OPERATING SYSTEMS

Unit1_Unit2_Unit3: Revision Class #2

Nitin V Pujari
Faculty, Computer Science
Dean => IQAC, PES University

OPERATING SYSTEMS

Course Syllabus => Unit 1

1	Introduction: What Operating Systems Do, Computer-System Organization	1.1, 1.2	21.4
2	Computer-System Architecture, Operating-System Structure & Operations	1.3,1.4,1.5	
3	Kernel Data Structures, Computing Environments	1.10, 1.11	
4	Operating-System Services, Operating-System Design and Implementation	2.1, 2.6	
5	Process concept: Process in memory, Process State, Process Control Block, Context switch, Process Creation & Termination,	3.1 - 3.3	
6	CPU Scheduling - Preemptive and Non-Preemptive, Scheduling Criteria, FIFO Algrorithm	5.1-5.2	
7	Scheduling Algorithms:SJF, Round-Robin and Priority Scheduling	5.3	
8	Multi-Level Queue, Multi-Level Feedback Queue	5.3	
9	Multiprocessor and Real Time Scheduling	5.5, 5.6	
10	Case Study: Linux/ Windows Scheduling Policies.	5.7	
11	Inter Process Communication - Shared Memory, Messages	3.4	
12.	Named and unnamed pipes (+Review)	3.6.3	

OPERATING SYSTEMS

Course Syllabus => Unit 2

13	Introduction to Threads, types of threads, Multicore Programming, Multithreading Models	4.1 - 4.3	42.8
14	Thread creation, Thread Scheduling	5.4	
15	Pthreads and Windows Threads	4.4	
16	Mutual Exclusion and Synchronization: software approaches,	6.1-6.2	
17	principles of concurrency, hardware support	6.3-6.4	
18	Mutex Locks, Semaphores	6.5, 6.6	
19	Classic problems of Synchronization: Bounded-Buffer Problem, Readers-Writers problem	6.7-6.8	
20	Dining-Philosophers Problem	6.8	
21	Synchronization Examples: Synchronisation mechanisms provided by Linux/Windows/Pthreads.	6.9	
22	Deadlocks: principles of deadlock, Deadlock Characterization	7.1-7.3	
23	Deadlock Prevention, Deadlock example	7.4-7.5	
24	Deadlock Detection, Algorithm	7.6	

OPERATING SYSTEMS

Course Syllabus => Unit 3

25	Main Memory: Hardware and control structures, OS support, Address translation	8.1	64.2
26	Dynamic linking, Swapping	8.2	
27	Memory Allocation (Partitioning, relocation), Fragmentation	8.3	
28	Segmentation	8.4	
29	Paging: OS Support, TLBs, Address Translation	8.5	
30	Structure of the Page Table	8.6	
31	Design Alternatives - Inverted Page Tables, Bigger Pages	8.7-8.8	
32	Virtual Memory: Demand Paging, Copy-OnWrite	9.1-9.3	
33	Page replacement policy - LRU etc. (In comparison with FIFO and Optimal)	9.4	
34	Page Replacement (contd.), Frame allocation	9.4,9.5	
35	Thrashing	9.6	
36	Case Study: Linux/ Windows Memory Management	9.10	

Revision Class Outline => Student Queries

- How valid is the diagram which shows the stack and heap expanding towards each other given that they can be on different segments ?
- Sir, in the textbook for shortest job first/next, they have spoken about a formula to find the CPU burst time of the next process (exponential average), could you explain that ?
- Sir, can you explain how the CPU burst prediction is done in detail ?
- Comparative study on various CPU scheduling strategies => like advantages and disadvantages

OPERATING SYSTEMS

Revision Class Outline => Student Queries

- Revision on Real time scheduling sir
- Revision for Round Robin Scheduling
- Recursive Mutexes, Semaphores, Spinlocks
- Revision of Banker's safety algorithm for detecting deadlocks

OPERATING SYSTEMS

Revision Class Outline => Student Queries

- Page replacement algorithm in general
- Demand paging performance calculation
- Inverted page table

Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**

- Real Time computing (RTC), or reactive computing is the computer science term for hardware and software systems subject to a "real time constraint"

- The term **scheduling analysis** in real time computing includes the analysis and testing of the scheduler system and the algorithms used in real time applications.

- For critical operations, a real time system must be tested and verified for performance.

