by parent yet.
- Orphan Process: parent process doesn't invoke wait() & is terminated leaving children Orphan.
  - init process is asigned to orphans.

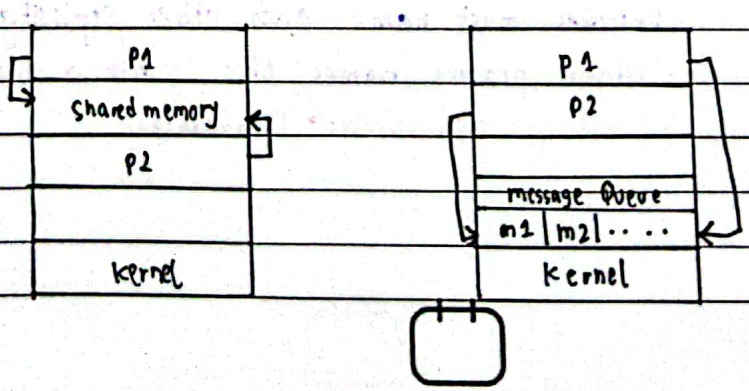
Interprocess Communication:

- Shared Memory: region of shared memory used by all processes
  - faster as no or fewer system calls.
  - restriction by OS is removed.
- Message passing: messages exchanged between processes.
  - useful for smaller amounts of data.

- Independent process: doesn't share data with other processes.

- Cooperating process: shares data, therefore can affect or be affected.

- Reasons for process Cooperation:
  - Info sharing: same piece of info needed by all the processes
  - Computation speedup: dividing into subtask for speed up.
  - Modularity: division into modules (threads)
  - Convenience.



Date: ___/___/___

Shared Memory:

Buffer: · region of space in shared memory
· filled by producer, emptied by consumer
~~or~~
· producer and consumer must be synchronized.
· Two Types of Buffers:

1- Unbounded:
· No limit on the size of buffer
· consumer may wait but producer keeps on producing

For Producer

```
while (true) {
    while (((in+1) % BUFFER_SIZE) == out)

    buffer[in] = next_produced;
    in = (in+1) % BUFFER_SIZE;
}
```

2- Bounded:
· fixed size
· consumer must wait if the buffer is empty

Message Passing:

Methods for logically implementing a link:

1- Direct Communication:
· link is established automatically
· Link is associated with exactly two processes
· Between each pair → only one link
· Processes must name each other explicitly. → symmetric
· either process names the other → asymmetric
· limited modularity → Disadvantage

Date: ___/___/___

2- Indirect Communication:

- mail box ~~system~~ → owned by either process or OS.
- link is established only if ~~both~~ a pair of processes share the same ~~the~~ mailbox
- link maybe associated with more than two processes
- Diff links may exist between ~~of~~ each pair of processes.

- There are different possibilities if multiple processer are to receive a message from a sender:
  - Allow a link to be associated with two processes at most.
  - Allow at most one process at a time
  - Allow the system to choose..

Synchronization:

- Message passing maybe blocking (synchronous) or non-blocking (asynchronous):

- Blocking Send: sending process is blocked until the message is received.
- Blocking Receive: receiving is blocked until the message is avaibable.

- Non-Blocking send: ~~send~~ sender sends the msg & continues
- Non-Blocking Receive: receiver receives a valid or null message.

Buffering:

1- Zero Capacity: Block send to ensure no msg is in queue
2- Bounded Capacity: finite length, if full → Block send
3- Unbounded Capacity: Infinite, no blocking.

Date: ___/___/___

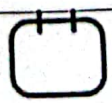
Examples of IPC Systems:

1. POSIX shared Memory:
   - unix oriented
   - shm_open(), ftruncate(), mmap() → creates a shared memory object which is in a memory mapped file.
   - Other processes other than the creator can open the and map the object using shm_open() & mmap().
   - shm_unlink() removes a shared memory object

2. Mach Message Passing:
   - employs mailboxes called ports
   - Remote Control procedure call (RPC) ~~is made → abstracts procedure calls between process on networked systems~~ implement with messaging
   - mach_msg() → send or receive
   - creator is default owner of port, rights can be transferred.
   - messages ~~Contai~~ consist of:
     - fixed-length header
     - variable sized body

   ★ Disadvantage:
   - Double copying, sender to kernel memory & kernel to recipient.
   - memory mapping is used to avoid this.

3- Windows:
   - communication via message passing facility called ALPC.
   - use ports like Mach.
   - as user establish communication by functioning as clients of
   - Three styles of message Passing:
     1- Double copying
     2- Shared memory
     3- Direct communication.



Date: ___/___/___

Communication in Client-Server Systems:

1-Pipes:

Ordinary Pipes:

- IPC, used in unix
- data structure
- allows pair of processes of same host → ~~con~~ to operate as consumer & producer.
- & pipe() creates pipe → two integers (file descriptors) →
  1. one to access writing end
  2. one to access reading
- Child inherits open files & pipes of parent
- uni-directional.

Named Pipes:

- FIFOs
- more powerful & feature rich
- appears as objects in the file system.
- bi-directional but one at a time.
- Two processes & must be on the same machine for FIFO

2- Sockets:

- endpoints of communication
- like a radio
- Network address (IP) and port nymeber; e.g: 161.25.19.8 : 1625 (host(IP) : port)
- low-level → communication of only unstructured data.
- communication exists between a pair of sockets

3- Remote Procedure Calls:

- transmission of structured data.
- allows a client to call a remote procedure the same way it does locally.
- calling remote procedure calls RPC system which calls stub procedure on the client side.
- function parameters are passed by RPC to stub.
- stub contacts a port on server.
- stub on server side receives msg & returns any results to client.
- network packets maybe lost or duplicated → solve this by using serial numbers & acknowledgements
- Port number of RPC maybe hard-wired
- Can be used to implement remote file service - NFS.

Date: ___/___/___

CH: 4 Threads & Concurrency

Thread: single-threaded process / multi-threaded process

- Basic unit of CPU utilization:
- Comprises of:
  - Thread ID
  - PC
  - registers
  - stack

code	data	files
registers	PC	stack
thread →		

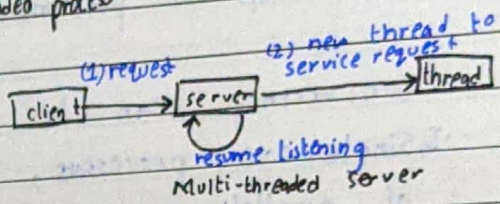
single-threaded process

code	data	files
registers stack PC	registers stack PC	registers stack PC

multi-threaded

for all threads same copy of data

- Context switching is faster in threads
- No system call is involved in threads
- Threads are interdependent.
- All user level threads are treated as a single task for OS.



Multi-threaded server

ps -ef → display the kernel threads

Benefits of Multi-threaded processes:

1. Responsiveness:
   - Allows a program to continue running, even if a part of it is blocked, as a result the program remains responsive to the user.

