Scheduling

CS3026 Operating Systems

Lecture 14

Priority Scheduling

Priority Scheduling

- Processes are scheduled according to a priority
- The CPU is allocated to the process with the highest priority
- There are pre-emptive and non-pre-emptive priority scheduling algorithms
- Policies
  - Shortest Job First (non-preemptive)
  - Shortest Remaining Time (pre-emptive)
  - Fixed Priority Scheduling (pre-emptive)
  - Multilevel Feedback Queue

Priority Scheduling

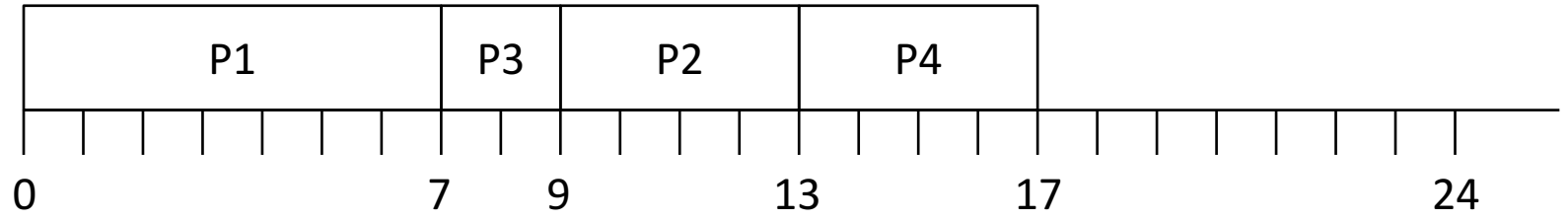
- Static Priorities
  - Priority specified at creation of a process
  - Used in real-time systems
- Dynamic Priorities
  - We analyse past performance to predict the future
  - Priority adapts to process behaviour (feedback scheduling)

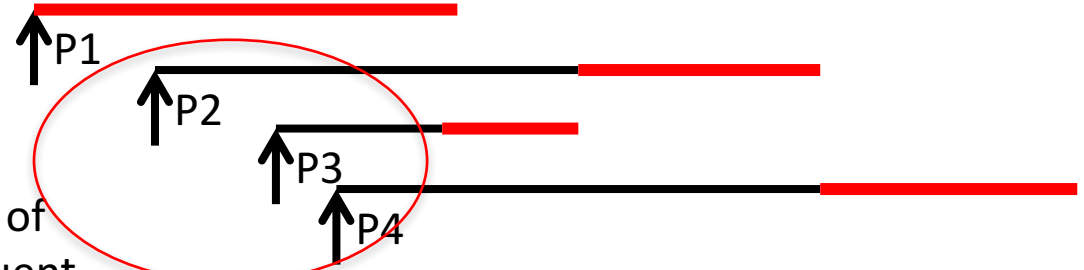
Shortest Job First(SJF)

- Non-preemptive scheduling policy
  - A burst of CPU time is not interrupted by the OS
- Also called
  - “Shortest Job Next (SJN)” or “Shortest process next (SPN)”
- Tries to always schedule the process with the shortest expected processing time next
  - Make it dependent on a predicted next CPU burst time
- SJF is optimal:
  - Minimum average waiting time for a given set of processes
  - Can be regarded as a benchmark for other scheduling algorithms
  - Assumes that we know the processing time / burst time of a process in advance
- Problem:
  - How do we know which process will have the shortest processing time?

Shortest Job First(SJF)

