Multilevel Queue:

With priority scheduling, have separate queues for each priority
High priority tasks are completed first.

P=0 | T_0 | T_1 | T_2 | T_3 | T_4 |

P=1 | T_5 | T_6 | T_7 |

P=2 | T_8 | T_9 | T_{10} | T_{11} |

⋮

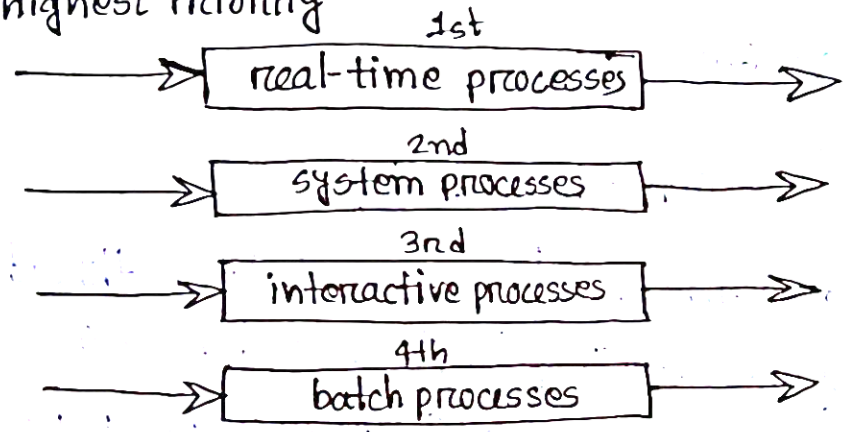
P=n | T_x | T_Y | T_z |

⇒ Priority = 0 ⇒ ready queue ⇒ RR
⇒ Priority = 1 ⇒ FCFS

CPU selects processes from the highes priority queue first always. and if the highest priority queue is empty; it moves to lower priority queue.

☐ Priority based-on process ~~time~~ type.

Highest Priority



Lowest P

Real time ⇒ live; less time gap, less latency
ex: pacemaker

system ⇒ the process which runs in the backend to manage hardware
ex: soundcard

interactive ⇒ keyboard, mail etc
has to type in keyboard to have text on screen; interaction

batch ⇒ most time consuming process
ex: backup

P ⇒ lowest ⇒ time ↑
P ⇒ highest ⇒ time ↓

upgrade a process => lower priority process to higher priority process

Feedback

MLFQ

aging => lower priority is move forward gradually

=> process can move between various queues

MLFQ rules: parameters

=> no of queues: levels of priority available

=> scheduling algorithm: ~~for~~ each queue uses a specific method

=> ~~ment~~ Upgrade Rules: When a process moves to a higher priority queue

=> Demotion Rules: When a process moves to a lower priority queue

=> Queue entry rules: Decides the queue for new or returning processes

Example:

Q_0 => RR ; q = 8ms

Q_1 => RR ; q = 16ms

Q_2 => FCFS

Thread Scheduling

User level thread	Kernel level thread
implemented by user level library	implemented by OS
OS doesn't recognize ULT	KLT are recognized by OS

To run process we need kernel; user-level cannot work directly it has to use kernel by LWP.

To access kernel, user go through LWP (light weight process)

To request kernel-level from user level use LWP method (PSC)

Kernel level can directly manupulate (SCS)

PTHREAD_SCOPE_SYSTEM ⇒ SCS

Linux & macOS only use PTHREAD_SCOPE_SYSTEM / SCS

Windows have both PCS & SCS

P_S_P ⇒ PSC

P_S_S ⇒ SCS.

Multiple-Processor scheduling

For multiprocessing there are 4 architectures:

① Muticore CPUs : one CPU multiple core

quad-core ⇒ has logical CPU sometimes

⇒ In 4 core CPU, there are 2 logical CPUs ; hardware level thread

② Multithread core : single core; contain 2 hardware-level thread that will work as logical core

③ NUMA ⇒ make connections through networking/bus connection of core of a system to another

④ Heterogeneous ⇒ ~~uses CPU & GPU~~ consists multiple CPUs

SMP

symmetric multiprocessor : each processor have same priority

ⓐ All thread in a common queue

ⓑ may have own private queue of thread

Why need scheduling algorithm? To upgrade performance to ensure that processes execute effitiently and have reduced WT.

