Review Test Submission: OS Exam1 5348

User	Yaokun Wu
Course	CS 5348.001 - Operating Systems Concepts - S21
Test	OS Exam1 5348
Started	2/26/21 10:03 AM
Submitted	2/26/21 11:29 AM
Status	Completed
Attempt Score	68.5 out of 144 points
Time Elapsed	1 hour, 25 minutes out of 2 hours
Instructions	

Question 1

7 out of 21 points

1. Assume that a computer system from time 0 to 59 always has 4 processes, P1 to P4. At time 0, all 4 processes are in the ready queue and they are in the order of P1, P2, P3, P4. The CPU scheduler (same as the one implemented in the process manager in simOS) picks the next process in its ready queue to run when the previous one has to stop execution due to various reasons. Each process is given a time quantum of 10 time units. When a process becomes ready, it will be added to the end of the ready queue. The following are all events that have taken place in the system from time 0 to 29.

At time 0: P1 is retrieved from ready queue for execution.
At time 5: P1 executes a command to read disk unit 1.
At time 15: P2's time quantum expires.
At time 22: P3 executes a sleep command to sleep for 15 time units.

At time 26: P3 got swapped out of memory.
At time 28: The current process in execution calls sem_signal(x) when x's count is -3.

From time 30 to time 54, no special events have taken place, but normal executions with several time quantum expiration events and P3's sleep time up event. From time 55 to 59, the following events have taken place.

At time 55: P1's read has completed.
At time 57: The current process in execution calls sem_wait(x) when x's count is -3.

(a) Which process is in execution at time **38**? **[A]**

(b) Which process is in execution at time **58**? **[B]**

(c) What times the timer interrupts have occurred between time 0 to 59. Note: you need to consider **all types** of events that may cause a timer interrupt. List all the time points in order, from smallest to largest. Use a comma to separate each and no space in your answer (e.g., 3,13,20,25). **[C]**

Note: This question tells you how the CPU scheduler schedule processes, so you can know exactly which process is in execution at each time instance.

Question 2

7.5 out of 10 points

2 (a) Consider the following events occurred in the system. Which of them will cause a **process switch**?

Question 3

0 out of 8 points

2 (b) Consider the events given in 2(a). Which of them will **NOT** cause a **context switch**? List the labels of all items that apply in alphabetical order with no space in between (e.g., ADG). **[B]**

Question 4

6 out of 10 points

2 (c) When a context switch is needed for the situations you considered in 2(b), which of the following registers should be saved.

Question 5

3 out of 9 points

3. The code below is not a typical way of handling fork() and pipe(), but to test your understanding of their behaviors.

```
int a = 0;   int b = 0;   int c, d;
int pfd[2];     // pipe file descriptor

void parent_function (int arg)
 {   a = 3;   b = 2;   c = 2*a + b;   d = a + 2*b;
     pipe (pfd);
     if (arg != 0) write (pfd[1], &c, sizeof(int));
     else write (pfd[1], &d, sizeof(int));
     printf ("parent:%d,%d\n", a, b);
     close (pfd[1]);
 }

 void child_function ()
 {   read (pfd[0], &a, sizeof(int));
     read (pfd[0], &b, sizeof(int));
     printf ("child:%d,%d\n", a, b);
     close (pfd[0]);
 }

 void main ()
 {   int ret = fork ();
     parent_function (ret);
     if (ret == 0) child_function ();
     wait();
 }
```

Give the output of the **child** process. Put each line of output in the corresponding box below, following the order of printing. We give 3 boxes for three lines of output. If the output only contains 1 line, you should put "**none**" (without the quotation marks) for lines 2 and 3. Your answer for each line should contain no space (e.g., parent:2,0), (e.g, child:2,0).

First output line: **[A]**

Second output line: **[B]**

Third output line: **[C]**

Note: The code above works fully. Pay more attention to the behaviors for system calls pipe() and fork(), and read() and write() are just used normally.