Process (Job)	Arrival Time	CPU Time (Burst)
P1	0	7
P2	2	4
P3	4	2
P4	5	4

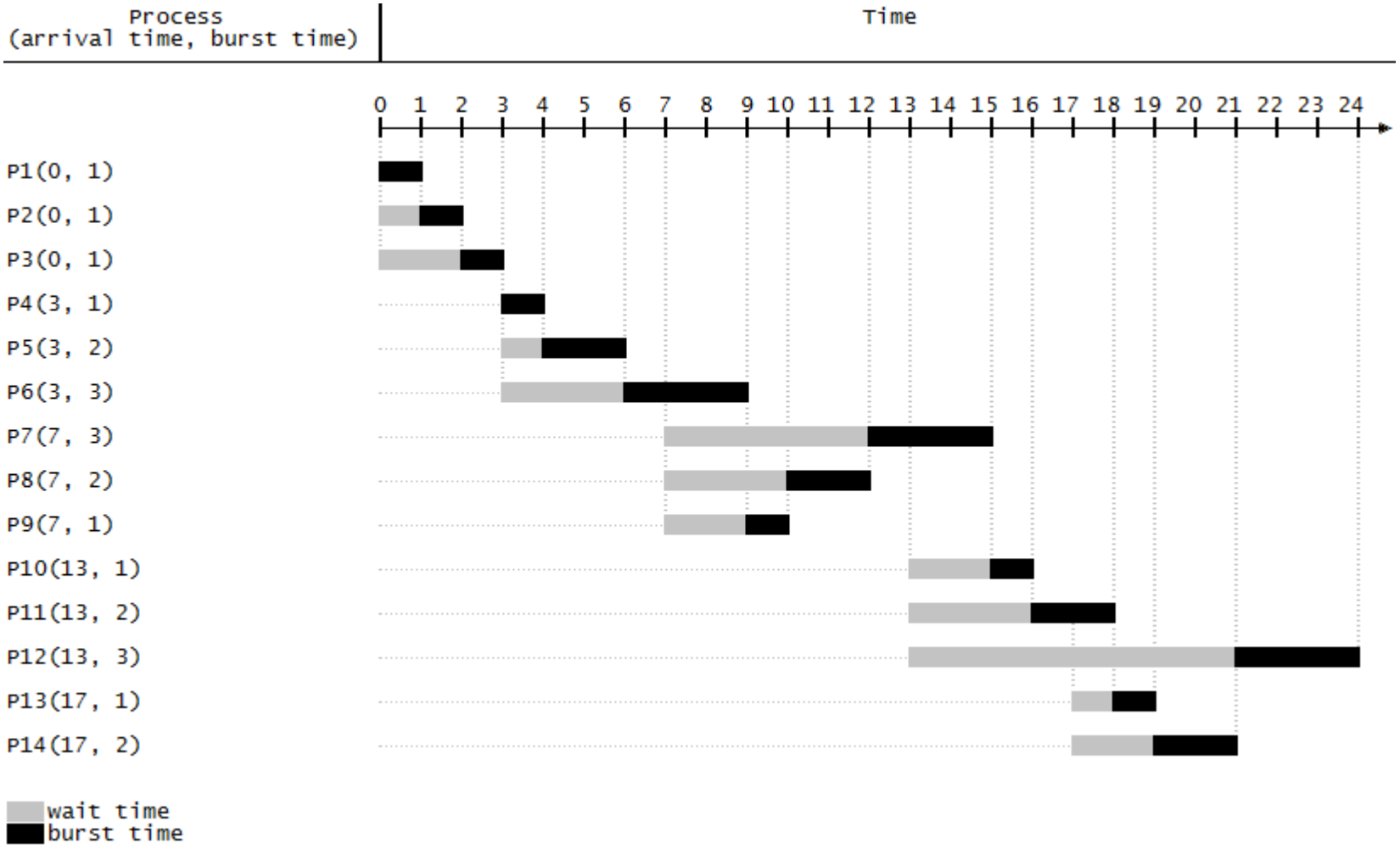




Arrivals of subsequent processes

Schedule sequence: [P1, P3, P2, P4]

Average Waiting Time: (0 + 7-4 + 9-2 + 13-5)/4 = 18/4 = 4.5



https://en.wikipedia.org/wiki/Shortest_job_next#/media/File:Shortest_job_first.png

Shortest Job First(SJF)

- Difficulty
  - How do we know which job has the shortest processing time?
- Necessity to estimate the required CPU processing time of each process
  - For batch jobs: past experience, specified manually
- Predict the length of the next CPU burst
  - For interactive jobs: analyse past bursts of CPU time, calculate exponential average

SJF for Interactive Systems

- We use the past to predict the future
  - Record a running average of each past burst time of interactive job

$$S_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

  - Where:
    - n = number of bursts so far
    - Si = predicted burst time for ith occurrence of processor burst
    - Ti = actual processor execution time for ith burst

Predicting Burst Time

- We use the past to predict the future
- Use exponential averaging
  - We record the duration of CPU bursts for all processes
  - We want to give greater weight to more recent burst instances, using a constant weighting factor α :
  - Where:

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n$$

    - n = number of bursts so far
    - S_{n+1} = predicted value for next CPU burst
    - T_n = actual execution time for n-th burst
    - α = constant weighting factor, ($0 < \alpha < 1$)

Predicting Burst Time

- Use exponential averaging

$$S_1 = T_1$$
$$S_{n+1} = \alpha T_n + (1-\alpha) S_n$$

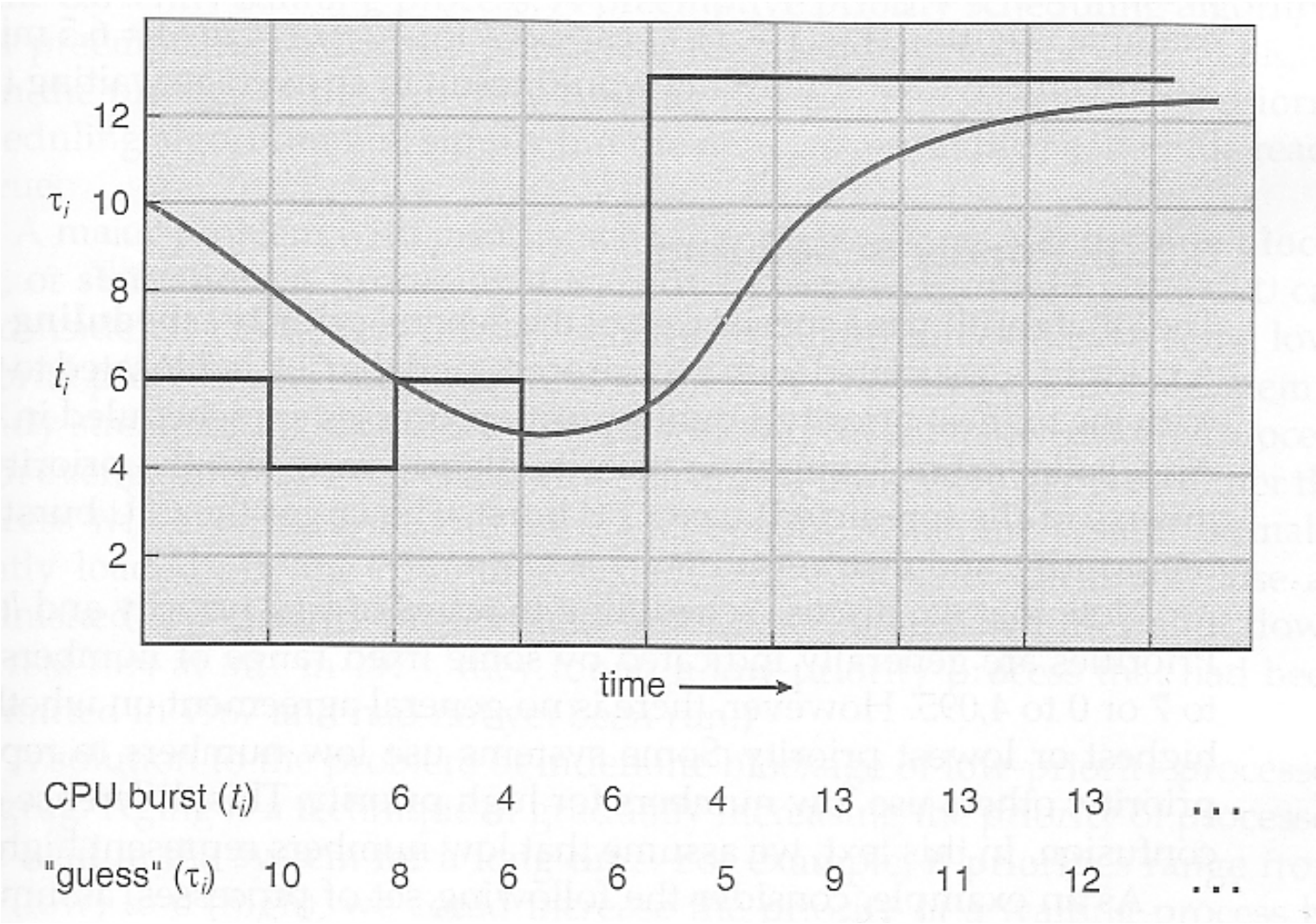
$$S_{n+1} = \alpha T_n + (1-\alpha)\alpha T_{n-1} + \ldots + (1-\alpha)^i \alpha T_{N-i} + \ldots + (1-\alpha)^n S_1$$

  - We want to give greater weight to more recent burst instances

- Example:
  - $\alpha = 0.8$

$$S_{n+1} = 0.8T_n + 0.16T_{n-1} + 0.032T_{N-2} + 0.0064T_{n-3} \ldots + (0.2)^n S_1$$

CPU burst Prediction vs Actual Time



CPU burst (t_i)		6	4	6	4	13	13	13	...
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

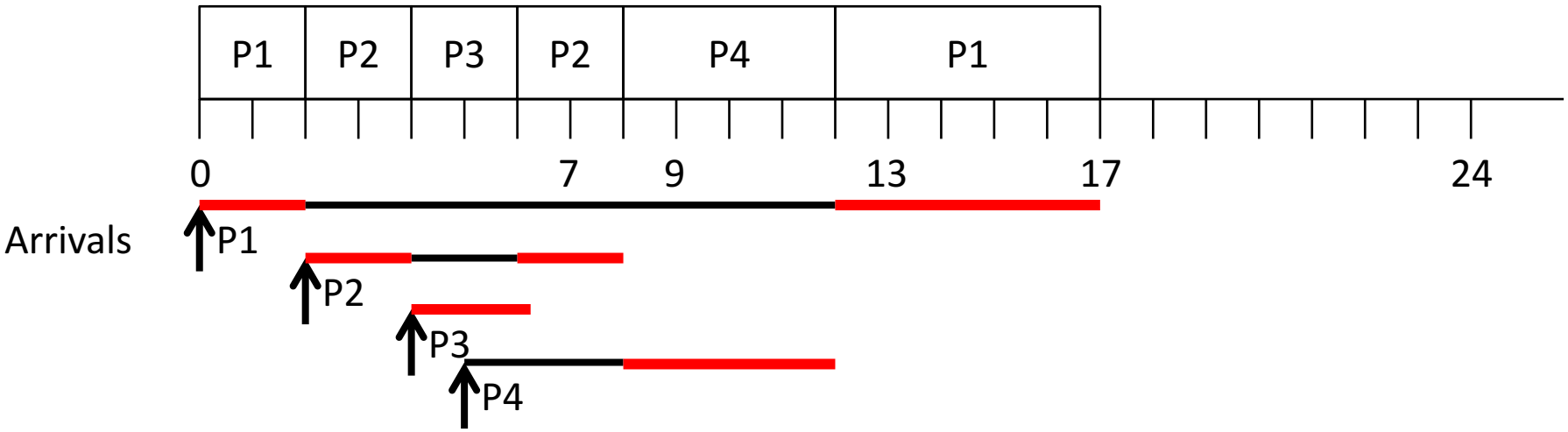
Shortest Remaining Time (SRT)

- Preemptive version of the SJF scheduling policy
  - Scheduler tries to select the process whose remaining run time is the shortest
- Same selection function as SJF
- Processes currently executing on CPU are preempted if a job with shorter estimated CPU time becomes ready
- Better turnaround performance
  - Shorter job is given immediate preference over a longer running job
  - Reduces waiting times

Shortest Remaining Time (SRT)

Process	Arrival Time	Total CPU Time
P1	0	7
P2	2	4
P3	4	2
P4	5	4

- SRT Schedule



- Average Waiting Time: (10 + 2 + 0 + 3)/4 = 3.75

Starvation in Priority Scheduling

- Starvation: Low-priority processes may never execute
- Prioritised processes will jump to the head of the queue
- Problem of SJF
  - Possibility of starvation for processes with longer execution time
- Solution to Starvation
  - Introduce concept of "**Aging**": as time progresses, increase the priority of processes

Highest Response Ratio Next (HRRN)

- Is a non-preemptive scheduling policy based on a ratio between time spent waiting and expected service time of a process

$$Ratio = \frac{time\ spent\ waiting + expected\ service\ time}{expected\ service\ time}$$

