02246 Assignment
Background and Getting Started[*]

1 Background

Schedulers are a vital part of any computing system in which multiple users, or processes, try to make use of a single resource. The idea is to *share* the resource between the different users, so that each user experiences a certain quality of service.

Any operating system that supports multi-tasking makes use of a scheduler as a central component — it has to time-share the use of the CPU between different processes, and the operating system itself. By interleaving the execution of the processes (executing one for a short period of time, before moving onto another) it appears to the user that they are running simultaneously. In reality, however, only one process is executing at any given time (assuming that the CPU has only one core).

Figure 1 shows a CPU scheduler, which takes a number of *jobs* as input (these are processes that we want to execute), and decides the order in which to execute them. Each job requires a certain amount of time to complete (which may or may not be known when the job starts), and we can split jobs up into fixed length time units, or *quanta*. The task of the scheduler is to determine the *order* in which these quanta are executed.

The use of schedulers is important not only for operating systems in desktop computers, but in embedded devices such as the iPhone, in network routers (to determine the order in which packets are output on an interface), in disk drives (to determine the order to perform read and write operations), and in printers (to schedule print jobs). Even in everyday situations, there are schedulers in operation — for example, when you go into a shop and take a ticket, hence joining a queue of people waiting to be served.

There are a number of important properties that we would like a scheduler to exhibit:

- *Completion* — can we be sure that every task will be able to finish executing? If not, we have a situation known as *starvation*.
- *Fairness* — does every task have an equal chance to run? Or are some tasks favoured over others (e.g. long tasks, or short tasks)? Perhaps we want to give *priority* to certain tasks that are more important, but we would still want to ensure completion of *all* tasks.
- *Response time* — how long does it take for a task to complete?

[*]Thanks to the original author of this document Michael Smith, and further contributors (Lijun Zhang, Kebin Zeng, Flemming Nielson, Alberto Lluch Lafuente, Andrea Vandin).

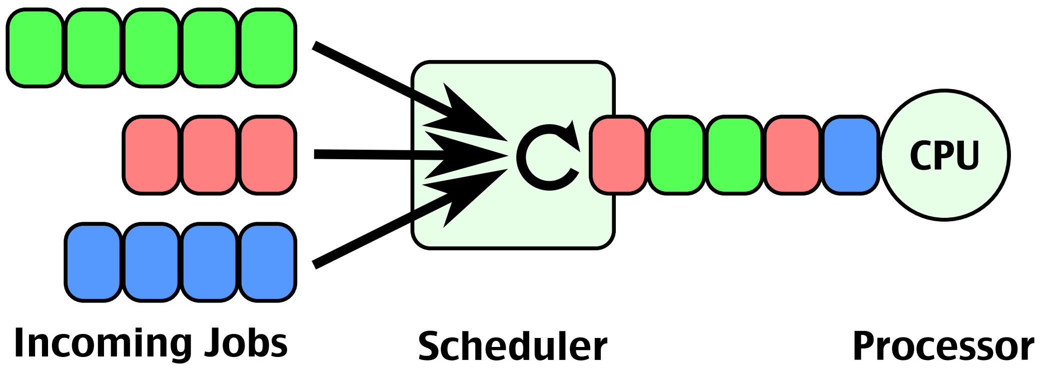

Figure 1: A CPU scheduler

- *Throughput* — what is the rate at which tasks complete? Note that it's possible to have a high throughput, but a long response time; for example, if there is a large queue of tasks waiting to execute.

- *Utilisation* — what proportion of the time are we actively making use of the resource? For example, a complicated scheduling algorithm may have a high overhead, meaning that we spend too much time using the processor to compute the schedule, rather than to actually execute tasks.

In the assignments for this course, you will look at developing several models of schedulers, of increasing complexity, using the PRISM model checker. The aim of this tutorial is to introduce you to PRISM, and some simple models of schedulers, which you will later build upon in the assignments. It's important to spend enough time to completely understand everything in this tutorial — if you encounter any difficulties, then please ask for help!

2 PRISM: An Introduction

2.1 Downloading and Installing PRISM

Before we start to do any modelling:

- PRISM (`http://www.prismmodelchecker.org/`) up and running on your laptop.

