Practical Assignment #2: Scheduling

1. Preliminaries

This assignment has two objectives: (i) to develop your understanding of CPU process scheduling through the construction of a simulation of the Round Robin scheduling strategy, (ii) develop your understanding of the Round Robin scheduling strategy through an investigation of performance under different time slice lengths and different dispatcher overheads.

Please take some time to read through this script completely and plan what you are going to do before you begin coding. You may wish to begin by reading the chapter in the course text on scheduling.

In the rest of this section we give the requirement specification for the Round Robin simulation program. In section 2 we describe the design. In section 3 we briefly describe the component framework you should use to implement it.

1.1 Discrete Event Simulation

Computer simulation is used to model the behaviour of a complex system. The intention is that the model has predictive properties i.e. it tells us something about the real world system represented.

Depending on the context, there are a variety of reasons for using a model. Primarily, it should be easier (and cheaper) to construct than its real world counterpart.

A model is an abstraction. It represents certain aspects of a system and not others. In the case at hand we wish to model the behaviour of an operating system kernel from the perspective of process scheduling. We're not interested in memory allocation, file handing, and device drivers and so on.

A 'discrete event' simulation is based on an abstract view of system behaviour as a sequence of events, each occurring at a particular time. An event denotes a change in the state of the system. Between events the system state does not change.

Change in the model state occurs at *discrete* times rather than continuously (as in the real world). The benefit of this abstraction is that changes in system state can take place in simulated time. Once an event is processed, the time can be advanced to when the next event occurs. The simulation is simpler and faster (in real time) than the real thing.

The key concepts of a discrete event simulation are: model, event, event queue, simulation clock, and simulated time.

- An event describes a change in model state.
- The event queue, as the name suggests, is a queue of pending events. The queue is ordered by when the events are due to occur.
- Prior to running the simulation, some initial event or events are inserted into the queue.
- The simulation clock is simply a counter used to track the advancement of simulated time.
- Simulation proceeds by repeatedly removing an event from the queue and processing it.
- Processing an event involves changing the state of the model, and causes the clock to advance.
- As a consequence of processing an event, further events may be generated and added to the queue.
- The simulation stops when the queue is empty.

1.2 The Simulator

A Java program is required that will simulate the execution of a set of programs on a computer under a Round Robin scheduling strategy. The class containing the main method should be called '`Simulator`'.

The program will be invoked with a number of command line arguments:

```
java Simulator <config file> <slice length> <dispatch overhead>
```

The first argument is the name of a configuration file that describes the I/O devices attached to the simulated system and gives the names of program files that are to be run.

The second and third arguments are the parameters for the Round Robin simulation. The slice length is the length of time a process is permitted to operate before pre-emption. The dispatch overhead is length of time that it takes the kernel to switch processes.

The program will simulate execution of the programs named in the configuration file. When simulation is complete, it will report the following:

- System time: the time at which execution completes i.e. the duration of the simulation.
- Kernel time: the amount of system time spent executing in kernel space.
- User time: the amount of system time spent executing in user space.
- Idle time: the amount of system time that the CPU was idle (waiting for completion of an I/O event).
- Context switches: the number of switches from the execution of one process to another.
- CPU utilization: the percentage of system time that the CPU spent executing in user space.

1.2.1 Configuration file

The format for the configuration file is as illustrated by the following example:

```
# I/O Devices attached to the system.
# I/O <device id> <type>
I/O 1 DISK
I/O 2 CDROM
# Programs
# PROGRAM <arrival time> <program file name>
PROGRAM 8 firefox.dat
PROGRAM 5 primescalculator.dat
PROGRAM 11 spice.dat
```

- A line beginning with '`#`' is a comment and is ignored by the simulator.
- A line beginning with '`I/O`' describes a device attached to the system. A device has a unique ID number and a non-unique type.
- A line beginning with 'PROGRAM' describes a program that is to be run during the simulation. A program has an arrival time and a file name. The arrival time describes the point in the simulation at which the program should be loaded. It is expressed as a number of 'virtual' time units.

1.2.2 Program file

The program files used by the simulator do not contain real programs. They contain abstract descriptions of programs, as illustrated by the following example:

```
# firefox.dat
```

```
# A program with relatively high I/O demands compared to CPU time.
CPU 2
IO 2
CPU 3
IO 5
CPU 1
IO 4
CPU 2
IO 2
CPU 3
IO 5
CPU 1
IO 4
CPU 2
IO 2
CPU 3
```

- A line beginning with '#' is a comment and is ignored by the simulator.
- A line beginning with 'CPU' is a 'process instruction'. It is an abstract description of a block of program code that is executed on the system CPU. To put it another way, it describes a CPU burst – a period of time during which the program only uses the CPU.
- A line beginning with 'IO' is an 'I/O instruction'. It is an abstract description of a block of program code representing I/O activity. To put it another way, it describes an IO burst – a period of time during which the program waits for I/O.
- The value following an 'IO' or 'CPU' keyword is the burst duration in system time units.

A program never contains two consecutive IO instructions or two consecutive CPU instructions.

The first and last instructions are always CPU instructions.

1.3 The Investigation

You should use your program to investigate the relative performance of the Round Robin scheduling strategy for (i) varying slice time lengths and (ii) varying context switching overheads.

When investigating one variable, keep the other fixed.

As the basis of your investigation you should use a set of program files (1.2.2) that represent a mix of behaviours: CPU bound, I/O bound, balanced.

Record your results and use them to construct two graphs: (i) slice time length (x- axis) against completion time (y-axis); (ii) context switching overhead (x- axis) against completion time (y-axis).

2. Simulator Design

In this section we describe the design. It is partially codified as a set of Java interface declarations that you should use. These are identified in the next section.

2.1 Conceptual Overview

The design comprises 4 major components: a kernel, a CPU, a system timer, and an event simulation driver.

The simulation driver sets up the simulation and manages the main simulation loop: the event queue and the processing of events.

The kernel simulates aspects of kernel behaviour relating to scheduling:

- Processes are represented by Process Control Blocks (PCBs).
- A 'Ready queue' holds processes waiting to be executed.
- One or more device queues hold processes waiting for I/O to complete.
- As events unfold, processes are moved between queues and onto and off the CPU.

The CPU simulates the processing of program instructions.

The system timer records the current system time as well as time spent in user space, time spent in kernel space, and time spent idle.

The CPU advances the timer to represent time spent processing a CPU instruction. The kernel advances the timer to represent the time spent in various operations. The kernel also uses the system timer to schedule time slice pre-emption time-outs.

Making connections with assignment one, the kernel component is shaped around a system call interface that provides facilities for the loading of programs, the creation of devices, the performing of I/O, and the termination of processes.

2.2 Event Simulation

The overall event simulation algorithm is as follows.

2.2.1 Set up

The driver is responsible for simulation set up.

1. For each program identified in the configuration file, it will create a 'load program' event and insert it in the event queue.
2. For each device identified in the configuration file, the driver will make a `MAKE_DEVICE` system call to the kernel.
3. System time is set to 0.

2.2.2 Main event loop

Once set up of the simulation is complete, the driver's main simulation loop begins.

Until the event queue is empty and the CPU is idle:

1. While the event queue contains an event, and the event is due to occur at or before the current system time, *st*, process the event.

2. Call the CPU to simulate execution of the currently scheduled process (if any) up to the time *tf* at which the next event is due to occur.

Note that the processing of events may cause further events to be placed on the event queue. Similarly, simulated execution by the CPU.

2.2.3 Event Processing

There are three types of event that need to be explicitly represented in the simulation driver:

Name	***Attributes***	***Description***
'load program'	Time at which the event is due to occur. Name of program to be loaded.	Used to indicate that an EXECVE system call must be made to the kernel to load a program.
'wake up'	Time at which the event is due to occur. ID of device handling the request. ID of the process that made the request.	Used to indicate that the kernel must be presented with a 'wake up' interrupt representing the completion of an I/O request.
'timeout'	Time at which the event is due to occur. The ID of the process to be pre-empted.	Use to indicate that the kernel must be presented with a 'timeout' interrupt from the system timer.

2.3 Kernel

The kernel simulates process scheduling. It provides a system call interface for the loading of programs, the creation of devices, the performing of I/O, and the termination of processes; and it provides an interrupt handler interface that enables it to receive interrupts from the system timer and from I/O devices.

2.3.1 System Calls

Four system calls are supported:

System Call Number	***Name***	***Arguments***	***Description***
1	MAKE_DEVICE	Device ID Device type	*Create a device object and I/O queue.*
2	EXECVE	Program file name	*Load the named program.*
3	IO_REQUEST	Device ID Request duration.	*Simulate a IO request on the given device for the given duration.*
4	TERMINATE_PROCESS		*Terminate execution of the current process.*

When the kernel receives a MAKE_DEVICE call, it creates an object to represent the device and a queue to holding processes when waiting for I/O requests on the device to complete.

When the kernel receives an EXECVE call, it creates a Process Control Block (PCB) for the program and then loads the list of instructions contained in the program file and attaches them to it. The PCB is placed at the end of the scheduling 'Ready Queue'.

A process control block contains a program counter (PC) that is used to keep track of the instruction currently being executed. Initially, PC is zero.

When the kernel receives an IO_REQUEST call:

1. The currently executing process is removed from the CPU and placed on the relevant device queue.
2. The timeout scheduled to mark the end of the current time slice is cancelled.
3. A 'wakeup' interrupt is scheduled for the point at which the request is due to complete.
4. The next available process is switched from the ready queue onto the CPU.
5. The system timer is called to set up a timeout to interrupt execution at the end of the new process time slice.

When the kernel receives a TERMINATE_PROCESS call:

1. The currently executing process is removed from the CPU and discarded.
2. The timeout scheduled to mark the end of the current time slice is cancelled.
3. The next available process is switched from the ready queue onto the CPU.
4. The system timer is called to set up a timeout to interrupt execution at the end of the new process time slice.

2.3.2 Interrupts

There are two types of interrupt that the kernel is required to handle.

Interrupt Number	***Name***	***Arguments***	***Description***
0	TIME_OUT	The ID of the process to be pre-empted.	Used to signal the end of a scheduled time slice.
1	WAKE_UP	The ID of the device issuing the interrupt. The ID of the process waiting on the device.	Used to signal completion of the end of an I/O request.

When the kernel receives a TIME_OUT interrupt marking the end of the current time slice:

1. The currently executing process is removed from the CPU and placed at the back of the ready queue.
2. The next available process is switched from the ready queue onto the CPU.
3. The system timer is called to set up a new timeout to interrupt execution at the end of the new process time slice.

When the kernel receives a WAKE_UP interrupt:

1. The relevant device queue is located.
2. The relevant process is removed.
3. Its PC is incremented to step over the `IO` instruction that has just been completed.
4. It is placed at the back of the ready queue.

2.4 The CPU

The CPU component serves to hold the currently executing process and to simulate its execution. It is possible that there is no currently executing process in which case the CPU is *idle*.

Requests to simulate execution come from the driver. Execution is always bounded: the CPU is requested to simulate execution up to a system time point *tp*.

Note that a process can only be on the CPU if the current program instruction (as identified by the program counter) is a "`CPU`" instruction.

- If the current instruction can complete in the given time, then the CPU will move to the next instruction in the 'program', which (by definition) must be an I/O instruction, if it exists.
    - The CPU processes an I/O instruction by making an IO_REQUEST system call to the kernel.
    - If there is no next instruction then the CPU makes a TERMINATE_PROCESS system call to the kernel. In either case, the effect will be to switch the current process out.
- If the current instruction cannot be completed in the given time then a record of the amount remaining is made.

In either case, the CPU will update the system timer to indicate the amount of time spent in user space processing.

2.5 System Time

The system timer is essentially a set of counters that track the passage of simulated time.

2.5.1 The System Timer

The CPU updates user time in the course of simulating execution. The kernel must update kernel time to simulate the time spent in its activities:

- At the end of a system call the kernel should update the kernel time by a fixed amount representing the cost of the call.
- If a call involved context switching then this dispatching overhead must be added to the cost. *Observing the effects of variation in the dispatching overhead is part of the investigation.*

The kernel also uses the system timer to set up TIME_OUT interrupts. The system timer should handle these requests by creating timeout events and placing them in the simulation event queue.

2.5.2 I/O Devices

When the kernel processes an IO_REQUEST, it places the currently executing process on the device queue. A WAKE_UP interrupt is scheduled for the point at which the request is due to complete.

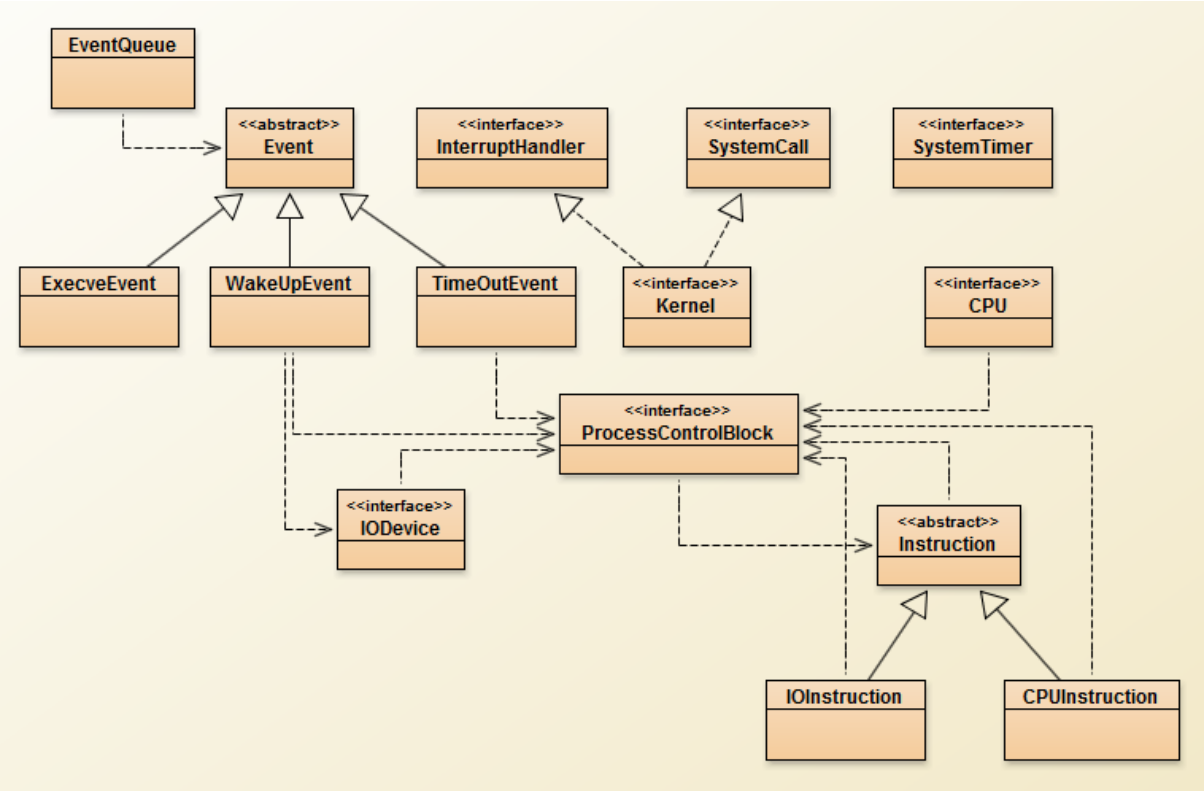
The point at which a request is due to complete depends on the required duration, *d*, and on prior I/O requests:

- We assume that the time, *tf*, at which the device becomes free is known.
- If the time *tf* is less than or equal to the current system time, *st*, then the device is free now. The new request will complete at *st*+*d*.
- If the time *tf* is greater than the current system time, *st*, then the request will complete at *tf*+*d*.

As with TIME_OUT interrupts, a request for a WAKE_UP interrupt is handled by creating a wakeup event and placing it in the simulation queue.

3. Java Components

We have developed a framework of Java components based on the described design. You should use these components to develop your simulator.



- Your kernel code should implement the given interface, similarly your CPU, system timer, IO device, and process control block entities.
- The code for events and the event queue is complete. Use this to develop your driver component.
- Your kernel should use the defined CPU, ProcessControlBlock, IODevice, and Instruction types.
- Your CPU must be able to process IOInstruction and CPUInstruction objects.
- CPUInstruction objects support the representation of incomplete execution (storing the time remaining – see 2.4).
- To emulate the 'rawness' of a real kernel system call interface, our Java interface uses a variable arguments parameter:

  ```
  public int syscall(int number, Object... varargs);
  ```

  The caller provides zero or more Object parameters depending on the system call number. The kernel must cast each object to the required type e.g a program filename in the case of an EXECVE call.

The components and associated Java Docs are available from the Vula Assignments page.

END