2. Resource Sharing:
   - No need for IPC techniques
   - Threads share memory & resource of the process by default, diff threads of activity within the same address space.

3. Economy:
   - less time & memory
   - faster context switching

4. Scalability:

Date: ___/___/___

Multi-core programming:

- each core can run a single thread
- Threads can run in parallel in a multi-core system.

- Concurrent system: allows more than one tasks by allowing all the tasks to make progress

- Parallel system: performs more than one task simultaneously.

∴ Concurrency can be achieved without parallelism

- In Single-core processors, schedulers switch rapidly, creating an illusion of parallelism.

Challenges in programming programs that are multithreaded for multicore systems:

1- Identifying task:
- examining applications to find areas that can be divided into separate, concurrent tasks.

2- Balance:
- ensure that tasks perform equal work of equal value.

3- Data splitting:
- dividing data to run on separate cores

4- Data Dependency:
- data accessed by tasks must be examined for dependencies b/w two or more tasks, to make sure execution of the tasks is synchronized.

5- Testing & Debugging:
- many diff execution paths are possible, therefore it is difficult to test & debug.

Date: ___/___/___

AMDAHL'S LAW:

- formula that identifies potential performance gains from adding additional computing cores to an application that has both serial and parallel components.

S = portion of application performed serially

N = " " " " parall

N = No. of processing cores.

$$\text{speed up} \leq \frac{1}{S + \left(\frac{1-S}{N}\right)}$$

* Maximum possible speedup is limited by the serial fraction of the program, regardless of how many processers are available. i.e: if a program has a large serial fraction, parallelizing it will not result in a significant speed up.

Types of parallelism:

1- Data Parallelism:

- dividing a large data into smaller subsets and distributing subsets of this data across multiple computing cores and performing the same operation on each.
- commonly used in appliation that involve large data sets.

2- Task Parallelism:

- distributing not data but tasks (threads) across multiple computing cores. Each thread performs a unique operation.

* Both are not mutually exclusive

Date: ___/___/___

Multi threading Models:

User threads: supported above the kernel and managed without kernel support

Kernel threads: supported and managed directly by the Os.

1 - Many-to-one Model:

- multiple user-level threads mapped to a single kernel-level thread.
- Thread management & scheduling is done in user space
- entire process blocks if a thread is blocked.
- Doesn't allow true parallelism.

2 - One-to-One Model.

- each user-level thread is mapped to a separate kernel-level thread.
- Provides true parallelism.
- significant overhead of creating & managing multiple kernel level threads.

3 - Many-to-many Model:

- maps many user-level threads to a smaller or equal no. of kernel threads.
- threads can run in parallel.
- if one thread is blocked, other is called for execution.

4 - ~~To~~ Two-level Model:

- hybrid of many-to-one and one-to-one.

Date: ___/___/___

Pthreads:

```
#include <pthread.h>

void * function (void* d);   → can be an object / value.

int main(){

    pthread_t thread;   pthread_attr_t attr;

    void * result;
```

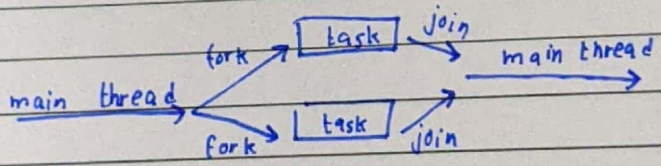
set default attributes of thred → `pthread_attr_init(&attr);` (OR &attr)

`pthread_create(&thread, NULL, function, &d);` → write NULL if no parameter is passed // creates thread

`pthread_join (thread, NULL);` // waits for thread to exit

Fork Join Model:

- ~~ena~~ parallel programming paradigm
- enable a program to divide its task into smaller subsets (tasks) and execute them in parallel on multiple processing units



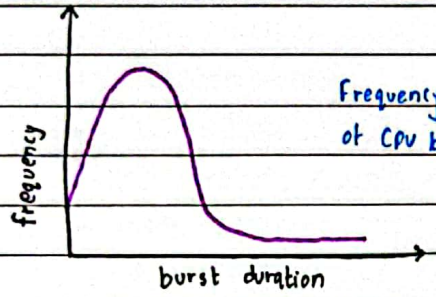
Date: ___/___/___

CH5: CPU Scheduling

CPU - I/O Burst Cycle:

- Process execution consists of a cycle of CPU execution & I/O wait, and they alternate in between these states.
- Begins ~~and ends~~ with a CPU burst.
- Ends " " CPU burst with a terminating system request.

Load store add store read from file	CPU burst
wait for I/O	I/O burst
store increment index write to file	CPU "
wait for I/O	I/O "
load store add store read from file	CPU "
wait for I/O	I/O "



Frequency curve of CPU bursts

* An I/O bound program has many short CPU bursts.
* A CPU bound program has few long CPU bursts.

* A read-queue is not necessarily a FIFO.

Pre-emptive & Non-preemptive scheduling:

Circumstances for CPU-Scheduling: When;

1 - A process switch from running to ~~ready~~ waiting state
2 - running to ~~waiting~~ ready
3 - waiting to ready
4 - a process terminates

non-preemptive
preemptive

Preemptive	Non-Preemptive
· CPU cycle allocated to a process for a limited	· process holds the CPU cycle till it completes its burst time.
· processes can be interrupted	· No interruption till termination or time up
· low priority process may starve because of a high priority one	· later comming processes with less CPU burst time may starve because of earlier long burst time processes.
· Overheads of scheduling	· No overheads
· cost associated	· no cost associated
· High CPU utilization	· Low CPU utilization
· less wait time	· high wait time

Date: ___/___/___

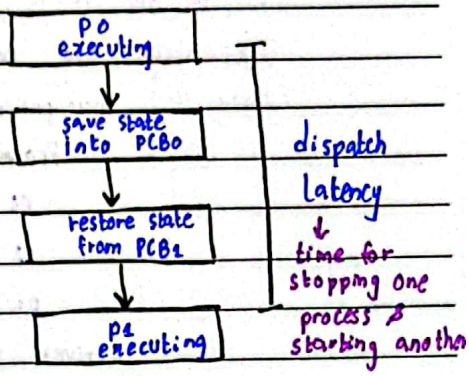
* Windows, linux, unix use Preemptive scheduling.

Dispatcher:

· Gives control of CPU's core to the process selected by scheduler.

function of dispatcher:

· switching context from one process to another
· switching to user mode
· jumping to the proper location in the user program to resume it



· Invoked during every context switch.

Scheduling Criteria:

· CPU Utilization:
  · CPU should be kept as busy as possible
  · 40% for a lightly loaded system
  · 90% for a heavy loaded system.

· Throughput:
  · No. of processes completed per unit time.

· Turn Around time:
  · interval from time of submission to time of completion.
  · = Burst Time + Waiting Time.

· Waiting Time:
  · sum of periods spent waiting in the ready queue.
  · Affected by CPU-Scheduling Algorithm.

