CPU Scheduling

Scheduling

- Teaching Schedule
  - Instructor schedule: who & when
  - Classroom schedule: what & when
- Doctor Appointment: who & when
- **Why do we need a schedule?**
- **Static** vs. **Dynamic**/Responsive Scheduling Algorithm
  - Class schedule vs. Air Traffic Controller

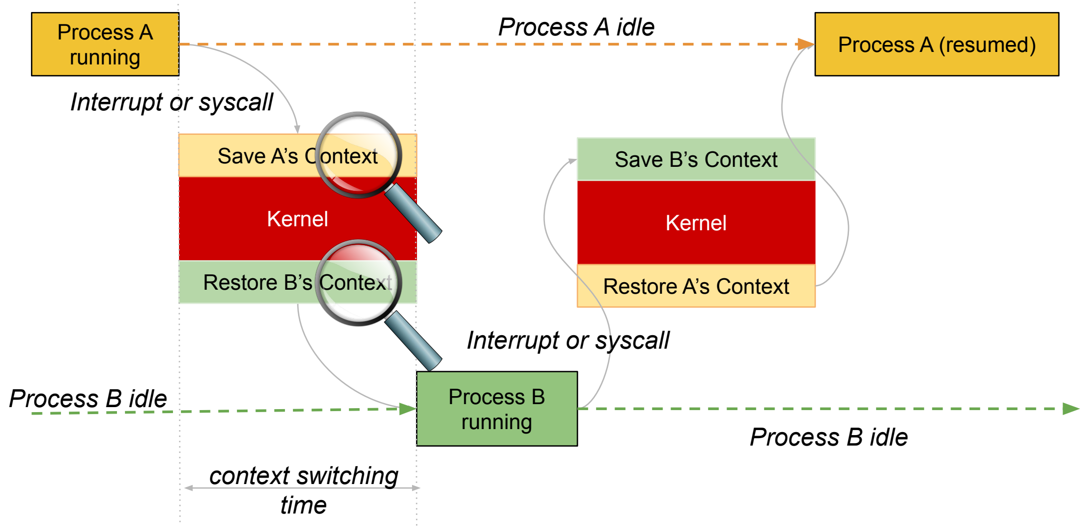
Thread/Process(or) Scheduling

- Dynamic scheduling
  - **who/what**: user processes competing for the same set of CPU(s)
  - **when**: when a process changes its state (state transition diagram)
- Scheduling Objective: keep the CPU occupied **all the time**! (*high utilization*)
  - User objective: (to save battery life) keeps the CPU idle most of the time (*low utilization*)

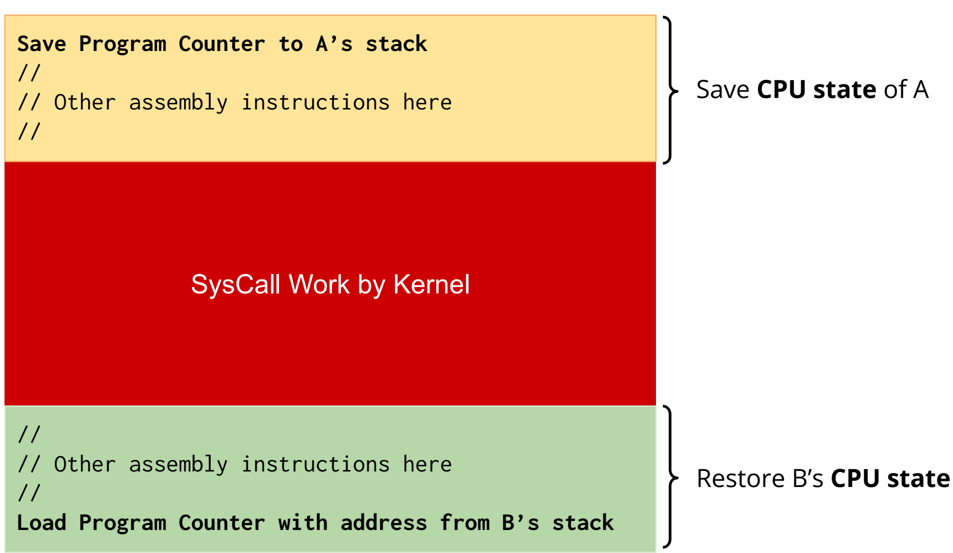
Important Takeaway Concepts

- The OS is a process(or) manager
- The OS *must run its code* on the same CPU(s) your processes run
- Hardware interrupts and syscalls enable OS to regain CPU control
- OS responsibility: *virtualize* the CPU
  - create an *illusion that your process owns* the CPU to itself throughout the process lifetime

Context Switching

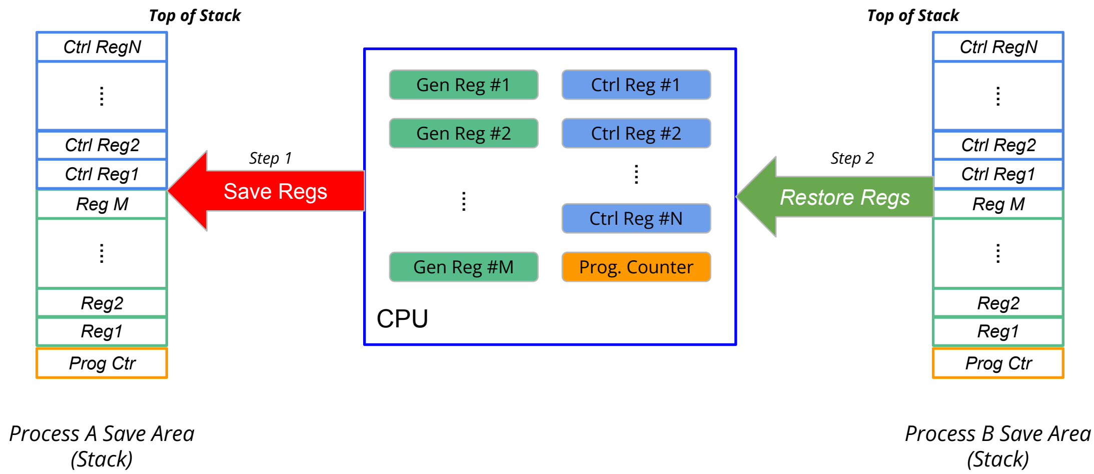


6



7

Context Switch from Process A to Process B

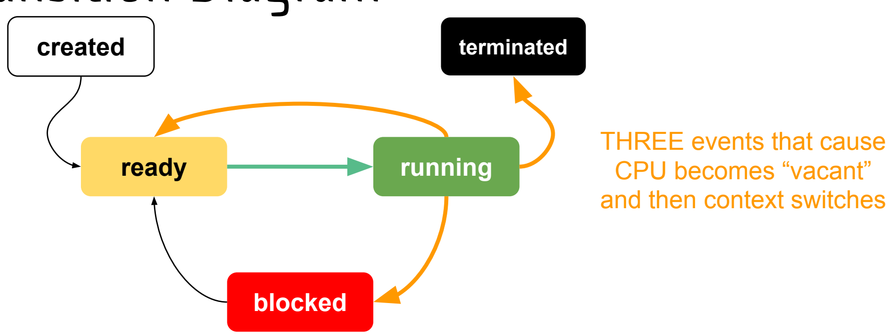


Processes - Threads - Jobs

We will use these terms interchangeably throughout this chapter

When does the OS schedule our processes/threads?

