```
/*
 * If the new process paused because it was
 * swapped out, set the stack level to the last call
 * to savu(u_ssav).  This means that the return
 * which is executed immediately after the call to aretu
 * actually returns from the last routine which did
 * the savu.
 *
 * You are        expected to understand this.
 */
```



---

## WeensyOS Minilab 1

Please check back every now and then, as we may clarify this problem set description.

The WeensyOS problem sets are a series of little coding exercises that are also complete operating systems. You could boot a WeensyOS operating system on real x86-compatible hardware! The purpose of the WeensyOS exercises is first, to teach some of the concepts we use in class through example, and second, to demystify operating systems in general. I also hope they are fun.

The first WeensyOS problem set concerns *processes*. The MiniprocOS is a tiny operating system that supports the major process primitives: creating a new process with `fork`, exiting a process, and waiting on a process that has exited. The only thing missing -- and it's a big one -- is process isolation: MiniprocOS processes actually share a single address space. (In a later minilab you will implement process isolation for memory.) In this problem set, you will actually implement the code that forks a new process, and see how system calls are implemented. You will also update the code that waits on an exited process to avoid busy waiting.

| | |
|---|---|
| **`weensyos1.tar.gz`** | Source code for WeensyOS 1, which builds this hard disk image: |
| `mpos.img` | MiniprocOS, plus two applications that create and run new processes. |

In this simple problem set, you'll browse, partially understand, and change this tiny operating system.

### Handing in

You will electronically hand in code and a small writeup containing answers to the numbered exercises. The problem set code, `weensyos1.tar.gz` (available on CourseWeb), unpacks into a directory called `weensyos1`. (We explain how to unpack it below.) You'll modify the code in this directory, and add a text file with your answers to the numbered exercises. When you're done, run the command **`make tarball`**. This should create a file named `weensyos1-`*`yourusername`*`.tar.gz`. You'll turn in this file to CourseWeb.

**Answer the numbered exercises** by editing the file named `answers.txt`. *No Microsoft Word documents* (or other binary format, except for PDF in special cases) *will be accepted!* For coding exercises, it's OK for `answers.txt` to just refer to your code (as long as you comment your code).

To review:

1. Download `weensyos1.tar.gz` and unpack it.
2. Do your work in the `weensyos1` directory.
3. Fill out the `answers.txt` file in that directory.
4. When you're done, run **`make tarball`** from the `weensyos1` directory. This will create a file named `weensyos1-`*`yourusername`*`.tar.gz`.
5. Submit that `weensyos1-`*`yourusername`*`.tar.gz` file to CourseWeb.

### Setting up

You could take one of the disk image files this minilab builds, write it to your laptop's hard drive, and boot up your operating system directly if you wanted! However, it's much easier to work with a *virtual* machine or *PC emulator*.

An emulator mimics, or *emulates*, the behavior of a full hardware platform. A PC emulator acts like a Pentium-class PC: it emulates the execution of Intel x86 instructions, and the behavior of other PC hardware. For example, it can treat a normal file in your home directory as an emulated hard disk; when the program inside the emulator reads a sector from the disk, the emulator simply reads 512 bytes from the file. PC emulators are much slower than real hardware, since they do all of the regular CPU's job in software -- not to mention the disk controller's job, the console's job, and so forth. However, debugging with an emulator is a whole lot friendlier, and you can't screw up your machine!

We've used two PC emulators. The **Bochs** emulator has pretty nice debugging support. The **QEMU** package is fast and sleek, but it might be *too* fast for some of our purposes. You will also need a copy of GCC that compiles code for an x86 ELF target. Recent Linux PCs have the right compiler already set up.

We strongly recommend that you use the CS 111 Ubuntu Distribution if you want to work from home. We've set up all the required tools on the machines in the Linux lab, and the SEASnet Linux servers. In the Linux lab, no special setup is required.

Now that you've got all the software set up (or you've just decided to use the Linux lab), it's time to download WeensyOS and take it out for a spin.

Unpack the source for weensyos1 using the following command.

```
% tar xzf weensyos1.tar.gz
```

(On Linux, you can just say "**`tar xzf weensyos1.tar.gz`**".) This should unpack the tarball into the `weensyos1` directory.

```
% ls weensyos1
COPYRIGHT    conf   mergedep.pl   mpos-app2.c  mpos-kern.h      mpos.h
GNUmakefile  elf.h  mkbootdisk.c  mpos-boot.c  mpos-loader.c    types.h
answers.txt  lib.c  mpos-app.c    mpos-int.S   mpos-symbols.ld  x86.h
bootstart.S  lib.h  mpos-app.h    mpos-kern.c  mpos-x86.c
%
```

Now that you've unpacked the source, it's time to give the OSes a whirl.

Change into the `weensyos1` directory and run the **`make`** program (which must be GNU make).

The WeensyOS `GNUmakefile` builds a hard disk image called `mpos.img`, which contains the MiniprocOS "kernel" and two applications, `mpos-app.c` and `mpos-app2.c`.