· Response:
  · Time from submission to the time of first response

* for interactive system it is more important to minimize variance in response time rather than the avg response time

Date: ___/___/___

Scheduling Algorithms

First-Come, First-Served Algorithm: (Non-Preemptive)

- process that requests the CPU first is allocated the CPU first.
- PCB of the process in the ready queue is linked to the tail of the queue, when CPU is free it is allocated to the process at the head of the queue.
- Disadvantage: avg waiting time is quite long

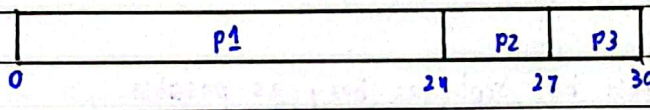
Convoy effect: short process behind long process (e.g: one CPU bound & many I/O bound processes).

#	Process	Burst time	WT	TA
	P1	24	0	24
	P2	3	24	27
	P3	3	27	30

Avg WT = $\frac{0+24+27}{3} = 17$

Avg TA = $\frac{24+27+30}{3} = 27$

- Arrives in order P1, P2, P3



Gantt Chart

Shortest Job First Algorithm: (Shortest-next-CPU-burst)

- process with shortest CPU burst
- if same burst for 2 processes, FCFS is used to assign CPU.

Determining length of Next CPU burst:

$$T_{n+1} = \alpha t_n + (1-\alpha)T_n$$

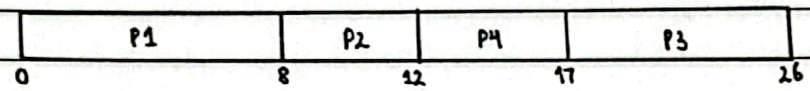
- t_n = most recent info
- T_n = Past history
- If $\alpha = 0$, then recent history has no effect
- if $\alpha = 1$, recent ~~history~~ CPU burst matters

Date: ___/___/___

~~Non-Preemptive SJF~~

Non-Preemptive SJF:

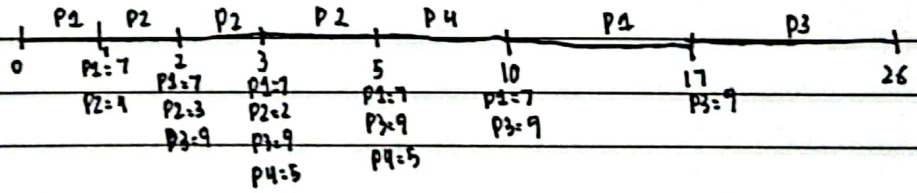
Process	Arrival Time	Burst Time	WT	TA
P1	0	8	0	8
P2	1	4	7	11
P3	2	9	15	24
P4	3	5	9	14



Avg WT = $\frac{0+7+15+9}{4}$ = 7.75ms Avg TA = $\frac{8+11+24+14}{4}$ = 14.25 ms

Preemptive SJF (Shortest Remaining Time First):

Process	AT	BT	WT	TA	
P1	0	8	9	17	Avg WT = 6.5
P2	1	4	0	4	
P3	2	9	15	24	Avg TA = 13
P4	3	5	2	7	



Date: 7 / 3 / 23

Determining Length of Next CPU Burst:

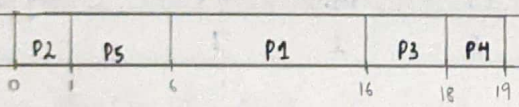
$T_{n+1} = \alpha t_n + (1-\alpha) T_n$

- T_{n+1}: predicted value for the next CPU burst
- t_n: recent info
- T_n: past history
- α: controls the relative weight of recent & past history

Priority Scheduling:

- priority no. for each process
- higher priority → executed first
- * SJF is a special case of priority scheduling.

Process	Burst Time	Priority	WT	TA
P1	10	3	6	16
P2	1	1	0	1
P3	2	4	16	18
P4	1	5	18	19
P5	5	2	1	6
			Avg: 8.2	Avg. 12



Problem of Priority Scheduling:

Starvation:

- Blocking: A process that is ready to run but waiting for CPU.

~~Starvation~~ Solutions:

1- Aging: gradually increasing the priority of processes that wait in the system for a long time.

2- Combining Round Robbin & priority scheduling, executing process with highest priority & running processes with the same priority using Round-Robbin.

Date: 7 / 3 / 23

Round Robbin → Mostly preemptive
→ Works on the basis of a ready queue (circular)
→ Time Quantum/Time slice

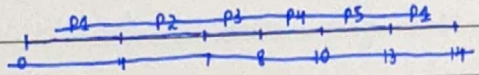
Quantum time = 2 unit

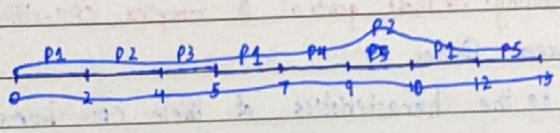
Process Id	Arrival Time	Burst Time	WT	TA
P1	0	5	8	13
P2	1	3	8	11
P3	2	1	2	3
P4	3	2	4	6
P5	4	3	7	10

Avg: ~~5.6~~ 5.8 Avg: ~~8.4~~ 8.6

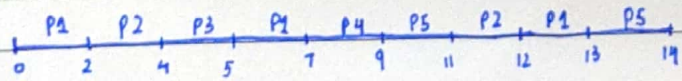
· preemptive

Red Ready Queue: ~~P1 P2 P3 P4 P5~~





Time	Ready Queue
0	P1
2	P2 P3 P1
4	P3 P1 P4 P5 P2
5	P1 P4 P5 P2
7	P4 P5 P1 P2 P1
9	P5 P1 P2 P1
11	~~P1 P5~~ P2 P1 P5
12	P3 P1 P5
13	P5



* Time Quantum should be large compared to context switch time, but not too large.
* 80 % of the CPU bursts should be shorter than the time Quantum.
* RR → FCFS, if time quantum is too large

Date: 8 / 3 / 23

→ separate queues for each distinct priority
→ O(n) search time in priority & RR

Multi-level Queue:

- Ready Queue partitioned into separate Queues base on type of processes:
  1. foreground (interactive) → · different response time requirements.
  2. Background (batch) → · diff scheduling algorithms for each.

- Four queues of diff types of processes:

1. Real-time processes — High Priority
2. System processes
3. Interactive processes
4. Batch processes — Low Priority

- Time-slice among queues
- preempt the processes.

Multi-level Feedback Queue Scheduling: → most general & complex CPU-scheduling Algorithm.

- Allows a process to move between Queues.
- Separating processes according to the characteristics of their CPU bursts.

How it works?

- each queue is assigned a time quantum
- If a process arriving in the highest priority queue does not finish within the time quantum it is moved to the tail of a lower priority queue.
- If the higher priority queue is empty, processes in the lower priority queue run.
- A process waiting too long in the lower priority queue is gradually moved up to a higher-priority queue - prevents starvation.

Date: ___/___/___

Multi-Processor Scheduling:

Asymmetric multiprocessing:

- Master processor/server - all system data structures ~~handle~~ accessed by only one core
- Other processors only execute user code.

