电子科技大学
格拉斯哥学院
Glasgow College, UESTC

UESTC4019: Real-Time Computer Systems and Architecture

Lecture 16

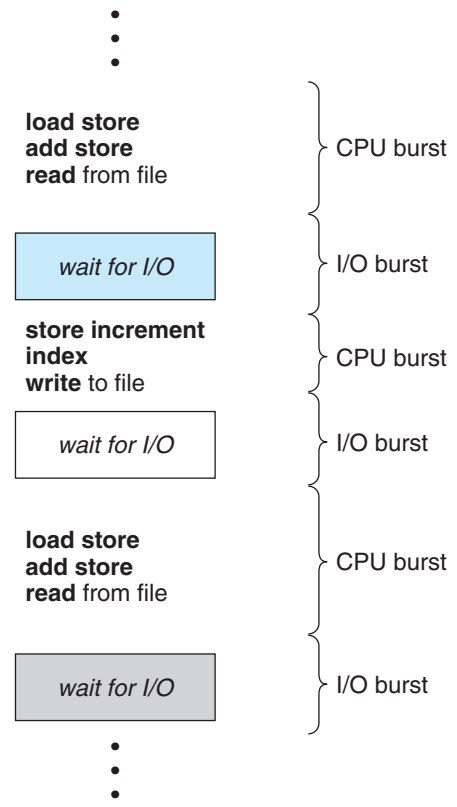
Scheduling

Objectives

- To introduce CPU scheduling, which is the basis for multiprogrammed operating systems
- To describe various CPU-scheduling algorithms
- To discuss evaluation criteria for selecting a CPU-scheduling algorithm for a particular system
- To examine the scheduling algorithms of several operating systems

Basic Concepts

- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a cycle of CPU execution and I/O wait
- CPU burst followed by I/O burst
- CPU burst distribution is of main concern



CPU Scheduler

- Short-term scheduler selects from among the processes in ready queue, and allocates the CPU to one of them
  - Queue may be ordered in various ways
- CPU scheduling decisions may take place when a process:
  1) Switches from running to waiting state
  2) Switches from running to ready state
  3) Switches from waiting to ready
  4) Terminates
- Scheduling under 1 and 4 is non-preemptive

CPU Scheduler

- All other scheduling is preemptive
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

CPU Scheduler

- Tasks are usually assigned with priorities
- At times it is necessary to run a certain task that has a higher priority before another task although it is running
- The running task is interrupted for some time and resumed later when the priority task has finished its execution. This is called preemptive scheduling
- In non-preemptive scheduling, a running task is executed till completion. It cannot be interrupted

CPU Scheduler

- Non-Preemptive (“Cooperative”)
  - Pro: Easier to manage shared memory
  - Con: Requires careful placement of explicit “yield” calls in source code of each thread to reduce latency
- Preemptive (event-driven context switch)
  - No explicit “yield” calls required in source code
  - More difficult to manage shared memory

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- CPU utilization – keep the CPU as busy as possible
- Throughput – no of processes that complete their execution per time unit
- Turnaround time – amount of time to execute a particular process
- Waiting time – amount of time a process has been waiting in the ready queue
- Response time – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)

Scheduling Algorithm Optimization Criteria

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

Process

- The fundamental concept in any operating system is the process
  - A process is an executing program
  - An OS can execute many processes at the same time (concurrency/multitasks)
  - Example: running a Text Editor and a Web Browser at the same time in the PC

- Processes have separate memory spaces
  - Each process is assigned a private memory space
  - One process is not allowed to read or write in the memory space of another process
  - If a process tries to access a memory location not in its space, an exception is raised (Segmentation fault), and the process is terminated
  - Two processes cannot directly share variables

Task and Thread

- One process can consists of one or more threads
  - Threads share the same memory space of the process, but each thread has its own user and kernel stack.

- Threads and processes are scheduled directly by the scheduler.

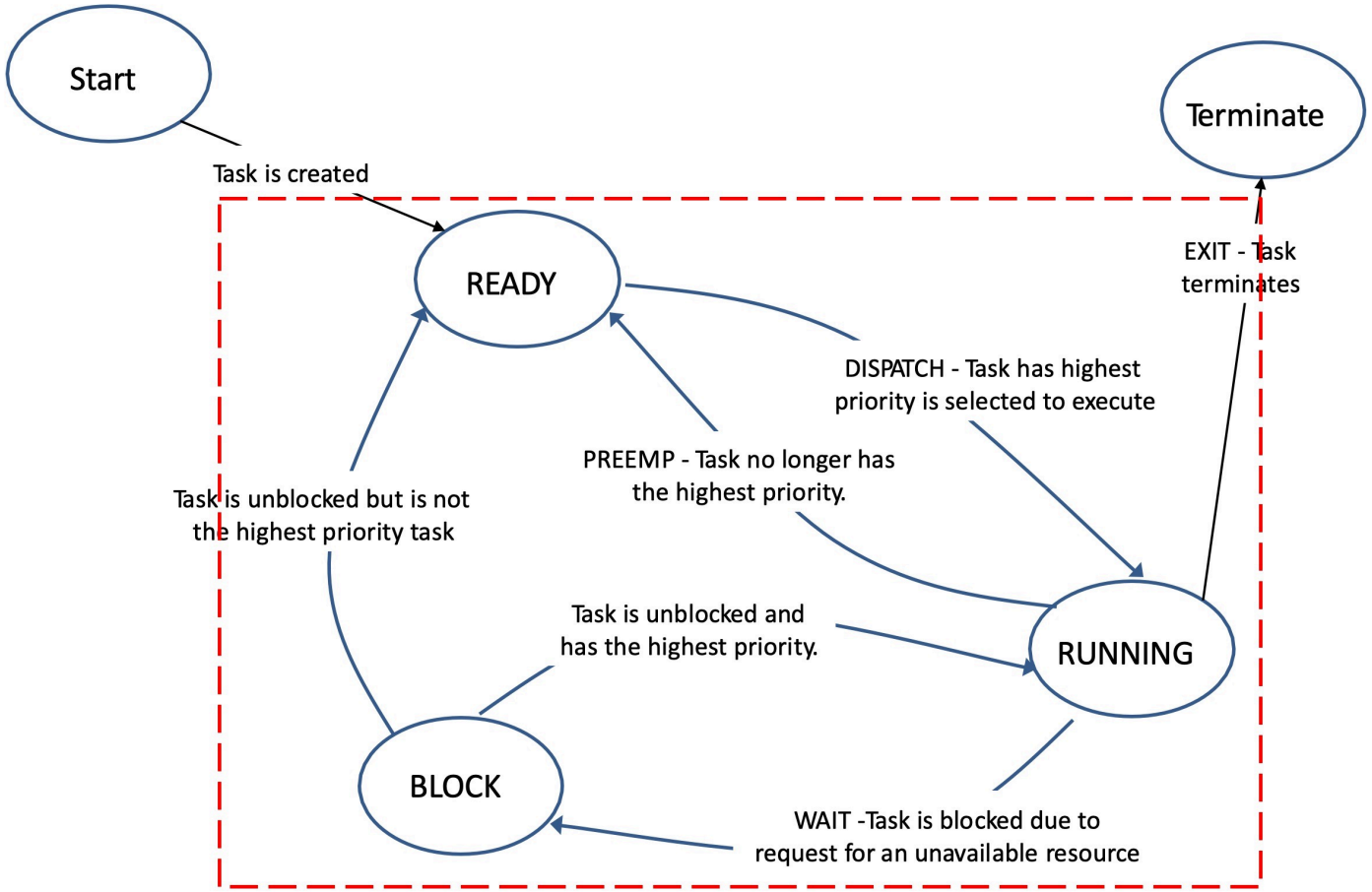
- Notes: we shall dispense with the fine distinction between “task” and “thread” and loosely treat them as synonymous!

