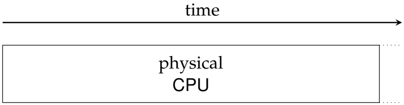
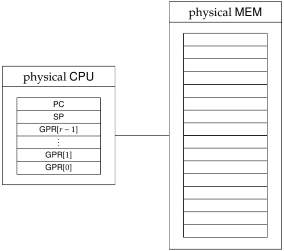
Concept: *virtualise* the processor

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

fetch		fetch	fetch	fetch	$\cdots$
decode	$\equiv$	decode	decode	decode	$\cdots$
execute		execute	execute	execute	$\cdots$

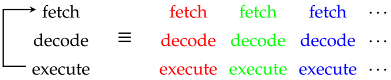
i.e.,



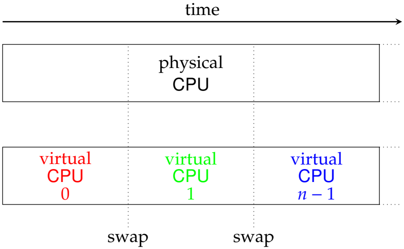


Concept: *virtualise* the processor

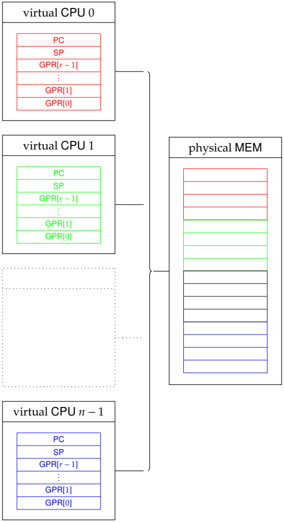
- n processes *cannot* have dedicated access to the physical processor ...
- ... but *if* execution could be st.



i.e.,



then they'd *appear* to.



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**.

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.

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

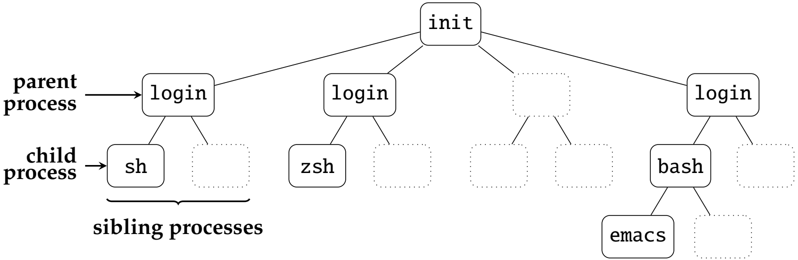
- *very* kernel- and hardware-specific (so these are *examples* only), and
- divided into per-process and per-thread.

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

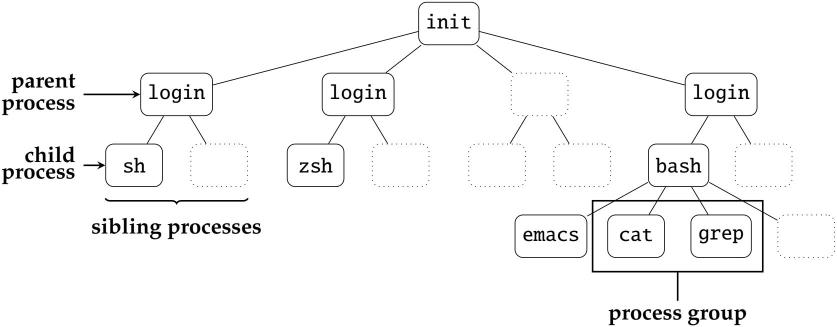
- POSIX says processes are organised
  1. into a **process hierarchy** [11, Sections 3.93 and 3.264], namely a tree, *and*
  2. **process groups** [11, Section 3.290] can be formed, e.g., to support collective communication.