Disadvantage:

- Master server becomes a bottleneck

Symmetric multi-processing (SMP):

- each processor is self-scheduling
- scheduler for each processor examine the ready queue for a thread

Two possible strategies for organizing threads to be scheduled:

race condition ← 1- Threads in a common ready queue

2- each processor may have its own private queue of threads

load Balancing:

- keeps workload evenly distributed across all processor in an SMP system
- Unecessary on systems with a common ready queue

Two approaches:

1- Push Migration:

- a specific task periodically checks load on each processor
- evenly distributes by moving thread from overloaded to idle or less busy processors.

2- Pull migration:

- when an idle processor pulls a waiting task from a busy processor.

Date: ___/___/___

Processor Affinity:

- OS with SMP avoiding migrating threads from one processor to another to avoid high cost of maintaining cache.

Two Forms:

1- Soft Affinity:

- No guarantee that a process will run on the same processor
- possible to migrate during load balancing.

2- Hard Affinity:

- allowing a process to specify a subset of processor on which it can run.

Heterogeneous Multiprocessing:

- a system in which more than one processor is connected and is active at the same time.

Date: ___/___/___

CH: 6 Synchronization Tools

Race Condition: → Solution: synchronization

- Several proceses accessing & manipulating the same data concurrently

The Critical section problem:

- To design a protocal that the processes can used to synchronize their activity to cooperatively share data.

entry section: where process request permission to enter critical section.

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

- Three requirements:

1- Mutual exclusion:
- only one process executing in its critical section at a time.

2- Progress:
- Only the processes not executing in the remainder section can enter the critical section, it no other process is executing in its critical section

3- Bounded waiting:
- limit to how many times other processes can enter their critical sections after a process has made a request.

- Kernel code subject to several possible race conditions:
  - processes in a kernel data structure updating the same file.
  - Two processes creating a child process using fork() at the same time
  - Kernel data structures for maintaining memory

- Two General approaches to handle critical sections in OS:

1- Preemptive:
- allows process to be preempted when in kernel mode

Disadvantages:
- subject to race condition.
- Difficult to design in smp architecture.

Advantages:
- More responsive - less risk of a kernel mode process will run for long period.
- More suitable for real time programming

Date: ___/___/___

2- Non-preemptive kernel:

- free from race conditions
- does not allow a process running in kernel mode to be preempted.

Peterson's Solution: → classic software based solution

- restricted to two processes that alternate execution between their critical sections & remainder sections.

```
int turn;
boolean flag[2];

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

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

Pi

Pj

- Two processes alternating, each process has a flag and a turn variable. Initially both processes set flag to false. Processes alternate setting the turn variable to their process no. To enter Critical section, a process sets its flag variable to true and checks if it is it turn to enter, if not the process enters a busy-waiting loop and continuously checks the turn variable. If it is its turn, it enters C.S and executes the code that accesses the shared resource. After it exits C.S, it sets its flag to false & the other process can now enter its C.S.

Date: ___/___/___

Limitations of Petersons solution:

1. limited to two processes
2. Busy waiting - consumes CPU resources
3. Non-preemptive.
4. limited to shared memory
5. limited to single critical section

Hardware Support for synchronization:

Memory Barriers / fences:

- hardware construct that enforces a specific order of memory operations in a computer system
- ensures operations performed by multiple processors are synchronized.
- ensures memory operations are visible to other processors
- low-level operations

Test & Set:

```
boolean test_and_set(boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
}
do{
    while (test_and_set(&lock));
        /* critical section */
    lock = false
        /* remainder section */
} while (true);
```

- executed atomically
- initially value of lock is false, when a process tries to enter the critical section, it calls test_and_set() which returns a temp variable set to the current value of lock, since lock is false initially, value returned is also false, ~~process en~~ lock is set to true and the process enters the critical section, a new process cannot enter ~~the~~ its critical section until the first process exits its critical section.

Date: __/__/__

Compare and Swap:

```
int compare_and_swap(int *value, int expected, int new_value){
    int temp = *value;

    if (*value == new_value)
        *value = new_value;

    return temp;
}

while (true) {
    while (compare_and_swap(&lock, 0, 1) != 0);
        /* critical section */
    lock = 0
        /* remainder section */
}
```

- Doesnt guarantee bounded waiting

~~Mutex locks~~

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = ~~lock_and_set~~ compare-and-swap (&lock, 0, 1)
    waiting[i] = false;
        // critical section
    j = (i+1) % n;
    while (j != i && !waiting[j])
        j = (j+1) % n;
    if (j == i)
        lock = FALSE;
    else
        waiting[j] = FALSE;
        // remainder section
} while (TRUE);
```


Date: ___/___/___

Mutex Lock: → software based solution to Critical section problem

- To protect critical sections and ✗ prevent race conditions.
- process must acquire the lock before entering the critical section.

```
while (true){
    acquire lock
        /* critical section */
    release lock
        /* remainder section */
}
```

```
acquire (){
    while (!available); // busy waiting
    available = false;
}
```

```
release(){
    available = true;
}
```

Advantage:

1 - Spin lock - no need for context switching

Disadvantages:

1 - Busy waiting - loop in the while loop
2 - Deadlock - if mutiple threads or processes acquire mutex locks in a different order.
3 - Complexity
4 - Scalability

Semaphores:

- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: wait() and signal().

```
wait(S){
    while (S<=0); // busy wait
    S--;
}
```

```
signal (S){
    S++;
}
```

- executed atomically

Date: ___/___/___

- Two types of Semaphores:

1- Binary semaphore:

- also known as mutex semaphore
- two possible values: 0 and 1
- If S=0 S=1, the thread or process can access the resource and then sets S to 0.
- when the thread or process has finished accessing the resource, it releases S and sets it back to 1.

2- Counting Semaphore:

- any non-negative integer value
- S is initialized to be the number of resources available.
- if S>0, the process can access the resource and waits on the counting semaphore.

Usage:

- can be used to limit the no. of threads/processes that can access the shared resource simultaneously.
- can ensure that a certain sequence of events occurs in the correct order.

Semaphore implementation to solve busy waiting:

```
type def struct {
    int val;
    struct process * list;
} semaphore;

wait (semaphore *S) {
    S->value --;
    if (S->value <0) {
        add this process to S->list;
        sleep();
    }
}
```

Date: ___/___/___

```
signal (semaphore *S){
        S->value++;
        if (S->value <=0){
                remove a process P from S->list;
                wakeup(P);
        }
}
```

- The processes instead of busy waiting are now added to a waiting list of the semaphore and are suspended, when the other process executes a signal() operation, a process in the waiting queue is restarted by wakeup() and is placed in the read queue.
- Negative value of S = magnitude of no. of processes waiting on S.
- list of waiting processes can be implemented by a link field in each process control Block (PCB).
- Inhibit interrupts to ensure that semaphore operations are executed atomically
- In this implementation, busy waiting is not eliminated completely.

