TDIU11

Operating Systems

CPU Scheduling

[SGG7/8/9] Chapter 5.1-5.4

Christoph Kessler, IDA,
Linköpings universitet

Overview: CPU Scheduling

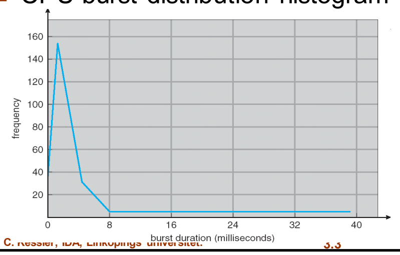
- CPU bursts and I/O bursts
- CPU Scheduling Criteria
- CPU Scheduling Algorithms

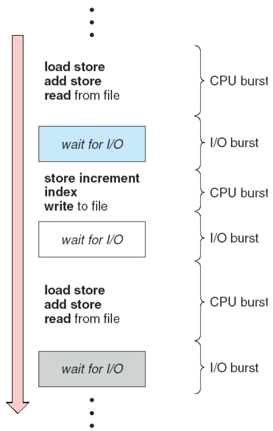
Optional additional material:

- Appendix: Multiprocessor Scheduling
- Appendix: Towards Real-Time Scheduling

Basic Concepts

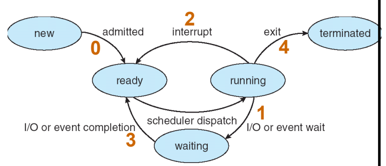
- Maximum CPU utilization obtained with multiprogramming
- **CPU–I/O Burst Cycle** – Process execution consists of a *sequence* of alternating CPU execution and I/O wait
  - **CPU burst** followed by **I/O burst**
- CPU burst distribution histogram





CPU Scheduler

- Selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process:
  0. Is admitted
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready state
  4. Terminates



- Scheduling under 1 and 4 only is ***nonpreemptive***
- All other scheduling is ***preemptive***

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the (short-term) CPU 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** – # 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** including output (for time-sharing environment)
- **Deadlines met?** – in real-time systems (later)

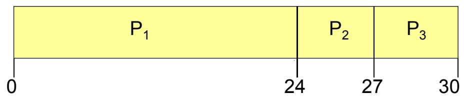
Optimization Criteria

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

First-Come, First-Served (FCFS, FIFO) 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

  Waiting time P_i = start time P_i – time of arrival for P_i
- Average waiting time: (0 + 24 + 27) / 3 = 17

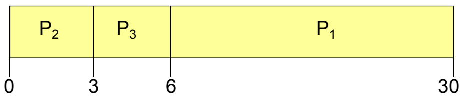
FCFS normally used for non-preemptive batch scheduling, e.g. printer queues (i.e., burst time = job size)

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!
- *Convoy effect* – short process behind long process
  - Idea: shortest job first?

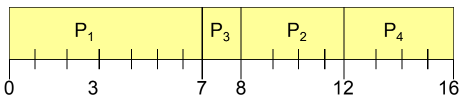
Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst.
- Use these lengths to schedule the shortest ready process
- Two schemes:
  - ***nonpreemptive SJF*** – once CPU given to the process, it cannot be preempted until it completes its CPU burst
  - ***preemptive SJF*** – preempt if a new process arrives with CPU burst length less than remaining time of current executing process.
    - Also known as Shortest-Remaining-Time-First (SRTF)
- 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, or predict from observations of the past

Example of Non-Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- with **non-preemptive SJF**:

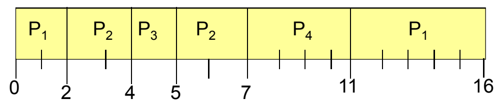


- Average waiting time = (0 + 6 + 3 + 7) / 4 = 4

Example of Preemptive SJF

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- with **preemptive SJF (= SRTF)**:

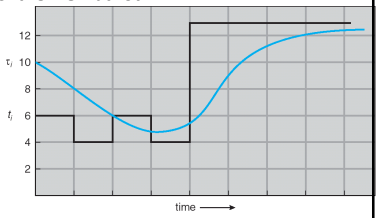


- Average waiting time = (9 + 1 + 0 +2) / 4 = 3

Predicting Length of Next CPU Burst

- Can only estimate the length
- Based on length of previous CPU bursts, using exponential averaging:

