Scheduling Simulator Framework

1. Introduction

In this document we describe the Java framework that may be used to construct simulations of program scheduling. The framework may be used to develop 'discrete event simulations'.

2. 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 according to 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.

3. Simulation principles

The framework supports the development of programs that simulate the execution of a set of programs under a specific scheduling strategy. A set of programs comprises a 'workload'.

3.1 Workload configuration

A simulator (built with the framework) accepts as input a workload configuration file. The file describes I/O devices that are to be represented, and identifies programs that are to be run e.g.

```
# I/O Devices attached to the system.
# DEVICE <device id> <type>
DEVICE 1 DISK
DEVICE 2 CDROM
# Programs
# PROGRAM <arrival time> <priority> <program file name>
PROGRAM 8 0 firefox.dat
PROGRAM 5 0 primescalculator.dat
PROGRAM 11 0 spice.dat
```

- A line beginning with '#' is a comment and is ignored by the simulator.
- A line beginning with 'DEVICE' 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, a priority, 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.

3.2 Program definition

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

```
# firefox.dat
# CPU <duration>
# IO <duration> <device id>
CPU 2
IO 2 1
CPU 3
IO 5 2
CPU 1
```

- 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 virtual time units.

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

The first instruction is always a CPU instruction.

4. Framework Design

In this section we describe the design.

4.1 Conceptual Overview

The framework defines five types of component: kernel, CPU, system timer, I/O device, and simulation configurator/runner.

- The kernel simulates aspects of kernel behaviour relating to scheduling:
  - Processes are represented by Process Control Blocks (PCB).
  - 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, the time spent in user space, the time spent in kernel space, and the time spent idle.
  The timer is advanced when:
  - the CPU processes an instruction, or
  - kernel code is executed, either through system call or interrupt.
- An I/O device simulates the infrastructure needed for an I/O call: the process queues for access to the resource and is released when the operation is completed.

Two kinds of components are defined in abstract: Kernel and ProcessControlBlock. These are interface declarations.

To develop a simulator you must construct:

(i) a concrete type of ProcessControlBlock,
(ii) a concrete type of Kernel, and
(iii) a driver program that can, using the simulation configurator/runner, set up and run a simulation i.e. use it to read in configuration data from a file, 'run' the programs, and write simulation data out to the screen.

4.2 Simulation Configurator/runner

The simulation configurator/runner provides facilities for simulation set up and execution.

Configuration involves reading in a workload configuration:

- For each program identified in the configuration file, create a 'load program' event and insert it in the event queue.
- For each device identified in the configuration file, perform a MAKE_DEVICE system call to the kernel.
- Then set system time to 0 in preparation for running the simulation.

Execution involves running the main simulation loop.

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

4.3 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.

4.3.1 System Calls

Four types of system call 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 an 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 its corresponding queue (where processes are held 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. A 'wakeup' interrupt is scheduled for the point at which the request is due to complete.
3. A scheduling decision is made: a process is switched from the ready queue onto the CPU.

When the kernel receives a TERMINATE_PROCESS call:

1. The currently executing process is removed from the CPU and discarded.
2. A scheduling decision is made: a process is switched from the ready queue onto the CPU.

If the kernel supports some form of pre-emptive scheduling then IO_REQUEST and TERMINATE_PROCESS system calls will also involve cancelling and setting timer interrupts.

1. The timeout scheduled to mark the end of the current time slice is cancelled.
2. The system timer is called to set up a timeout to interrupt execution at the end of the new process time slice.

4.3.2 Interrupts

There are two types of interrupt that the kernel may be required to handle (depending on the scheduling strategy it implements).

Interrupt Number	***Name***	***Arguments***	***Description***
1	TIME_OUT	The ID of the process to be pre-empted.	*Used to signal the end of a scheduled time slice.*
2	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 WAKE_UP interrupt from an I/O device:

1. The relevant device is located.
2. The relevant process is removed from its queue.
3. The process PC is incremented to step over the I/O instruction that has just been completed.
4. The process is placed at the back of the ready queue.

If the kernel is pre-emptive, when it 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.

4.4 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 simulator's configurator/runner. Execution is usually 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 processing user program instructions.