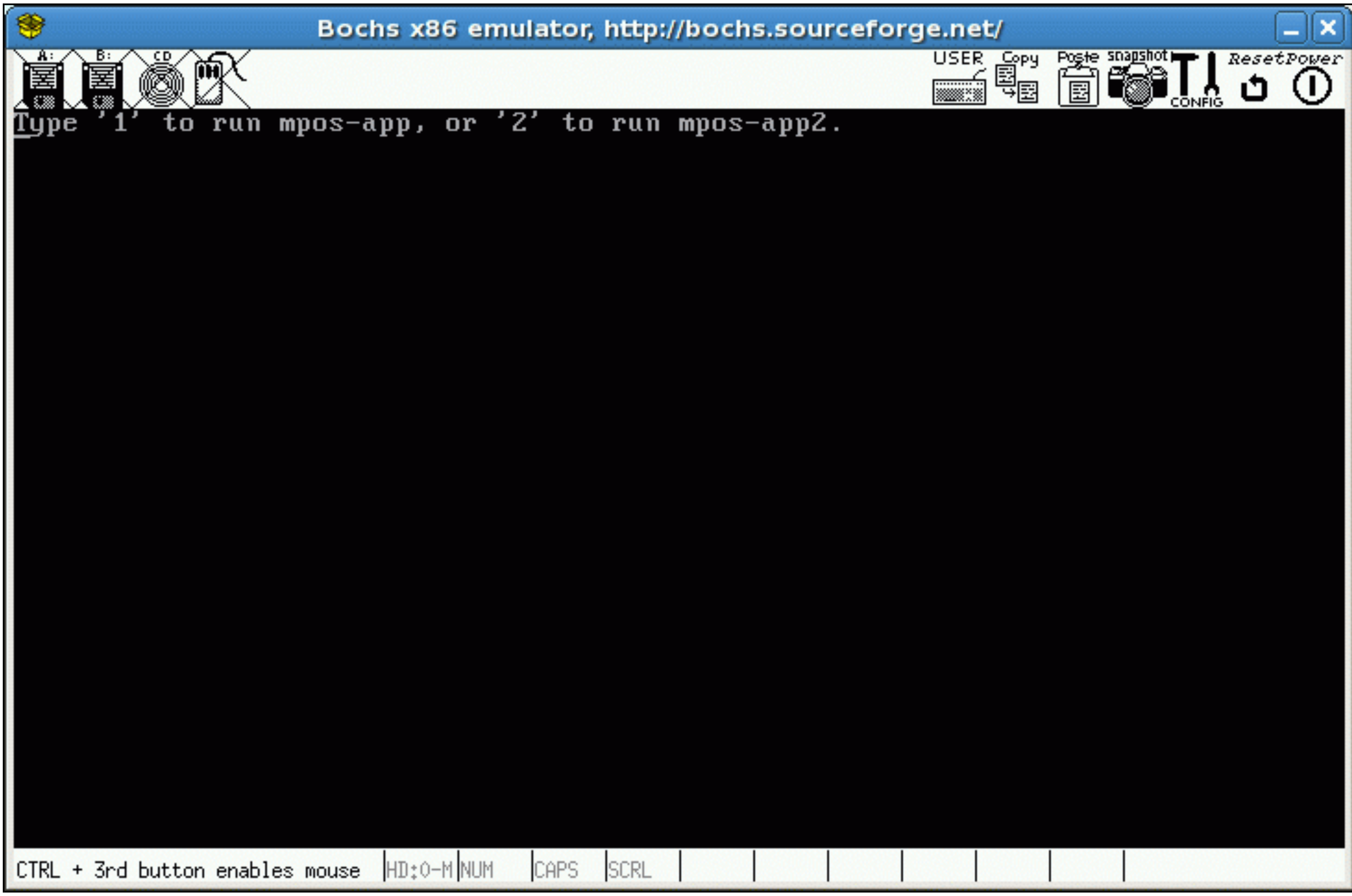
Make's output should look something like this:

```
% make
+ hostcc mkbootdisk.c
+ as bootstart.S
+ cc mpos-boot.c
+ ld mpos-bootsector
+ as mpos-int.S
+ cc mpos-kern.c
+ cc mpos-x86.c
+ cc mpos-loader.c
+ cc lib.c
+ cc mpos-app.c
+ ld mpos-app
+ cc mpos-app2.c
+ ld mpos-app2
+ ld mpos-kern
+ mk mpos.img
%
```