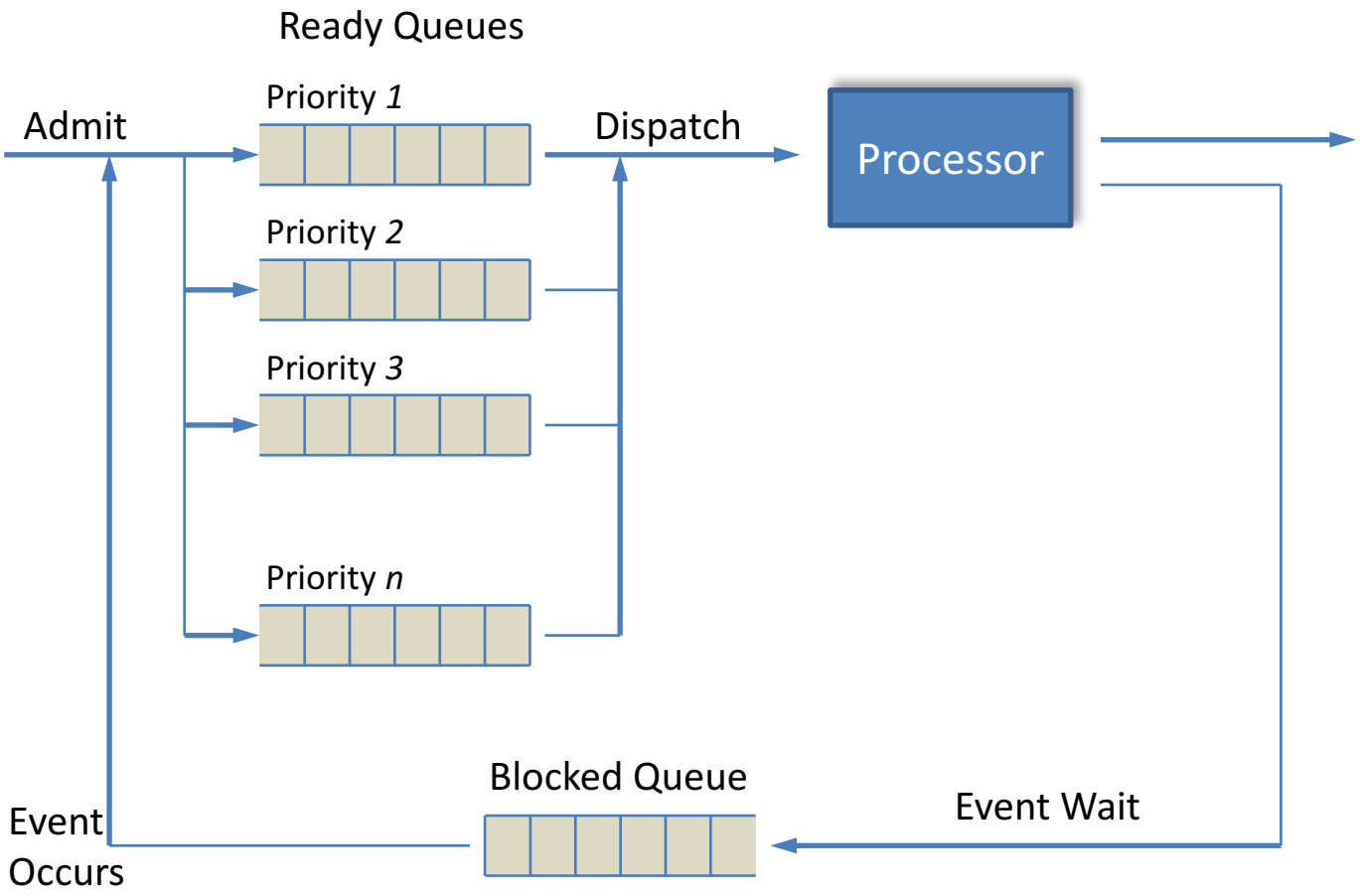
- Expected service time has to be estimated as for the SJF scheduling policy
- Avoids starvation with aging
  - Accounts for the age of the process
  - While shorter jobs are favoured, aging eventually increases the ratio for longer running processes that pushes them past the shorter processes

Multilevel Queue Scheduling

Multilevel Queue Scheduling

- Ready queue is partitioned into separate queues
- We can distinguish between different types of processes
  - “foreground”: interactive processes, direct user interaction, need high responsiveness
  - “background”: batch processes, no user interaction, need high throughput, long CPU bursts
- We can choose different scheduling policies based on type of process for each queue
- Use n Ready queues:
  - each queue has different priority
  - is managed according to a particular scheduling policy

Multilevel Queue Scheduling



Multilevel Queue Scheduling

- Each queue has its own scheduling algorithm
  - Foreground processes: use queue managed according to RR, higher priority
  - Background processes: use queue managed according to FCFS, lower priority
- Scheduling must be done across queues
  - Fixed priority scheduling
    - Serve all from foreground, then from background
    - Possibility of starvation
  - Time slicing
    - Each queue gets a certain amount of CPU time which it can schedule amongst its processes
    - E.g.: 80% foreground with RR, 20% background with FCFS