Limitations of Semaphores:

1 - More complex to implement than mutex locks.
2. Overhead due to synchronization
3. deadlocks
4 - priority inversion

Date: ___/___/___

Liveness:

- It is the ability of a system to continue to make progress over time
- Liveness failure: process waiting indefinitely

Situations leading to liveness failures:

1- Deadlock:

- occurs when two or more processes are blocked, waiting for each other to release a resource, so that they can proceed.

2- Priority inversion:

- occurs when a lower priority task holds a lock or resource that is required by a higher priority task

To solve it use:

- Priority inheritance:

  - priority of a task holding a lock or resource is temporarily elevated to the priority of the highest priority task waiting.

Date: ___/___/___

CH:7 SYNCHRONIZATION EXAMPLES

Classical Problems of Synchronization:

1- Bounded-Buffer Problem:

```
int n;
semaphore mutex = 1
"         empty = n
"         full = 0
```

Producer

```
while (true) {
    wait (empty);
    wait (mutex);

    signal (mutex);
    signal (full);
}
```

Consumer

```
while (true) {
    wait (full);
    wait (mutex);

    signal (mutex);
    signal (full);
}
```

- The producer must not insert data when buffer is full.
- The consumer must not remove data when buffer is empty.
- Producer and consumer should not insert & remove simultaneously.

2- Readers Writers Problem:

- writer and any other thread (reader or writer) try to access the shared resource simultaneously.

Writer Process

```
do {
    wait (wrt);
    /* writes */
    signal (wrt)
} while (true);
```

Reader

```
do {
    wait (mutex);
    readcnt++;
    if (readcnt == 1)
        wait (wrt);
    signal (mutex);
    wait (mutex);
    readcnt--;
    if (readcnt == 0)
        signal (wrt);
    signal (mutex);
} while (true);
```

Date: ___/___/___

int readcnt=0; → it ~~we~~ a process wants to update it, it should always acquire mutex first, tracks no. of processes

~~int wrt=1~~

Semaphore mutex:1

Semaphore wrt:1 → for both reader and writer

3- Dining Philosopher's problem:

- 5 philosophers & 5 chopsticks with a bowl of rice at the center.
- Two chopsticks to eat.

Solution 1:

- represent each chopstick with a semaphore
- A philosopher tries to grab a chopstick by executing wait() operation.
- Chopsticks are released by executing signal().

- Semaphore chopstick[5]; // binary semaphore
- All initialized to 1

```
while (true){
    wait (chopstick[i]);
    wait (chopstick[(i+1)%5]);
        /* eats */
    signal (chopstick[i]);
    signal (chopstick[(i+1)%5]);
        /* thinks */
}
```

* Creates a deadlock if all philosophers become hungry at the same time setting each chopstick to 0, grabbing right chopstick takes forever.

Other solutions:

- Allow at most 4 philosopher to sit simultaneously.
- Philosopher picks chopsticks only if both chopsticks are available.

Date: ___/___/___

- Asymmetric solution: odd philosopher first picks up ~~rig~~ left then right chopstick, even philosopher picks up right then left.

Date: ___/___/___

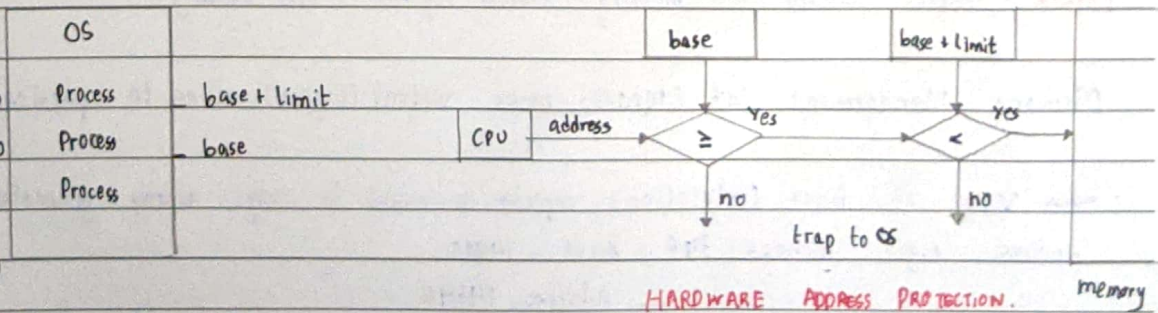
CH:9 Main Memory

- CPU can only access Main memory & registers directly
- Registers accessible within 1 cycle of CPU clock.
- Main memory access through memory bus takes multiple clock cycles, leading to CPU stalls
- **Cache** is used to mitigate the latency of main memory access.
  - ↳ Fast memory on CPU chip that stores frequently accessed instruction & data
  - ↳ speeds up memory access
- Hardware based protection mechanism are necessary to ensure proper system operation, this is done by using base and limit registers.

Base & Limit Registers:

Base & limit register define the logical address space.

- Base register holds the smallest legal physical memory address
- limit register specifies the size of the range.
- CPU compares the addresses generated in user mode with the registers.
- ~~If a user pro~~ A trap is generated if a user program attempts to access memory outside this address.
- Base & Limit registers can only be loaded by the OS using privileged instructions executed in kernel mode, preventing access from user programs.
- OS in kernel mode has unrestricted access to both OS memory & user's memory allowing OS to manage processes, perform I/O etc.

