

Notes on the assignment

Remember to call the queue in which the sent messages are stored “@received”.

Otherwise the `give_lots` method won't work.

Alternatively you may alter the `give_lots` method.

`Thread.abort_on_exception=true` will help you to see the errors in your threads.

`Thread.critical` needs to be set to true and then reset to false around any accesses to data that could be performed “simultaneously” by more than one thread.

Put

```
@received ||= []
```

in every method that refers to your @received queue/array. This assigns it the value of a new Array if and only if the variable is currently empty.

Notes on the assignment

Thread objects and actual threads.

Like Java, Ruby has Thread objects which are used to refer to and control the actual threads. A Thread object is just like any other object in the sense that any thing (including code running in another thread) can call its methods.

In the

`single_producer_single_consumer.rb`
program, the producer thread calls:

```
@consumer.give("data", value)
```

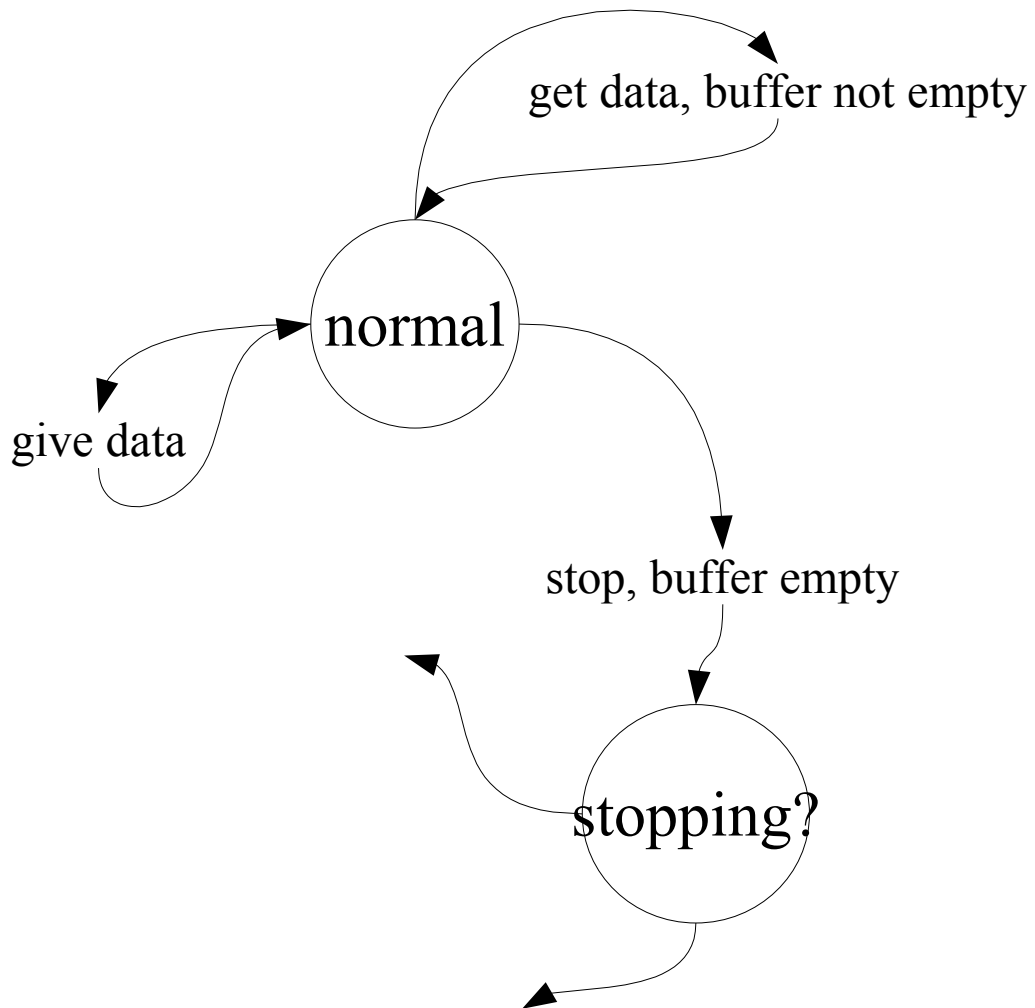
So the producer thread is calling code inside the consumer Thread object.

At the same time the consumer thread can be calling code inside the consumer Thread object, inside the `receive_list` method.

Notes on the assignment

Multiple producers and multiple consumers.

It may help if you draw a state diagram for the buffer. What the buffer does depends on the state it is currently in.



Think of the priority section of the assignment as the bonus section.