4.5 The System Timer

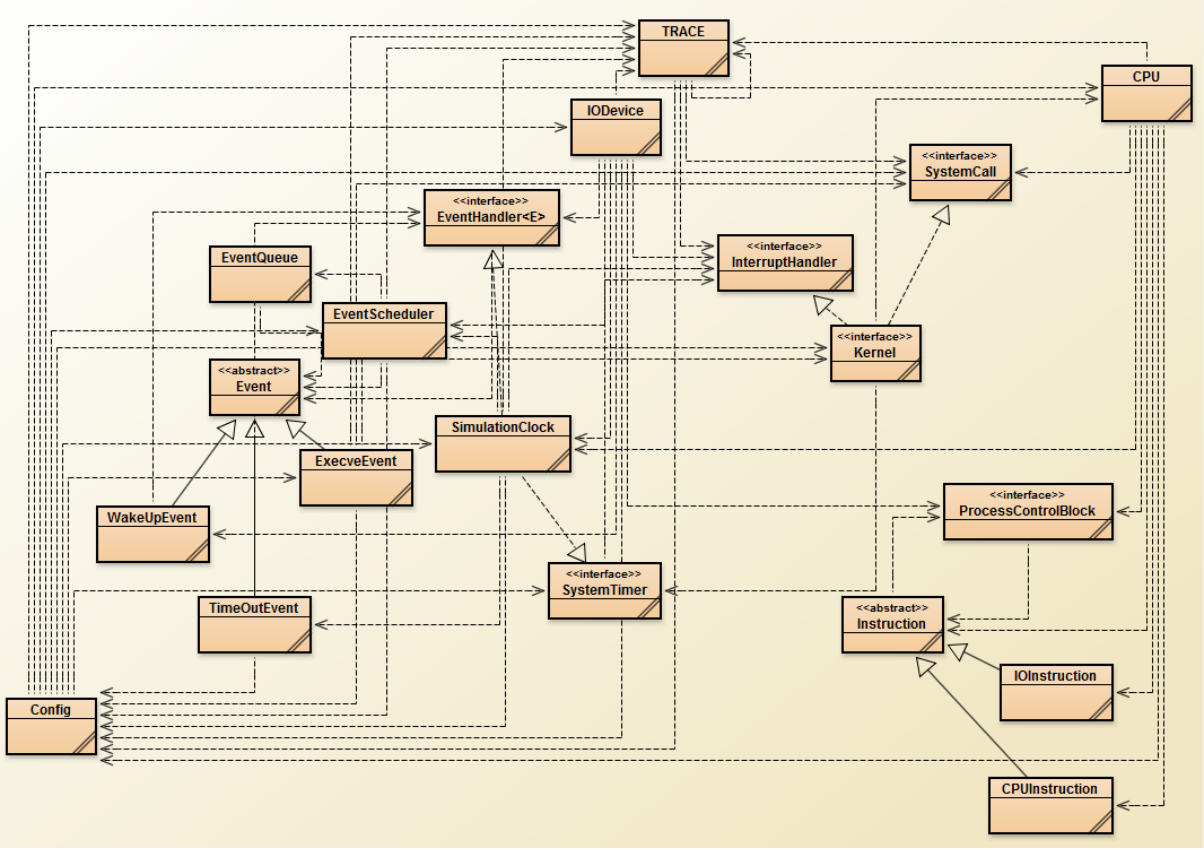
The system timer is essentially a set of counters that track the passage of simulated time.

- The CPU updates user time in the course of simulating execution, and updates kernel time in the course of context switching.
- The discrete event processing infrastructure updates kernel time to represent time spent in system calls and interrupt handling i.e. Kernel implementations do not need to do this.

A kernel may use the system timer to set up TIME_OUT interrupts. (The system timer handles these requests by creating 'timeout' events and placing them in the simulation event queue.)

5 Java Components

The following class diagram depicts the framework structure:



The framework is distributed in a '`.zip`' file containing a source tree structure bearing the name '`src`', a set of tests contained in a directory called '`tests`', and a '`makefile`'.

The '`src`' directory contains sub folders called '`simulator`' and '`programgenerator`'.

- The `simulator` package contains the framework.
- The `programgenerator` package contains a program for generating simulation pseudo programs. (See section 3.2.)

