

**Academic Affairs Assessment of Student Learning**

**Report for Academic Year 2023-2024**

**Department/Program: Master of Science in Computer Science**

**Assessment Coordinator’s Name: Michael R. Anderson**

**Assessment Coordinator’s Email Address: andersmr@wvstateu.edu**

1. **Which learning outcomes did you measure this past year?** The MSCS program measured PLOs 3, 4, and 5.
2. **In which course(s) were assessments conducted?**

PLOs 3, 4, and 5 were measured in CS 510. PLOs 4 and 5 were measured in CS 540.

**How did you assess the selected program learning outcomes?** For PLOs 3, 4, 5, measurement is done by correlating performance on class projects with PLO levels.

1. **How many students were included in the assessment(s) of each PLO in a course?**

For measurements in CS 510, thirteen students were included. For measurements in CS 540, eight students were included.

1. **How were students selected to participate in the assessment of each outcome?**

All students enrolled in the class were included.

1. **In general, describe how each assessment tool (measure) was constructed** (i.e. in-house, national, adapted).

The projects were created by the course instructor.

1. **Who analyzed results and how were they analyzed**

The results were discussed by the department assessment coordinator and the department graduate coordinator, who was also the course instructor-of-record.

1. **Provide a summary of the results/conclusions from the assessment of each measured Program Learning Outcome.**

*Data:*

*CS 510:*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| PLO results by student |  |  |  |  |  |  |
| Term |  |  PLO 3 |  PLO 4 |  PLO 5 |  | Raw Score | Conversion: |
| Spring 2024 | 3 | 3 | 3 |  | 26 | Score | PLO Score |
| Spring 2024 | 3 | 3 | 3 |  | 27 | 0-11 | 1 |
| Spring 2024 | 3 | 3 | 3 |  | 23 | 12-20 | 2 |
| Spring 2024 | 3 | 3 | 3 |  | 29 | 21-30 | 3 |
| Spring 2024 | 2 | 2 | 2 |  | 19 |  |  |
| Spring 2024 | 3 | 3 | 3 |  | 26 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 20 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 18 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 17 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 16 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 17 |  |  |
| Spring 2024 | 1 | 1 | 1 |  | 10 |  |  |
| Spring 2024 | 2 | 2 | 2 |  | 12 |  |  |
|  |  |  |  |  |  |  |  |  |
| Average:  |  | 2.3 | 2.3 | 2.3 |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| Distribution by student of PLO results |  |  |  |  |  |
|  | Introducing | Developed | Mastered |  |  |  |
| PLO 3 | 1 |  | 7 |  | 5 |  |  |  |
| PLO 4 | 1 |  | 7 |  | 5 |  |  |  |
| PLO 5 | 1 |  | 7 |  | 5 |  |  |  |

*CS 540:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| PLO results by student |  |  |  |  |  |
| Term |  |  PLO 4 |  PLO 5 |  | Raw Score | Conversion: |
| Spring 2024 | 3 | 3 |  | 20 | Score | PLO Score |
| Spring 2024 | 3 | 3 |  | 20 | 0-7 | 1 |
| Spring 2024 | 3 | 3 |  | 20 | 8-14 | 2 |
| Spring 2024 | 3 | 3 |  | 16 | 15-20 | 3 |
| Spring 2024 | 3 | 3 |  | 16 |  |  |
| Spring 2024 | 3 | 3 |  | 15 |  |  |
| Spring 2024 | 2 | 2 |  | 14 |  |  |
| Spring 2024 | 2 | 2 |  | 11 |  |  |
|  |  |  |  |  |  |  |  |
| Average:  |  | 2.8 | 2.8 |  |  |  |  |
|  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |
| Distribution by student of PLO results |  |  |  |  |
|  | Introducing | Developed | Mastered |  |  |
| PLO 4 | 0 |  | 2 |  | 6 |  |  |
| PLO 5 | 0 |  | 2 |  | 6 |  |  |

Overall, students seem to be near the expected final level for these PLOs.

1. **What are next steps?** (e.g., will you measure this same learning outcome again? Will you change some feature of the classroom experience and measure its impact? Will you try a new tool? Are you satisfied?)

Data collection efforts will continue, with collection happening in more of the courses shown on the curriculum map. Longer term, evaluation of class projects for assessment will be more formalized.

1. **Please attach an example of the assessment tool used to measure your PLO(s).** These can be added as an appendix, a link to the assessment, or sent separately in email with your report.

Simulation Exercises for Operating Systems

**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 For 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

© Copyright 2023 Pearson Education, Inc. All Rights Reserved.

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

©