Simulation Exercises for Operating Systems

Shivakant Mishra

Simulation Exercise 1: Process Management Simulation

Goal: To simulate five process management functions: process creation, replacing the current process image with a new process image, process state transition, process scheduling, and context switching.

You will use Linux system calls such as *fork*( ), *wait*( ), *pipe*( ), and *sleep*( ). Read man pages of these system calls for details.

This simulation exercise consists of three types of Linux processes: *commander*, *process manager*, and *reporter*. There is one commander process (this is the process that starts your simulation), one process manager process that is created by the commander process, and a number of reporter processes that get created by the process manager, as needed.

Commander Process

The commander process first creates a pipe and then a process manager process. It then repeatedly reads commands (one command per second) from the standard input and passes them to the process manager process via the pipe. There are four types of commands:

1. **Q**: End of one unit of time.
2. **U**: Unblock the first simulated process in blocked queue.
3. **P**: Print the current state of the system.
4. **T**: Print the average turnaround time, and terminate the system.

Command **T** appears exactly once, being the last command.

Simulated Process

Process management simulation manages the execution of *simulated* processes. Each simulated process is comprised of a program that manipulates (sets/updates) the value of a single integer variable. Thus the state of a simulated process at any instant is comprised of the value of its integer variable and the value of its program counter. A simulated process' program consists of a sequence of instructions. There are seven types of instructions as follows:

1. **S** *n*: Set the value of the integer variable to *n*, where *n* is an integer.
2. **A** *n*: Add *n* to the value of the integer variable, where *n* is an integer.
3. **D** *n*: Subtract *n* from the value of the integer variable, where *n* is an integer.

4. **B**: Block this simulated process.
5. **E**: Terminate this simulated process.
6. **F** *n*: Create a new simulated process. The new (simulated) process is an exact copy of the parent (simulated) process. The new (simulated) process executes from the instruction immediately after this (**F**) instruction, while the parent (simulated) process continues its execution *n* instructions after the next instruction.
7. **R** *filename*: Replace the program of the simulated process with the program in the file *filename*, and set program counter to the first instruction of this new program.

An example of a program for a simulated is as follows:

```
S 1000
A 19
A 20
D 53
A 55
F 1
R file_a
F 1
R file_b
F 1
R file_c
F 1
R file_d
F 1
R file_e
E
```

You may store the program of a simulated process in an array, with one array entry for each instruction.

Process Manager Process

The process manager process simulates five process management functions: creation of new (simulated) processes, replacing the current process image of a simulated process with a new process image, management of process state transitions, process scheduling, and context switching. In addition, it spawns a reporter process whenever it needs to print out the state of the system.

The process manager creates the first simulated process (process id = 0). Program for this process is read from a file (filename: *init*). This is the only simulated process created by the process manager on its own. All other simulated processes are created in response to the execution of the **F** instruction.

Process manager: Data structures

The process manager maintains six data structures: *Time*, *Cpu*, *PcbTable*, *ReadyState*, *BlockedState*, and *RunningState*. *Time* is an integer variable initialized to zero. *Cpu* is used to simulate the execution of a simulated process that is in running state. It should include data members to store a pointer to the program array, current program counter value, integer value, and time slice of that simulated process. In addition, it should store the number of time units used so far in the current time slice.

PcbTable is an array with one entry for every simulated process that hasn't finished its execution yet. Each entry should include data members to store process id, parent process id, a pointer to program counter value (initially 0), integer value, priority, state, start time, and CPU time used so far.

ReadyState stores all simulated processes (PcbTable indices) that are ready to run. This can be implemented using a queue or priority queue data structure. *BlockedState* stores all processes (PcbTable indices) that are currently blocked. This can be implemented using a queue data structure. Finally, *RunningState* stores the PcbTable index of the currently running simulated process.

Process manager: Processing input commands

After creating the first process and initializing all its data structures, the process manager repeatedly receives and processes one command at a time from the commander process (read via the pipe). On receiving a **Q** command, the process manager executes the next instruction of the currently running simulated process, increments program counter value (except for **F**or **R** instructions), increments *Time*, and then performs scheduling. Note that scheduling may involve performing context switching.

On receiving a **U** command, the process manager moves the first simulated process in the blocked queue to the ready state queue array. On receiving a **P** command, the process manager spawns a new reporter process. On receiving a **T** command, the process manager first spawns a reporter process and then terminates after termination of the reporter process. The process manager ensures that no more than one reporter process is running at any moment.

Process manager: Executing simulated processes

The process manager executes the next instruction of the currently running simulated process on receiving a **Q** command from the commander process. Note that this execution is completely confined to the *Cpu* data structure, i.e. *PcbTable* is not accessed.

Instructions **S**, **A** and **D** update the integer value stored in *Cpu*. Instruction **B** moves the currently running simulated process to the blocked state and moves a process from the ready state to the running state. This will result in a context switch. Instruction **E** terminates the currently running simulated process, frees up all memory (e.g. program

array) associated with that process and updates the *PcbTable*. A simulated process from the ready state is moved to running state. This also results in a context switch.

Instruction **F** results in the creation of a new simulated process. A new entry is created in the *PcbTable* for this new simulated process. A new (unique) process id is assigned and the parent process id is process id of the parent simulated process. Start time is set to the current *Time* value and CPU time used so far is set to 0. The program array and integer value of the new simulated process are a copy of the program array and integer value of the parent simulated process. The new simulated process has the same priority as the parent simulated process. The program counter value of the new simulated process is set to the instruction immediately after the **F** instruction, while the program counter value of the of the parent simulated process is set to *n* instructions after the next instruction (instruction immediately after **F**. The new simulated process is created in the ready state.

Finally, the **R** instruction results in replacing the process image of the currently running simulated process. Its program array is overwritten by the code in file *filename*, program counter value is set to 0, and integer value is undefined. Note that all these changes are made only in the *Cpu* data structure. Process id, parent process id, start time, CPU time used so far, state, and priority remain unchanged.

Process manager: Scheduling

The process manager also implements a scheduling policy. You may experiment with a scheduling policy of multiple queues with priority classes. In this policy, the first simulated process (created by the process manager) starts with priority 0 (highest priority). There are a maximum of four priority classes. Time slice (quantum size) for priority class 0 is 1 unit of time; time slice for priority class 1 is 2 units of time; time slice for priority class 2 is 4 units of time; and time slice for priority class 3 is 8 units of time. If a running process uses its time slice completely, it is preempted and its priority is lowered. If a running process blocks before its allocated quantum expires, its priority is raised.

Process manager: Context Switching

Context switching involves copying the state of the currently running simulated process from *Cpu* to *PcbTable* (unless this process has completed its execution), and copying the state of the newly scheduled simulated process from *PcbTable* to *Cpu*.

Reporter Process

The reporter process prints the current state of the system on the standard output and then terminates. The output from the reporter process appears as follows:

```
**************************************************************
The current system state is as follows:
**************************************************************\\
```

CURRENT TIME: *time*

RUNNING PROCESS:
pid, ppid, priority, value, start time, CPU time used so far

BLOCKED PROCESSES:
Queue of blocked processes:
pid, ppid, priority, value, start time, CPU time used so far
…
pid, ppid, priority, value, start time, CPU time used so far

PROCESSES READY TO EXECUTE:
Queue of processes with priority 0:
pid, ppid, value, start time, CPU time used so far
pid, ppid, value, start time, CPU time used so far
…
…
Queue of processes with priority 3:
pid, ppid, value, start time, CPU time used so far
pid, ppid, value, start time, CPU time used so far
****

Simulation Exercise 2: Main Memory Allocation

Goal: To simulate and evaluate different memory allocation/deallocation techniques, *first fit*, *next fit*, *best fit*, and *worst fit*, when a linked list is used to keep track of memory usage.

Assume that the memory is 256 KB and is divided into units of 2 KB each. A process may request between 3 and 10 units of memory. Your simulation consists of three components: *Memory* component that implements a specific allocation/deallocation technique; *request generation* component that generates allocation/deallocation requests; and *statistics reporting* component that prints out the relevant statistics. The Memory component exports the following functions:

1. `int allocate_mem(int process_id, int num_units)`: allocates `num_units` units of memory to a process whose id is `process_id`. If successful, it returns the number of nodes traversed in the linked list. Otherwise, it returns -1.

2. `int deallocate_mem(int process_id)`: deallocates the memory allocated to the process whose id is `process_id`. It returns 1, if successful, otherwise –1.

3. `int fragment_count( )`: returns the number of holes (fragments of sizes 1 or 2 units).

You will implement a separate Memory component for each memory allocation/deallocation technique. The request generation component generates allocation and deallocation requests. For allocation requests, the component specifies the process id of the process for which memory is requested as well as the number of memory units being requested. For this simulation, assume that memory is requested for each process only once. For deallocation requests, the component specifies the process id of the process whose memory has to be deallocated. For this simulation, assume that the entire memory allocated to a process is deallocated on a deallocation equest. You may generate these requests based on some specific criteria, e.g. at random or from a memory allocation/deallocation trace obtained from some source.

There are three performance parameters that your simulation should calculate for all four techniques: average number of external fragments, average allocation time in terms of the average number of nodes traversed in allocation, and the percentage of times an allocation request is denied.

Generate 10,000 requests using the request generation component, and for each request, invoke the appropriate function of the Memory component for each of the memory allocation/deallocation technique. After every request, update the three performance parameters for each of the techniques.

The statistics reporting component prints the value of the three parameters for all four techniques at the end.

Simulation Exercise 3: Virtual Memory Simulation

Goal: To simulate and evaluate a virtual memory system, and experiment with different page replacement algorithms. You will need a threads package, e.g. pThreads thread package.

Assume that you have a 16-bit address space, 16 KB of main memory, and 2 KB page size. Virtual memory simulation consists of three components: *virtual address generation* component, *address translation* component, and s*tatistics reporting* component. Implement each component by a separate thread.

The virtual address generation component generates a sequence of 16-bit virtual addresses and writes them in an integer buffer *inBuffer* of size 10. Write a function *getNextVirtualAddress*( ) for generating virtual addresses. This function may generate virtual addresses at random or based on a trace obtained from some source.

The address translation component implements virtual address to physical address translation using a page replacement algorithm. This component reads the next virtual address from *inBuffer* and translates that address to a physical address. It prints the virtual address and corresponding physical address in a file. It also increments an integer variable (*numberOfPageFaults*) on every page fault. Use appropriate bit operations (<<, >>, ~, |, &, etc.) to implement this address translation. Implement a separate version of this component for every page replacement algorithm you want to experiment with.

The statistics reporting component prints the total number of page faults (*numberOfPageFaults*) at the end.