Note: In case you are not familiar with sytem calls read() and write(), here is the spec. Function int ret = read(int fd, char *str, int size) can be used for reading from any file descriptor fd, what is being read is placed in str, size is the number of bytes to be read, and the number of bytes actually being read is returned to ret, where ret <= size. The case 0 < ret < size happens when there is data in fd, but the number of bytes is less than size. The case ret = 0 happens when read() encounters EOF (otherwise, read() will wait). If ret = 0 or ret = -1, then str will not be touched (unchanged). The parameters and return values for write() are very similar to those for read(). If an fd opened for write is closed after some writing, reader will get EOF after exhausting all data in fd. The code does not do return value check so that it will not be too complex.

Question 6

21 out of 21 points

4. Consider the following pseudo code for thread execution. For “create-thread”, the first parameter is the thread name, and the second is the function to be executed by the thread.

Function F() is coded in pseudo instructions. The indexes are for reference only. We use S1 and T1 to refer to the execution of the first instruction in F() by thread S and thread T, respectively. The same applies to the other instructions. (Note: There are two instructions in F() indexed by 2 and the same for index 4. This is to make your analysis simpler because combination of the two instructions will not make a difference to the result.)

```
int a = 10;

F ()
{  1.  load a;
   2.  add -3;   store a;
   3.  load a;
   4.  add 1;   store a;
}

main()
{  create-thread ( “S”, F() );
   create-thread ( “T”, F() );
   wait for threads to finish;
   print a;
}
```

(a) What is the minimal value that may be printed by the above program. **[A]**

(b) What is the maximal value that may be printed by the above program. **[B]**

(c) Give an execution sequence that will yield the maximal print out value. In your answer, the instruction indexes should be separated by comma and there should be no space in between. (An example can be: S1,S2,S3,S4,T1,T2,T3,T4) (There are many correct answers and no problem as long as your answer is correct and follows the format.) **[C]**

Question 7

0 out of 10 points

5. Consider two threads P and Q with their code (at the instruction level) shown below.

(Thread P)
P1 load a
P2 add b
P3 add 1
P4 store a

(Thread Q)
Q1 load c
Q2 add 2
Q3 store a
Q4 store b

Alice is trying to analyze the code and find out what are the possible values for a and b when running P and Q concurrently. She tries to simplify the analysis by combining instructions that can be considered together as one unit without eliminating any possible results for a and b. The goal is to get a minimal number of units in P and in Q so that the analysis can be done most efficiently.

List all the units that need to be considered in the analysis for each thread after you determine all the possible instruction combinations. Each unit should be a sequence of instruction labels without any separator and multiple units should be separated by a comma and should be given in sequence. For example, if (P1 and P2) can be considered together in the analysis and no more combinable instructions in P, then your answer for P should be “P1P2,P3,P4” (without the quotation marks of course).

P’s units to be considered during the analysis: **[P]**

Q’s units to be considered during the analysis: **[Q]**

Just in case this may help: In homework 1, we have

(Thread P)		(Thread Q)	
P1	load a	Q1	load 4
P2	add 1	Q2	store b
P3	store a	Q3	load a
P4	load b	Q4	add b
P5	add 2	Q5	store a
P6	store b		

And from the analysis in the answer key, the list of units for P should be expressed as: P1P2,P3,P4P5,P6 and for Q should be expressed as Q1Q2,Q3,Q4Q5.

Question 8

0 out of 4 points

6. Consider the following code for signaling and signal handling by parent and child processes, respectively. What are the possible outputs from this program?

```
int x, y;
void catch_ctlc (int sig_num)
 { x = 3;  y = 6;}

int main()
 {  signal(SIGINT, catch_ctlc);

    int pid = fork ();
    if (pid != 0)
        kill (pid, SIGINT);
    else { x = 1;  y = 2;
             printf ("(%d,%d)\n", x, y);
           }
    wait();
 }
```

List all possible outputs in the blank. Multiple possible outputs should be listed in ascending order (first sort by x and then sort by y) with no space or separator in between (e.g., (0,1)(2,0)(2,2)(3,4)). **[A]**

Question 9

0 out of 15 points

7. In the lecture, we have shown an example on how to use locks to synchronize concurrent threads A, B, and D. The example is repeated below.

```
lock(X); A; unlock(X); unlock(Y);
lock(X); B; unlock(X); unlock(Z);
lock(Y); lock(Z); D;
```

We can use integer semaphores to replace locks and achieve the same synchronization. In the following code, X, Y, Z are semaphores.

```
wait(X); A; signal(X); signal(Y);
wait(X); B; signal(X); signal(Z);
wait(Y); wait(Z); D;
```

(a) For the same synchronization, we must use 3 locks, but we can use only 2 semaphores. Rewrite the code above to use only two semaphores X and Y. You can cut and paste the same code from above to the corresponding blank below (line by line) and modify them so that only two semaphores are used. (Minimize your changes and do not change the code format.)

The modified code for thread A: **[A]**

The modified code for thread B: **[B]**

The modified code for thread C: **[C]**

(b) What should the initialization values for the semaphores be?

Initialization value for X: **[X]**

Initialization value for Y: **[Y]**

Question 10

18 out of 18 points

8. Consider the bakery problem, but we replace the bakery by a mock club restaurant. The restaurant has an entertainment center (EC) which has **N** seats. A customer can enjoy in the entertainment center while waiting for a table to become available (and order food at this time), if one is not available at the moment. Once a customer gets the table and goes to the table, the seat in the entertainment center becomes available. When the entertainment center is full, the new customer has to wait outside.

There are **K** dining tables in the dining hall and **M** waiters. If a dining table is available, the customer can go to the table and wait for the waiter to bring food. After the customer gets food, she/he eats and leaves after dining.

A waiter, when becomes available, can serve a customer at an occupied table (in the order of the table being occupied). If there is no occupied table that has not been served, then the waiter waits.

We use semaphores to synchronize the activities. The code for the waiter and the code for the customer are as follows, except that the customer code is only partially finished. In the code, we use 4 semaphores: EC, waiter, table-occupied, and table.

```
waiter:
    loop in working hours
      wait (table-occupied);
      serve by bringing food to the table;
      signal (waiter);
    endloop;

customer:
    wait (EC);
    order food and enjoy EC while waiting;
    wait (table);
    [A]
    food has been brought by waiter, go ahead to eat;
    [B]
```

Finish the missing statements in the two blank boxes in the customer function. Each box may contain 0 to 5 statements. Each statement is a wait or a signal on a semaphore. No counters or new semaphores should be used. If no statement is needed for a blank, you should put "**none**" in it (without the quotation marks). Each statement should end with a semicolon and your answer should not contain any space (e.g., wait(EC);wait(table);signal(waiter);signal(EC);).

Note: you can initialize the semaphores on your scratch paper to verify the correctness of your answers.

Question 11

6 out of 18 points

9. Consider the thread example program we have posted, in which each thread increments a global "counter" that is initialized to 0. (Limit is a large positive integer.)

```
function increment ()
```

```
{  for (j=0; j<Limit; j++)
     {  sem_wait (&mutex);
        counter++;
        sem_post (&mutex);
}    }
```

We create **6** threads, T1, T2, ..., T6, to run increment() on a **single core** computer. Also, we use various synchronization (sync) schemes listed below to replace the semaphore used in increment() given above.

A. Peterson's lock, B. Lock using test-set instructuion,
C. Disable interrupt, D. A turn variable,
E. Integer semaphore (i.e., not to change the original scheme).

In D, the "turn" variable is the same as that in the "take turn" lock solution we introduced in the lectures. If "turn" is set to Ti, then Ti can enter the critical section. Initially, "turn" is set to T1. After exiting the critical section, Ti will set "turn" to Tj, j = (i+1)%6. For example, the pseudo code for T1 is as follows.

```
while turn != T1 do nothing;
< critical section >
turn := T2;
```

(a) Answer the two questions below. List all the schemes that satisfy the question in **alphabetical order** with no space (For example, ACD). If there is no scheme satisfying the question, then put "**none**" (without the quotation marks of course).

Which sync scheme(s) can fully avoid busy waiting if used in increment()? **[A1]**

Which sync scheme(s) may cause deadlock if used in increment()? **[A2]**

b) Rank B, C, E in terms of the **busy waiting time** overheads (probabilistically) they may induce if used in increment(). Your answer should be a permutation of BCE and should contain no space. If your answer is BEC, it means B has the lowest overhead and C has the highest. **[B]**

Note: when answering the questions, carefully consider how each scheme is implemented.

Monday, April 26, 2021 10:17:15 PM CDT

← **OK**