Tasks & Task States

- The basic building block of software written under an RTOS is the task.
- To the RTOS, a task is somewhat like a function!
- Each application task managed by an RTOS is always in one of three states:

1. Running – the CPU is executing the instructions of this task! There is only one task in running state at any given time
2. Ready – Some other task is now running, CPU will be directed to run this task when it becomes available. (task already loaded in memory)
3. Blocked – the task will not be run (it is probably waiting for some external event).

Task/Thread States



Scheduling

- Scheduling is the method by which threads, processes or data flows are given access to system resources .

- Many real time systems are built with operating systems providing multitasking facilities, in order to:
  - Ease the design of complex systems (one function = one task).
  - Increase efficiency (I/O operations, multi-processor architecture).

Scheduling

- But the use of task scheduling implies that task schedulers must
  - Stay predictable
  - Take urgency/criticality task constraints into account

- Multitasking makes the predictability analysis difficult to do

Why Need Task Scheduling Models?

- The purpose of a real-time scheduling algorithm is to
  - ensure that critical timing constraints are met. (such as deadlines and response time,)
  - decisions are made that favour the most critical timing constraints, even at the cost of others
- How can we check and ensure that every tasks will meet their timing requirements?
- For a complex system (large number of tasks), the designer needs the help of analytical
- methods/schedulability tests to check the design.

Scheduling algorithms

- A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms
- There are six popular process scheduling algorithms:

1. First-Come, First-Served (FCFS)
   - Easy implementation
   - Poor in performance
2. Shortest-Job-First (SJF)
   - Minimising waiting time
   - Hard to implement in interactive system as required CPU time is not know
3. Shortest Remaining Time First (SRTF)
   - processor is allocated to the job closest to completion
   - Hard to implement in interactive system – unknown remaining time

Scheduling algorithms

4. Priority Based (PB)
   - Each process is assigned a priority
   - Process with highest priority executed first, Process with same priority are execute on FCFS
   - How to decide priority of a process?
5. Round Robin(RR)
   - Each process is provided a fix time to execute
   - Process pre-empted after executed for a given time period
6. Multiple-Level Queues (MLQ)
   - Processes are grouped into different queue (CPU bound, IO bound etc..)
   - Each queue can have its own scheduling algorithms and priorities.

First- Come, First-Served (FCFS) Scheduling

Process	Burst Time
P_1	24
P_2	3
P_3	3

- Suppose that the processes arrive in the order: P_1 , P_2 , P_3
  The Gantt Chart for the schedule is:

- Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

FCFS Scheduling (Cont.)

.

Suppose that the processes arrive in the order:

$$P_2 , P_3 , P_1$$

- The Gantt chart for the schedule is:

- Waiting time for $P_1 = 6; P_2 = 0; P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3
- Much better than previous case
- **Convoy effect** - short process behind long process
  - Consider one CPU-bound and many I/O-bound processes

Shortest-Job-First (SJF) Scheduling

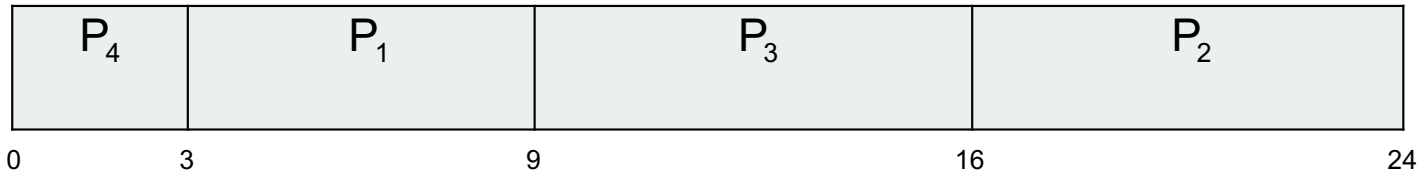
- Associate with each process the length of its next CPU burst
  - Use these lengths to schedule the process with the shortest time

- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU request
  - Could ask the user

Example of SJF

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

- SJF scheduling chart



- Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

Shortest-Remaining-Time-first (SRTF) Scheduling

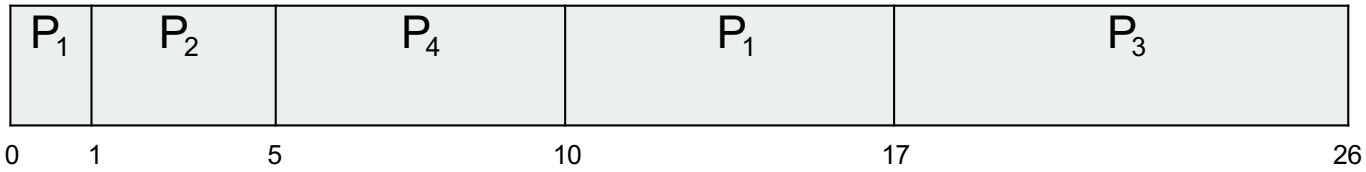
- Processor is allocated to the job closest to completion
- Hard to implement in interactive system – unknown remaining time

Example of SRTF

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

Priority Based (PB) Scheduling

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer ≡ highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is priority scheduling where priority is the inverse of predicted next CPU burst time
- Problem ≡ **Starvation** – low priority processes may never execute
- Solution ≡ **Aging** – as time progresses increase the priority of the process

Example of Priority Scheduling

Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- Priority scheduling Gantt Chart

