CSC 246 001 Midterm Exam

Qizheng Jin

TOTAL POINTS

63 / 100

QUESTION 1

Q1 12 pts

1.1 **0 / 2**

- 0 pts Correct

✓ **- 2 pts** *Wrong (C)*

1.2 **2 / 2**

✓ **- 0 pts** *Correct*

- 2 pts Wrong (B)

1.3 **2 / 2**

✓ **- 0 pts** *Correct*

- 2 pts Wrong (D)

1.4 **0 / 2**

- 0 pts Correct

✓ **- 2 pts** *Wrong. (A)*

1.5 **0 / 2**

- 0 pts Correct

✓ **- 2 pts** *Utilization is just a percentage.*

- 2 pts Turnaround time (TAT) is the amount of time taken to complete a process or fulfill a request.

- 2 pts Response time is the total amount of time it takes to respond to a request for service.

1.6 **0 / 2**

✓ **- 2 pts** *If we always give high priority processes long quantum, this process will take most of cpu time. Which is not a feedback scheduler.*

- 2 pts For old processes, the scheduler will give more quantum than it already has, but it can be short.

- 0 pts Click here to replace this description.

- 2 pts New processes can have a short quantum.

QUESTION 2

Q2 20 pts

2.1 **3.5 / 5**

- 0 pts Correct

- 2.5 pts Messages are sent and received using system calls and waiting for kernel intervention can slow
down the performance of message passing.

✓ **- 2.5 pts** *Shared memory only requires an initial system call for establishing the shared memory segment.*
Once the segment has been created, accessing the shared memory is performed in user mode and requires no kernel intervention.

✓ **+ 1 pts** *attempt credit*

2.2 **5 / 5**

✓ **- 0 pts** *Correct*

- 1.25 pts 1) running

- 1.25 pts 2) Ready

- 1.25 pts 3) waiting

- 1.25 pts 4) running

2.3 **4.5 / 5**

- 0 pts Correct

✓ **- 2.5 pts** *(1) If the quantum length is too long, most processes are going to finish before they hit the end of their quantum, they'll finish their burst. So, it's behaving more like first come first served. It may cause poor response to short interactive requests.*

✓ **+ 1 pts** *attempt credit*

- 2.5 pts (2) If the quantum length is too short, there are a lot more context switches, and since context switch has overhead, more and more time is wasted to context switching, which may lower the CPU efficiency.

✓ **+ 1 pts** *attempt credit*

2.4 **2 / 5**

- 0 pts Correct

✓ **- 2 pts** *This doesn't guarantee progress.*

✓ **+ 1 pts** *attempt credit*

✓ **- 3 pts** *If one thread wants to get into the critical section, it's dependent on some other thread to let it in, so this*
fails to satisfy the progress that each thread can enter its critical section without needing help from some
other thread.

✓ **+ 1 pts** *attempt credit*

QUESTION 3

Q3 20 pts

3.1 **2 / 5**

- 0 pts Click here to replace this description.

✓ **- 3 pts** *exec() will load the code and data of the application specified by the command and override the*
current process. In other words, the shell does not exist after exec() is called. So after the application finishes, it will simply terminate instead of waiting for the next command from the user.

- 5 pts exec() will load the code and data of the application specified by the command and override the
current process. In other words, the shell does not exist after exec() is called. So after the application finishes, it will simply terminate instead of waiting for the next command from the user.

- 0.5 pts Will exit instead of return. Exec system call won't return anything.

- 1 pts Explain it. Why it won't loop.

- 1 pts The process will not be stacked, it will just exit for successfully called the exec.

3.2 **5 / 10**

- 0 pts Correct

- 7 pts while (true) {
get_command(command, parameter)
if (fork()==0) {
exec(command, parameters)
} else {

wait(&status);

}

}

Your code did not work.

- 5 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

} else {

wait(&status);

}

}

Your code did not wait. So the main process will fork() forever and nver stop.

- 7 pts Code missing. You need to fork and make sure the parent process is waiting for the execution of child process.

✓ **- 5 pts** *while (true) {*

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

} else {

wait(&status);

}

}

Your code did not wait and just exit. for parent process the pid will be greater than 0.

- 10 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

} else {

wait(&status);

}

}

- 5 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

} else {

wait(&status);

}

}

Your code doesn't work because you used command before get command.

3.3 **2 / 5**

- 0 pts Correct

- 1 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

}

}

You put the get_command in child process, but what if we are reading from stdin?

- 3 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

}

}

Why you wait?

- 5 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

}

}

✓ **- 3 pts** *while (true) {*

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

}

}

The parent process will execute the command too.

- 3 pts while (true) {

get_command(command, parameter)

if (fork()==0) {

exec(command, parameters)

}

}

Wrong answer.

QUESTION 4

Q4 20 pts

4.1 **5 / 5**

✓ **- 0 pts** *Correct*

- 4 pts | A | B | C | D |

0----8----12----15----16

- 1 pts Average waiting time: (0 + 5 + 7 + 6) / 4 = 4.5

+ 1 pts attempt credit

4.2 **1 / 5**

- 0 pts Correct

✓ **- 4 pts** *graph wrong*

✓ **- 1 pts** *waiting time wrong (3.5)*

✓ **+ 1 pts** *attempt credit*

4.3 **5 / 5**

✓ **- 0 pts** *Correct*

- 4 pts | A | B | A | D | A | C |

0-----3----7-------9----10----13---16

- 1 pts Average waiting time: (5 + 0 + 8 + 0) / 4 = 3.25

- 4 pts | A | B | A | D | A | C |

0-----3----7-------9----10----13---16

Average waiting time: (5 + 0 + 8 + 0) / 4 = 3.25

4.4 **1 / 5**

- 0 pts Correct

- 4 pts | A | B | C | A | B | D |

0-----6----9------12----14---15---16

- 1 pts Average waiting time: (6 + 8 + 4 + 6) / 4 = 6

- 1 pts | A | B | C | A | B | D |

0-----6----9------12----14---15---16

Average waiting time: (6 + 8 + 4 + 6) / 4 = 6

You misunderstand this assumption

✓ **- 4 pts** | *A* | *B* | *C* | *A* | *B* | *D* |

0-----6----9------12----14---15---16

Average waiting time: (6 + 8 + 4 + 6) / 4 = 6

- 1 pts (8+6+4+4)/4 = 5.5 For your assumption the result should be like that.

QUESTION 5

Q5 28 pts

5.1 **3 / 6**

- 0 pts Correct

- 1 pts 12 possible orders in total

- 2 pts 12 possible orders in total

✓ **- 3 pts** *12 possible orders in total*

- 4 pts 12 possible orders in total

5.2 **0 / 2**

- 0 pts Correct. 1/12

✓ **- 2 pts** *Wrong*

5.3 **5 / 5**

✓ **- 0 pts** *Correct*

- 3 pts Both a = 1 and b = 1 cannot be guaranteed

- 1 pts Lack of the initial values of semaphores

- 1 pts a = 1 cannot be guaranteed

5.4 **15 / 15**

✓ **- 0 pts** *Correct*

- 5 pts 1) should be incorrect

- 5 pts 2) should be correct

- 5 pts 3) should be correct

- 3 pts 1) lack of reasons

- 3 pts 1)'s reason is not correct

CSC 246: Concepts and Facilities of Operating Systems for Computer Scientists (Section 001)

Midterm Exam – Spring 2023

Name Qizheng Jin Unity ID qjin3

Please read the following instructions carefully:

- This exam is closed book and closed notes.
- Your work must be individual. Cheating will be punished according to university academic integrity code.

About this exam:

- The total point value of the exam is 100.
- Problem 1 is a list of multiple-choice questions.
- Problem 2 is a list of short answer questions.
- Problems 3-5 are problems require longer answers.

Problem	Possible Points
1. Multiple Choice	12
2. Short Answer	20
3. Processes	20
4. Scheduling	20
5. Threads and Synchronization	28
Total	100

1. Multiple Choice Questions (12 pts, 2 pts each)

Mark the **best** answer for each of the following questions.

B a) A microkernel is a kernel _____.

A. containing many components that are optimized to reduce resident memory size
B. that is compiled to produce the smallest size possible when loading the operating system into main memory
C. that is stripped of all nonessential components
D. has very limited or no structure at all

B b) Which of the following would you expect to be stored in a Process Control Block for some process, P?

A. All the executable code for P
B. A copy of all the CPU Registers for P
C. A copy of the stack for P
D. A copy of the text section for P

D c) _____ saves the state of the currently running process and restores the state of the next process to run.

A. The CPU scheduler
B. Swapping
C. A trap instruction
D. A context switch

D d) Which is NOT shared by threads in a process?

A. Program Counter
B. Heap
C. Opened files
D. Code

A e) ____ is the number of processes that are completed per time unit.

A. CPU utilization
B. Response time
C. Throughput
D. Turnaround time

A f) A multilevel feedback queue scheduler generally assigns a long quantum to _____

A. high priority processes.
B. low priority processes.
C. new processes.
D. old processes.

2. Short Answer (20 pts, 5 pts each)

Provide brief answers to the following questions.

a) For inter-process communication, explain why shared memory is typically faster than message passing during the communication.

It can directly gain the information from the memory, but the communication needs to convey the information through a pipe between child and parent process, which takes more time.

b) Processes (or threads) can be in one of three states: Running, Ready, or Waiting. For each of the following examples, write down which state the process (or thread) is in:

1) Spin-waiting for a variable to become zero.

Running

2) Having just completed an I/O, waiting to get scheduled again on the CPU.

Ready

3) Blocking on a condition variable waiting for some other thread to signal it.

Waiting

4) Scanning through the buffer cache within the kernel in response to read().

Running

c) For Round Robin scheduling, it's important to choose a proper quantum length. Explain (1) what is the issue if we make the quantum length too long, and (2) what is the issue if we make the quantum length too short?

If it is too long, ~~there will~~ it will make no difference with other kind of scheduling because the Ready Queue won't work effectively and processes can't be arranged fair enough.

If it is too short, the processes will change very quickly, and it takes more resources to switch between the processes, which is a kind of waste.

d) Consider the following flawed solution to the critical section problem.

```
int turn = 0;

ThreadA() {
  for ( int i = 0; i < 999;
        i++ ) {
    while ( turn == 1 ) {
    }
    critical Section;
    turn = 1;
    remainder Section;
  }
}
```

```
ThreadB() {
  for ( int i = 0; i < 1000;
        i++ ) {
    while ( turn == 0 ) {
    }
    critical Section;
    turn = 0;
    remainder Section;
  }
}
```

There are three requirements that a correct solution to the critical section problem must satisfy. Which of these isn't satisfied by this code? Briefly explain.

After some loops, it will cause deadlock. When turn=1, it will stuck in the while loop in Thread A, because it is

It should give out the signal that a thread has finished its critical section, like Peterson algorithm.

3. Processes (20 pts)

Considering the following C code. Pretend this is an implementation of a UNIX shell. Like any shell, the job of this program is to repeatedly read commands from the user and execute them. Each user command gives the name of an application. The shell is supposed to run the application, wait for it to finish and then read the next command from the user.

```
while (true) {
      get_command(command, parameters);
      exec(command, parameters); //this is the exec() system call
}
```

a) Using the above code, what problem will happen after a user issues a command? Explain. (5 pts)

If the command causes a dead loop, it will stuck in the exec() and can't come out, and the shell won't stop it automatically.

b) Using a few lines of code change/addition (hint: using fork()), you can fix the problem. Show how this can be fixed (10 pts)

```
while true {        pid-t pid;
    get-command (command, parameters);
    fork( );
    exec (command, parameters);
    if ( pid!=0) { printf " Can't execute the command ");
                   exit (0);
    }.          }.
```

If parent process can't get signal from child process, then terminate the shell

c) A UNIX shell also supports background execution of a command. That is, once the application is started, the shell is ready to receive the next command, without waiting for the finish of the previous command. Modify your code for problem b) to support background execution. (5 pts)

```
while true {        pid-t pid
    get-command (command, parameters);
    fork( );
    exec (command, parameters);
    while (exec( )!=0)) get-command (command, parameters);.
    if (pid!=0) { printf " Can't execute the command ");
                  exit (0);
                }.
}.
```

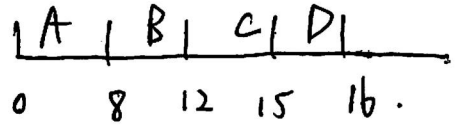
4. Scheduling (20 pts)

The table below shows four processes that are to be scheduled on a single CPU. Each process has an arrival time before which it cannot run, a CPU burst time - the number of time units it needs to run, and a priority number (assume smaller number represents higher priority). For SJF (shortest job first) scheduling, assume that the scheduler knows the length of each CPU burst. Break any ties by choosing the process that appears earlier in the table. For each scheduling, draw the Gantt chart and calculate the average waiting time of the four processes.

Process	Arrive	Burst	Priority
A	0	8	5
B	3	4	3
C	5	3	6
D	9	1	1

3<5.
3<6.

a) First come first served scheduling: (5 pts)

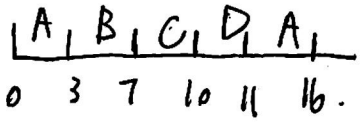


Turnaround: (8+9+10+7)/4 = 8.5

Waiting: (0+5+7+6)/4 = 4.5

b) Shortest job first (SJF) scheduling: (5 pts)

5

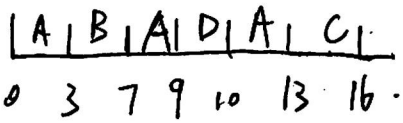


Turnaround: (16+4+5+2)/4

Waiting: (8+0+2+1)/4 = 11/4

c) Preemptive priority scheduling: (5 pts)

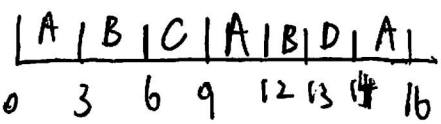
5-2=3.



Turnaround: (13+4+11+1)/4

Waiting: (5+0+8+0)/4 = 13/4

d) Round-robin scheduling (quantum = 3): (5 pts)



Turnaround: (16+10+4+5)/4

Waiting (8+6+1+4)/4 = 19/4

RQ: ABCABD.A

5. Threads and Synchronization (28 pts)

a) Consider the following three threads. Pretend that the individual assignment statement will each execute atomically.

```
1  threadP(){
2    a = 1;
3    b = 2;
4  }
5
6  threadQ(){
7    b = 1;
8  }
9
10 threadT(){
11   a = 2;
12 }
```

1) List all the possible interleavings of the execution of statements in these three threads. For example, "a = 1; b = 2; b = 1; a = 2" is one possible execution order. (6 pts)

① a=1 b=2 b=1 a=2. ④ a=1 b=2 a=2 b=1

② b=1 a=1 b=2 a=2 ⑤ b=1 a=2 a=1 b=2.

③ a=2 a=1 b=2 b=1 ⑥ a=2 b=1 a=1 b=2

2) If all interleavings are as likely to occur, what is the probability to have a = 1 and b = 1 after all threads complete execution? (2 pts)

① X ② X ③ √ ④ X ⑤ X ⑥ X ∴ $\frac{1}{6}$

3) Revise the original code using semaphores to guarantee after all threads complete execution, it always has a = 1 and b = 1. Show clearly how many semaphores are used, what are their initial values, and where you insert the acquire () and release () operations (using line numbers). (5 pts)

T → P → Q

sem TDone = 0 sem PDone = 0.

~~Thread X {~~

```
thread P() {
  acquire(TDone);
  a=1;
  b=2;
  release(PDone);
}.

thread Q() {
  acquire(TDone);
  acquire(PDone);
  b=1;
}

thread T() {
  a=2
  release(TDone);
}
```

b) Consider two threads A and B that perform two operations each. Let the operations of thread A be A1 and A2; let the operations of thread B be B1 and B2. We require that threads A and B each perform their first operation before either can proceed to the second operation. That is, we require that A1 be run before B2 and B1 before A2. Consider the following solutions based on semaphores for this problem (the code run by threads A and B is shown in two columns next to each other). For each solution, explain whether the solution is correct or not. If it is incorrect, you must also point out why the solution is incorrect. (15 pts, 5pts each)

$A_1 \rightarrow B_2$. 对. $B_1 \rightarrow A_2$ 对.

1)
```
sem A1Done = 0;
sem B1Done = 0;

ThreadA(){                          ThreadB(){
A1;                                 B1;
acquire( B1Done );                  acquire( A1Done );
release( A1Done );                  release( B1Done );
A2;                                 B2;
}                                   }
```

若 A_1, B_1, B_1 A_1 B_2 A_2

Incorrect. In Thread B, it will release B_1 after get signal from A1Done, but ~~if wants to~~ it will only release (A1Done) after it get signal from B1Done, which causes a dead lock.

2)
```
sem A1Done = 0;
sem B1Done = 0;

ThreadA(){                          ThreadB(){
A1;                                 B1;
acquire( B1Done );                  release( B1Done );
release( A1Done );                  acquire( A1Done );
A2;                                 B2;
}                                   }
```

A_1 B_1 A_2/B_2 B_2/A_2.

correct

3)
```
sem A1Done = 0;
sem B1Done = 0;

ThreadA(){                          ThreadB(){
A1;                                 B1;
release( A1Done );                  release( B1Done );
acquire( B1Done );                  acquire( A1Done );
A2;                                 B2;
}                                   }
```

A_1 B_1 B_2/A_2.

correct

This is one scratch page.