State Transition Diagram

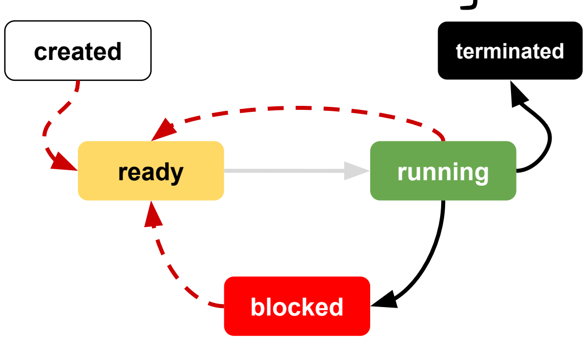


created ↠ ready: *the process just created, ready to use the CPU*
ready ↠ running: ***this is the scheduler's responsibility***
running ↠ ready: *the process time slice expired*
running ↠ blocked: *the process made a blocking system call (read(), sleep, **I/O requests**)*
blocked ↠ ready: the blocking system call completed, the process is ready to use the CPU again
running ↠ terminated: *the process exit normally (or with error)*

Preemptive vs. Cooperative Scheduling

- Non-preemptive / Cooperative
  - Temporary monopoly: once the CPU is allocated to a user process, the process keeps it
  - Scheduling decisions are made only when the user processes voluntarily release the CPU
  - **Transition Events: Running => Terminated and Running => Blocked**
- Pre-emptive
  - Each process is assigned a time-slice to use the CPU
  - The system can preempt a running process and assign the CPU to another process
  - **Transition Events: Running => Ready and Blocked => Ready**

State Transition Diagram



5 events (out of 6) => 5 intervention points

- Two voluntary actions by the user program to "release" the CPU (**COOPERATIVE SCHEDULERS**)
- Three async actions that *may initiate the OS logic* to kick out the current occupant of the CPU (**PREEMPTIVE SCHEDULERS**)

ready ↠ running: the process is dispatched by the OS to use the CPU
running ↠ blocked: the process made a blocking system call (read(), sleep, or **I/O requests**….)
running ↠ terminated: the process exit normally (or with error)
created ↠ ready: the process just created, ready to use the CPU
running ↠ ready: the process time slice expired
blocked ↠ ready: the blocking system call completed, the process is ready to use the CPU again

CPUs are fast
I/O devices are slooooow

CPU Speed and Clock Cycle

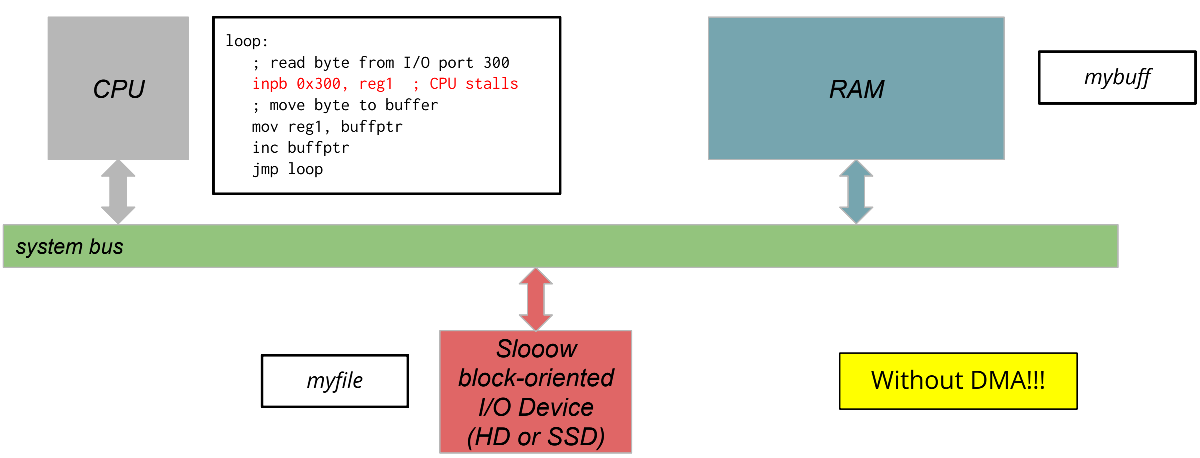
Speed	Clock Cycle	Operation Range
1Hz	1 sec	
1 kilo Hz = 10^3Hz	10^{-3} seconds = 1 millisec	*I/O devices*
1Mega Hz = 10^6Hz	10^{-6} seconds = 1 microsec	
1 Giga Hz = 10^9Hz	10^{-9} seconds = 1 nanosec	*CPUs*
1 Tera Hz = 10^{12}Hz	10^{-12} seconds = 1 picosec	

I/O devices can be 10^6 slower than CPU

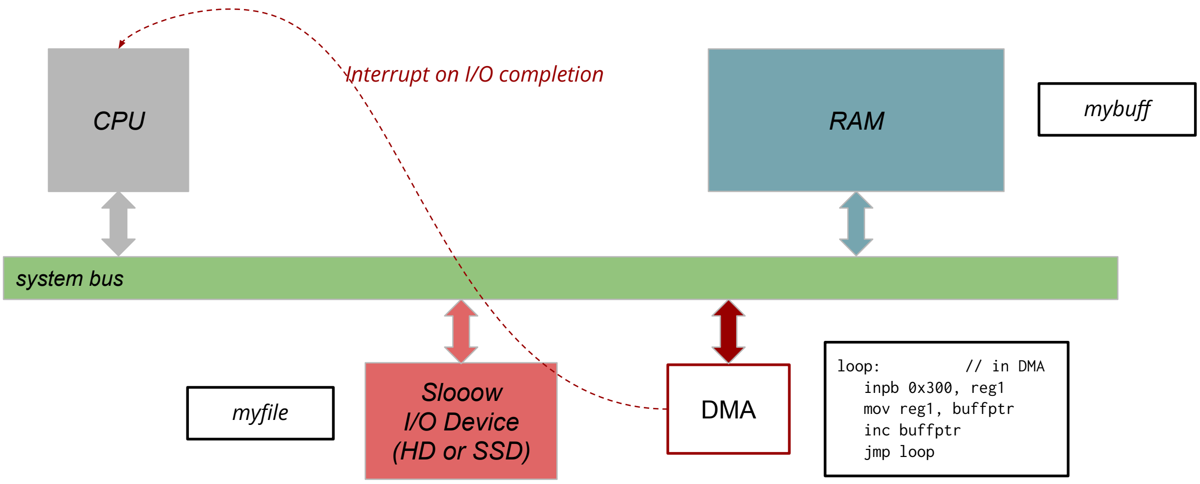
Handling I/O Operations using DMA

- CPUs operate in nanoseconds while I/O devices operate in millisecond
  - CPU speed in GHz (10^9 cycles/sec or 10^{-9} seconds/cycle)
  - HD access time in milliseconds, SSD access time in microseconds
- Direct handling of I/O operations by the CPUs *lower the CPU utilization* by many orders of magnitude (the CPU will spend most of its time **WAITING**)
- Delegate *block-oriented* I/O operations to dedicated I/O processors (DMA Controller / **Direct Memory Access** Controller)

