**CSE3502, Operating Systems**

**MidtermFull Name: \_\_\_Cora Meador\_\_\_\_ KSU ID: \_\_000811057\_\_\_\_\_\_ Score: \_\_\_\_\_\_\_\_**

Problem 1 (42 points)

Please give **brief and precise** answers to the following questions.

1. What is a process? What are the differences of threads and processes?

*A process is an instance of a program that is using resources like CPU and memory. Processes can be created with the fork() system call, which creates a full copy of the process with its own memory. A thread is similar to a process, but there can be many threads in a single process. It is created with the clone() system call, where two processes operate on the same code while sharing a common memory space. In linux, processes and threads are called “tasks” and are the same object.*

2. In your own words, give advantages and disadvantages of implementing threads at user-level, instead of kernel-level.

*Threads implemented at the kernel level are scheduled by the operating system, and they can be given a slice of time to run by themselves on the cpu, which can allow them to run in parallel with other processes. However, there is some overhead with the way cpu caches memory, and system calls are expensive. With user-level threading, it can be cheaper to allocate memory. Also, schedule time can be performed by the application, which may be more suited for the specific purpose. However, the thread cannot be run in parallel with other threads in the application space.*

3. Compare the advantages and disadvantages of implementing a centralized run queue and a distributed run queue for a multiprocessor system?

*With a distributed run queue, memory can be cached and more easily accessed. There is also a shorter distance and faster access rate from each CPU to the queue closest to it. However, there can be an unequal load from one CPU to another, and it can take a longer time to “steal” from a fuller queue. A centralized queue does not need to make the sizes of each queue equal, however there can be a bottleneck when each CPU gets the next item. There is also overhead with copying in the new cache.*

4. What is lock and why we need lock in the OS?

*The lock is used to secure a “critical region” of code, where is is possible for two threads to access the same data simultaneously. When one thread caches the data for processing, it puts a lock on the location, which prevents the other threads from changing the data until the calculation is done. This makes sure that the calculation is always correct and not susceptible to random interactions.*

5. List at least three ways of mutex exclusion in the OS.

*Sleep and Wakeup*

*Recursive*

*Spinlock*

*Mutex with Conditions*

6. Compare busy-waiting and sleep-and-wake approaches for mutual exclusion (advantages and disadvantages).

***Spinlock:*** *The thread requests the locked information repeatedly until it is unlocked. This consumes CPU, but it has a faster pickup time. It is best when the time expected to wait is short*

***Sleep and Wakeup:*** *When a critical region is locked, threads are put to sleep for a designated time until the thread in the critical region is done. This does not consume as much cpu, however there is some overhead because sleep() is a system call. This is best used when there is a longer expected wait time.*

7. Explain the difference between kernel mode and user mode. Why we need two modes in the OS. How could we interact between user space and kernel space?

*Kernel mode operates with a high level of privilege and can allow access to the file system and process scheduling, while user space has the permission to run programs and manipulate most other files. User space is a subset of kernel space. An API is used to make a system call in user space, for example in the C standard library.*

Problem 2 (30 points)

(2.1) For the next two questions, assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

main() {

int a = 0;

int rc = fork();

a++;

if (rc == 0) {

rc = fork();

a++;

} else {

a++;

}

printf(“Hello!\n”);

printf(“a is %d\n”, a);

}

Assuming fork() never fails, how many times will the message “Hello!\n” be displayed? And explain your answer.

a) 2

b) 3

c) 4

d) 6

**e) None of the above**

**There are 2^n times the message is printed, and n = the number of times fork is called. The program only forks if the child is calling the fork, and because there is a always at least one child with value 0, the program will fork an infinite number of times.**

What will be the largest value of “a” displayed by the program? And explain your answer.

a) Due to race conditions, “a” may have different values on different runs of the program.

b) 2

c) 3

d) 5

**e) None of the above**

**Becuase the program forks an infinite number of times, there will be no hightst value of a, although a race condition does exist whenever a is incremented.**

(2.2) For the next two questions, assume the following code is compiled and run on a modern Linux machine (assume any irrelevant details have been omitted):

int balance = 0;

void \*mythread(void \*arg) {

int i;

for (i = 0; i < 200; i++) {

balance++;

}

printf(“Balance is %d\n”, balance);

return NULL;

}

int main(int argc, char \*argv[])

pthread\_t p1, p2, p3;

pthread\_create(&p1, NULL, mythread, “A”);

pthread\_join(p1, NULL);

pthread\_create(&p2, NULL, mythread, “B”);

pthread\_join(p2, NULL);

pthread\_create(&p3, NULL, mythread, “C”);

pthread\_join(p3, NULL);

printf(“Final Balance is %d\n”, balance);

}

Assuming none of the system calls fail, when thread p1 prints “Balance is %d\n”, what will p1 say is the value of balance? And explain your answer.

a) Due to race conditions, “balance” may have different values on different runs of the program.

**b) 200**

c) 400

d) 600

e) None of the above

Pthread\_join waits for the other theads to finish before combining the results, so there is no race condition.

Assuming none of the system calls fail, when the main parent thread prints “Final Balance is %d\n”, what will the parent thread say is the value of balance? And explain your answer.

a) Due to race conditions, “balance” may have different values on different runs of the program.

b) 200

c) 400

**d) 600**

e) None of the above

Pthread\_join waits for the other theads to finish before combining the results, so there is no race condition. They all share the same variable, but they do not access it at the same time

Problem 3 (18 points)

Assume that the following processes are to be executed on a uniprocessor system.

Process Arrival Time CPU burst

P1 0 5

P2 1 5

P3 5 3

P4 6 2

1) Show the scheduling order (fill Process ID: 1/2/… in the table) for these processes under First-In-First-Out (FIFO), Shortest-Job First (SJF), and Round-Robin (RR) with a quantum = 1 time unit. Assume that the context switch overhead is 0 and new processes are added to the head of the queue except for FIFO.

|  |  |  |  |
| --- | --- | --- | --- |
| Time | FIFO | SJF | RR |
| 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 2 |
| 2 | 1 | 1 | 1 |
| 3 | 1 | 1 | 2 |
| 4 | 1 | 1 | 1 |
| 5 | 2 | 3 | 3 |
| 6 | 2 | 3 | 4 |
| 7 | 2 | 3 | 2 |
| 8 | 2 | 4 | 1 |
| 9 | 2 | 4 | 3 |
| 10 | 3 | 2 | 4 |
| 11 | 3 | 2 | 2 |
| 12 | 3 | 2 | 1 |
| 13 | 4 | 2 | 3 |
| 14 | 4 | 2 | 2 |

2) What are the average response time and turnaround time (TRT) under First-In-First-Out (FIFO), Shortest-Job First (SJF), and Round-Robin (RR)?

|  |  |  |
| --- | --- | --- |
| Scheduler | | Avg |
| FIFO | response time | 4 |
| TRT | 6.75 |
| SJF | response time | 2.75 |
| TRT | 5.5 |
| RR | response time | 0 |
| TRT | 9.25 |

Problem 4. (10 points)

Your project partner started to write the code for implementing multiple threads with lock. Your job is to check the correctness for the code.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5 | Thread 1:  pthread\_mutex\_lock(&m1);  pthread\_mutex\_lock(&m2);  do\_something();  pthread\_mutex\_unlock(&m2);  pthread\_mutex\_unlock(&m1); | Thread 2:  pthread\_mutex\_lock(&m2);  pthread\_mutex\_lock(&m1);  do\_something();  pthread\_mutex\_unlock(&m1);  pthread\_mutex\_unlock(&m2); |

1) If you execute the program with one thread 1 and one thread 2, will the deadlock happen in the program? Why and which line of code generate the deadlock?

*Yes, because m1 is locked by thread 1, so that when thread 2 attempts to access the variable in line 4, it does not have ownership of the lock. Likewise, thread 2 locks m2 and owns the lock, so thread 1 cannot proceed with unlocking m1 for thread 1 to use on line 4. This is like the Dining Philosophers problem.*

2) If you execute the program with two thread 1, will the deadlock happen in the program? Why and which line of code generate the deadlock?

The deadlock would not occur, because the first thread would be able to finish using the variables, unlock them, and pass them on to the other copy of the process. This is because they both attempt to lock the same variables at the same time.