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

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

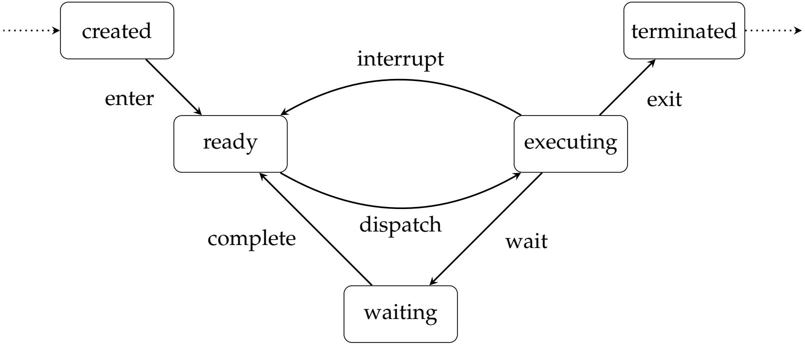


- Example:
  - we have three logged-in users,
  - one of whom is executing an instance of `emacs`,
  - and *then* executes `cat foo.txt | grep bar`



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

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.

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

- POSIX says we need
  - `fork` [11, Page 881]:
    - create new child process with unique PID,
    - replicate state, including
      - execution context (e.g., register content),
      - address space (e.g., stack segment),
      - ...

      of parent in child,
    - return from `fork` in parent *and* child processes, st. their return values are

$$\begin{array}{rcl} \text{parent} & \rightsquigarrow & \text{PID of child} \\ \text{child} & \rightsquigarrow & 0 \end{array}$$

  - `exec` [11, 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** [3] performs various tasks related to the latter.

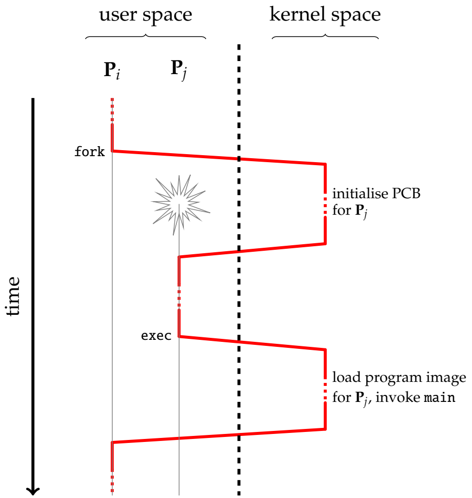
Mechanism: POSIX(ish) system call interface (6) – creation

