



COMP2211 Operating Systems: Scheduling

Mantas Mikaitis

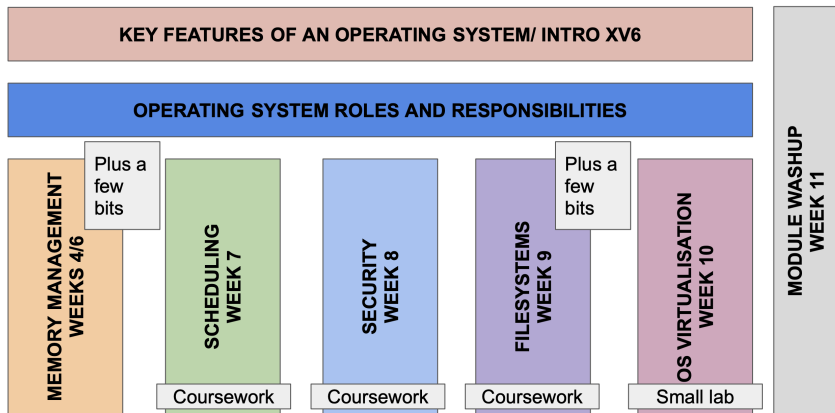
School of Computing, University of Leeds, Leeds, UK

Semester 1, 17-18 November 2022

Week 7, Lectures 1 and 2



Our position in the module



Some remarks

The presented material is based on Chapter 5 of *Operating System Concepts, 10th ed.* by A. Silberschatz, P. B. Galvin and G. Gagne [1].

For these slides we have used images and text from some of the slides provided by the above authors [2] as well as from the 10th of edition of OSC [1].

Recommended reading this week: [1, Ch. 5] (OSC), [3, Ch. 7] (XV6), [4, Ch. 7–10] (OSTEP).

Mid-module survey



Objectives

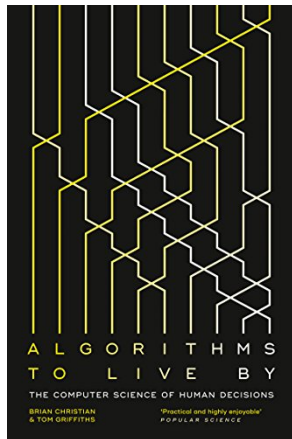
- To introduce **CPU scheduling**, which is the basis for **multiprogrammed** operating systems.
- To describe various **CPU-scheduling algorithms** and understand pros and cons of each.
- To discuss **evaluation criteria** for selecting a CPU-scheduling algorithm for a particular system.
- To understand challenges with scheduling in **multiprocessor** and **real-time systems**.

A General Problem

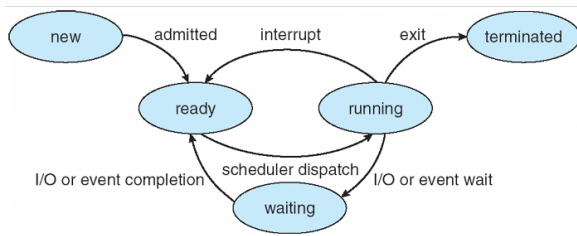
We are going to look at scheduling in operating systems, but it is a general problem (see week 7 lab thought exercise).

“So what to do, and when, and in what order? Your life is waiting.”

From Algorithms to Live By,
Chapter 5 on Scheduling.

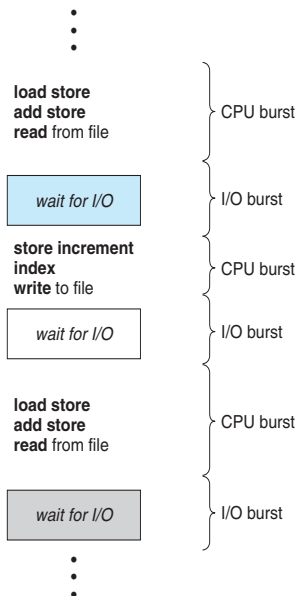


Why CPUs need scheduling?