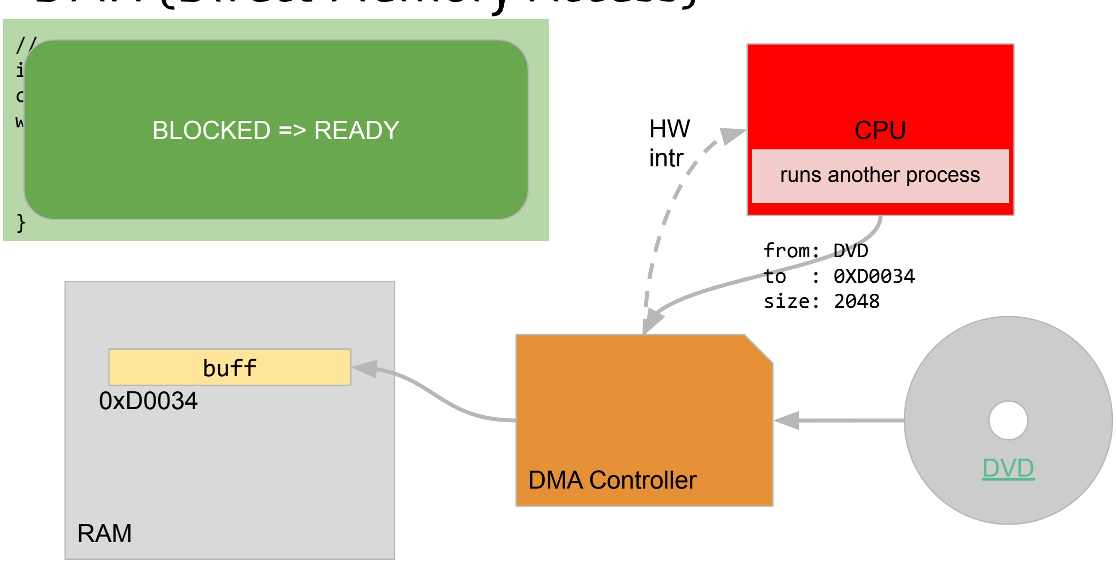
read (myfile_fd, &mybuff, sizeof(mybuff))



read (myfile_fd, &mybuff, sizeof(mybuff))

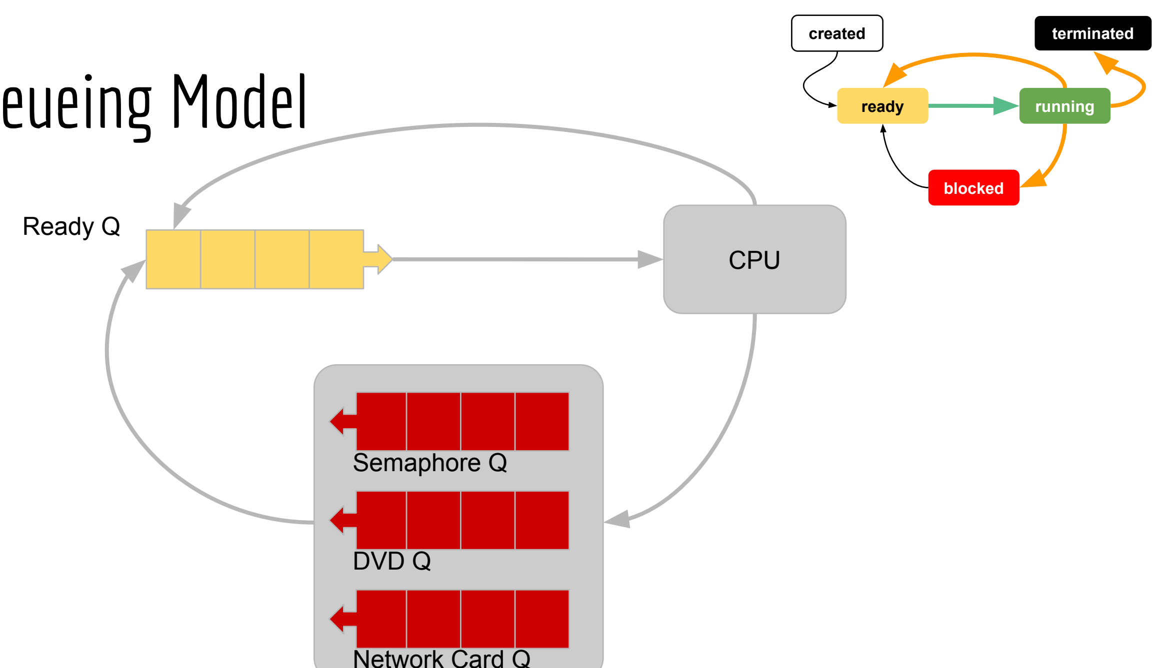


DMA (Direct Memory Access)



Scheduler Queuing Model

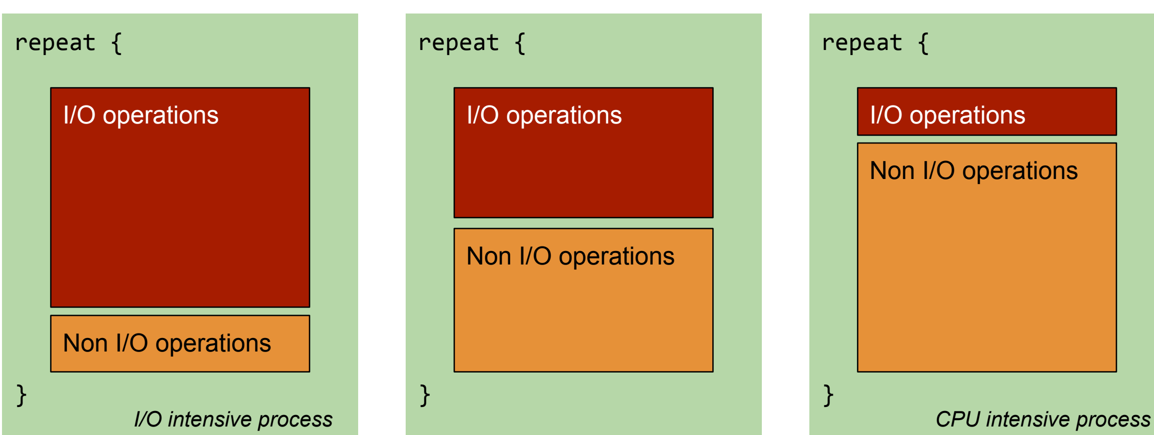
Queueing Model



Linux Source Code:

[kernel/sched.c]

Process Execution Pattern



… - **CPU burst** - **IO burst** - **CPU burst** - **IO burst** - **CPU burst** - ...

Types of Scheduling

- Short-Term Scheduling (or CPU Scheduler)
  - Decision to select a process (from the Ready Q) to use the CPU
- Medium-Term Scheduling
  - Decision to bring processes into memory (swapping in) or kick them out into swap space (swapping out)
  - Linux swap partition (type 82)
- Long-Term Scheduling
  - Decision to admit new processes into the system

Scheduling Objectives

- Max. CPU utilization: Keeps the CPU 100% utilized
- Max. Throughput: keeps as many active processes as possible
- Min. Turnaround time: total *lifetime* of a process
- Min. Waiting time: the total amount of time spent by a process *outside of CPU*
  - Either waiting in the ready queue or blocked
- Min. Response Time: (for interactive processes) the time for the system to respond to a user request
- *These objectives may be conflicting with each other*

Scheduling Algorithms

- Non-preemptive / Cooperative Algorithms: *CPU can't be stolen from the current process*
    - First-Come First-Served
    - Shortest-Job-First
- Preemptive Algorithms: *CPU can be stolen from the current process*
    - Round-Robin
    - Shortest Remaining Time (preemptive version of SJF)
    - Multilevel Queue
    - Multilevel Feedback Queue
    - Priority Scheduling

Non-Preemptive Scheduling Algorithms

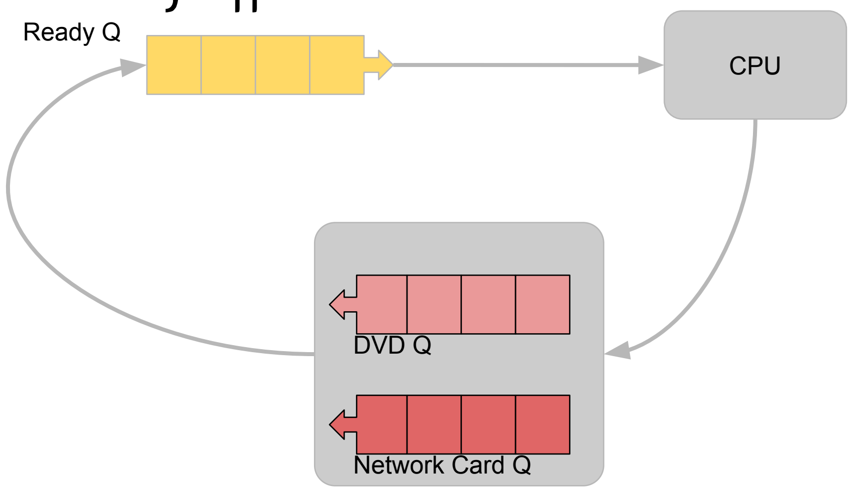
Processes are allowed to finish their entire CPU burst (without being kicked out of the CPU)