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;
}
```

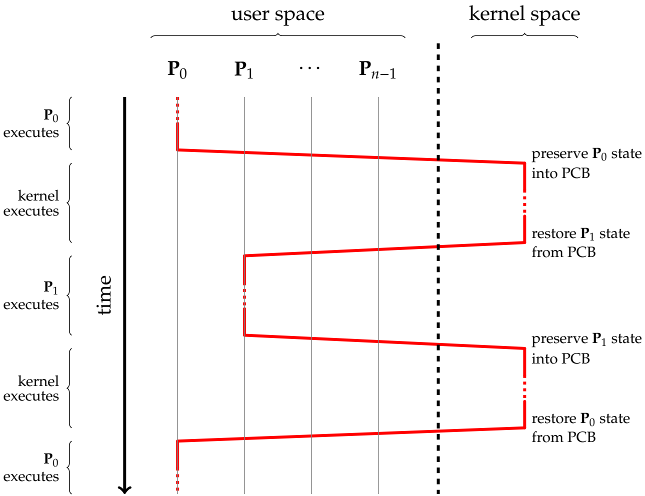


Mechanism: POSIX(ish) system call interface (7) – control

- POSIX says *we* need
  - `wait` [11, Page 2181]:
    - suspend execution until process (or group thereof) terminates,
    - receive error status of said process.
  - `sleep` [11, Page 1963]:
    - suspend execution for specified time period,
    - close to, but not quite `yield`.

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

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



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.

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

- POSIX says we need
  - `exit` [11, Page 785]:
    - perform normal termination,
    - invoke call-backs, flush then close open files,
    - pass exit status to parent process (via `wait`).
  - `abort` [11, Page 556]:
    - perform abnormal termination.

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

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.

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.

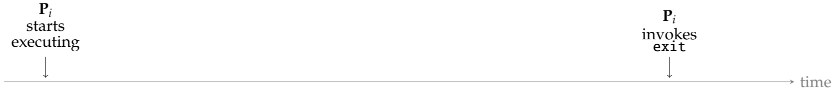
Definition

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.

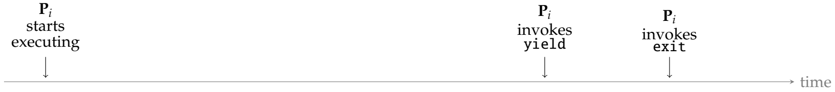
Concept (3)
Question #1: scheduler invocation

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



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

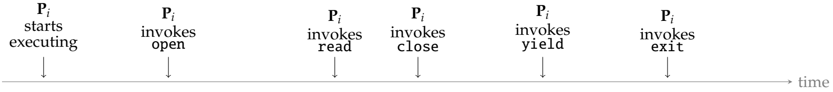
- Idea: we *could* invoke the scheduler when



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

Concept (3)
Question #1: scheduler invocation

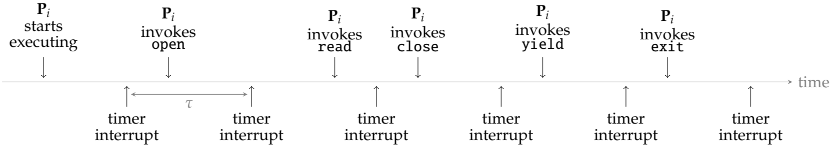
- **Idea**: we *could* invoke the scheduler when



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,

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**) τ,
   - configure a (hardware) **timer** st. an interrupt is requested every τ time units.

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**.

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.

- **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	✓	✓	✓	✓			✓		✓

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

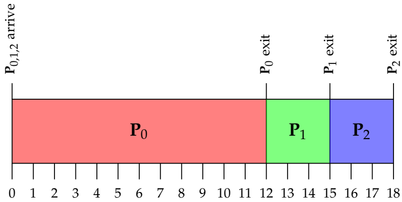
1. **random** $\Rightarrow$ $i \xleftarrow{\$} \{0, 1, \ldots n-1\}$
2. **round-robin** $\Rightarrow$ $i \leftarrow j \pmod{n}$
3. **First-Come First-Served (FCFS)** $\Rightarrow$ $i \leftarrow \arg\min_{0 \le k < n} \mathbf{P}_k[\text{arrival time}]$
4. **Shortest Job First (SJF)** $\Rightarrow$ $i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

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:

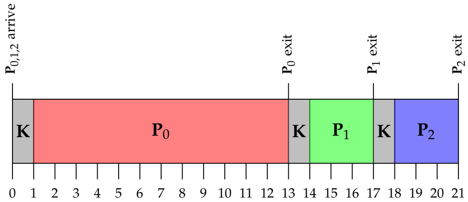
throughput	=	3/18										=	0.17
av. turn-around time	=	(	12	+	15	+	18	)	/	3		=	15.00
av. waiting time	=	(	0	+	12	+	15	)	/	3		=	9.00
av. response time	=	(	0	+	12	+	15	)	/	3		=	9.00

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

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:

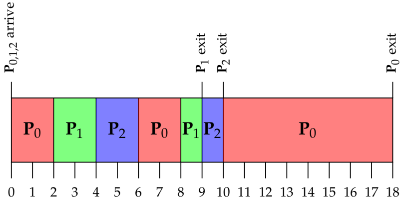
$$
\begin{array}{lcl}
\text{throughput} & = & 3/21 & & & & & & & & & = & 0.14 \\
\text{av. turn-around time} & = & (& 13 & + & 17 & + & 21 &) & / & 3 & = & 17.00 \\
\text{av. waiting time} & = & (& 1 & + & 14 & + & 18 &) & / & 3 & = & 11.00 \\
\text{av. response time} & = & (& 1 & + & 14 & + & 18 &) & / & 3 & = & 11.00
\end{array}
$$

Example (round-robin, timer pre-emption: $\tau = 2$, 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:

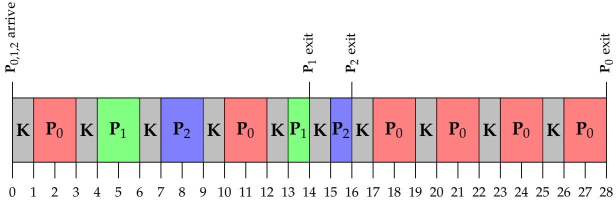
throughput	=	3/18										=	0.17
av. turn-around time	=	(	18	+	9	+	10	)	/	3		=	12.33
av. waiting time	=	(	6	+	6	+	7	)	/	3		=	6.33
av. response time	=	(	0	+	2	+	4	)	/	3		=	2.00

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

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:

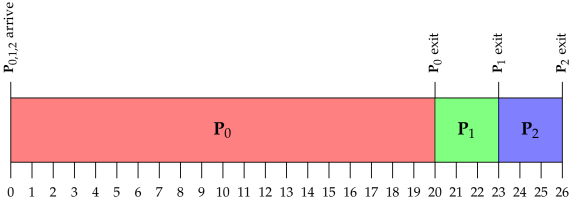
throughput	=	3/28									=	0.11
av. turn-around time	=	(	28	+	14	+	16	)	/	3	=	19.33
av. waiting time	=	(	16	+	11	+	13	)	/	3	=	13.33
av. response time	=	(	1	+	4	+	7	)	/	3	=	4.00

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

Consider the processes

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

yielding



which can be evaluated as follows:

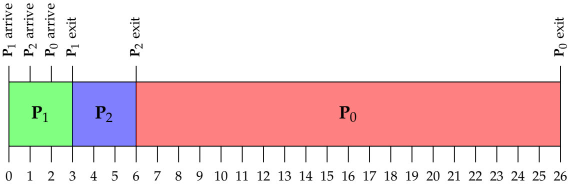
$$
\begin{array}{lcl}
\text{throughput} & = & 3/26 & & & & & & & & & = & 0.12 \\
\text{av. turn-around time} & = & (& 20 & + & 23 & + & 26 &) & / & 3 & = & 23.00 \\
\text{av. waiting time} & = & (& 0 & + & 20 & + & 23 &) & / & 3 & = & 14.33 \\
\text{av. response time} & = & (& 0 & + & 20 & + & 23 &) & / & 3 & = & 14.33
\end{array}
$$

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

Consider the processes

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

yielding



which can be evaluated as follows:

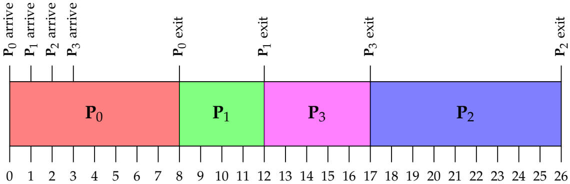
throughput	=	3/26									=	0.12
av. turn-around time	=	(	26	+	3	+	6	)	/	3	=	11.67
av. waiting time	=	(	4	+	0	+	2	)	/	3	=	2.00
av. response time	=	(	6	+	0	+	3	)	/	3	=	3.00

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:

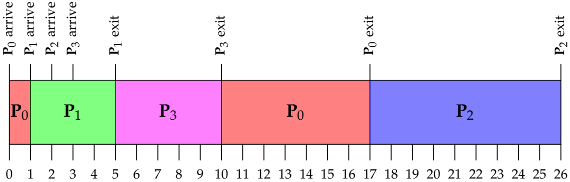
$$
\begin{array}{lcllllllllllll}
\text{throughput} & = & 4/26 & & & & & & & & & & = & 0.15 \\
\text{av. turn-around time} & = & (& 8 & + & 12 & + & 26 & + & 17 &) & / \; 4 & = & 15.75 \\
\text{av. waiting time} & = & (& 0 & + & 7 & + & 15 & + & 9 &) & / \; 4 & = & 7.75 \\
\text{av. response time} & =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

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:

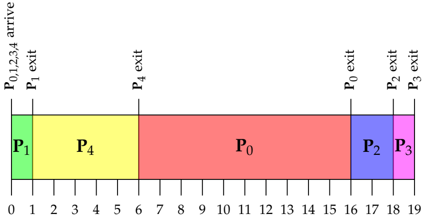
$$\begin{array}{lcccccccccccccc}
\text{throughput} & = & 4/26 & & & & & & & & & & & = & 0.15 \\
\text{av. turn-around time} & = & (& 17 & + & 5 & + & 26 & + & 10 &) & / & 4 & = & 14.50 \\
\text{av. waiting time} & = & (& 9 & + & 0 & + & 15 & + & 2 &) & / & 4 & = & 6.50 \\
\text{av. response time} & = & (& 0 & + & 1 & + & 17 & + & 5 &) & / & 4 & = & 5.75
\end{array}$$

