# Sample Mid-Term Exam 1

CS 5460/6460, Fall 2009

September 24

Name: _____

**Instructions:** This is a sample mid-term that is shorter (the real one will have 4 questions) and easier than the real one, but with the same style of questions. You will have 1 hour and 20 minutes to complete the real one, which is an open-book, open-note, closed-computer exam.

Here is a little program that is compiled as `inc` to be used in some of the questions:

```
#include <unistd.h>

int main() {
  char buf[1];
  read(0, buf, 1);   buf[0]++;   write(1, buf, 1);
  return 0;
}
```

**1**) The following program was meant to print "2" to stdout:

```
#include <unistd.h>

int main (int argc, char **argv, char **envp) {
  int xfds[2], yfds[2];
  char *argv2[2] = { "inc", NULL };

  pipe(xfds);
  pipe(yfds);

  write(xfds[1], "0", 1);
  if (!fork()) {
    dup2(xfds[0], 0);
    dup2(yfds[1], 1);
    execve("inc", argv2, envp);
  }
  if (!fork()) {
    dup2(yfds[0], 0);
    execve("inc", argv2, envp);
  }
  return 0;
}
```

Sometimes it works right, but sometimes the program exits before printing anything, even though none of the system or library calls fail. How can it be fixed?

**2**) What are the possible outputs of the following program? You can assume that none of the system or library calls fail.

```
#include <unistd.h>
#include <sys/wait.h>

int main (int argc, char **argv, char **envp) {
  int fds[2], status;
  char buf[1], *argv2[1] = { NULL };
  pid_t pid;

  pipe(fds);

  if (!fork()) {
    dup2(fds[0], 0);
    execve("inc", argv2, envp);
  }
  if (!fork()) {
    dup2(fds[0], 0);
    execve("inc", argv2, envp);
  }
  write(fds[1], "01", 2);

  return 0;
}
```

You can just list the possible outputs, but if you explain your reasoning, the explanation could be worth partial credit even if you list the wrong outputs.

**3**) In a program that contains the declarations

```
struct clown_t {
  int shoe_size;
  int cream_pies;
};
static struct clown_t *binky;
```

the original programmer had written

```
binky->cream_pies++;
```

in a procedure that is used in multiple threads. Another programmer later "repaired" the statement to ensure that multiple threads don't try to read the `binky` variable at the same time:

```
struct clown_t *b;

lock();
b = binky;
unlock();

b->cream_pies++;
```

Although no two threads no use the `binky` variable at the same time, explain what is wrong with the repair and how to fix it.