Operating Systems **Midterm Exam** ECE344, Fall 2019

Duration: 1 hour 15 min

Examiners: M.Stumm, A. Goel

Please fill your student number and name below and then read the instructions below carefully.

Student Number: |___|___|___|___|___|___|___|___|___|___|

First Name: |___|___|___|___|___|___|___|___|___|___|___|___|___|___|

Last Name: |___|___|___|___|___|___|___|___|___|___|___|___|___|___|

Instructions

Examination Aids: No examination aids are allowed.

Do not turn this page until you have received the signal to start.

Do not remove any sheets from this test book. Answer all questions in the space provided. No additional sheets are permitted. Use the last blank page as scratch space. This page will not be marked.

This exam consists of 5 questions on 11 pages (including this page). The value of each part of each question is indicated. The total value of all questions is 42.

For the written answers, explain your reasoning clearly. Be as brief and specific as possible. Clear, concise answers will be given higher marks than vague, wordy answers. Marks will be deducted for incorrect statements in an answer. Please write legibly!

Work independently.

MARKING GUIDE

Q1:________	(10)
Q2:________	(10)
Q3:________	(5)
Q4:________	(7)
Q5:________	(10)
TOTAL:________	(42)

Question 1. **Multiple Choice** [10 MARKS]

For each of the questions below, circle the correct answer. There is only one correct answer.

1) Which of the following is responsible for translating a process's virtual memory address into physical memory address?

A. Compiler

B. Kernel

[✓ C.] Hardware

D. Memory

2) Which of the following will *not* happen during a system call?

A. Kernel saves all the general purpose registers

B. Processor saves the program counter

C. Processor executes the syscall instruction

[✓ D.] In Lab 3, when a thread invokes a system call, another thread runs user threads cannot run when any of them makes a system call

3) Which of the following *may not* happen during a context switch between two processes?

A. Kernel saves general purpose registers

B. Processor saves the program counter

[✓ C.] Processor executes the `syscall` instruction context switch can happen due to interrupts

D. Kernel restores the MMU registers

4) Which of the following *isn't* provided in the original Unix Operating System?

A. execv()

B. fork()

C. getpid()

[✓ D.] pthread_create()

5) Which of the following will *not* cause a process to trap into the kernel?

[✓ A.] Save the current execution context This is what getcontext does in user space

B. Execute an illegal instruction

C. Access kernel space memory

D. Write to a peripheral device register

6) Which of the following is *not* the purpose of a system call?
A. Allow a process to create kernel threads
[✓ B.] Allow a process to invoke the C printf function   printf is a library function.
C. Allow the C printf function to perform its operation
D. Allow processes to communicate with each other

7) Which of the following is *false* when deadlock happens?
A. Some processes in the system can still make progress
B. The wait-for dependency graph must contain a cycle
[✓ C.] Resolving the deadlock requires killing all the deadlocked processes   Just need to kill enough processes so that cycles are broken
D. Deadlock could have been caused by synchronization primitives other than locks

8) Which of the following is *not* a requirement for a mutual exclusion solution?
A. No two threads are simultaneously in the critical section
B. A thread outside a critical section may not block another thread
[✓ C.] A thread must contain a critical section   If a thread doesn't use shared variables, it doesn't need any critical section
D. A thread should not wait forever to enter its critical section

9) Which of the following will *not* occur for the Unix `wait` system call?
A. Parent waits for child to exit
B. Parent exits before child exits
[✓ C.] Child waits for parent to exit   Unix wait does not allow a child to wait for a parent
D. Child exits before parent exits

10) When critical sections are long and contended, the best type of mutual exclusion lock is:
A. Atomic instruction
B. Interrupt disabling
C. Spin lock
[✓ D.] Blocking lock

Question 2. **Synchronization** [10 MARKS]

Mark has created a simple function called `add` that adds 1 to each element of a global array called `Array`. All the elements of `Array` are initialized to 0. Assume `size` is the size of the array (a constant). Also, assume that the only way that `Array` is updated is using the `add` function.

An invariant (constraint) that should hold when the `add` function is not executing is that all elements in the array have the same value. Mark is a meticulous programmer and uses the `assert` function in the `add` function to check the invariant, i.e., that all the elements of the array have the same value:

```
int Array[size];

add() {
    int i;
    for (int i = 0; i < size; i++) {
        if (i < size - 1) { // no check needed for last element
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
    }
}
```

Part (a) [2 MARKS] When Mark runs the `add` function within a single thread, it works as expected. Mark has recently learned how threads can be used to speed up programs and decides to create two threads, each of which run the `add` function. Mark finds that his multi-threaded program crashes randomly. Provide a scenario that could lead to a crash.

There are many possible scenarios. The reason for the crash is that the assert will fail since different invocations of the add function can update different elements in any order.

Part (b) [2 MARKS] Mark has learned that locks can fix random crashes and decides to fix the problem by using locks as follows:

```
int Array[size];
lock l;

add() {
    int i;
    lock(l);
    for (int i = 0; i < size; i++) {
        if (i < size - 1) { // no check needed for last element
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
    }
    unlock(l);
}
```

Will his solution fix the random crashes? If so, briefly explain why. If not, provide a scenario that could lead to a crash.

This code will fix the random crashes. The locking ensures that each invocation of add will run to completion before another add runs, ensuring that the assertion will remain true.

Part (C) [3 MARKS] Mark has recently learned that using fine-grained locks can provide better parallelism. He decides to create a lock for each element of the array and lock this element when updating it, as shown below.

```
int Array[size];
lock l[size];

add() {
    int i;
    for (int i = 0; i < size; i++) {
        // lock current element
        lock(l[i]);
        if (i < size - 1) { // no check needed for last element
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
        // unlock current element
        unlock(l[i]);
    }
}
```

Does this fine-grained locking solution work? If so, briefly explain why. If not, provide a scenario that could lead to unexpected behavior.

This code will not work and cause crashes again. The problem is that while each element of the array is updated atomically, the assertion is checking the current and the next value. This assertion may not hold during concurrent add operations. For example, say Thread 1 updates Array[0]. Before it updates Array[1], say Thread 2 starts running. When it checks Array[0] and Array[1], they will not be the same, thus the assertion will fail.

Part (D) [3 MARKS] As Mark learns more about fine-grained locking, he decides to use the strange locking scheme shown below.

```
int Array[size];
lock l[size];

add() {
    int i;
    // lock first element
    lock(l[0]);
    for (int i = 0; i < size; i++) {
        if (i < size - 1) { // no check needed for last element
            // lock next element
            lock(l[i+1]);
            // check that current element is the same as the next element
            assert(Array[i] == Array[i + 1]);
        }
        // then update current element
        Array[i] += 1;
        // unlock current element
        unlock(l[i]);
    }
}
```

Does this fine-grained locking solution work? If so, briefly explain why. If not, provide a scenario that could lead to unexpected behavior.

This code works correctly. The reason is that it solves the problem in the previous version. In this case, the code locks the current and the next element before checking the assertion, which guarantees that these two values will be the same. Then the code unlocks the current element, allowing any concurrent later adds() to proceed to update earlier elements of the array. In effect, the threads are pipelining updates to the array.

It may appear that the locking is assymmetric (more lock calls than unlock calls) but that is not the case.
Iteration i = 0: lock(0), lock(1), unlock(0)
Iteration i = 1: lock(2), unlock(1)
...
Iteration i = size - 2: lock(size - 1), unlock(size - 2)
Iteration i = size - 1: unlock(size - 1) (last iteration doesn't perform a lock)

Question 3. Unix Processes [5 MARKS]

When you run the `cockroach` function, it seems to slow your machine down.

```
void cockroach()
{
    for (int i = 0; i < 20; i++) {
        fork();
    }
}
```

Part (a) [2 MARKS]

How many processes does the cockroach function create? Explain why.

2^20 - 1 = (1 + 2 + 2^2 + 2^3 + ... + 2^19)

The first call to fork creates one additional process. Now the two processes create two additional processes. Then the four processes create four additional processes ...

Part (b) [3 MARKS]

Write a function similar to the `cockroach` function that creates only 20 processes. You may only use the `fork`, `wait`, `execv` and `exit` system calls. Long solutions will be penalized.

```
void well_behaved_process()
{
    for (int i = 0; i < 20; i++) {
    // Add your solution here
       if (fork()) {
          exit(0);
       }

       or

       if (fork() == 0) {
          exit(0);
       }
    }
}
```

Question 4. Unix Processes [7 MARKS]

The `power` program accepts two numbers, a base and an exponent, as command-line arguments. This program should output the base to the power of the exponent, followed by the base to the power of the exponent – 1, followed by the base to the power of the exponent – 2, and so on. This should repeat until 1 is printed. You may assume that the two arguments are non-negative numbers. A sample output of the `power` program is shown below:

```
> ./power 2 3
8
4
2
1
```

Below, we have provided a partial listing of the `power` program. Complete the program so that it works correctly. You can only add code in the locations indicated by **bold comments** below, and you cannot comment out any code. Your solution *must* invoke the `execv` system call shown in the code. Long solutions will be penalized.

```
int
main(int argc, char **argv)
{
    int base = atoi(argv[1]);
    int exp = atoi(argv[2]);
    // your code here (if needed)
    int i, power = 1;

    // calculate and print exp^base
    for (i = 0; i < exp; i++)
        power = power * base;

    printf("%d\n", power);
    if (exp > 0) {
        char exp_buffer[20];
        // snprintf converts the number exp – 1 to a string,
        // which is stored in exp_buffer
        snprintf(exp_buffer, 20, "%d", exp - 1);
        char *args[4] = {argv[0], argv[1], exp_buffer, NULL};
        execv(“./power”, args);
    }
    // your code here (if needed)



    return 0;
}
```

Question 5. **Scheduling** [10 MARKS]

Consider the 8 threads shown below, with their associated arrival and processing times:

Job	Arrival Time	Processing Time	Job	Arrival Time	Processing Time
A	0	6	E	8	1
B	2	3	F	9	2
C	3	1	G	12	1
D	5	4	H	13	2

You realize that shortest remaining time (SRT) scheduling can pre-empt processes frequently thus reducing throughput. You design a variant of SRT scheduling called shortest remaining time with time slice (SRTT) scheduling. SRTT scheduling uses shortest remaining time scheduling, but has a time slice of 2 time units. Thus threads are not pre-empted unless they have run at least 2 time units.

When there is a tie, assume first-come-first-served (FIFO) scheduling. If a thread finishes without using up its time slice, scheduling occurs immediately. Also assume that threads arrive just before the time slice expires.

Show the sequence of threads that run in each time unit on a single processor using SRTT scheduling in the boxes shown below. Under each box, provide your reason for scheduling the thread using letters S, F, and T:

1) S: <u>S</u>hortest remaining time

2) F: tie, so <u>F</u>IFO scheduling

3) T: thread is not pre-empted due to <u>T</u>ime slice

	↓A		↓B	↓C		↓D			↓E	↓F
Job	A	A	B	B	B	C	A	A	E	A
Reason	S	S	S	T	F	S	F	S	S	F
	0	1	2	3	4	5	6	7	8	9

			↓G	↓H						
Job	A	F	F	G	H	H	D	D	D	D
Reason	S	S	F	S	S	S	S	S	S	S
	10	11	12	13	14	15	16	17	18	19

[Use the space below for rough work.]