CMPSCI 377: Operating Systems

Lab 2 – Simulating Scheduling

Due: Thursday, October 21

For this and subsequent labs, you will be developing a ***trace-driven simulator*** for an operating system. Rather than implement a real operating system, which can be both time-consuming and error-prone, most researchers use *simulation* to see if their ideas are worth implementing. A *trace* is a representation of program activity, often gathered automatically from real programs. We will be using traces to "drive" our simulation, that is, as input to the simulator.

Schedulers

You will develop two schedulers: a first-come, first-serve (FCFS) scheduler and a multi-level feedback queue (MLFQ) with three queues. You will also need to **generate a set of *synthetic traces*** to test your scheduler implementations. The idea is that by creating simple traces whose properties you understand, you can verify that your scheduler achieves the desired schedule. For each scheduler and set of traces, output statistics of the run, including completion time and wait time for each process as well as the average across processes. Notice that your synthetic traces for MLFQ will need to contain some I/O operations.

Trace format

Our traces will be text files. Each trace file represents one process, and each line is one trace record, containing the following information:

```
<repetitions> <token> <operation-type> (<hex address> | <integer>)
```

repetitions = repetitions of this operation (= one fewer than # of times it's executed)
token – ignore for now
operation-type = string indicating what this trace record corresponds to:

R, r	Read from memory
W, w	Write to memory
I, i	Instruction execution
<	File read
>	File write
S	Sleep
(	File open
)	File close
F	Fork
T	Exit
end	End of trace

For read, write, and instruction execution, the rest of the line is a hexadecimal number corresponding to the page number of the virtual address that is read/written/executed. For sleep, the rest of the line is an integer corresponding to the sleep time; for file I/O, it's the number of bytes read or written, and for file opening and closing, it's the file id.

Implementation

You will implement your simulator by extending the abstract class `AbstractScheduler`, given on the back of this page. I have included the class `RRScheduler` (round-robin) as an example. The simulation framework is provided for you (`psimjava`) and both the abstract scheduler class and psimjava will be available for download on the course homepage (www.cs.umass.edu/~emery/cmpsci377).

AbstractScheduler.java

```java
import psimjava.*;

abstract public class AbstractScheduler extends Scheduler {
  public AbstractScheduler (String name, Input input) {
    super (name, input);
  }

  // These methods are used to add and remove process to and from a
  // scheduler.
  abstract public void add(UserProcess process);
  abstract public void remove(UserProcess process);

  // This method must return the next process to be dispatched.
  abstract public UserProcess schedule();

  // This method will be called if the quantum expired while the indicated
  // process was running.
  abstract public void quantumExpired(UserProcess process);

  // This method will get called whenever a process becomes unblocked;
  // e.g., a Disk I/O operation finished.
  abstract public void unblocked(UserProcess process);

  // The appropriate method will be called whenever a process is blocked.
  abstract public void blockedOnPageFault(UserProcess process);
  abstract public void blockedOnFileRead(UserProcess process);
  abstract public void blockedOnFileWrite(UserProcess process);
  abstract public void blockedOnSleep(UserProcess process);

  // These are informational methods that must be provided.
  abstract public UserProcess getCurrentProcess();
  abstract public boolean noRunnableProcesses();
}
```

RRscheduler.java

```java
import psimjava.*;
import java.util.*;

class RRScheduler extends AbstractScheduler {
  // I'm considering just three states a process can be in -
  // running, runnable and blocked.  The currently running process will also
  // appear in the runQueue.
  private UserProcess currentProcess; // currently scheduled process
  private List runQueue = Collections.synchronizedList(new ArrayList(1000));
  private List waitQueue = Collections.synchronizedList(new ArrayList(1000));

  public RRScheduler(String name, Input input) {
    super(name, input);
  }

  public UserProcess schedule() {
    if (runQueue.isEmpty())
      return null;
    UserProcess process = (UserProcess)runQueue.remove(0);
    runQueue.add(process);
    currentProcess = process;
    return process;
  }

  public boolean noRunnableProcesses() {     return runQueue.isEmpty();  }

  public UserProcess getCurrentProcess() {     return currentProcess;  }

  public void unblocked(UserProcess process) {
    waitQueue.remove(process);
    runQueue.add(process);
  }

  public void add(UserProcess process) {     runQueue.add(process);  }

  public void remove(UserProcess process) {    runQueue.remove(process);   }

  // For round-robin scheduling we really don't care why the process blocked.
  void blocked(UserProcess process) {
    runQueue.remove(process);
    waitQueue.add(process);
  }

  public void blockedOnPageFault(UserProcess process) { blocked(process); }
  public void blockedOnSleep(UserProcess process) { blocked(process); }
  public void blockedOnFileRead(UserProcess process) { blocked(process); }
  public void blockedOnFileWrite(UserProcess process) { blocked(process); }

  public void quantumExpired(UserProcess process) {
  }
}// class RRScheduler
```