

MIDTERM EXAM

- This test contains 9 questions with a total of 74 points
 - 1-5 6 points each
 - 6 8 points each
 - 7-9 12 points each
- 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) (6 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) (6 points) The `waitpid()` system call has three possible options: `WNOHANG`, `WUNTRACED`, and `WCONTINUED`. The function can be called with zero, one, two or three of these options. Explain how one argument (options) is used to handle all the possible options.

```
pid_t waitpid(pid_t pid, int *status, int options)
```

Each of the three option flags (e.g. `WNOHANG`) are defined as integers that do not have any 1 bits in common (e.g. 2 (0010) and 4 (0100) don't have any 1 bits in common, but 2 (0010) and 3 (0011) both have the 2nd bit set). The flags are combined into a single integer using bitwise-or (`|`).

```
waitpid(pid, &status, WNOHANG | WUNTRACED);
```

In the function, the options flag is compared to each option using bitwise-and (`&`).

```
if (option & WNOHANG)
```

- 3) (6 points) Solutions to the critical section problem must satisfy which requirements? Briefly explain each.

Mutual Exclusion: If a process is executing in its critical section, then no other processes can be execution in the same critical section.

Progress: If no process is executing in its critical section and some process wants to enter, it should not be postponed indefinitely.

Bounded Waiting: There exists a bound on the numbers of times a process will be kept waiting while other process *cut* in front of it.

- 4) (6 points) The follow code compiles but does not work. Explain the problem and how to fix it.

```

// prototypes from regex.h
// int regcomp(regex_t *preg, const char *regex, int cflags);
// int regexec(const regex_t *preg, const char *string, size_t nmatch, regmatch_t pmatch[], int eflags);

regex_t *reg_expression;
result = regcomp(reg_expression, "[0-9]*$", REG_EXTENDED);
...

result = regexec(reg_expression, "42", 0, 0, 0);

```

Similar to many system functions, `regcomp()` expects a pointer to memory that can be filled in by the function. The pointer `reg_expression` is not pointing to memory that `regcomp()` can use. It can be fixed by declaring a `regex_t` structure and passing the address:

```

regex_t reg_expression;
result = regcomp(&reg_expression, "[0-9]*$", REG_EXTENDED);
...

result = regexec(&reg_expression, "42", 0, 0, 0);

```

5) (6 points) Describe what happens when the following program is executed.

```

#include <stdio.h>

void f(int c)
{
    printf("%d\n", c);
    f(c+1);
}

int main()
{
    f(0);
    return 0;
}

```

Function `f()` is a recursive function without a base case. That means it calls itself over and over until the memory allocated for the process' run-time stack is full (not enough room to allocate another activation record). At that point the program will cease to run. The problem is called *stack overflow*. The error printed when the process terminates should be "stack overflow" but more typically it is "segmentation fault" or "bus error."

Before the process terminates, it prints increasing integers starting at zero (0,1,2,...).

6) (8 points) Consider the code:

initialization:	thread1	thread2
int x = 1;	x = y + x;	y = x + y;
int y = 1;		

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

x = 2 thread1 runs to completion before thread2 runs
y = 3

x = 3 thread2 runs to completion before thread3 runs
y = 2

x = 2 thread1 fetches the value of y before thread2 assigns new value to y
y = 2 AND
 thread2 fetches the value of x before thread1 assigns new value to x

7) (12 points) Write a program to demonstrate how the Spawn class can be used to creates a new process

running a standard Linux command (pick any command that reads and writes) and pass data to/from your sample program to the process created by Spawn. In other words, write a program that uses the Spawn class to pass data to and receive data from a process running a standard Linux command.

```
class Spawn
{
public:
    Spawn(bool pipe_input, bool pipe_output, const char
*executable, ...); // null terminated list of const char * args
    ~Spawn();
    char getchar();
    int putchar(char c);
    int putstr(const char *str);
    void close_input();
private:
    // put anything you want here
};

// this program prints the command line arguments sorted alphabetically
// it uses the Linux sort utility and the Spawn class to sort the arguments
int main(int argc, char *argv[])
{
    Spawn sort(/* pipe_input = */ true,
              /* pipe_output = */ true,
              "/usr/bin/sort", 0);

    for (int i = 1; argv[i]; i++)
    {
        sort.putstr(argv[i]);
        sort.putchar('\n');
    }
    sort.close_input();

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

8) (12 points) When automatically testing a program (call the program being tested the *target program*),

there is always the chance the target program will enter an infinite loop and thus the testing program will be stuck forever waiting for the target program to terminate. Describe how the testing program can be implemented so that it will not become stuck if the target program enters an infinite loop. The target program should be killed if it runs more than 10 seconds. Your solution cannot use sleep() or a busy wait. You do not have to write the code, just explain how the testing program can manage target programs with an infinite loop.

Testing program uses fork() and exec() to start the target program (the one being tested)

Testing program sets a signal handler for SIGALRM

Testing program calls alarm() which tells the OS to send the process a SIGALRM signal after a predetermined time interval

Testing program waits() for process

If testing program receives SIGALRM, it kills child process (program being tested) by sending it a SIGKILL

If the target program terminates before

9) (12 points) Create a solution to the bounded buffer problem. Specifically, write a producer thread that

generates integers and places them in a shared buffer. The producer should not block unless the buffer is full. Write a consumer thread that reads integers from the buffer. The consumer should not block unless the buffer is empty. Assume the buffer is of size N. Use conditional variables for synchronization. Don't worry about syntactic details (such as how to initialize a conditional variable, or the exact arguments to wait() and signal()). If the algorithm is correct, you will get full credit.

This is working code. I do not expect students to write working code on exams.

```
const int N = 3; // size of buffer
int data = 0;
int buffer[N];

int size = 0;
int first;
int last;

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t full = PTHREAD_COND_INITIALIZER;
pthread_cond_t empty = PTHREAD_COND_INITIALIZER;

void *producer_code(void *)
{
    while(1)
    {
        pthread_mutex_lock(&mutex);
        while (size == N)
            pthread_cond_wait(&full, &mutex);
        if (size == 0)
        {
            first = 0;
            last = 0;
        }
        else
            last = (last + 1) % N;
        size++;
        pthread_cond_signal(&empty);
        buffer[last] = data++;
        pthread_mutex_unlock(&mutex);
        // use data
    }
    return 0;
}

void *consumer_code(void *)
{
    while (1)
    {
        pthread_mutex_lock(&mutex);
        while (size == 0)
            pthread_cond_wait(&empty, &mutex);
        cout << "read " << buffer[first] << endl;
        first = (first + 1) % N;
        size--;
        pthread_cond_signal(&full);
        pthread_mutex_unlock(&mutex);
    }
    return 0;
}
```