

FINAL EXAM

- This test contains 9 questions worth a total of 104 points
 - 1-4 6 points each
 - 5 8 points each
 - 5 12 points each
 - 7-9 20 points
- You have 110 minutes to complete this exam.
- You may **not** use your text, or any other reference material.
- Do not turn this page or turn over the exam until instructed to do so.

Hints

- Don't spend too much time on short answer questions. If you don't know the answer right away, move on and come back.
- There may be some questions that will require more time to think and organize your thoughts. If you find yourself stuck on one of these questions, save it until you have answered all the questions you can answer quickly. If you spend too much time on one question you may get frustrated and that will impair your ability to answer the other questions.
- Don't turn your test in early. If you have time left over, use it to review your answers. Students who turn tests in early often make trivial mistakes that they would catch if they reviewed their answers.
- Students that get the highest grades usually go over their answers several times.
- The space below a question **does not** indicate how long of an answer I am expecting. Sometimes I want to start the next question on a new page.
- Double check that you understand the question before you answer it. It is very likely you will not get any points if you don't understand the question. **If you don't understand the question, ask me.**

- 1) (6 points) What type of process scheduling can lead to race conditions? What should the program do to prevent race conditions?

Preemptive scheduling: processes can be interleaved in any way which can lead to race conditions. Programs must take this into account and protect critical sections (e.g. by using a semaphore).

- 2) (6 points) Explain how the shell uses the environment variable PATH.

PATH is a string that contains a set of directories separated by colons.

When the user types a command at the shell prompt, if the command is the full path of a file (starts with a /) the shell executes the file (assuming it exists and that it is executable). Otherwise, the shell looks for the given command in each directory in the PATH.

- 3) (6 points) ctime(t) converts calendar time into a string of the form: "Wed Jun 30 21:49:08 1993\n"

```
char *ctime(const time_t *calendar_time);
```

This function is not thread safe. What does this function do that makes it unsafe? How could the function be rewritten to make it thread safe?

Since we know the function is not thread safe it must use the same buffer each time it is called. Thus if two threads call the function at the exact same time the buffer may become corrupt (both threads writing to it at the same time).

We could make the function thread safe by passing in a buffer into which it would put the string form of the given time. If the buffer was passed into the function then when multiple threads call the function they would not interfere with each other.

- 4) (6 points) On machines with multiple processors race conditions usually cause more problems than on single processor machines. Explain why.

On single processor machines processes have to be interleaved exactly right for a race condition to influence the program outcome. Since schedulers usually let a process complete a CPU-burst, it is very unlikely that processes will be interleaved in such a way that a race condition will influence the outcome.

On multiprocessor machines more than one process is running at a time, so for a race condition to alter the output, two processes don't have to be interleaved they just have to be executing at the same time. Thus it is much more likely that a race condition will influence the program outcome.

5) (8 points) Consider the code:

initialization:	thread1	thread2	thread 3
int y = 42;	y = y + 1;	y = y + 2;	y = y + 3;

Give all the possible values for y after running this code. Give at least one explanation for each value.

- y = 43 thread1 starts but is interrupted after getting the value of y but before setting the value. Then thread2 and thread3 run until completion. Finally thread 1 completes.
- y = 44 thread2 starts but is interrupted after getting the value of y but before setting the value. Then thread1 and thread3 run until completion. Finally thread 2 completes.
- y = 45 thread3 starts but is interrupted after getting the value of y but before setting the value. Then thread1 and thread2 run until completion. Finally thread 3 completes.
- y = 46 thread1 executes, then thread3 starts but is interrupted after getting the value of y but before setting the value. Then thread2 runs until completion. Finally thread 3 completes.
- y = 47 thread2 executes, then thread3 starts but is interrupted after getting the value of y but before setting the value. Then thread1 runs until completion. Finally thread 3 completes.
- y = 48 The three threads run one after the other (not interleaved) in any order.

6) (10 points) Assume the following function is atomic. Use it to solve the critical section problem. Solutions to the critical section problem must satisfy three requirements. Which does your solution satisfy and which does it not satisfy? Explain your answer.

```
// C version
void swap(bool *a, bool *b)
{
    bool temp = *a;
    *a = *b;
    *b = temp;
}
```

```
// C++ version
void swap(bool &a, bool &b)
{
    bool temp = a;
    a = b;
    b = temp;
}
```

P_i

```
bool key = true; // shared by all processes
```

```
while (true)
{
```

```
    bool my_key = false;
    while (my_key == false)
        swap(key, my_key);
```

```
    critical section
```

```
    swap(key, my_key);
```

```
    remainder section
```

```
}
```

1. Mutual exclusion is satisfied: only one process can *grab* the “false” value
2. Progress is satisfied: when one or more processes attempt to enter the critical section, one of them always succeeds.
3. Bounded wait is not satisfied: bad luck could indefinitely starve a process. Specifically, one process' time slice could always be when another process was executing the critical section.

- 7) (20 points) Write pseudocode for a mailbox class that allows two threads to communicate by sending and receiving messages to/from the mailbox. The constructor should take the size of the buffer as its only argument. The send should be non-blocking if there is room in the buffer, blocking if there is not room. The receive should be blocking if the buffer is empty, non-blocking if the buffer is not empty. Assume that many threads use a single instantiation of the Mailbox class. Use conditional variables to provide the synchronization.

```
// this will print out "hello from thread1" and "hello from thread2" in a random order
Mailbox mb(10); // instantiate the Mailbox with a buffer size of 10

thread1: mb.send("hello from thread1");

thread2: mb.send("hello from thread2");

thread3: char *buf1 = mb.receive();
         char *buf2 = mb.receive();
         cout << buf1 << endl;
         cout << buf2 << endl;
```

8) (20 points) The function `char *remote_system(char *cmd, char *hostname, int port)` executes the given command on the given server and returns what the command wrote to standard output. Outline how this function could be implemented using sockets. Include an outline of both the client (the implementation of the function) and the server (which will be running on the given host).

The first step is to develop a communication protocol.

client → server: null-terminated string that contains the command to be executed

server → client: size of the reply in set number of bytes (say 4) + bytes in the reply

`remote_system()` on the client

- create a socket and connect to the given host/port
- send the command to the client
- block on receiving from the socket receiving with a buffer of 4 bytes
- once the size is read, dynamically allocate a buffer to hold the result
- continue receiving from the socket until all the bytes have been received
- return the address of the dynamically allocated buffer

`remote_system()` on the server

- fork a child process to handle this request
- modify the command so that its output is redirected to a file
- using `stat`, find the size of the file
- send the size to the client
- read the file and send the bytes (the buffer size doesn't matter much) to the client
- close the socket

9) (20 points) Sea World has a new program where kids can swim in the shark tank. In order to avoid kids getting eaten by the sharks, kids and sharks cannot be in the tank at the same time. Multiple kids can get into the tank as long as there are no sharks in the tank. Multiple sharks can get into the tank as long as there are no kids in the tank. Write pseudocode to implement a solution to this problem, specifically write the code that sharks will execute and write the code that kids will execute. Use semaphores for synchronization. Assume each kid and each shark is represented by its own thread.

This solution is very similar to the readers-writers solution

```
// global variables
sem tank_mutex = 1;
sem kid_mutex = 1;
int kid_count = 0;
sem shark_mutex = 1;
int shark_count = 0;
```

```
kid()
{
    while (1)
    {
        wait(kid_mutex);
        kid_count++;
        if (kid_count == 1)
            wait(tank);
        signal(kid_mutex);
        swim();
        wait(kid_mutex);
        kid_count--;
        if (kid_count == 0)
            signal(tank_mutex);
        signal(kid_mutex);
        eat();
    }
}
```

```
shark()
{
    while (1)
    {
        wait(shark_mutex);
        shark_count++;
        if (shark_count == 1)
            wait(tank);
        signal(shark_mutex);
        swim();
        wait(shark_mutex);
        shark_count--;
        if (shark_count == 0)
            signal(tank_mutex);
        signal(shark_mutex);
        eat();
    }
}
```