Multicore Processor

memory stall ⇒ the time it takes to delete a file; CPU does not process during this period.
⇒ the time CPU remains idle

MMS

If one thread has memory stall ; switch to another thread

Each core has more than 1 hardware threads ⇒ short question
⇒ Minimum 2 thread

Logically it has 8 cores but
phycically 4 cores

hyperthreading ⇒ when physical core has multiple cores
Intel

Two levels of scheduling ⇒ (i) software thread } short question
(ii) hardware thread }

Load balancing

It evenly distributes workload across CPUs — Pull vs Push

Push Migration: A periodic task checks CPU loads.
Moves task from overloaded CPU to less loaded CPU

Pull Migration: Idle CPU actively pull tasks from busy CPU

Process Affinity

A Relationship between processors ; Ex: course prerequisite

① Soft affinity ⇒ The OS tries to keep a thread on the same processor but doesn't guarantee it.

② Hard affinity ⇒ A process specifies the exact processors it can run on; cannot work on new thread without completing a particular thread

Impact of Load balancing:

Moving thread to another processor for load balancing can result in cache loss and also reduce performance

⇒ It uses on server-site NUMA
⇒ limitation: slow access
⇒ assign memory closes to the CPU the thred is running on

Real-Time CPU Scheduling

Hard ⇒ task must be serviced by its deadline

Soft ⇒ Critical tasks have the highest priority
No guarantee of when they will be scheduled, but they are prioritized.

Event Latency

⇒ Request করা থেকে answer/response আসার time ⇒ Latency

Diagram
① Interrupt L: hardware level interrupt
Time from interrupt arrival to starting its service routine
② Dispatch L: Time for the scheduler to switch from one process to another.

periodic ⇒ task that require CPU time at regular, fixed intervals
constant interval has:
p = period
d = deadline
t = time
Rate of periodic task is 1/p

Algorithm Evaluation

1. Deterministic modeling
   will check which algorithm is better by comparing ~~WT~~ waiting time

Little's Formula: ⇒ short question

n = length of queue ; n/l ; how many processes in queue
W = avg. WT in queue
λ = avg. arival rate in queue
short math from ⇒ $n = \lambda \times W$

Simulation

simulations have limited accuracy
Simulation is more accurate than queuing Models

Implementation

⇒ Expensive & time consuming

Lecture-8

To avoid Race condition synchronization is used

First produce then consume

Synchronization comes when we use something shared

Individuals like personal computer doesn't need synchronization

Inconsistancy will appear if there is no synchronization

Producer ⇒ counter++;

consumer ⇒ counter--

Producer: Produces data and adds it to the buffer; stops if the buffer is full

consumer: consume data from buffer; but waits if the buffer is empty.