- Average waiting time = 8.2 msec

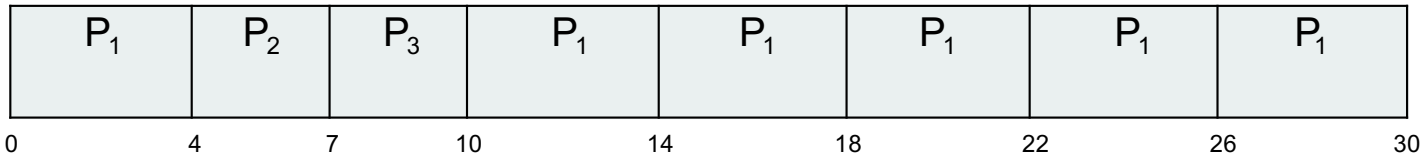
Round Robin (RR)

- Each process gets a small unit of CPU time (**time quantum** q), usually 10-100 milliseconds. After this time has elapsed, the process is preempted and added to the end of the ready queue.
- If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units at once. No process waits more than $(n-1)q$ time units.
- Timer interrupts every quantum to schedule next process
- Performance
  - q large ⇒ FIFO
  - q small ⇒ q must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

Process	Burst Time
P_1	24
P_2	3
P_3	3

- The Gantt chart is:

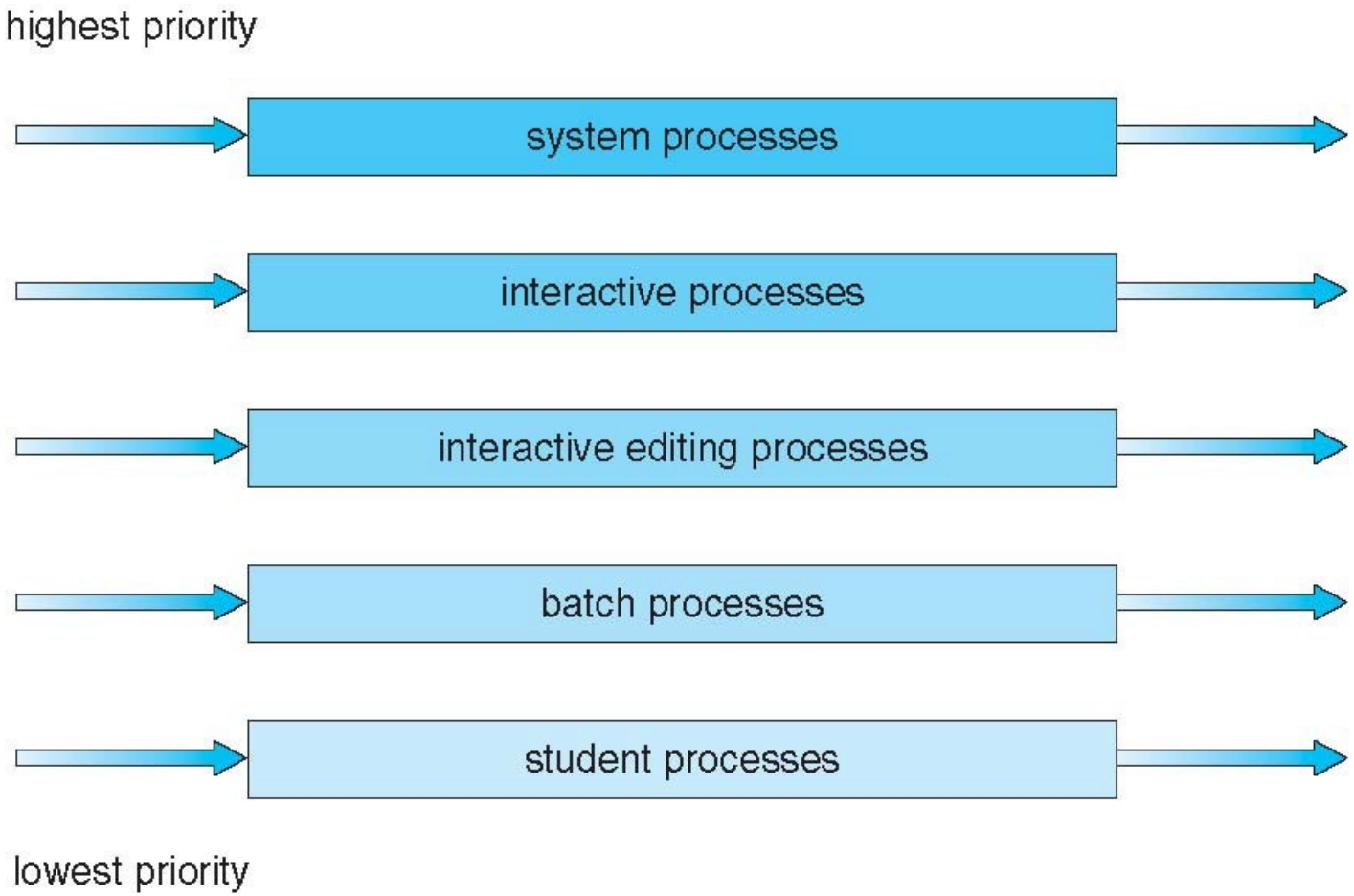


- Typically, higher average turnaround than SJF, but better response
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

Multilevel Queue

- Ready queue is partitioned into separate queues, eg:
  - **foreground** (interactive)
  - **background** (batch)
- Process permanently in a given queue
- Each queue has its own scheduling algorithm:
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multilevel Queue Scheduling



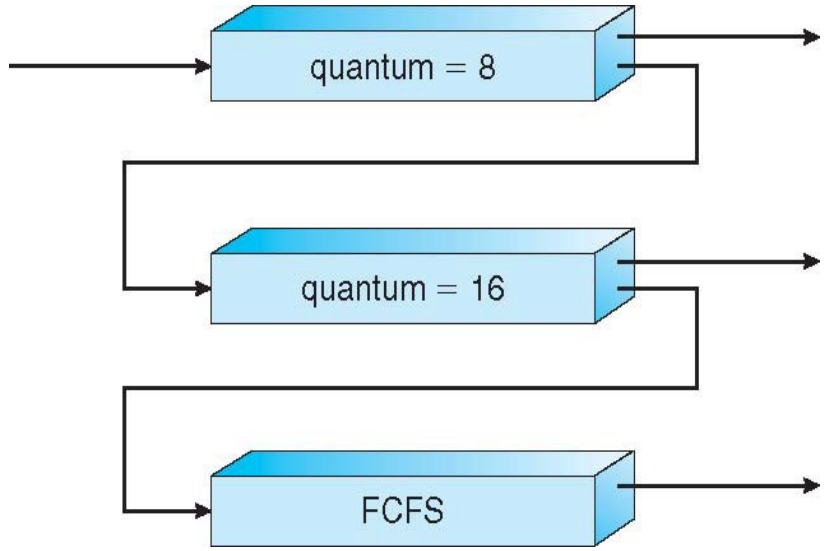
Multilevel Feedback Queue

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue

- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

- Scheduling
  - A new job enters queue Q_0 which is served FCFS
    - When it gains CPU, job receives 8 milliseconds
    - If it does not finish in 8 milliseconds, job is moved to queue Q_1
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds
    - If it still does not complete, it is preempted and moved to queue Q_2