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

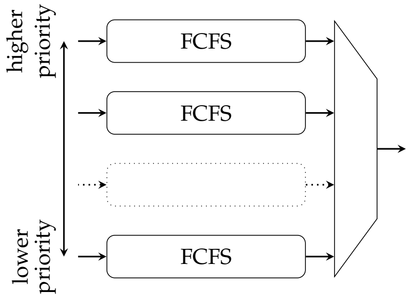
- **Problem(s)**:
  1. since

$$\begin{array}{lll} \text{interactive} & \simeq & \text{I/O-bound} \\ & \Rightarrow & \text{short CPU-bursts} \\ & & \text{many I/O-waits} \\ & \Rightarrow & \text{typically executes for} < \tau \end{array} \qquad \begin{array}{lll} \text{non interactive} & \simeq & \text{CPU-bound} \\ & \Rightarrow & \text{long CPU-bursts} \\ & & \text{few I/O-waits} \\ & \Rightarrow & \text{typically executes for} = \tau \end{array}$$

     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.

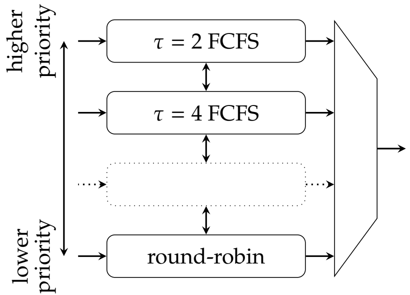
- Idea: **Multi-level Queue Scheduling (MQS)**.
  - maintain l separate queues, potentially managed using different scheduling algorithms,
  - use a **scheduling class** to assign each process to a level upon arrival,
  - select a $\mathbf{P}_i$ from the highest non-empty level.

Example



- 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



- 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

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

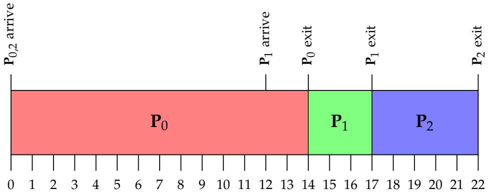
  is the "**age**" of $\mathbf{P}_j$, i.e., the time spent waiting since last executed.

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

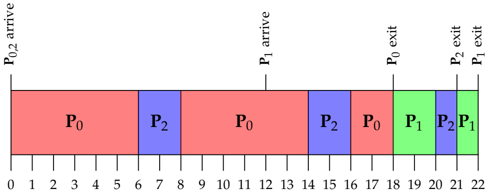


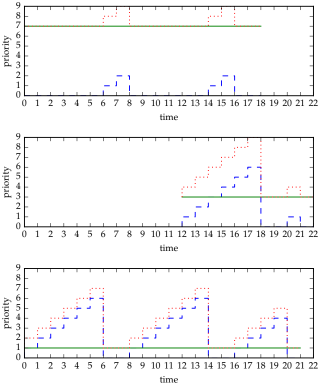
Example (priority-based + ageing, timer pre-emption: $\tau = 2$, context switch cost: 0)

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





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 [11, Section D1.2]

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.

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.

References

[1] *Wikipedia: Process*. URL: http://en.wikipedia.org/wiki/Process_(computing) (see p. 45).

[2] *Wikipedia: Scheduling*. URL: http://en.wikipedia.org/wiki/Scheduling_(computing) (see p. 45).

[3] J.R. Levine. *Linkers & Loaders*. Morgan-Kaufmann, 2000. URL: http://www.iecc.com/linker (see pp. 10, 11).

[4] R. Love. "Chapter 5: Process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 45).

[5] R. Love. "Chapter 6: Advanced process management". In: *Linux System Programming*. 2nd ed. O'Reilly, 2013 (see p. 45).

[6] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 3: Process concept". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 45).

[7] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 5: Process scheduling". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 45).

[8] A.S. Tanenbaum and H. Bos. "Chapter 2.1: Processes". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 45).

[9] A.S. Tanenbaum and H. Bos. "Chapter 2.4: Sheduling". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 45).

[10] A.S. Tanenbaum and H. Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015.

[11] *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, 10, 11, 13, 15, 16, 43).