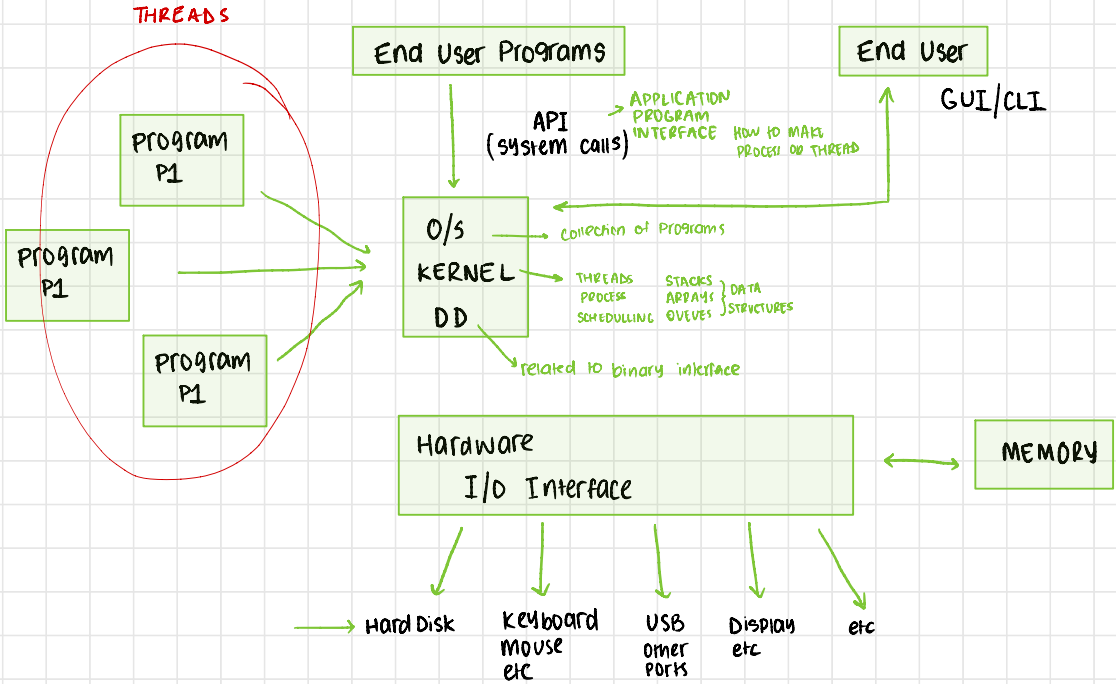
Operation Systems

Operating System

ABI
APPLICATION BINARY INTEFACE

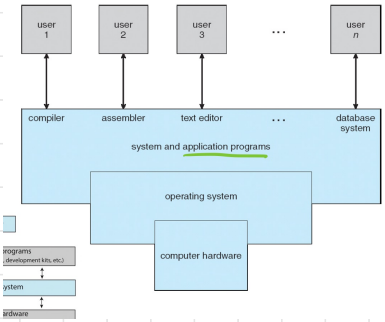


3 CONCEPTS in O/S

- Virtualization → PROCESS IS A VIRTULISATION → not in BATCH
  architecture, H/W for each user programs
  give acess of all process to hardware together
- Concurrency → multiple threads → inside the process
  - similar to a process but part of a process
  - divide data into chunks
    - T_1 T_2 T_3 → threads
    - 50K 50K 50K
  - for (i=0 200000000
    a[i]: b[i] + c[i]
- Persistence
  - not in memory
    data/program storage that can survive reboots
  - How to run more than one instruction stream in one program
    using API's

CMP 1

- Computer system can be divided into four components:
  - Hardware – provides basic computing resources → CPU, memory, I/O devices
  - Operating system – controls hardware and coordinates application programs use
  - Application programs – define ways to use these resources to solve computer problems → word processors, compilers, spread sheets, web browsers
  - Users



→To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory

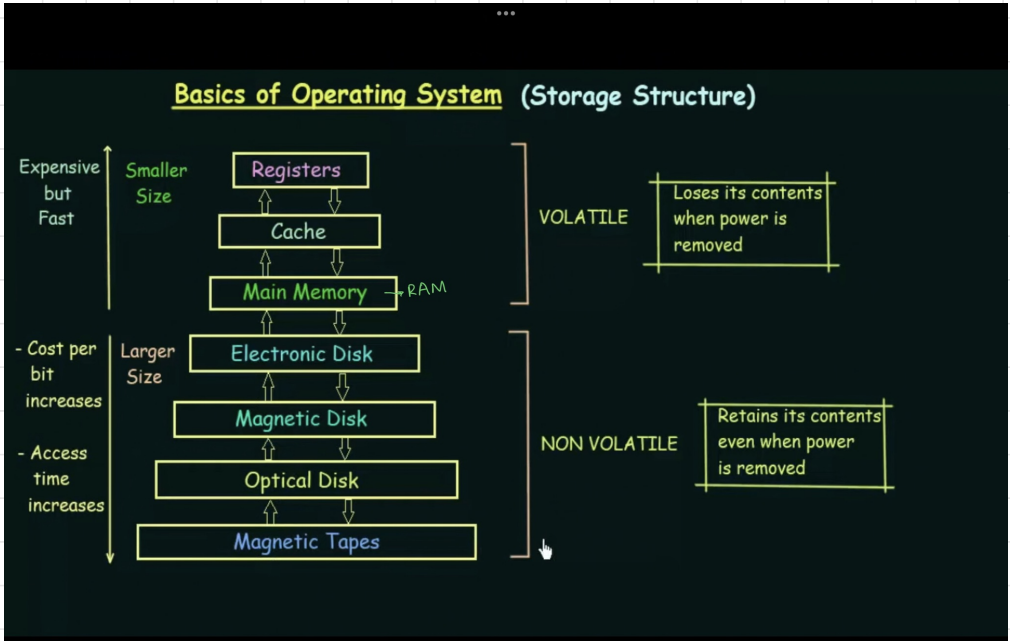
Some important terms:

1) Bootstrap Program:→The initial program that runs when a computer is powered up or rebooted.
→It is stored in the ROM.
→It must know how to load the OS and start executing that system.
→It must locate and load into memory the OS Kernel.

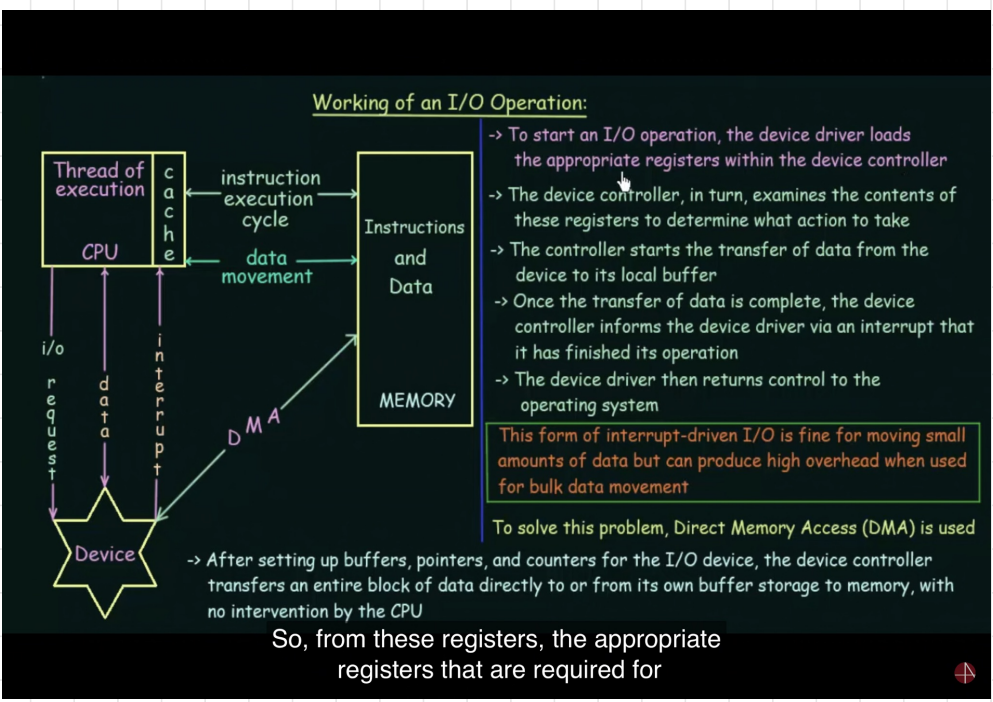
2) Interrupt: → The occurence of an event is usually signalled by an Interrupt from Hardware or Software.
→ Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by the way of the system bus.

3) System Call (Monitor call): → Software may trigger an interrupt by executing a special operation called System Call.

So, that Operating system is already residing or stored



which basically are secondary storage devices, are nonvolatile.

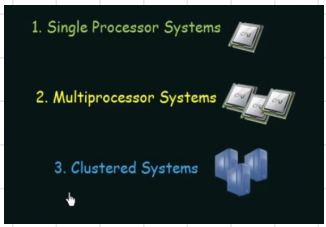


So, from these registers, the appropriate registers that are required for

Single Processor System

↳ Single Processor

↳ has multiple special purpose processor which device specific tasks



Multiprocessor System

↳ has 2 or more processors

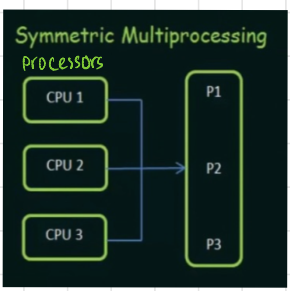
↳ they share clock/memory/devices/bus

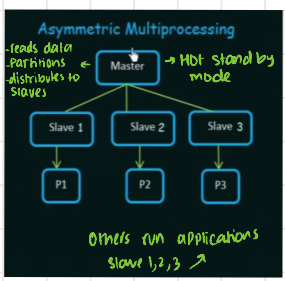
Adv

↳ increased throughput → amount of data transfered from 1 place to another → more processors

↳ economy of sales → as processors share resources

↳ increased reliability → if 1 processor fails other will work





↳ all cpus similar

↳ all processors acess same memory

↳ master and slave

↳ all processors acess different memory

↳ independent memory

Clustered Systems

↳ multiple processors

↳ composed of 2 or more individual systems

Adv

↳ high availability → if 1 system fails the others can take care of task

↳ can also be structured symmetrically or asymmetrically

OS TYPES

Multiprogramming

↳ if didn't exist then 1 job would take all CPU time until completion
↳ when us I/O devices CPU would be idle

↳ executes multiple jobs

↳ no user interaction with comp system

↳ processes are in RAM

↳ only one process executed at a time

↳ if I/O operation occurs only then other process executed

Timesharing/Multitasking

↳ CPU executi multiple jobs

↳ switches so quickly that user interaction with programs is possibl

↳ allows many users to share the comp simultaneously

↳ CPU scheduling

↳ Processe are in CPU

↳ multiple processes running concurrently and prempted after a fixed time slice

↳ increases CPU utilization

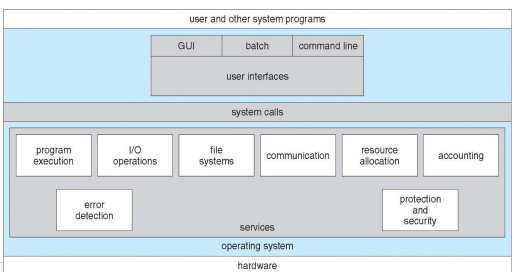
↳ dosent leave CPU idle

OPERATING SYSTEM SERVICES

CHP 2

One set of operating-system services provides functions that are helpful to the user:

- **User interface** - Almost all operating systems have a user interface (**UI**).
  - Varies between **Command-Line** (**CLI**), **Graphics User Interface** (**GUI**), **Batch**
- **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
- **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
- **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
- **Communications** – Processes may exchange information, on the same computer or between computers over a network
  - Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
  - May occur in the CPU and memory hardware, in I/O devices, in user program
  - For each type of error, OS should take the appropriate action
- **Resource allocation** - When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
  - Many types of resources - CPU cycles, main memory, file storage, I/O devices.
- **Accounting** - To keep track of which users use how much and what kinds of computer resources
- **Protection and security** - The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
  - **Protection** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts



CLI
↳ type
↳ interpreters are known as shells

GUI
↳ icons

IPC

IF PEARS PUC

SYSTEM CALLS

System calls

↳ Provide an interface to the services made available by an OS in kernel mode

↳ Made by programs to access certain resources

MODES A PROGRAM CAN EXECUTE

User mode (safer mode hence mostly used)

no direct access to

↳ memory / hardware / resources

↓

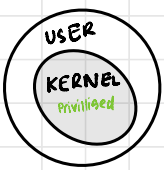
when needs access to Privisilled resources it makes a system call to switch from user to kernel mode

Kernel mode (doesn't happen in User mode)

direct access to (Privilege mode)

↳ memory / hardware / resources

if program crashes during execution whole system comes crashes

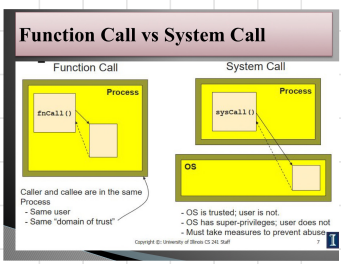


Pass Parameter in OS

↳ registers

↳ Pointers

↳ stack



TYPES OF SYSTEM CALLS

1. Process Control
- load
- execute
- create process
  - fork on Unix-like systems or
  - NtCreateProcess in the Windows
- terminate process
- get/set process attributes
- wait for time, wait event, signal event
- allocate free memory

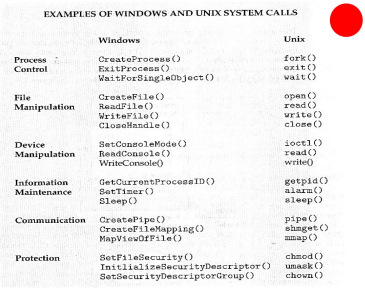
2. File management
- create file, delete file
- open, close
- read, write, reposition
- get/set file attributes

3. Device Management
- request device, release device
- read, write, reposition
- get/set device attributes
- logically attach or detach devices

4. Information Maintenance
- get/set time or date
- get/set system data
- get/set process, file, or device attributes

5. Communication
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote device

6. Security
- Protection
- Control access to resources
- Get and set permissions
- Allow and deny user access


EXAMPLES OF WINDOWS AND UNIX SYSTEM CALLS

	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

DIPS FC

FIPC

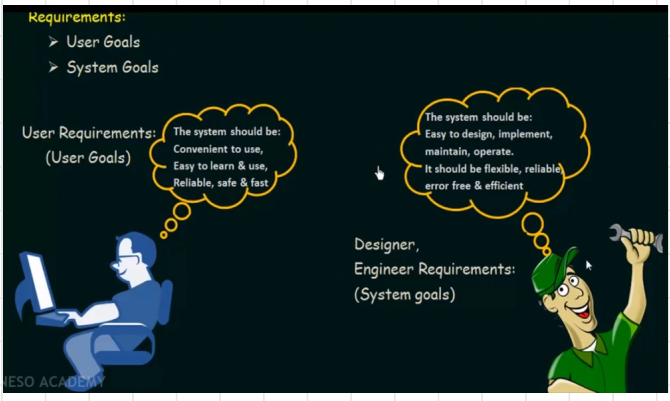
SYSTEM PROGRAMS

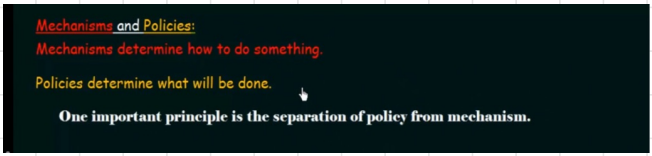
- **File management** - Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
  - Some ask the system for info - date, time, amount of available memory, disk space, number of users
  - Others provide detailed performance, logging, and debugging information
- **File modification**
  - Text editors to create and modify files
  - Special commands to search contents of files or perform transformations of the text
- **Programming-language support** - Compilers, assemblers, debuggers and interpreters sometimes provided
- **Program loading and execution**- debugging systems for higher-level and machine language
- **Communications** - Provide the mechanism for creating virtual connections among processes, users, and computer systems

FIPC

- **Background Services**
  - Launch at boot time
  - Provide facilities like disk checking, process scheduling, error logging, printing
  - Run in user context not kernel context
  - Known as **services**, **subsystems**, **daemons**
- **Application programs**
  - Run by users

OS DESIGN and IMPLEMENTATION





OS STRUCTURES

1) Simple structure

↳ if one program fails entire system crashes

↳ not well protected/structure

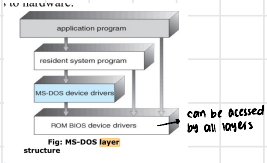


Fig: MS-DOS layer structure

2) Monolithic structure

↳ too many functions packed in one level → kernel part

↳ hence implementation/maintainence difficult

↳ too change one thing will have to touch entire kernel

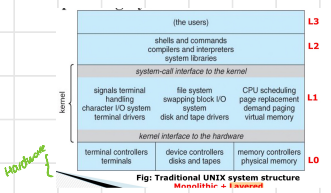


Fig: Traditional UNIX system structure

Monolithic + Layered

3) Layered structure

pros

↳ functionalities not packed into some layer

↳ easy to implement/debug

↳ if one layer problem then only have to look at that layer

↳ hardware protected by layers above/no direct access by the user

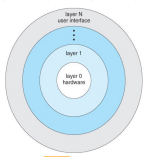
cons

↳ a layer can only use those below the layer

↳ not efficient

↳ when a layer wants to use services of below layer it will have to go down one by one

↳ may not be fast



4) Microkernels structure

pros

↳ micro kernel provides core functionalities

↳ other services are implemented as a user level program

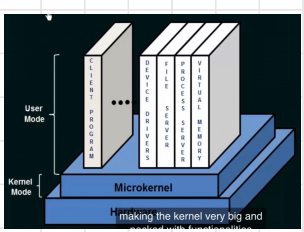
↳ communication b/w client programs and system programs as made through message passing

↳ can continue running mostly in user mode so less chances of system crash

cons

↳ performance decrease due to increase system overhead

↳ due to message passing



5) Module structure ↗ best

↳ uses OOP techiniques

↳ core kernel has core functionalities

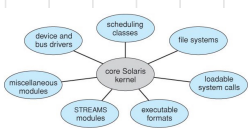
↳ other functionalities are in the form of modules

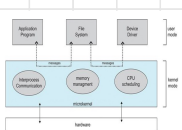
↳ modules loaded into kernel when required

↳ doesnt have to go through all layers like in layer approach

↳ each module can communicate with other module directly

↳ no need for message passing hence no performance decrease





CHP 3

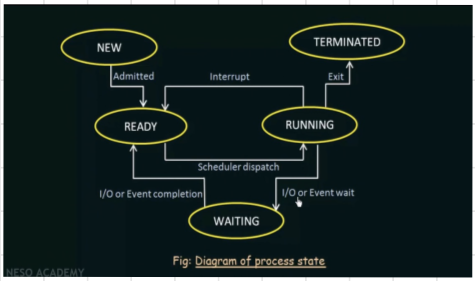


Fig: Diagram of process state

PCB

process state → running/waiting/ready...
process number → process ID
program counter → address of next instruction
registers
memory limits
list of open files
• • •

SNCR

Process Scheduling

- The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.
- The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.
- To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

Scheduling Queues

JOB QUEUE — As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.

READY QUEUE — The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.**

Process alternate b/w these 2 states

- **I/O–bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU–bound process** – spends more time doing computations; few very long CPU bursts

Context switch

- ↳ P_1 executing (current program)
- ↳ interrupt occurs
- ↳ system saves state of P_1 context (present in PCB of the process)
- ↳ CPU switches to P_2 and suspends P_1
- ↳ P_2 executes and finishes
- ↳ restore state of P_1
- ↳ resume execution of P_1

Pure overhead → cost (resources/time spent...) that is involved in doing something

↳ does no usefull work while switching

Kernels actions in context switch

- ↳ Saving the current Process state
- ↳ selecting the next process
- ↳ loading the next process state
  - ↳ restoring saved process
- ↳ Handing control to next process
  - ↳ resuming execution

fork():

↳ creates a seperate duplicate process

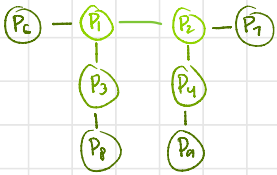
↳ Parent —its duplicate→ child

↳ ID is different

main(){

fork()
fork()
fork()
}

no of fork calls
$2^3 = 8$ processes created



exec():

