# DM510: CPU Scheduling

Lars Rohwedder

# Disclaimer

These slides contain (modified) content and media from the official Operating System Concepts slides: https://www.os-book.com/OS10/slide-dir/index.html

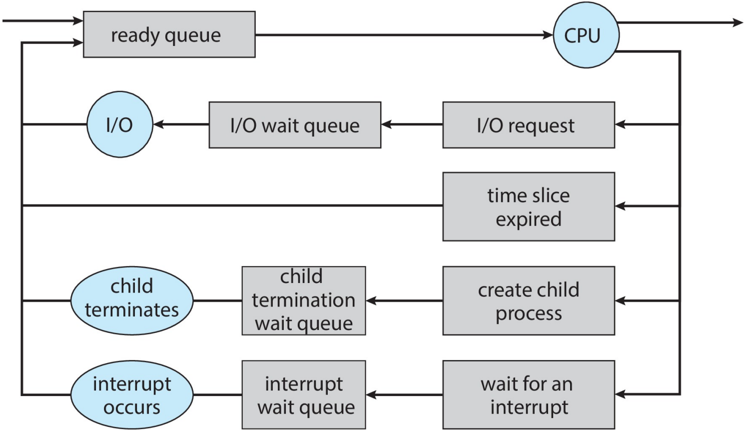
## Today's lecture

- Chapter 5 of course book

# Overview

# Setting

- Typically many processes compete for computation time on CPU
- Processes ready to run wait in a queue
- **Key question:** How does the kernel decide which process to run?



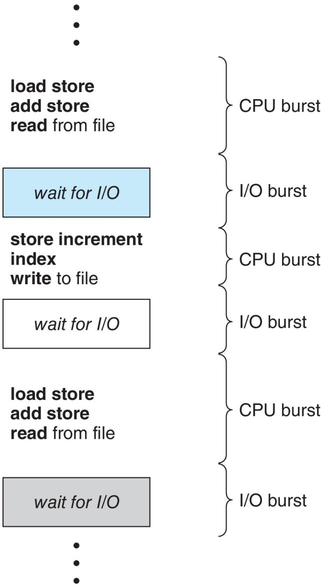
# Scheduling criteria

For choosing a scheduling strategy, we can optimize several criteria that are sometimes contradictory (i.e., we might to decide what is more important)

- **CPU utilization:** Executing as many process instructions as possible
- **Throughput:** Complete as many processes/tasks per time unit as possible
- **Waiting time:** Minimize time a process waits in ready queue
- **Response time:** Minimize time between incoming request and response
- **Fairness:** Make sure that every process/task gets a fair share of CPU time and no task "starves", i.e., never completes
- **Efficiency of algorithm:** Scheduling algorithm itself should not create significant latency/overhead

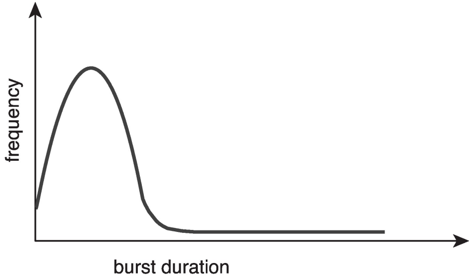
# CPU-I/O cycle

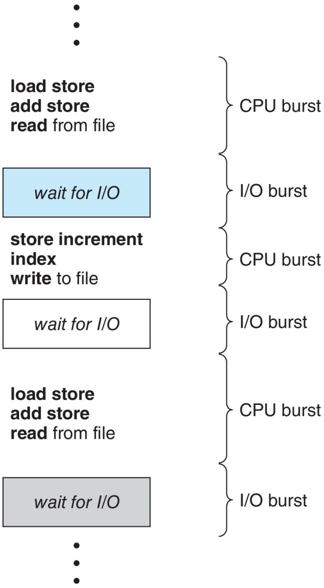
- Typically, processes do not want CPU all the time
- They have **CPU bursts**, in which they execute instructions on the CPU, then need to waits for I/O (**I/O burst**)



# CPU-I/O cycle

- Typically, processes do not want CPU all the time
- They have **CPU bursts**, in which they execute instructions on the CPU, then need to waits for I/O (**I/O burst**)
- Typical distribution: many short CPU bursts, very few long ones



# Preemption

- A scheduler is **non-preemptive** if it allows processes to continue running until they voluntarily suspend (e.g., because of an I/O burst)
- A scheduler that possibly preempts (interrupts) a process currently running on CPU is called **preemptive**
- Preemption is used in all major operating systems, but requires careful programming practices to avoid race conditions

# Algorithms

# First-come-first-serve (FCFS)

- Schedule processes in the order they arrive

# First-come-first-serve (FCFS)

- Schedule processes in the order they arrive

**Example**

| process | burst time | waiting time |
|---|---|---|
| P2 | 3 | 24 0 |
| P3 | 3 | 27 3 |
| P1 | 24 | 0 6 |
| average | | 3 |

# First-come-first-serve (FCFS)

- Schedule processes in the order they arrive
- Suffers from **convoy effect**: long process delays many small processes

**Example**

**Example 2**

| process | burst time | waiting time |
|---|---|---|
| P1 | 3 | 0 |
| P2 | 24 | 24 |
| P3 | 3 | 27 |
| average | | 17 |

# Shortest-job-first (SJF)

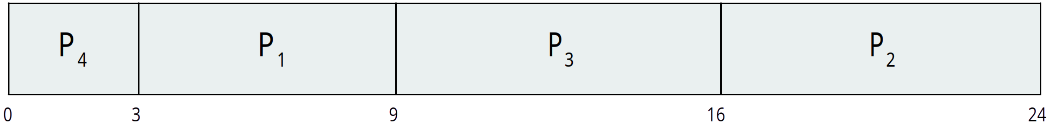
- Schedule processes increasingly by burst time

# Shortest-job-first (SJF)

- Schedule processes increasingly by burst time
- Minimizes average waiting time

**Example**

| process | burst time | waiting time |
|---|---|---|
| P1 | 6 | 3 |
| P2 | 8 | 16 |
| P3 | 7 | 9 |
| P4 | 3 | 0 |
| average | | 7 |

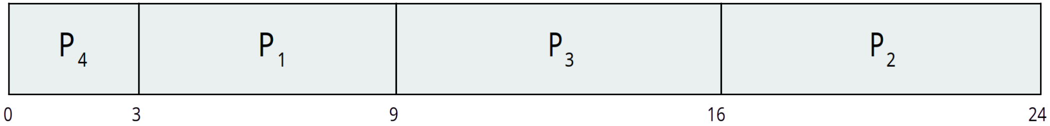


# Shortest-job-first (SJF)

- Schedule processes increasingly by burst time
- Minimizes average waiting time
- How do we know the burst time in advance? *Either provided by process or via estimate*

**Example**

| process | burst time | waiting time |
|---|---|---|
| P1 | 6 | 3 |
| P2 | 8 | 16 |
| P3 | 7 | 9 |
| P4 | 3 | 0 |
| average | | 7 |



# Estimation of burst time (e.g. for SJF)

- Guess next burst time based on previous ones from same process
- Can use different algorithms for prediction

# Estimation of burst time (e.g. for SJF)

- Guess next burst time based on previous ones from same process
- Can use different algorithms for prediction

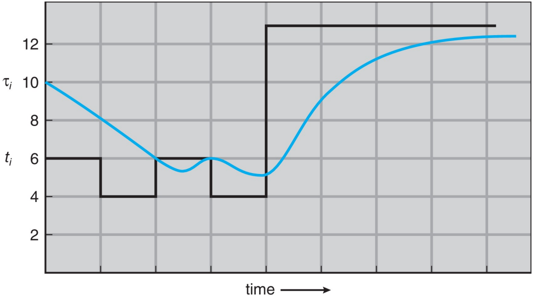
**Example: exponential smoothing**

Choose appropriate value $\alpha \in [0, 1]$ and define guess

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

For typical choice of $\alpha = 1/2$ this simplifies to

$$\tau_{n+1} = \frac{1}{2}t_n + \frac{1}{4}t_{n-1} + \frac{1}{8}t_{n-2} + \cdots$$



# Shortest-remaining-time (SRT)

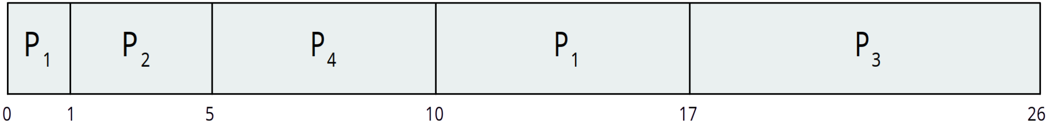
- Preemptive version of SJF: schedule the process with shortest (remaining) burst time, preempting current process if shorter one arrives

# Shortest-remaining-time (SRT)

- Preemptive version of SJF: schedule the process with shortest (remaining) burst time, preempting current process if shorter one arrives

**Example**

| process | arrival time | burst time | waiting time |
|---|---|---|---|
| P1 | 0 | 8 | 10-1 |
| P2 | 1 | 4 | 1-1 |
| P3 | 2 | 9 | 17-2 |
| P4 | 3 | 5 | 5-3 |
| average | | | 6.5 |



# Round-robin (RR)

- Choose time quantum $q$ (typically 10-100ms)
- Preempt a process if it has run continuously for $q$ time. Afterwards, put process at end of queue
- Low values of $q$ would lead to high overhead due to context-switches

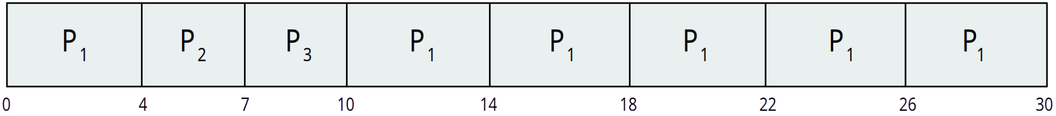
# Round-robin (RR)

- Choose time quantum $q$ (typically 10-100ms)
- Preempt a process if it has run continuously for $q$ time. Afterwards, put process at end of queue
- Low values of $q$ would lead to high overhead due to context-switches

**Example**

$q = 4$

| process | waiting time |
|---|---|
| P1 | 24 |
| P2 | 3 |
| P3 | 3 |



# Priority scheduling

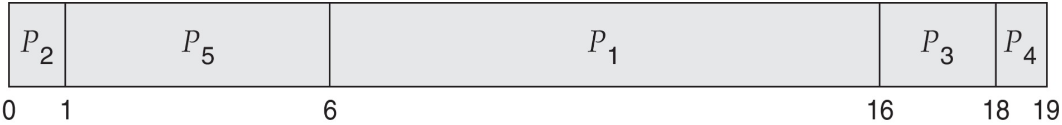
- Each process has priority (integer number)
- Kernel schedules process with highest priority (smallest number), either preemptively or non-preemptively
- Can lead to starvation ...
  Solution: **aging**, ie., priority increases over time

# Priority scheduling

- Each process has priority (integer number)
- Kernel schedules process with highest priority (smallest number), either preemptively or non-preemptively
- Can lead to starvation . . .
  Solution: **aging**, ie., priority increases over time

**Example**

| process | waiting time | priority |
|---|---|---|
| P1 | 24 | 3 |
| P2 | 1 | 1 |
| P3 | 2 | 4 |
| P4 | 1 | 5 |
| P5 | 5 | 2 |



# Priority scheduling with round-robin

- Run process with highest priority
- If multiple processes have highest priority, do round robin on them

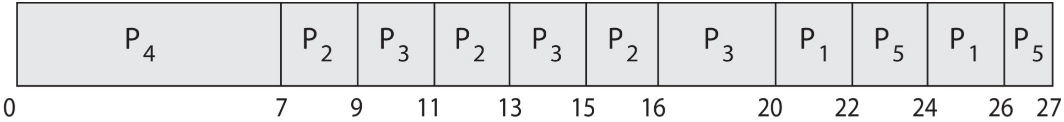
# Priority scheduling with round-robin

- Run process with highest priority
- If multiple processes have highest priority, do round robin on them

**Example**

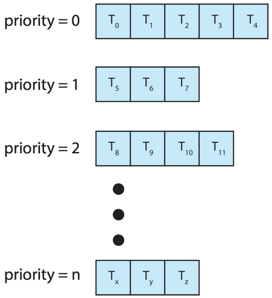
$q = 2$

| process | waiting time | priority |
|---|---|---|
| P1 | 4 | 3 |
| P2 | 5 | 2 |
| P3 | 8 | 2 |
| P4 | 7 | 1 |
| P4 | 3 | 3 |



# Multilevel queue (e.g. for priority scheduling)

- To implement priority scheduling, use separate queue for each priority level
- Scheduler runs next task from first non-empty queue
- Scheduler can decide different algorithm (e.g. round-robin) for each queue




e.g. realtime processes

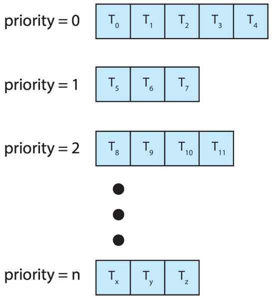
e.g. system processes

e.g. interactive processes

e.g. batch processes

# Multilevel queue (e.g. for priority scheduling)

- To implement priority scheduling, use separate queue for each priority level
- Scheduler runs next task from first non-empty queue
- Scheduler can decide different algorithm (e.g. round-robin) for each queue




e.g. realtime processes

e.g. system processes

e.g. interactive processes

e.g. batch processes

**Multilevel feedback queue**

- Extension where process can be moved between queues (**upgraded** or **demoted**)
- Can be used e.g. to implement aging

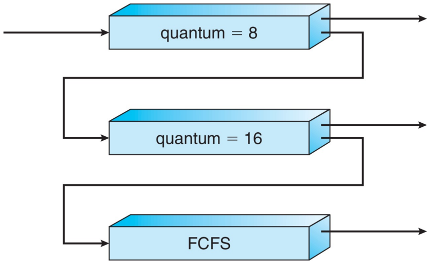
## Example of multilevel feedback queue

Queues:

- $Q_0$: RR with $q = 8ms$
- $Q_1$: RR with $q = 16ms$
- $Q_2$: FCFS

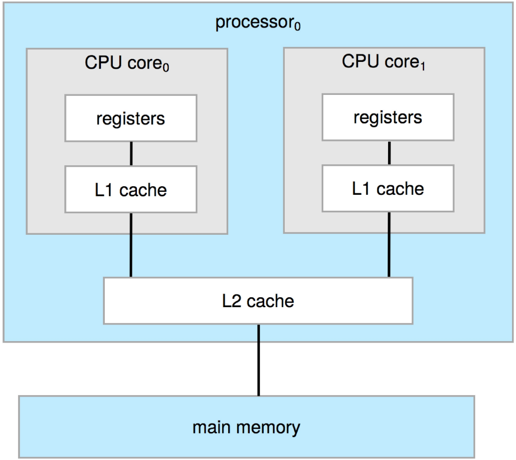
Scheduling:

- New processes go to $Q_0$
- If process running in $Q_0$ needs to be preempted (does not finish CPU burst in $8ms$), move process to $Q_1$
- If process running in $Q_1$ needs to be preempted (does not finish CPU burst in $16ms$), move process to $Q_2$



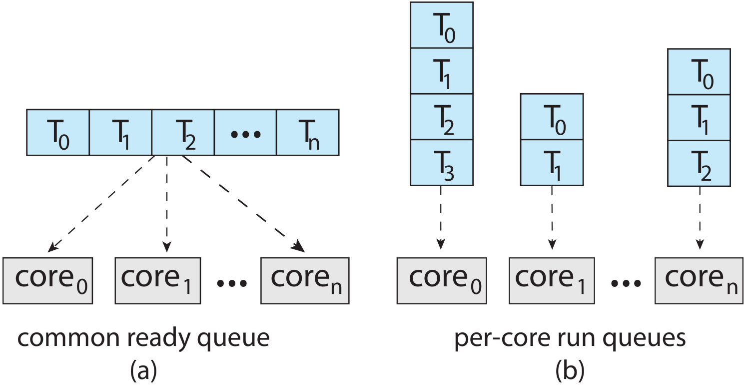
# Multi-Core Scheduling

# Setting


processor$_0$
CPU core$_0$
registers
L1 cache
CPU core$_1$
registers
L1 cache
L2 cache
main memory

# Scheduling on multi-core systems

- Cores can share queues or have separate ones



## Scheduling on multi-core systems

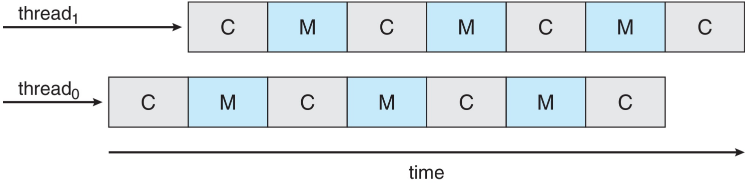
- Cores can share queues or have separate ones
- If core has its own cache, may want to keep threads on same core. **Hard affinity**: Thread is guaranteed to run only on specific core. **Soft affinity**: attempted

# Scheduling on multi-core systems

- Cores can share queues or have separate ones
- If core has its own cache, may want to keep threads on same core. **Hard affinity**: Thread is guaranteed to run only on specific core. **Soft affinity**: attempted
- May need to balance loads. **Push migration**: core gives work to other cores if overloaded, **pull migration**: core takes work from other cores if underloaded

# Scheduling on multi-core systems

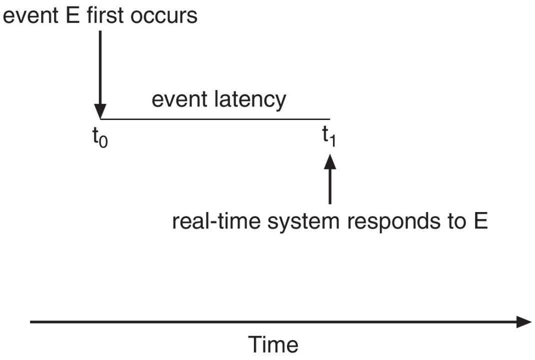
- Cores can share queues or have separate ones
- If core has its own cache, may want to keep threads on same core. **Hard affinity**: Thread is guaranteed to run only on specific core. **Soft affinity**: attempted
- May need to balance loads. **Push migration**: core gives work to other cores if overloaded, **pull migration**: core takes work from other cores if underloaded
- **Multi-threading/hyper-threading:** Some processors can run several threads (with their own register set, etc.) interleaved on one core, executes one thread while other is in **memory stall**, i.e., waiting for RAM access. To system, looks like more cores



# Real-time Scheduling

# Setting

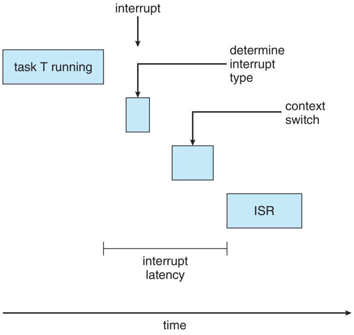
- **Soft real-time system:** missing deadlines is tolerated in extreme cases
- **Hard real-time system:** tasks guaranteed to meet their deadline (fixed bound on event latency)



# Latency

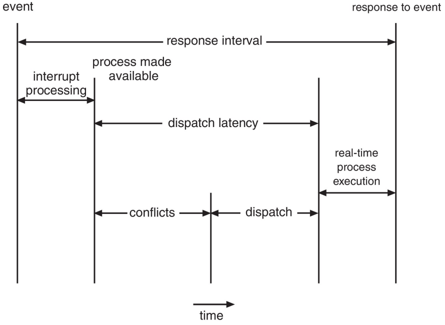
## Interrupt latency

Time between interrupt appearing and interrupt handler running




## Dispatch latency

- Preempt ongoing task and schedule high priority process
- Possibly release resources needed

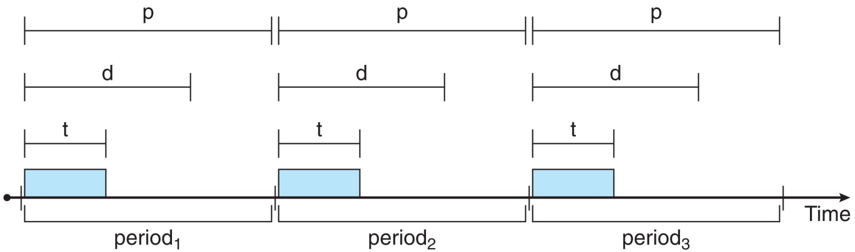



High priority for real-time task $\rightsquigarrow$ low, predictable latency (**soft real-time**)

# Periodic tasks

To design hard real-time systems, software must follow strict specification on their CPU usage: a process emits tasks **periodically**.

- period $p$: at which rate does the process emit tasks
- deadline $d$: how long after each task is emitted does it need to be completed
- processing time $t$: how long does each task need on CPU
- hard real-time guarantees can be proven, assuming low enough system load and all components follow specification



# Evaluating Schedulers

# Evaluation

- Can be via mathematical models (e.g. queueing theory), but often unrealistic
- More practical: **simulations** on data from traces of live system