

### MIDTERM EXAM

- This test contains 8 questions with a total of 68 points
  - 1-3           4 points each
  - 4-5           8 points each
  - 6-7           12 points each
  - 8              16 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. However, there should be enough room for your answer under each question. If you find you need lots more room you might not be answering the question.
- 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) (4 points) What type of process scheduling can lead to 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.

2) (4 points) Briefly outline how the functions `regex()` and `regcomp()` were used to implement regular expressions in the `locate` assignment.

The regular expression (read from the command line) is compiled using `regcomp()` (`regcomp()` puts the compiled expression into a `regex_t` structure). Then each filename was compared to the compiled regular expression using the `regex_t` and `regex()`.

3) (4 points) Given the following enumerated type, declare and initialize a variable that represents the days Tuesday and Thursday.

```
enum Days {MONDAY = 1,
           TUESDAY = 2,
           WEDNESDAY = 4,
           THURSDAY = 8,
           FRIDAY = 16,
           SATURDAY = 32,
           SUNDAY = 64
};
```

`Days class = Days(TUESDAY | THURSDAY);`

Since each entry in this enumerated type is assigned a power of 2, we can use bitwise-or (the single `|`) to combine multiple values. For example, `TUESDAY` is `00010` and `THURSDAY` is `001000` so `TUESDAY | THURSDAY` is `001010` == `10`. Thus the value `10` can be used to represent both `TUESDAY` and `THURSDAY`.

4) (8 points) Consider the code:

initialization:	<code>thread1</code>	<code>thread2</code>
<code>int y = 42;</code>	<code>y = y + 1;</code>	<code>y = y + 2;</code>

Give all the possible values for `y` after running this code. Explain each one.

- `y = 45` `thread1` completes before `thread2` runs  
-or- `thread 2` complete before `thread1` runs
- `y = 43` `thread1` starts but is interrupted after getting the value of `y` but before setting the value. Then `thread2` runs 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` runs until completion. Finally `thread1` completes.

5) (8 points) The following is a complete and working program. What does it print? Explain how it works. Include a discussion of processes.

```
#include <iostream>
#include <string>
#include <stdlib.h>
#include <assert.h>
using namespace std;

void print(string str)
{
    cout << str << endl;
}

void print_twice(string str)
{
    cout << str << endl;
    cout << str << endl;
}

void call(void (ptr)(string), string data)
{
    pid_t pid = fork();
    assert(pid >= 0); // placeholder for real error checking
    if (pid == 0)
    {
        ptr(data);
        exit(0);
    }
}

int main()
{
    call(print, "hello");
    call(print_twice, "bye");
}
```

This program prints “hello” once and “bye” twice. The strings (or even the characters) can be interleaved in any order.

The call() function creates a new process each time it is called. Thus this program creates two child processes. Each process created by call() invokes the function passed into call(). In this program one process executes the print() function and one process executes the print\_twice() function.

- 6) (12 points) The which command takes an argument and indicates which (if any) of the directories listed in the PATH environment variable contain an executable file with the same name as the argument. Outline how one would implement which. Include the system functions that would have to be called. If you forget the exact name of the system calls simply state what has to be done (for example, if you forgot that the getenv(char \*name) function returns the value of an environment variable simply write “use a system call to get the value of the environment variable”). You don't have to write the code, just provide an outline of what has to happen to implement this.

The first step is to get the value of the PATH variable by calling the getenv() function. The PATH value contains a series of directories separated by colons. This string will have to be parsed into its components (this can be done from scratch or a function like strtok() can be used).

Each directory in the PATH must be read (this is accomplished using opendir() and readdir()). The filename of each file (sub directories should be skipped) in this directory must be compared to the given target. If it matches then stat() can be used to find out if the user has execute permission on the file. If so, the full path of the file is printed.

- 7) (12 points) Outline how Linux pipes, fork(), and exec() are used to implement pipes in the Bash shell. For example, how `$ ls | grep | sort` would be handled by the Bash shell. You don't have to provide the code, simply provide an overview of what is done to implement a Bash pipe.

The shell creates a pipe for each neighboring command to communicate. In the above example, one pipe for ls and grep to communicate and one pipe for grep and sort to communicate. Then it forks a process to run each command. Each child process connects its input and output to the appropriate pipe **before** it calls exec() on the target command.

8) (16 points) Write a multi-threaded program to simulate tourists visiting the Empire State Building. When a tourists walks into the Empire State Building he waits in a queue for the elevator. When it is his turn he enters the elevator (which holds 40 people), rides to the top, exits the elevator, looks at the view for a random amount of time, gets back into the elevator, rides to the ground floor, and then exits the elevator. Assume that each tourist is modeled using a thread, and visits the Empire State Building in an infinite loop (while (true) { visit building; do something else;}). Also assume there is an operator thread which makes the elevator go up and down and tells the tourists when to get off of the elevator. Your solution should fill the elevator for each trip up and each trip down (the management does not want to waste any electricity on partially filled elevators). This means that at any given time there will be 40 or 0 people at the top. Since tourists look at the view for a random amount of time, there is the possibility that a tourist will look at the view for 0 time. Do not worry about the real life problem of one tourist making all the others wait. Use semaphores to provide synchronization.

```
// this solution allows the next round of passengers to start getting onto the elevator before it is empty
// this is a bit rude, but there will never be more than 40 people on the elevator at any one time
semaphore operator = 0
semaphore elevator = 40
semaphore top = 0
semaphore bottom = 0
semaphore mutex = 1
int count = 0
```

Tourist thread:

```
while(true)
    wait(elevator)
    wait(mutex)
    count++
    if (count == 40) // if I am last one on
        signal(operator)
    signal(mutex)
    wait(top)

    get off elevator
    look around random time

    wait(mutex)
    count--
    if (count == 0) // if I am last one back on
        signal(operator)
    signal(mutex)
    wait(bottom)
    signal(elevator)

    get off elevator
    remainder section
```

Operator thread:

```
while(true)
    wait(operator)
    make elevator go up
    for (i = 0; i < 40; i++)
        signal(top)
    wait(operator)
    make elevator go down
    for (i = 0; i < 40; i++)
        signal(bottom)
```