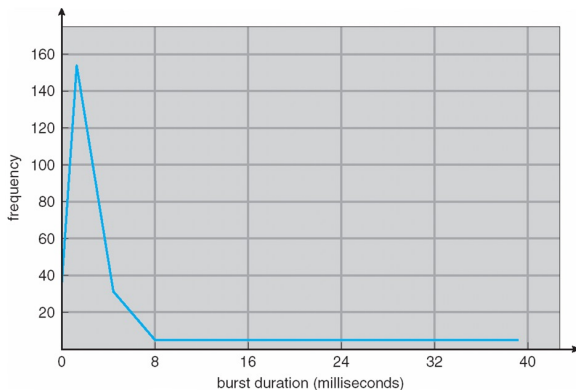


Why CPUs need scheduling?

- Processes go through multiple phases of CPU-I/O over their lifetime.
- Maximum CPU utilization through **multiprogramming**.
- When processes wait for IO, CPU can be used for something.
- What to run next? There is a need for **scheduling**.



Typical CPU Burst Lengths



Usually many short and a few long CPU bursts.

CPU utilization

When CPU becomes idle, OS finds work (waiting process queue).

- **CPU scheduler** selects a process from the ready queue and allocates CPU to it.
- Queue may be ordered in various ways.
- CPU scheduling decisions may take place when a process changes state:
 - 1 running → waiting,
 - 2 running → ready,
 - 3 waiting → ready,
 - 4 terminates.
- For 1 and 4, scheduling is **nonpreemptive** (run as long as needed) while for 2 and 3 **preemptive** (may interrupt a running process).

Challenges with Preemptive Scheduling

A few scenarios that cause problems:

- 1 Process 1 is writing data, is preempted by process 2 that reads the same data.
- 2 Process 1 asks kernel to do some important changes, process 2 interrupts while they are being done.

Disabling interrupts

Irrespective of the challenges, most modern operating systems are fully preemptive when running in kernel mode, but disable interrupts on certain small areas of code.

Dispatcher

Dispatcher gives control of the CPU to the scheduled process.

- **Switching context.**
- Switching to **user mode** (kernel tasks in **supervisor mode**).
- Jumping to the proper location in the previously interrupted user program (set the **Program Counter** register).

Dispatch latency

Time it takes for the dispatcher to stop one process and start another running.

Scheduling Criteria

- **CPU utilization**—reduce amount of time CPU is idle.
- **Throughput**—number of processes completed 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).

When designing a scheduler

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

First-Come, First-Served (FCFS) Scheduling

Process	Burst time
P_1	24
P_2	3
P_3	3

If processes arrive in sequence we have the following schedule:



Waiting time for $P_1 = 0$, $P_2 = 24$, and $P_3 = 27$.

Average waiting time: $\frac{0+24+27}{3} = 17$.

First-Come, First-Served (FCFS) Scheduling

If processes arrive instead as P_2 , P_3 , P_1 :



Waiting time for $P_1 = 6$, $P_2 = 0$, and $P_3 = 3$.

Average waiting time: $\frac{6+0+3}{3} = 3$.

Substantial reduction from the previous case but in general not good.

Issue with FCFS

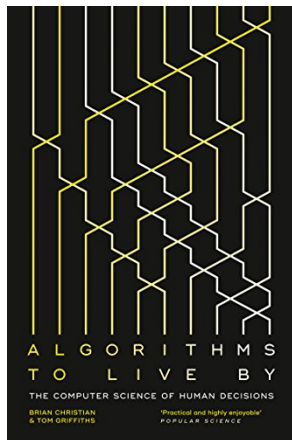
Convoy effect—short jobs can be held waiting by long jobs.

Note that **FCFS is nonpreemptive**.

Questions?

“there’s nothing so fatiguing as the eternal hanging of an uncompleted task,”

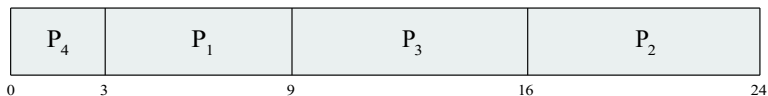
William James. From Algorithms to Live By, Chapter 5 on Scheduling.



Shortest-Job-First (SJF)

- Append each process with the length of next CPU burst.
- Schedule jobs with shortest time.
- SJF is optimal, but difficult to know future CPU burst lengths.
- Ties broken with FCFS scheduling.
- Better name **shortest-next-CPU-burst**.

Process	Next burst time
P_1	6
P_2	8
P_3	7
P_4	3



Average waiting time: $\frac{3+16+9+0}{4} = 7$.

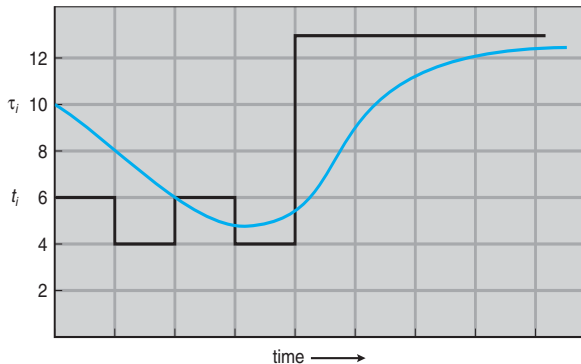
Predicting Lengths of Future CPU Bursts

Make an assumption

Next CPU burst likely similar to the past bursts.

- t_n —actual length of the CPU burst n .
- τ_{n+1} —predicted value of the next burst.
- $0 \leq \alpha \leq 1$.
- $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$.
- We can tune this model through α (usually set to 0.5).

Example Prediction of CPU Bursts



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

Predicting Lengths of Future CPU Bursts

Model of CPU Burst Lengths

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

- $\alpha = 0$, $\tau_{n+1} = \tau_n$ —recent history does not count.
- $\alpha = 1$, $\tau_{n+1} = t_n$ —only the actual last CPU burst counts.
- Expand the formula:
$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots + (1 - \alpha)^{n+1} \tau_0.$$
- Example: $\alpha = 0.5$, $\tau_4 = 0.5t_3 + 0.25t_2 + 0.125t_1 + 0.0625\tau_0$.

Exponential average of past CPU bursts

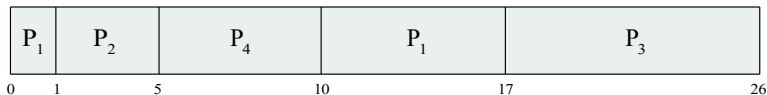
Each successive term has lower weighting than the newer ones, with the initial guess having the lowest.

Shortest-remaining-time-first

If we allow SJF to be preemptive, it can interrupt a currently running process if it would run longer than some new process.

Consider

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Average waiting time is 6.5—standard SJF would result in 7.75.

Shortest-remaining-time-first (Question, 3min)

Take 3 minutes to schedule the following processes with a **preemptive shortest-job-first scheduler**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	9
P_3	2	7
P_4	3	2
P_5	4	3

Shortest-remaining-time-first (Question, 3min)

Take 3 minutes to schedule the following processes with a **preemptive shortest-job-first scheduler**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Arrival time	Next burst time
P_1	0	8
P_2	1	9
P_3	2	7
P_4	3	2
P_5	4	3

Answer

P_1 runs 0 to 3; P_4 interrupts, runs 3 to 5; P_5 runs 5 to 8; P_1 continues, runs 8 to 13; P_3 then runs; finally P_2 is run.

Priority scheduling

Shortest-job-first is a specific case of general scheduler that decides by priorities.

- A priority (integer) associated with each process.
- CPU allocated to a process of highest priority.
- **Starvation**—low priority processes may not execute.
- **Aging**—increase the priority proportional to waiting time.
- **Internal priorities**—time limits, memory requirements, ratio of average I/O burst.
- **External priorities**—importance of the process, type and amount of funds being paid for the CPUs, who is asking to run the process, and other.

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



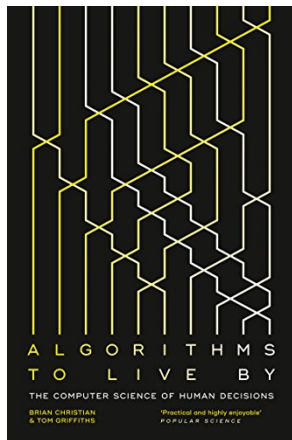
Preemptive priority scheduling

Priorities may change while a process is running.

Questions?

“they wrote up a fix and beamed the new code across millions of miles to Pathfinder. What was the solution they sent flying across the solar system? Priority inheritance.”

From Algorithms to Live By,
Chapter 5 on Scheduling.



Round Robin (RR) Scheduling

- **Time quantum** (q) is defined.
- CPU scheduler assigns the CPU to each process for an interval of up to 1 quantum.
- Queue treated as First-In-First-Out.
- Interrupts every quantum to schedule next process.
- **RR is therefore preemptive.**
- No process allocated for more than q in a row (unless there is only one).

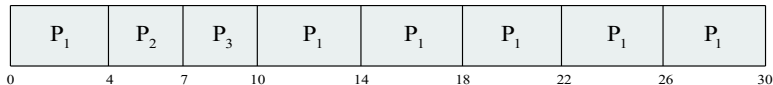
Round Robin (RR) Scheduling

- If there are n processes waiting, each process is guaranteed to get $1/n$ of CPU's time in chunks of time quantum q .
- Each process must wait no longer than $(n - 1) \times q$ time units until its next turn to run.

Round Robin (RR) Scheduling

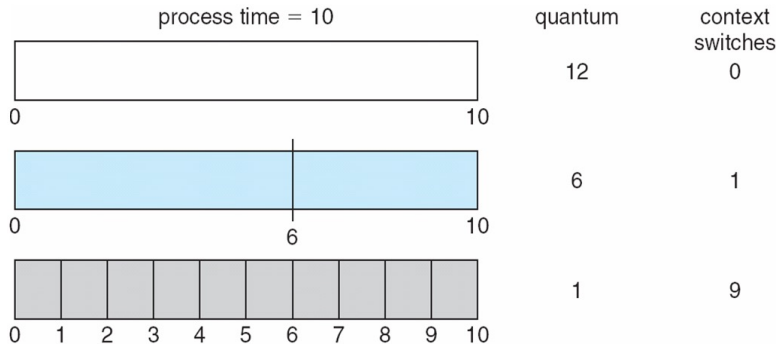
Take $q = 4$.

Process	Burst time
P_1	24
P_2	3
P_3	3



- Small quantum—too many interrupts will reduce performance.
- Big quantum—scheduler similar to FCFS.
- Need a balance (according to OSC, usually $q = 10$ to 100 ms).
- Context switch around 10 microseconds (small fraction of q).

Round Robin (RR) Scheduling

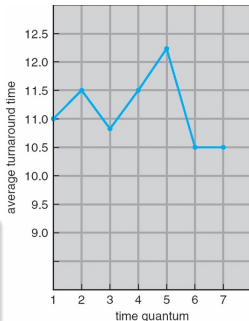


Round Robin (RR) Scheduling

- **Turnaround time** depends on the size of the quantum.
- However, it does not necessarily improve with the size of q .

Rule of Thumb

80% of CPU bursts should be shorter than q .



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Round Robin (RR) Scheduling (Question, 3min)

Take 3 minutes to schedule the following process queue with a **round robin scheduler** with $q = 3$. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Burst time
P_1	5
P_2	12
P_3	3
P_4	1

Round Robin (RR) Scheduling (Question, 3min)

Take 3 minutes to schedule the following process queue with a **round robin scheduler** with $q = 3$. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	Burst time
P_1	5
P_2	12
P_3	3
P_4	1

Answer

P_1 runs 0 to 3; P_2 runs 3 to 6; P_3 runs 6 to 9; P_4 runs 9 to 10; P_1 runs 10 to 12; P_2 runs 12 to 15; P_2 runs 15 to 18; P_2 runs 18 to 21.

Multilevel Queue Scheduling

- With previous algorithms, it takes $\mathcal{O}(n)$ to search the queue.
- Assign processes to different queues, by priority.
- Can also assign to queues by process types:
 - ① Queue for **background processes** (for example, batch processing)
 - ② Queue for **foreground processes** (interactive)
- Each queue can have different scheduling algorithms, depending on needs.
- Scheduling may be required among queues: commonly fixed-priority preemptive scheduling.

Multilevel Queue Scheduling

Example queues in decreasing priority level:

- 1 Real-time processes
- 2 System processes
- 3 Interactive processes
- 4 Batch processes

Multilevel priority queue

No process in a lower priority queue runs while there are processes waiting in the higher priority queues. High priority queues preempt lower priority ones.

Time slicing

Another possibility is to allocate time among queues. Example: 80% to foreground queue and 20% to the background queue.

Multilevel Feedback Queue Scheduling

Dynamic queueing

Instead of fixing processes to queues, allow them to move.

Multilevel feedback queue defined by

- number of queues,
- a scheduling alg. for each queue,
- a method to upgrade a process to higher priority queue,
- a method to downgrade a process, and
- a method to determine which queue to assign process at the start.

Multilevel feedback queue

Most general CPU scheduling algorithm due to many parameters in the definition.

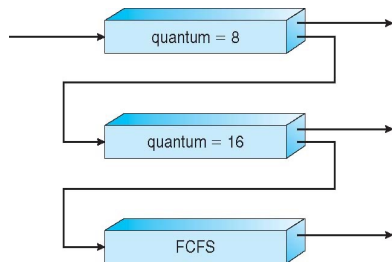
Multilevel Feedback Queue Scheduling (Example)

Three queues (from the top):

- Q0—RR with $q = 8$ ms.
- Q1—RR with $q = 16$ ms.
- Q2—FCFS.

Scheduling:

- 1 A new job enters Q0 and gets 8 ms.
- 2 Not finished in 8 ms—move to Q1.
- 3 Not finished in queue 1 in another 16 ms—move to Q2.
- 4 Scheduled in FCFS in Q2 when queue 0 and 1 empty.



Starvation in Q2

To prevent starvation we may move old processes to Q0/1.

Advantages and Disadvantages of Scheduling Algorithms

Algorithm	(dis)advantages
FCFS	Convoy effect a problem—long jobs hold the queue.
SJF	Need to predict future CPU burst lengths.
Preemptive SJF	Better average waiting time than SJF.
Priority scheduler	Starvation.
RR	Need to tune time quantum to avoid expensive context switch.
Multilevel queue	Faster search than $\mathcal{O}(n)$.
Multilevel feedback queue	Configuration can be expensive. Starvation.

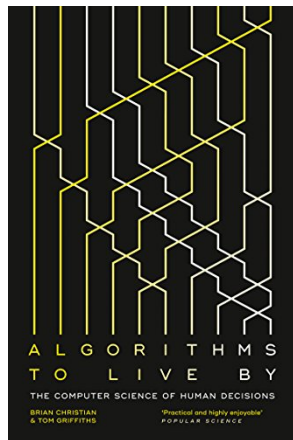
Practice

There is no perfect algorithm for all cases. It is a tradeoff based on requirements of the system and usually a combination of scheduling algorithms is implemented (See OSC OS examples [1, Sec. 5.7]).

Questions?

"In fact, the weighted version of Shortest Processing Time is a pretty good candidate for best general-purpose scheduling strategy in the face of uncertainty."

From Algorithms to Live By,
Chapter 5 on Scheduling.



Multi-Processor Scheduling

Traditionally term **multi-processor** referred to systems with multiple physical cores. Now we use it to describe systems with either several physical or virtual cores/threads.

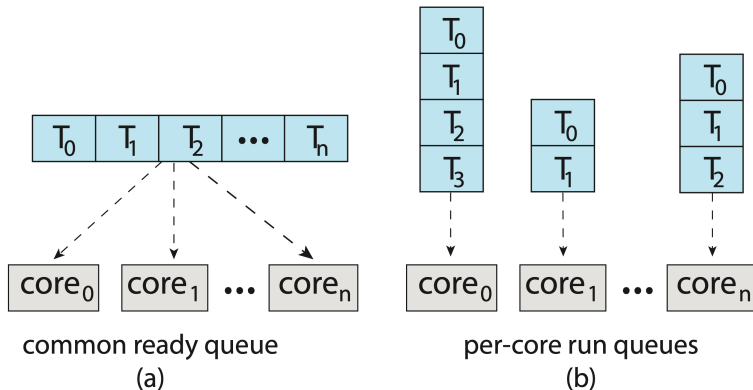
One approach to scheduling is to have one master processor handling scheduling (**assymmetric multiprocessing**). Master becomes potential bottleneck.

Another is **symmetric multiprocessing (SMP)**—each processor handles its scheduling. Most common (Windows, Linux, macOS, Android, iOS).

Multi-Processor Scheduling: SMP

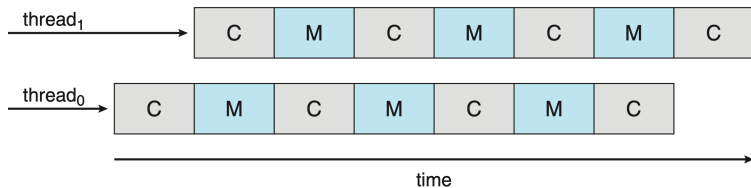
Two approaches in SMP

- 1) Common ready queue—each processor takes processes/threads from that queue (potential clashes).
- 2) Each processor has its own queue.

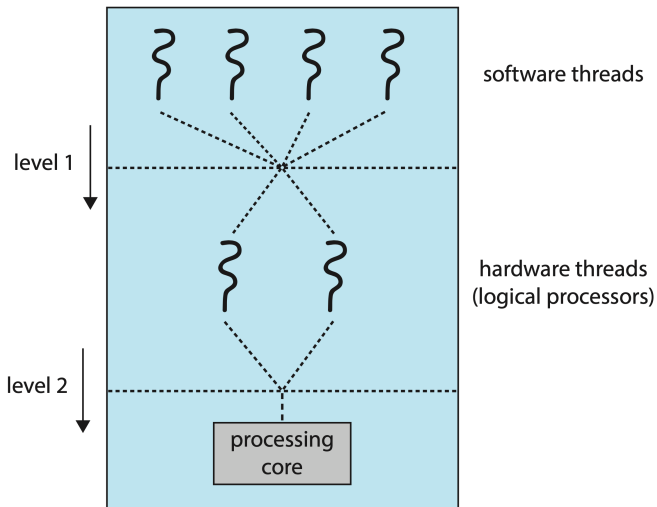


Multicore Processors

- Relatively recent trend is to place multiple cores on chip (**multicore**).
- Speed and energy efficiency.
- **Memory stall**—cores spend significant amount of time for memory (since these days cores are much faster than memory).
- **Multithreading**—hardware assisted multiple threads per core.
- When one thread is in memory stall, work on another.
- OS sees different hardware threads as separate CPUs.



Multicore Processors: Two Levels of Scheduling



Load balancing

- With SMP we need to utilize all CPUs efficiently.
- Load balancing attempts even distribution.
- Only necessary on systems with separate queues for each CPU.
- **Push migration**—a task checks the load on each CPU and moves threads from CPU to CPU to avoid imbalance.
- **Pull migration**—idle processor pulls waiting tasks from busy processors.

Processor Affinity

- When a thread runs a processor, the cache is “warmed up” for that thread.
- We say that a task has affinity for the processor it’s running on.
- When a task is moved, say due to load balancing, we have a big overhead in terms of cache.
- Invalidating and repopulating caches is expensive.
- **Soft affinity**—OS will attempt to keep the process on the same core, but load balancing can move it.
- **Hard affinity**—processes specify a list of processors on which to run.
- Usually both methods are available.

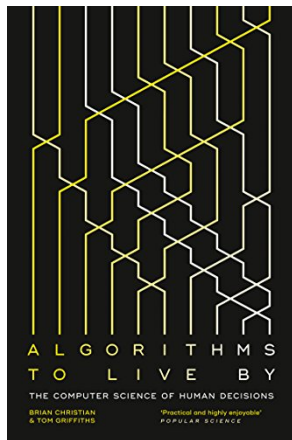
Implications on scheduling

Load balancing and processor affinity both may have implications on scheduling.

Questions?

“the Linux core team, several years ago, replaced their scheduler with one that was less “smart” about calculating process priorities but more than made up for it by taking less time to calculate them.”

From Algorithms to Live By,
Chapter 5 on Scheduling.



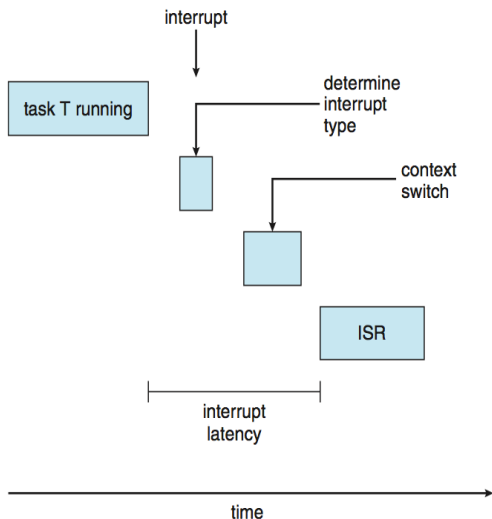
Real-Time CPU Scheduling

- Real-time systems categorized into two:
 - ① **Soft real-time:** guarantee preference for critical processes.
 - ② **Hard real-time:** guarantee completion by deadline.
- Two types of latencies affect performance:
 - ① **Interrupt latency:** time from arrival to interrupt service routine.
 - ② **Dispatch latency:** time for dispatcher to stop current process and start another.

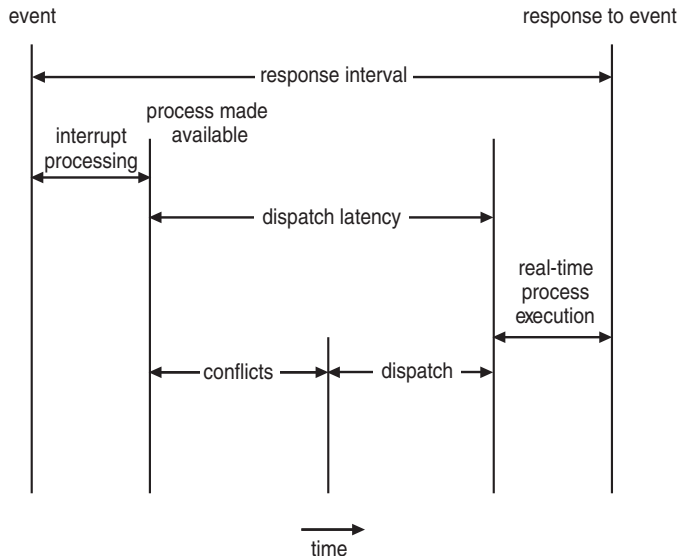
Hard real-time systems

Various latencies should be bounded to meet the strict requirements of these systems.

Real-Time CPU Scheduling



Real-Time CPU Scheduling



Priority-Based Scheduling

Real-time systems

It is essential to have a priority-based preemptive scheduling for real-time systems. Usually real-time processes have highest priority.

Priority-based preemptive scheduling gives us soft real-time functionality.

Additional scheduling features required for hard real-time.

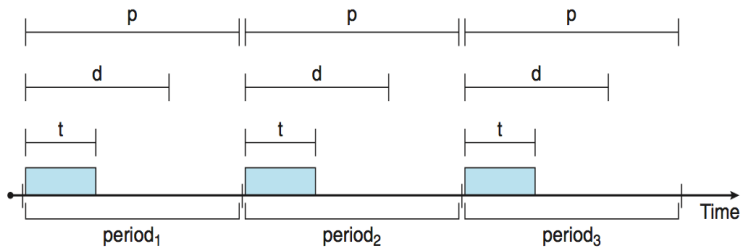
Some definitions:

- Processes are periodic—require CPU at constant intervals.
- Processing time t , deadline d , period p . Here $0 \leq t \leq d \leq p$.

Admission control

Schedulers take advantage of these details and assign priorities based on deadlines and period. **Admission control** algorithm may reject the request as impossible to service by the required deadline.

Priority-Based Scheduling



Rate-Monotonic Scheduling

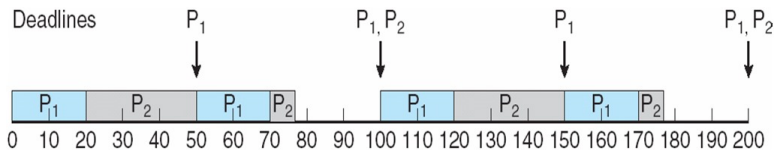
- Upon entering the system, each periodic task assigned priority $\propto \frac{1}{p}$.
- Rationale: prioritize processes that require CPU more often.

Example:

Process	p	t	d
P_1	50	20	50
P_2	100	35	100

P_1 has higher priority since the period is shorter.

Rate-monotonic scheduler:



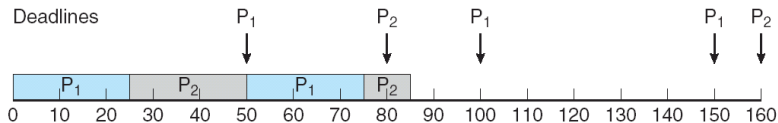
Rate-Monotonic Scheduling

Now we make the requirements more strict for P_2 :

Process	p	t	d
P_1	50	25	50
P_2	80	35	80

P_1 has higher priority since the period is shorter.

Rate-monotonic scheduler:

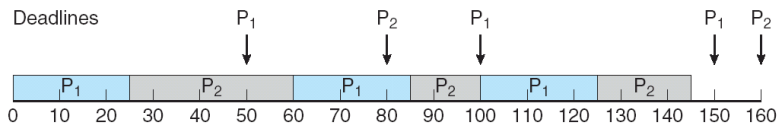


P_2 failed to complete by $d = 80$! The total CPU utilization is $25/50 + 35/80 = 0.94$, but the problem was that the scheduler starts P_1 again before P_2 completes.

Earliest-Deadline-First Scheduling

I think you have been using this one in the past weeks! 😊

Priorities not fixed in advance—the earlier the deadline, the higher priority.



At time 50 process P_2 is not preempted by P_1 because its next deadline (80) is earlier than process P_1 's next deadline at time 100.

EDF Scheduling

No requirement of the period, just the deadline, therefore processes do not need to be periodic as with rate-monotonic scheduling.

Earliest-Deadline-First Scheduling (Question, 5min)

Take 5 minutes to schedule the following process queue with a **Earliest-Deadline-First Scheduling**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

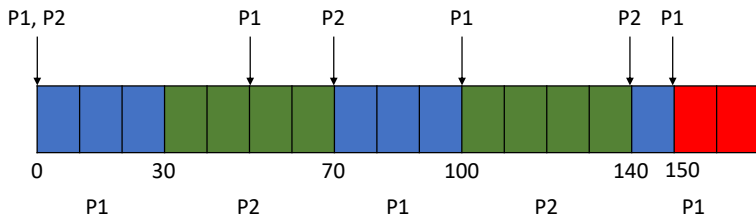
Process	p	t	d
P_1	50	30	50
P_2	70	40	70

Don't forget the aforementioned **admission control**.

Earliest-Deadline-First Scheduling (Question, 5min)

Take 5 minutes to schedule the following process queue with a **Earliest-Deadline-First Scheduling**. Feel free to discuss with your peers. Volunteers for the solution welcome at the end.

Process	p	t	d
P_1	50	30	50
P_2	70	40	70



Scheduling in XV6

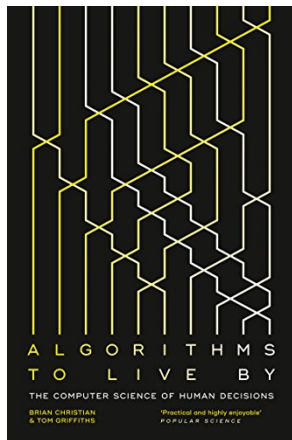
Scheduling occurs in two situations:

- Running process runs `sleep` or `wait`.
- XV6 periodically forces scheduling (round-robin with quantum of ~ 100 ms).
- Scheduler exists as a separate thread per CPU.
- Queue of up to 64 processes available.
- See `kernel/proc.c` for further detail. Scheduler in the function `void scheduler(void)`.

Questions?

“there’s no choice but to treat that unimportant thing as being every bit as important as whatever it’s blocking.”

From Algorithms to Live By,
Chapter 5 on Scheduling.




Conclusion


- We have learned fundamentals of CPU scheduling.
- Please study the recommended reading materials, especially OSC.
- Check the past exam papers (<https://students.leeds.ac.uk/exampapers>).
- Martin and me usually available online and on-site (office open hours Mon. 11-12) for questions.
- Next week: Martin is back with security in OS.
- **Coursework (20%) deadline 30th of November, 14:30.**


Mid-module survey



References I

 A. Silberschatz, P. B. Galvin, and G. Gagne
Operating System Concepts. 10th edition
Wiley. 2018

 A. Silberschatz, P. B. Galvin, and G. Gagne
Operating System Concepts. 10th edition. Accompanying slides
<https://www.os-book.com/OS10/slide-dir/index.html>
2020

 R. Cox, F. Kaashoek, and R. Morris
xv6: a simple, Unix-like teaching operating system
<https://pdos.csail.mit.edu/6.828/2022/xv6/book-riscv-rev3.pdf>
Version of Sep. 5, 2022

References II



R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau
Operating Systems: Three Easy Pieces. Version 1.0
<https://pages.cs.wisc.edu/~remzi/OSTEP/>
Arpaci-Dusseau Books. Aug., 2018