1. $t_n =$ actual lenght of n^{th} CPU burst
2. $\tau_{n+1} =$ predicted value for the next CPU burst
3. $\alpha, 0 \le \alpha \le 1$
4. Define : $\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$



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

Examples of Exponential Averaging

- Extreme cases:
  - α =0
    - $\tau_{n+1} = \tau_n$
    - Recent history does not count
  - α =1
    - $\tau_{n+1} = \alpha\, t_n$
    - Only the latest CPU burst counts
- Otherwise: Expand the formula:

$$\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\alpha\, t_{n-1} + \dots$$
$$+(1 - \alpha)^j \alpha\, t_{n-j} + \dots$$
$$+(1 - \alpha)^{n+1} \tau_0$$

  - Since both α and $(1 - \alpha)$ are less than 1, each successive term has less weight than its predecessor

Priority Scheduling

- A priority value (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer $\equiv$ highest priority)
  - preemptive
  - nonpreemptive
- SJF is a priority scheduling where priority is the 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

Convention here:
1 = highest priority,
5 = lowest priority

- Priority scheduling Gantt Chart

- Average waiting time = 8.2

Round Robin (RR)

- Each process gets a small unit of CPU time:
  - ***time quantum***, usually 10-100 milliseconds.
- After this time has elapsed, the process is preempted and added to the end of the ready queue.
- Given n processes in the ready queue and time quantum q, 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.
- Performance
  - q very large $\Rightarrow$ FCFS
  - q very small $\Rightarrow$ many context switches
  - q must be large w.r.t. context switch time, otherwise too high overhead

Example: RR with Time Quantum q = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

- The Gantt chart is:

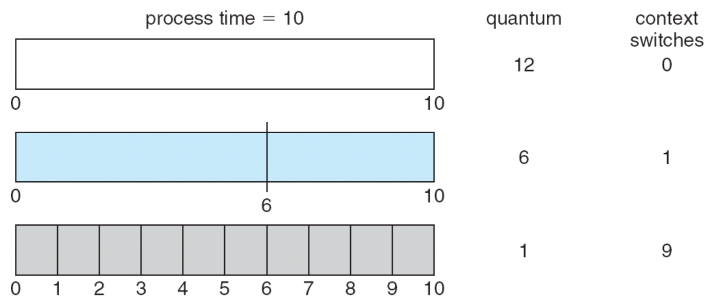
P_1	P_2	P_3	P_4	P_1	P_3	P_4	P_1	P_3	P_3

0 20 37 57 77 97 117 121 134 154 162

- Typically, higher average turnaround than SJF, but better *response*

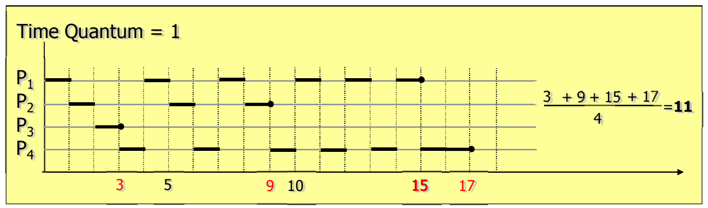
Time Quantum and Context Switches

Smaller time quantum $\Rightarrow$ more context switches



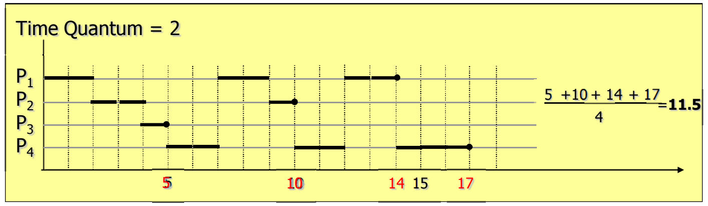
RR: Turnaround Time Varies With Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7

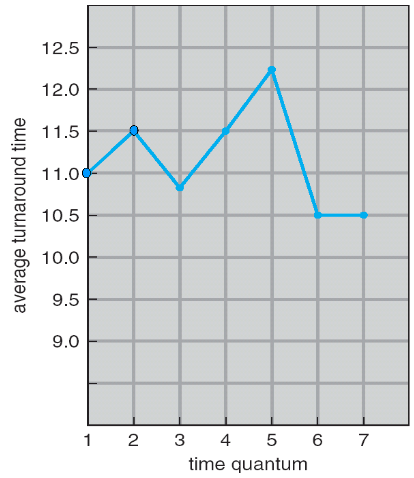


RR: Turnaround Time Varies With Time Quantum

process	time
P_1	6
P_2	3
P_3	1
P_4	7



RR: Turnaround Time Varies With Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Problems with RR and Priority Schedulers

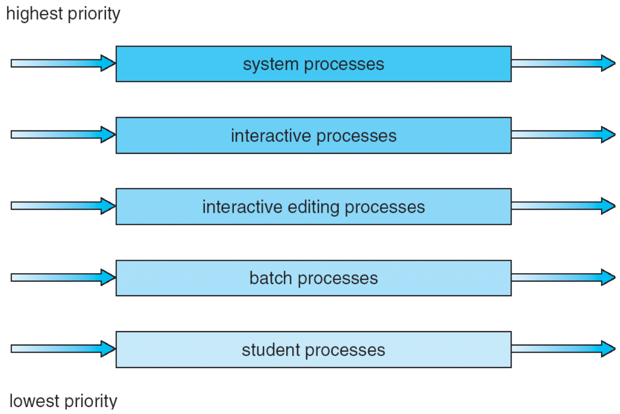
- Priority based scheduling may cause *starvation* for some processes.
- Round robin based schedulers are maybe *too* ”fair”... we sometimes want to prioritize some processes.
- Solution: Multilevel queue scheduling ...?

Multilevel Queue

- Ready queue is partitioned into separate queues, e.g.:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done also between the queues:
  - Fixed priority scheduling
    - Serve all from foreground queue, then from background queue.
    - Possibility of starvation.
  - Time slice
    - Each queue gets a certain share of CPU time which it can schedule amongst its processes
    - Example: 80% to foreground in RR, 20% to background in FCFS

Useful when processes are easily classified into different groups with different characteristica...

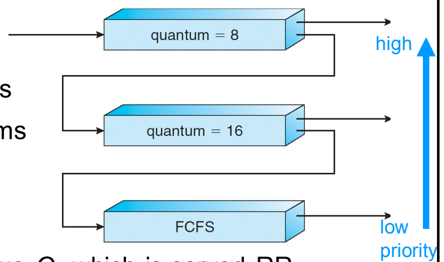
Multilevel Queue Scheduling



Multilevel Feedback Queue

- A process can move between the various queues
  - aging can be implemented this way
- Time-sharing among the queues in priority order
  - Processes in lower queues get CPU only if higher queues are empty

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with q = 8 ms
  - Q_1 – RR with q = 16 ms
  - Q_2 – FCFS
- Scheduling:
  - A new job enters queue Q_0 which is served RR.
  - When it gains CPU, the job receives 8 milliseconds.
  - If it does not finish in 8 milliseconds, it is moved to Q_1.
  - At Q_1 the job is again served RR and receives 16 additional milliseconds.
  - If it still does not complete, it is preempted and moved to Q_2.

Multilevel Feedback Queue

- **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
  - priority level of each queue

Summary: CPU Scheduling

- **Goals:**
  - Enable multiprogramming
  - CPU utilization, throughput, ...
- **Scheduling Algorithms**
  - Preemptive vs Non-preemptive scheduling
  - RR, FCFS, SJF
  - Priority scheduling
  - Multilevel queue and Multilevel feedback queue
- **Appendix:** Multiprocessor Scheduling
- **Appendix:** Towards Realtime Scheduling
- **In the book (Chapter 5):** Scheduling in Solaris, Windows, Linux

TDIU11

Operating Systems

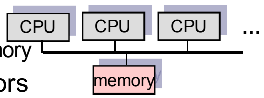
APPENDIX: Multiprocessor Scheduling

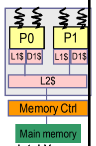
Optional

Christoph Kessler, IDA,
Linköpings universitet

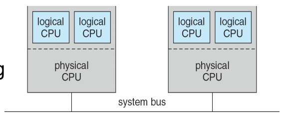
Multiprocessor Scheduling

- CPU scheduling more complex if multiple CPUs available
  - Multiprocessor (SMP)
    - homogeneous processors, shared memory
  - (homogeneous) Multi-core processors
    - cores share L2 cache and memory
  - Multithreaded processors
    - HW threads / logical CPUs share basically everything but register sets
      - HW-Contextswitch-on-Load
      - Cycle-by-cycle interleaving
      - Simultaneous multithreading





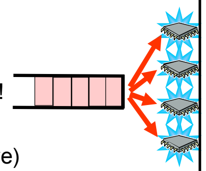
Intel Xeon Dualcore(2005)



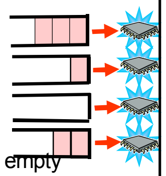
- Parallel jobs: *Work sharing*
  - Centralized vs. local task queues, load balancing
- Supercomputing applications often use a fixed-sized process configuration ("SPMD", 1 thread per processor) and implement own methods for scheduling and resource management (goal: load balancing)

Multiprocessor Scheduling

Multi-CPU scheduling approaches
for sequential and parallel tasks

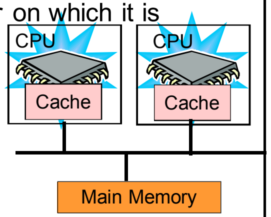


- **Common ready queue** (SMP only) – critical section!
  supported by Linux, Solaris, Windows XP, Mac OS X
  - **Job-blind scheduling** (FCFS, SJF, RR – as above)
    - schedule and dispatch one by one as any CPU gets available
  - **Affinity based scheduling**
    - guided by data locality (cache contents, loaded pages)
  - **Co-Scheduling / Gang scheduling** for *parallel* jobs →
- **Processor-local ready queues**
  - Load balancing by **task migration**
    - Push migration vs. pull migration (*work stealing*)
      - Linux: Push-load-balancing every 200 ms, pull-load-balancing whenever local task queue is empty



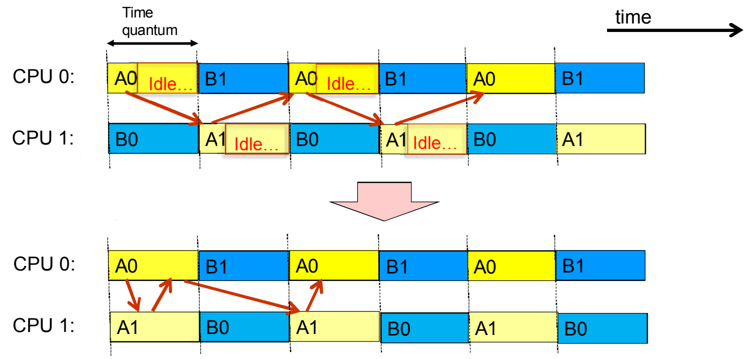
Affinity-based Scheduling

- Cache contains copies of data recently accessed by CPU
- If a process is rescheduled to a different CPU (+cache):
  - Old cache contents invalidated by new accesses
  - Many cache misses when restarting on new CPU
  - → much bus traffic and many slow main memory accesses
- Policy: Try to avoid migration to other CPU if possible.
- A process has **affinity** for the processor on which it is currently running
  - **Hard affinity** (e.g. Linux): Migration to other CPU is forbidden
  - **Soft affinity** (e.g. Solaris): Migration is possible but undesirable



Scheduling Communicating Threads

- Frequently communicating threads / processes (e.g., in a *parallel* program) should be scheduled simultaneously on different processors to avoid idle times



Co-Scheduling / Gang Scheduling

- Tasks can be parallel (have >1 process/thread)
- Global, shared RR ready queue
- Execute processes/threads from the same job simultaneously rather than maximizing processor affinity
- Example: **Undivided Co-scheduling algorithm**
  - Place threads from same task in adjacent entries in the global queue

A1	A2	A3	A4	B1	B2	B3	C1	C2	C3	C4	C5	…

  - Window on ready queue of size #processors
  - All threads within same window execute in parallel for at most 1 time quantum

☺ RR → fair (no indefinite postponement)

☺ Programs designed to run in parallel profit from multiprocessor env.

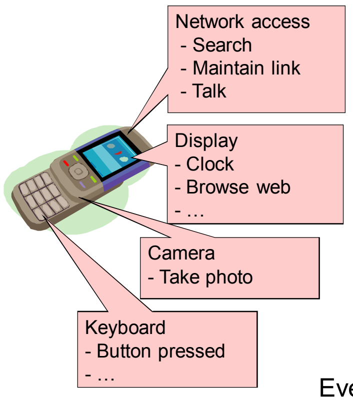
☹ May reduce processor affinity

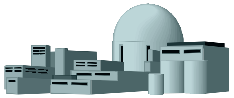
APPENDIX:

Towards Real-Time Scheduling

(optional, maybe useful for Paper 1)

Real-Time Systems





Event-driven (e.g. sensor input) and/or time-driven

Deadlines

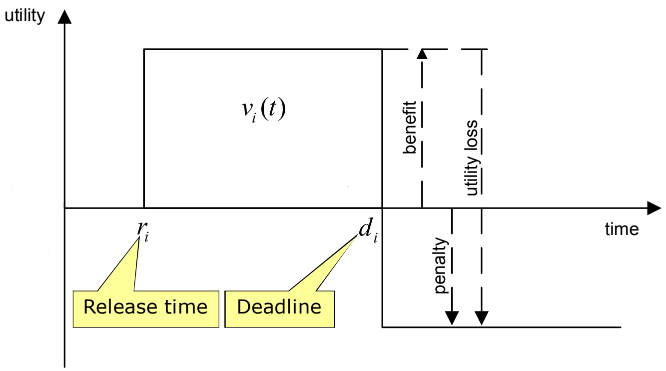
Real-Time Scheduling

- ***Hard real-time*** **systems**
  - required to complete a critical task within a guaranteed amount of time
  - missing a deadline can have catastrophic consequences
- ***Soft real-time*** **computing**
  - requires that critical processes receive priority over less important ones
  - missing a deadline leads to degradation of service
    - e.g., lost frames / pixelized images in digital TV
- Often, periodic tasks or reactive computations
  - require special scheduling algorithms: RM, EDF, ...

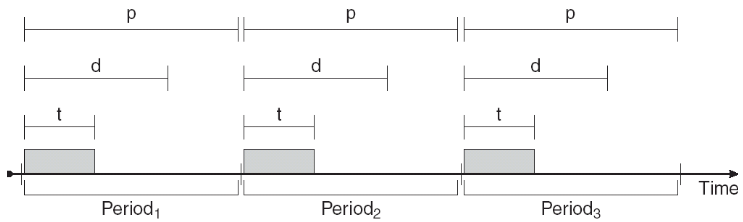
Utility Function

(Value of result relative to time)

Hard real-time system:



Real-Time CPU Scheduling



- Time triggered or event triggered (e.g., sensor measurement)
- **Periodic tasks**
  - Each task *i* has a ***periodicity*** p_i, a (relative) ***deadline*** d_i and a ***computation time*** t_i
    - *CPU utilization of task i:* t_i / p_i
- **Sporadic tasks** (no period, but relative deadline)
- **Aperiodic tasks** (no period, no deadline)

Real-Time Scheduling Algorithms

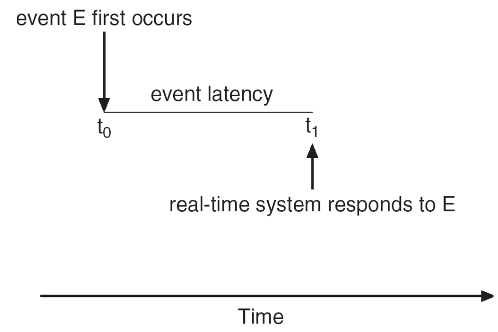
Fundamental algorithms for real-time scheduling of periodic tasks include:

- **Rate-Monotonic Scheduling** (RM)
  - Fixed priorities
- **Earliest-Deadline First** (EDF)
  - Dynamically updated priorities
  - A variant of preemptive SJF (SRTF)
- Details in Paper 1

Chang Liu and James W. Layland: "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment". *Journal of the ACM* Volume 20 (1973): 46-61.

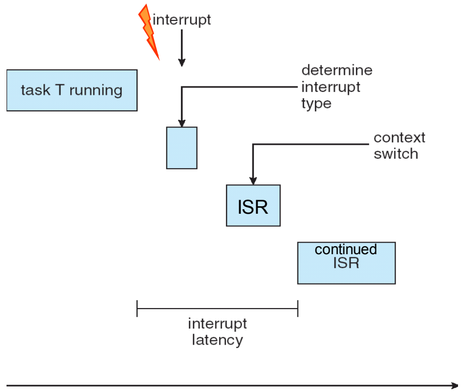
Minimizing Latency

- **Event latency** is the amount of time from when an event occurs to when it is serviced
- Interrupt latency + Dispatch latency



Interrupt Latency

- ***Interrupt latency*** is the period of time from when an interrupt arrives at the CPU to when it is serviced.



Dispatch Latency

- ***Dispatch latency*** is the amount of time required for the scheduler to stop one process and start another.