Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨Daniel.Page@bristol.ac.uk⟩

February 9, 2016

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

Continued from last lecture ...

Notes:

COMS20001 lecture: week #15

- ... so far so good, *but*

1. **mechanism** $\Rightarrow$ **dispatcher**
2. **policy** $\Rightarrow$ **?**

Notes:

- ... so far so good, *but*

  1. **mechanism** $\Rightarrow$ **dispatcher**
  2. **policy** $\Rightarrow$ **scheduler**

  so we need to answer
  - when should the **scheduler** be invoked, and
  - which **scheduling algorithm** should it use.

Notes:

Concepts (1)

Definition (pre-emptive multi-tasking vs. co-operative multi-tasking)

A multi-tasking kernel (and hence the scheduler) may be

- **pre-emptive** if invocation of the scheduler is *forced on* the currently executing process, or
- **co-operative** (i.e., *not* pre-emptive) if invocation of the scheduler is *volunteered by* the currently executing process.

- Eh?
  - a co-operative process might be
    - *intentionally* co-operate, i.e., via a `yield` system call,
    - *unintentionally* co-operate, i.e., via a blocking I/O system call such as `read`

    with the latter effectively acting as a pre-emption opportunity,
  - an interrupt-generating **timer** is typically used to force a mode switch, and allow pre-emptive scheduling.

Notes:

- Co-operative multi-tasking is simpler, and more light-weight in the sense there is less for the scheduler to do; the same simplicity also makes it easier to reason about. However, the major disadvantage is the potential to impact on the goals of fairness and liveness: a process which does not co-operate, e.g., never invokes `yield`, could monopolise the processor and hence prevent other processes from executing.
- It should be obvious that pre-emption requires a mode switch into kernel mode: if no such switch can be forced, the executing process will simply continue to execute because the kernel cannot interrupt it! Under a pre-emptive strategy the kernel could still schedule processes each time an interrupt or system call is made, since these result in a switch into kernel mode. Example, cases include when
  - the current process terminates,
  - the current process goes from running to blocked (e.g., performs some I/O operation),
  - the current process goes from blocked to ready (e.g., when that I/O operation completes), *or*
  - an interrupt of some sort.

  There is no *guarantee* system calls will occur, or when they will if they do. As a result, it is attractive to *force* the last case to occur using a (regular) timer: this *guarantees* an interrupt always occurs regularly, and hence that a mode switch occurs st. scheduling can take place.
- Most scheduling algorithms, or at least the ones covered here, might be *better suited* to either co-operative or pre-emptive multi-tasking. That said, however, they will normally be *viable* (to a greater or lesser extent) for either strategy.

Concepts (2)

Definition (**short-term scheduler** vs. **long-term scheduler**)

A scheduler is typically classified as being

- a **short-term scheduler** is invoked frequently, and tasked with selecting a process to execute from the ready queue, or
- a **long-term scheduler** is invoked infrequently, and tasked with
  - controlling the degree of multi-programming, and
  - ensuring an effective process mix

with intermediate points (cf. **medium-term scheduler**) possible but more loosely defined.

Notes:

Concepts (3)

Definition (*x*-**bound**)

A process is said to be *x*-**bound** if *x* limits how quickly it can execution: it will be

- **CPU-bound** (or **compute-bound**) if the time it takes to complete is dominated by the instruction execution rate of the processor, or
- **I/O-bound** if the time it takes to complete is dominated by the time waiting for I/O operations to complete

noting that **memory-bound** processes are a sub-class of I/O-bound processes relating specifically to memory access.

Definition (*x*-**burst** and *x*-**wait**)

The terms *x*-**burst** and *x*-**wait** describe periods of time when a process is either doing *x* or waiting for *x* respectively. Note that

- CPU-bound processes typically have *long* CPU-bursts and *few* I/O-waits, while
- I/O-bound processes typically have *short* CPU-bursts and *many* I/O-waits.

Notes:

- A *typical* process will alternate between CPU-bursts and I/O-waiting, in a kind of cycle: this is obvious, if you consider that such processes will typically accept input, compute some output, then produce that output.
- In rough terms you could think of compute- and I/O-bound processes as those which *mostly* do computation (which is local to the processor) or I/O (which involves devices remote from the processor).
  This is too simplistic in some cases, however, because a process might not do much I/O yet still be I/O-bound (e.g., if the associated device is slow relative to the processor). As such, whatever *bottleneck* limits (or bounds) the process will be important: if the process is *x*-bound for whatever *x* is, then it should execute faster if we could make *x* less of a bottleneck.
- The fact that processes can be blocked by I/O, even if technically this is a one-off vs. long-term limit, offers a neat motivation for supporting the concept of multi-tasking at all.
  Imaging each process spends some fraction of time c doing computation. One might argue that having $n = 100/c$ processes would yield perfect, 100% utilisation of the processor due to multi-tasking (vs. much less if multi-tasking were *not* used): there would always be one process able to make progress if it were scheduled for execution. Even if there *were* n such processes available to support this fact, it is too simple a model because, in reality, however, each process also spends a fraction b blocked waiting for access to or for the I/O device itself. Therefore, the probability that *all* processes are blocked is b^n and so the processor is likely idle for $1 - b^n$ of the time.

- Problem: we cannot *know* what
  1. the next burst and/or wait period, or
  2. the total (and so remaining) execution time

  for a given process will be.
- Solution: estimate the next burst period, i.e.,
  1. let

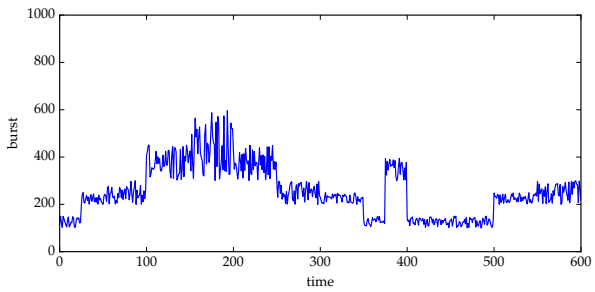
     R_i denote the measured burst period at time i
     S_i denote the smoothed burst period at time i
  2. update the smoothed estimate via a moving average, i.e.,

     $$S_{i+1} = (1 - \alpha) \cdot S_i + \alpha \cdot R_i$$

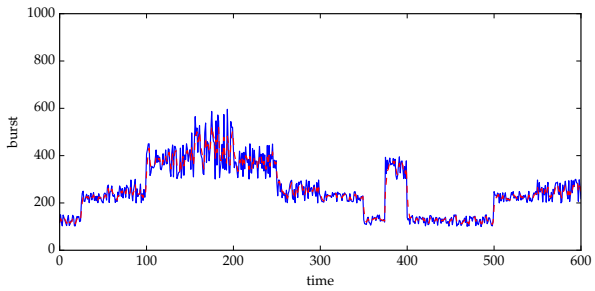
     where $\alpha = 0.5$ say,

  and approximate remaining execution time by most recent estimate.

Notes:



Notes:



Notes:

Definition (arrival time)

The **arrival time** of a process is typically defined as the point where it enters the ready queue, i.e., the first point in time when it *can* be executed; in theory it *should* be distinguished from the process **creation time** (or **submission time**), but in practice the two are often conflated.

Notes:

Definition (batch system)

A **batch system** (or **batch processing system**) is st. all processes (aka. **jobs**) are specified *before* execution, and complete without interaction with a (human) user; this implies all input is also available *before* execution. The resulting processes are often CPU-bound.

Definition (interactive system)

An **interactive system** is st. processes may be specified *before* execution, and complete with interaction with a (human) user. The resulting processes are often I/O-bound.

Definition (real-time system)

A **real-time system** is st. a **deadline** (i.e., a constraint on response time) is imposed

- **soft real-time** deadlines are less strict, st. missing one is unattractive yet tolerable, whereas
- **hard real-time** deadlines are very strict, st. missing one is disastrous.

A deadline typically stems from a need to respond to an event (e.g., a hardware interrupt), which can **periodic** or **aperiodic**.

Notes:

- A further type of scheduler, namely **execute-to-completion**, is typically used by batch systems. In this case, the kernel knows the set of processes which need to be executed *before* their execution: it can compute then use a schedule for execution, and execute each process to completion with the only intervention occurring when one exits and another needs to be created.
- Although standard examples of batch processing normally relate to business applications (e.g., payroll) or system maintenance (e.g., a regular backup or integrity check of a file system invoked by a `cron` job), it is important to realise this model is valid in a wide range of contexts. For example, the queuing system on modern HPC installations is basically a batch processing system: jobs queue for access to (a set of) computational resource (e.g., nodes in a cluster computer), and then execute to completion.

- Problem: a effective *general-purpose* scheduling algorithm is illusive, since
  1. we have numerous (sometimes mutually exclusive) criteria, and
  2. said criteria *plus* the workload wrt. processes can change dynamically.
- Compromise:

	Efficiency	Fairness	Liveness	Utilisation	Throughput	Turn-around	Responsiveness	Proportionality	Predictability
batch system	✓	✓	✓	✓	✓	✓			
interactive system	✓	✓	✓	✓			✓	✓	
real-time system	✓	✓	✓	✓			✓		✓

Notes:

- The sorts of criteria that may need to be considered include
  - goals
    - fairness,
    - liveness,
    - ...
  - metrics
    - utilisation,
    - throughput,
    - turn-around time,
    - waiting time,
    - response time,
    - ...

    and
  - constraints
    - priorities,
    - soft real-time,
    - hard real-time,
    - ...

  but such a list could go on and on. One criteria that is often ignored is that of **graceful degradation**: the idea is that, ideally, as the system is placed under increased load (i.e., needs to support more processes) the performance should deteriorate gradually (and ideally in a predictable way) rather than abruptly.
- In some more detail, note that
  - throughput is another way to say completion rate, i.e., the number of processes completed per time unit,
  - turn-around time measures the number of time units taken to complete a given process, i.e., the process latency,
  - waiting time is the number of time units a given process spends in ready queue versus actually executing,
  - response time is the number of time units between arrival (or submission) of and first output from a given process (where first output is often interpreted as first time it executes, i.e., moves from ready to executing).

Scheduling Algorithms (2) – FCFS

- Idea: **First-Come First-Served (FCFS)**, i.e.,
  - schedule processes via normal ready queue (i.e., FIFO)

  st.
  - good: simple to implement and understand,
  - bad: arrival time has a significant impact, leading to **convoy effect**,
  - bad: average waiting time is typically high.

Notes:

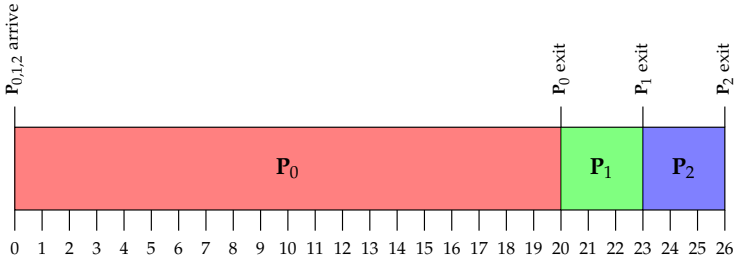
Scheduling Algorithms (3) – FCFS

Example (FCFS, no pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	0	20	0
$\mathbf{P}_1$	0	3	0
$\mathbf{P}_2$	0	3	0

yielding



which can be evaluated as follows:

throughput	=	3/26	=	0.12
av. turn-around time	=	(20 + 23 + 26) / 3	=	23.00
av. waiting time	=	(0 + 20 + 23) / 3	=	14.33
av. response time	=	(0 + 20 + 23) / 3	=	14.33

Notes:

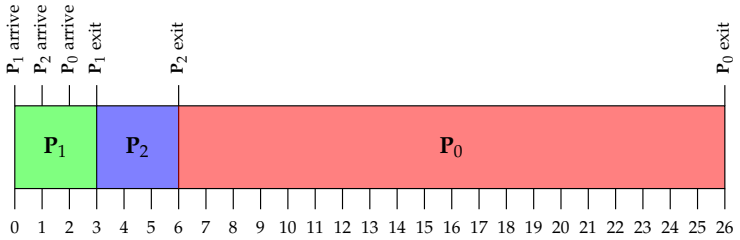
Scheduling Algorithms (3) – FCFS

Example (FCFS, no pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	2	20	0
$\mathbf{P}_1$	0	3	0
$\mathbf{P}_2$	1	3	0

yielding



which can be evaluated as follows:

throughput	=	3/26	=	0.12
av. turn-around time	=	(26 + 3 + 6) / 3	=	11.67
av. waiting time	=	(4 + 0 + 2) / 3	=	2.00
av. response time	=	(6 + 0 + 3) / 3	=	3.00

Notes:

Scheduling Algorithms (4) – SJF

- Idea: **Shortest Job First (SJF)**, i.e.,
  - assign a priority based inversely proportional on remaining time,
  - schedule process with highest priority in ready queue

  st.

  - good: can operate with or without pre-emption,
  - good: with global knowledge of processes, minimises waiting time,
  - bad: requires accurate estimate duration!

Notes:

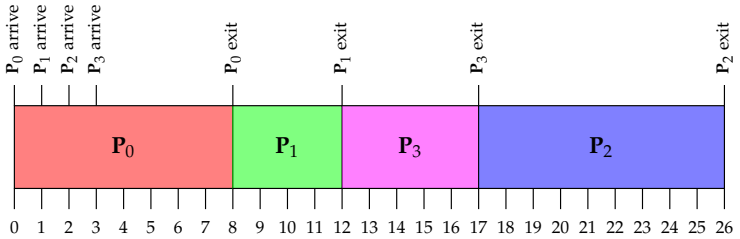
- It makes sense to rename the pre-emptive variant of SJF as **Shortest *Remaining* Time Left**, because a process will no longer necessarily execute to completion: this implies a choice, at each pre-emption point, based on the remaining time rather than total duration.
- The point about global knowledge is important, but maybe somewhat subtle. The idea is that if the algorithm has accurate information about every possible process, then it can minimise waiting time; this requirement is hard to achieve, however, since a) we need to estimate duration, and b) the arrival time of processes may differ (e.g., at time t, the algorithm does not know about a process due to arrive at time $t + 1$).

Example (SJF, no pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	0	8	0
$\mathbf{P}_1$	1	4	0
$\mathbf{P}_2$	2	9	0
$\mathbf{P}_3$	3	5	0

yielding



which can be evaluated as follows:

throughput	=	4/26	=	0.15
av. turn-around time	=	(8 + 12 + 26 + 17) / 4	=	15.75
av. waiting time	=	(0 + 7 + 15 + 9) / 4	=	7.75
av. response time	=	(0 + 8 + 17 + 12) / 4	=	9.25

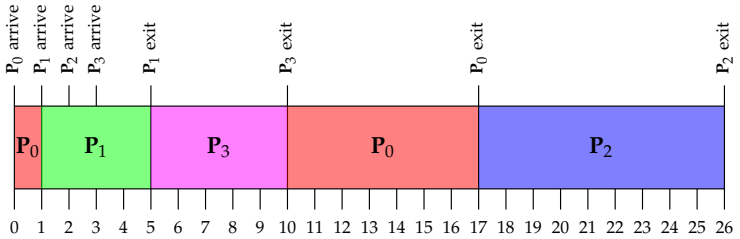
Notes:

Example (SJF, with arrival pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	0	8	0
$\mathbf{P}_1$	1	4	0
$\mathbf{P}_2$	2	9	0
$\mathbf{P}_3$	3	5	0

yielding



which can be evaluated as follows:

throughput	=	4/26	=	0.15
av. turn-around time	=	(17 + 5 + 26 + 10) / 4	=	14.50
av. waiting time	=	(9 + 0 + 15 + 2) / 4	=	6.50
av. response time	=	(0 + 1 + 17 + 5) / 4	=	5.75

Notes:

Scheduling Algorithms (6) – static-priority

- Idea: **static-priority**, i.e.,
  - allow user to define per-process priorities,
  - schedule process with highest priority in ready queue

  st.
  - good: can operate with or without pre-emption,
  - bad: can lead to starvation, so ideally need some form of **dynamic-priority** (e.g., **ageing**).

Notes:

- As already alluded to by the title, priorities can be classified in various different ways. For example, they may be
  - static or dynamic, meaning they stay fixed or can change over time, or
  - internal or external, meaning they stem from the system (e.g., memory requirements) or user.
- Different presentations of priority-based scheduling make different choices wrt. whether a small (resp. large) number means high (resp. low) priority, and what meaning (if any) positive and negative numbers have. For instance, the convention in UNIX-like kernels is that a "niceness" value represents the user-defined priority level: −20 and 19 are the highest and lowest priority respectively, with the default being 0. Here we've opted for the (hopefully) more intuitive and certainly simply "larger means higher".
- Various common sense approaches to priority ageing are possible: incrementing or "boosting" the priority of processes that sit in the ready queue but are not selected by the scheduler would solve the starvation issue (eventually). The issue of assigning and managing priorities is tricky, however, in so far as corner cases such as **priority inversion** [3] can occur.

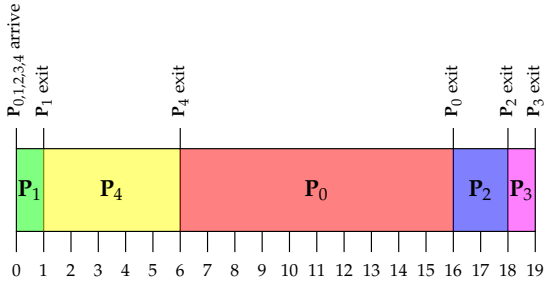
Scheduling Algorithms (7) – static-priority

Example (static-priority, no pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	0	10	3
$\mathbf{P}_1$	0	1	5
$\mathbf{P}_2$	0	2	2
$\mathbf{P}_3$	0	1	1
$\mathbf{P}_4$	0	5	4

yielding



which can be evaluated as follows:

throughput	=	5/19	=	0.26
av. turn-around time	=	(16 + 1 + 18 + 19 + 6) / 5	=	12.00
av. waiting time	=	(6 + 0 + 16 + 18 + 1) / 5	=	8.20
av. response time	=	(6 + 0 + 16 + 18 + 1) / 5	=	8.20

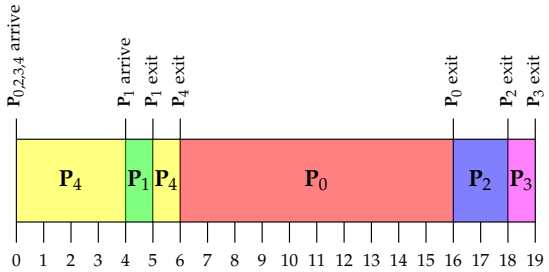
Notes:

Example (static-priority, with arrival pre-emption)

Consider the processes

Process	Arrive	Burst	Priority
$\mathbf{P}_0$	0	10	3
$\mathbf{P}_1$	4	1	5
$\mathbf{P}_2$	0	2	2
$\mathbf{P}_3$	0	1	1
$\mathbf{P}_4$	0	5	4

yielding



which can be evaluated as follows:

throughput	=	5/19	=	0.26
av. turn-around time	=	(16 + 5 + 18 + 19 + 6) / 5	=	12.80
av. waiting time	=	(6 + 0 + 16 + 18 + 1) / 5	=	8.20
av. response time	=	(6 + 4 + 16 + 18 + 0) / 5	=	8.80

Notes:

- Idea: **round-robin**, i.e.,
  - fix **time quantum** (or **time slice**) τ; pre-empt every τ time units,
  - schedule processes via cyclic ready queue

  st.

  - good: each process gets $1/n$ share of processor in chunks of τ time units,
  - good: no process waits longer than $\tau \cdot (n-1)$ time units,
  - bad: need to carefully select τ, since
    - too short means many context switches, so high overhead, while
    - too long means poor responsiveness, degrading to FCFS.

Notes:

- The choice of τ clearly depends on the context, but, to be at least a little more concrete, a rough guide might be 10ms to 100ms. Modulo responsiveness, a longer quantum will act to amortise the cost of context switches; when it becomes shorter, this overhead starts to dominate, until the point at which the processor spends more time context switching than doing useful computation!

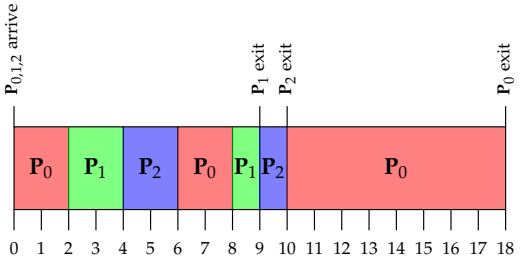
Scheduling Algorithms (9) – round-robin

Example (round-robin, quantum = 2, context switch cost = 0)

Consider the processes

Process	Arrive	Burst	Priority
P_0	0	12	0
P_1	0	3	0
P_2	0	3	0

yielding



which can be evaluated as follows:

throughput	=	3/18	=	0.17
av. turn-around time	=	(18 + 9 + 10) / 3	=	12.33
av. waiting time	=	(6 + 6 + 7) / 3	=	6.33
av. response time	=	(0 + 2 + 4) / 3	=	2.00

Notes:

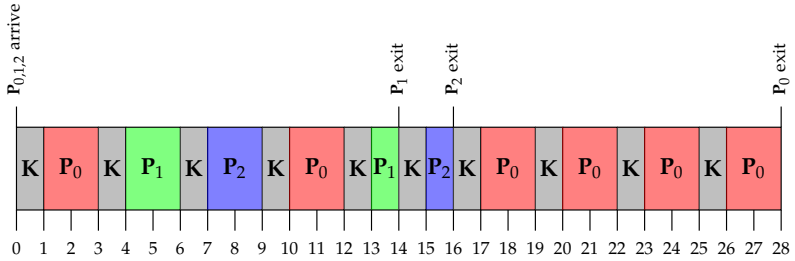
Scheduling Algorithms (9) – round-robin

Example (round-robin, quantum = 2, context switch cost = 1)

Consider the processes

Process	Arrive	Burst	Priority
P_0	0	12	0
P_1	0	3	0
P_2	0	3	0

yielding



which can be evaluated as follows:

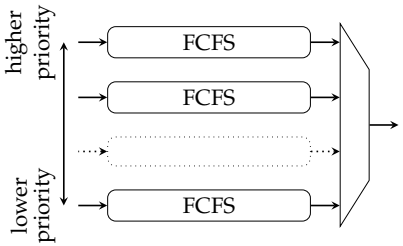
throughput	=	3/28	=	0.11
av. turn-around time	=	(28 + 14 + 16) / 3	=	19.33
av. waiting time	=	(16 + 11 + 13) / 3	=	13.33
av. response time	=	(1 + 4 + 7) / 3	=	4.00

Notes:

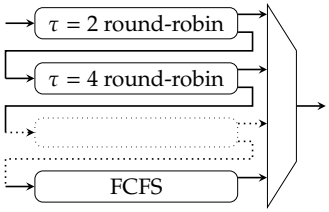
- Idea: **Multi-level Queue Scheduling (MQS)** and **Multi-level Feedback Queue Scheduling (MFQS)**, i.e.,
  - maintain multiple levels of ready queue, each potentially managed using a different scheduling algorithm,
  - select processes based on scheduling decisions between those queues, e.g., using **priority classes**

e.g.,

Example (MQS)



Example (MFQS)



st. MQS demands static decision re. queue choice, whereas MFQS allows dynamic movement between queues.

Notes:

- MQS basically groups processes into **priority classes**, st. a given class will contain processes with similar properties (e.g., wrt. burst and wait behaviour) and requirements.

Conclusions

Quote

An application must be able to manage itself, either as a single process or as multiple processes. Applications must be able to manage other processes when appropriate.

Applications must be able to identify, control, create, and delete processes, and there must be communication of information between processes and to and from the system.

Applications must be able to use multiple flows of control with a process (threads) and synchronize operations between these flows of control.

– POSIX [4, Section D1.2]

Notes:

Conclusions

- Take away points:
  - This is a broad and complex topic: it involves (at least)
    1. a hardware aspect:
       - an interrupt controller,
       - a timer
    2. a low(er)-level software aspect:
       - an interrupt handler,
       - a dispatcher algorithm
    3. a high(er)-level software aspect:
       - some data structures (e.g., process table),
       - a scheduling algorithm,
       - any relevant POSIX system calls (e.g., `fork`)
  - Keep in mind that, even then,
    - we've excluded and/or simplified various (sub-)topics,
    - there are numerous trade-offs involved, meaning it is often hard to identify one ideal solution.

Notes:

- A non-exhaustive list of topics *not* covered, or covered in a more superficial level than ideal, include
  - other scheduling algorithms (e.g., lottery [7], or Linux O(1) [2] or CFS [1]),
  - scheduling algorithms with advanced requirements (e.g., in the context of real-time constraints, or for multi-/many-core processors),
  - a range of additional POSIX functionality.

  Some of these *are* covered in the recommended reading: see in particular [6, 5].

References

[1] Wikipedia: Completely Fair Scheduler.
http://en.wikipedia.org/wiki/Completely_Fair_Scheduler.

[2] Wikipedia: O(1) scheduler.
http://en.wikipedia.org/wiki/O(1)_scheduler.

[3] Wikipedia: Priority inversion.
http://en.wikipedia.org/wiki/Priority_inversion.

[4] Standard for information technology - portable operating system interface (POSIX).
Institute of Electrical and Electronics Engineers (IEEE) 1003.1, 2008.
http://standards.ieee.org/findstds/standard/1003.1-2008.html.

[5] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 5: Process scheduling.
In *Operating System Concepts*. Wiley, 9th edition, 2014.

[6] A.S. Tanenbaum and H. Bos.
Chapter 2.4: Sheduling.
In *Modern Operating Systems*. Pearson, 4th edition, 2015.

[7] C.A. Waldspurger and W.E. Weihl.
Lottery scheduling: flexible proportional-share resource management.
In *Operating Systems Design and Implementation (OSDI)*, number 1, 1994.

Notes: