Rules:
- Do all your own work. Nothing says your neighbor has any better idea what the answer is.
- Do not discuss this exam outside of class until after 4:00pm.

Suggestions (mostly the obvious):
- When in doubt, state any assumptions you make in solving a problem. If you think there is a misprint, ask me.
- Read the questions carefully. Be sure to answer all parts.
- Identify your answers clearly.
- Watch the time/point tradeoff: 95ts/50min works out to 31.6s/pt.
- Problems are not necessarily in order of difficulty.
- Be sure you have all pages. Pages other than this one are numbered “n of 7”.

Encouragement:
- Good Luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Possible</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Total:</td>
<td>95</td>
<td></td>
</tr>
</tbody>
</table>
Answer the following questions clearly, concisely, and (where possible) correctly.

1. (10) Busywaiting is often denigrated in the concurrent programming community as a crude and inefficient practice. However, it has its place.
   
   (a) First, what is busywaiting?

   (b) Under what circumstances would it be appropriate to choose a busywaiting approach?

2. (10) Semaphore waiting lists are often implemented as queues served in FIFO order. Could they be implemented as stacks? What problems might this cause?
3. (10) What will be the output of the following program? Explain.

```c
int main(int argc, char *argv[]){
    printf("Hello.\n");
    fork();
    fork();
    fork();
    printf("Goodbye!\n");
    return 0;
}
```

4. (10) A programmer dissatisfied with the behavior of a C library function can redefine it without limiting the capabilities of the program. (That is, there is nothing the program could have done before the redefinition that it could not do afterwards.) A system call, however cannot be replaced without limiting the program. Why is this?
5. (15) Given the following program:

```c
int main(int argc, char *argv[]){
    int n;
    for(n = 1; n < argc; n++)
        execvp( argv[n], &(argv[n]) );
    return 0;
}
```

Suppose that it is compiled in the current directory as `myprog` and that one gives the command:

```
% myprog myprog ls myprog myprog myprog
```

(a) How many times will `execvp()` be called during the run of this program?

(b) What will be the expected output of the run?

(c) Explain.
6. (20) Assume you have hacked the kernel and are using a version of MINIX with a 1 second(!) quantum. Suppose there are five user-level jobs, labeled A through E, that require the following amounts of time to run (if they could get the whole CPU to themselves).

<table>
<thead>
<tr>
<th>Job</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2 sec.</td>
</tr>
<tr>
<td>B</td>
<td>3 sec.</td>
</tr>
<tr>
<td>C</td>
<td>4 sec.</td>
</tr>
<tr>
<td>D</td>
<td>5 sec.</td>
</tr>
<tr>
<td>E</td>
<td>2 sec.</td>
</tr>
</tbody>
</table>

Supposing that these five jobs become runnable at very close to the same time and are initially added to the run queue in alphabetical order, answer the following questions.

Recall that within a priority class, MINIX uses round-robin scheduling. You may assume that these are the only five jobs in the system and that the overhead required for context switches is negligible.

(a) After how many seconds will each job terminate? Briefly explain your reasoning.

(b) Now suppose that job D issues an I/O request after each 1.4s it is allowed to run. This I/O request takes 200ms (0.2s) to complete and causes D to block. Now how long will it take for each job to complete? Again, explain your reasoning.
7. (20) Consider the following situation: A very narrow hurricane has washed out all but one lane of the Lake Pontchartrain Causeway\(^1\). Given that it is a very large lake, going around is impractical, so it is necessary to come up with a system to keep the bridge open. The conditions:

- Cars arrive at random intervals from either the north or south.
- The remains of the bridge are only one car wide and cars cannot back up. That is, a car that meets another car is stuck forever.
- Whenever a car wants to enter the bridge, it calls the function `enter_bridge(int direction)` with a pre-defined integer constant indicating the direction. This will be either `NORTH` or `SOUTH`. When it wants to leave, it calls `exit_bridge(int dir)`.

Using semaphores and the C-like syntax used for semaphore examples in class and in Tanenbaum and Woodhull, develop a solution to the problem. Implement `enter_bridge()` and `exit_bridge()` and whatever auxiliary data and functions you may need. **Be sure to explain briefly why your solution works.**

**For partial credit:** produce a solution that allows cars to cross the bridge without risking meeting another car on the way (and getting stuck forever).

**For more partial credit:** produce a solution that guarantees that no car will have to wait forever to cross.

**For full credit:** produce a solution that does all of the above and allows multiple cars travelling in the same direction to be on the bridge at a time. (It is 24 miles long, after all.)

Write your code below (and on page 7 if necessary):

---

\(^1\)Lake Pontchartrain is a lake 41 miles long and 24 miles wide north of New Orleans, La. The causeway is the bridge that spans the “short” direction and is one of the two roads out of the city.
Extra space for problem 7