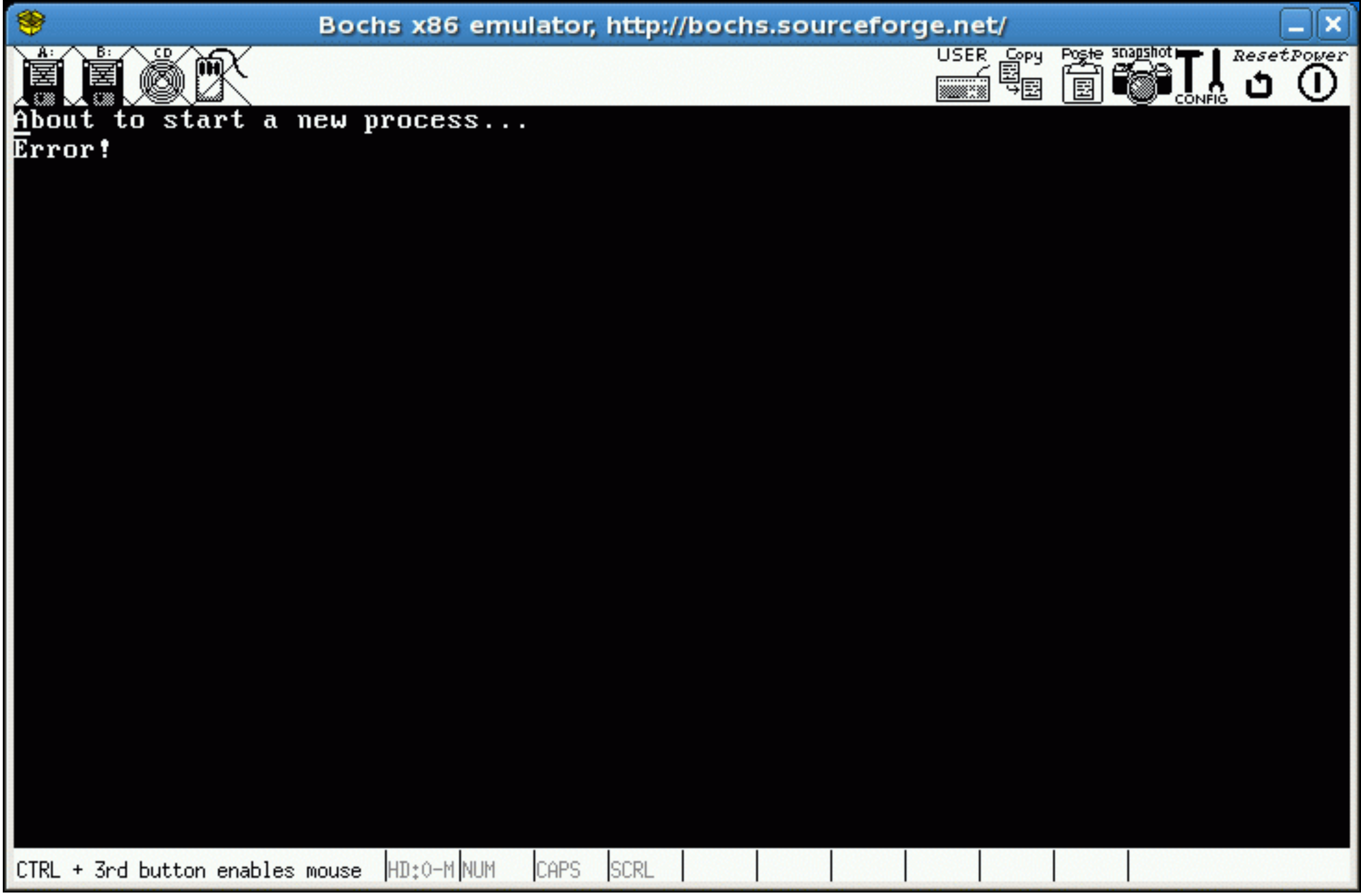
Now that you've built the OS disk image, it's time to run it! We've made it very easy to boot a given disk image; just run this command:

```
% make run-mpos
```

This will start up Bochs. After a moment you should see a window like this!




Hit "1" to try to run the first application, and you should see a window like this:




To quit Bochs, click the "Power" button in the upper-right corner. (Very funny, Bochs.)

**QEMU Note.** If you're running QEMU instead of Bochs, run the MiniprocOS with **`qemu -hda mpos.img`**. (The `-hda` option stands for Hard Disk A.) QEMU doesn't have a funky power button; just hit Control-C in the terminal to quit. QEMU will sometimes "grab" the keyboard, which prevents you from doing anything else. If you appear to have lost control of your computer, check QEMU's title bar: it may say something like "Press Ctrl-Alt to exit grab". Press Ctrl-Alt and things should return to normal.

### MiniprocOS Application

You're now ready to start learning about the OS code!

Start first with the application, `mpos-app.c`. This application simply starts a single child process and waits for it to exit. It uses system calls that implement the process functions we discussed in class: `fork` starts a new process; `exit` exits a process; and `wait` returns a process's exit status.

> Read and understand the code in `mpos-app.c`.

How are those system calls implemented? As discussed in class, to call a system call, the application program executes a *trap*: an instruction that initiates a protected control transfer to the kernel. The system call's arguments are often stored in machine registers, and that's how MiniprocOS does it. Likewise, the system call's results are often returned in a machine register. On Intel 80386-compatible machines (colloquially called "x86es"), the interrupt instruction is called `int`, and registers have names like `%eax`, `%ebx`, and so forth. A special C language statement, called `asm`, can execute the interrupt instruction and connect register values with C-language variables.

Read and understand the comments in `mpos-app.h`. This file defines MiniprocOS's system calls. Also glance through the code, to see how system calls actually work!

The MiniprocOS *kernel* handles these system calls.

This kernel is different from conventional operating system kernels in several ways, mostly to keep the kernel as small as possible. For one thing, the kernel shares an address space with user applications, so that user applications could write over the kernel if they wanted to. This isn't very *robust*, since the kernel is not isolated from user faults, but for now it is easier to keep everything in the same address space. Another difference is that MiniprocOS implements *cooperative multitasking*, rather than *preemptive multitasking*. That is, processes give up control *voluntarily*, and if a process went into an infinite loop, the machine would entirely stop. In preemptive multitasking, the kernel can *preempt* an uncooperative process, which forces it to give up control. Preemptive multitasking is more robust than cooperative multitasking, meaning it's more resilient to errors, but it is slightly more complex. All modern PC-class operating systems use preemptive multitasking for user-level applications, but the kernel itself usually switches between internal tasks using cooperative multitasking.

MiniprocOS's main kernel structures are as follows.

**`struct process_t`**
: This is the *process descriptor* structure, which stores all the relevant information for each process. It is defined in `mpos-kern.h`.

**`process_t miniproc[];`**
: This is an array of process descriptor structures, one for each possible process. MiniprocOS supports up to 15 concurrent processes, with process IDs 1 to 15. The process descriptor for process *`I`* is stored in `miniproc[`*`I`*`]`. Initially, only one of these processes is active, namely `miniproc[1]`. The `miniproc[0]` entry is never used.

**`process_t *current;`**
: This points to the process descriptor for the currently running process.

The code in `mpos-kern.c` sets up these structures. In particular, the `start()` function initializes all the process descriptors.

Read and understand the code and comments in `mpos-kern.h`. Then read and understand the memory map in `mpos-kern.c`, the picture at the top that explains how MiniprocOS's memory is laid out. Then look at `start()`.

The code you'll be changing in MiniprocOS is the function that responds to system calls. This function is called `interrupt()`.

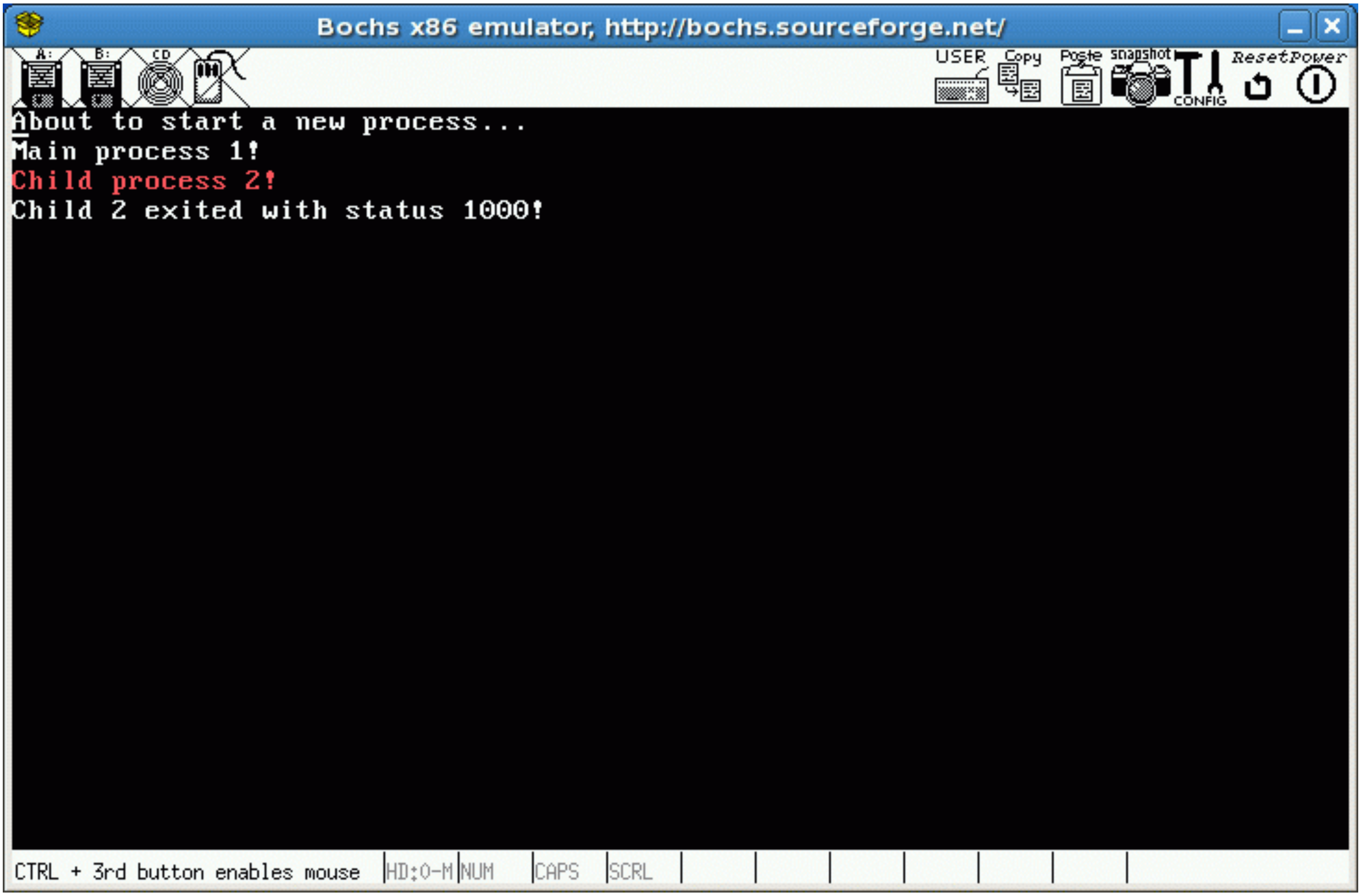
Read and understand the code for `interrupt()` in `mpos-kern.c`. Concentrate on the simplest system call, namely `sys_getpid/INT_SYS_GETPID`. Understand how the `sys_getpid` application function (in `mpos-app.h`) and the `INT_SYS_GETPID` clause in `interrupt()` (in `mpos-kern.c`) interact.

**Exercise 1.** Answer the following question: Say you replaced `run(current)` in the `INT_SYS_GETPID` clause with `schedule()`. The process that called `sys_getpid()` will eventually run again, picking up its execution as if `sys_getpid()` had returned directly. When it does run, will the `sys_getpid()` call have returned the correct value?

You may have noticed, though, that the `sys_fork()` system call isn't working! Your job is to write the code that actually creates a new process.

**Exercise 2.** Fill out the `do_fork()` and `copy_stack()` functions in `mpos-kern.c`.

Congratulations, you've written code to create a process -- it's not that hard, no? (Our version is less than 20 lines of code.) Here's what you should see when you're done:




Now take a look at the code in `mpos-app.c` that calls `sys_wait()`. Also look at the `INT_SYS_WAIT` implementation in `mpos-kern.c`. The current system call design uses a *polling* approach: to wait for process 2 to exit, a process must call `sys_wait(2)` over and over again until process 2 exits and the `sys_wait(2)` system call returns a value different from `WAIT_TRYAGAIN`.

We'll see more about polling later in the quarter, but for now, notice that polling approaches like this often reduce *utilization*. A process uses CPU time to call `sys_wait(2)` over and over again, leaving less CPU time for others. An alternative approach, which can improve utilization, is called *blocking*. A blocking implementation would put `sys_wait(2)`'s caller to sleep, then wake it up once process 2 had exited and a real exit status was available. The sleeping process doesn't use any CPU. A process that is asleep because the kernel is waiting for some event is called *blocked*.

**Exercise 3.** Change the implementation of `INT_SYS_WAIT` in `mpos-kern.c` to use blocking instead of polling. In particular, when the caller tries to wait on a process that has not yet exited, that process should block until the process actually exits.

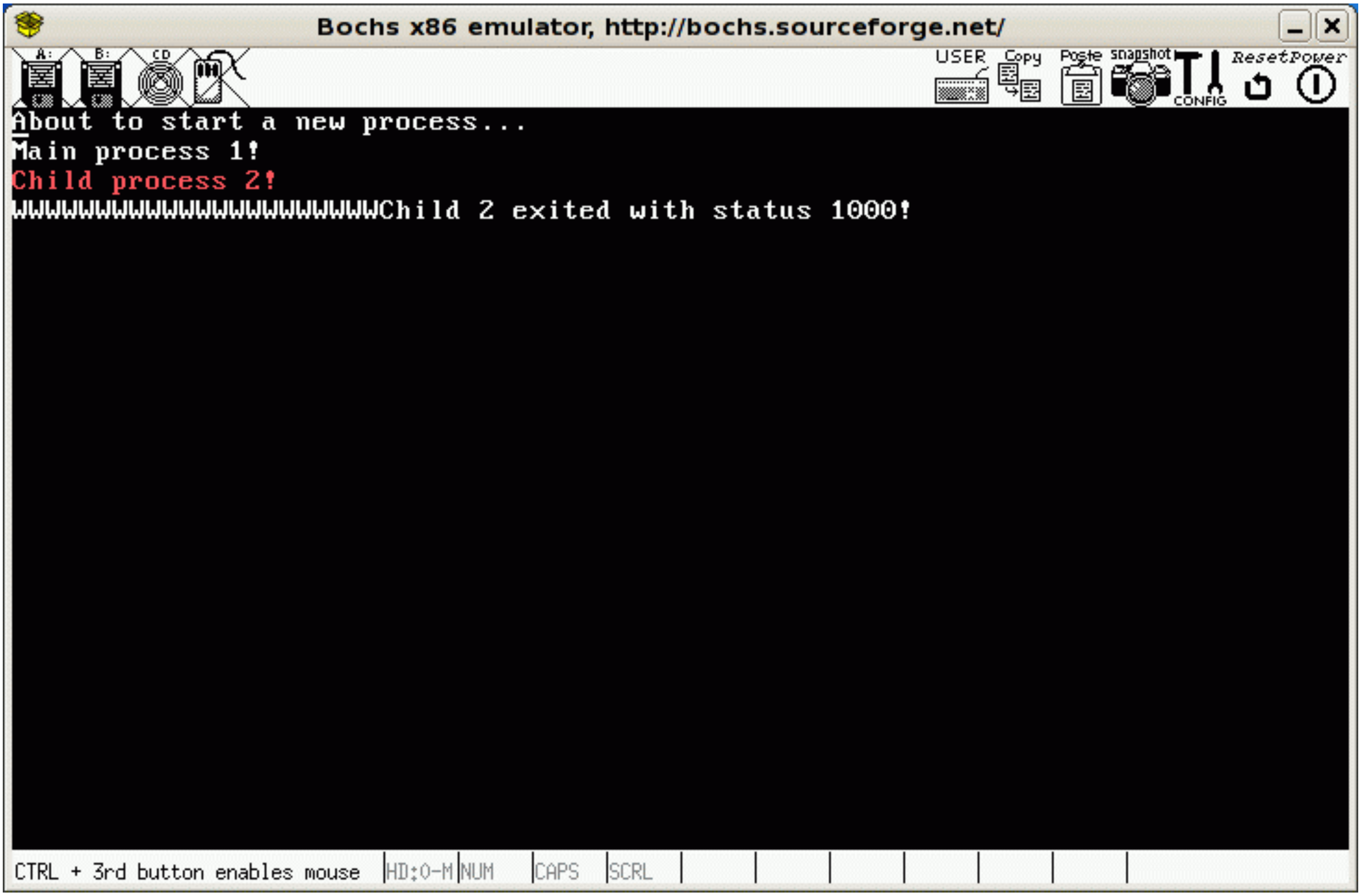
**Important Hint:** Make sure that your blocking version of `sys_wait()` has *exactly the same* user-visible behavior as the original version, except that it blocks and so never returns `-2`. See `mpos-app.h` for an English description of the current behavior.

To implement Exercise 3, you will probably want to add a field to the process descriptor structure. This field will indicate whether or not a process is waiting on another process. You will change `INT_SYS_WAIT` to add the calling process to this "wait queue", and `INT_SYS_EXIT` to wake any processes that were on the "wait queue". There are several ways to do this; describe how you did it in `answers.txt`.

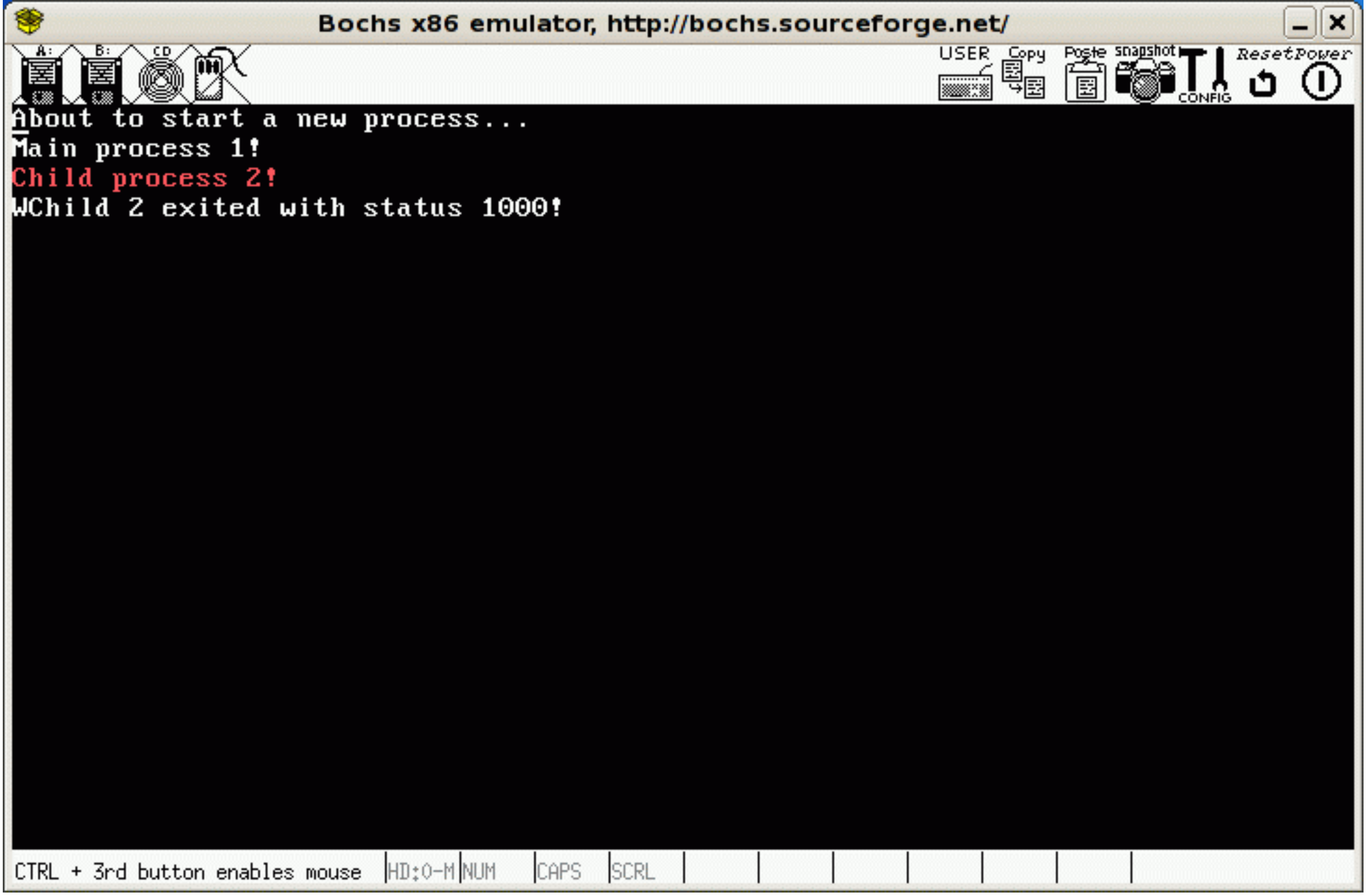
To check your work, try changing the `sys_wait()` loop in `mpos-app.c` to look like this:

```
                do {
                        status = sys_wait(p);
                        app_printf("W");
                } while (status == WAIT_TRYAGAIN);
```

A polling implementation of `sys_wait` would produce output like this:

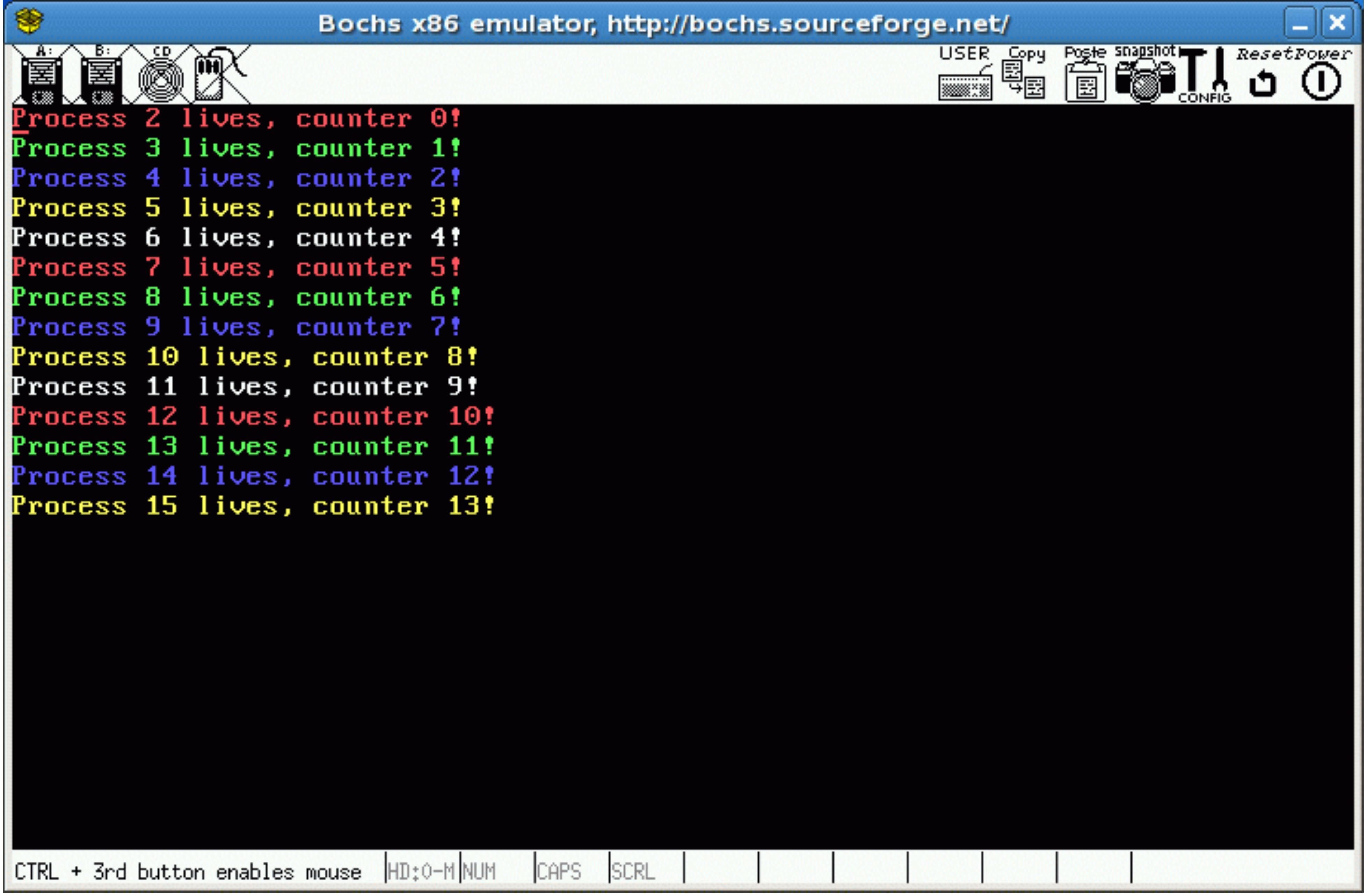



You want it to produce output like this:




### Cleaning Up Processes

Now try running the other MiniprocOS application. You should see something like this (different processes generally print their lines in different colors):



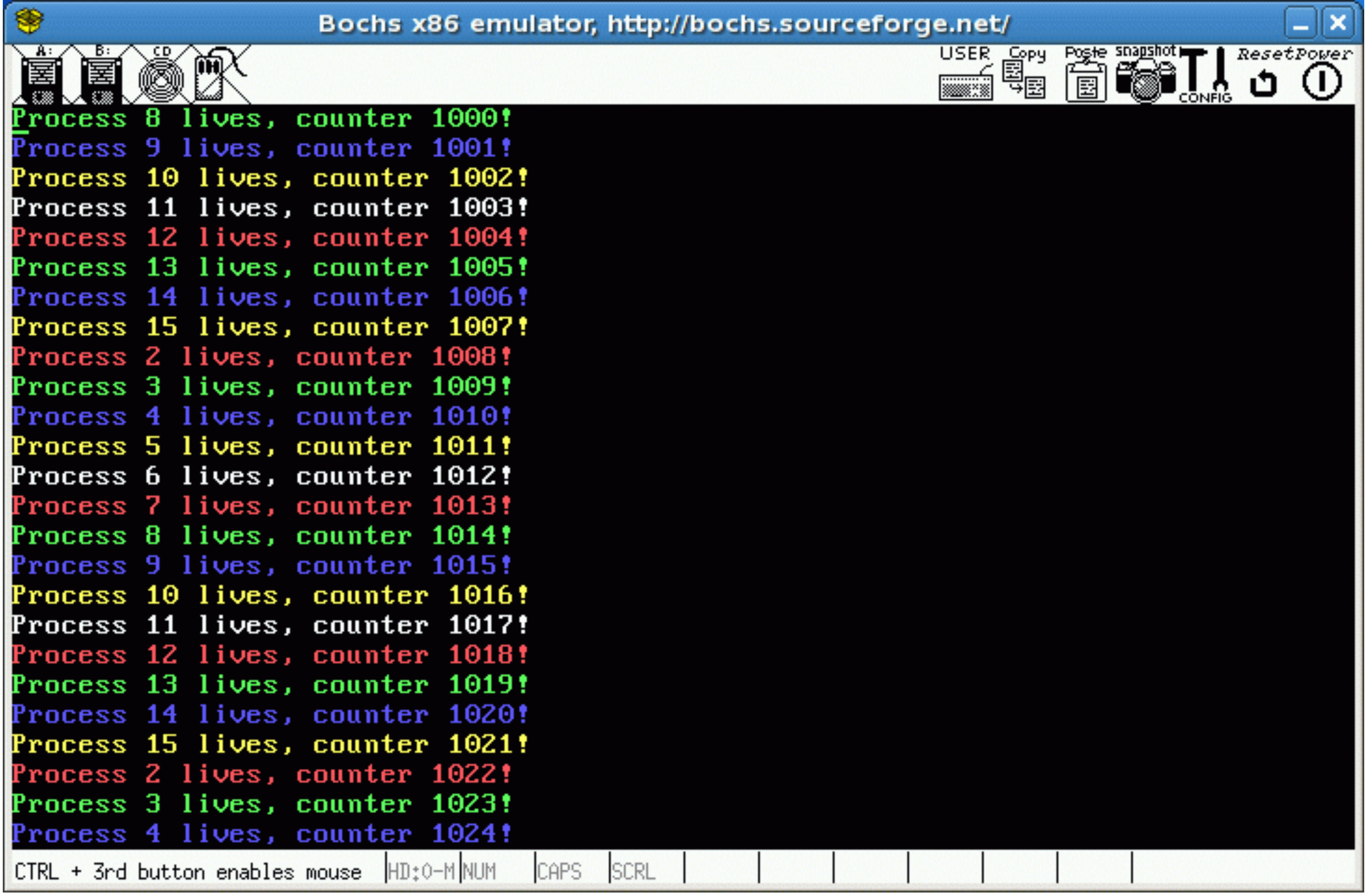

The MiniprocOS2 application, in `mpos-app2.c`, tries to run 1024 child processes.

> Read and understand `mpos-app2.c`.

Unfortunately, your current kernel code doesn't seem able to run more than 15 total processes, ever! It looks like old, dead processes aren't being cleaned up, even after we call `sys_wait()` on them. This is what we call a bug.

> **Exercise 4.** Find and fix this bug.

When you've completed this exercise, the application should count up to 1024, like this:




Your colors may differ, however, depending on how you implement `sys_wait()`. One common implementation strategy ends with several red lines in a row. If you see this in your code, try to figure out why!

**This completes the minilab.** But here are some extra credit opportunities, if you're interested.

**Extra-Credit Exercise 5.** Our version of `sys_fork()`, with its dirt simple stack copying strategy, works only for simple programs. For example, consider the following function definition:

```
void start(void) {
        int x = 0;  /* note that local variable x lives on the stack */
        /* YOUR CODE HERE */
        pid_t p = sys_fork();
        if (p == 0)
                /* YOUR CODE HERE */;
        else if (p > 0)
                sys_wait(p); // assume blocking implementation
        app_printf("%d", x);
        sys_exit(0);
}
```

In a system with true process isolation, the child process's `x` and the parent process's `x` would be different variables, and changes in one process would not affect the other's `x`. But in MiniprocOS, this is not always the case! For this exercise, produce a version of that code with the following properties:

1. The code uses only local variables.
2. In a system with correct process isolation, the code would print "`10`".
3. In MiniprocOS, the code would print "`11`".

**Hint:** It isn't easy to get this to work because the compiler tends to optimize away important assignment statements or shift them to unfortunate places. Mark a variable as `volatile` to tell the compiler not to optimize references to it. Doing this correctly is tricky, but if you can understand the difference between `volatile int *x` and `int * volatile x` you can do this problem.

**Extra-Credit Exercise 6.** MiniprocOS miniprocesses have some aspects of threads. For instance, like threads, they all share a single address space. A big difference from threads is that we create a new process by forking. New threads are created in a different way. Introduce a new system call,

```
pid_t sys_newthread(void (*start_function)(void));
```

that creates a new process in a thread-like way. The new process should start with an *empty* stack, not a copy of the current stack. Rather than starting at the same instruction as the parent, the new thread should start by executing the `start_function` function: that is, that function's address becomes the new thread's instruction pointer.

**Extra-Credit Exercise 7.** Introduce a `sys_kill(pid)` system call by which one process can make another process exit. Use this system call to alter `mpos-app2.c`'s `run_child()` function so that the even-numbered processes kill off all odd-numbered processes (except for thread 1). Running the application should print out "`Process N lives`" messages only for even-numbered values of `N`.