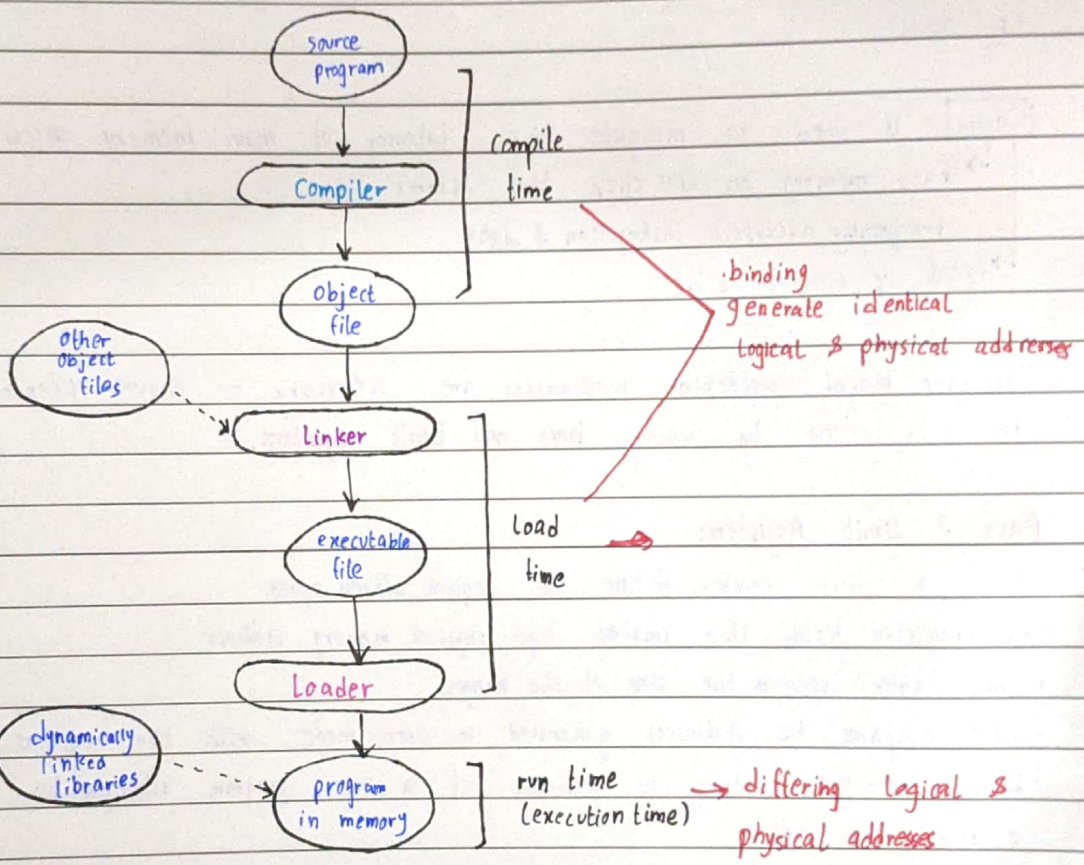


HARDWARE ADDRESS PROTECTION.

Date: ___/___/___

Address Binding:

- A program resides on a disk as a binary executable file, executed in main memory.



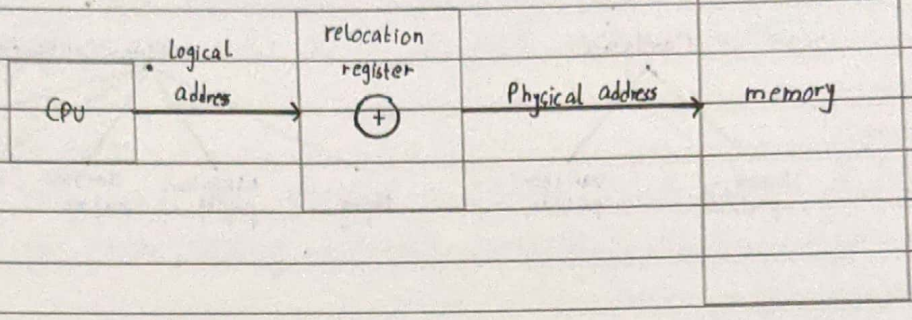
Logical VS Physical Address:

Logical - address generated by CPU
Physical - address loaded into memory address register of the memory.

- Memory Management Unit (MMU): maps virtual (logical) address to physical at run time.
- ~~base~~ Value of base (relocation) register is added to every address generated by a user process e.g: address 346, base = 14000
  ∴ address = 14346

Date: ___/___/___

- User program deals with logical address.



Dynamic loading:

- A routine is loaded into physical address space only when needed.
- Doesn't require help from OS.

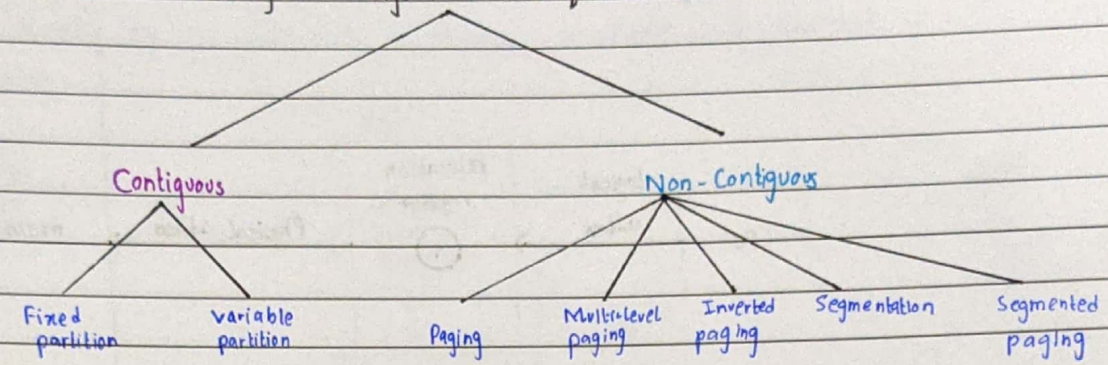
Dynamic linking & Shared libraries:

Dynamically linked libraries (DLL): system libraries linked to user programs at run time.

- reduces size of executable images, as ~~there is no~~ DLLs can be shared among multiple processes, so that one instance of DLL is in memory.
- DLLs support library updates, allowing programs to automatically use new versions without relinking.
- Requires help from OS, to protect processes and manage memory address

Date: ___/___/___

Memory Management Techniques



Memory Protection

- Preventing a process from accessing memory that it does not own.
- Achieved by combining the use of relocation and limit register.
- By checking every address against these registers, the system can prevent a process from modifying the OS or other user programs.
- Relocation registers allow dynamic allocation to CPU.

Memory Allocation:

Contiguous:

Variable Partition (Dynamic storage allocation):

- Assigning processes to variable sized partitions (holes), scattered throughout.
- If no space → wait queue.
- If two holes adjacent → merged to form one hole.
- If hole too large → split into two → one used by process.

Which hole to allocate?

- better than worst fit in terms of storage & speed
- first fit faster than best fit.

First fit: first hole big enough
Best fit: Smallest hole " "
Worst fit: Largest hole

Fixed Partition. (Static):

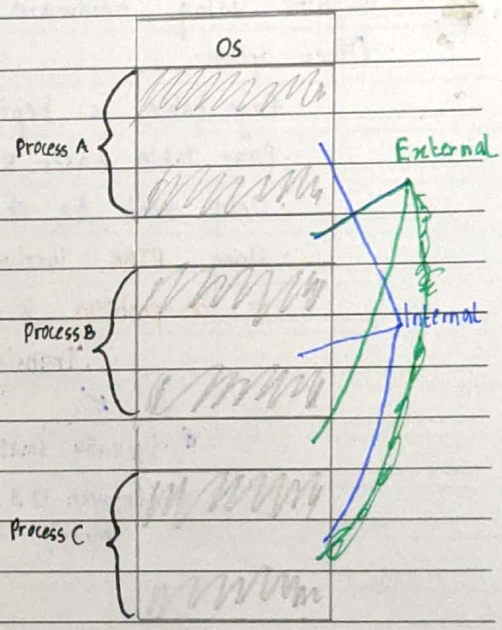
- physical memory divided into fixed-size partitions (pre-determined size).
- If partition not large enough, process cannot be loaded.

Date: ___/___/___

Problems of Contiguous Memory Allocation:

- External Fragmentation:
  - Memory blocks become scattered as processes are loaded & unloaded, creating small non-contiguous blocks, making it difficult to allocate larger processes that require contiguous memory space.