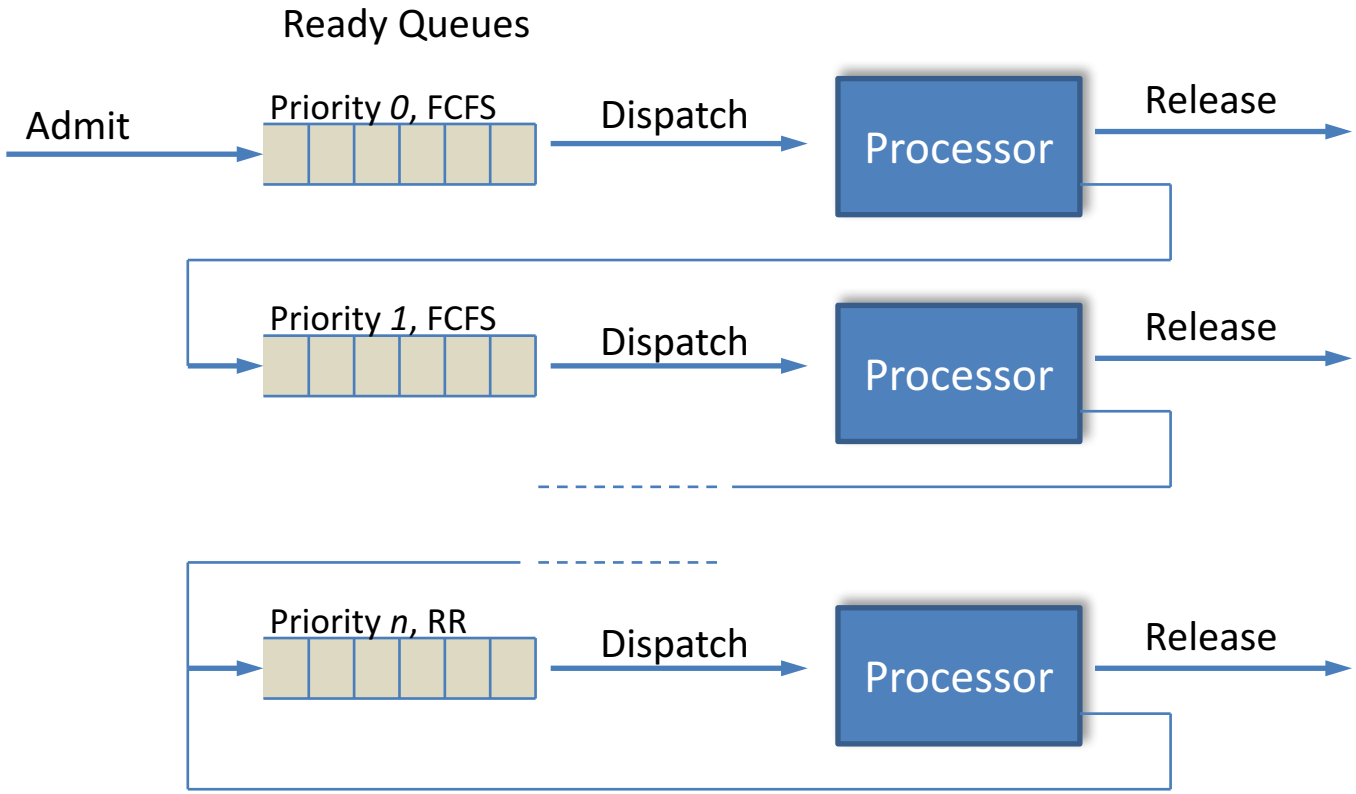
Feedback Scheduling

- Selection procedure:
  - “Penalize processes that have been running longer”
  - A process is downgraded according to “CPU time consumed so far”
    - The more processing time so far, the lower its priority
- Decision mode: is pre-emptive scheduling
- Multiple queues
  - one per priority
- Starvation may occur

Multilevel Feedback Queue

- A multilevel feedback queue scheduler is defined by the following parameters
  - Number of queues
  - Scheduling algorithms for each queue
  - Method used to determine when to promote or demote a process (change to a different queue)
  - Method used to determine which queue a process will enter when that process needs service

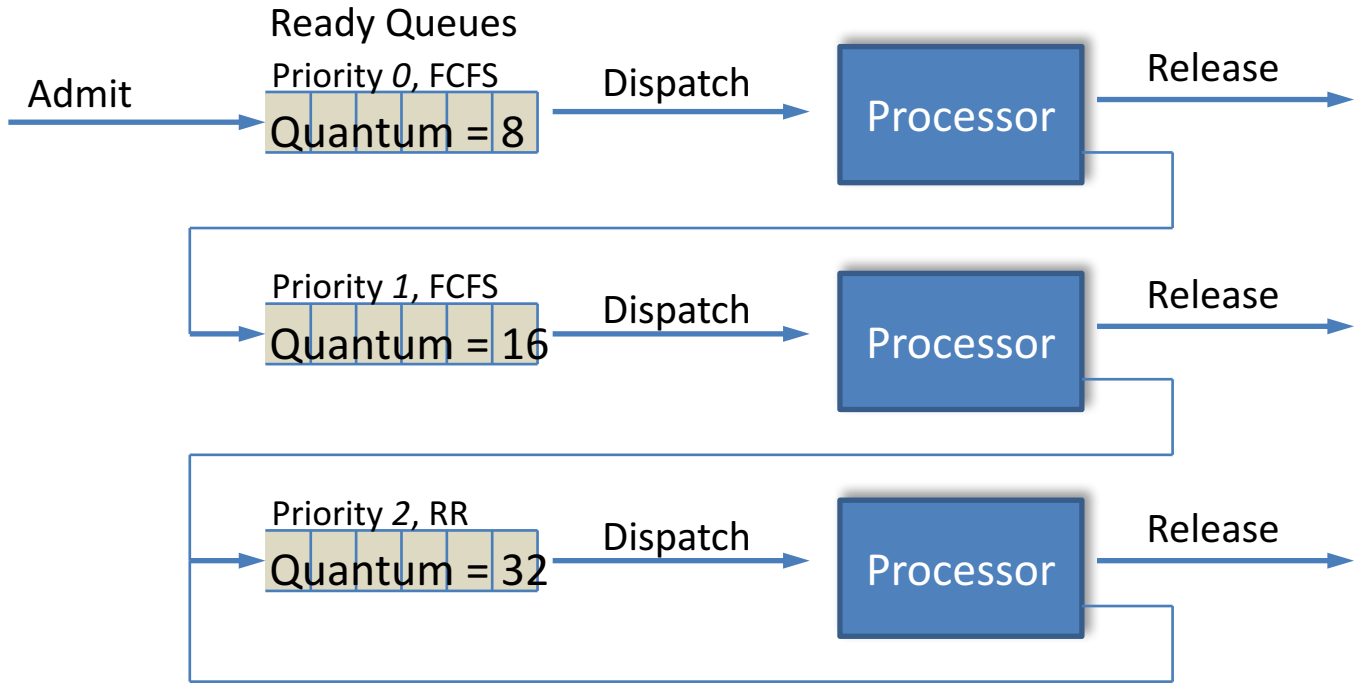
Multilevel Feedback Queues



Multilevel Feedback Queues

- Problem
  - Long processes are demoted to lowest-priority queue, may need very long time to finish
  - Danger of starvation:
    - if many short processes enter system, higher-priority queues are served first
    - Lower-priority queues may never be served
- Counter measure
  - Introduce different time quantums for queues
    - Lower-priority queue is allowed 2^i time units for its quantum then the previous higher-priority queue

Multilevel Feedback Queues
Different Quantum Time



- Effect:
  - Long processes get more CPU time to recover from their demotion
  - Long processes get more CPU time, when they finally get it

Multilevel Feedback Queues Different Quantum Time

- When process arrives
  - put into highest priority queue
- After each pre-emption
  - Moved to lower-priority queue
- For all queues:
  - Scheduling procedure: RR for the lowest-priority queue, FCFS for all other queues
  - Length of time quantum is short for high priority, and longer for low priority
    - Opportunity for lower-priority job to occupy CPU longer
- Effect:
  - Processes trickle down the priority queues
  - Short processes stop earlier in this descent
  - Long processes get more CPU time, when they get it

Example

- Three queues
  - Q_0 – FCFS with time quantum 8 milliseconds
  - Q_1 – FCFS time quantum 16 milliseconds
  - Q_2 – RR, time quantum 32 milliseconds
- Scheduling
  - A new process enters queue Q_0, is added at the end of Q_0 (served FCFS)
    - When process gets CPU, it receives a time quantum of 8 milliseconds
    - If process executes longer than its time quantum, it is preempted and moved to Q_1
  - A process entering queue Q_1 is also added at the end of the queue (served FCFS)
    - When the process gets CPU, it receives a time quantum of 16 milliseconds
    - If process does not finish within its time quantum, it is preempted and moved to Q_2
  - At queue Q2, processes are scheduled according to RR and pre-empted after a time quantum of 32 Millisections

Lottery Scheduling

- Basic idea
  - Give processes “lottery tickets” for various resources, such as CPU time
  - For each scheduling decision, a lottery ticket is chosen at random and the process holding that ticket gets resource
  - For example:
    - Scheduler holds lottery 50 times per second
    - Each winner receives 20msec of CPU time

Lottery Scheduling

- Possible manipulations
  - Prioritisation
  - Ticket exchange
- Prioritising Processes
  - More important processes can be given extra tickets, increases their odds of winning
  - E.g.: 100 tickets, one process holds 20 of them, therefore a 20% chance of winning, in the long run will receive 20% of CPU time
- Ticket exchange
  - A client process may give tickets to its server to increase the server's chances of getting CPU time earlier, so it can service the client earlier (client "bribes" server)

Fair-Share Scheduling

- Fair-share scheduling takes process ownership into account
  - Recognizes the fact that processes are owned by users
  - Each user get's fair share of resources and not the single process, independent of the number of processes that currently run
- Applications, run by a user, may consist of a set of processes
  - User experience (responsiveness) depends on the application's overall performance
- Can be extended to user groups

Fair-Share Scheduling

- Scheduling of process is done on the basis of a combined preference / priority value
  - The priority of the process
  - Its recent CPU usage
  - The recent CPU usage of the user / group to which the process belongs
  - The weight of a group (its “importance” in getting CPU time)
- Calculate
  - A new priority for the next scheduling decision
  - Scheduling decisions will be distributed across groups according to their weight

Scheduling Algorithm Goals

- All categories
  - Fairness: give each process a fair share of the CPU
  - Policy enforcement: stated policy is carried out, e.g. Certain processes are of higher priority
  - Balance: distribute processing load evenly
- Batch systems
  - Throughput: maximise jobs per hour
  - Turnaround time: minimize time between submission and termination
  - Utilisation: keep the CPU busy at all times
- Interactive systems
  - Response time: respond to all interactive requests quickly
  - Proportionality: meet the user's expectations regarding responsiveness
- Real-time Systems
  - Meeting deadlines: e.g. I/O device generates data at a regular rate, process has to be scheduled in time to retrieve this data and avoid losing data
  - Predictability: process scheduling has to be highly regular and predictable, e.g. to avoid problems on multimedia systems – deteriorating sound quality during audio processing, audio / video jitter

Suitability of Scheduling Algorithms

- Batch processing systems
  - First-Come First-Served
  - Shortest Job First
  - Shortest Remaining Time Next
- Interactive Systems
  - Round-Robin Scheduling
  - Priority Scheduling
    - Shortest Process Next (pre-emptive version of Shortest Job First, make an estimate for the next processor burst based on previous observations)
    - Multiple Queues, one per priority
    - Multi-level feedback queue (process migration, different quantum per queue)
  - Lottery Scheduling
  - Fair-Share Scheduling

Realtime Scheduling

Real-Time Operating Systems

- Computations have to meet deadlines
- Used in situations where it is critical that a computer system reacts to events in time
  - E.g.: Control systems in power plants, Air traffic control, Telecommunication
- Events occur in “real time”
- The computer system must be able to keep up with them
  - Processes must be scheduled so that they are finished processing such an event before given deadlines
- Example:
  - A bit stream from a sensor has to be processed, loss of data has to be avoided (patient monitoring systems)
- Problem of latency

Scheduling in Real-Time Systems

- In real-time scheduling, time is essential
  - A process has to meet deadlines reliably
- What do we want to avoid:
  - Missing a Deadline
- A set of processes has to be scheduled so that all of them meet deadlines reliably
- How to order / prioritise them?
- How does the requirement of meeting deadlines and allow a system to reliably react to events impact on CPU utilisation?

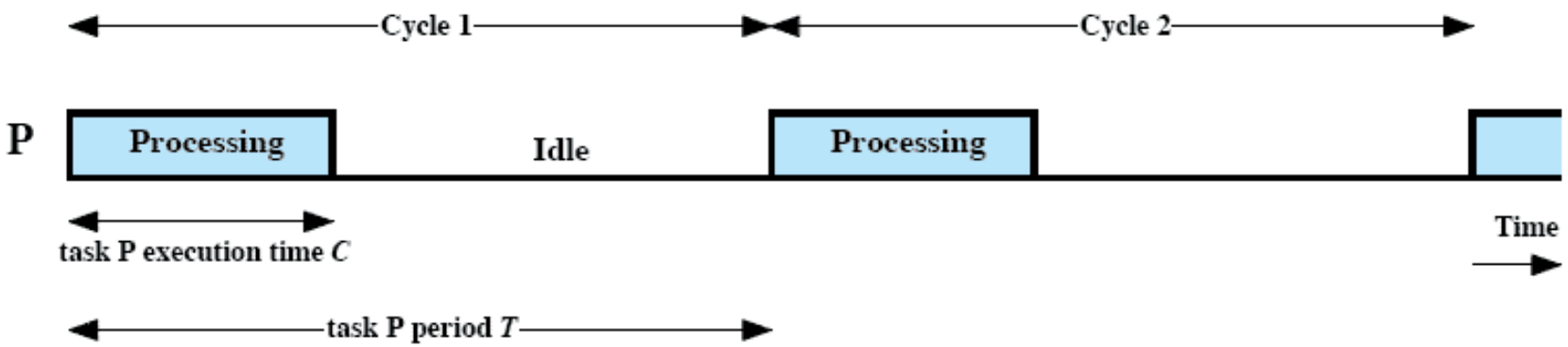
Hard and Soft Real-Time

- Hard Real-Time
  - There are absolute deadlines that always have to be met
  - If a deadline is missed, there may be unacceptable danger, damage or a fatal error in the system controlled by real-time processing
- Soft Real-Time
  - Each processing of an event has an associated deadline that is desirable, but not mandatory
  - System continues to operate, even if a deadline has not been met
    - Missing an occasional deadline for scheduling a process (e.g. missing sensor input) is tolerable, system is able to recover from that

Real-time Scheduling

- Scheduling for meeting deadlines
  - Hard real-time: deadlines are non-negotiable
  - Soft real-time: deadlines are specified, not mandatory
- Occurrence of process scheduling
  - Periodic vs. Aperiodic processes
- Priorities
  - Priorities fixed or dynamic
- Schedulability
  - Test, whether a set of scheduled tasks will all finish before a deadline

Periodic Events



- Events that occur at regular intervals
  - Events occur exactly *T* time units apart
    - Frequency or rate of occurrence: 1/T
- Processing time to handle such an event has to take place within these *T* time units
  - CPU execution time *C*: *C <= T*

Aperiodic Events

- Aperiodic events
  - Events occur unpredictably
- Has a deadline by which a computer system has to react
  - Processing for handling such an event has to be finished by deadline

Real-Time Scheduling Algorithms

- Static table-driven approaches:
  - A static analysis is performed “offline”
  - Result is a schedule, determines at runtime, when a particular task must be started
  - Applicable to periodic task occurrence
- Analysis uses
  - Arrival time, Execution time, Ending deadline, Relative priority of a task
- Scheduler tries to develop a schedule that meets the requirements of all tasks involved
  - Inflexible/static: any changes to any task requirements requires a rescheduling
- Schedules according to the deadline of a task
  - Ordering according to “earliest deadline first”

Earliest Deadline First (EDF)

- Selection function:
  - Schedules process with earliest completion deadline
- Is a preemptive scheduling policy
- Minimises number of deadline misses

Real-Time Scheduling Algorithms

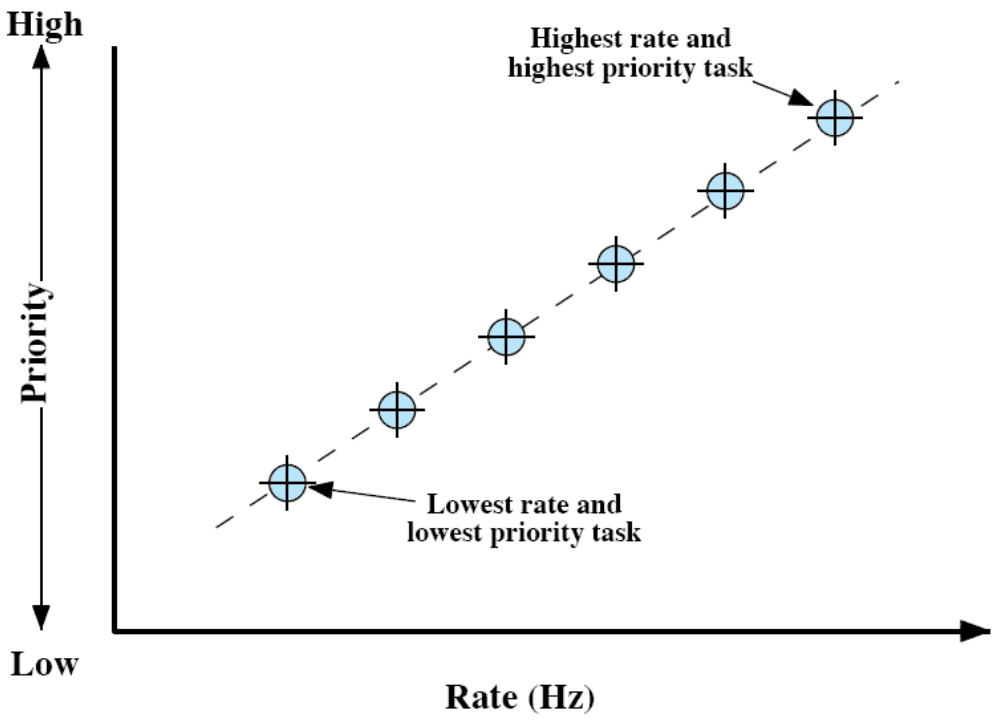
- Static priority-driven preemptive approaches:
  - A static analysis is performed “offline”
  - No schedule created
  - Static analysis is used to assign priorities to processes
    - Rate-monotonic Algorithm
  - A traditional priority-driven preemptive scheduler can then be used

Rate Monotonic Scheduling (RMS)

- Is a periodic task scheduling method
- Selection function:
  - Shorter period <==> higher priority
- Is preemptive scheduling policy
- Assign a fixed, unchanging priority to each process based on the rate / frequency of occurrence of the handled event
  - The higher the rate (shorter time), the higher the priority

Rate-Monotonic Scheduling

- RMS assigns priorities to tasks on the basis of their arrival rate (period)
  - the highest-priority task is the one with the shortest period
- If we plot the priority of tasks as a function of their rate, the result is a monotonically increasing function



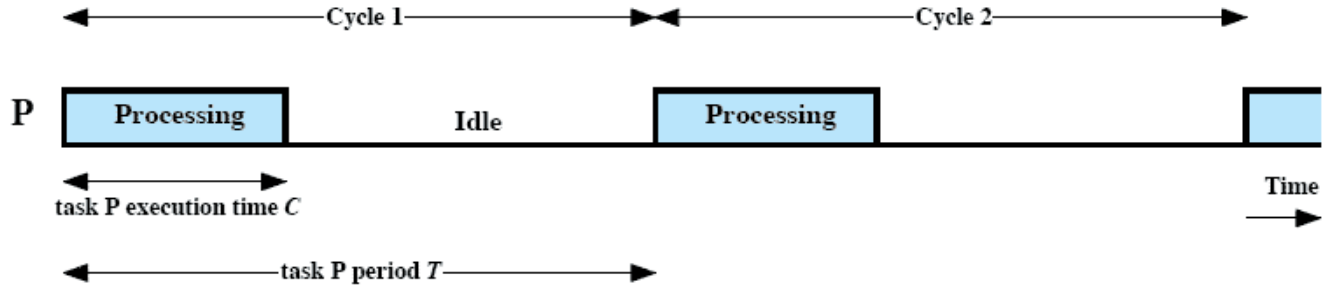
Effectiveness of a Periodic Scheduling Algorithm

- A periodic scheduling algorithm is effective when it guarantees that all hard deadlines are met
  - In order to meet all deadlines, schedulability must be guaranteed

Real-time Scheduling: Schedulability

- Schedulability analysis:
  - Given a set of tasks – which schedule guarantees that all of them meet their deadlines?
- Schedulability analysis can be:
  - Static: is done “offline” before any tasks starts executing
  - Dynamic: continuous analysis during the execution of a schedule
- Schedulability analysis may result in a schedule or “plan” according to which processes are dispatched

Effectiveness of a Periodic Scheduling Algorithm



- Criterion for Schedulability
  - Given
    - ***n*** periodic events, each event is handled by a process
    - T_i : The period of time for an event ***i*** to re-occur
      - This is the maximum time a system has after each occurrence for scheduling the corresponding process and execute it
    - C_i : The actual CPU time a process responsible for a particular event ***i*** needs
  - Then the following condition must hold:
    - The total utilization of the processor cannot exceed 100%

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

Example Schedulability

- Three video streams handled by three processes
  - Stream 1 handled by process A:
    - Frame rate: Every 30 msec one frame
    - CPU time needed to decode stream: 10 msec per frame
  - Stream 2 handled by process B:
    - Frame rate: Every 40 msec one frame
    - CPU time: 15 msec
  - Stream 3 handled by process C:
    - Frame rate: Every 50msec one frame
    - CPU time: 5msec
- Each CPU burst handles one frame, must meet deadline (finished before next frame has to be displayed)
- Is our system performance good enough to schedule all three streams without delay?
- C_i / P_i is the fraction of CPU time used by process i

$$\frac{10}{30} + \frac{15}{40} + \frac{5}{50} = 0.8$$

  - E.g.: Process A consumes 10/30 of the CPU time available

Effectiveness of a Periodic Scheduling Algorithm

- The sum of the processor utilisations of the individual tasks cannot exceed 100%

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

- This condition provides a bound on the number of tasks that a perfect scheduling algorithm could successfully schedule.
- For the Rate-Monotonic Scheduling algorithm, this bound is lower
  - It was shown that the following condition holds:

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