Race condition

It needs multiple entity/thread

↓

2 or more

concurrent program / parallel compiting ⇒ appears Race condition

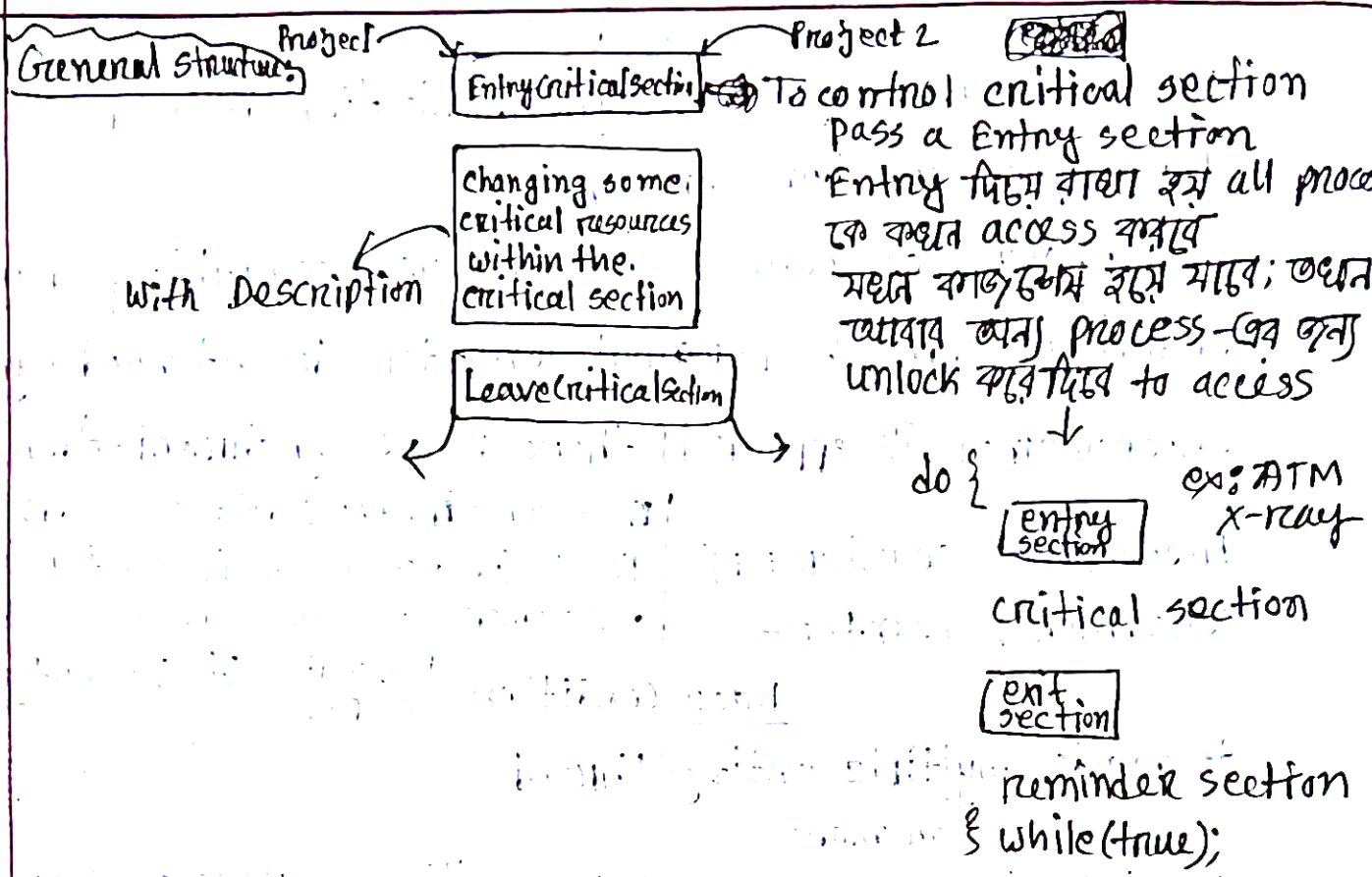
When two process generate id; they may be similar

System's process id can never be duplicate

- Mutual exclusion is a method by which we avoid race condition

Critical section problem: এক সময়ে কোনো processor দুইটা critical section access করতে পারে না; automatic lock হয়ে যায়।

- কোনো কাজ complete হওয়া পর্যন্ত critical section অন্য কাজ করতে allow করবে না। ⇒ critical section access করতে হলে must be atomic হতে হবে।



To solve critical section there are 3 methods.