- Internal Fragmentation:
  - Occurs when the allocated partition to a process is much larger than the process, leading to memory wastage.



Solutions:

- Compaction: (for external fragmentation)
  - Shuffle memory blocks, so that they are contiguous
  - Only possible if relocation is dynamic
  - Change base register & move program & data.

- Non-Contiguous Memory Allocation

1- Paging:

- Division of Physical & virtual memory into fixed sized blocks called frames & pages respectively.
- Pages of a process are ~~to~~ Loaded into frames on execution.

index to page table ← Page number | Page offset → where to place in a frame.

p	d
m-n	n

2^m = logical address space size
2^n = page size.

- How MMU maps logical into physical address?
  - using page number as index extracts frame number.
  - replaces page number with frame number
  - frame number + page offset = physical address

- Calculation to place pages in frames:
  = (frame No. x page size) + offset → (where the part of a page is in the page)

Page 0

0	0	a
1	1	b
2	2	c
3	3	d

→ offset of a is 0, as it is at the start of the page.

Date: ___/___/___

- No external fragmentation, as free frames can be allocated.
- Internal Fragmentation : c
  - worst case : n pages + 1 byte.

- Frame Table: - keeps track of frames
  - Tells status of the frame

Implementation of Page Table:

- If done using hardware register, context-switch time is large.
- Other option:
  Page table is kept in the main memory.
  - Page Table Baser Register (PTBR) points to the page table.
  - Only need to change PTBR to change page Tables.
  - Using PTBR involves 2 memory accesses which results in a delay, solution to this problem is to use:
    - Translation Look Aside Buffer (TLB):
      - typically small (between 32 & 1,024 entries)
      - Associative, high speed memory.
      - Contains few page table entries
      - MMU checks page no. in TLB
      - If page found in TLB → TLB hit, otherwise TLB miss.
      - If TLB is full, Least Recently Used (LRU) page entry is replaced.
      - Contains page & Frame Number
      - Some TLBs store Address Space Identifier (ASID) → identifies each process uniquely, as a result TLB can store entries for diff processes.
      - If no ASID, TLB is ~~fush~~ flushed for diff processes.

Date: ___/___/___

Hit Ratio: Percentage of times the desired page no. is found in TLB. (α)

Effective Access Time (EAT): avg time required to access the desired byte in memory.

Formula:

Hit ratio = α Miss ratio = $(1-\alpha)$

EAT = α x memory access time (hit) + $(1-\alpha)$ x memory access time (miss)

Protection: (Paging):

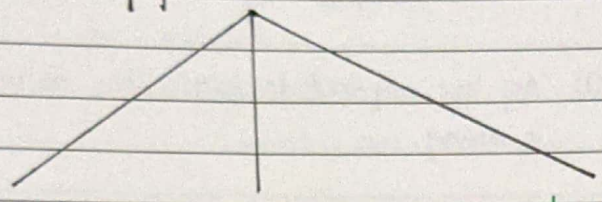
- Achieved using protection bits, kept in the page table.
- Protection bits define a page to be read-write or read-only.
- Valid-Invalid bit:
  - attached to each entry in page table.
  - Valid: → page is in process's logical address space.
  - Invalid: → page is not in " " " "
- Page Table length Register (PTLR):
  - indicates size of the page table which is checked against size of every process.

Sharing in Paging:

- Reentrant (Read-Only) code shared among processes that appears in the same location in the physical address space.
- Only one copy of code in physical address space.

Date: ___/___/___

Structure of the Ppage Table.

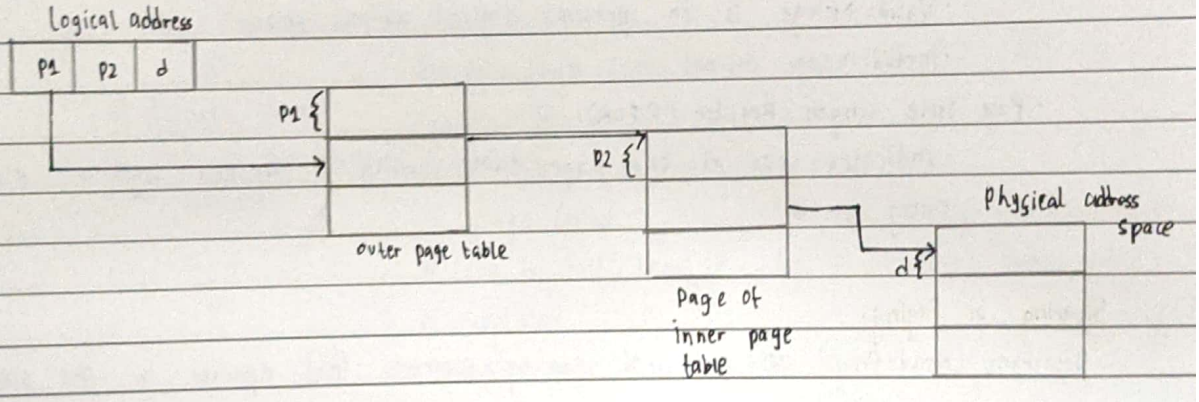


Hierarchal:

- If page table becomes extremely large, it is sensible to divided it:

  e.g: A Two-Level page-table

- In a two-level page table the logical address is divided into multiple levelsg of page tables:
  - 1st level – contains entries pointing to 2nd page table
  - 2nd level – consists of individual page tables.



Hashed:

- hash function takes page no. as input and generates hash value which determines where this page table entry is stored.
- same concept as hashing (OSA), in case of collision separate chaining.

Inverted:

- Entries are based on physical ~~address~~ memory frames instead of logical ~~pa~~ address
- Each entry contains physical frame and mapping info for the process. (page no.).
- ~~Fo~~ Used for systems with memory constraints.

Date: ____ / ____

Swapping:

- process swapped temporarily out of main memory to a backing store.
- Increases degree of multiprogramming.
- Standard swapping: Swapping entire process.
- Most OS ~~now~~ only swap pages of a process rather than the entire process.

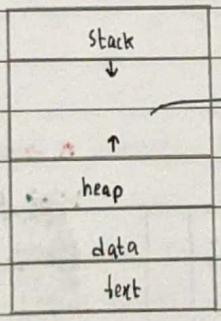
Date: ___/___/___

CH# 10 "VIRTUAL MEMORY"

- Memory management technique
- Illusion of having more physical memory than actually available
- Process having size > size of main memory can be executed.
- Increases Degree of Multi-programming
- Part of a process is moved to the main memory, as a result more no. of processes can be moved into the main memory.
- Virtual Memory is an area of secondary storages used by OS for swapping.

Virtual address space:

- Each process in OS has its own virtual address space (logical address space), which is divided into # pages.