- Run the PRISM tutorial Part 1 (`http://www.prismmodelchecker.org/tutorial`) until and including section "Model checking with PRISM" to get a feeling for what PRISM can do. You are not expected to understand all the concepts in the tutorial; we will cover them during the course.

- Get familiar with the manual. In particular with the section on the PRISM language (`http://www.prismmodelchecker.org/manual/ThePRISMLanguage/Introduction`) and on Property Specification (`http://www.prismmodelchecker.org/manual/PropertySpecification` `Introduction`). You will need to consult it very often.

3 A Simple PRISM Model of a Scheduler

In the practical parts of the assignments for this course, we're going to be looking at *schedulers*. In particular, we'll take a simple scenario where we have some *clients*, who generate *jobs* of varying length, which are processed in some order determined by a scheduler. We'll assume that clients only generate one job at a time, and that there is only one processor in the system — i.e. only one job can be being executed at any one time. To take a look at the model described in this section, load `FCFS.nm` in PRISM.

We'll begin by describing the module that corresponds to a client, which produces jobs that have a length ranging between 1 and 5 time units. We will, for now, choose the length of a job non-deterministically, by allowing several commands to be enabled at the same time. The client produces a new job with the *create* action, executes the job (reducing its length by one) with the *serve* action, and completes the job with the *finish* action:

```
module Client

    // State of the job (0=inactive, 1=active)
    state : [0..1] init 0 ;
    // Length of the job
    task : [0..5] init 0 ;

    // Create a new job — length chosen non-deterministically
    [create]  state = 0  →  (state' = 1)  ∧  (task' = 1) ;
    [create]  state = 0  →  (state' = 1)  ∧  (task' = 2) ;
    [create]  state = 0  →  (state' = 1)  ∧  (task' = 3) ;
    [create]  state = 0  →  (state' = 1)  ∧  (task' = 4) ;
    [create]  state = 0  →  (state' = 1)  ∧  (task' = 5) ;

    // Execute the job
    [serve]  state = 1  ∧  task > 0  →  (task' = task − 1) ;

    // Complete the job
    [finish]  state = 1  ∧  task = 0  →  (state' = 0) ;

endmodule
```

We are explicitly modelling the completion of a task as a separate step, as in many scenarios this takes some time to occur. For example, in an operating system, where we might need to raise an interrupt to inform a client that its job has completed, or in a web service scenario, where we need to send a message back to the client over the network. This is a modelling decision, however, and we could equally have made a different choice.

In PRISM, we can easily create multiple clients with the same behaviour as the above, by renaming the variables and actions. We'll create two clients to use in our model:

```
module $Client_1 = Client$ [  $state = state_1$
                              $task = task_1$
                              $create = create_1$
                              $serve = finish_1$
                              $finish = finish_1$   ]
endmodule

module $Client_2 = Client$ [  $state = state_2$
                              $task = task_2$
                              $create = create_2$
                              $serve = finish_2$
                              $finish = finish_2$   ]
endmodule
```

We will analyse a PRISM model with the following setup for the system, which consists of the two *Client* modules in parallel with a *Scheduler* module — we'll give two alternative definitions for this in the next two sections:

```
system
   $Scheduler \parallel Client_1 \parallel Client_2$
endsystem
```

Recall that this form of synchronisation in PRISM means that all actions that occur syntactically in more than one of the modules *must* be synchronised over in the system. We now need to specify the *Scheduler* module, and we'll consider two different scheduling strategies for it — First Come, First Served (FCFS), and Shortest Remaining Time (SRT). The files corresponding to the two PRISM models are `FCFS.nm` and `SRT.nm` respectively.

3.1 A First Come, First Served (FCFS) Scheduler

An FCFS scheduler has the property that it always chooses to execute the job that arrived first. We'll model this by a scheduler that can hold up to two jobs in a queue. New jobs enter at the end of the queue, and move forward when there is a vacant slot. Only the job at the front of the queue can be executed.

```
module Scheduler

    // First job in the queue (0=empty, 1=Client1, 2=Client2)
    $job_1$ : [0..2] init 0 ;
    // Second job in the queue
    $job_2$ : [0..2] init 0 ;

    // Place a new job at the end of the queue (if it is empty)
    [$create_1$]  $job_2 = 0 \rightarrow (job_2' = 1)$ ;
    [$create_2$]  $job_2 = 0 \rightarrow (job_2' = 2)$ ;

    // Shift the queue if there is an empty slot
    [ ]  $job_1 = 0 \wedge job_2 > 0 \rightarrow (job_1' = job_2) \wedge (job_2' = 0)$ ;

    // Execute the job at the head of the queue
    [$serve_1$]  $job_1 = 1 \rightarrow$ true ;
    [$serve_2$]  $job_1 = 2 \rightarrow$ true ;

    // Complete the job at the head of the queue
    [$finish_1$]  $job_1 = 1 \rightarrow (job_1' = 0)$ ;
    [$finish_2$]  $job_1 = 2 \rightarrow (job_1' = 0)$ ;

endmodule
```

The resulting PRISM model has 83 states.

An important question we'd like to ask is the following. If a job is created by a client (i.e. $task > 0$), then can we be sure that it will eventually complete (i.e. that at some point in the future, $task = 0$)? We can express this property in CTL as follows, for the first client, $Client_1$:

$$AG((task_1 > 0) \Rightarrow AF(task_1 = 0))$$

To check this property in PRISM, load the file `FCFS.pctl`, and verify the first property in the list — you should find that it's satisfied in the model. However, the formula encoded in the file `FCFS.pctl` actually is $(task_1 > 0) \Rightarrow AF(task_1 = 0)$.

- Is this adequate given the version of PRISM that you are running?
- If not, how would you modify `FCFS.pctl`?

There is a similar story for $Client_2$, where we replace $task_1$ with $task_2$ in the above — this corresponds to the second property in `FCFS.pctl`.

3.2 A Shortest Remaining Time (SRT) Scheduler

An SRT scheduler always chooses the job with the least time remaining to execute. Previously, we modelled the scheduler using a queue for the jobs, but this time we can simply record whether or not there is a waiting job from each of the clients. In the case when both clients have a waiting job, we choose to execute the one with the least time remaining (i.e. the *task* variable has the smallest value). The module, which can be found in `SRT.nm`, is defined as follows:

```
module $Scheduler$

    // Is there a job waiting for Client 1?
    $job_1$ : bool init false ;
    // Is there a job waiting for Client 2?
    $job_2$ : bool init false ;

    // Record that there is a job waiting
    [$create_1$]  $job_1 =$ false $\rightarrow$ ($job_1' =$ true) ;
    [$create_2$]  $job_2 =$ false $\rightarrow$ ($job_2' =$ true) ;

    // Execute the job that has the least remaining time
    [$serve_1$]  $job_1 =$ true $\wedge$ $job_2 =$ false $\rightarrow$ true ;
    [$serve_2$]  $job_1 =$ false $\wedge$ $job_2 =$ true $\rightarrow$ true ;
    [$serve_1$]  $job_1 =$ true $\wedge$ $job_2 =$ true $\wedge$ $task_1 \leq task_2$ $\rightarrow$ true ;
    [$serve_2$]  $job_1 =$ true $\wedge$ $job_2 =$ true $\wedge$ $task_2 \leq task_1$ $\rightarrow$ true ;

    // Complete any job that has finished
    [$finish_1$]  $job_1 =$ true $\rightarrow$ ($job_1' =$ false) ;
    [$finish_2$]  $job_2 =$ true $\rightarrow$ ($job_2' =$ false) ;

endmodule
```

This time, the PRISM model has only 48 states. Intuitively, this is because we are not recording the *order* in which jobs arrive, but only the *presence* of a job from each of the clients.

We can now try to verify the same property as for the FCFS scheduler:

$$AG((task_1 > 0) \Rightarrow AF(task_1 = 0))$$

This time, the property should *not* be satisfied in the model.

This is because it's possible, for example, for $Client_1$ to create a job of length 5, and for $Client_2$ to create a constant stream of short jobs, of length 1. Because of the scheduling policy, we'll always choose to execute the jobs from $Client_2$, hence there is no guarantee that the longer job from $Client_1$ will ever get to execute. This illustrates that the SRT scheduling policy suffers from *starvation*.

Now it is time to try this in PRISM. One way is to load the properties from `SRT.pctl`.

- Do they give the correct answer?
- If not, how would modify the properties in `SRT.pctl`?