Real-Time System Scheduling

- Task: sequence of statements + data + execution context (processor and floating point processor)

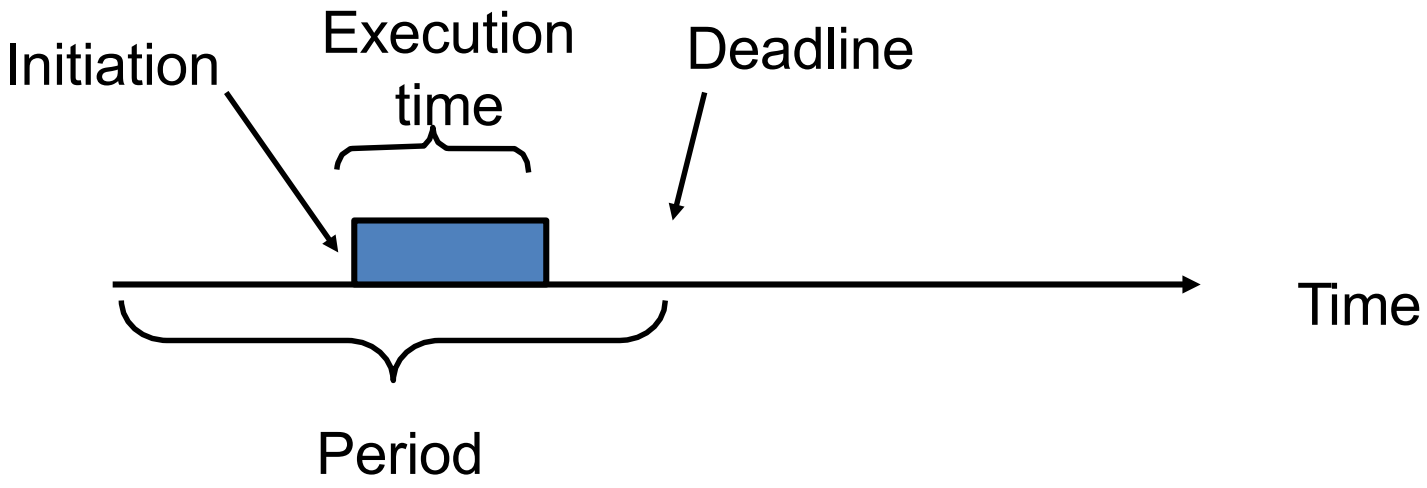
- Usual task types :
  - Repetitive tasks (**periodic, sporadic**)
    - **Periodi**c tasks are started at regular intervals and has to be completed before some deadline
    - **Sporadic** tasks are appeared irregularly, but within a bounded frequency
  - Non repetitive task (**aperiodic**)
    - **Aperiodic** tasks – event driven, parameters are completely unknown
  - Urgent and/or critical task.
  - Independent task or dependent task

Real-Time System Scheduling

- Main goal of an RTOS scheduler: meeting all deadlines
- Fairness or throughput is not important
- Worry about worst case performance
- Typical RTOS are based on fixed-priority preemptive scheduler
  - Assign each process a priority
  - At any time, scheduler runs highest priority process that is in "**ready state**". (ready to run)
  - Process runs to completion unless waiting for resource/IO or preempted

RTOS Task Model

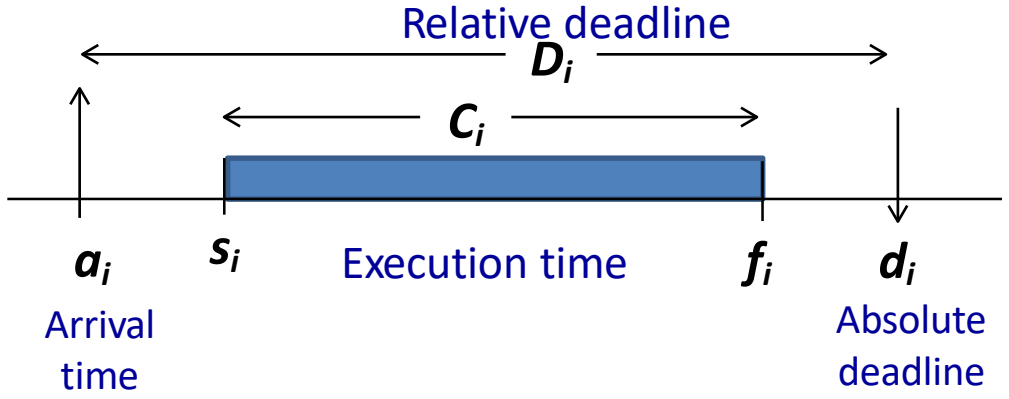
- Each task a triplet: (execution time, period, deadline)
- Can be initiated any time during the period



RTOS Task Model

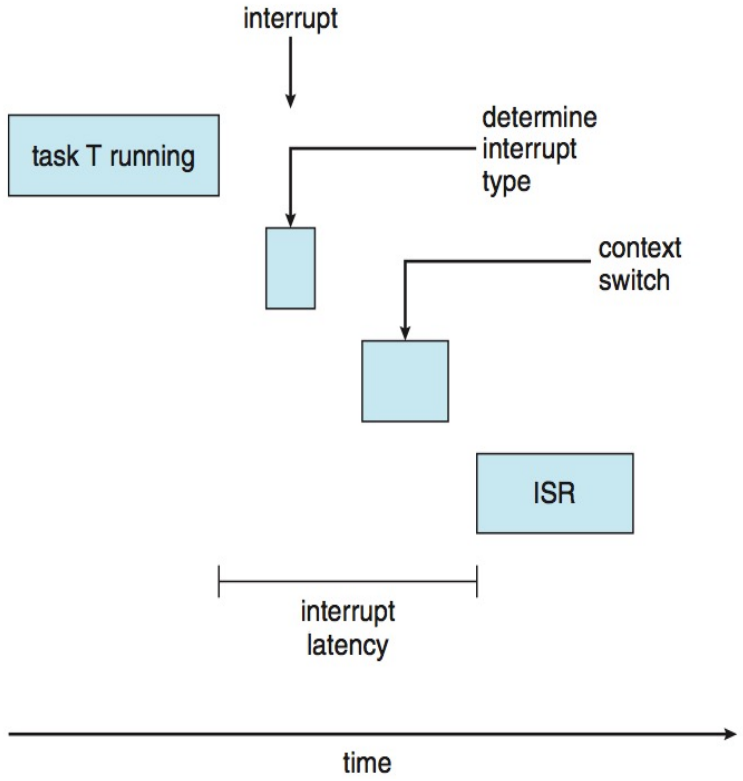
- a_i (arrival time) is the time a task becomes ready for execution.
- C_i (computation/execution time) is the worst-case execution time of the task under the assumption that the task is not interfered once it is run.
- d_i (absolute deadline) / D_i (relative deadline):
  - Absolute deadline means a value with respect to the global time of the entire system.
  - Relative deadline means relative to the arrival time of the respective task.