First-Come First-Served

- Select the “oldest” (frontmost) processes from the ready queue
- A short processes may have to wait a loooong time before it can run
- Favors CPU-bound processes
    - I/O-bound processes have to wait for CPU-bound processes to complete
    - Convoy effect
        - One CPU-bound process and many I/O-bound processes
        - All the I/O bound processes trailing behind the CPU-bound process

FIFO Convoy Effect

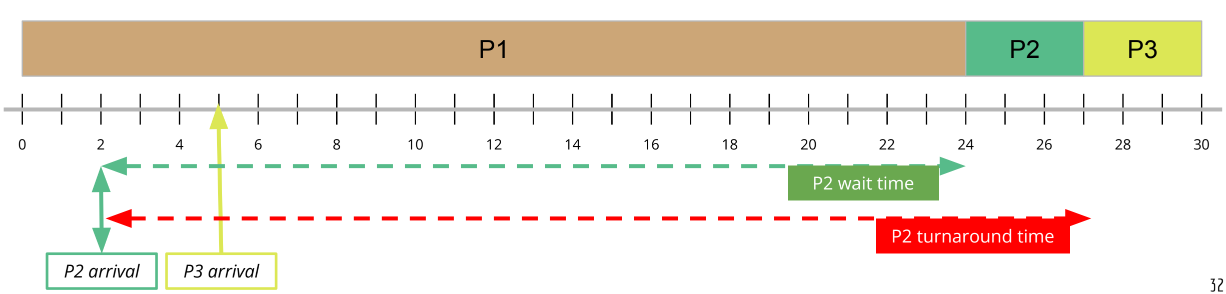
ONE ***CPU intensive job*** *+ MANY* ***I/O intensive*** *jobs*



Examples & Gantt Chart

FCFS Example

Process	CPU Burst Time	Arrival Time	Wait Time	Turnaround Time
P1	24	0		
P2	3	2		
P3	3	5		

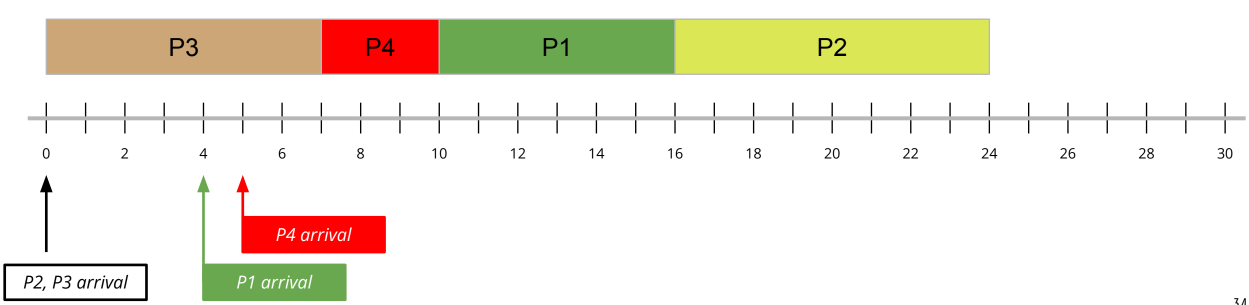


Shortest Job First

- Select a process with the shortest ***expected*** processing time/service time (or ***expected*** next CPU burst)
- Also called Shortest Process Next
- Short processes jump ahead of long processes (*possibility of starvation*)
- How to determine processing/service time
    - Batch jobs: **supplied** by the user
    - Interactive users: **estimated** the next CPU burst from the history of previous CPU bursts

SJF Example

Process	CPU Burst Time	Arrival	Wait Time	Turnaround Time
P1	6	4		
P2	8	0		
P3	7	0		
P4	3	5		



Implementation Issues

How long does it take you to drive to campus ***tomorrow***?

SJF: Estimate (Next) Service Time

- A **pure SJF algorithm** is impossible to implement
  - The **actual value** of the next CPU burst is unknown. Estimation is required
- How to estimate the next CPU burst ($\boldsymbol{\tau}_{n+1}$) from previous actual CPU bursts: $t_1, t_2, t_3, \ldots, t_n$
- Simple Average $\boldsymbol{\tau}_{n+1} = (t_1 + t_2 + t_3 + \ldots + t_n) / n$
- Exponential Average vs. Simple Average

Simple Average vs. Exponential Average

$$\tau_N = \frac{t_1+t_2+\cdots+t_N}{N}$$

$$\tau_{N+1} = \frac{t_1+t_2+\cdots+t_N+t_{N+1}}{N+1}$$

$$= \frac{t_1+t_2+\cdots+t_N}{N+1} + \frac{t_{N+1}}{N+1}$$

$$= \frac{N\,\tau_N}{N+1} + \frac{t_{N+1}}{N+1}$$

$$\tau_{N+1} = \frac{N}{N+1}\tau_N + \frac{1}{N+1}\,t_{N+1}$$

$$\tau_{N+1} = (1-s)\tau_N + s\,t_{N+1}$$

s is fixed constant between 0 and 1.
The weights are not affected by the number of measurements

As we accumulate more measurement (N >>) the weight on previous estimate ($\boldsymbol{\tau}_N$) overpowers the weight on the recent measurement

Simple Average vs. Exponential Average

CPU burst: 7, 8, 2, 3, 4

$$\tau_1 = t_1 = 7$$
$$\tau_2 = \frac{1}{2}(7+8) = 7.5$$
$$\tau_3 = \frac{1}{3}(7+8+2) = 5.67$$
$$\tau_4 = \frac{1}{4}(7+8+2+3) = 5$$
$$\tau_5 = \frac{1}{5}(7+8+2+3+4) = 4.8$$

Line Graph Plot Example

$$\tau_0 = 5 \qquad s = 0.8$$
$$\tau_1 = 0.2\ \tau_0 + 0.8\ t_1$$
$$= (0.2)(5) + (0.8)(7) = 6.6$$
$$\tau_2 = 0.2\ \tau_1 + 0.8\ t_2$$
$$= (0.2)(6.6) + (0.8)(8) = 7.72$$
$$\tau_3 = 0.2\ \tau_2 + 0.8\ t_3$$
$$= (0,2)(7.72) + (0.8)(2) = 3.14$$
$$\tau_4 = 0.2\ \tau_3 + 0.8\ t_4$$
$$= (0.2)(3.14) + (0.8)(3) = 3.03$$
$$\tau_5 = 0,2\ \tau_4 + 0.8\ t_5$$
$$= (0.2)(3.03) + (0.8)(4) = 3.8$$

Preemptive Scheduling Algorithms

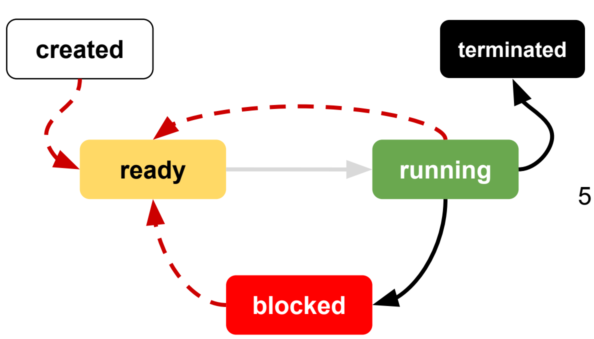
A process may be kicked-out of the CPU in the middle of its running state

