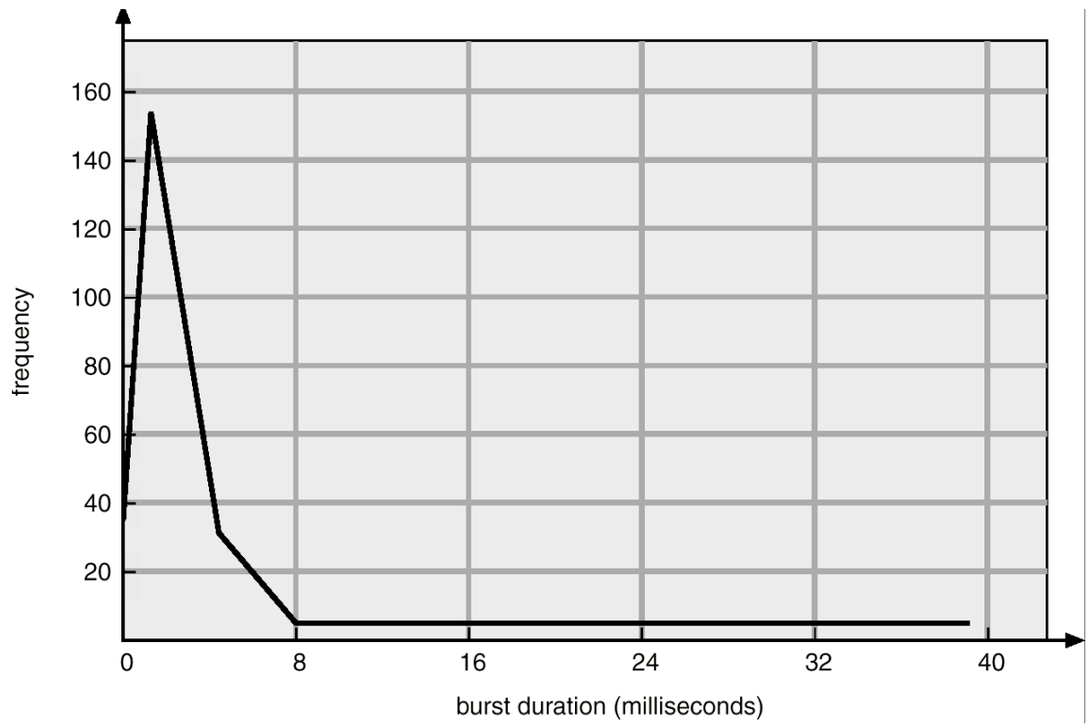
Scheduling processes/ threads

Different systems of scheduling for different purposes

- Batch systems
  - Keep the machine going
- Time-sharing systems
  - Keep the users going
- Real-time systems (including multimedia, virtual reality etc)
  - Always deal with the important things first



Graph showing frequency of CPU-burst times of threads/processes.

Levels of scheduling

Batch systems

- Very long-term scheduler
  - before work can be submitted
  - can this user afford it?
  - administrative decisions - students can't enter jobs between 10pm and 6am
- Long-term scheduler
  - may enforce administrative decisions
  - which jobs (currently spooled) should be accepted into the system
  - need to know about resource requirements
  - How many CPU seconds?
  - How many files, tapes, pages of output?
  - (need a way of encouraging users to try to be accurate in their estimation)
  - it is common for jobs with small resource requirements to run sooner - why?
  - invoked when jobs leave the system
- Medium-term scheduler
  - if things get out of balance suspend this process and swap it out
- Short-term scheduler (sometimes called the dispatcher)
  - which of the runnable jobs should go next
- Dispatcher
  - The code which performs the context switch from one process to another.

What about desktop or phone?

Desktop OS

- Most of the scheduling levels on the previous slide do not exist. Why?
- Scheduler designed to optimise responsiveness for the user, this means input/output bound threads will be scheduled quickly when input becomes available.
  - And CPU bound threads will be penalised (only if IO threads are ready to run)

Phone OS

- Usually one process runs in the foreground
- We don’t want too much work in the background - to reduce energy requirements.

Scheduling algorithms

FCFS - first come first served

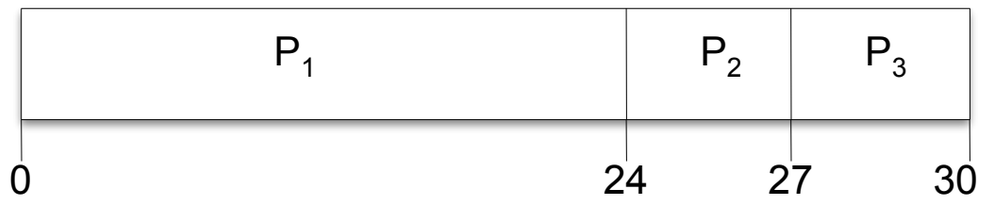
- no time wasted to determine which process should run next
- little overhead as context switch only when required

Example:

Process	Burst Time
P_1	24
P_2	3
P_3	3

Suppose that the processes arrive in the order: P_1 , P_2 , P_3

The **Gantt** Chart for the schedule is:

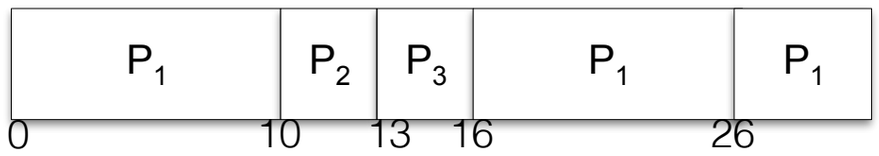


Average waiting time: (0 + 24 + 27)/3 = 17

Round-robin

- Round-robin scheduling
  - A pre-emptive version of FCFS.
- Need to determine the size of the time slice or time quantum.
- What is wrong with treating every process equally?
  - no concept of priorities
  - doesn't deal with different types of process - compute bound vs IO bound
- One way to tune this is to change the length of the time slice.
  - What effect does a long time slice have?
  - What effect does a short time slice have?
- What is the average waiting time here?

time slice = 10, what if the time slice = 5?



