CSCI3150 Intro to Operating Systems, Spring 2023 MIDTERM EXAMINATION - Answers

T1. Intro and Arch Support(1 mark)

Multiple choices. There are one or more correct choices for each question. Mark all of them. For each question, you will get 0.5 marks only if your answer is 100% correct, otherwise you will get 0.

Question 1.a (0.5 marks) (Multiple choices) Which of the following is/are always protected instructions/operations?

(A) Load (read from memory)
(B) Store (write to memory)
(C) Halt instruction
(D) Modify the CPU mode bit
(E) Call instruction
(F) Operations within an event handler
(G) None of the above

Answer: C, D, F

Question 1.b (0.5 marks) (Multiple choices) When a user application executes a divide-by-zero instruction, it causes a fault and the application is subsequently terminated. Which of the followings are true:
(A) OS is the first to detect that a divide-by-zero fault has occurred
(B) Right before this fault occurs, the CPU has to be in kernel mode
(C) The CPU directly terminates the faulting application
(D) None of the above

Answer: D

T2. Processes (1 mark)

Question 2. Read the following program and answer the questions:

```
void main(){
    int i;
    fork();
    for(i=0;i<5;i++)
    {
        int pid = fork();
        if(pid == 0)
            printf("*\n");
        else
            printf("o\n");
    }
    return;
}
```

a) How many processes are created when we execute this program once in a shell, including the first process created by the shell. No explanation is needed.

b) How many times would * and o be printed?

Answers:
a) 64

Explanation:
The first fork() would create a child process, and there are two processes after the first fork(). In the iteration, each existing process would produce a new child process, one with the pid=0, the other with non-zero pid. Then
a) The number of processes after 1 iterations is: $2 * 2 = 4$
The number of processes after 2 iterations is: $2 * 2 * 2 = 8$
.....
The number of processes after 5 iterations is: $2^6 = 64$
b) In the iteration, each existing process would produce a new child process.
The number of * is: $2 + 4 + 8 + 16 + 32 = 62$
The number of o is: $2 + 4 + 8 + 16 + 32 = 62$

T3. Semaphores (2 marks)

Question 3: There are 5 operations, denoted as A, B, C, D, and E. Operation C must be executed after operations A and B are completed. Operation E must be executed after operations C and D are completed. Please use the wait() and signal() operations of semaphores to describe the synchronization relationships between these operations and explain the semaphores used and their initial values.

You can write pseudo code, and use `finish_X()` to represent the instructions of operation `X` without the semaphore.

Answer:

```
Semaphore SAC=0; //Control the execution order of A and C.
Semaphore SBC=0; //Control the execution order of B and C.
Semaphore SCE=0; //Control the execution order of C and E.
Semaphore SDE=0; //Control the execution order of D and E.

A(){
 finish_A();
 signal(SAC);
}
B(){
 finish_B();
 signal(SBC);
}
C(){
 wait(SAC);
 wait(SBC);
 finish_C();
 signal(SCE);
}
D(){
 finish_D();
 signal(SDE);
```

```
}
E(){
 wait(SCE);
 wait(SDE);
 finish_E()
}
```

T4. Producer/Consumer Problem (1 mark)

Question 4: Consider the producer/consumer problem: There are multiple producer threads and multiple consumer threads. All of them operate on a shared bounded buffer. A producer inserts data into the buffer one item at a time, and a consumer removes data from the buffer one item at a time. The correctness criteria is that the producers cannot insert into a buffer that is full, and consumers cannot remove any items from an empty buffer.

Now consider the following code:

```
void *producer(void *arg) {
   while(true) {
        lock_acquire(&lock);
        while (number_of_items(buffer) == MAX) //MAX is the buffer length.
            cv_wait(&cv, &lock);
        insert_item(buffer); //inserts an item into shared buffer
        cv_broadcast(&cv);
        lock_release(&lock);
   }
}
void *consumer(void *arg) {
    while(true) {
        lock_acquire(&lock);
        while (number_of_items(buffer) == 0)
            cv_wait(&cv, &lock);
        tmp = consume_item(buffer); //removes an item from the shared buffer
        cv_broadcast(&cv);
        lock_release(&lock);
        printf("%d\n", tmp);
    }
}
```

Assume the lock and condition variable are properly initialized. Now, does this code work correctly? Why? (Notice the use of `broadcast()` for condition variables.)

Answer:

Yes. This code seemingly makes the mistake of only using one condition variable in the producer/consumer problem, but solves it by using broadcast to wake all waiting threads. Because each thread re-checks the condition when awoken, only those who should be able to make progress will do so. Of course, this can be inefficient, which is why we normally use two CVs and signal accordingly thus waking up only those who need to be awoken.