*(able to run only **part** of its CPU burst)*

Shortest Remaining Time (SRT)

- Preemptive version of Shortest Job First
  - Shortest **Remaining CPU burst**
- *The currently running process is preempted when a (new) process (re)enters the ready queue*
- When a process is preempted (from the CPU), the process uses only a fraction of its CPU burst
  - Its remaining CPU burst will be used to dispatch it (later)

State Transition Diagram

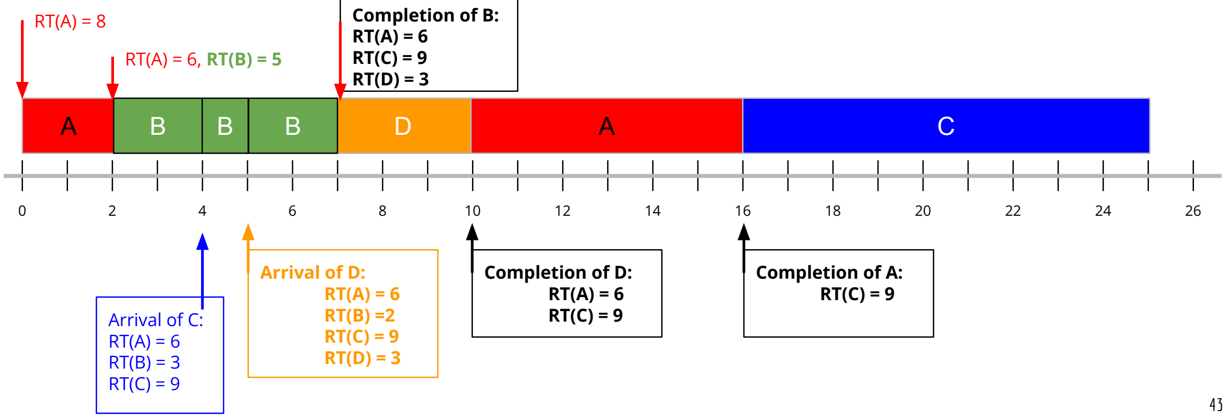


5 events (out of 6) => 5 intervention points

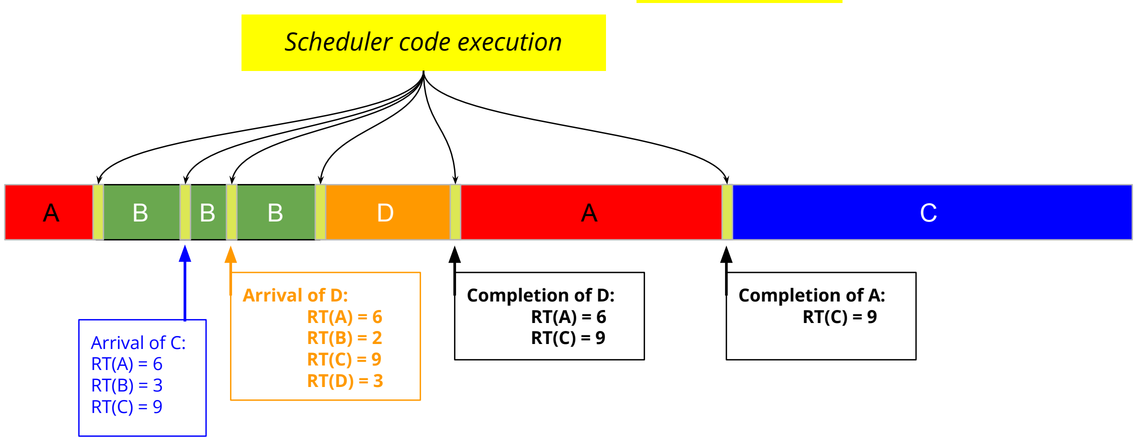
- Two voluntary actions by the user program to "release" the CPU (**COOPERATIVE SCHEDULERS**)
- Three async actions that *may initiate the OS scheduler* to kick out the current occupant of the CPU (**PREEMPTIVE SCHEDULERS**)

SRT Example

Process	CPU Burst Time	Arrival Time	Wait Time
A	8	0	
B	5	2	
C	9	4	
D	3	5	



Gantt Chart: Theoretical vs. *"Reality"*

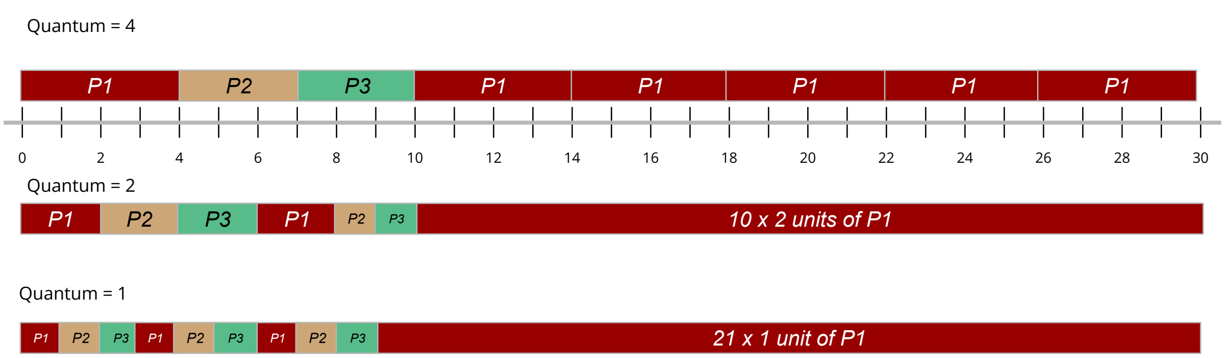


Round Robin

- The OS sets a fixed time quantum (in millisecs) for all the processes to use the CPU
  - The OS sets a timer (in hardware)
  - Processes with CPU burst > quantum will be preempted (by the timer interrupt) and placed back to the ready queue
  - Processes with CPU burst < quantum will continue to do I/O (use its I/O burst)
- CPU-bound processes will likely **use up all the assigned quantum time**
- I/O-bound processes will use only **a fraction of the quantum time** and then blocked for I/O

RR Example

P1: 24 units   P2: 3 units   P3: 3 units



Round Robin Quantum Time

- Too short: too much overhead for context switching
  - Quantum time should be relatively large compared to context switch time
- Too long: RR behaves like FCFS
- Perform better for interactive systems (ex: your EOS GUI sessions)
  - Interactive sessions: short CPU bursts, and long I/O bursts
  - Interactive sessions are able to finish all their CPU burst without being preempted
- ***However***, in a mixed system
  - I/O-bound processes will suffer (blocked I/O queue most of the time).
    - Quantum time is <<< typical I/O time
  - CPU-bound processes will monopolize the CPU (in the Ready Q most of the time, possibly ahead of I/O-bound processes)

Priority-Based Scheduling
(must be preemptive)

Multilevel Queue (1)

- Mixing CPU-intensive and I/O intensive processes in one queue does not seem to be a good idea
- Use **several** ready queues
  - *assign different priority levels to the queues*
- Assign user processes *permanently* to one of the ready queues
- Each queue may run its own scheduling algorithm

Multilevel Queue (2)

- How to “schedule” the queues (which queue to select processes from)?
  - Fixed-priority (**preemptive**) scheduling
    - dispatch processes from a lower priority queue when none can be dispatched from a higher level ones)
  - Time-sliced among the queues
    - Assign a certain “time slice” to each queue
    - Continue to dispatch from one queue until time slice for one queue expires, then move on to the lower queue