- **Revision on Real time scheduling sir**

- A realtime scheduling system is composed of the scheduler, clock and the processing hardware elements.

- In a real time system, a process or task has schedulability; tasks are accepted by a real time system and completed as specified by the task deadline depending on the characteristic of the scheduling algorithm

- **Revision on Real time scheduling sir**

- Modeling and evaluation of a real time scheduling system concern is on the analysis of the algorithm capability to meet a process deadline.

- Realtime programs must guarantee response within specified time constraints, often referred to as "deadlines"

- **Revision on Real time scheduling sir**

- A deadline is defined as the time required for a task to be processed.

- For example, in a realtime scheduling algorithm a deadline could be set to 15 microseconds.

- In a critical operation the task must be processed in the time specified by the deadline (i.e. 15 microseconds).

- **Revision on Real time scheduling sir**
  - The algorithms used in realtime scheduling analysis “can be classified as preemptive or non preemptive".

Three types of RTOS systems are

Hard Real Time :

In Hard RTOS, the deadline is handled very strictly which means that given task must start executing on specified scheduled time, and must be completed within the assigned time duration.

Example: Medical critical care system, Aircraft systems, etc

- **Revision on Real time scheduling sir**

Firm Real time:

These type of RTOS also need to follow the deadlines. However, missing a deadline may not have big impact but could cause undesired effects, like a huge reduction in quality of a product.

Example: Various types of Multimedia applications.

- **Revision on Real time scheduling sir**

Soft Real Time:

Soft Real time RTOS, accepts some delays by the Operating system. In this type of RTOS, there is a deadline assigned for a specific job, but a delay for a small amount of time is acceptable. So, deadlines are handled softly by this type of RTOS.

Example: Online Transaction system and Livestock price quotation System.

Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**
  - **Task** => A set of related tasks that are jointly able to provide some system functionality.
  - **Job** => A job is a small piece of work that can be assigned to a processor, and that may or may not require additional resources.
  - **Release time of a job** => It's a time of a job at which job becomes ready for execution.
  - **Execution time of a job** => It is time taken by job to finish its execution.
  - **Deadline of a job** => It's time by which a job should finish its execution.

- **Revision on Real time scheduling sir**

- **Maximum** => It is the allowable response time of a job is called its **relative deadline**.

- **Response time of a job** => It is a length of time from the release time of a job when the instant finishes.

- **Absolute deadline** => This is the relative deadline, which also includes its release time.

