

MIDTERM EXAM

- This test contains 8 questions with a total of 60 points
 - 1-5 5 points each
 - 6-7 10 points each
 - 8 15 points
- You have 50 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. If you don't understand the question, ask me.

1) (5 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.

2) (5 points) What is the run time stack? Include an explanation of when items are added and removed from the stack. 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 placed on the run time stack. When the function terminates its activation record is popped from the run time stack.

3) (5 points) On machines with multiple processors race condition 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.

4) (5 points) What factors should be considered when determining the length of a time quantum (time slice) used by a preemptive scheduler?

Average CPU burst length: want to pick a time quantum long enough so most processes block or terminate before using up their time quantum

Responsiveness: want a time quantum short enough so the system provides adequate responsiveness.

Context switching time: if the time quantum is too short, too much time will be spent switching contexts. In other words, the CPU will spend so much time switching contexts that significantly less real work will be done.

5) (5 points) Assume two processes (P_0 and P_1) run the following code. Explain why this is a race condition. Make sure your explanation covers all of the important issues.

```
Pi:
while(flag == true)
    ; // empty loop
flag = true;
// critical section
flag = false;
```

The goal of this code is to make processes take turns executing the *critical section* code. Only one process should ever *think* flag is false because right after checking if flag is false the flag is set to true.

The problem occurs when one process is preempted right after the while loop and a second process runs the code. Both processes will *think* the flag was false and both will execute the critical section code at the same time.

- 6) (10 points) When a child process exits a SIGCHLD signal is sent to the parent. Modify the parent process code in the following program so the parent process exits when the child quits. Don't alter the functionality of the parent process (it should continue executing its loop). Don't forget the task parent processes are suppose to perform before exiting. SIGCHLD signals are also sent when a child process blocks. Assume that the parent only received a SIGCHLD signal if the child has exited.

```
void sigchld_handler(int signo)
{
    if (wait(NULL) <= 0)
        perror("wait()");
    exit(0);
}

int
main()
{
    if (fork() == 0)
    {
        // simulate performing a long task
        // sleep 0 - 10 seconds
        sleep(rand() % 10);
    }
    else
    {
        if (signal(SIGCHLD, sigchld_handler) != 0)
        {
            perror("signal(SIGCHLD, sigchld_handler)");
            exit(1);
        }
        while (true)
        {
            printf("still running, press return to get status\n");
            getc(stdin);
        }
    }
}
```

7) (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 could always get the CPU when another process is in the critical section.

8) (15 points) An old one-lane bridge is on a north-south highway. Since it is a one-lane bridge, at any time all the cars must be going the same direction (north or south). Since it is an old bridge, only three cars can be on it at any time. Write code to simulate cars driving on the bridge. Use semaphores for synchronization. Hint: first write the solution for any number of cars and then modify it so only three cars can be on the bridge at once.

```
semaphore bridge = 1;
semaphore total_north = 3;
semaphore total_south = 3;
semaphore north_mutex = 1;
semaphore south_mutex = 1;
int north_count = 0;
int south_count = 0;
```

north:

```
wait(total_north);
wait(north_mutex);
north_count++;
if (north_count == 1)
    wait(bridge);
signal(north_mutex);
drive_on_bridge();
wait(north_mutex);
north_count--;
if (north_count == 0)
    signal(bridge);
signal(north_mutex);
signal(total_north);
```

south:

```
wait(total_south);
wait(south_mutex);
south_count++;
if (south_count == 1)
    wait(bridge);
signal(south_mutex);
drive_on_bridge();
wait(south_mutex);
south_count--;
if (south_count == 0)
    signal(bridge);
signal(south_mutex);
signal(total_south);
```