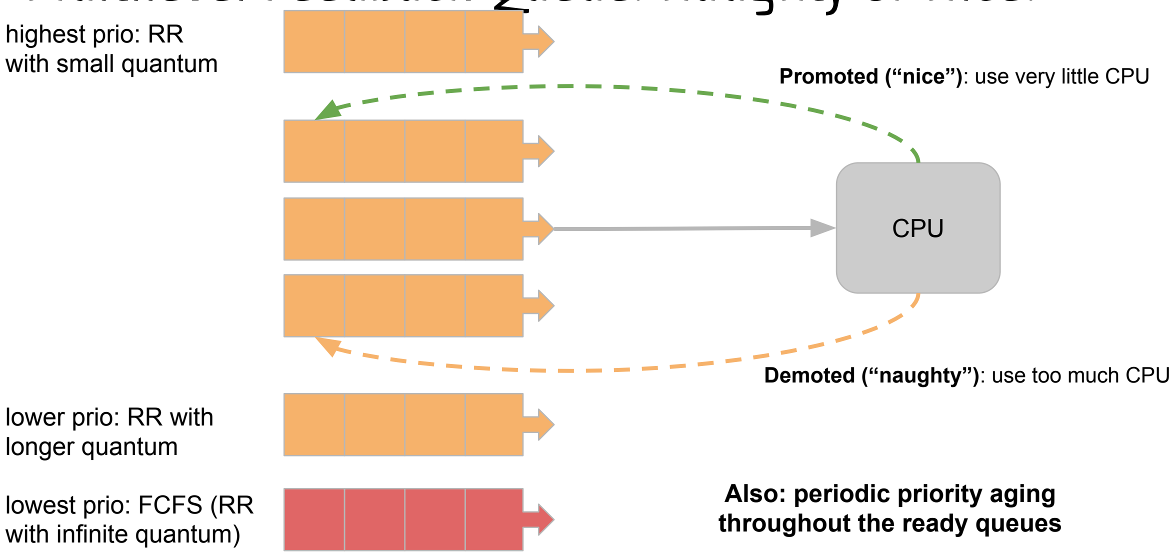
Multilevel Feedback Queue Scheduling

- A variant that combines Round Robin (RR) and Multilevel Queue (MQ)
- Unlike MQ**, MFQ allows processes to migrate** among different queues
  - **Promoted** to a queue of higher priority
  - **Demoted** to a queue of lower priority
- Longer RR quantum time for lower priority queues
- Shorter RR quantum time for higher priority queues

Used in Windows NT

Multilevel Feedback Queue: Naughty or Nice?

Multilevel Feedback Queue: Naughty or Nice?



Multilevel Feedback Queue Scheduling

- Promote processes that use too little CPU time (move them to a higher priority queue)
- Penalize processes that use too much CPU time (move them to a lower priority queue)
- CPU intensive processes will *eventually demote* into the **lowest prio Q**
- I/O intensive processes will *eventually promote* to the **highest prio Q**
- **Priority aging: avoids starvation**

Priority Scheduling

- Select processes with higher priority first (before any other processes)
- **Priority scheduler is ALWAYS preemptive**. Why?
- Two opposing interpretations of “priority numbers”
  - Lower numbers mean higher priority. (like TODO list)
  - Higher numbers mean higher priority. (like GPA)
- Risk: Processes with lower priority may starve
  - Solution: apply priority aging to avoid starvation (elevate priority level periodically)

Thread Scheduling

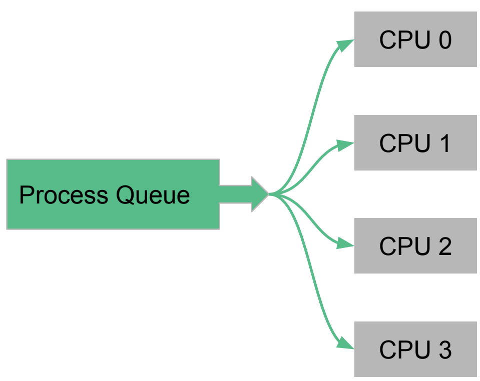
- KLTs are **scheduled** by the OS
- ULTs are **managed** by the thread library
- ULTs must first be mapped to a KLT before it can run on a CPU
- Mapping Models
  - One-to-One => All threads in a process are scheduled by the OS
  - Many-to-One or Many-to-Few => Not all threads are schedulable by the OS

Multiprocessor Scheduling

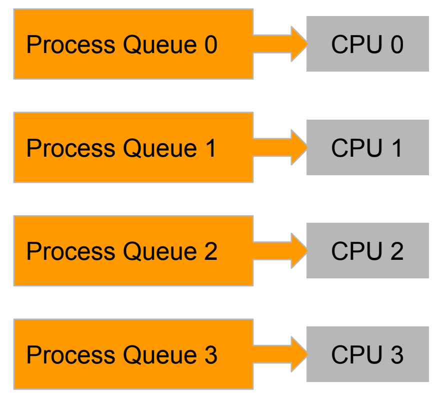
Multiprocessor Scheduling Issues

- Where to run the OS code?
  - Asymmetric Multiprocessing: OS code runs only on specific CPUs
  - Symmetric Multiprocessing: OS code can run on any CPUs
- Load balancing, Process Migration, and Processor Affinity
  - Load balancing requires process migrations from one CPU to another
  - Costly Cache Repopulation
  - Processor Affinity: ability to bind a process to a particular CPU (in a multi CPU system)
  - Conflicting objectives between load balancing and processor affinity

Global vs. Partitioned

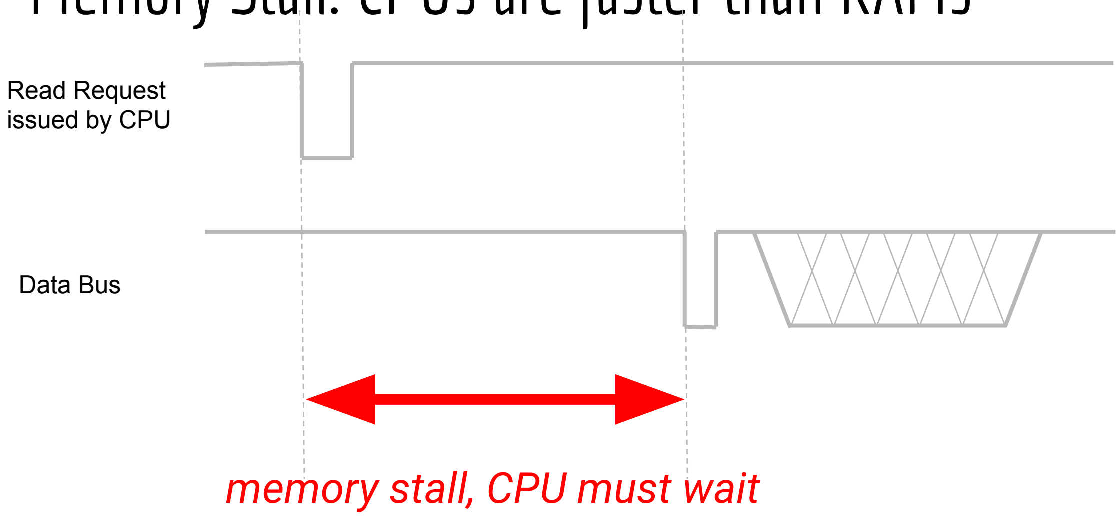


Global: Shared Process Queue



Partitioned: Dedicated Process Queues

Memory Stall: CPUs are faster than RAMs



Multicore Scheduling Issues

Hardware Features (but also scheduling issues)

- An N-core CPU *appears to be* N separate CPUs to the OS
- To keep CPU cores busy during a memory stall, CPU architects design **hardware multithreading**. (Intel Hyper-Threading technology)
  - Each core now appears as TWO separate CPUs to the OS