- Parameters controlled by RTOS:
  - s_i (start time) is the time when RTOS begins executing the task. It cannot be earlier than a_i.
  - f_i (finishing time) must be at least (s_i + C_i) if the task is not suspended mid way.



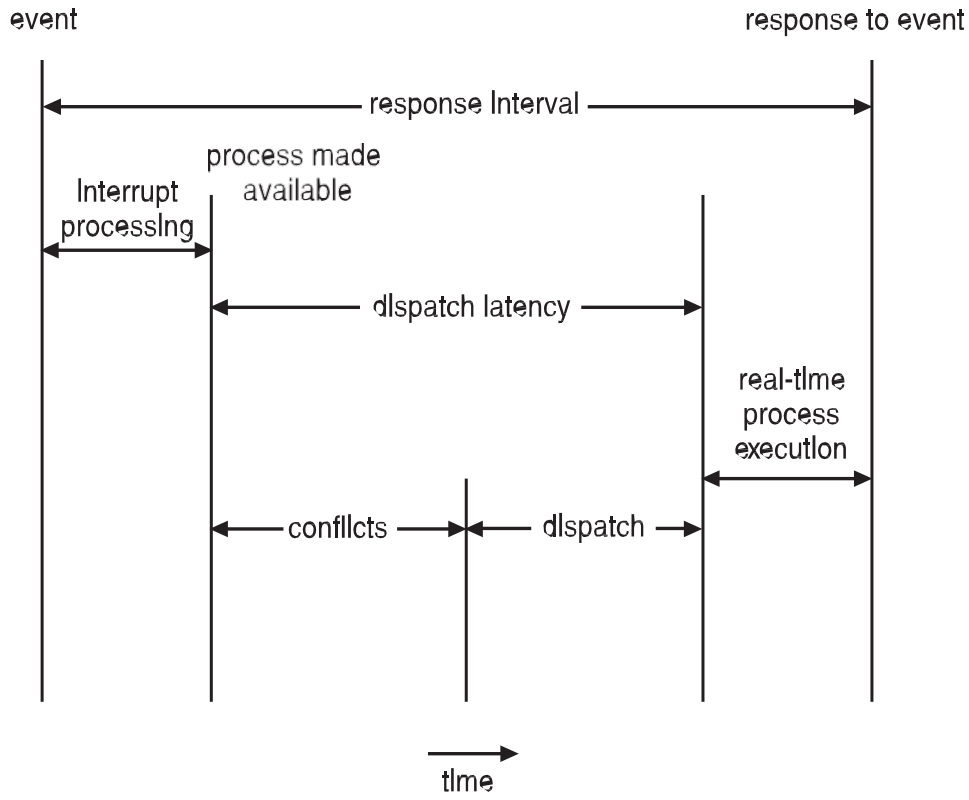
Real-Time CPU Scheduling

- Can present obvious challenges
- **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 deadline
- Two types of latencies affect performance
  1. Interrupt latency – time from arrival of interrupt to start of routin that services interrupt
  2. Dispatch latency – time for schedule to take current process of and switch to another