↳ program in parameter will replace entire process

↳ ID remains same

Interrupt

cause OS to change CPU from current task and to run a kernel routine

CHP5

CPU SCHEDULER

↳ whenever CPU becomes idle

↳ OS selects a process in ready queue to execute

CPU scheduling

Dispatcher

↳ giving control to process selected by short term scheduler (CPU scheduler)

↳ time to stop one process and start another is dispatch latency

CPU scheduling circumstances

1. switch from running to waiting (I/O request)
2. switch from running to ready (interrup occurs)
3. switch from waiting to ready (completion of I/O)
4. when process terminates

choice given will CPU given to same process OR to some other process

no choice of scheduling as a new process must be selected

2 ways CPU SCHEDULING TAKES PLACE

Preemptive scheduling

↳ circumstance 2 and 3

↳ CPU can be taken away from a process before it has

↳ went into ready state

non preemptive scheduling/cooperative

↳ circumstance 1 and 4

↳ CPU will never be taken away from a process until and unless it has

↳ completed execution (terminated)

↳ went into waiting state

CONS

shared memory

↳ P_1 writing a sentence in shared memory

↳ P_2 came, P_1 preempted/halted

↳ CPU given to P_2

↳ P_2 when reads from shared memory

↳ P_2 reads inconsisted data as P_1 wasnt done writing

FCFS (First Come First serve)

- ↳ when process enters ready queue
  PCB linked to tail of the queue
- ↳ avg waiting time long
- ↳ non preemptive

turnaround time = Completion time - Arrival time

waiting time = turnaround time - Burst time

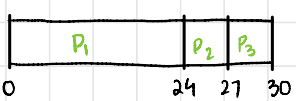
CONS

- ↳ dissasterous to allow one process CPU for extended time
- ↳ CONVOY EFFECT
  - ↳ higer burst time arrive before smaller
  - ↳ smaller process have to wait long time

Q)

process	Burst Time ms
P_1	24
P_2	3
P_3	3

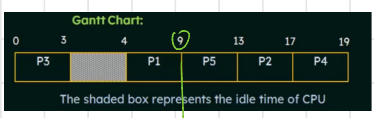
Gantt Chart



waiting time = 0 + 24 + 27 = 51 ms

avg waiting time = 51/3 = 17 ms

Process ID	Arrival Time	Burst Time
~~P1~~	~~4~~	~~5~~
P2	6	4
~~P3~~	~~0~~	~~3~~
P4	6	2
P5	5	4



Process ID	Completion Time	Turnaround Time	Waiting Time
P1	9	9 - 4 = 5	5 - 5 = 0
P2	17	17 - 6 = 11	11 - 4 = 7
P3	3	3 - 0 = 3	3 - 3 = 0
P4	19	19 - 6 = 13	13 - 2 = 11
P5	13	13 - 5 = 8	8 - 4 = 4

avg turnaround time = $\frac{5+11+3+13+8}{5}$ = 8 units

avg waiting time = $\frac{0+7+0+11+4}{5}$ = 4.4 units

SJF (shortest job first)

↳ if CPU burst are same FCFS is used to break tie

↳ preemptive or non preemptive

CONS

↳ no way to know length of next CPU burst

turnaround time = Completion time - Arrival time

waiting time = total waiting time - ms process executed - Arrival time

SOLUTION

↳ we can approximate

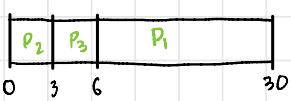
↳ we can predict using previous CPU burts

NON PREEMPIVE

Q)

process	Burst Time ms
P_1	24
P_2	3
P_3	3

Gantt Chart



waiting time = 0+3+6 = 9 ms (as no arrival time given assume 0)

avg waiting time = 9/3 = 3 ms

PREEMPTIVE

Process ID	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Gantt Chart:

0	1	5	10	17	26
P1	P2	P4	P1	P3	

P_1

waittime = (10-1-0) + (1-0-1) + (17-0-2) + (5-0-3)

avg = 26/4 = 6.5

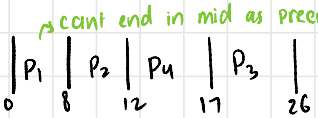
turnaround time = (17-0) + (5-1) + (26-2) + (10-3)

avg = 52/4 = 13

NON PREEMPTIVE

ve

can't end in mid as preemph



INTER PROCESS COMMUNICATION (IPC)

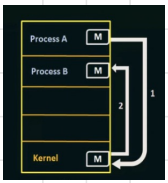
↳ Mechanisim for Processes executing concurrently to communicate and synchronise their actions

MESSAGE PASSING

↳ TWO OPERATIONS

↳ send message → fixed size message/variable

↳ receive message



In order to communicate using message passing

↳ establish a communication link b/w them

↳ Physical (shared memory/ hardware bus)

↳ logical (logical properties)

↳ exchange messages via send/receive

Priority Scheduling

↳ CPU allocated by priority

CON: Starvation → low priority may never execute

SOLUTION: Aging → increase priority as time progresses

turn around time: Completion – Arrival time

Waiting time: Total waiting time – ms process executed – Arrival time

↓ Preemptive

Waiting time: turn around time – burst time

↓ non preemptive

PREMPTIVE

Process ID	Arrival Time	Burst Time	Priority
P1	0	~~11~~ 9	2
P2	5	28	0
P3	12	2	3
P4	2	~~10~~ 7	1
P5	9	16	4

olution: Gantt Chart:

0	2	5	33	40	49	51	67
P1	P4	P2	P4	P1	P3	P5	

Waiting time: (40-2-0) + (5-0-5) + (49-0-12) + (33-3-2) + (51-0-9)

avg = 29/5

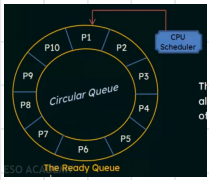
NON PREMPTIVE

→ best for time sharing

Round Robin (RR) Scheduling

↳ small time slices for each process (time quantum)

↳ FCFS but has premption



FIFO in ready queue

Tail → new proceses

Head → nex process to get CPU

2 Possibilities

↳ burst time < time quantum

↳ CPU released

↳ CPU scheduler gets next process in ready queue

↳ burst time > time quantum

↳ timer goess off

↳ interrupt to OS

↳ context switch, process moved to tail of ready queue

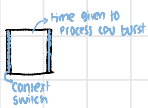
↳ CPU scheduler gets next process in ready queue

CONS

time quantum

↳ too small : too many context switches

↳ too big : starve for long time



Process ID	Burst Time
P1	24
P2	3
P3	3

Gantt Chart:

0	4	7	10	14	18	22	26	30
P1	P2	P3	P1	P1	P1	P1	P1	

Method 1

Turn Around time = Completion time - Arrival time

Waiting time = Turn Around time - Burst time

Process ID	Completion Time	Turnaround Time	Waiting Time
P1	30	30 - 0 = 30	30 - 24 = 6
P2	7	7 - 0 = 7	7 - 3 = 4
P3	10	10 - 0 = 10	10 - 3 = 7