Real-Time Scheduling:
Scheduling with Time Constraints

Goal: make the OS respond within a given time limit

Time Constraints on Tasks

- **Start time**: tasks must begin at or no later than a specific time instance
- **End time (*deadline*)**: tasks must complete at or no later than a specific time instance
- **Periodic**: tasks must repeat at a specific rate
- Examples
  - Sustain "continuous" audio/video stream (YouTube streaming, Zoom video streaming) *Periodic and deadline constraints*
  - Automotive control: anti-lock brakes *Start, End (deadline), and Periodic constraints*
  - Aircraft Control: fly-by-wire *Start time constraints*
  - Autonomous Car: vehicle response to sensory input *Start time constraints*

Real-Time Systems: Categories

- Soft RT Systems
  - Guarantee that critical processes will be given preference over non-critical ones
  - But do not guarantee time constraints
- Hard RT Systems
  - Must guarantee that tasks be serviced by its deadline

Real-Time Schedulers

- Objective: schedule tasks to minimize latency
- Required features of an RTS
  - Priority-based
  - Preemptive (when a higher priority process becomes available, lower priority processes are preempted from the CPU)
- Priority-based + Preemptive RTS only guarantee *soft real-time*
- RT Scheduling Algorithms
  - **Rate-Monotonic Scheduler**
  - **Earliest-Deadline First Scheduler**

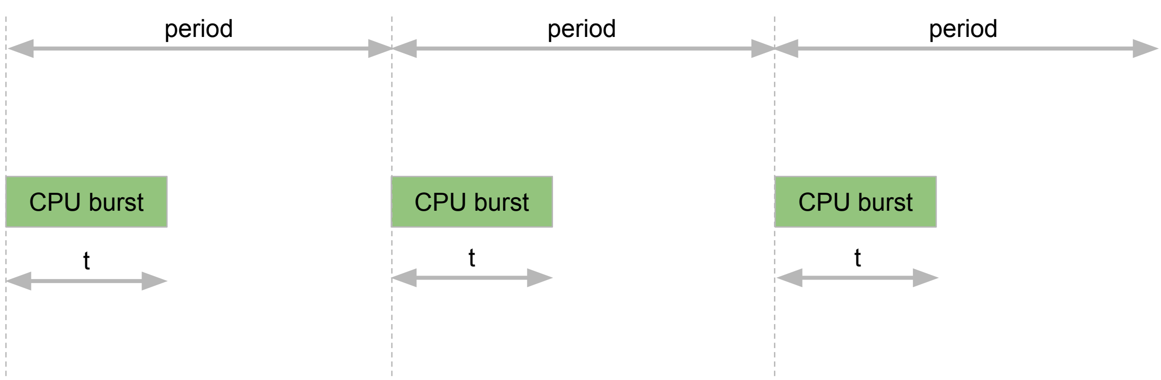
Rate-Monotonic Schedulers (RMS)

Rate: how frequently a periodic task uses the CPU

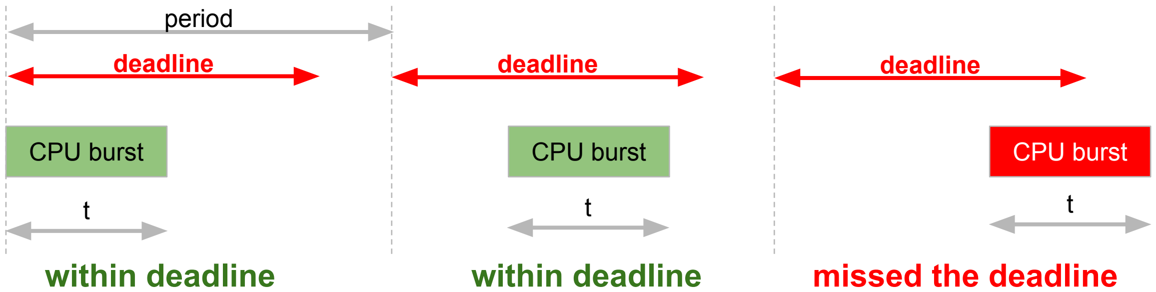
Model for RMS

- Tasks/processes are assumed to be *periodic with a fixed CPU burst*
  - CPU burst and “I/O bursts” are two fixed values
- Time Constraint (Deadline): Processes must be **completed** within a given time limit
- Each process is parameterized by these three numbers
  - (CPU burst, periodic interval, time constraint/deadline)

IDEAL Execution Pattern of a **Periodic** Task



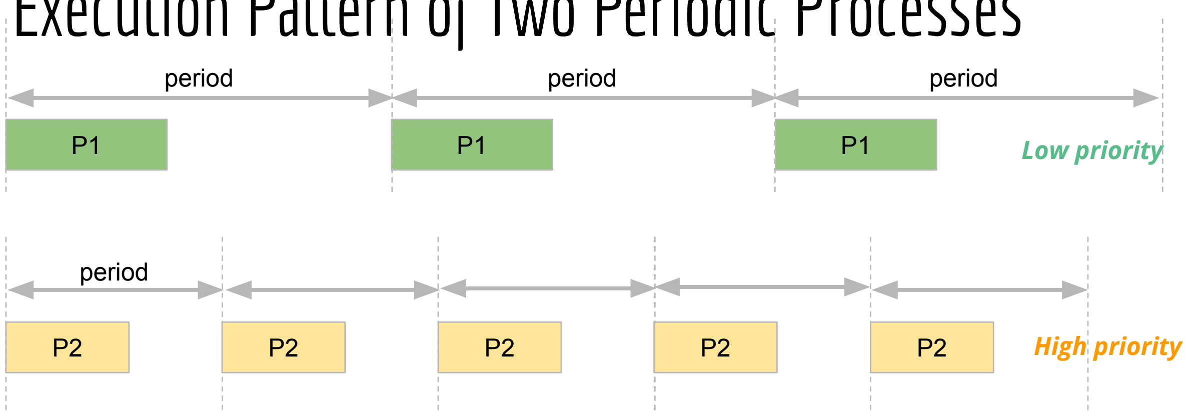
Periodic Tasks with deadline



The above timing diagram shows an IDEAL case where the task runs without being preempted.

In actual RT scheduling algorithms, ***processes are preempted and CPU bursts may split into several chunks of execution***

Execution Pattern of Two Periodic Processes



Rate-Monotonic Scheduler:

shorter period ⇒ more frequent use of CPU ⇒ higher priority

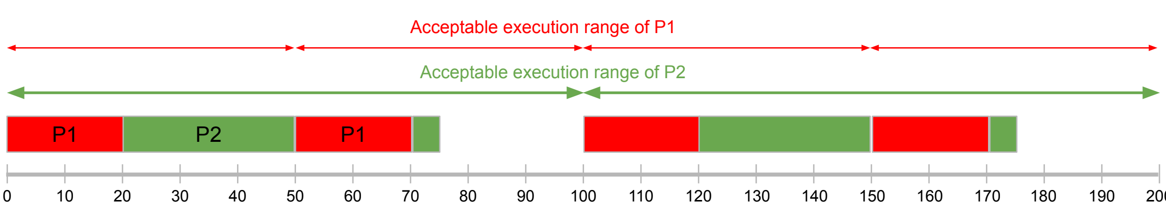
longer period ⇒ less frequent CPU use ⇒ lower priority

Rate-Monotonic Scheduling Algorithm

- Assign **static** priority, inverse of the task period
  - Longer period (lower rate of CPU use) => lower priority
  - Shorter period (higher of CPU use) => higher priority
- Preemptive scheduling algorithm
  - Tasks with lower priority are preempted (from the CPU) if a higher priority task becomes ready/available to run

RMS Examples

Process	Period	Deadline	CPU burst	% of CPU Utilization (CPU/Period)
P1	50	50	20	40% (= 20/50)
P2	100	100	35	35% (= 35/100)



P2 was interrupted at 50 by P1 (higher priority),
P2 continues with its remaining 5 units of CPU at 70

RMS Examples

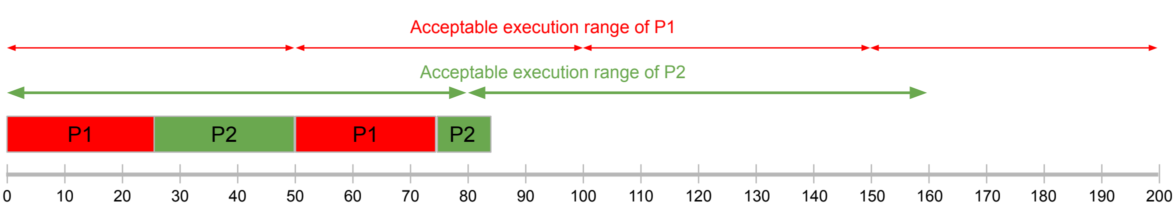
Process	Period	Deadline	CPU burst	% of CPU Utilization (CPU/Period)
P1	50	50	40	80% (= 40/50)
P2	100	100	35	35% (= 35/100)

Total CPU utilization = 80% + 35% = 115% > 100%

Impossible to schedule the two periodic tasks!