① Mutual exclusion: দুইটা process কখনো simultaneously critical section a access করবে না। => mutual understanding

② Progess: if no one is in critical section, waiting prousses must get a chance to enter

Important
③ Bounded waiting: a process has a limit on how long it waits for its ~~turn~~ turn

Challange: ① Each process runs at non-zero speed
② No fixed relation between process speeds

Peterson Solution

Proposed a mechanism to avoid race condition (load & store)
load ⇒ read (load by read from 2nd memory to main memory)
store ⇒ store memory

limitations: no guarantee to work on mondern architecture
but give idea how it works

Two mechanism it follow ⇒ load & store

Has two parameter ⇒ turn & flag

turn define করে critical section-এ enter করার turn

flag decide করে কোন process queue থেকে আসছে

queue is considered as array flag[i].

turn = true/false

Three CS requirement ⇒ Explain peterson solution

no. of entity require for peterson solution

true/false এর abbriviation

Problem of Peterson ⇒ reorder operation ⇒ single thread
multi "

single thread-এ কোনো problem নাই

Multi " -এ

Thread 1 Thread 2

flag = true;

100 ⇒ output

⇒ Exam-ior MCA

Lecture-9

Synchronization hardware:

With the help from hardware, critical sections let one task run at a time. or, we can say that, [Hardware ensures that only one task runs at a time in critical section.]

Uniprocessors: Disabling interrupts ensures that the currently running code cannot be preempted

Limitations: ① Disabling interrupts in inefficient as it affects only one processor.
② It is not scalable for modern multiprocessor OS

3 form of Hardware support:

① Memory barriers
② Hardware instructions
③ Atomic variables

Memory Barriers

Memory models: Define how memory changes are seen by processors

These model can be either:

① Strongly ordered: Memory modifications are immediately visible to all processor.

② Weakly ordered: Memory modifications may not be immediatly visible.

A memory barrier is an instruction that makes sures memory changes are shared with all processors

Hardware Instructions

H.Is are special commands that let a processor change data without interruption. such as:

① Test-and-set instruction: checks and changes a value at the same time.

② compare & swap instruction: compares two values and swap them if they are same.

Atomic Variables

Atomic variables ⇒ Variables that can be updated without interruption

⇒ often used with instructions like compare & swap

⇒ Can handle basic data types (int, bool)

Example: The "increment()" operation safely increases a value without interruption.

Mutex Locks

OS use mutex locks to manage critical sections.

Mutex Locaks is simplest

It protects the critical section by:

① calling aquire() to lock

② calling release() to unlock.

> Spinlock: A type is a lock that continuasly checks if a lock is available (busy-waiting) until it can process

Lock status is boolean

true ⇒ lock is available

false ⇒ lock is in use

It is usally implemented via compare & swap.

It requires busy waiting - waste CPU cycle and for this spinlock is required.

Semaphore

A semaphore is a tool used to synchronize processes more effectively than mutex locks.

⇒ Semaphore S → int value

⇒ Accessed only through two atomic operations:

① wait() → decrease value and may block process

② signal() → increase value and may wake process

Types of semaphore:

① counting ⇒ Value can be any integer (+infinity to -infinity)

② Binary ⇒ Value is 0 or 1; works like a mutex lock

To ensure one task happens before another:

P1 ⇒ Performs S1, then signal(synch)

P2 ⇒ Perform wait(synch) before S2

```
P1:
  S1;
  signal(synch);
P2:
  wait(synch);
  S2;
```

Implementation:

① Initialization: Set the semaphore value; create a waiting queue

② wait() operation: (i) Decrease semaphore [reduces value by 1]
(ii) Check availability [if S->value<0; resource unavailable]
(iii) block the process [block()]

③ signal() operation: (i) increase semaphore [increase value by 1]
(ii) check waiting queue [if S->value <=0; waiting for resource]
(iii) wake up a process [remove from waiting queue, move it to ready queue]

④ Atomic Execution: Ensure wait() & signal() execute automatically to avoid race conditions.

Each semaphore has a waiting queue.

Each entry in waiting queue has two data items

① value

② pointer to next record in the list

Block: Add the progress to the waiting queue

wakeup: Move process from waiting queue to the ready queue

Problems with semaphores:

Incorrect Usage:

⇒ Using signal() before wait()

⇒ Using wait() multiple times without signal

⇒ missing wait() or signal() operations.

These mistakes can cause synchronization issues or deadlocks.