Average Turn Around time
= (30 + 7 + 10) / 3
= 47 / 3 = 15.66 ms

Average waiting time
= (6 + 4 + 7) / 3
= 17 / 3 = 5.66 ms

Method 2

Waiting time = Last Start Time - Arrival Time - (Preemption x Time Quantum)

Process ID	Waiting Time
P1	26 - 0 - (5x4) = 6
P2	4 - 0 - (0x4) = 4
P3	7 - 0 - (0x4) = 7

Average waiting time
= (6 + 4 + 7) / 3
= 17 / 3 = 5.66 ms

ishma hafeez
notes

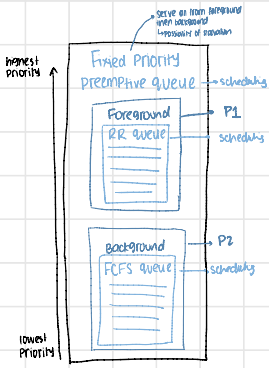
NO TALKING CLASS SAY BAHIR

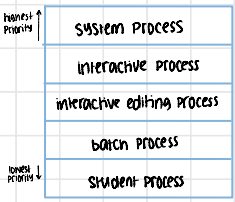
Multi-level Queue Scheduling

Scheduling algorithms for situations in which processes are classified in different groups

E.g

Foreground (interactive)	Background (batch)
↳ need to be quick	↳ doesn't need to be as fast
↓	↳ FCFS
Priority will be higher	
↳ Round Robin	





↳ Partions ready queue into several seperate queues

↳ Scheduling takes place within and among these queues

↳ Each process permamentaly assigned to 1 queue

based on memory size, Piority and process type

↳ Each queue has it's own scheduling algorithm (FCFS, SJF, RR...)

Multi-level Feedback Queue Scheduling

↳ allows processes to move between queues

↳ seperate processes according to their CPU bursts

↳ If a process uses too much CPU time

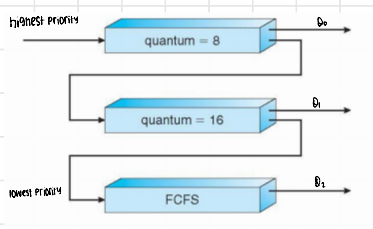
then moved to lower priority queue (so everybody gets a chance)

↳ I/O and interactive processes are in higher Priority queues

as they need quick responses

↳ A process in lower priority queue for too long → aging

may be moved to higher priority queue (prevents starvation)



- 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

Parameters

↳ no. of queues

↳ scheduling algo for each queue

↳ method to upgrade process to higher priority

↳ method to demote process to lower priority

↳ method to determine queue when process needs service

↳ demote/upgrade

Multi-Processor Scheduling

NOTES MISSING

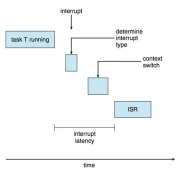
- CPU scheduling more complex when multiple CPUs are available
- **Homogeneous processors** within a multiprocessor
- **Asymmetric multiprocessing** – only one processor accesses the system data structures, alleviating the need for data sharing
- **Symmetric multiprocessing (SMP)** – each processor is self-scheduling, all processes in common ready queue, or each has its own private queue of ready processes
- **Processor affinity** – process has affinity for processor on which it is currently running

Understanding

- **soft affinity:** When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so known as **soft affinity**.
- **hard affinity:** allowing a process to specify a subset of processors on which it may run.

Real-Time CPU Scheduling

- **Soft real-time systems** – no guarantee as to when critical real-time process will be scheduled
- **Hard real-time systems** – task must be serviced by its deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routine that services interrupt
  2. Dispatch latency – time for schedule to take current process off CPU and switch to another



Multiple-Processor Scheduling – Load Balancing

- If SMP, need to keep all CPUs loaded for efficiency
- **Load balancing** attempts to keep workload evenly distributed
- **Push migration** – periodic task checks load on each processor, and if found pushes task from overloaded CPU to other CPUs
- **Pull migration** – idle processors pulls waiting task from busy processor

CONFUSED

LECTURE 19,20,21

Process Concept

↳ Process are executed programs that have

↳ Resource Ownership — unit → process/task

↳ Process includes virtual space (to hold process image)

↳ OS prevents unwanted interference b/w processes

↳ Scheduling / Execution

- Process follows an execution path that may be interleaved with other processes
- Process has an execution state (Running, Ready, etc.) and a dispatching priority and is scheduled and dispatched by the operating system

- Today, the unit of dispatching is referred to as a **thread** or **lightweight process**

MOTIVATION

- Most modern applications are multithreaded
- Threads run within application (process)
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Thread control block

↳ information associated with each thread

↳ Program counter ↳ CPU registers

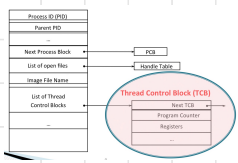
↳ CPU scheduling info ↳ Pending I/O info

Process control block (PCB)

↳ information associated with each process

↳ Memory management info

↳ Accounting info



Thread Operations

↳ Spawn : a thread within a process may spawn another thread

↳ provides new thread instruction pointer, arguments, register, stack

↳ Block : a thread needs to wait for an event

↳ saves its user registers, program counter, stack pointers

↳ Unblock: when the event for which the block occurs

↳ Finish: when thread completes

↳ its register context and stacks are dellocated

Thread States

↳ running

↳ ready

↳ blocked

Multithreaded Applications

- An application that creates photo thumbnails from a collection of images may use a separate thread to generate a thumbnail from each separate image.
- A web browser might have one thread display images or text while another thread retrieves data from the network.
- A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background.

Linux Kernel is also multithreaded

Single vs Multithreaded Webserver

- How a web-server (as a single process) increase its response time?

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead?

Single vs Multithreaded Webserver

- Assume: 1000~ request per second
  - Performance of single threaded server
  - Working of a multi-threaded server.
  - What benefits do we get using a multithreaded server?

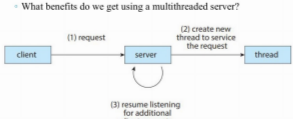


Figure 4.2 Multithreaded server architecture.