→ used for dynamically linking libraries & share da memory
↳ sparse address space

Page Fault: → Occurs due to demand paging → ~~valid-invalid bits~~ - tells whether the page is in memory or not & legal or illegal

1- reference made to the page
2- Interrupt that occurs when a process or program attempts to access a page of memory not currently present in main memory.
3- Due to interrupt, control transfers from user to OS.
4- OS now looks for the page in secondary storage, moves it into main memory and 5- then updates the page table.
6- OS now transfers control back to user

~~EAT~~

Let p → probability of page fault

EAT = p x (page fault time)$_{ms}$ + (1-p) x (main memory access time (MM))$_{ns}$

page fault time = fault overhead + swap out + swap in

Date: ___/___/___

Pure Demand Paging: bringing a page into memory when needed.

- In case of instruction, if a page fault occurs, the instruction need to be restarted, fetched and executed again.

Copy-on-Write:

- Allows parent & child processes to share pages
- When a process modifies the shared page only then that page is copied

Page Replacement:

- If there is no free frame then we replace a page.
- Reduce page faults.

1 - FIFO:

Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,1,2,0

* → page fault
v → page hit

f_3			1	1	1	1	0	0	0	3	3	3	3	2	2
f_2		0	0	0	0	3	3	3	2	2	2	2	1	1	1
f_1	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0
	*	*	*	*	v	*	*	*	*	*	*	v	*	*	v

* Replace frame wise, f1 then f2 the f3 ....
* If page exists in either of the frames it is a hit otherwise fault.

~~Page hit~~ ~~Page fault~~
Page miss

Page hit = 3

Page fault / miss = 12

Hit ratio = $\frac{\text{No. of hits}}{\text{No. of references}} = \frac{3}{15} = \frac{1}{5}$

Date: ___/___/___

Q.

f3			3	3	3	2	2	2	2	2	4	4	
f2		2	2	2	1	1	1	1	1	3	3	3	
f1	1	1	1	4	4	4	5	5	5	5	5	5	
	*	*	*	*	*	*	*	✓	✓	*	*	✓	

Hits = 3

page fault = 9

Ref string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Repeating the question above using 4 frames

f4				4	4	4	4	4	4	3	3	3	
f3			3	3	3	3	3	3	2	2	2	2	
f2		2	2	2	2	2	2	1	1	1	1	5	
f1	1	1	1	1	1	1	5	5	5	5	4	4	
	*	*	*	*	✓	✓	*	*	*	*	*	*	

Hits = 2

page faults = 10

Belady's anomaly: page faults increases on increasing No. of frames.

2- Optimal Page replacement:

- Replace the page that will not be used for the longest period of time.

f4				2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
f3			1	1	1	1	1	4	4	4	4	4	4	1	1	1	1	1	1	1
f2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f1	7	7	7	7	7	3	3	3	3	3	3	3	3	3	3	3	3	7	7	7
	*	*	*	*	✓	*	✓	*	✓	✓	✓	✓	✓	*	✓	✓	✓	*	✓	✓

Ref String: 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

Hit = 12

fault = 8

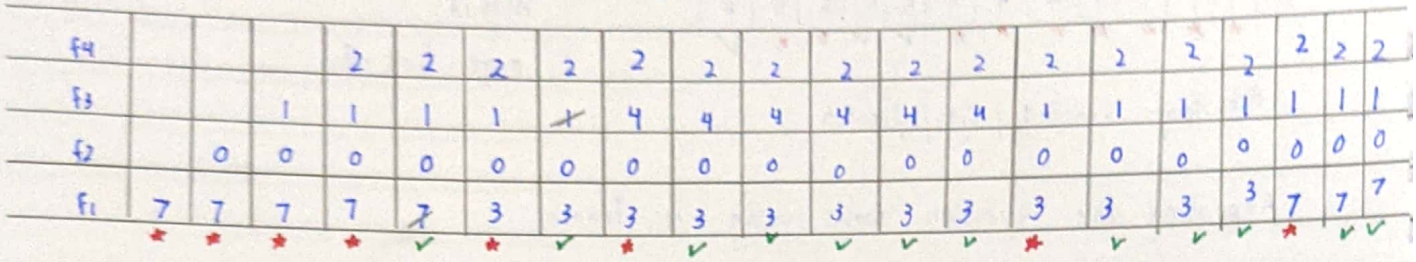
replace either 3 or 4 as both are not in the list

* Replace the page farthest away from the page you are currently trying to add to the frames.
* if more than 1 pages are not in the list, replace either.

Date: ___/___/___

3- LRU Page Replacement:

- Replace the least recently used page in the past.



Ref String: 7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1

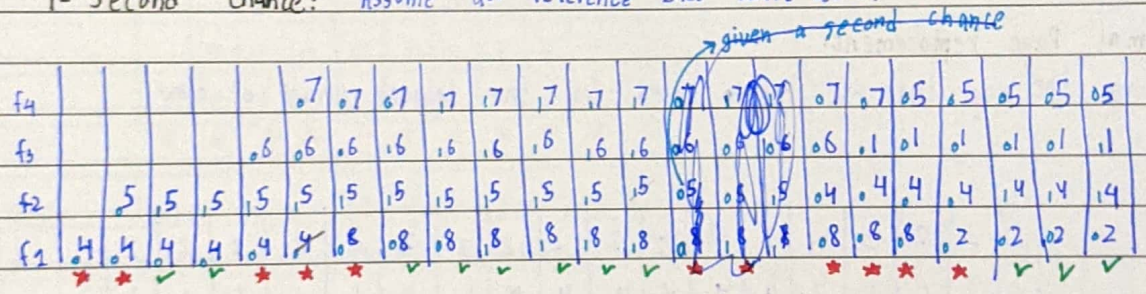
Hits = 12

faults = 8

4- LRU Approximation:

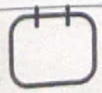
- approximates the behavior of LRU without incurring its high computational overhead

i- Second Chance: (FIFO) Assume all reference bits start at 0.



Ref String: 4,5,5,5,6,7,8,6,7,8,6,7,8,4,1,5,2,4,4,1

- Replace using fifo
- If reference bit of a page in frame is '1', decrement it to '0' if you need replacement, look for other pages in other frames and replace the page whose reference bit is '0'.



Date: ___/___/___

Frames Allocation Algorithms:

1- Fixed:

i) Equal:

No. of frames of a process = Total No. of frames / No of processes

ii) Proportional:

No. of frames of a process p_i = (size of process p_i / total size of all processes) x Total No. of frames.

2- Priority:

- Use a proportional algorithm but with priorities rather than size.
- If p_i generates a fault replace the frame that has low priority process.

Global vs Local Replacement:

Global:

- process selects a replacement frame from set of all frames.
- ~~more~~ execution time varies greatly as a result.
- Greater throughput.

Local:

- process selects a replacement frame from its own set.
- More efficient.
- Memory underutilized.

4 - 0

Date: ___/___/___

Thrashing:

- Occurs when a significant amount of time is spent by the system on excessive paging activity, resulting in a decrease in overall system performance
- Due to vicious cycle involving high page fault rates.

Solutions:

- Increase Main Memory
- long term scheduler.