Lecture 10

Directions: P = T or P

if resource → process ⇒ assign

if process → resource ⇒ waiting

$(R_1) \rightarrow P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow (R_1)$ ⇒ deadlock

Each process has sequence of use: (Request allocation process)

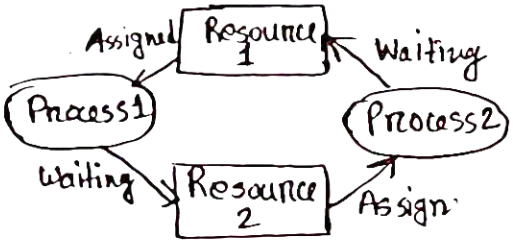
① Request (Ask OS for a resource)

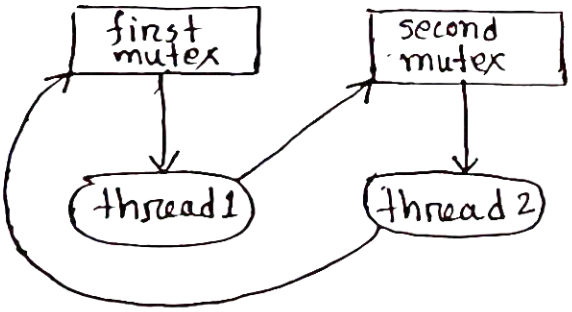
② Use (Perform tasks with it)

③ Release (Free it after use)

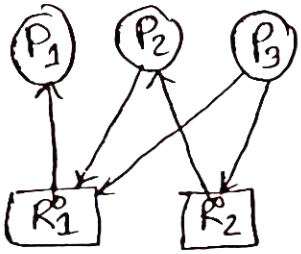
Each resource type R_i has W_i instances

Resource Allocation Graph: (RAG)





Deadlock



Deadlock free

$R_1 \rightarrow P_1$

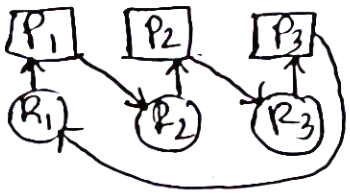
$R_2 \rightarrow P_2 \rightarrow R_1$

$P_2 \rightarrow R_1$

$P_3 \rightarrow R_1, R_2$

$P_2 \rightarrow R_1$

if no cycle occurs ⇒ No deadlock



holding ⇒ assign

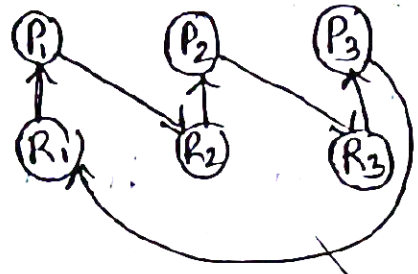
$(P_1) \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_1 \rightarrow (P_1)$ ⇒ Deadlock

Process involved: P_1, P_2, P_3

Resources " : R_1, R_2, R_3

Question: ①

RAG



$P_1 \rightarrow R_2 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_1 \rightarrow P_1$

② Answer: Yes, there is a dead lock.

The deadlock involves:

Processes: P_1, P_2, P_3

Resources: R_1, R_2, R_3

∴ All the processes and resources are involved

Deadlock Characterization

Deadlock occurs ~~when~~ if these four conditions (Coffman condition) hold at the same time:

① Mutual Exclusion: Only one process uses a resource at a time

② Hold & Wait: Processes hold resources while waiting for others.

③ No Preemtion: Resources ~~are~~ can't be taken ~~away~~ from a process unless it's done using them.

④ Circular Wait: Process are stuck in loop, waiting for the next.

If we able to violate any of these four then deadlock won't occur.