Process

↳ A program in execution

↳ can have 1 or more threads

Threads

↳ unit of execution within a process

↳ It compromises of

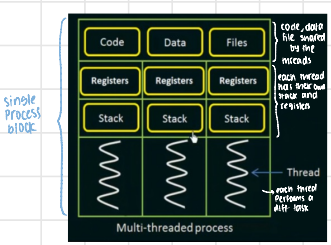
1. Thread ID
2. Program counter
3. Register set
4. Stack

↳ shares code section, data section, os resources with other threads of same processes

S.N.	Process	Thread
1.	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2.	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3.	In multiple processing environments each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5.	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6.	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

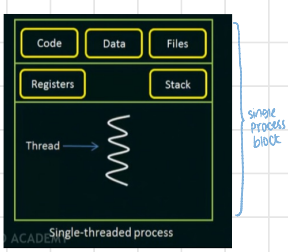
Multi Thread

↳ Performs multiple tasks at a time



Single Thread

↳ Performs one task at a time



MULTI-THREADING BENEFITS

↳ Responsiveness: may continue execution if process is blocked/performing lengthy operation
dedicated threads for handling user events

↳ Resource Sharing: threads share memory and recources of a process → allows diff threads activity within same address space
easier than shared memory/message passing

↳ Economy: cheaper than process creation as threads share recources
allocating memory/recources is costly
thread switching has lower overhead than context switching

↳ Scalibility: utilization of multiple cores for parallel execution
increases concurrency

Multicore Programing

PARALLELISM

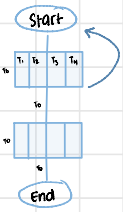
↳ act of managing multiple computations simultaneously

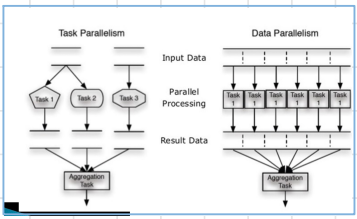
DATA PARALLELISM ← IMP → TASK PARALLISM

↳ focus on distributing data across diff parallel computing nodes

↳ focus on distributing threads across diff parallel computing nodes

Data Parallelism	Task Parallelism
Same operations are performed on different subsets of same data.	Different operations are performed on the same or different data.
Synchronous computation	Asynchronous computation
Speedup is more as there is only one execution thread operating on all sets of data.	Speedup is less as each processor will execute a different thread or process on the same or different set of data.
Amount of parallelization is proportional to the input data size.	Amount of parallelization is proportional to the number of independent tasks to be performed
Designed for optimum load balance on multi processor system.	Load balancing depends on the availability of the hardware and scheduling algorithms like static and dynamic scheduling.





AMDAHL'S LAW

↳ speed up in latency of a task execution

$$\text{Speed up} \leq \frac{1}{S + \frac{1-S}{N}}$$

S → serial, N → no of cores, (1−S) → parallel

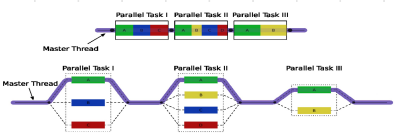
B) $S = 25\%$, $N = 2$

$$\text{Speed up} \leq \frac{1}{0.25 + \frac{0.75}{2}} = 2.28$$

CONCURRENCY

↳ act of managing multiple computations at the same time but not simultaneously → control is switched

Fork-Join model



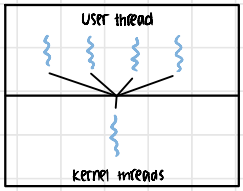

```
solve(problem):
    if problem is small enough:
        solve problem directly (sequential algorithm)
    else:
        for part in subdivide(problem)
            fork subtask to solve part
        join all subtasks spawned in previous loop
        combine results from subtasks
```

- Multicore systems putting pressure on programmers, challenges include
  - Dividing activities
    - What tasks can be separated to run on different processors
  - Balance
    - Balance work on all processors
  - Data splitting
    - Separate data to run with the tasks
  - Data dependency
    - Watch for dependences between tasks
  - Testing and debugging
    - Harder!!!!

MultiThreading → multiple threads at the same time

establish relationship between user and kernel thread

Many to one



PROS

↳ multiple user threads mapped to 1 kernel thread

↳ thread management is done by threaded library → so efficient

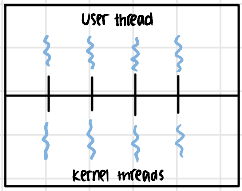
CONS

↳ if 1 thread makes a blocking system call entire process will be blocked

↳ only 1 thread can acess kernel at a time.

↳ multiple threads are unable to run in parallel on multiprocessors

One to one



PROS

↳ maps each user thread to kernel thread

↳ more concourrency → as allows another thread to run when a thread makes a blocking system call

↳ allows multiple threads to run parallel on multi processors

CONS

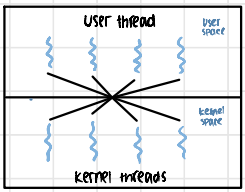
↳ each user thread requires a corresponding kernel thread

↳ overhead of creating kernel threads burdens performance

restricts no. of threads supported by the system

↳ e.g ↳ 4 core processor
↳ 5 threads
then only 4 threads work parallel
hence no of threds restricted

Many to Many



PROS

↳ kernel threads are ≤ to user threads

↳ more concorcuncy

↳ no limit to creating user threads

↳ kernel threads can run parallel on a multi processor

Hyper Threading / SMT (Simultaneous Multi Threading)

↳ more than 1 multithreading going on in the same system

e.g ↳ 4 core → virtually/logically divided into multiple processors
↳ 8 threads support

TYPES OF THREADS

user threads

↳ implemented by user

↳ requires no hardware support

↳ easy to implement

↳ if one thread blocked, entire process blocked

kernel threads

↳ implemented by kernel

↳ requires hardware support

↳ difficult to implement

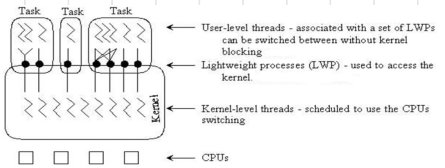
↳ if one thread blocked, another thread continues execution

POSIX PThreads

↳ Win 32 Threads

↳ Java Threads

- POSIX Pthreads
  - May be provided either as user-level or kernel-level
  - A POSIX standard (IEEE 1003.1c) API for thread creatio and synchronization
  - API specifies behavior of the thread library, implementation is up to development of the library
- Win32
  - Kernel-level library on Windows system
- Java
  - Java threads are managed by the JVM
  - Typically implemented using the threads model provided by underlying OS



- Linux kernel has processes and lightweight processes (LWP).
- Difference: LWPs share same address space and other resources like open files etc. As some resources are shared so these processes are considered to be light weight as compared to other normal processes and hence the name light weight processes.
- Therefore, effectively we can say that threads and light weight processes are same.
- Thread is the term used at the user level while LWP is a term used at kernel level.

Threaded Libraries

$$SUM = \sum_{i=1}^{N} i$$

APPLICATION → user function() → wrapper arm libraries

LIBRARY

KERNEL

Asynchronus Threading

↳ once parent creates child thread, parent resumes execution

↳ they execute concurrently and independently of one another

↳ Threads are independent → no dependency

↳ little data sharing

Synchronus Threading

↳ Once parent creates 1 or more children

↳ it waits all child to terminate before it resumes

↳ Threads by parent work cocurrently

↳ significant data sharing among threads

Race condition

↳ when several processes manipulate/acess the same data concurrently and outcome depends on order in which acess take place

results in inconsistency

SOLUTION

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- All solutions below based on idea of locking
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
  - Atomic = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Process Synchronization

The orderly execution of cooperating process that share an address space

Producer and consumer run concurrently using

BUFFER

unbounded buffer

↳ has a size on buffer

↳ consumer waits for new items

↳ producer can always produce new items

bounded buffer

↳ fixed buffer size

↳ consumer must wait if buffer empty

↳ producer must wait till buffer full

↳ use counter variable (++ produced, -- consumed)

Two approaches depending on if kernel is preemptive or non-preemptive

- Preemptive – allows preemption of process when running in kernel mode
- Non-preemptive – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

Critical section Problem

↳ a system with n processes

↳ each process has a segment of code called critical section

↳ where process may be changing common variables
updating table
writing a file

Three requirements

↳ mutual exclusion:

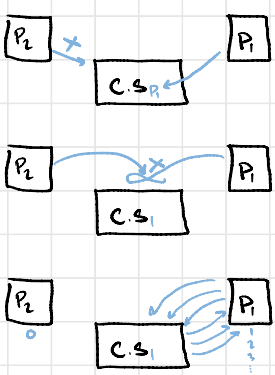
If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

↳ Progress:

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.

↳ bounded waiting:

There exists a bound, or limit, 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.



Peterson's solution → humble

↳ software solution to the critical section problem

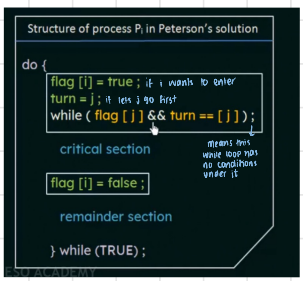
↳ 2 processes that alternate execution → b/w critical section, remainder section

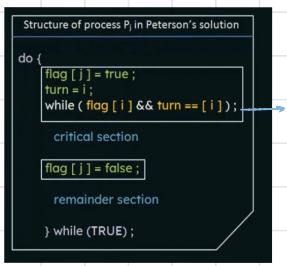
↳ the 2 processes share 2 variables

↳ int turn ; → indicates whose turn to enter critical section

↳ boolean flag[2]; → indicates if process ready to enter critical section

↳ satisfies all 3 critical section requirements





→ satisfies mutual exclusion

Test and set Lock

↳ hardware solution to the critical section problem

↳ processes share shared lock variable = 0 → unlocked, 1 → lock } helps satisfy mutual exclusion

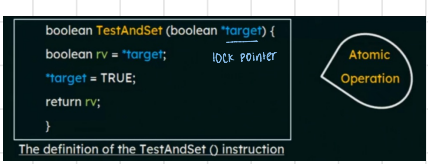
* lock = 0 in begining

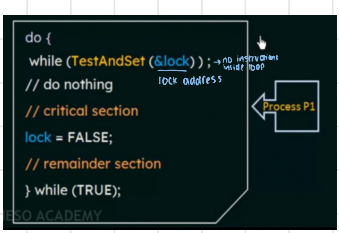
↳ if lock = 1, then wait till it becomes free

↳ if lock = 0, then execute critical section and lock = 1

↳ satisfies mutual exclusion

↳ does not satisfy bounded-waiting





when while = 0 → F
= 1 → T

Atomic Operation → hardware instruction

↳ single, uninterrupted operation

eg

TestAndSet() is run as a single operation which can not be interrupted

Bounded-waiting Mutual Exclusion with test_and_set

```
do {
   waiting[i] = true;
   key = true;
   while (waiting[i] && key)
      key = test_and_set(&lock);
   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);
```

Compare and Swap Instruction

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

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

   return temp;
}
```

- If two CAS instructions are executed simultaneously (each on a different core), they will be executed sequentially in some arbitrary order.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

   /* critical section */

   lock = 0;

   /* remainder section */
}
```

On Intel_x86 architectures

```
lock cmpxchg <destination operand>, <source operand>
```

Mutex locks/spin lock

↳ to protect critical section/prevent race conditions

↳ aquire() } both calls must be atomic

↳ release() }

↳ requires busy waiting

while process in critical section any other process that tries to enter critical section must loop continuously to check if condition true

↳ wastes CPU cycles → that some other process could use productively

-
```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
```
-
```
release() {
   available = true;
}
```
-
```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Pthreads Synchronization – Mutex Lock

#include <pthread.h>
Pthread_mutex_t mutex;

/* create the mutex lock */
Pthread_mutex_ init(&mutex,NULL);

1st Arg: Mutex initiliazer
2nd Arg: Attributes, NULL means no error checks will be performed

Pthreads Synchronization – Mutex Lock

- The mutex is acquired and released with the pthread_mutex_lock() and pthread_mutex_unlock() functions.
- If the mutex lock is unavailable when pthread_ mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_ unlock().

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;
```

```
void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);

    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++)
        ;

    printf("\n Job %d has finished\n", counter);

    pthread_mutex_unlock(&lock);

    return NULL;
}
```

```
int main(void)
{
    int i = 0;
    int error;

    if (pthread_mutex_init(&lock, NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    while (i < 2) {
        error = pthread_create(&(tid[i]),
                               NULL,
                               &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created :[%s]",
                   strerror(error));
        i++;
    }

    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization

Semaphore

↳ Software solution to the critical section problem

↳ technique to manage concurrent processes

↳ an int variable S (always +ve) shared b/w threads
(S → semaphore)

↳ S can only be accessed through 2 atomic operations

↳ wait() → to decrement
S<=0 → some process already in critical section so can't use shared resource → wait
S>0 → can use shared resource → don't wait

↳ signal() → to increment → tell other processes that shared resource is free to be used

```
Definition of wait ():
P (Semaphore S) {
    while (S <= 0)
    ; // no operation
    S-- ; → when S>0
}
```

```
Definition of signal ():
V (Semaphore S) {
    S++ ;
}
```

* S=1 in begining

TWO TYPES OF SEMAPHORES

Binary Semaphore

↳ S can only have 2 values
0 → some process already executing so not free to use shared resource
1 → free to use shared resource

↳ behave similarly to mutex locks (provide mutual exclusion)

↳ can implement a counting semaphore S as a binary semaphore

Counting Semaphore

↳ S can multiple values

↳ used to control acess to a resource that has multiple instances

↳ S = no. of instances of a resource

SEMAPHORE DISADVANTAGE

↳ requires **BUSY WAITING**

↳ while a process is in critical section any other process that tries to enter critical section must loop continously to check if condition true

↳ wastes CPU cycles → that some other process could use productively

* This type of semaphore also known a spinlock (as process 'spins' waiting for lock)

SOLUTION TO BUSY WAITING

↳ modify wait() and signal()

to block() → Places process (waiting state) in waiting queue

to wake up() → move process from waiting to ready queue

↳ instead of busy waiting, block itself

transfers control to CPU scheduler (which selects another process to execute)

hence no wastage of CPU time

PROBLEM

↳ **Deadlock**

↳ set of blocked processes, each holding a resource and waiting to aquire a resource, held by another process

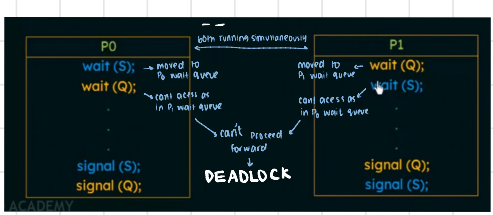
↳ **Starvation**

↳ when a process is postponed because it requires a resource to run, that is never allocated to this process

indivisibility

when 1 process modifies semaphore

no other process can simultaneously modify that semaphore value

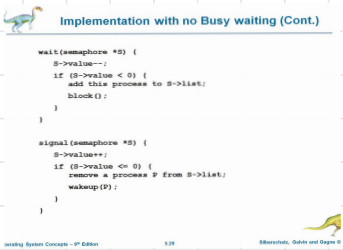


DEADLOCK VS STARVATION

↳ process stuck in circular waiting for the resources

↳ process waits for a resource indefinitely

deadlock implies starvation BUT starvation does not impy deadlock



POSIX - Semaphores

#include <semaphore.h>
Sem_t sem;
/* Create the semaphore and initialize it to 1 */
Sem_init(&sem, 0, 1);
The sem_init() function is passed three parameters:
1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

POSIX - Semaphores

/* acquire the semaphore */
Sem_wait(&sem);

/* critical section */

/* release the semaphore */
Sem_post(&sem);

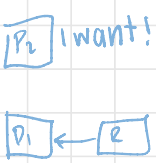
Priority Inversion

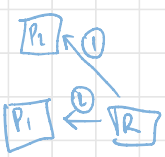
↳ scheduling problem when lower priority process holds a lock needed by a higher priority process

SOLUTION →

Priority Inheritance Protocol

↳ when a job blocks a higher priority job it ignores the current lower priority job executes higher priority job's critical section then releases lock and returns to the current lower priority job





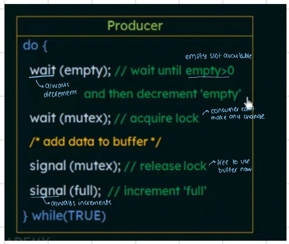
CLASSICAL PROBLEMS OF SYNCHRONIZATION

1. Bounded buffer Problem

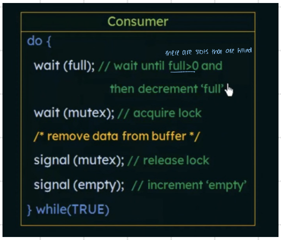
- shared buffer
- buffer of n slots, each capable of storing one unit of data
- Producer process
  - insert data in empty slot of buffer
  - not insert data when buffer is full
- consumer process
  - remove data from filled slot of buffer
  - not remove data when buffer is empty

Solution using semaphores

- n buffers → each can hold one item
- m(mutex) → binary semaphore
  - aquire lock
  - release lock
- empty → counting semaphore → initial value = no of slots in buffer since initially all slots are empty
- full → counting semaphore → initial value = 0



```
Producer
do {
    wait (empty); // wait until empty>0
                     and then decrement 'empty'
    wait (mutex); // acquire lock
    /* add data to buffer */
    signal (mutex); // release lock
    signal (full);  // increment 'full'
} while(TRUE)
```

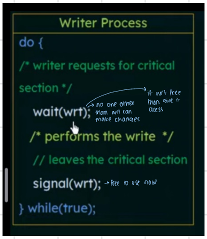
```
Consumer
do {
    wait (full); // wait until full>0 and
                    then decrement 'full'
    wait (mutex); // acquire lock
    /* remove data from buffer */
    signal (mutex); // release lock
    signal (empty);  // increment 'empty'
} while(TRUE)
```


2. The Readers Writers Problem

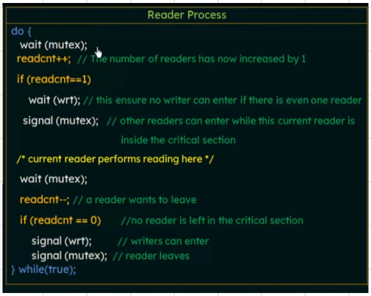
- shared database
- Readers → only want to read
- Writers → want to read and write
- if a writer and a writer/reader acess database simultaneously → PROBLEM
- give exclusive acess to database → SOLUTION

Solution using semaphores

- mutex → a semaphore → initialize to 1
- wrt → a semaphore → initialized to 1
- readcount → counts how many readers reading same data → initialized to 0
  - whenever modified wait(mutex) to be called



```
Writer Process
do {
/* writer requests for critical
section */
    wait(wrt);
  /* performs the write */
   // leaves the critical section
    signal(wrt);
} while(true);
```

```
Reader Process
do {
  wait (mutex);
  readcnt++;  // The number of readers has now increased by 1
  if (readcnt==1)
     wait (wrt); // this ensure no writer can enter if there is even one reader
   signal (mutex);   // other readers can enter while this current reader is
                        inside the critical section
  /* current reader performs reading here */
  wait (mutex);
  readcnt--; // a reader wants to leave
  if (readcnt == 0)      //no reader is left in the critical section
     signal (wrt);        // writers can enter
     signal (mutex);  // reader leaves
} while(true);
```

→ used in OS resource allocation → Processes → resources

3. The Dinning-Philosophers Problem

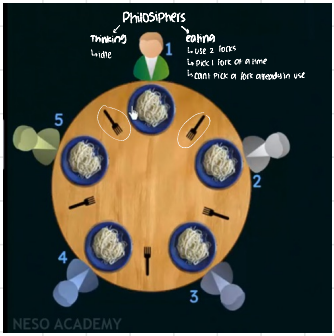
↳ 5 Philosophers, 5 forks

↳ Eating → use 2 forks for eating; Pick 1 fork at a time; can't pick if fork already in use } can't eat unless has 2 forks

↳ Thinking → idle

↳ when a Philosophers eat he uses 2 forks

↳ NO 2 adjacent Philosophers try to eat at the same time (as forks are limited) → SOLUTION



SOLUTION USING SEMAPHORES

↳ fork[5] (as 5 forks) → a binary semaphore array (an element initialized to 1; 1 → free, 0 → in use)

↳ grab fork → wait() → used on 2 forks

↳ release fork → signal()

The structure of philosopher i

```
do {
wait (chopstick [i] ); first chopstick
wait(chopstick [ (i + 1) % 5] ); second chopstick
                                 (its adjacent chopstick)
....
// eat
signal(chopstick [i]);
signal(chopstick [(i + 1) % 5]);
// think
}while (TRUE);
```

can lead to DEADLOCK

all 5 hungry and grab left fork

all elements of fork = 0

when grabbing right fork, they will all be delayed forever

solutions to DEADLOCK

1. allow at most 4 philosophers (no of forks will remain 5)
2. allow to pick forks ONLY if both forks available
3. use asymmetric solution
   - ↳ an odd P first picks right fork then left fork
   - ↳ an even P first picks left fork then right fork

deadlock implies starvation

BUT

starvation does not imply deadlock