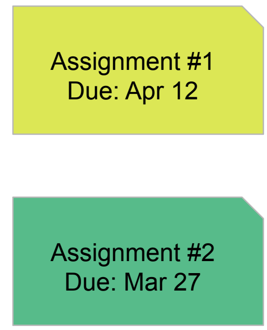
RMS Examples

Process	Period	Deadline	CPU burst	% of CPU Utilization (CPU/Period)
P1	50	50	25	50% (= 25/50)
P2	80	80	35	43.75% (= 35/80)



P2 was interrupted at 50 by P1 (higher priority),
P2 missed the deadline at 80

Earliest Deadline First

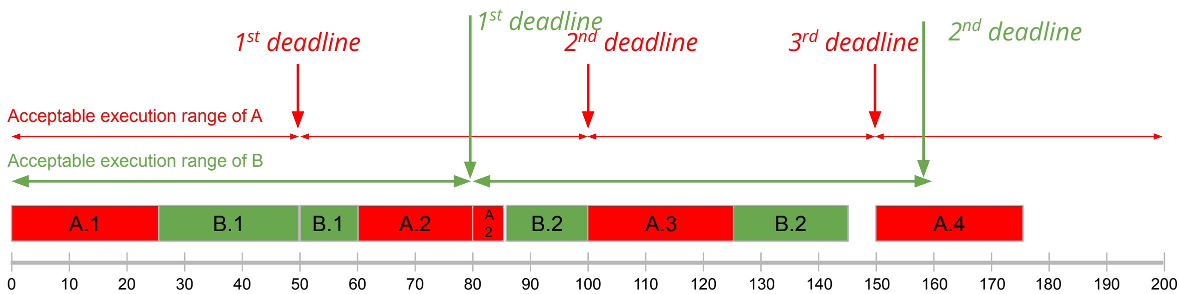


Earliest-Deadline First (EDF)

- Assign priority **dynamically** based on the **next deadline**
  - Earlier deadline => higher priority
  - Later deadline => lower priority
- Theoretically optimal algorithm (CPU utilization may be close to 100%)
- *The algorithm also works for **non-periodic** processes and **variable** CPU bursts, but the **next deadline** must be announced to the system*

EDF Examples

Process	Period	Deadline	CPU burst
A	50	50	25
B	80	80	35



At time 50:
B was preempted (due to 2nd arrival of A).
nextDL(A) = 100, nextDL(B) = 80
B has higher priority (resume)

At time 60: only A is available
At time 80:
A was preempted (due to 2nd arrival of B).
nextDL(A) = 100, nextDL(B) = 160
A has higher priority (resume)

At time 100:
B was preempted (due to 3nd arrival of A).
nextDL(A) = 150, nextDL(B) = 160
A has higher priority (dispatched)

Linux Scheduling

O(1) Scheduler in Linux 2.6.8

- Data structures: array of linked lists (`runqueue`)
  - arr[0] is the list of highest priority processes
  - arr[N-1] is the list of lowest priority processes
  - One linked list holds processes of the same priority
- Two copies of runqueue: `active` and `expired`
  - When a process did not use all its quantum, half of the unused quantum is added to the next round (and the process stays in the "`active`" runqueue)
  - When a process used all its quantum, it is moved from `active` to `expire`
  - When the `active` run queue is "empty", swap active and expired

Linux 2.6.23: Completely Fair Scheduler

- Fairness: with N processes in the system, each process should receive 1/N of the CPU time
- On an ***ideal multitasking CPU***, these N processes would run in parallel at equal speed (1/N of the CPU speed)
  - Impossible to run them in parallel on real CPUs
- Approach for implementing a fair scheduler
  - **Virtual runtime**: the amount of CPU that a process **should have gotten on an ideal multitasking CPU**

Linux: Completely Fair Scheduler

- Similar idea as the Multilevel Feedback Queue
  - MFQ: promote / demote priority levels based on process execution pattern
  - CFS: adjust virtual runtime based on process execution pattern
- Nice value: `-20` (high priority) to 0 (normal priority) to `+19` (low priority)
  - Default nice value is zero (normal priority)
  - Lower number means high priority
- Preemptive scheduler (because of priority based algorithm)
- Processes are organized into a Red-Black tree (with virtual runtime as key)

Linux CFS: separating "naughty" from "nice"

- Runtime decay rate calculated from the process nice value
  - Nice < 0 (high priority): decay rate < 1.0
  - Nice = 0 (normal priority): no decay
  - Nice > 0 (low priority): decay rate > 1.0
- Virtual runtime vs. Physical runtime
  - Physical runtime: total CPU time that **has been used** by this processes
  - Virtual runtime: physical runtime after being adjusted by the **decay rate**
    - Lower priority processes: virtual runtime > physical runtime
    - Higher priority processes: virtual runtime < physical runtime
    - Normal priority processes: virtual runtime = physical runtime
- CFS selects the process with the **smallest virtual runtime**

CFS: Mix of I/O bound and CPU bound jobs

- Execution patterns
  - I/O bound jobs use only a **fraction of its quantum** (before it blocks for I/O)
  - CPU bound jobs will exhaust its **entire quantum** (before it is preempted)
- Virtual runtime of I/O bound jobs will be smaller than CPU bound jobs
  - I/O bound jobs are dispatched ahead of CPU bound jobs
- Starvation is avoided
  - Processes that have not used CPU will have lower physical runtime value (hence lower virtual runtime)

Linux CFS: Real-Time Scheduling

- Priority range 0-139 (static priority number)
  - Recall: lower number means higher priority in Linux
- Real-time tasks: 0-99
  - Soft Realtime Scheduler (does not guarantee deadline)
  - Can be scheduled using either RR or **preemptive FCFS**
  - POSIX Thread options: SCHED_RR or SCHED_FIFO
  - **Preemptive FCFS:** a variant of FCFS where preemption allowed by higher prio processes
- Normal tasks: 100-139
  - Normal tasks with nice value -20 are assigned priority value 100
  - Normal tasks with nice value +19 are assigned priority value 139

Linux: Static Priority vs. Dynamic Priority

- Static priority [0,139]
- Dynamic priority = func (static_priority, average_sleep_time)
  - Adjustment range: [-5, +5]
  - Sleeping too much => increase dynamic priority
  - Working too much => decrease dynamic priority

Windows Scheduler

- 32 priority levels
  - Smaller number means lower priority
  - Levels 1-15 for "variable class"
  - Levels 16-31 for "real-time class"
- Six Base Priority Classes
  - IDLE (4), BELOW_NORMAL(6), NORMAL (8), ABOVE_NORMAL (10), HIGH_PRIORITY (13), REALTIME (24)
- Seven Relative Priority Classes
  - IDLE, LOWEST, BELOW_NORMAL, NORMAL, ABOVE_NORMAL, HIGHEST, TIME_CRITICAL
- Table 6.22 in textbook