Test 1

● Graded

Student

Adam Fenjiro

Total Points

53 / 100 pts

Question 1

CPU Modes

Resolved 2 / 4 pts

+ 1 pt System mode all instructions are available

+ 1 pt User mode certain instructions not available

✔ **+ 1 pt** Mode set to system on system call interrupt

✔ **+ 1 pt** OS sets mode to user when control is returned to user process

+ 0 pts None of the above are mentioned or stated incorrectly.

Regrade Request **Submitted on: Feb 17**

Explained in detail and went over everything mentioned in gradescope

You never indicated that certain instruction cannot be executed in user mode and that all instructions can executed in system mode. You only indicated that the OS runs in kernel mode and user programs run in user mode.

Reviewed on: Feb 17

Question 2

Thread Address Space

0 / 4 pts

+ 4 pts Code, data, heap and NOT stack

✔ **+ 0 pts** Anything else

Question 3

Address Space Contents **4** / 4 pts

3.1 **Heap** **1** / 1 pt

✔ **+ 1 pt** Heap

+ 0 pts Anything else

3.2 **Data** **1** / 1 pt

✔ **+ 1 pt** Data

+ 0 pts Anything else

3.3 **Code** **1** / 1 pt

✔ **+ 1 pt** Code

+ 0 pts Anything else

3.4 **Stack** **1** / 1 pt

✔ **+ 1 pt** Stack

+ 0 pts Anything else

Question 4

Timer Interrupt **Resolved** **4** / 6 pts

+ 6 pts Entirely correct

+ 1 pt Timer set on boot

✔ **+ 4 pts** When timer fires, OS regains control

+ 1 pt OS gives CPU to another process

+ 2 pts Identified the timer but did not explain in detail or details were incorrect/vague.

+ 0 pts No answer or did generally explain a timing based sequence of events.

Regrade Request **Submitted on:** Feb 17

Explained in detail and mentioned everything that was in gradescope

There's no mention of the timer being set on boot. There's no mention of the OS taking the process off the CPU. An instruction does not have a timer running alongside it, a process does.

Reviewed on: Feb 17

Question 5

Misc. Scheduling and Threads — **4** / 8 pts

5.1 **Running to Ready** — **2** / 2 pts

- ✔ **+ 2 pts** False
- **+ 0 pts** True

5.2 **Ready to Running** — **0** / 2 pts

- **+ 2 pts** True
- ✔ **+ 0 pts** False

5.3 **Thread I/O** — **2** / 2 pts

- ✔ **+ 2 pts** True
- **+ 0 pts** False

5.4 **Thread Scheduling** — **0** / 2 pts

- **+ 2 pts** False
- ✔ **+ 0 pts** True

Question 6

Code to Address Space Mapping — **4** / 8 pts

6.1 **Globals** — **2** / 2 pts

- ✔ **+ 2 pts** Data
- **+ 0 pts** Anything else

6.2 **Statics** — **0** / 2 pts

- **+ 2 pts** Data
- ✔ **+ 0 pts** Anything else

6.3 **Locals** — **0** / 2 pts

- **+ 2 pts** Stack
- ✔ **+ 0 pts** Anything else

6.4 **Dynamically allocated** — **2** / 2 pts

- ✔ **+ 2 pts** Heap
- **+ 0 pts** Anything else

Question 7

Base Bounds Memory Management **2** / 10 pts

7.1 **Bounds Physical** **0** / 2 pts

+ 2 pts 4095 (or 4096)

✔ **+ 0 pts** Anything else

7.2 **Interprocess** **2** / 2 pts

✔ **+ 2 pts** Cannot be determined from the information given.

+ 0 pts Anything else

7.3 **Reference** **0** / 3 pts

+ 1 pt VA + BASE = 5144

+ 1 pt > Bounds=4096

+ 1 pt Interrupt is raised

+ 3 pts Entirely correct

✔ **+ 0 pts** No answer, incorrect translation, or none of explanation points above

7.4 **Bounds Virtual** **0** / 3 pts

+ 3 pts 2048+X<4096 X=2047 (Off by 1 OK)

+ 2 pts Correct answer with no or vague explanation.

✔ **+ 0 pts** Anything else

Question 8

Instruction Interleaving **11** / 14 pts

8.1 **Trace Values** **6** / 6 pts

+ 6 pts Entirely correct

- ✔ **+ 1 pt** 2 in R after both LOAD instructions
- ✔ **+ 2 pts** 1 in R after SUB
- ✔ **+ 2 pts** 3 in R after ADD
- ✔ **+ 1 pt** Store of 1 then store of 3; 3 persists

8.2 **Understand context-switch** **0** / 3 pts

+ 3 pts Entirely correct

+ 1 pt From a starting value of 2

+ 2 pts Value of R in PB not seen in PA because of the context switch

- ✔ **+ 0 pts** No answer. Incorrect or invalid answer/explanation

8.3 **Generate interleaving to specified outcome** **5** / 5 pts

- ✔ **+ 5 pts** Entirely correct

+ 3 pts All instructions in PA then PB or vice versa

+ 2 pts Execution values entirely correct

+ 0 pts No answer, loads interleaved, missing instructions

Question 9

Process Tree created with fork() **11** / 17 pts

9.1 **One fork** **5** / 5 pts

✔ **+ 5 pts** Two processes with parent child relationship

+ 0 pts Anything else

9.2 **Conditional second fork** **6** / 6 pts

✔ **+ 6 pts** Child in previous tree creates a child; Tree height is 3 with three nodes

+ 0 pts Anything else

9.3 **Conditional third fork** **Resolved** **0** / 6 pts

+ 6 pts Root of previous tree creates a child; tree height still 3; four nodes; root has two children

✔ **+ 0 pts** Anything else

Regrade Request **Submitted on: Feb 17**

Properly added a child to the root node and did not add a node to the last child, but added only 1 extra node to P2 since it is considered parent of P3

This makes it incorrect. There is no easy way to give partial credit on a problem like this. I can't know what you are thinking to cause a mistake so I try to keep it relatively easy and keep the number of points relatively small.

Reviewed on: Feb 17

Question 10

Hardware-based Mutual Exclusion **0** / 14 pts

10.1 **Create mutual exclusion with hardware instruction** **Resolved** **0** / 8 pts

+ 8 pts Entirely correct

+ 2 pts Lock variable initialized correctly

+ 3 pts while loop condition correct

+ 1 pt ; after the while

+ 2 pts Lock variable reset after critical section

✔ **+ 0 pts** Incorrect.

Regrade Request **Submitted on: Feb 17**

Explained how CS works and why it is mutual exclusive

The problem was to show how to use it. The description is only of the instruction, which was given in the test problems. This answer does not show how to use the instruction to achieve mutual exclusion.

Reviewed on: Feb 17

10.2 **Create execution table + bounded waiting argument** **Resolved** **0** / 6 pts

+ 2 pts Gives an execution table

+ 3 pts Execution table demonstrates violation of bounded waiting correctly

+ 1 pt States that bounded waiting does not hold

✔ **+ 0 pts** Indicates it provides bounded waiting or no answer or argues from an invalid solution.

Regrade Request **Submitted on: Feb 17**

Explained how CS is bounded waiting and gave an example of execution table

The execution table shows that you don't really understand how to use the instruction to achieve mutual exclusion. There's no way to show bounded waiting does not hold in a given solution when there's no solution.

Reviewed on: Feb 17

Question 11

Software-based Mutual Exclusion **11** / 11 pts

+ 3 pts Essentially correct execution but does not demonstrate P1 getting two turns before P0

+ 5 pts Correct execution in which P0 gets two turns before P1 get one

✔ **+ 11 pts** Correct execution table in which P1 gets two turns before P1 gets 1 and P1 executes every instruction in the while loop at line 2

+ 0 pts Incorrect, ambiguous or no execution

– 2 pts No line numbers

NAME: Adam FANJIRO

CS 3331 Exam 1

Answer Sheet

Feb 12, 2025

1. [4 points]

- CPU modes are operating modes of the CPU. There is two execution modes: User mode and Kernel (or system) mode, they are controlled by a mode bit. The operating system is responsible of running the kernel mode, while the user programs are responsible of running the user mode.
- When running a user programme, the mode is user for now. When there is a system call, there is an interrupt, which suspends the execution of the program and switches to Kernel mode and gives control to OS, which analyzes the system call and gives it the specific system call handler, when taken care of, the context switches back to user mode and the process keeps running.

2. [4 points]

- ○ Code
- ● Data
- ○ Heap
- ● Stack

3. (4 points)

(a) [1 point] The heap section of the address space holds dynamically allocated memory.

(b) [1 point] The section of the address space that holds global and static data is the data section.

(c) [1 point] The section of the address space that holds instructions is the code section.

(d) [1 point] The section of the address space that holds activation records is the stack section.

4. [6 points]

when a program is running, the process is running and the CPU is running on a user mode, however, alongside that, every instruction running has a timer along side it that creates an interrupt upon running out, which forces the CPU to get the OS to gain back control in kernel mode, making sure that the instruction is not monopolizing.

5. (8 points)

(a) [2 points] F A transition from the running state to the ready state is often caused by a request for I/O.

(b) [2 points] F A transition from the ready state to the running state is often caused by a timer interrupt.

(c) [2 points] T In user-level threads, when one thread in a task blocks on I/O, all the other threads are also blocked.

(d) [2 points] T In user-level threads, scheduling decisions are always made at the kernel level.

6. (8 points)

(a) [2 points] The address of `totalAccounts` will be in the data section.

(b) [2 points] The address of the variable `transactionCount` will be in the stack section.

(c) [2 points] The address of the variable `accounts` will be in the code section.

(d) [2 points] After a successful call to `malloc` at line 16, the value of `accounts` will be an address in the heap section of the address space.

7. (10 points)

(a) [2 points]

3054

(b) [2 points]

Cannot be determined.

(c) [3 points]

Blocks execution since it is larger than what it can handle

(d) [3 points]

Cannot be determined.

8. (a) [6 points]

Process A Instructions	R	Process B Instructions	R	count value
				2
		LOAD R, count	2	2
LOAD R, count	2			2
		ADD R, #1	3	2
SUB R, #1	1			2
STORE R, count	1			1
		STORE R, count	3	3

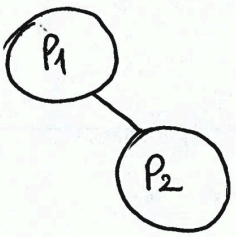
(b) [3 points]

- if both processes use the same CPU register R, it means that they load the ressources from the same register R and write to the same register R, thus storing in memory from the same register R.
- since the instruction ADD R,#1 is before the SUB R,#1, then we will sub from the value of count after adding #1 to the shared register R. Thus, we will SUB starting from the value 3.

(c) [5 points]

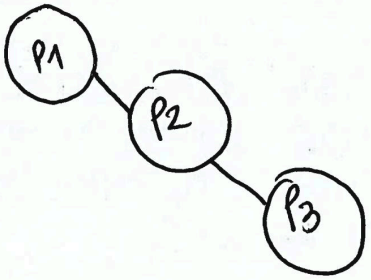
Process A Instructions	R	Process B Instructions	R	count value
		[illegible]		2
		LOAD R, count	2	2
		ADD R, #1	3	2
		STORE R, count	3	3
LOAD R, count	3			3
SUB R, #1	2			3
STORE R, count	2			2

9. (a) [5 points]



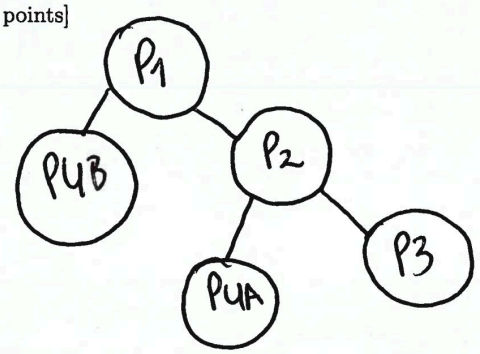
*cpid = fork() means we fork and create a child process

(b) [6 points]



*(cpid == 0) fork() means we fork from child process

(c) [6 points]



#(cpid != 0) fork() means we fork from processes that are not children to that process.

10. (14 pts)

(a) [8 points]

- Compare-and-Swap (CS) instruction is a solution to synchronizing threads. It is a solution on the hardware level. to prove that it is mutually exclusive, we will use the proof by contradiction and see if it holds or contradicts.
- How it work? it is a boolean return type function that takes a pointer with its old and new value. If the pointer value is not equal the old value, it return false. else it sets the pointer to the new value and return true. Thus it is an atomic instruction and the proof holds with no contradicts. CS is mutually exclusive

(b) [6 points]

shared int value=b; CS is an atomic instruction thus it ensures mutual exclusion, progress, but no bounded waiting.

Process N:	Process A:	Process B:	Process C:	...
CS(ptrN, n-1, n)	CS(ptr1, 10, 1)	CS(ptr2, 1, 2)	CS(ptr3, 2, 3)	

Process A	Process B	...	Process N	value
				10
				10
LOAD R, value				1
CS(ptr, 0, 1)				1
STORE R, value				1
	LOAD R, value			2
	CS(ptr, 1, 2)			
	STORE R, value		LOAD R, value	
			CS(ptrN, n-1, n)	n
			STORE R, value	

Using the induction proof, the analogy I explained through the execution table shows no bounded number of processes that can enter their critical section. Thus no bounded waiting for CS.

11. [11 points]

P_0 Instructions	flag[0]	P_1 Instructions	flag[1]
		1: flag[1]=TRUE	T
1: flag[0]=TRUE	T		
		2: while (flag[0]) {	T
2: while (Flag[1]) ?	T		
		3: flag[1] = FALSE	F
3 [illegible]	[illegible]	4: while (flag[0]) ?	F
		5: flag[1] = TRUE	T
3: flag[0] = FALSE	F		
4: while (flag[1])	F		
5: flag[0] = TRUE	T		
2: while (flag[1])	T		
		2: while flag[0]	T
		3: flag[1] = FALSE	F
[illegible]		4: while flag[0]	F
3: flag[0] = false	F		
4: while (flag[1])	F		
// critical	F		
		5: flag[1] = true	T
2: while (flag[1])	F		
2: flag[0] = false	F		
3: while (flag[1]	F		
4: flag[0] = true	T		
		2: while (flag[0])	T
		3: flag[1] = false	F
2: while (flag[1]	T		
// critical			

P1 executed at least time ✓

#1

P0 executed critical section twice ✓

2 #