Scheduling processes/threads

Different systems for different purposes

Batch systems

Keep the machine going

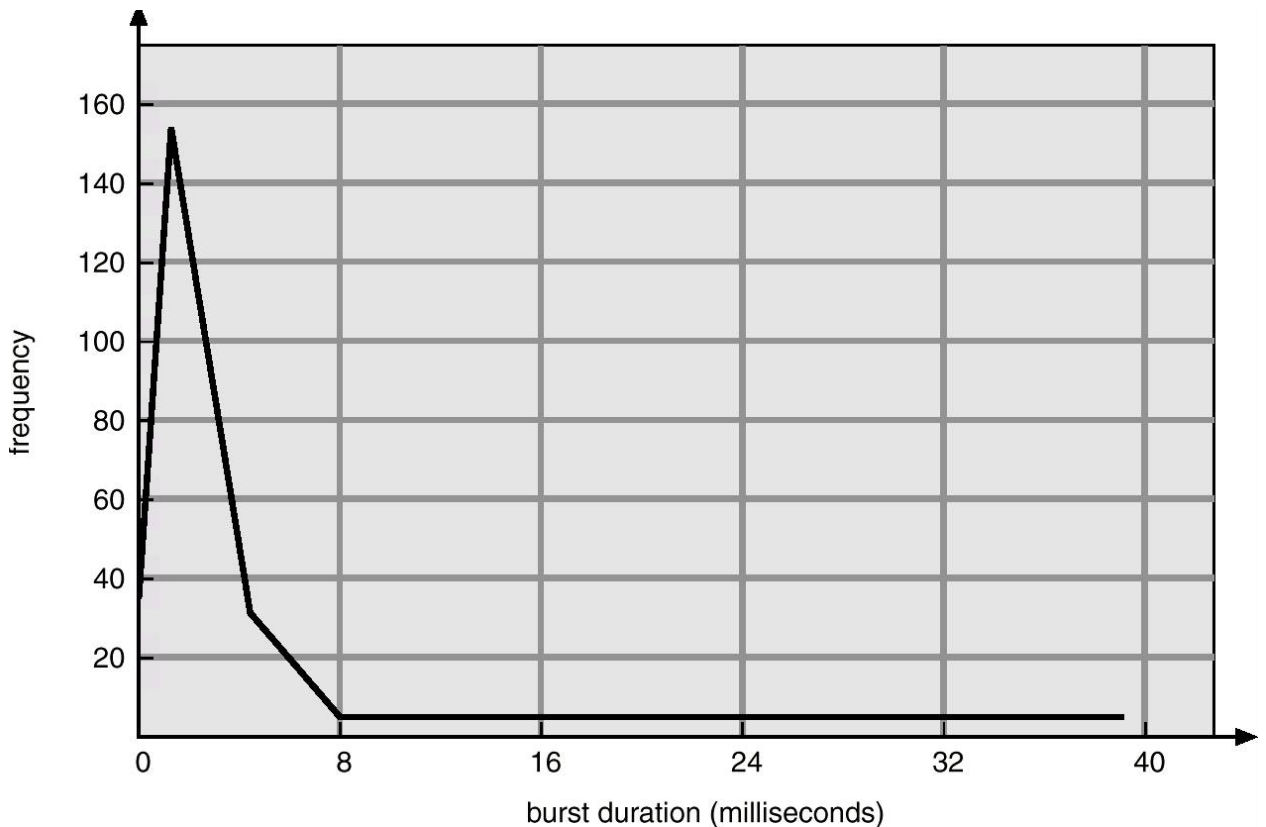
Time-sharing systems

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.



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 thing which performs the context switch from one process to another.

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:

<u>Process</u>	<u>Burst Time</u>
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 preemptive version of FCFS.

Need to determine the size of the time slice or time quantum.

What is wrong with treating every process equally?

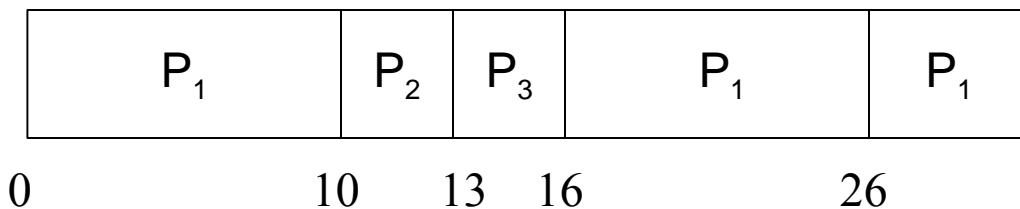
- no concept of priorities
- doesn't deal with different types of process - compute bound vs IO bound

One way to tune this is to change 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



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 at the beginning 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 preemption.

If a process becomes ready with a shorter burst time than the remaining burst time of the running process then the process is preempted.

This is simply a priority mechanism.

Preemptive SJF

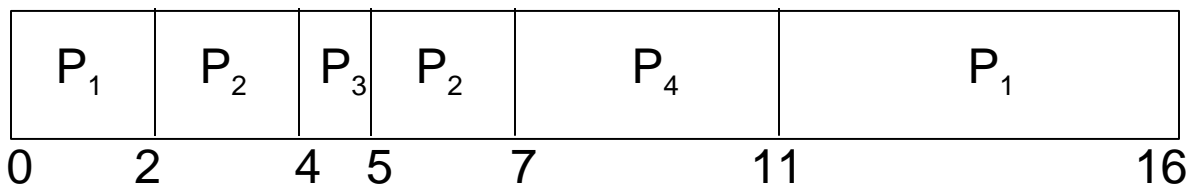
<u>Time</u>	<u>Process</u>	<u>Arrival Time</u>	<u>Burst</u>
-------------	----------------	---------------------	--------------

	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?

Explicit priorities

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.

Priorities can vary over the life of the process.

The system makes changes according to the process' behaviour CPU usage, IO usage, memory requirements.

Aging priorities sometimes are used to allocate percentages of process time.

e.g. 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 preempt processes if a better choice arrives.

Multiple queues

Either process stays on original queue

batch system, user may decide explicitly how it should be handled

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.

Why 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 aging is employed to prevent starvation.

Priorities are recomputed every second.

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 defines which algorithm to apply.

For time-sharing processes, Linux uses a prioritized, credit based algorithm.

- The process with the most credits wins.

- Every clock tick the running process loses a credit.

- When it reaches 0 another process is chosen.

- The crediting rule is run when no runnable process has any credits left.

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

- This means that waiting processes get extra credits and will run quickly when they need to.

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.

Before next time

Read from the textbook

Chapter 6 Scheduling

6.1 Basic Concepts

6.2 Scheduling Criteria

6.3 Scheduling Algorithms

6.4 Multiple-Processor Scheduling

6.6 Thread Scheduling

6.7.1 Solaris scheduling

6.7.2 Windows XP scheduling

6.7.3 Linux scheduling

6.8 Java Thread Scheduling

6.9 Algorithm Evaluation