CS 111 Discussion Notes—Minilab 2
Updated 15 May 2013

What the WeensyOS2 assignment is

SchedOS is another small operating system, like MiniprocOS. The assignment has two parts. The first part concerns the scheduler, and comprises exercises 1-4 (and optionally 7.) You will write some scheduling algorithms and answer some scheduling-related questions. It is a good idea to review the corresponding sections (5.5 and 6.3) in the textbook.

The second part comprises exercises 5 and 6 (and optionally 8) and focuses on synchronization. The kernel uses clock interrupts to interrupt processes, and this leads to a race condition. You will use one or more of the suggested methods to fix this race condition.

You should not need to modify the application programs (schedos-?.c) except for exercises 6 and 8.

Remember, the minilabs are **individual work only**—no groups. (You can share ideas but not code.)

What you need to do for WeensyOS2

1. **Read and understand:**
    1. The lab assignment
    2. schedos-1.c (and 2-4)
    3. The comments in schedos-app.h
    4. The process descriptor structure in schedos-kern.h
    5. In kern.c:
        1. The comments at the top
        2. The start(), interrupt(), and schedule() functions
2. **Solve** the exercises:
    1. A quick question. It is a good idea to **review** sections 5.5 and 6.3 in the textbook.
    2. Implement strict priority scheduling.
    3. Calculate metrics for the scheduling algorithms. **Be precise! Show your work!**
    4. Two options, or both for extra credit:
        1. Update priority scheduling to let processes change their priorities.
        2. Implement proportional share scheduling.
    5. Another question. **Be careful** in explaining your answer!
       Remember to go back to algorithm 0 and turn clock interrupts on for exercise 6.
    6. Implement a synchronization mechanism to fix a race condition. The suggested methods are:
        1. Implement a system call to perform the entire critical section. The kernel doesn't interrupt itself, so during the call the kernel doesn't get interrupted.
        2. Write system calls that implement lock_acquire() and lock_release() operations, and use them.
        3. Use an atomic operation in x86sync.h, either directly or as part of a lock datatype.
    7. Extra credit: implement another scheduling algorithm.
    8. Extra credit: implement a second synchronization mechanism.

The (updated) SchedOS code defines some preprocessor symbols and names to use when implementing the exercises. It would be much appreciated if you would use these names instead of others. They are:

1. in schedos-1.c : `__EXERCISE_8__` (for using exercise 8 code instead of exercise 6 code)
2. in schedos-app.h : `sys_priority` and `sys_share` (system call names for exercises 4.a and 4.b)
3. in schedos-kern.c :
    1. `__PRIORITY_*__` (for initializing priorities for processes in 4.a)
    2. `__SHARE_*__` (for initializing shares for processes in 4.b)
    3. `__EXERCISE_*__` (for switching between different scheduling algorithms)

Notes on the code

schedos-1.c: Lines 7–13 describe what the schedos-?.c programs do. Lines 17–19 give the symbol and color that is printed. The loop in lines 36–40 is run RUNCOUNT (= 320) times.

Question 1: Is the statement at line 38 atomic? Why or why not?

schedos-symbols.ld: This is where the cursorpos shared memory variable is instantiated. It uses bytes 0x198000 and 0x198001. If you need to instantiate other shared memory variables, add them here.

In SchedOS is that all processes agree that shared memory uses the byte range [0x198000, 0x200000). However, all memory is technically "shared" because there are no access controls.

schedos-app.h: This is where RUNCOUNT and the system calls are defined. If you write more system calls, put them in this file. You can copy and paste from these system calls or the calls in MiniprocOS to use as starting points for your own system calls.

schedos.h: This file defines the set of interrupt numbers. You may run into problems if you define numbers larger than 51. If this happens, you can either pass more parameters to system calls to help distinguish them, or use #defines to use the numbers for different exercises. There may also be better ways to fix the problem. In any of these cases, document any fixes in your answers.txt file.

The second important item in this file is the declaration of cursorpos. If you add more symbols to schedos-symbols.ld, also declare them here.

schedos-kern.h: lines 14–21 define the process state types, which are identical to those of MiniprocOS. Lines 24–33 describe the process structure. You may need to make changes to it. The symbol HZ is defined at line 37. You will need to change this for exercise 6. The rest of the file is similar to mpos-kern.h from MiniprocOS.

schedos-kern.c top matter (lines 1–73): This code defines: the number of processes (5 = #0 + four actual processes); the location of the bottom of the first process's stack and the size of each process stack; an array of process descriptors (again, process 0 is always empty); the current pointer; and an integer for the scheduling algorithm. Also defined are preprocessor symbols to use for exercise 4 and scheduling algorithm numbers to use for the algorithms that you write.

schedos-kern.c start function (lines 84–138): This function first sets up the hardware. It initializes memory segments, turns off clock interrupts (line 91), and clears the console. Lines 95–117 initialize the process descriptors and load the four schedos-?.c programs. Line 121 sets the initial cursor position. Line 130 sets the scheduling algorithm to use (currently the default algorithm 0). Lastly, line 133 starts the first program running.

schedos-kern.c interrupt function (lines 153–200): This function handles interrupts, in much the same way that it handled them in MiniprocOS. One difference is the last case, INT_CLOCK, which is a hardware interrupt that is called HZ times per second. You can add handlers to this function for any system calls that you write.

schedos-kern.c schedule function (lines 217–237): This is where scheduling algorithm 0 is defined, and where you will add new scheduling algorithms.

x86sync.h: This file defines some synchronization primitives that you can use, either directly or in building larger locks. The first, `atomic_swap(addr,val)`, at once sets `*addr = val` and returns the old value that was stored at `addr`. The second, `compare_and_swap(addr,expected,desired)`, at once returns the current value stored at `addr` and conditionally updates it to equal `desired` (if `*addr` was equal to `expected`). The third, `fetch_and_add(addr,delta)`, at once adds `delta` to `*addr` and returns the old value of `*addr`.