Simulators constructed using the framework can be placed in the top level of the `src` tree.

The `makefile` can be used to (i) compile code to a '`bin`' sub folder, and (ii) generate framework Javadocs, placing them in a '`doc`' sub folder.

The following targets are defined:

- `clean` – remove all '`.class`' files and any `doc` sub directory.
- `framework` – compile the code in the `simulator` package.
- `generator` – compile the code in the `programgenerator` package.
- `simulator` – compile the code at the top of the `src` tree.
- `doc` – generate Javadocs for the `simulator` package.

Your first step, on downloading and unpacking the materials, should be '`make docs`'. The documents are also available as a separate '`.zip`' package.

Compiled files, i.e. '`.class`' files, are written to a '`bin`' directory. The Java class path should point to this directory when simulator code is run. For example, say we have the simulator '`src/SimulateFCFS.java`'. This class is compiled to '`bin/SimulateFCFS.class`', and is run with the following command:

```
java –ea –cp bin SimulateFCFS
```

5.1 Simulator API

The Java documentation provides you with the 'Simulator API', that is, the classes, interfaces, methods and constants that you may use when building a scheduling simulator. It hides the discrete event components that work behind the scenes.

The Kernel and ProcessControlBlock interfaces define types of object that you must implement.

- The Instruction, CPUInstruction, and IOInstruction classes are required when constructing an implementation of ProcessControlBlock.
- The CPU, IODevice and Config classes, the SystemCall and InterruptHander interfaces, and the ProcessControlBlock.State enumerated type are required when constructing a Kernel implementation.

The Config class is the scheduler configurator/runner. It must be used by a simulator main program/method to load a workload and simulate its execution.

It also serves to collate the components required by a kernel implementation.

- A kernel uses it to obtain the current configurations CPU and SystemTimer, and to store and retrieve I/O devices.

5.2 Kernel Interface

To emulate the 'rawness' of a real kernel system call interface, the Kernel interface uses variable argument parameters:

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

public void interrupt(int interruptType, Object... varargs);
```

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

5.3 Simulator main program

A simulator based on the framework typically has something like the following in its main method:

```
TRACE.SET_TRACE_LEVEL(level);
final Kernel kernel = new MyKernel();
Config.init(kernel, dispatchCost, syscallCost);
Config.buildConfiguration(configFileName);
Config.run();
SystemTimer timer = Config.getSystemTimer();
System.out.println(timer);
System.out.println("Context switches:"
                   +Config.getCPU().getContextSwitches());
System.out.printf("CPU utilization: %.2f\n",
                   ((double)timer.getUserTime())/timer.getSystemTime()*100);
```

Line-by-line:

1. Set the level of trace detail (see the final section of this document).
2. Create the kernel.
3. Initialise the simulation with the given kernel, dispatch cost and system call cost.
4. Build the workload configuration described in the given configuration file.
5. Run the simulation.
6. Get the SystemTimer object.
7. Print the SystemTimer (outputs a string describing system time, kernel time, user time, idle time).
8. Print the number of context switches recorded by the CPU.
9. Calculate and print the CPU utilisation based on the available timings.

5.4 TRACE

The TRACE module provides facilities for tracing execution of a simulation. A method is provided that permits the trace 'level' to be set. The level is treated as a binary number on which a bit-wise comparison is performed. Each bit represents a toggle for a particular piece of trace output.

The following table describes the effects:

Bit	***Value***	***Effect***
0	1	*Trace context switches.*
1	2	*Trace system call entry.*
2	4	*Trace system call exit.*
3	8	*Trace interrupt handler entry.*
4	16	*Trace interrupt handler exit.*
5	32	*Trace behind-the-scenes discrete events generation and process.*

By way of an example, setting the trace level to 11 causes context switch, system call entry and interrupt hander entry events to be displayed.

The TRACE module also provides a '`printf`' method that may be used to develop further trace facilities in user/application code – such as concrete kernel classes. The method is similar to the

regular ‘System.out.`printf()`’ command except its first parameter is a trace level value, *v*. The method will only produce output if `TRACE.GET_TRACE_LEVEL()&`*v*`!=0`.

The framework currently, as indicated by the table only uses the first 6 bits of a Java `int` for trace configurations. The rest are available for use in applications.

END