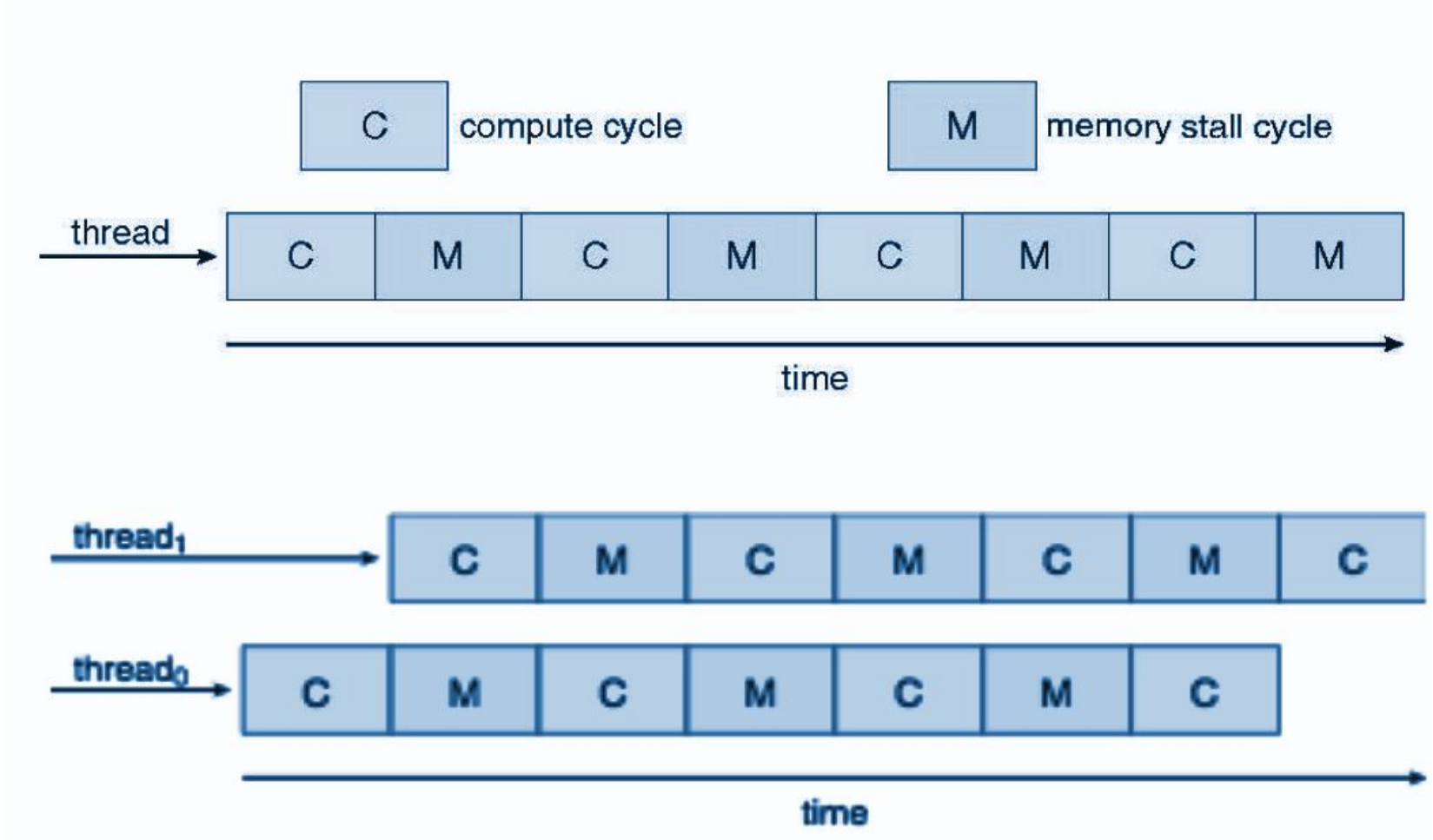
Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**

General-Purpose Operating System	Real-Time Operating System
It used for typically servers, desktop PCs and Laptops.	It is only applied to the embedded application.
Process-based Scheduling.	Time-based scheduling used like round-robin scheduling.
Interrupt latency is not considered as important as in RTOS.	Interrupt lag is minimal, which is measured in a few microseconds.
No priority inversion mechanism is present in the system.	The priority inversion mechanism is current. So it can not modify by the system.
Kernel's operation may or may not be preempted.	Kernel's operation can be preempted.
Priority inversion remain unnoticed	No predictability guarantees

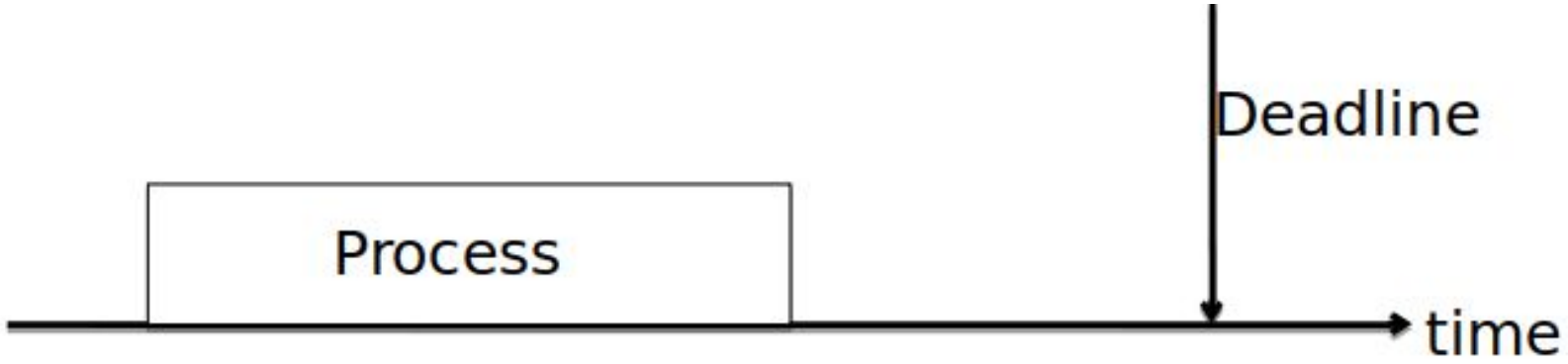
Revision Class Outline => Student Queries

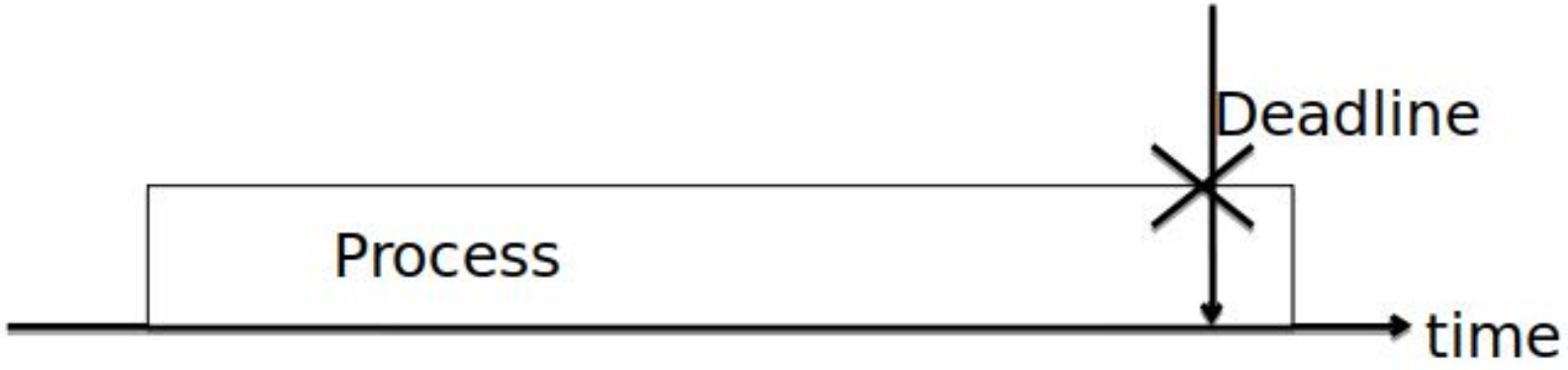
- **Revision on Real time scheduling sir**



Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**





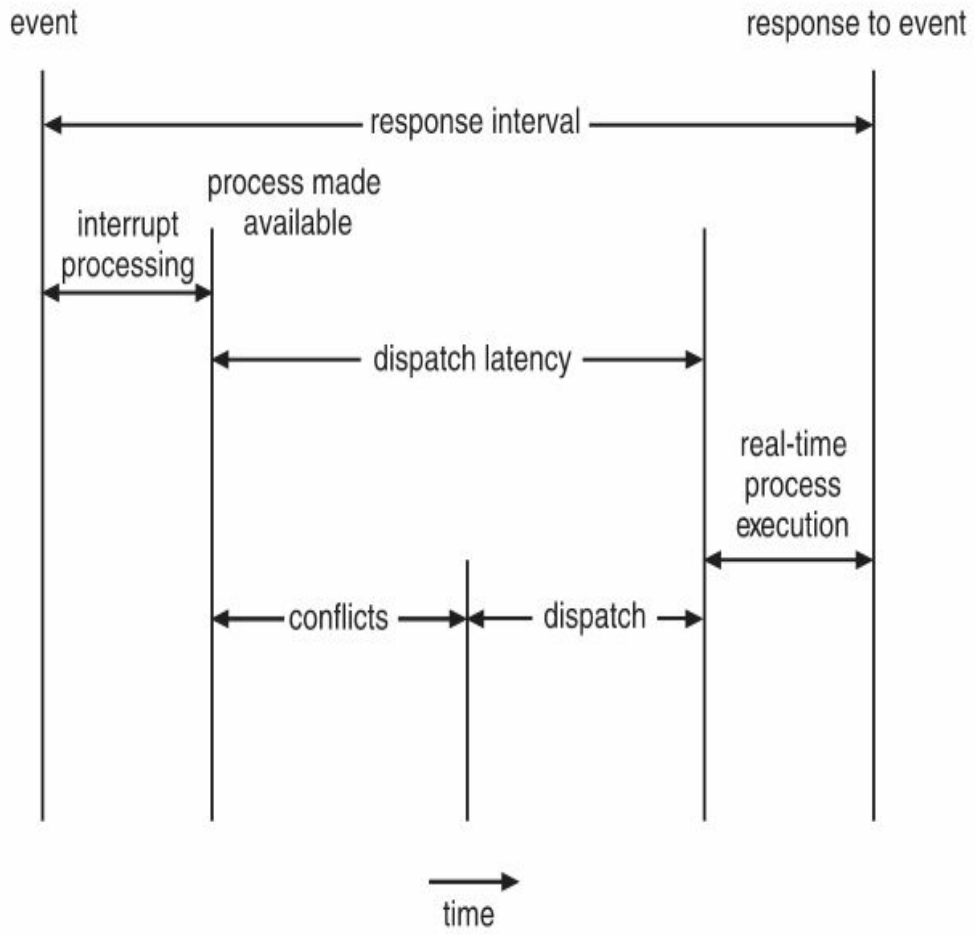
Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**

- Conflict phase of dispatch latency:

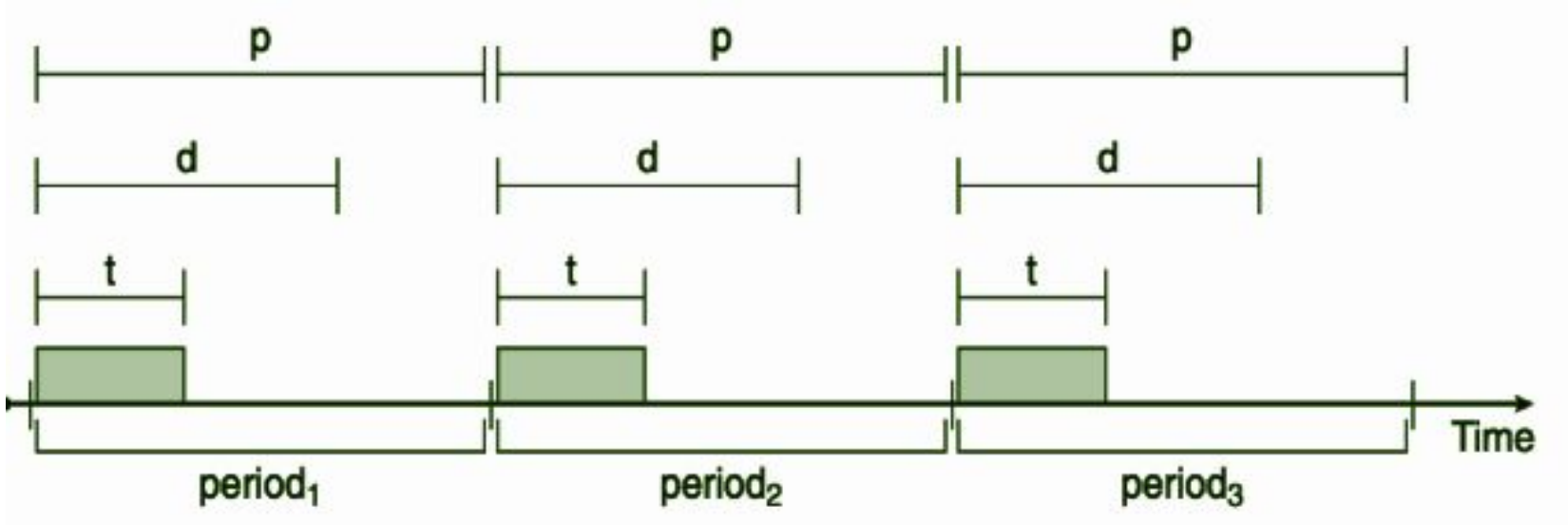
1. Preemption of any process running in kernel mode

2. Release by low-priority process of resources needed by high-priority processes



Revision Class Outline => Student Queries

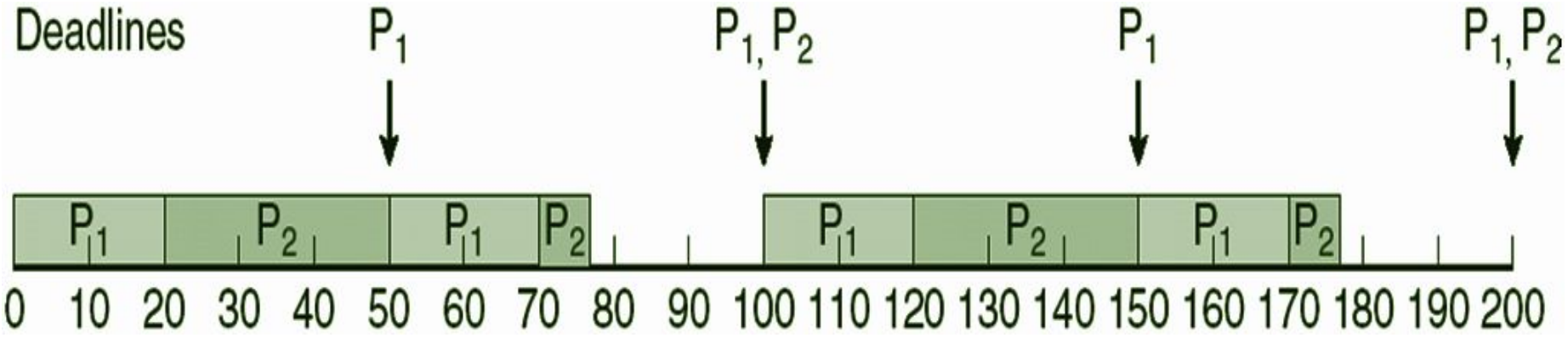
- **Revision on Real time scheduling sir**
  - Many real-time processes are periodic, i.e., they require CPU at constant intervals
  - Has processing time t, deadline d, period p $0 \leq t \leq d \leq p$; Rate of periodic task is 1/p



- **Revision on Real time scheduling sir**

Rate Monotonic Scheduling

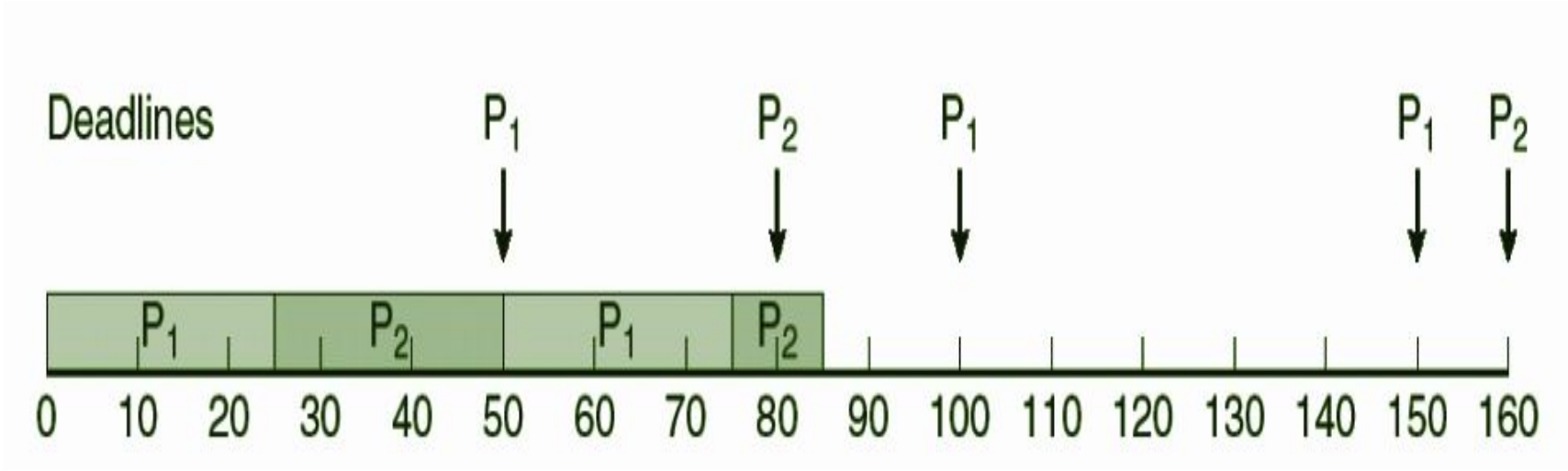
- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
  - P1 => 20 is assigned a higher priority than P2 => 30



- **Revision on Real time scheduling sir**

Rate Monotonic Scheduling

- A priority is assigned based on the inverse of its period
  - Shorter periods = higher priority
  - Longer periods = lower priority
  - P1 => 20 is assigned a higher priority than P2 => 30

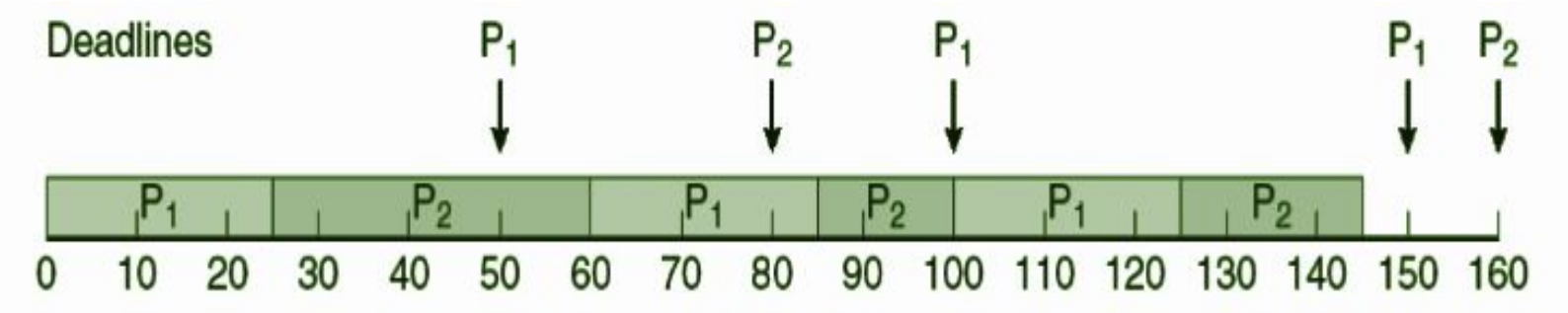


Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**

  Earliest Deadline First Scheduling

  - Priorities are assigned according to deadlines
    - the earlier the deadline, the higher the priority;
    - the later the deadline, the lower the priority



Revision Class Outline => Student Queries

- **Revision on Real time scheduling sir**

  Proportional Share Scheduling

  - *T* shares are allocated among all processes in the system
  - An application receives *N* shares where *N* < *T*
  - This ensures each application will receive ***N*** / *T* of the total processor time

- Recursive Mutexes, Semaphores, Spinlocks

Recursive Mutexes

- The difference between a recursive and non-recursive mutex has to do with ownership.
- In the case of a recursive mutex, the kernel has to keep track of the thread which actually obtained the mutex the first time around so that it can detect the difference between recursion vs. a different thread that should block instead.
- There is a question of the additional overhead of this both in terms of memory to store this context and also the cycles required for maintaining it.

- **Recursive Mutexes, Semaphores, Spinlocks**

Recursive Mutexes

- Because the recursive mutex has a sense of ownership, the thread that grabs the mutex must be the same thread that releases the mutex.
- In the case of non-recursive mutexes, there is no sense of ownership and any thread can usually release the mutex no matter which thread originally took the mutex.
- In many cases, this type of "mutex" is really more of a semaphore action, where you are not necessarily using the mutex as an exclusion device but use it as synchronization or signaling device between two or more threads.

- **Recursive Mutexes, Semaphores, Spinlocks**

Recursive Mutexes

- Another property that comes with a sense of ownership in a mutex is the ability to support priority inheritance.
- Because the kernel can track the thread owning the mutex and also the identity of all the blocker(s), in a priority threaded system it becomes possible to escalate the priority of the thread that currently owns the mutex to the priority of the highest priority thread that is currently blocking on the mutex.
- This inheritance prevents the problem of priority inversion that can occur in such cases.
- Not all systems support priority inheritance on such mutexes, but it is another feature that becomes possible via the notion of ownership.

- **Recursive Mutexes, Semaphores, Spinlocks**

Recursive Mutexes

classic VxWorks RTOS kernel, they define three mechanisms:

- **mutex** - supports recursion, and optionally priority inheritance. This mechanism is commonly used to protect critical sections of data in a coherent manner.

- **binary semaphore** - no recursion, no inheritance, simple exclusion, taker and giver does not have to be same thread, broadcast release available. This mechanism can be used to protect critical sections, but is also particularly useful for coherent signalling or synchronization between threads.

- **counting semaphore** - no recursion or inheritance, acts as a coherent resource counter from any desired initial count, threads only block where net count against the resource is zero.

This varies somewhat by platform - especially what they call these things, but this should be representative of the concepts and various mechanisms at play.

- **Recursive Mutexes, Semaphores, Spinlocks**

Spinlocks

- When two or more processes require dedicated access to a shared resource, they might need to enforce the condition that they are the sole process to operate in a given section of code. The basic form of locking in the Linux kernel is the spinlock.

- Spinlocks take their name from the fact that they continuously loop, or spin, waiting to acquire a lock. Because spinlocks operate in this manner, it is imperative not to have any section of code inside a spinlock attempt to acquire a lock twice. This results in deadlock.

- Recursive Mutexes, Semaphores, Spinlocks

Spinlocks

- Preemption is disabled during the lock. This ensures that any operation in the critical section is not interrupted.
- The drawback of spinlocks is that they busily loop, waiting for the lock to be freed.
- They are best used for critical sections of code that are fast to complete.

- **Recursive Mutexes, Semaphores, Spinlocks**

Semaphores

- For code sections that take time, it is better to use another for example in kernel locking utility: the semaphore.
- Semaphores differ from spinlocks because the task sleeps, rather than busy waits, when it attempts to obtain a contested resource.
- One of the main advantages of Semaphores is that a process holding a semaphore is safe to block; they are SMP and interrupt safe

Revision Class Outline => Student Queries

- **Recursive Mutexes, Semaphores, Spinlocks**

Mutex	Semaphore
The concepts about the mutex is much like lock. When you want to enter a room , you must acquire the lock then having the permission to enter.	Semaphore can set the initial value to N>1.
In computer science , the mutex is used for protect critical section preventing other thread to mutual access the same code to avoid race condition.	We can use the value to restrict how many users in this process will be permitted to access the critical section.
“Mutexes are typically used to serialise access to a section of re-entrant code that cannot be executed concurrently by more than one thread.	Mutex can only be released by user who holds it. But in semaphore mechanism any user can signal to any specific semaphore in kernel. This can ensure the thread concurrency.
A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.”	“A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).”

- **Recursive Mutexes, Semaphores, Spinlocks**

Mutex/Semaphore	Spinlock
Used when the task can sleep while waiting for the lock.	Used when the task can't sleep while waiting for the lock.
It can be held for a long duration of time.	It can be held for a short duration of time.
It can only be used in process context, not in interrupt context.	It can only be used in the Interrupt context, not in the Process context.
High overhead locking.	Low overhead locking.

Page Replacement

1. Find the location of the desired page on disk

2. Find a free frame:

    i. If there is a free frame, use it

    ii. If there is no free frame, use a page replacement algorithm to

    iii. Select a **Victim** frame

        - Write **Victim** frame to disk if **dirty**

Page replacement algorithm in general

Page Replacement

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Continue the process by restarting the instruction that caused the trap

Note: Potentially 2 page transfers for page fault – increasing EAT, if the page is swapped

Page replacement algorithm in general

Performance of Demand Paging - Worst Case

- Three major activities
  - Service the interrupt => careful coding means just several hundred instructions needed
  - Read the page => lots of time
  - Restart the process => again just a small amount of time

Demand paging performance calculation

Performance of Demand Paging - Worst Case

Demand paging performance calculation

- Page Fault Rate 0 <= p <= 1
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)
  - EAT = (1 – p) x memory access
    + p (page fault overhead => + [swap page **out]** + swap page **in** + Restart Overhead)

Demand Paging - Performance Calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if No Page Fault i.e p=>0 then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 0) * 200 + 0 * PFST

  => 200ns

Demand paging performance calculation

Demand Paging - Performance Calculation

Demand paging performance calculation

- Memory Access Time (MAT) = 200 nanoseconds

- if Page Fault i.e p=>1
  then EAT =>(1 – p) x MAT+ P * Page Fault Service Time (PFST)

  => (1 - 1) * 200 + 1 * PFST

  => PFST

Demand Paging - Performance Calculation

Demand paging performance calculation

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

THANK YOU

Nitin V Pujari
Faculty, Computer Science
Dean => IQAC, PES University

nitin.pujari@pes.edu

For Course Deliverables by the Anchor Faculty click on www.pesuacademy.com