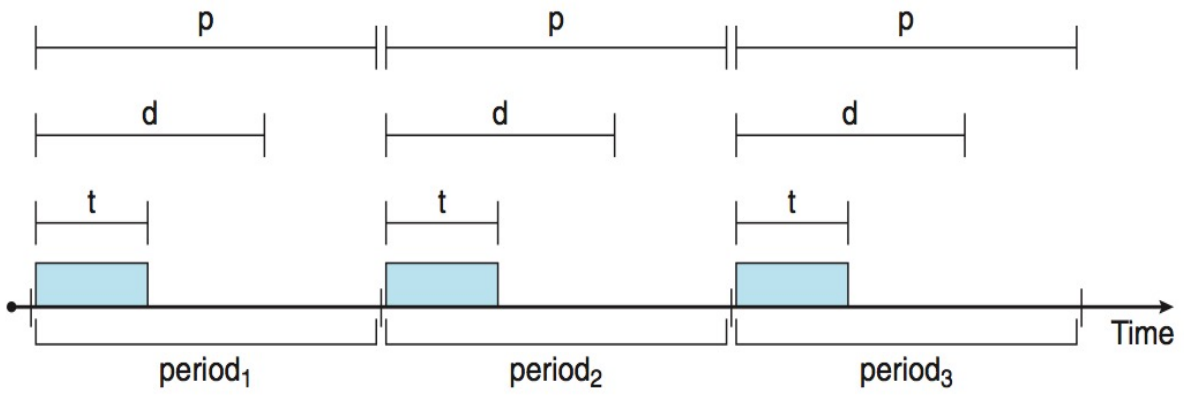
Real-Time CPU Scheduling

- 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



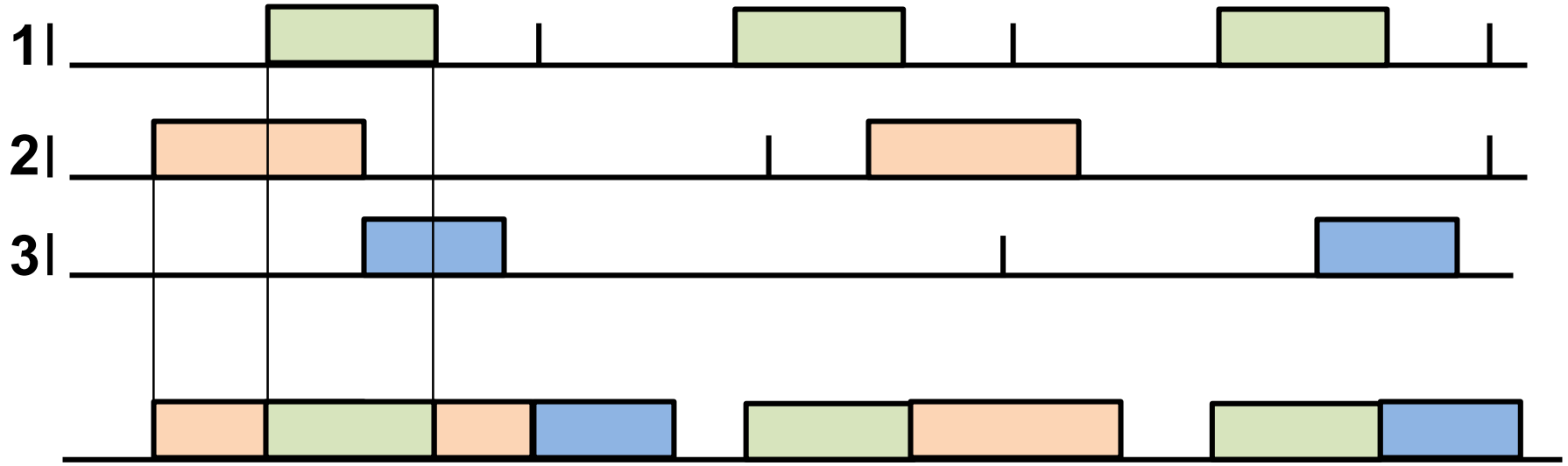
Priority-based Scheduling

- For real-time scheduling, scheduler must support preemptive, priority- based scheduling
  - But only guarantees soft real-time
- For hard real-time must also provide ability to meet deadlines
- Processes have new characteristics: **periodic** ones 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 $1/p$



Example of Priority-based Scheduling

- Always run the highest-priority runnable process



Rate Monotonic Scheduling

- Rate-monotonic scheduling (RMS) - a scheduling algorithm used in real-time operating systems (RTOS).
- From Liu & Layland, 1973 (JACM)
- Each task is assigned with a static-priority
- It assigns priority according to period - A task with a shorter period has a higher priority.
- The static priorities are assigned on the basis of the cycle duration of the job, tasks with shorter periods get the higher priorities!

Rate Monotonic Example (1 of 2)

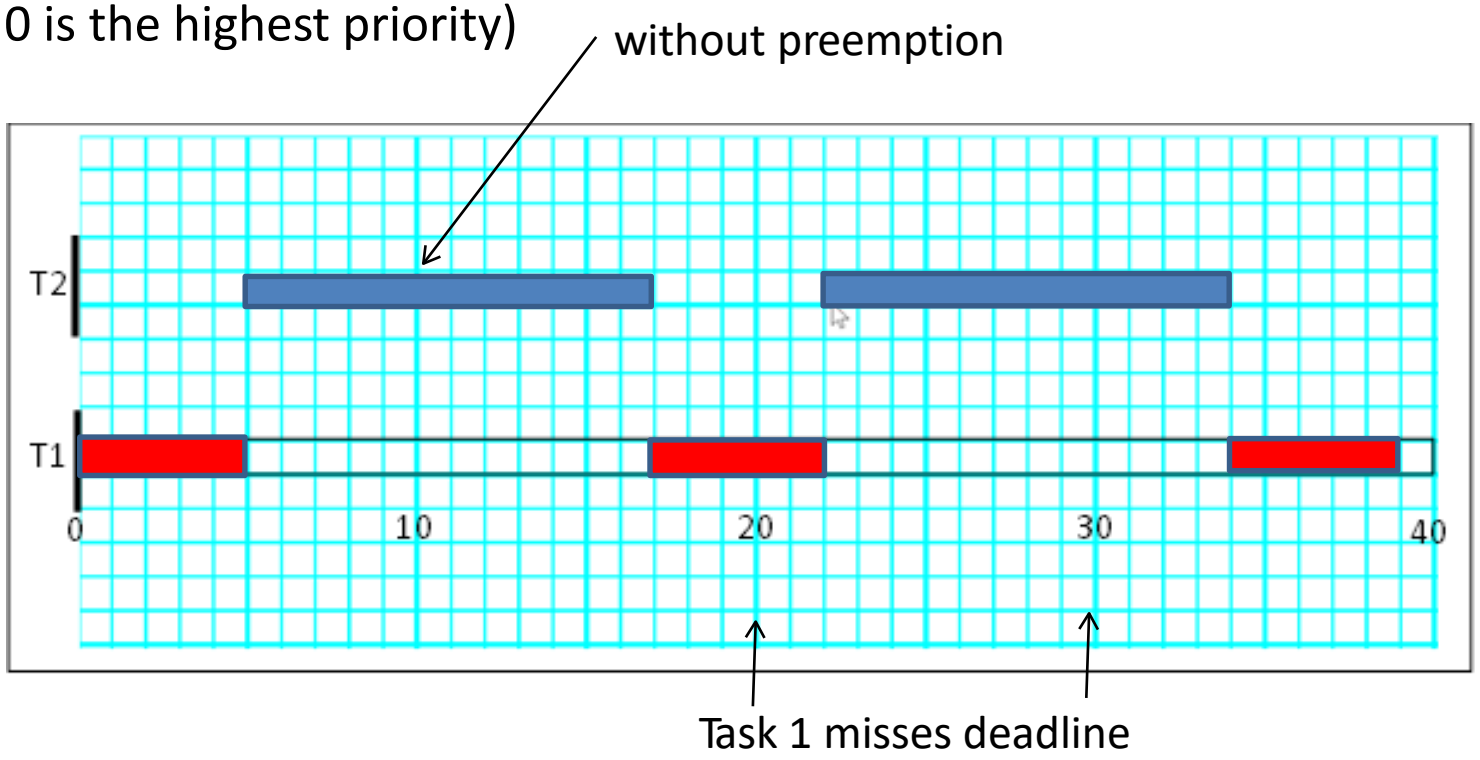
Two tasks T1 and T2 have the following parameters:

T1: - C1 = 5 (computation time)

- D1 =10 (relative deadline – shorter cycle duration)
- Priority = 0 (assume here 0 is the highest priority)

T2: - C2 = 12

- D2 = 30
- Priority = 1



Rate Monotonic Example (2 of 2)

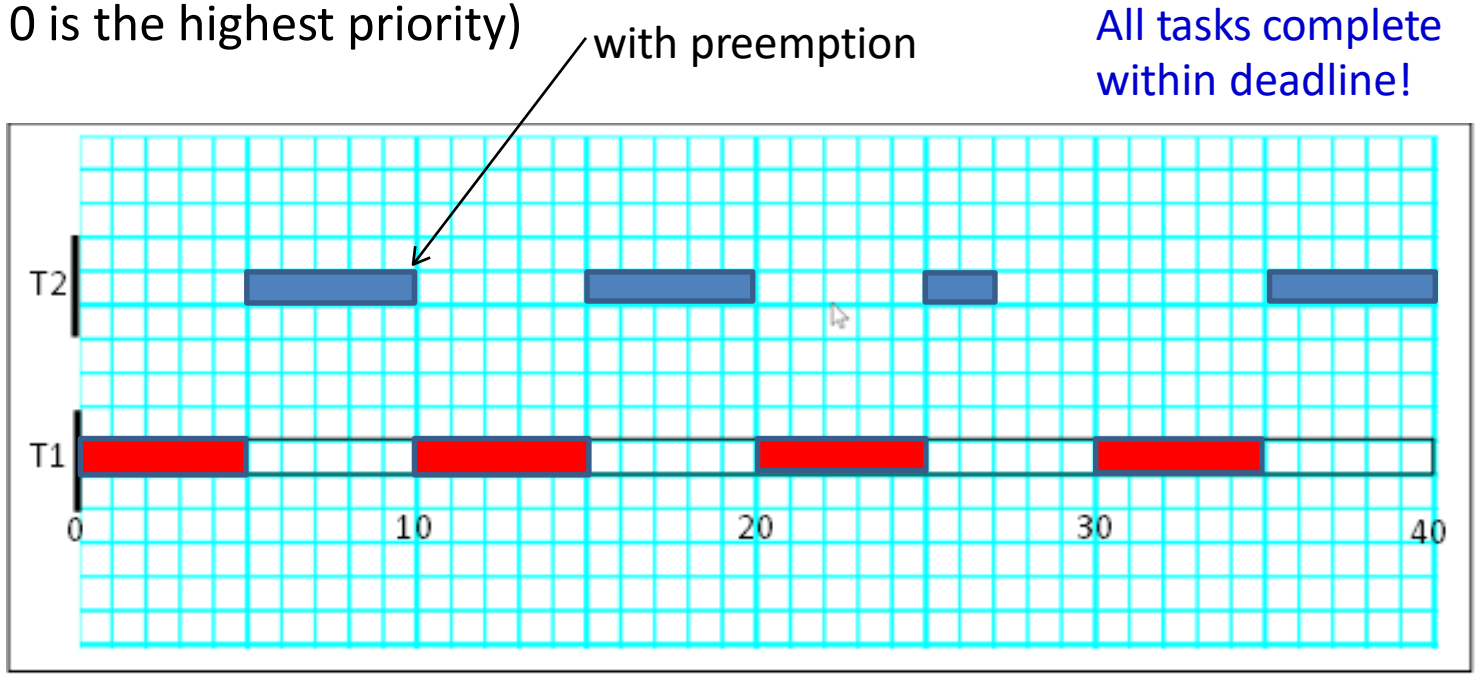
- Two tasks T1 and T2 have the following parameters:

T1: - C1 = 5 (computation time)

- D1 =10 (relative deadline)
- Priority = 0 (assume here 0 is the highest priority)

T2: - C2 = 12

- D2 = 30
- Priority = 1



Rate-Monotonic Analysis (RMA)

- Used on tasks scheduled by many different systems to reason about schedulability. (mathematical and scientific model)
- Liu & Layland (1973) proved that a set of *n* periodic tasks will always meet deadlines if the CPU utilization is below a specific bound.

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1)$$

- where C_i is the computation time, T_i is the release period (with deadline one period later), and *n* is the number of processes to be scheduled.
  - For example, $U \leq 0.8284$ for two processes/tasks.
- When the number of processes tends towards infinity, this expression will tend towards: $\lim_{n\to\infty} (\sqrt[n]{2} - 1) = \ln 2 \approx 0.69314\ldots$.
  - A rough estimate that RMS can meet all of the deadlines if CPU utilization is less than 69.32%.

Rate-Monotonic Analysis Example

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

Rate-Monotonic Analysis Example

Process	Execution Time	Period
P1	1	8
P2	2	5
P3	2	10

The utilization will be:

$$\mathrm{U} = \sum_{i=1}^{n} \left(\frac{C_i}{T_i}\right) = \frac{1}{8} + \frac{2}{5} + \frac{2}{10} = 0.725$$

The sufficient condition for 3 processes, under which we can conclude that the system is schedulable is:

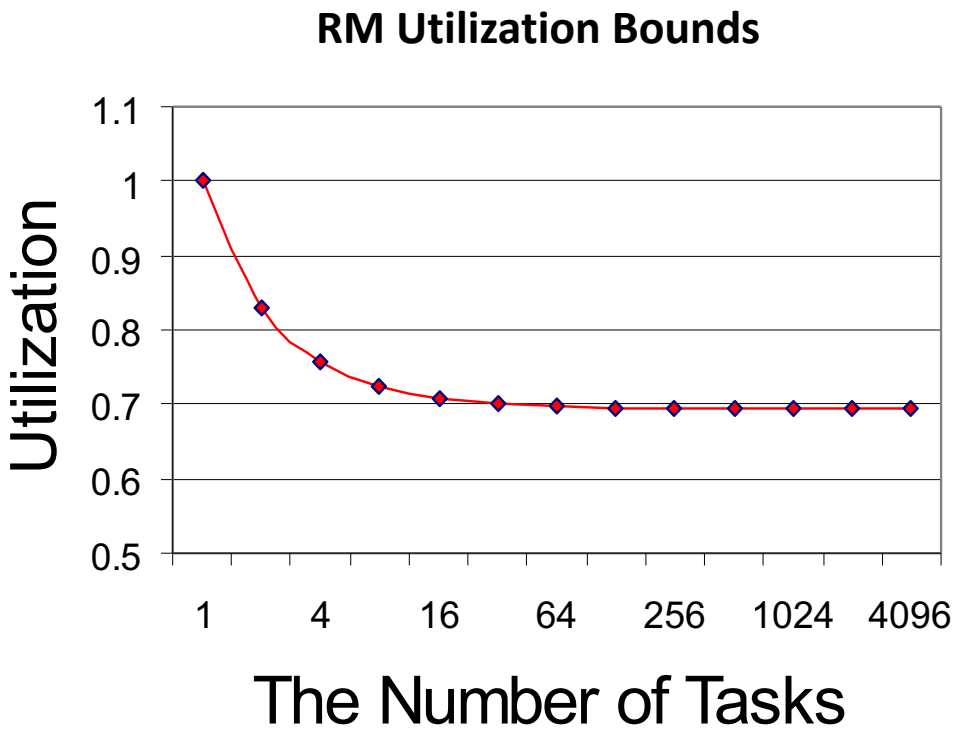
$$U_L = n\left(2^{\frac{1}{n}} - 1\right) = 3\left(2^{\frac{1}{3}} - 1\right) = 0.77976$$

Since 0.725< 0.77976, therefore the system is surely schedulable.

Rate-Monotonic Utilization Bound

- Real-time system is schedulable under Rate-Monotonic if

$$\sum U_i \leq n\,(2^{1/n}-1)$$



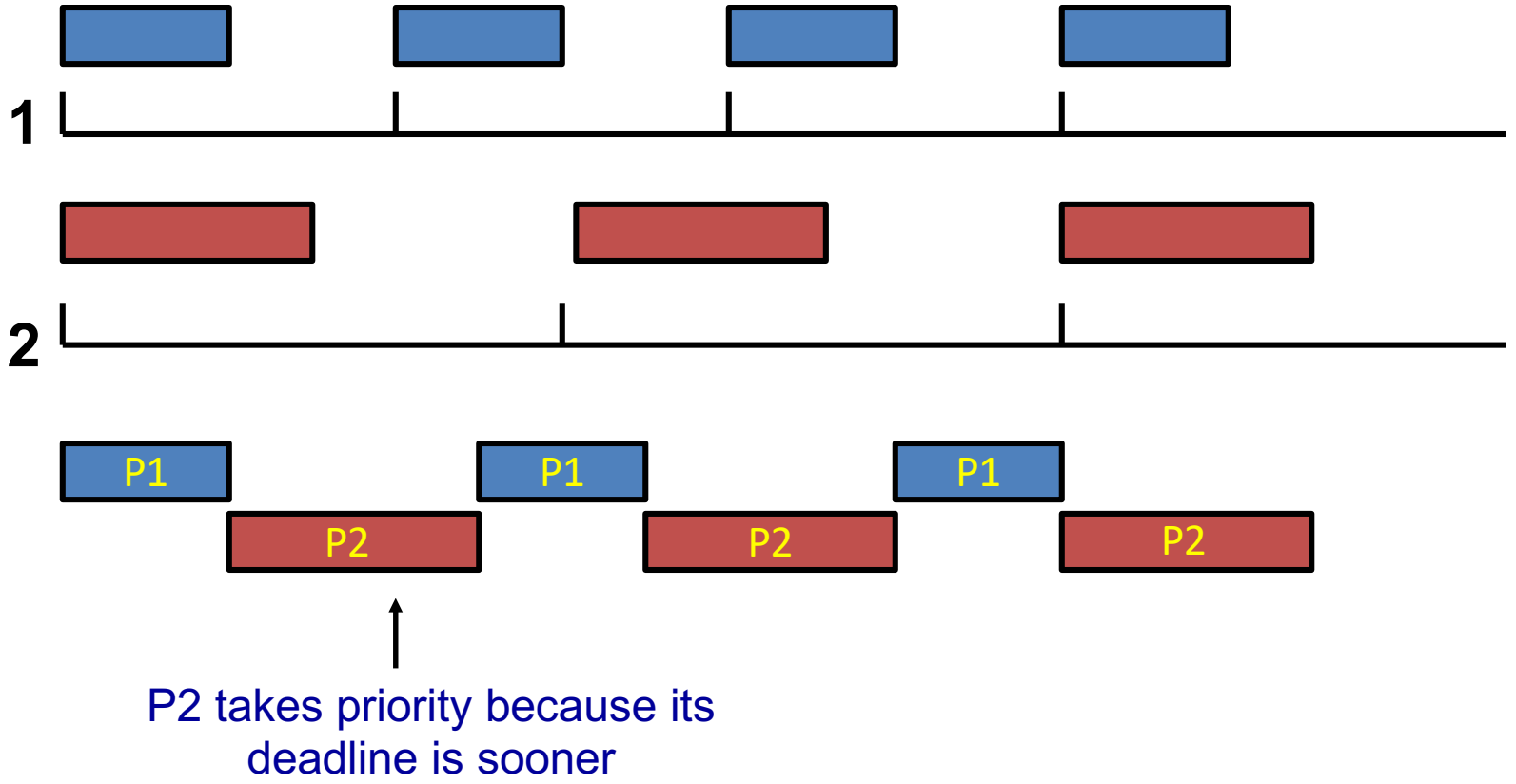
Earliest Deadline First Scheduling

- Given a finite set of non-repeating tasks with deadlines and no precedence constraints, a simple practical scheduling algorithm is Earliest Due Date (EDD) or Earliest deadline first (EDF), also known as Jackson's algorithm:
  - Optimal dynamic priority scheduling
  - A task with a shorter deadline has a higher priority
  - Executes a job with the earliest deadline
- The EDF strategy simply executes the tasks in the same order as their deadlines, with the one with the earliest deadline going first. If 2 tasks have the same deadline, then their relative order does not matter.
- Test of schedulablility:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$$

Earliest Deadline First Example

- p1 = (10,20,20) p2 = (15,30,30) utilization is 100%



Earliest Deadline First - Result

- Earliest deadline first scheduling is optimal:
  - If a dynamic priority schedule exists, EDF will produce a feasible schedule
- Earliest deadline first scheduling is efficient:
  - A dynamic priority schedule exists if and only if utilization is no greater than 100%

Rate-Monotonic vs Earliest Deadline First

- Rate Monotonic
  - Simpler implementation, even in systems without explicit support for timing constraints (periods, deadlines)
  - Predictability for the highest priority tasks
  - Only guarantees feasibility at 69% utilization, EDF guarantees it at 100%

- EDF
  - Full processor utilization
  - Misbehavior during overload conditions
  - Less predictable: can’t guarantee which process runs when

Example Question-1

- What is pre-emptive scheduling and non-preemptive scheduling? Briefly describe the pros and cons of the latter.

Example Question-1

- What is pre-emptive scheduling and non-preemptive scheduling? Briefly describe the pros and cons of the latter.

- Pre-emptive scheduling is based on timer interrupts, where a running thread may be interrupted by the OS and switched to the ready state at will (usually if something more important comes through) or when it has exceeded its timing allocation
- Non-preemptive scheduling means once a thread is in the running state, it continues until it completes, or gives up the CPU voluntarily

Example Question-1

- Pros:
  - Non-preemptive threads are likely to monopolies the CPU until it given up, thus all threads must work co-operably to not hold the CPU unnecessarily so that other threads can have the CPU time needed.
  - Another issue is such kind of scheduling cannot ensure the response time of each thread.
- Cons:
  - This type of scheduling has minimum contact switching and does not face deadlock problem.

Example Question-2

- Briefly explain the terms deadlock and starvation.

Example Question-2

- Briefly explain the terms deadlock and starvation.

- Deadlock is the phenomenon that arises when a number of concurrent processes all become blocked and cannot proceed without a resource that another holds, and all cannot release any resources that it is currently holding
- Starvation is the phenomenon which arises when a process is never given a resource that it requires to proceed (includes CPU time), even if it repeatedly becomes available, as it is always allocated to another waiting process (higher priority)

Example Question-3

- What are the key services provided by an OS?

Example Question-3

- What are the key services provided by an OS?
    - Program creation
    - Program execution
    - Access to I/O devices
    - Controlled access to files
    - System access
    - Error detection and response
    - Accounting