For singal instance ⇒ circle means deadlock but,

For multiple " ⇒ circle doesn't means deadlock

RAG

Assignment edge ⇒ Resource → Process (assign)

Requesting edge ⇒ Process → Resources (waiting)

• (dot) represents instance

∴ R_1 has 3 instances

~~claim edge ⇒ dashed line~~

~~waiting~~

Basic facts:

If graph has:-

- no cycles ⇒ no deadlock
- has cycles:-
  - ⇒ if one instance ⇒ deadlock confirmed
  - ⇒ if multiple " ⇒ possibility of deadlock; not ~~sure~~ confirmed

Methods of Handling Deadlocks:

1. Deadlock prevention
2. Deadlock avoidance
3. Detection and recovery: Allow deadlocks & then detect and recover them
4. Ignorance: Ignore & pretend that DL never occured

Deadlock Prevention: Violate one of four conditions of DL

ME: Applies only to non-sharable resources

H&W: Processes must request ~~to~~ all ~~users~~ resources at once or none

No preemption: Process releases all resources if they can't get more

Circular wait: Request resources in specific order.

Deadlock Avoidance:

- Processes declare maximum resource needs
- The system checks resource allocation to prevent deadlock.
- Based on available, used and maximum resources Resource allocation state is defined

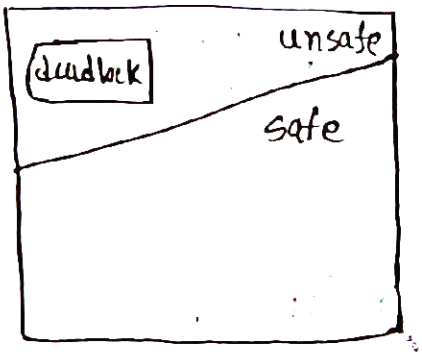
☐ system in safe state => No deadlocks
" in unsafe state => possibility of deadlock

Avoidance => ensures that a system will be never enter an unsafe state.

Safe state: Processes can finishish without occurring deadlocks

=> system is in safe state if processes can finish in a specific order
=> Each process waits for previous one to release resources
=> Finished processes release resources for others.

Safe, unsafe, Deadlock state:



has => holded
Max => needed
Free => available

Avoidance Algorithm:

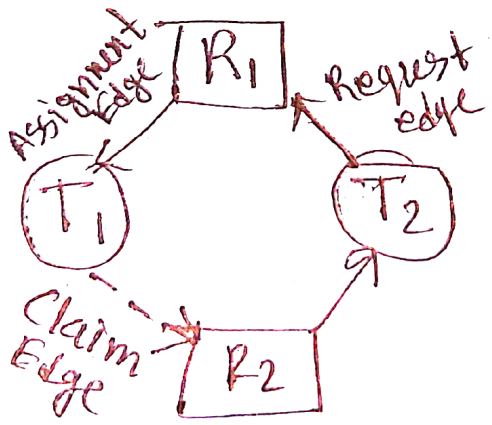
① Single instance of a resource type (Use RAG)
② Multiple " " a " " (Use Banker's algorithm)

RAG scheme:

Claim edge (- - - - ->): maybe resource is needed

request edge (——>): resource is must needed (P→R)

assign (R→P)



3part of hardware support:

① Memory barrier

② Hardware instruction

③ Atomic variables

Memory barrier:

An instruction that make sures that memory changes are shared to all processors.

Memory models define how memory changes are seen by processor

These models can be:

① Strongly ordered: Memory changes are visible to processor

② Weakly ordered: Memory changes may not immediatly visible to processors

Hardware Instruction: It is a speacial command that let a processor change data without interrupt. suchas

Test & check: checks and changes a value at the same time

Compare & swap: Compares two values and swap them if they are same

Atomic variables: Variables that can be updated without interruption

used in compare & swap

can handle basic data types

Example: increment();

Mutex lock

Used to manage critical sections
It protects critical sections by
① Calling aquire() => lock
② calling release() => unlock
if lock = true => lock available
for ~~not available~~ in use.

semaphore

A tool that is used to synchronize processes more effectively that mutex lock.
can be access only through 2 atomic operations
① wait() => decrese value & block process
② signal() => increase value & wake process

Types of semaphore:
① counting value can be any integer in range of (+∞ to -∞)
② binary (0 & 1); mutex lock

implementation:
initialization set the semaphore value by initializing a queue

wait(): ① decrease semaphore
② check availability
③ block process
signal(): ① increase semaphore

Atomic execution: