# Concurrent Computing (Operating Systems)

Daniel Page

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

February 9, 2018

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

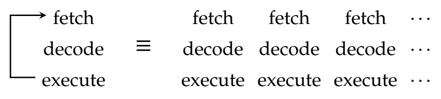
1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material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

Notes:

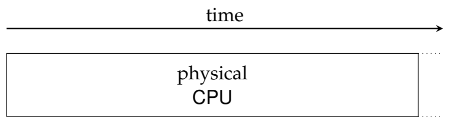
Notes:

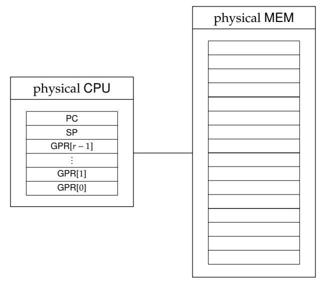
## Concept: *virtualise* the processor

- 1 process *does* have dedicated access to the physical processor.
- We know execution is st.



i.e.,



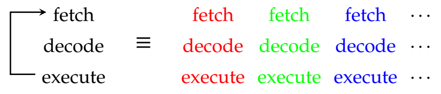


Notes:

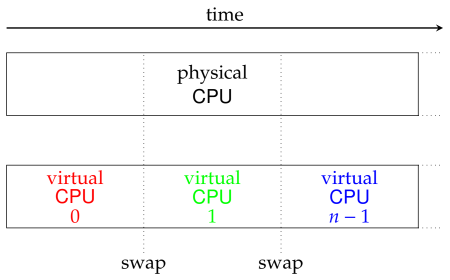
- By saying the physical processor swaps between processes, we basically mean it ensures the right **execution context** at the right time: if, for example, the physical processor uses the PC for some *i*-th process, the next instruction fetched and then executed will therefore be associated with *that* process rather than some other. This is the essence of **multi-tasking**, which can be described in terms of *multiplexing* the *n* processes onto the 1 processor.
- Since we use the POSIX standard as an example throughout, you might want to refer to the specific (and extensive) associated definitions *it* uses: pertinent examples include
  - program [14, Section 3.300],
  - process [14, Section 3.289],
  - thread [14, Section 3.396], and
  - signal [14, Section 3.344].

## Concept: *virtualise* the processor

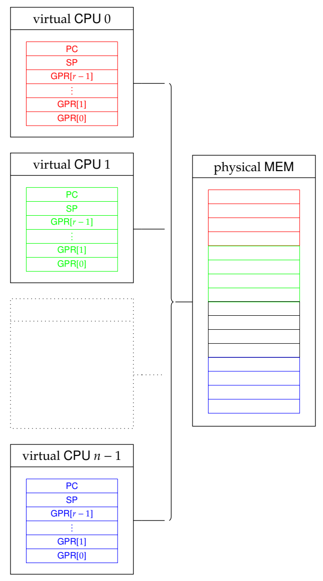
- *n* processes *cannot* have dedicated access to the physical processor ...
- ... but *if* execution could be st.



i.e.,



then they'd *appear* to.



Notes:

- By saying the physical processor swaps between processes, we basically mean it ensures the right **execution context** at the right time: if, for example, the physical processor uses the PC for some *i*-th process, the next instruction fetched and then executed will therefore be associated with *that* process rather than some other. This is the essence of **multi-tasking**, which can be described in terms of *multiplexing* the *n* processes onto the 1 processor.
- Since we use the POSIX standard as an example throughout, you might want to refer to the specific (and extensive) associated definitions *it* uses: pertinent examples include
  - program [14, Section 3.300],
  - process [14, Section 3.289],
  - thread [14, Section 3.396], and
  - signal [14, Section 3.344].

## Concept (1)

**Definition**

The terms **uni-programming** and **multi-programming** are used, respectively, to describe cases where one or many programs execute simultaneously. In the latter case, execution could be

- parallel (or *truly*-parallel), e.g., as realised via **multi-processing**, or
- concurrent (or *pseudo*-parallel), e.g., as realised via **multi-tasking**.

Notes:

- The terminology here can be confusing, and is made less exact in certain contexts. The basic idea is as follows:
  - Under a multi-programming kernel, one or more programs are resident in memory (at the same time); at a given point in time, any one can be executed. This contrasts with uni-programming, where only one program can ever be resident and hence executed.
  - The similar sounding terms uni- and multi-*processing* relate to hardware rather than software: a multi-processing system will have many physical processors.
  - As such, multi-processing is one way to realise multi-programming. Another way is multi-*tasking*, where many programs share one given processor: this is essentially what **time-sharing** or **time-slicing** means.

## Concept (2)

**Definition**

A **process** is an active instance of a given, passive program image. Each process constitutes

1. $n \geq 1$ **execution contexts** (viz. **threads**), each for an independent instruction stream, *plus*
2. associated state, i.e.,
   - an address space, and
   - a set of resources

   which is shared between them.

**Definition**

A **context switch** is the act of, or mechanism for, changing the active execution context: performing a context switch will typically involve

- suspending execution of one process, then
- resuming execution of another process.

Notes:

- The terminology to describe processes can change a little depending on the context. For example, in batch processing systems it is common to hear the term **job** instead (also occurring in various UNIX-related contexts, e.g., the BASH job control commands). Also note that Linux uses the term **task** specifically st. traditional meaning implied by the terms process and thread can be avoided.
  In addition, keep in mind that processes and threads are may be termed heavy-weight and light-weight processes respectively: the reason to do so, and hence the meaning of the terms, depends on the context to some extent. One interpretation is that a light-weight process is a thread supported in user space. Another is simply that because threads share the resources of an associated process, their representation includes less (than said process) and so is lighter-weight; they are typically easier to create for the same reason (given no need to allocate said resources).
- A more intuitive way to separate the concepts of processes and threads, is via their role: the former is really an entity used to group related resources that support execution, whereas the latter is an entity that captures an independent instruction stream *being* (or at least having the potential to be) executed by the processor. Or, put another way, you could think of them as a trade-off:
  - processes offer more isolation, but a less complex, less flexible abstraction of execution, whereas
  - threads offer less isolation, but a more complex, more flexible abstraction of execution.
- If you want, you could indulge in some program vs. process philosophy:
  - A process is "*more than*" a program, since the former is a (stateful) instance of the latter: there can be more than one process all stemming from the same program (e.g., more than one instance of `emacs`).
  - A program is "*more than*" a process, in the sense the former captures all things that could happen, but the latter captures a specific case of what *is* happening; it is sometimes true that one program forms multiple processes (e.g., executing `gcc` might execute `cpp`, `cc1` and so on).

  Either way, there are some obviously sane reasons for wanting to support processes at all. For example, they allow

  - explicit expression of concurrent tasks, which is a) convenient (e.g., enables multiple users to share hardware resources), and b) efficient (i.e., often leads to greater utilisation of hardware resources),
  - explicit expression of divide-and-conquer style tasks,
  - explicit isolation of one task from another.

## Mechanism: POSIX(ish) system call interface (1) – representation

- Each process is represented by the kernel
  - in a **process table**,
  - each entry in which is a data structure termed a **Process Control Block (PCB)**

  e.g.,

| Process management | Memory management | Resource management |
|---|---|---|
| processor state | MMU state | user ID |
| process ID | text segment info. | group ID |
| process status | data segment info. | working directory |
| process hierarchy | stack segment info. | file descriptors |
| scheduling info. | | |
| signalling info. | | |
| accounting info. | | |
| ⋮ | ⋮ | ⋮ |

  noting the entries are
  - *very* kernel- and hardware-specific (so these are *examples* only), and
  - divided into per-process and per-thread.

Notes:

- A process, user and group ID are almost always acronym'ised as PID, UID and GID; depending on how they are implemented, thread-specific versions might also exist, e.g., TID for thread ID.
- It should be clear that for a thread to be deemed independent, there must be a) an independent PC (and register file state more generally), plus b) an independent SP and stack region. This fact is reflected in how the entries are split between per-process and per-thread.
  The latter requirement may not be obvious, but exists for essentially the same reason we maintain a stack frame for nested function call instances: the frame, or **activation record**, captures the state of that instance in the same way. That is, if a caller instance calls some other callee function, the latter is allocated a new frame; when the callee returns and unwinds the callee frame, the old caller frame is intact, meaning the caller instance can continue executing. The same is true of threads, in the sense that if execution of one thread is suspended and another resumed, we want the latter thread to resume in the same state as when it was last suspended, *not* corrupted by whatever the last executing thread was.

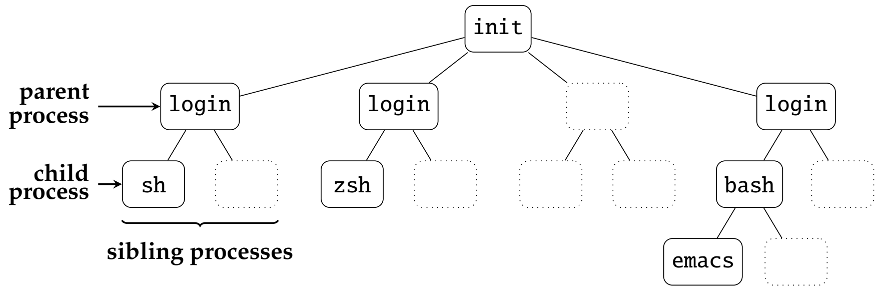
## Mechanism: POSIX(ish) system call interface (2) – representation

- POSIX says processes are organised
  1. into a **process hierarchy** [14, Sections 3.93 and 3.264], namely a tree, *and*
  2. **process groups** [14, Section 3.290] can be formed, e.g., to support collective communication.

Notes:

## Mechanism: POSIX(ish) system call interface (2) – representation

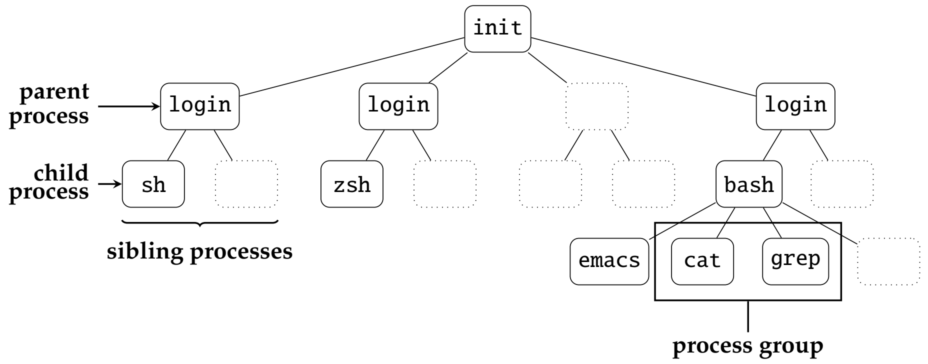
- Example:
  - we have three logged-in users,
  - one of whom is executing an instance of `emacs`.



Notes:

## Mechanism: POSIX(ish) system call interface (2) – representation

- Example:
  - we have three logged-in users,
  - one of whom is executing an instance of `emacs`,
  - and *then* executes `cat foo.txt | grep bar`



Notes:

## Mechanism: POSIX(ish) system call interface (3) – representation

**Definition**

An **orphaned process** is one which is still executing even though it has no parent (i.e., the parent has terminated).

**Definition**

A **zombie process** (or **defunct process**) is one whose parent has (been) terminated, but still has an active PCB.

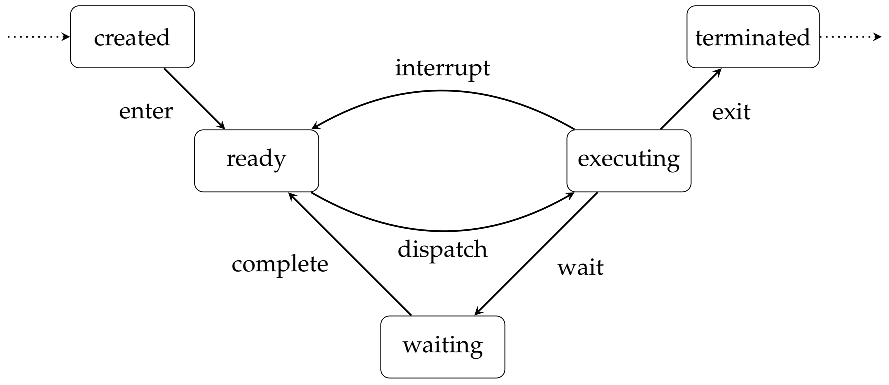
**Definition**

A **foreground process** (resp. **background process**) executes with (resp. without) direct interaction with a user. Uses for a background process (which may be termed a **daemon process**) include logging or maintenance services.

Notes:

## Mechanism: POSIX(ish) system call interface (4) – representation

- As execution progresses, the **process status** changes



under control of a **scheduler**, which tracks processes via **scheduling queues**, e.g.,

- $1 \times$ **ready queue** : processes that can be executed
- $n \times$ **waiting queue** : processes whose execution is blocked

Notes:

- There is no definitive set of correct terms for nodes and edges in an FSM-like diagram of this sort: the FSM itself, and said terms are often dependent on the kernel in question and level of detail. For instance, you can see a simpler FSM in [13, Figure 2-2] and different terms in [10, Figure 3.2]. The rationale for those shown is as follows:
  - The terms ready and executing are often replaced by runnable and running; given the use of execute vs. run elsewhere, the terms used here are mainly for consistency, with ready indicating readiness for execution vs. *being* executed.
  - The terms dispatch and interrupt might be confusing. The former is due to the fact the short-term scheduler uses the so-called the dispatcher to pass control to the process it has selected, i.e., perform the actual context switch; the latter captures the fact execution is interrupted, which *is* often because of an interrupt occurring, but also conflates the two unnecessarily to some extent.
    Either way, it is important to see that dispatch vs. schedule is a separation of mechanism from policy: dispatching is doing the context switch to have some process executed next, while scheduling is deciding which process that is.

  Likewise, other descriptions may use process state to refer to what is here termed process *status*; the reason for sticking carefully to the latter is to be consistent. The process *state* is *all* (stateful) information associated with it, whereas the *status* is the specific item of information which details, for example, whether or not it is ready for execution.
- The scheduling queues are typically implemented as a linked list of PCBs; the process table (which includes a PCB for every process) could also be implemented as a similar list, which is sometimes termed a **job queue** or **task queue**. In addition, a pointer to the process (clearly there can only be one) currently with executing status, and hence being executed, would typically maintained: this allows direct access to said PCB, which is crucial to minimise overhead during a context switch.

## Mechanism: POSIX(ish) system call interface (5) – creation

- A process may be created at
  - implicitly, at boot-time (e.g., `init`), *or*
  - explicitly, at run-time.

Notes:

- It is important to see that the semantics of `fork` are *one* option among a design space of options wrt.
  1. execution, i.e.,
     - the parent process waits for the child process to terminate before continuing to execute, *or*
     - the parent process continues to execute concurrently with the child process

     which can be interpreted as meaning process creation is blocking or non-blocking respectively, and
  2. address space, i.e.,
     - the child process is initialised using a duplicate of the parent address space, *or*
     - the child process is initialised with a new address space

     which you could summarise as meaning the semantics are to clone an existing process, or simple create a new one from scratch.
- Although it is a specific example, `fork` highlights some reasons that process creation is, more generally, a costly operation. An obvious reason is that (in "traditional UNIX" at least), explicitly creating a copy of the parent address space is a *huge* overhead; fortunately, it can be mitigated by techniques such as **copy-on-write** as supported by virtual memory.
- By studying the manual pages for `fork` and `exec`, e.g.,

  ```
  man exec
  ```

  it should become clear where `argc` and `argv` arguments to a standard `main` function come from: you may have (correctly) thought of these as copies of the command line argument, but given that the command line interpreter (e.g., instance of `bash`) is an executing process using `fork` to create new processes, it is in fact `exec` which loads the image including `main` and then invokes it with the arguments.
- In Windows, the `CreateProcess` system call more or less combines `fork` and `exec` into one: for each process created, it *demands* a new image is also loaded. Also in contrast to Linux, the Windows process model is st. no hierarchy is enforced: although a parent process still creates a child process, the handle held by the former for the latter can be passed freely to another process (i.e., there is no relationship and hence hierarchy enforced).

## Mechanism: POSIX(ish) system call interface (5) – creation

- POSIX says we need
  - `fork` [14, Page 881]:
    - create new child process with unique PID,
    - replicate state (e.g., address space) of parent in child,
    - parent and child *both* return from `fork`, and continue to execute after the call point,
    - return value is 0 for child, and PID of child for parent.
  - `exec` [14, Page 772] and friends:
    - replace current process image (e.g., text segment) with with new process image: effectively this means execute a new program,
    - reset state (e.g., stack pointer); continue to execute at the entry point of new program,
    - no return, since call point no longer exists.

  where a **loader** [5] performs various tasks related to the latter.

Notes:

- It is important to see that the semantics of `fork` are *one* option among a design space of options wrt.
  1. execution, i.e.,
     - the parent process waits for the child process to terminate before continuing to execute, *or*
     - the parent process continues to execute concurrently with the child process

     which can be interpreted as meaning process creation is blocking or non-blocking respectively, and
  2. address space, i.e.,
     - the child process is initialised using a duplicate of the parent address space, *or*
     - the child process is initialised with a new address space

     which you could summarise as meaning the semantics are to clone an existing process, or simple create a new one from scratch.
- Although it is a specific example, `fork` highlights some reasons that process creation is, more generally, a costly operation. An obvious reason is that (in "traditional UNIX" at least), explicitly creating a copy of the parent address space is a *huge* overhead; fortunately, it can be mitigated by techniques such as **copy-on-write** as supported by virtual memory.
- By studying the manual pages for `fork` and `exec`, e.g.,

  ```
  man exec
  ```

  it should become clear where `argc` and `argv` arguments to a standard `main` function come from: you may have (correctly) thought of these as copies of the command line argument, but given that the command line interpreter (e.g., instance of `bash`) is an executing process using `fork` to create new processes, it is in fact `exec` which loads the image including `main` and then invokes it with the arguments.
- In Windows, the `CreateProcess` system call more or less combines `fork` and `exec` into one: for each process created, it *demands* a new image is also loaded. Also in contrast to Linux, the Windows process model is st. no hierarchy is enforced: although a parent process still creates a child process, the handle held by the former for the latter can be passed freely to another process (i.e., there is no relationship and hence hierarchy enforced).

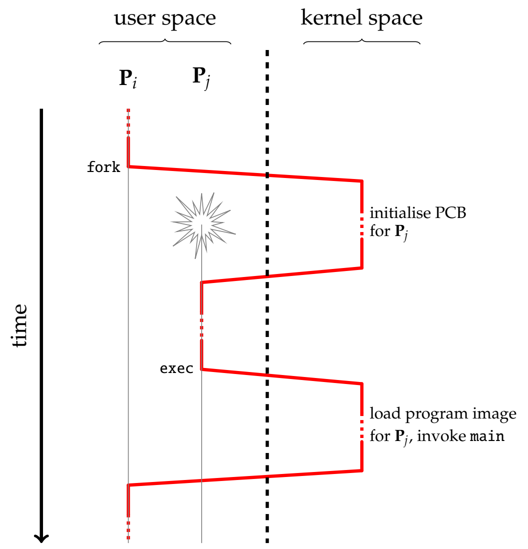
## Mechanism: POSIX(ish) system call interface (6) – creation

Listing

```
int main( int argc, char* argv[] ) {
  pid_t pid = fork();

  if      ( pid >  0 ) { // parent = P_i
    ...
  }
  else if( pid == 0 ) { // child  = P_j
    exec( ... );
  }
  else if( pid <  0 ) { // error
    abort();
  }

  return 0;
}
```


user space
kernel space
$\mathbf{P}_i$
$\mathbf{P}_j$
time
fork
initialise PCB for $\mathbf{P}_j$
exec
load program image for $\mathbf{P}_j$, invoke main

Notes:

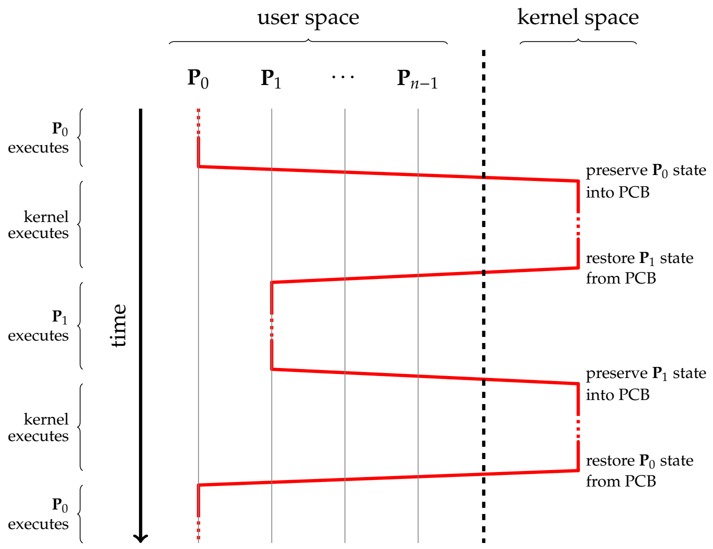
## Mechanism: POSIX(ish) system call interface (7) – control

- POSIX says *we* need
  - `wait` [14, Page 2181]:
    - suspend execution until process (or group thereof) terminates,
    - receive error status of said process.
  - `sleep` [14, Page 1963]:
    - suspend execution for specified time period,
    - close to, but not quite `yield`.

  *plus* the *kernel* needs to be able to context switch ...

Notes:

## Mechanism: POSIX(ish) system call interface (8) – control



Notes:

- As the diagram shows, but doesn't outright explain, efficiency of context switches is important: they represent pure overhead [15] wrt. useful execution by any of the user space processes.
- The normal definition of a context switch implicitly applies to user space processes: the idea is we suspend one process and resume another. However, one could interpret the term context more generally: an interrupt invokes a change of execution context, in the sense it causes a switch from user to kernel mode.
  The subtle, yet vitally important implication is that the kernel also has an execution context of sorts: this includes an address space, which, for example, houses a kernel text, data and stack segment(s) and allows the kernel to execute instructions and hence functions in the same way a user process does. As a result, *each* mode is typically initialised with an independent stack region. When the kernel is initialised, it will set the value of SP, for each mode, to some address st. that mode can create and unwind stack frames just like a "normal" program.
- The diagram illustrates the fact that realising each context switch needs two mode switches (to enter and leave kernel mode), but also performs some additional steps (e.g., saving and restoring state to and from PCBs). It *must* be realised in kernel mode, because only the kernel is able to access *all* the state (including any protected resources) associated with each process. A mode and context switch *are* related, but also different concepts: the former relates more to hardware (the processor) while the latter relates more to software (the processes).

## Mechanism: POSIX(ish) system call interface (9) – termination

- A process may terminate due to
  - exit,
  - controlled error,
  - uncontrolled error, or
  - signal

  which can be classified as normal, abnormal and external events.

Notes:

- The concepts of orphaned and zombie processes both relate to semantics of process termination. Note, for example, that
  - Some kernels prevent orphaned processes from existing by enforcing application of **cascading termination**: if the parent terminates, the (sub-)tree of child processes are implicitly terminated as well.
  - Linux in fact allows orphaned processes, in the sense it forces the `init` process to "adopt" them.
  - When a process terminates via `exit`, it can return an exit status (or code) to the parent; the parent process receives this via `wait`. The child process PCB must be retained until the parent invokes `wait`, because the PCB houses associate state *including* the exit status. So if the parent fails to do so, the child is a zombie.
- Returning from `main` (which, recall, has an `int` return type normally) has the same semantics as controlled termination using `exit`. The `EXIT_SUCCESS` and `EXIT_FAILURE` constants defined `stdlib.h` (symbolically) represent two obvious exit statuses, but a zero (resp. non-zero) are more generally interpreted as success (resp. failure).
- Whenever a process terminates due to a signal, the kernel will manage the signal-based IPC mechanism so *it* terminates the process. The signal *source* may be another process *or* the kernel itself, however: the former depends on the source process having permission to signal the target. Although a range of situations might prompt the OS to terminate a process (which hasn't caused an error per se), the Linux Out-Of-Memory (OOM) mechanism, an overview of which is presented by

  `http://linux-mm.org/OOM_Killer`

  is a good example.

## Mechanism: POSIX(ish) system call interface (9) – termination

- POSIX says we need
  - `exit` [14, Page 785]:
    - perform normal termination,
    - invoke call-backs, flush then close open files,
    - pass exit status to parent process (via `wait`).
  - `abort` [14, Page 556]:
    - perform abnormal termination.

  where, in both cases, the associated PCB is (eventually) removed.

Notes:

- The concepts of orphaned and zombie processes both relate to semantics of process termination. Note, for example, that
  - Some kernels prevent orphaned processes from existing by enforcing application of **cascading termination**: if the parent terminates, the (sub-)tree of child processes are implicitly terminated as well.
  - Linux in fact allows orphaned processes, in the sense it forces the `init` process to "adopt" them.
  - When a process terminates via `exit`, it can return an exit status (or code) to the parent; the parent process receives this via `wait`. The child process PCB must be retained until the parent invokes `wait`, because the PCB houses associate state *including* the exit status. So if the parent fails to do so, the child is a zombie.
- Returning from `main` (which, recall, has an `int` return type normally) has the same semantics as controlled termination using `exit`. The `EXIT_SUCCESS` and `EXIT_FAILURE` constants defined `stdlib.h` (symbolically) represent two obvious exit statuses, but a zero (resp. non-zero) are more generally interpreted as success (resp. failure).
- Whenever a process terminates due to a signal, the kernel will manage the signal-based IPC mechanism so *it* terminates the process. The signal *source* may be another process *or* the kernel itself, however: the former depends on the source process having permission to signal the target. Although a range of situations might prompt the OS to terminate a process (which hasn't caused an error per se), the Linux Out-Of-Memory (OOM) mechanism, an overview of which is presented by

  `http://linux-mm.org/OOM_Killer`

  is a good example.

## An Aside: Threads vs. Processes

**Definition**

If a process has $n > 1$ threads, it is said to be **multi-threaded**; those threads might be realised in

- user space (viz. a **user thread**) yielding an $n$-to-1 mapping, or in
- kernel space (viz. a **kernel thread**) yielding a 1-to-1 mapping, or
- a hybrid, yielding an $n$-to-$m$ mapping

from threads to kernel-exposed **schedulable entities**.

**Definition**

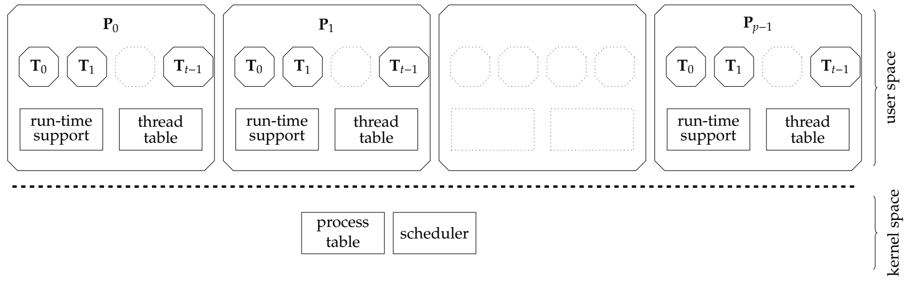
A **software thread** (resp. **hardware thread**) uses software (resp. hardware) resources to capture each execution context.

Notes:

- Fundamentally, the difference between user or kernel threads boils down to the definition of what a schedulable entity is. If the kernel "knows about" threads (i.e., they are realised in kernel space), then threads are the entity controlled by the scheduler. If not (i.e., they are realised in user space), then processes are the entity controlled by the scheduler: the user threads are managed in user space, with the kernel oblivious of this fact.
- Ungerer et al. [16] present a good survey of processor design options relating to hardware multi-threading. Use of such an approach can be summarised as allowing concurrent execution of instructions from various *different* threads using *one* execution pipeline; this suggests support for multiple register files in hardware (on per thread).

## An Aside: Threads vs. Processes – User vs. Kernel Threads

- Roughly per [13, Sections 2.24-2.26]



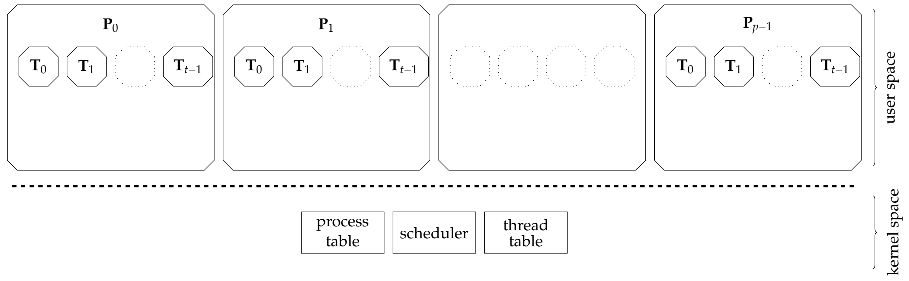
one can identify advantages for both

- user thread implementation, e.g.,
  - can port to a kernel *without* thread support,
  - can implement process-specific policies,
  - lower-overhead creation and context switch,
  - *all* threads are blocked by blocking system call
  - ...

Notes:

## An Aside: Threads vs. Processes – User vs. Kernel Threads

- Roughly per [13, Sections 2.24-2.26]



one can identify advantages for both

- kernel thread implementation, e.g.,
  - blocking system calls are thread-specific,
  - easier to cope with interrupts,
  - `kill` and `fork` need different semantics.
  - ...

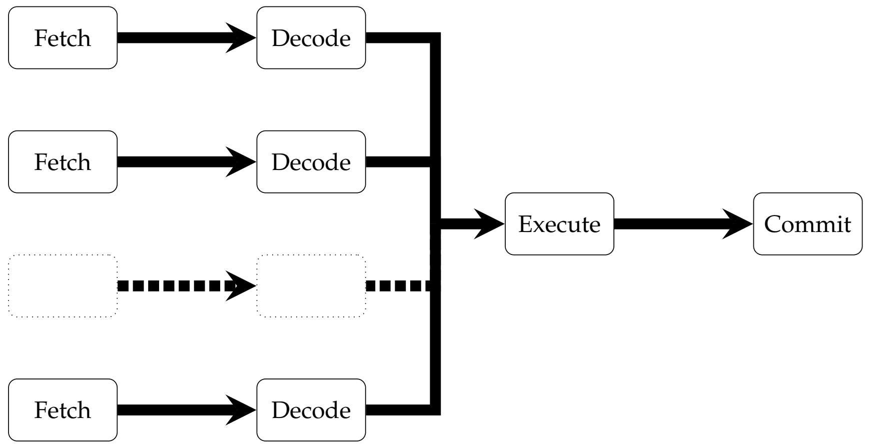
Notes:

- Most implementations of x86-32 and x86-64 are
  - *deeply* pipelined, and/or
  - *highly* superscalar,

  for example the Pentium 4 has roughly 20 pipeline stages ...
- .. this has (at least) two implications: the
  1. demand for instructions to execute is high, and
  2. cost of a pipeline stall or flush is high.

Notes:

- Idea: **Simultaneous MultiThreading (SMT)**, or Intel Hyper-Threading (e.g., for Pentium 4):



  i.e.,
  - replicate parts of the processor that house execution context, and thus
  - improve utilisation of (later) execution unit(s).

Notes:

- From an architectural viewpoint, SMT gives some clear advantages. In particular, a superscalar design suffers acutely from the von Neumann bottleneck: the impact of memory latency makes it hard to keep the execution units at a high level of utilisation. Allowing two threads to be interleaved potentially allows a better instruction mix, and so the potential of latency hiding (e.g., by one compute-bound thread of another memory-bound thread).

## Concept (1)

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

$$\begin{array}{rcl}\textbf{mechanism} & \Rightarrow & \textbf{dispatcher} \\ \textbf{policy} & \Rightarrow & \textbf{scheduler}\end{array}$$

we need to answer the following questions:

1. when should the **scheduler** be invoked, and
2. which **scheduling algorithm** should it use, or, given the ready queue

$$Q = \{\mathbf{P}_i \mid 0 \le i < n\}$$

which $\mathbf{P}_i$ should be selected for execution.

Notes:

- Generalising further still, another component, namely an **allocator**, might be used to decide *where* a process scheduled for execution is executed: this is clearly a choice that might need to be made within the context of multi-core processors or distributed systems, for example.
- What happens if $Q = \emptyset$, i.e., there is *no* process on the ready queue to select?! One way to resolve this corner-case is via a dummy, **idle process** designed to occupy the processor (e.g., via an infinite loop of NOP instructions) until a real process later becomes ready: this is arguably easier than using a special-case in the scheduling algorithm.
- Another corner-case is where $Q = \{\mathbf{P}_i\}$, i.e., the ready queue has one entry: the question here is whether or not it makes sense to *continue to* invoke the scheduler, the intuition being that once the single $\mathbf{P}_i$ has been executed we can simply let it execute-to-completion at which point there is nothing else to do. The mistake here is that $Q$ *only* captures the ready queue at the point we examine it. It *could be* that a process moves from having a waiting to ready status (e.g., it was waiting for an I/O operation to complete, and then said operation *did* complete): we would expect such a $\mathbf{P}_j$ to be considered by the scheduler, so clearly need to (continue to) invoke it.

## Concept (2)
### Question #1: scheduler invocation

**Definition**

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 mix of processes

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

Notes:

- Based on the definition of pre-emptible and non pre-emptible resources, pre-emptive vs. co-operative multi-tasking is a natural distinction wrt. the processor: access to the processor is only pre-emptible, so as to permit pre-emptive multi-tasking, iff. the kernel can revoke access or take it away from the process it was previously allocated to.
  It should be obvious revoking and allocating said resource will require a mode switch, st. the kernel executes: if no such switch can be forced, the executing process will simply continue to execute because the kernel cannot interrupt it! As such, opportunities for kernel to perform scheduling 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 occurs

  with a (fully) pre-emptive strategy implying use of all four, and a co-operative strategy implying use of the first two.
  Even so, 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 interrupts will always occur regularly, and hence that a mode switch occurs st. scheduling can take place.
- Co-operative multi-tasking is usually 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 that does not co-operate, e.g., never invokes `yield`, can monopolise the processor and hence prevent other processes from executing.
- 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. That is, *when* the scheduler is invoked is largely orthogonal to what it does (i.e., the algorithm).

## Concept (2)
Question #1: scheduler invocation

**Definition**

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.

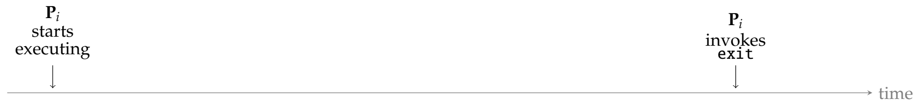
Notes:

- Based on the definition of pre-emptible and non pre-emptible resources, pre-emptive vs. co-operative multi-tasking is a natural distinction wrt. the processor: access to the processor is only pre-emptible, so as to permit pre-emptive multi-tasking, iff. the kernel can revoke access or take it away from the process it was previously allocated to.
  It should be obvious revoking and allocating said resource will require a mode switch, st. the kernel executes: if no such switch can be forced, the executing process will simply continue to execute because the kernel cannot interrupt it! As such, opportunities for kernel to perform scheduling 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 occurs

  with a (fully) pre-emptive strategy implying use of all four, and a co-operative strategy implying use of the first two.
  Even so, 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 interrupts will always occur regularly, and hence that a mode switch occurs st. scheduling can take place.
- Co-operative multi-tasking is usually 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 that does not co-operate, e.g., never invokes `yield`, can monopolise the processor and hence prevent other processes from executing.
- 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. That is, *when* the scheduler is invoked is largely orthogonal to what it does (i.e., the algorithm).

## Concept (3)
Question #1: scheduler invocation

- Idea: we *could* invoke the scheduler when



1. a process terminates, i.e., **execute-to-completion**,

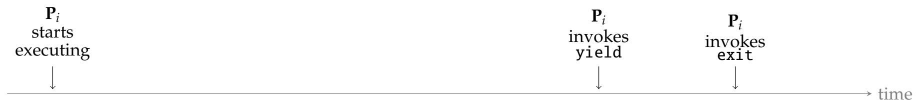
Notes:

- At face value, one might argue there is no difference between intentional and unintentional co-operation: both stem from the kernel and therefore scheduler being able to execute as the result of a system call, and the subsequent switch from user to kernel mode. However, the process status (wrt. the process invoking the system call) could differ in practice: in the former case the process *can* be executed but opts not to be, and as a result remains on the ready queue, whereas, in the former case, the process could actually be blocked so *cannot* be executed.
- The choice of $\tau$ clearly depends on the context: there is no single, optimal value. In particular, it influences the trade-off between overhead and responsiveness in the sense that
  - a larger $\tau$ implies worse responsiveness, but the overhead of context switching is amortised. whereas
  - a smaller $\tau$ implies better responsiveness, but the overhead of context switching starts to dominate: at an extreme, the processor will spend longer context switching than doing useful computation!

  Since it is unrealistic to assume a single, static choice of $\tau$ is ideal for every situation, keep in mind that it *is* possible to make the selection on a dynamic basis instead. One way to prioritise execution of one process over another is to execute it for longer: this can clearly be supported by selecting $\tau$ on a per-process basis.

## Concept (3)
Question #1: scheduler invocation

- Idea: we *could* invoke the scheduler when


$\mathbf{P}_i$ starts executing
$\mathbf{P}_i$ invokes yield
$\mathbf{P}_i$ invokes exit
time

1. a process terminates, i.e., **execute-to-completion**,
2. a process *intentionally* co-operates, e.g., via a `yield` system call,

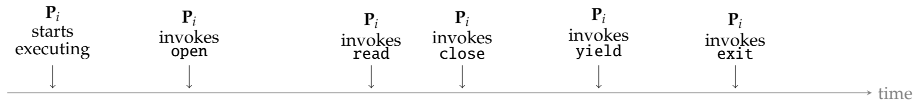
Notes:

- At face value, one might argue there is no difference between intentional and unintentional co-operation: both stem from the kernel and therefore scheduler being able to execute as the result of a system call, and the subsequent switch from user to kernel mode. However, the process status (wrt. the process invoking the system call) could differ in practice: in the former case the process *can* be executed but opts not to be, and as a result remains on the ready queue, whereas, in the former case, the process could actually be blocked so *cannot* be executed.
- The choice of $\tau$ clearly depends on the context: there is no single, optimal value. In particular, it influences the trade-off between overhead and responsiveness in the sense that
  - a larger $\tau$ implies worse responsiveness, but the overhead of context switching is amortised. whereas
  - a smaller $\tau$ implies better responsiveness, but the overhead of context switching starts to dominate: at an extreme, the processor will spend longer context switching than doing useful computation!

  Since it is unrealistic to assume a single, static choice of $\tau$ is ideal for every situation, keep in mind that it *is* possible to make the selection on a dynamic basis instead. One way to prioritise execution of one process over another is to execute it for longer: this can clearly be supported by selecting $\tau$ on a per-process basis.

## Concept (3)
Question #1: scheduler invocation

- Idea: we *could* invoke the scheduler when


$\mathbf{P}_i$ starts executing
$\mathbf{P}_i$ invokes open
$\mathbf{P}_i$ invokes read
$\mathbf{P}_i$ invokes close
$\mathbf{P}_i$ invokes yield
$\mathbf{P}_i$ invokes exit
time

1. a process terminates, i.e., **execute-to-completion**,
2. a process *intentionally* co-operates, e.g., via a `yield` system call,
3. a process *unintentionally* co-operates, e.g., via a `read` system call,

Notes:

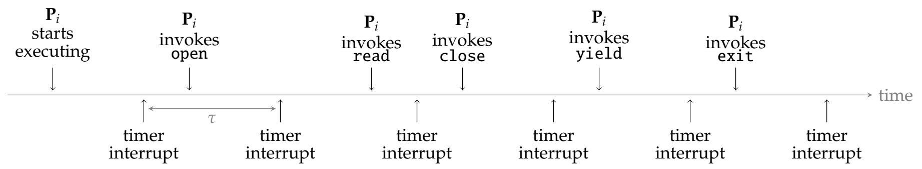
- At face value, one might argue there is no difference between intentional and unintentional co-operation: both stem from the kernel and therefore scheduler being able to execute as the result of a system call, and the subsequent switch from user to kernel mode. However, the process status (wrt. the process invoking the system call) could differ in practice: in the former case the process *can* be executed but opts not to be, and as a result remains on the ready queue, whereas, in the former case, the process could actually be blocked so *cannot* be executed.
- The choice of $\tau$ clearly depends on the context: there is no single, optimal value. In particular, it influences the trade-off between overhead and responsiveness in the sense that
  - a larger $\tau$ implies worse responsiveness, but the overhead of context switching is amortised. whereas
  - a smaller $\tau$ implies better responsiveness, but the overhead of context switching starts to dominate: at an extreme, the processor will spend longer context switching than doing useful computation!

  Since it is unrealistic to assume a single, static choice of $\tau$ is ideal for every situation, keep in mind that it *is* possible to make the selection on a dynamic basis instead. One way to prioritise execution of one process over another is to execute it for longer: this can clearly be supported by selecting $\tau$ on a per-process basis.

## Concept (3)

### Question #1: scheduler invocation

- **Idea**: we *could* invoke the scheduler when



1. a process terminates, i.e., **execute-to-completion**,
2. a process *intentionally* co-operates, e.g., via a `yield` system call,
3. a process *unintentionally* co-operates, e.g., via a `read` system call,
4. a pre-organised interrupt is requested:
    - fix a **time quantum** (or **time slice**) $\tau$,
    - configure a (hardware) **timer** st. an interrupt is requested every $\tau$ time units.

Notes:

- At face value, one might argue there is no difference between intentional and unintentional co-operation: both stem from the kernel and therefore scheduler being able to execute as the result of a system call, and the subsequent switch from user to kernel mode. However, the process status (wrt. the process invoking the system call) could differ in practice: in the former case the process *can* be executed but opts not to be, and as a result remains on the ready queue, whereas, in the former case, the process could actually be blocked so *cannot* be executed.
- The choice of $\tau$ clearly depends on the context: there is no single, optimal value. In particular, it influences the trade-off between overhead and responsiveness in the sense that
    - a larger $\tau$ implies worse responsiveness, but the overhead of context switching is amortised. whereas
    - a smaller $\tau$ implies better responsiveness, but the overhead of context switching starts to dominate: at an extreme, the processor will spend longer context switching than doing useful computation!

  Since it is unrealistic to assume a single, static choice of $\tau$ is ideal for every situation, keep in mind that it *is* possible to make the selection on a dynamic basis instead. One way to prioritise execution of one process over another is to execute it for longer: this can clearly be supported by selecting $\tau$ on a per-process basis.

## Concept (4)

### Question #2: scheduler algorithm

**Definition**

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**

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**

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:

- 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.

## Concept (5)
### Question #2: scheduler algorithm

**Definition**

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:

## Concept (6)
### Question #2: scheduler algorithm

- Challenge: given

$$Q = \{\mathbf{P}_i \mid 0 \leq i < n\}$$

and potentially other input such as

1. what has happened $\Rightarrow$ record previous behaviour
2. what will happened $\Rightarrow$ estimate future behaviour
3. what should happen $\Rightarrow$ user input

select a $\mathbf{P}_i$ to optimise

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

Notes:

- A batch system is an example of where the normally challenging issue of estimating what will happen (in the future) is easy: the processes that need to be executed are specified and hence known beforehand. As such, it might be reasonable to employ execute-to-completion: the kernel can (pre-)compute then use a schedule for execution, and execute processes to completion with the only intervention (and hence overhead resulting from context switches) occurring when one exits and another needs to be created.
- The quality metrics are meant to illustrate one reason why an effective *general-purpose* scheduling algorithm is illusive: 1) we have numerous (sometimes mutually exclusive) metrics, and 2) said metrics *plus* the workload wrt. processes might change dynamically.
- The sorts of criteria that may need to be considered include
  - goals
    - fairness,
    - liveness,
    - ...
  - metrics
    - efficiency,
    - 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 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 the ready queue versus actually executing (i.e., *not* the time spent blocked in a waiting queue: it is “waiting to execute” not “waiting for resource”),
  - response time is the number of time units between the arrival of and first output from a given process (where first output is interpreted as first time it executes, i.e., moves from ready to executing).

## Implementation (1)
### Algorithms

- (Some) idea(s): at the $j$-th scheduling algorithm invocation, select $\mathbf{P}_i$ st.

1. **random** $\Rightarrow i \overset{\$}{\leftarrow} \{0, 1, \ldots n-1\}$
2. **round-robin** $\Rightarrow i \leftarrow j \pmod{n}$
3. **First-Come First-Served (FCFS)** $\Rightarrow i \leftarrow \underset{0 \le k < n}{\arg\min}\ \mathbf{P}_k[\text{arrival time}]$
4. **Shortest Job First (SJF)** $\Rightarrow i \leftarrow \underset{0 \le k < n}{\arg\min}\ \mathbf{P}_k[\text{remaining time}]$
5. **priority-based** $\Rightarrow i \leftarrow \underset{0 \le k < n}{\arg\max}\ \mathbf{P}_k[\text{priority}]$
6. $\cdots \Rightarrow \cdots$

Notes:

- Notice that a pre-emptive variant of round-robin means each process will get a $1/n$ share of processor in chunks of $\tau$ time units, and no process will ever wait for longer than $\tau \cdot (n-1)$ time units.
- For FCFS, the arrival time has a significant impact (which is obvious in the sense it dictates how "first come" is interpreted); typical waiting times under FCFS are high.
- The idea of SJF is to order processes based (inversely) on an estimation of remaining execution time. *If* the kernel had perfect and global knowledge in this respect, it could in fact *minimise* waiting time. This requirement is obviously hard to achieve, however, since a) we need to estimate duration, which by definition will be imperfect, and b) the arrival time of processes has an impact (e.g., at time $t$, the algorithm does not know about a process due to arrive at time $t+1$).
- 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.
- A **priority** is simply some explicit information (typically a number) that captures the importance of one process relative to another wrt. the scheduling problem: in a sense they provide an abstraction *of* the scheduling problem. They can be classified in different ways, e.g. as being
  - static or dynamic, meaning it remains fixed or can change over time, or
  - internal or external, meaning it stems from the system or user.

  For clarity, it is useful to introduce the term **effective priority** and distinguish this carefully from the above: in essence, this is what the scheduler uses to prioritise one process over another, i.e., select one process rather than another for execution. As such, you could think about it as the output from *any* scheduling algorithm, whereas, in contrast, a priority as defined above is the input to a priority-based scheduling algorithm which makes specific use of it to compute said output. This implies the effective priority is decoupled from (because it is computed as a function of) other information, *including* the priorities as defined above.
- Note that a priority-based scheduler needs to make a choice 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: $-20$ (resp. 19) is the highest (resp. lowest) priority (with the default being 0). Here we opt for the more intuitive and certainly simply "larger means higher".

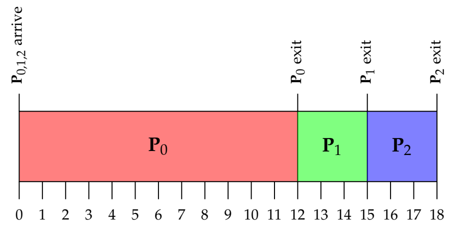
## Implementation (2)
### Algorithms

**Example (round-robin, no pre-emption, context switch cost: 0)**

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| $\mathbf{P}_0$ | 0 | 12 | $\perp$ |
| $\mathbf{P}_1$ | 0 | 3 | $\perp$ |
| $\mathbf{P}_2$ | 0 | 3 | $\perp$ |

yielding



which can be evaluated as follows:

$$
\begin{array}{lclcl}
\text{throughput} & = & 3/18 & = & 0.17 \\
\text{av. turn-around time} & = & (\ 12 + 15 + 18\ )\ /\ 3 & = & 15.00 \\
\text{av. waiting time} & = & (\ 0 + 12 + 15\ )\ /\ 3 & = & 9.00 \\
\text{av. response time} & = & (\ 0 + 12 + 15\ )\ /\ 3 & = & 9.00
\end{array}
$$

Notes:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

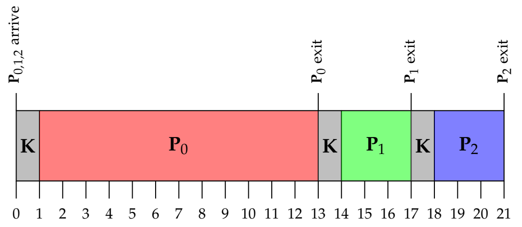
## Implementation (2)
Algorithms

### Example (round-robin, no pre-emption, context switch cost: 1)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 0 | 12 | $\perp$ |
| **P**$_1$ | 0 | 3 | $\perp$ |
| **P**$_2$ | 0 | 3 | $\perp$ |

yielding



which can be evaluated as follows:

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| throughput | = | 3/21 | | | | | | | | | = | 0.14 |
| av. turn-around time | = | ( | 13 | + | 17 | + | 21 | ) | / | 3 | = | 17.00 |
| av. waiting time | = | ( | 1 | + | 14 | + | 18 | ) | / | 3 | = | 11.00 |
| av. response time | = | ( | 1 | + | 14 | + | 18 | ) | / | 3 | = | 11.00 |

Notes:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

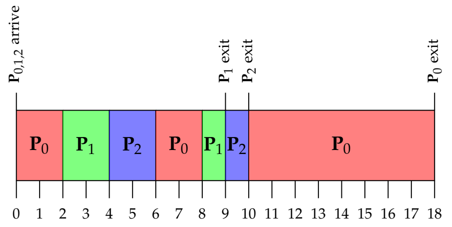
## Implementation (2)
Algorithms

### Example (round-robin, timer pre-emption: $\tau = 2$, context switch cost: 0)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 0 | 12 | $\perp$ |
| **P**$_1$ | 0 | 3 | $\perp$ |
| **P**$_2$ | 0 | 3 | $\perp$ |

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:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

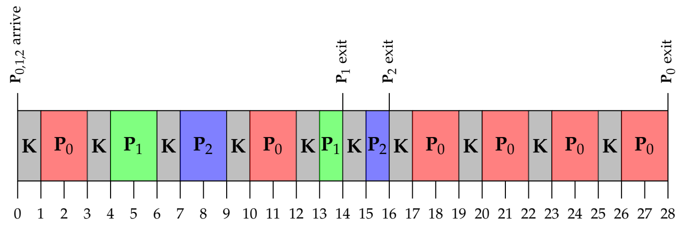
## Implementation (2)
Algorithms

### Example (round-robin, timer pre-emption: $\tau = 2$, context switch cost: 1)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 0 | 12 | $\perp$ |
| **P**$_1$ | 0 | 3 | $\perp$ |
| **P**$_2$ | 0 | 3 | $\perp$ |

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:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

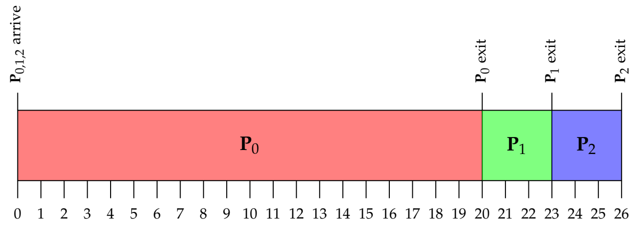
## Implementation (2)
Algorithms

### Example (FCFS, no pre-emption, context switch cost: 0)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 0 | 20 | $\perp$ |
| **P**$_1$ | 0 | 3 | $\perp$ |
| **P**$_2$ | 0 | 3 | $\perp$ |

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:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

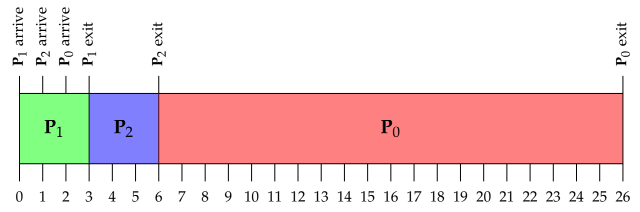
## Implementation (2)
### Algorithms

**Example (FCFS, no pre-emption, context switch cost: 0)**

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 2 | 20 | $\perp$ |
| **P**$_1$ | 0 | 3 | $\perp$ |
| **P**$_2$ | 1 | 3 | $\perp$ |

yielding



which can be evaluated as follows:

$$\begin{array}{lcl}
\text{throughput} & = & 3/26 & = & 0.12 \\
\text{av. turn-around time} & = & ( 26 + 3 + 6 ) / 3 & = & 11.67 \\
\text{av. waiting time} & = & ( 4 + 0 + 2 ) / 3 & = & 2.00 \\
\text{av. response time} & = & ( 6 + 0 + 3 ) / 3 & = & 3.00
\end{array}$$

Notes:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

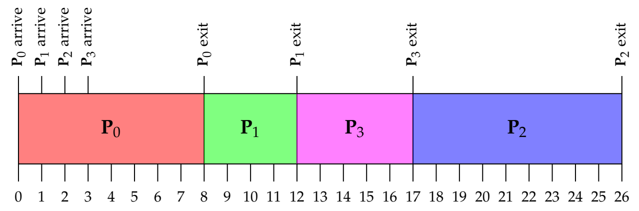
## Implementation (2)
### Algorithms

**Example (SJF, no pre-emption, context switch cost: 0)**

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| **P**$_0$ | 0 | 8 | $\perp$ |
| **P**$_1$ | 1 | 4 | $\perp$ |
| **P**$_2$ | 2 | 9 | $\perp$ |
| **P**$_3$ | 3 | 5 | $\perp$ |

yielding



which can be evaluated as follows:

$$\begin{array}{lcl}
\text{throughput} & = & 4/26 & = & 0.15 \\
\text{av. turn-around time} & = & ( 8 + 12 + 26 + 17 ) / 4 & = & 15.75 \\
\text{av. waiting time} & = & ( 0 + 7 + 15 + 9 ) / 4 & = & 7.75 \\
\text{av. response time} & = & ( 0 + 8 + 17 + 12 ) / 4 & = & 9.25
\end{array}$$

Notes:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

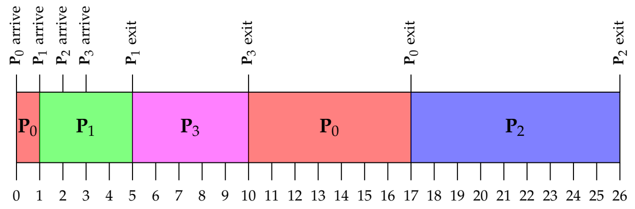
## Implementation (2)
Algorithms

### Example (SJF, arrival pre-emption, context switch cost: 0)

Consider the processes

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

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:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

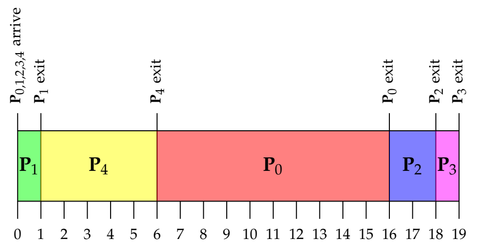
## Implementation (2)
Algorithms

### Example (priority-based, no pre-emption, context switch cost: 0)

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:

- Even if a more advanced algorithm than random or round-robin is actually used, these can still be useful: if the algorithm executes but produces a *set* of candidate processes rather than than a single process, a random or round-robin algorithm *could* then be used to select from the set (without minimal additional overhead). An obvious example is where several processes have the same priority, which a priority-based scheduler might therefore be unable to select between.

## Implementation (3)
Improvements

- Problem(s):
  1. since

     interactive $\simeq$ I/O-bound $\Rightarrow$ short CPU-bursts, many I/O-waits $\Rightarrow$ typically executes for $< \tau$

     non interactive $\simeq$ CPU-bound $\Rightarrow$ long CPU-bursts, few I/O-waits $\Rightarrow$ typically executes for $= \tau$

     a round-robin scheduler can be viewed as *penalising* an interactive process,
  2. under a priority-based scheduler, high-priority processes can monopolise the processor so cause starvation wrt. any low-priority processes.
- Solution(s): support dynamic priorities, and so (temporarily) "boost" the probability a given process is scheduled.

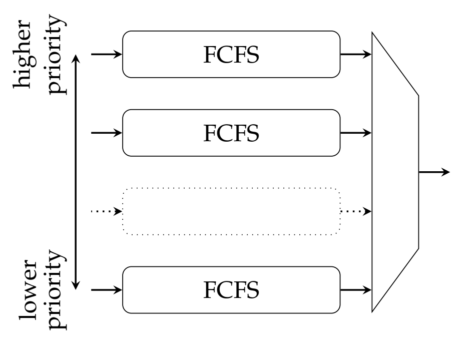
Notes:

- The cause of starvation is fairly obvious: if a high-priority process is *always* scheduled before a low-priority process and the priorities are static, any instance of the former will monopolise the processor and prevent an instance of the latter ever being executed (at least until it terminates).

## Implementation (4)
Improvements: multi-level scheduler algorithms

- Idea: **Multi-level Queue Scheduling (MQS)**.
  - maintain $l$ separate queues, potentially managed using different scheduling algorithms,
  - use a **scheduling class** to assign each process to a level upon arrival,
  - select a $\mathbf{P}_i$ from the highest non-empty level.

Example



Notes:

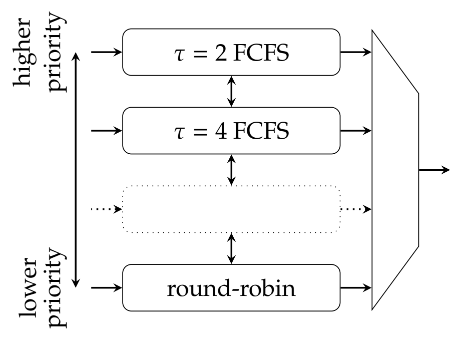
- The concept of a scheduler class is fairly intuitive: it represents a set of processes whose behaviour and/or requirements are all the same, with clear examples including classes of CPU- or I/O-bound processes.
- In general, the behaviour of MFQS is st. CPU-bound (resp. I/O-bound) processes tend to migrate toward the the lower (resp. higher) levels. This is what we wanted, in the sense it offers a boost to interactive processes that might otherwise be penalised. That is, having blocked (repeatedly) they are promoted to a higher level so the probability they will be selected for execution by the scheduler is higher (they get another chance vs. the chance they wasted by blocking and so being descheduled before their time quantum completed). Equally, interactive processes will often be blocked due to an I/O operation: this implies they are often not on the ready queue (so not eligible for selection), allowing any non-interactive processes in lower levels to be selected and hence execute.
- The per level scheduling algorithms are selected as part of an overall design, so these are examples only. However, the selection here is motivated by an intuitive argument: basically
  - the higher priority levels are FCFS because we want to optimise for responsiveness for interactive processes,
  - the lowest priority level is round robin because we want to permit some degree of fairness for other processes.
- Keep in mind that, as described at least, neither MQS nor MFQS provide a solution for starvation. That is, a process in the highest level is *still* able to monopolise the processor: assuming ti stays in the highest level, it will always be selected before some other process in a lower level. Various possibilities can address the problem directly, e.g., the processes can be rebalanced via a (randomised) reset of their levels.

## Implementation (4)

Improvements: multi-level scheduler algorithms

- Idea: **Multi-level Feedback Queue Scheduling (MFQS)**.
  - maintain $l$ separate queues, potentially managed using different scheduling algorithms,
  - use a **scheduling class** to assign each process to a level upon arrival,
  - select a $\mathbf{P}_i$ from the highest non-empty level,
  - allow processes to migrate between levels: if a process executes for
    - $< \tau$, promote to higher level
    - $= \tau$, demote to lower level

Example



Notes:

- The concept of a scheduler class is fairly intuitive: it represents a set of processes whose behaviour and/or requirements are all the same, with clear examples including classes of CPU- or I/O-bound processes.
- In general, the behaviour of MFQS is st. CPU-bound (resp. I/O-bound) processes tend to migrate toward the the lower (resp. higher) levels. This is what we wanted, in the sense it offers a boost to interactive processes that might otherwise be penalised. That is, having blocked (repeatedly) they are promoted to a higher level so the probability they will be selected for execution by the scheduler is higher (they get another chance vs. the chance they wasted by blocking and so being descheduled before their time quantum completed). Equally, interactive processes will often be blocked due to an I/O operation: this implies they are often not on the ready queue (so not eligible for selection), allowing any non-interactive processes in lower levels to be selected and hence execute.
- The per level scheduling algorithms are selected as part of an overall design, so these are examples only. However, the selection here is motivated by an intuitive argument: basically
  - the higher priority levels are FCFS because we want to optimise for responsiveness for interactive processes,
  - the lowest priority level is round robin because we want to permit some degree of fairness for other processes.
- Keep in mind that, as described at least, neither MQS nor MFQS provide a solution for starvation. That is, a process in the highest level is *still* able to monopolise the processor: assuming ti stays in the highest level, it will always be selected before some other process in a lower level. Various possibilities can address the problem directly, e.g., the processes can be rebalanced via a (randomised) reset of their levels.

## Implementation (5)

Improvements: preventing starvation in priority-based scheduling

- Idea: compute and use

$$\mathbf{P}_j[\text{priority}] = \underbrace{\mathbf{P}_j[\text{base priority}]}_{\text{static}} + \underbrace{\alpha(\mathbf{P}_j)}_{\text{dynamic}}$$

in the scheduling algorithm, where

$$\alpha(\mathbf{P}_j)$$

is the "**age**" of $\mathbf{P}_j$, i.e., the time spent waiting since last executed.

Notes:

- Note that the definition suggests that the age associated with a process is reset once it is executed, st. the effective priority reverts to the base priority every time the process executes.

## Implementation (6)

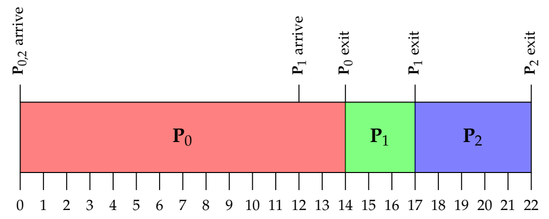
Improvements: preventing starvation in priority-based scheduling

### Example (priority-based, timer pre-emption: $\tau = 2$, context switch cost: 0)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| $\mathbf{P}_0$ | 0 | 14 | 7 |
| $\mathbf{P}_1$ | 12 | 3 | 3 |
| $\mathbf{P}_2$ | 0 | 5 | 1 |

yielding



Notes:

- The first example without ageing is meant to illustrate two points: 1) the high-priority $\mathbf{P}_0$ will monopolise the processor (in this case for 7 consecutive time quanta), a problem that might be exacerbated if it executed for longer or indefinitely, 2) although $\mathbf{P}_2$ has to wait for $\mathbf{P}_0$, it competes with $\mathbf{P}_1$ as well: the arrival time of $\mathbf{P}_1$ means it is the highest priority process once $\mathbf{P}_0$ terminates, so $\mathbf{P}_2$ *still* waits irrespective of the fact it was already waiting for quite some time.
- The plots on the right-hand side are a little confusing due to the colours used: the idea is that each one represents a process, i.e., $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ (from top-to-bottom). Within each plot, you can see a) the process base priority (which is a horizontal line), b) the process age (which grows in a stepped manner while the process exists in the ready queue), and c) the process effective priority (which is the sum of the base priority and age).

## Implementation (6)

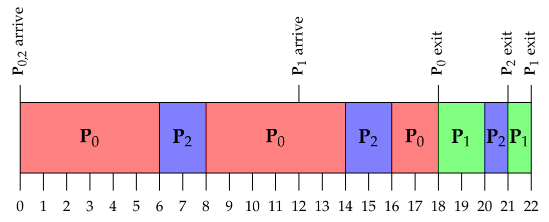
Improvements: preventing starvation in priority-based scheduling

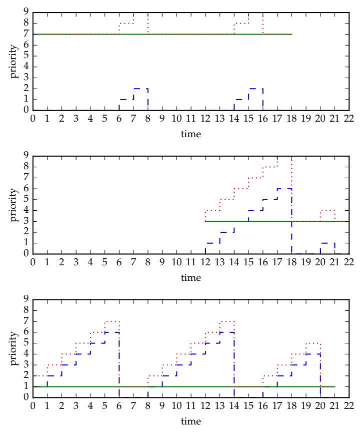
### Example (priority-based + ageing, timer pre-emption: $\tau = 2$, context switch cost: 0)

Consider the processes

| Process | Arrive | Burst | Priority |
|---|---|---|---|
| $\mathbf{P}_0$ | 0 | 14 | 7 |
| $\mathbf{P}_1$ | 12 | 3 | 3 |
| $\mathbf{P}_2$ | 0 | 5 | 1 |

yielding





Notes:

- The first example without ageing is meant to illustrate two points: 1) the high-priority $\mathbf{P}_0$ will monopolise the processor (in this case for 7 consecutive time quanta), a problem that might be exacerbated if it executed for longer or indefinitely, 2) although $\mathbf{P}_2$ has to wait for $\mathbf{P}_0$, it competes with $\mathbf{P}_1$ as well: the arrival time of $\mathbf{P}_1$ means it is the highest priority process once $\mathbf{P}_0$ terminates, so $\mathbf{P}_2$ *still* waits irrespective of the fact it was already waiting for quite some time.
- The plots on the right-hand side are a little confusing due to the colours used: the idea is that each one represents a process, i.e., $\mathbf{P}_0$, $\mathbf{P}_1$, and $\mathbf{P}_2$ (from top-to-bottom). Within each plot, you can see a) the process base priority (which is a horizontal line), b) the process age (which grows in a stepped manner while the process exists in the ready queue), and c) the process effective priority (which is the sum of the base priority and age).

## 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 [14, 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 device
    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 [17], 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 [12, 9].

## Additional Reading

- *Wikipedia: Process*. URL: http://en.wikipedia.org/wiki/Process_(computing).
- *Wikipedia: Scheduling*. URL: http://en.wikipedia.org/wiki/Scheduling_(computing).
- R. Love. "Chapter 5: Process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.
- R. Love. "Chapter 6: Advanced process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 3: Process concept". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 5: Process scheduling". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A.S. Tanenbaum and H. Bos. "Chapter 2.1: Processes". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.
- A.S. Tanenbaum and H. Bos. "Chapter 2.4: Sheduling". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.

Notes:

## References

[1] *Wikipedia: Completely Fair Sscheduler (CFS)*. URL: http://en.wikipedia.org/wiki/Completely_Fair_Scheduler (see p. 104).

[2] *Wikipedia: O(1) scheduler*. URL: http://en.wikipedia.org/wiki/O(1)_scheduler (see p. 104).

[3] *Wikipedia: Process*. URL: http://en.wikipedia.org/wiki/Process_(computing) (see p. 105).

[4] *Wikipedia: Scheduling*. URL: http://en.wikipedia.org/wiki/Scheduling_(computing) (see p. 105).

[5] J.R. Levine. *Linkers & Loaders*. Morgan-Kaufmann, 2000 (see pp. 25, 27).

[6] R. Love. "Chapter 5: Process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 105).

[7] R. Love. "Chapter 6: Advanced process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 105).

[8] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 3: Process concept". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 105).

[9] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 5: Process scheduling". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see pp. 104, 105).

[10] A. Silberschatz, P.B. Galvin, and G. Gagne. *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 24).

[11] A.S. Tanenbaum and H. Bos. "Chapter 2.1: Processes". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 105).

[12] A.S. Tanenbaum and H. Bos. "Chapter 2.4: Sheduling". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see pp. 104, 105).

[13] A.S. Tanenbaum and H. Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015 (see pp. 24, 41, 43).

[14] *Standard for Information Technology - Portable Operating System Interface (POSIX)*. Institute of Electrical and Electronics Engineers (IEEE) 1003.1-2008. 2008. URL: http://standards.ieee.org (see pp. 6, 8, 15, 17, 19, 25, 27, 31, 35, 37, 101).

[15] C. Li, C. Ding, and K. Shen. "Quantifying the cost of context switch". In: *Experimental Computer Science (ExpCS)*. 2. 2007 (see p. 34).

[16] T. Ungerer, B. Robič, and J. Šilc. "A survey of processors with explicit multithreading". In: *ACM Computing Surveys* 35.1 (2003), pp. 29–63 (see p. 40).

[17] C.A. Waldspurger and W.E. Weihl. "Lottery scheduling: flexible proportional-share resource management". In: *Operating Systems Design and Implementation (OSDI)*. 1. 1994 (see p. 104).

Notes: