

MIDTERM EXAM

- This test contains 8 questions with a total of 65 points
 - 1-5 5 points each
 - 6-7 10 points each
 - 8 20 points
- You have 75 minutes to complete this midterm.
- 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) (5 points) From a programming perspective, what is the main ramification of 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.

2) (5 points) Assuming that the following code compiles without error, tell me everything you can about the variable ptr.

```
void *f(int a, double b);  
...  
ptr = f;
```

ptr is a pointer to a function that returns a void * and takes the one int and one double argument. It would be declared as follows:

```
void * (*ptr)(int, double);
```

3) (5 points) Consider the code:

initialization:	thread1	thread2
int y = 0;	r1 = x;	r2 = y;
int x = 0;	y = 1;	x = 2;

Give all the possible value pairs for r1 and r2 after running this code. Explain each one.

(r1, r2) = (0,1) thread1 completes before thread2 runs
(r1, r2) = (2,0) thread2 completes before thread1 runs
(r1, r2) = (0,0) thread1 is interrupted after r1=x, then thread2 runs, then thread1 completes
-or-
thread2 is interrupted after r2 = y, then thread1 runs, then thread2 completes

- 4) (5 points) When a program is running what causes a *stack overflow* error? Include an explanation of the stack that is overflowing. Giving an example and drawing a picture may help explain it.

When a function is called, information about the function is placed into a structure called an activation record. The run time stack is a stack of activation records. When a function is called its activation record is pushed onto the run time stack. When the function terminates its activation record is popped off of the run time stack. If enough functions are called then the memory allocated for the stack is used up. This error is called stack overflow. The following program always causes stack overflow.

```
void f()
{
    f();
}
```

- 5) (5 points) Explain how to implement a function that takes a variable number of arguments. Include a description of how to determine the number of arguments. Mention how argument type can be handled. You don't need to remember the exact syntax, but I do expect you to touch on all the critical components.

Declare the function with at least one parameter followed by “...”

```
void f(int i, ...);
```

Inside the function call the `va_start` macro to initiate the parsing of the arguments. Then call the `va_arg` macro to extract each argument. The type of the argument must be passed to `va_arg`.

You can either terminate the arguments with a known value (such as `NULL`) or you can pass the number of arguments in a different argument (usually the first).

You must know the type of each argument in order to extract it. The types of the arguments can be predetermined (e.g. all `char *`) or another argument can be used to hold the types of the variable arguments.

As an example, consider `printf()`. The first argument contains a format string that indicates the number of arguments and the type of each argument:

```
printf(format_string, ...);
```

6) (10 points) The Linux command `wc` counts the letters, words, and lines in a file. Write a program that prompts the user for a filename and uses the `Spawn` class to create a process that runs `wc`. Use `Spawn::getchar()` to read the output of `wc` and then print it to standard output. Assume you have a `Spawn` class **without** `putchar()` (the extra credit).

```
int main()
{
    string filename;
    cout << "enter a filename: ";
    cin >> filename;

    Spawn wc("wc", filename.c_str(), 0);

    char c;
    while ((c = wc.getchar()) != EOF)
        cout << c;
}
```

7) (10 points) Remote procedure call (RPC) is a paradigm where a function call on one computer is executed on another computer. For example, consider the following function:

```
void calculate_prime_factor(string hostname, double number, double &factor1, double &factor2)
{
    // perform this calculation on a different computer
}
```

When it is called, the actual calculation is done on the computer named hostname. Outline how RPC can be implemented using the client-server paradigm and sockets. Hint: start by thinking about how the function is called on the client machine, then think about how the request is handled on the server machine. Make sure you include a description of the messages passed between the client and server.

The basic idea is that the arguments and an indication of which function is to be executed needs to be sent from the client to the server. On the server the function will be executed and the results will be sent back to the client. An easy interface for RPC is to provide a function-stub (like above) for each function that can be executed remotely. The code to communicate with the server can be placed in this function:

Client:

```
void calculate_prime_factor(string hostname, double number, double &factor1, double
&factor2)
{
    create a socket and connect to hostname
    send a message to client with "calculate_prime_factor" and number
    wait for the client to send back a message that will contain the two factors
    extract the results from this message and set factor1 and factor2
}
```

Server:

```
create a socket and wait for a client
once connected to a client, fork a new process (or thread) to perform the function
receive the message from the client, extract the function name and argument(s)
if (function_name == calculate_prime_factor)
    call calculate_prime_factor(...)
    send results back to client
else if (function_name == ...)
...
else error (unknown function)
```

- 8) (20 points) When customers walk into a bakery they take a number and wait to be served. When one of the bakery workers becomes available he serves the next customer in line. Using pthreads and semaphores write pseudo-code to model this interaction. Assume that there are N bakery workers and an unknown number of customers. Make sure your model includes a means for the worker and customer to communicate (e.g. a buffer). Your solution must allow N customers to be served at the same time (concurrently with other customers). Your solution does not have to implement taking a number but the customers must be served in a first come first served order. Unlike a real bakery the customer and worker will exchanged strings, not money and baked goods.

```
//shared variables
Semaphore entry = 0
Semaphore server[N] = {0} // each worker has a semaphore
Semaphore waiting_customer = 1
string buffer[N] // buffer for customer/worker communication (index == worker's index)
int next_server //index of the first worker ready to help a customer

// customer's code
wait(entry)
i = next_server
buffer[i] = request
signal(server[i]) // wakeup my server
wait(server[i]) // wait for my server to finish my request
result = buffer[i] // get my results from the buffer

// code for worker my_id
while (true)
{
    wait(waiting_customer) // make sure only one server at a time is next_server
    next_server = my_id
    signal(entry) // allow a customer into the bakery
    wait(server[my_id]) // wait for the customer to come in and get my_id out of next_server
    signal(waiting_customer) // now that a customer is in the bakery I can let next worker go
    handle request in buffer[my_id]
    put result back into buffer[my_id]
    signal(server[my_id]) // wake up customer now that request has been served
}
```