Minimising average wait time

If we could choose the process which was going to use the CPU for the smallest amount of time we would have an algorithm which minimised the average wait.

- For the example on page 3 the average wait time would be 3.

SJF – shortest-job first

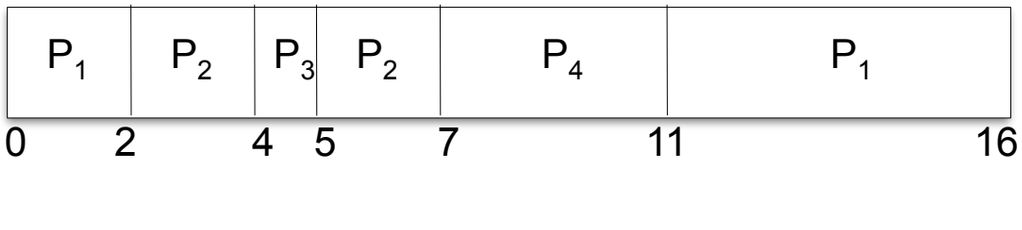
- Unfortunately we don't know which is the process with the shortest CPU burst.
- Use the previous CPU bursts to estimate the next.

We may use a different method of pre-emption

- If a process becomes ready with a shorter burst time than the remaining burst time of the running process then the process is pre-empted.
- This is simply a priority mechanism.

Pre-emptive SJF

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4



- Average waiting time = (9 + 1 + 0 +2)/4 = 3
- What is the average waiting time without preemption?

Handling priorities

Explicit priorities

- Unchanging
- Set before a process runs.
- When a new process arrives it is placed in the position in the ready queue corresponding to its priority.
- It is possible to get ***starvation***.

Variable priorities

- Priorities can vary over the life of the process.
- The system makes changes according to the process' behaviour: CPU usage, IO usage, memory requirements.
- If a process is not running because it has a low priority we can increase the priority over time - this is one way of ageing the priority
  - Or a process of a worse priority might be scheduled after five processes of a better priority.
  - This prevents starvation, but better priority processes will still run more often.
- Can pre-empt processes if a better choice arrives.

Multiple queues

- Either a process stays on its original queue
- or processes move from queue to queue.
- Some are absolute - worse priority queues only run a process if no better queues have any waiting.
- Some have different selection strategies.
  - Lower priority queues might occasionally be selected from.
- Some allocate different time slices.
- Processes can be moved from queue to queue because of their behaviour.
  - CPU intensive processes are commonly put on worse priority queues.
  - What behaviour does this encourage?
- Processes which haven't run for a long time can be moved to better priority queues.

Moving between queues

Level	time-slice	frequency of selection
1	10	1
2	100	0.1
3	200	only if none at levels 1 & 2

Multiple processors

We presume all processes can run on all processors (not always true)

Maintain a shared queue.

Is this preferable?

Let each processor select the next process from the queue.

Or let one processor determine which process goes to which processor.

UNIX process scheduling

- Every process has a *scheduling priority* associated with it; larger numbers indicate worse priority.
- Priorities can be changed by the *nice* system call.
  - Ordinary users can only *nice* their own processes upwards (i.e. worse priorities).
- Processes get worse (higher) priorities by spending time running.
  - There is a worst level which all CPU bound processes end up at.
  - This means round-robin scheduling for these processes.
- Process ageing is employed to prevent starvation.
- Priorities are recomputed every second.

Old Linux process scheduling

- Linux uses two process-scheduling algorithms:
  - A time-sharing algorithm for most processes.
  - A real-time algorithm for processes where absolute priorities are more important than fairness.
- A process's scheduling class defined which algorithm to apply.
- For time-sharing processes, Linux used a prioritized, credit based algorithm.
  - The process with the most credits won.
  - Every clock tick the running process lost a credit.
  - When it reached 0 another process was chosen.

$$credits := \frac{credits}{2} + priority$$

- The crediting rule was run when no runnable process had any credits left.
- This meant that waiting processes got extra credits and would run quickly when no longer waiting.

Linux real-time scheduling

- Linux implements the FIFO and round-robin real-time scheduling classes (POSIX.1b); in both cases, each process has a priority in addition to its scheduling class.
  - The scheduler runs the process with the highest priority; for equal-priority processes, it runs the longest-waiting one.
  - FIFO processes continue to run until they either exit or block.
  - A round-robin process will be preempted after a while and moved to the end of the scheduling queue, so that round-robin processes of equal priority automatically time-share between themselves.

Current Linux process scheduling

- Linux 2.6.23 upwards - Completely Fair Scheduler https://www.linuxjournal.com/node/10267
- Goal: keep thread selection O(1) regardless of the number of threads
- Interactive processes get priority
- Uses virtual run time (vruntime) which depends on
  - a fair CPU share per thread. e.g. If 4 threads each gets 1/4 of the CPU time
  - the wait time for the thread
  - the priority (including nice value) of the thread, better priorities get smaller runtime values
  - and how long a thread has been running
- Uses a red-black tree of tasks (according to vruntime) - task with smallest vruntime is selected to run
- Over time the vruntime of the running thread will be greater than the left most node in the tree and it will be preempted.
- Also complex load balancing is done (multi-core, NUMA)

The scheduler has been improved over the years

- To improve interactive response times
- and for better multicore performance

Before next time

Read from the textbook

6.6 Real-Time Scheduling