```
2230    /*
2231     * If the new process paused because it was
2232     * swapped out, set the stack level to the last call
2233     * to savu(u_ssav).  This means that the return
2234     * which is executed immediately after the call to aretu
2235     * actually returns from the last routine which did
2236     * the savu.
2237     *
2238     * You are     expected to understand this.
2239     */
```

CS 111
Operating
Systems
Principles
Winter 2011

WeensyOS Minilab 2

This second of the Weensy OS problem sets introduces you to another weensy operating system. WeensyOS 2 shows off *scheduling* and *synchronization*.

Handing in

You will electronically hand in code and a small writeup containing answers to the numbered exercises.

The problem set code, `weensyos2.tar.gz`, unpacks into a directory called `weensyos2`. (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 `weensyos2.tar.gz` from CourseWeb and unpack it.
2. Do your work in the `weensyos2` directory.
3. Fill out the `answers.txt` file in that directory.
4. When you're done, run **`make tarball`** from the `weensyos2` directory. This will create a file named `weensyos2-`*`yourusername`*`.tar.gz`.
5. Submit that `weensyos2-`*`yourusername`*`.tar.gz` file to CourseWeb.

Part 1: Scheduling

> Please note that you can do Parts 1 and 2 in either order.

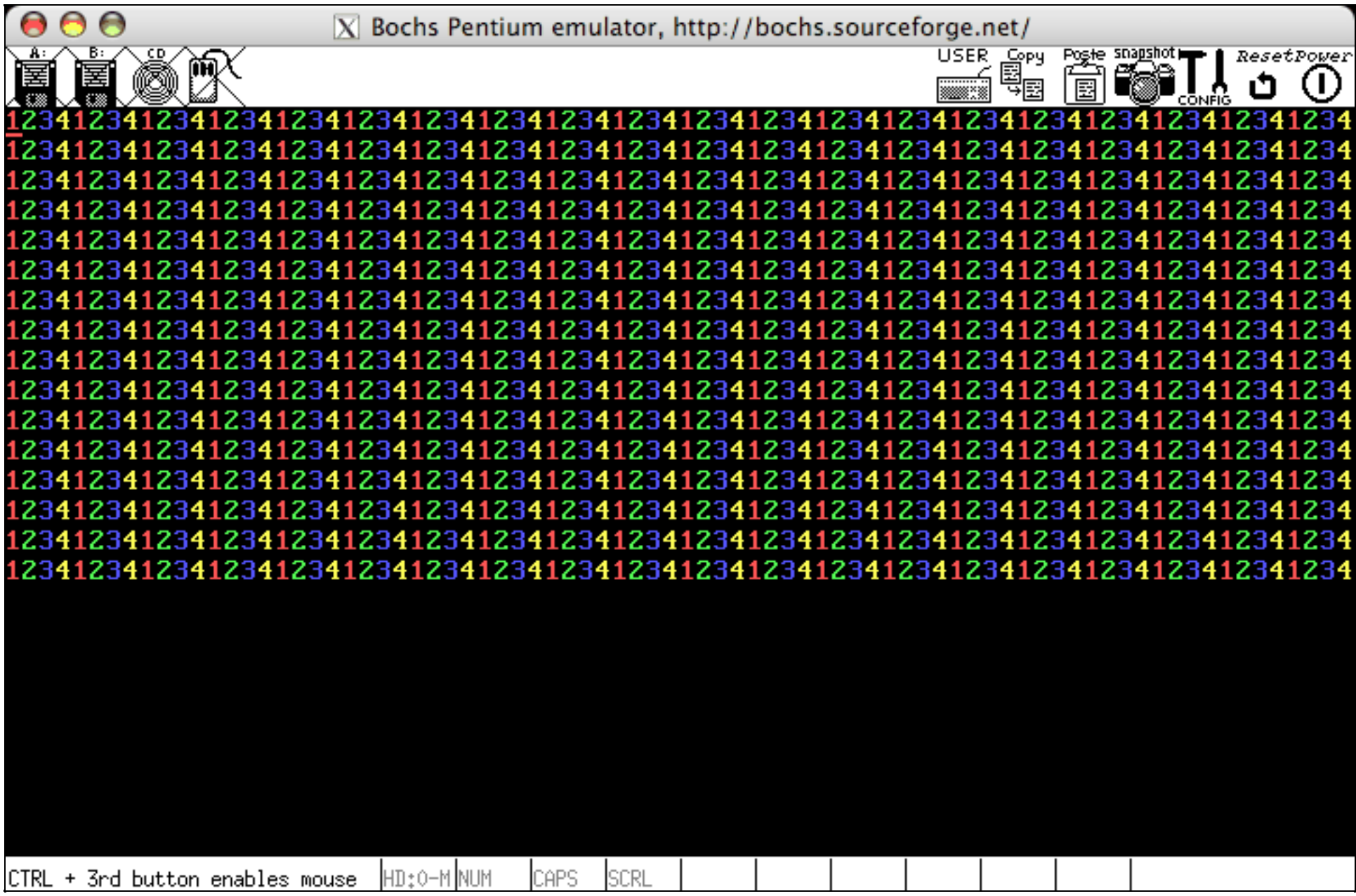
First, you must set up your machine to compile and run WeensyOSes. You can use the setup that worked for you for WeensyOS 1.

> Download and unpack the source for weensyos2.

```
% gtar xzf weensyos2.tar.gz
% ls weensyos2
COPYRIGHT     lib.c          schedos-4.c      schedos-loader.c    x86sync.h
GNUmakefile   lib.h          schedos-app.h    schedos-symbols.ld
answers.txt   mkbootdisk.c   schedos-boot.c   schedos-x86.c
bootstart.S   schedos-1.c    schedos-int.S    schedos.h
conf          schedos-2.c    schedos-kern.c   types.h
elf.h         schedos-3.c    schedos-kern.h   x86.h
%
```

Change into the `weensyos2` directory and run **`make run`**.

This builds and runs the WeensyOS 2 operating system, the "scheduler OS" or SchedOS. As before, this will start up Bochs. After a moment you should see a window like this:



The SchedOS consists of a kernel and four simple user processes. The `schedos-1` process prints 320 red `"1"`s, the `schedos-2` process prints 320 green `"2"`s, and so forth. Each process yields control to the kernel after each character, so that the kernel can choose another process to run. Each process exits after printing its 320 characters. The four processes coordinate their printing with a shared variable, `cursorpos`, located at memory address 0x198000. The kernel initializes `cursorpos` to point at address 0xB8000, the start of CGA console memory. Processes write their characters into `*cursorpos`, and then increment `cursorpos` to the next position.

Read and understand the SchedOS process code. Specifically, read and understand `schedos-1.c`.

Read and understand the comments in `schedos-app.h`. The basic feeling should be familiar to you from WeensyOS 1.

The kernel's job is very simple. At boot time, it initializes the hardware, initializes a process descriptor for each process, and runs the first process. At that point it loses control of the machine until a system call or interrupt occurs. System calls and interrupts effectively call the kernel's `interrupt` function. Note that this simple kernel *has no persistent stack*: every time a system call occurs, the kernel stack starts over again from the very top, and any previous stack information is thrown away. Thus, all persistent kernel data must be stored in global variables.

Read and understand the following pieces of kernel code. Again, don't worry about every last detail; just get a feeling for the high-level structure and purpose of each function.

1. The process descriptor structure `process_t` defined in `schedos-kern.h`. This is a lot like the process descriptor structure from WeensyOS 1.
2. The comments at the top of `schedos-kern.c`.
3. The `start` function from `schedos-kern.c`, which initializes the kernel.
4. The `interrupt` function from `schedos-kern.c`, which handles all interrupts and system call traps.

SchedOS supports two system calls, `sys_yield` and `sys_exit`. The `sys_yield` call yields control to another process, and `sys_exit` exits the current process, marking it as nonrunnable. The kernel implementations of these system calls (in `interrupt()`) both call the `schedule` function. This function is SchedOS's scheduler: it chooses a process from the current set of runnable processes, then runs it. In the first part of this problem set, you'll focus on this function, and SchedOS's scheduling algorithms.

Read and understand the `schedule` function from `schedos-kern.c`.

Exercise 1. What is the name of the scheduling algorithm `schedule()` currently implements? (What is `scheduling_algorithm 0`?)

Exercise 2. Add code to `schedule()` so that `scheduling_algorithm 1` implements strict priority scheduling. Your implementation should give `schedos-1` higher priority than `schedos-2`, which has higher priority than `schedos-3`, which has higher priority than `schedos-4`. Thus, process IDs correspond to priority levels (assuming that numeric priority levels are defined as usual, where smaller *priority levels* indicate higher *priority*). You will also need to change `schedos-1.c` so that the `schedos` processes actually exit via `sys_exit()`, instead of just yielding forever. Test your code.

Please note although SchedOS's current processes never block, your scheduler must work correctly even if processes blocked and later became runnable again.

Exercise 3. Calculate the *average turnaround time* and *average wait time* across all four jobs for `scheduling_algorithm`s 0 and 1. Assume that printing 1 character takes 1 millisecond and that everything else, including a context switch, is free.

Now complete *at least one* of Exercises 4A and 4B.

Exercise 4A. Add another scheduling algorithm, `scheduling_algorithm 2`, that acts like `scheduling_algorithm 1` except that priority levels are defined by a separate `p_priority` field of the process descriptor. Also implement a system call that lets processes set their own priority level. If more than one process has the same priority level, your scheduler should alternate among them.

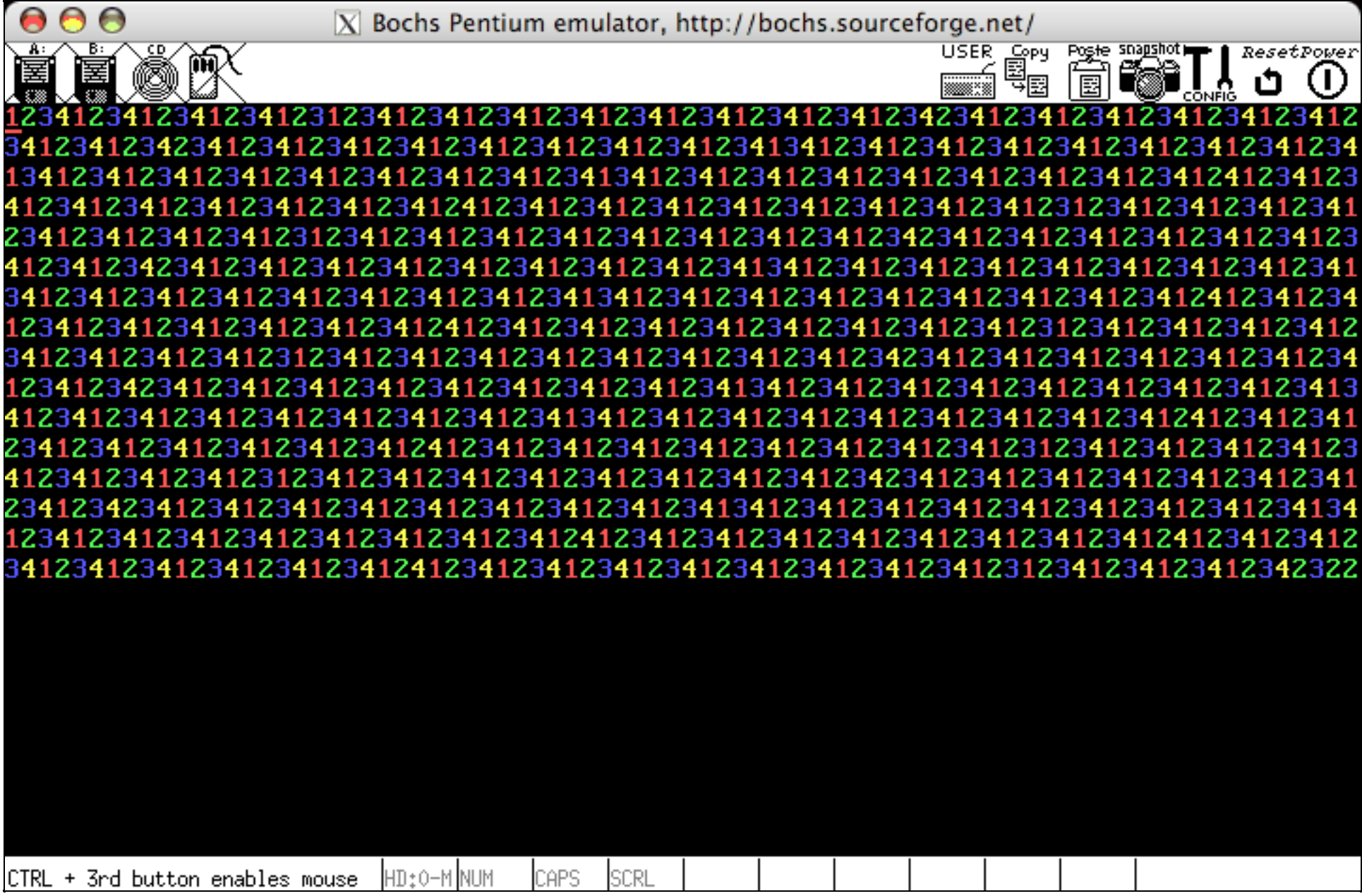
Exercise 4B. Add another scheduling algorithm, `scheduling_algorithm 3`, that implements proportional-share scheduling. In proportional-share scheduling, each process is allocated an amount of CPU time proportional to its share. For example, say `schedos-1` has share 1 and `schedos-4` has share 4. Under proportional-share scheduling, `schedos-4` will run 4 times as often as `schedos-1` (at least until it exits); so we might expect to see output like "`441444414444144...`". (Note that this is a form of priority scheduling, but the priority levels are defined differently: larger shares indicate *higher* priority.) Also implement a system call that lets processes set their share.

Part 2: Synchronization

In this section of the problem set, you'll investigate synchronization issues. But synchronization isn't interesting without concurrency, and right now, our operating system is cooperatively multithreaded: each application decides when to give up control. We introduce concurrency by turning on *clock interrupts* and introducing *preemptive multithreading*. When a clock interrupt happens, the CPU will stop the currently-running process -- *no matter where it is* -- and transfer control to the kernel. This indicates that the current process's *time quantum* has expired, so the kernel will switch to another process. However, note that clock interrupts *will never affect the kernel*: this simple kernel runs with interrupts entirely disabled. Interrupts can only happen in user level processes.

Change `scheduling_algorithm` back to 0. Then change the `interrupt_controller_init(0)` call in `schedos-kern.c` to `interrupt_controller_init(1)`. This turns on clock interrupts.

After running **`make run`**, you should see a window like this:



Clock interrupts are occasionally preempting SchedOS processes, breaking up the steady round-robin order.

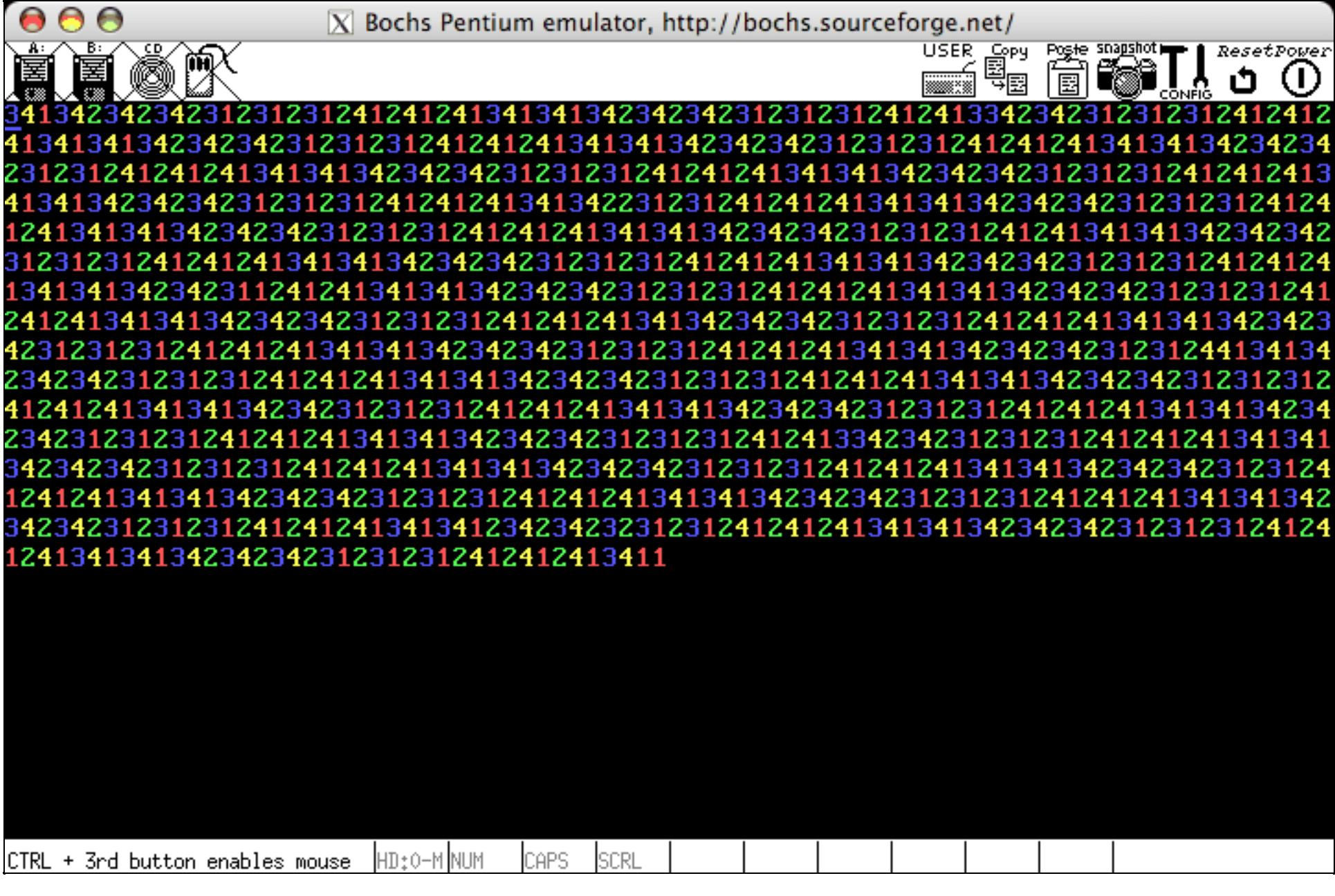
> **Note:** It is better to do this portion of the lab using bochs. Qemu's interrupt handling is too fast; you may not be able to observe the same interrupt effects on qemu.

> **Exercise 5.** Refer to the picture above (not your own SchedOS, which may differ). During the execution of which process does the first known timer interrupt occur? Explain your answer.

But we're not done! Let's cause clock interrupts to happen a little bit more frequently.

> The `HZ` constant in `schedos-kern.h` equals the number of times per second that the clock interrupts the processor. It is set to 100 by default, meaning the clock interrupts the processor once every 10 milliseconds. Set this constant to 1000, so that the clock interrupts the processor every milliscond.

After running **`make run`**, you should see a window similar to this:



Note that the output has less than 320*4 characters! Clearly there is a race condition somewhere! (Your particular output may differ. You may actually see 320*4 characters *with occasional gaps*, which demonstrates a related race condition. If you still see 320*4 characters with no gaps, try raising `HZ` to 2000 or 3000.)

> **Exercise 6.** Implement a synchronization mechanism that fixes this race condition. Your code should always print out 320 * 4 characters with no spaces. (However, it is OK for the ordering of characters to vary. For instance, you might end with a string of the same character, depending on precisely how timer interrupts arrive.)

There are lots of ways to implement the synchronization mechanism; here are a couple.

- Implement a new system call that atomically prints a character to the console.
- Use the atomic operations in `x86sync.h` directly.
- Use the atomic operations in `x86sync.h` to build a lock data type, then use `lock_acquire` and `lock_release` operations. **Note that all four processes must share the same lock!** It does no good to implement a different lock object per process.
- Implement new system calls that provide `lock_acquire` and `lock_release` operations.

However, you must not turn off clock interrupts. That would be cheating. Some hints:

- You may need to use typecasts to get the `x86sync.h` atomic operations to work.
- Note that `cursorpos` points to a 16-bit integer, so the C statement `cursorpos++;` actually increments the address stored in `cursorpos` by **2** bytes, not one.
- If you create a lock object, make sure that all four processes *share* a single lock object. (There's no critical section if each process uses a private lock!) You can tell each lock's address by looking in

the `obj/schedos-[1-4].sym` files, which tells you where each symbol is located. Note that `cursorpos` has the same address in each process.

This completes the minilab.

Extra-Credit Exercise 7. Implement another interesting scheduling algorithm—for example, lottery scheduling or multilevel queue scheduling (Google for more information). Explain how your scheduling algorithm is supposed to work, describe any new system calls you supplied, and code the algorithm with a new `scheduling_algorithm` value.

Extra-Credit Exercise 8. Implement more than one synchronization mechanism for printing characters. Use preprocessor symbols so that your code can be compiled with either mechanism.