HW3: Multitasking Tiny-UNIX on the SAPC

Assigned: 23 March 2021 **Due:** class time 13 April 2021

1. Introduction

The objective of this assignment is to implement a multitasking Tiny-UNIX on the SAPC. The students are asked to expand the Tiny-UNIX kernel in hw2 to multitask three programs, each of which will only make writes (to TTY0 or TTY1) and exit system calls.

In this assignment, the console output port (COM2) is the resource shared among the three programs. As in hw2, the output is interrupt-driven. This is a producer-consumer situation, with 3 producers filling the output buffer of COM2 and one consumer, the transmit interrupt handler, outputs the characters. When the output buffer is full, the processes will block in the "write" operation, giving up the CPU for other processes to use. This eliminates spin loops in the kernel.

The students will make use of the provided CPU context switching program (asmswtch.s) in writing a non-preemptive scheduler (one which never grabs the CPU away from a process which can be run) to control the running of the processes. Process switching will only occur when a process blocks (for output) or exits. After a process exits, it is a "zombie" in UNIX parlance: all that is really left of it is its exit value, waiting to be picked up by its parent. Since there is no parent process, the students will simply print out the exit values with the shutdown message of the kernel, after the completion of all three processes.

2. Discussions

The user program (here a bundle of 3 user programs) to be run on the SAPC has to be built in the same way as in hw2, downloaded and run. As in hw2, the user process is running in supervisor mode, sharing a common stack area between user and kernel for each process. There will be separate stacks for each of the three processes. The program begins with startup0, then startup1. Then it calls the kernel initialization, which sets up the three processes' initial data. We will follow UNIX in reserving process 0 for the kernel itself, and require that it never block. It is ready to run at all times and if no user process has anything to do, the kernel process just loops until it exits.

A common process entry data structure, PEntry, is used to keep track of the process data:

```
typedef struct {
  int p_savedregs[N_SAVED_REGS];      /* saved non-scratch registers */
  ProcStatus p_status;                /* RUN, BLOCKED, or ZOMBIE */
  WaitCode p_waitcode;                /* valid only if status=BLOCKED: TTY0_OUT, etc. */
  int p_exitval;                      /* valid only if status=ZOMBIE */
} PEntry;
```

In the beginning, the processes' statuses are marked as RUN. Their saved-pc entries are initialized to ustart1, ustart2 or ustart3, respectively; their saved-esp entries to three different stack areas; and their savedebp field to 0 (it controls backtrace during debugging).

Once the initialization phase is complete, the process 0 settles down to a loop calling the scheduler, trying to find and schedule a user process that can be run. But, if there are none, it keeps looping. The loop ends when the number of ZOMBIE processes equals the number of user processes. In addition to calling the scheduler, it turns on interrupts in the loop and then off again (if interrupts are never turned on, the queue can't be drained and blocked processes will remain blocked). The scheduler should be called with interrupts off, as it is clearly critical section code. Once the main loop ends, process 0 should go into a wait loop to make sure the output queue drains and then do some finishing up, printing the exit code of each user process.

The scheduler makes use of the supplied asmswtch.s to perform a process switch. We'll go over the assembler code in lecture. This will in turn calls the respective C user module with entry point main1, main2, or main3. The syscalls in the user code cause execution of the kernel, eventually returning to the user code. However, when ttywrite encounters a full output buffer, it blocks the process by calling a function, sleep(OUTPUT), in the scheduler(status changed from RUN to BLOCKED). The scheduler saves that process context and chooses another process to run; using the assembler asmswtch.s code to do the tricky stuff. When asmswtch returns, the system should be running the new process.

As the output drains at the interrupt (i.e., via the interrupt handler) and a spot becomes available in the output queue, a wakeup(OUTPUT) function is invoked which will run the processes that have been blocked on waitcode=OUTPUT (status changed from BLOCKED to RUN)

Finally the user code does a sys call exit. The kernel sets the process state to ZOMBIE and calls schedule and that finds another process to run.

3. Files

Copy the files from hw2/soln/ and move them to your hw3 directory. Add the files provided in my hw3 directory to yours.

<u>Shared between user and kernel:</u>
tsyscall.h: syscall numbers, as in hw2
tty_public.h: TTY device numbers, as in hw2

<u>Kernel files:</u>
ioconf.h, tty.h: I/O headers used only by kernel, as in hw2
tsystm.h: syscall dispatch, kernel function prototypes as in hw2
startup0.s, startup.c, sysentry.s: assembly & C startup, syscall handler, as in hw2
tunix.c: kernel startup, shutdown, C trap handler, much like hw2
sched.c and sched.h: scheduler code: functions schedule, sleep, wakeup.
asmswtch.s: process switch, in assembly, provided.
ioconf.c, io.c: device independent I/O system, as in hw2
tty.c: terminal driver, from hw2 but changed to block, unblock

<u>User-level files:</u>
tunistd.h: syscall prototypes, as in hw2
crt01.s: as in hw2 crt0.s, but calls main1, starts at ustart1
crt02.s, crt03.s: calls main2, main3, starts at ustart2, ustart3.

uprog1.c uprog2.c, uprog3.c: tiny-UNIX programs starting at main1, main2, main3
ulib.s: same as hw2

<u>makefile</u>: name the executable tunix.lnx. Be sure to add a new rule for sched.o and update the rule for tunix.o to reflect their header inclusions. Add new rules for other new files too.

4. Suggested steps

1. Make sure your hw2 solution is fully working, or use the provided solution. Note that the provided solution has a "debug_log" service that writes notes to memory, and prints out the log when the kernel shuts down. Add this logging functionality to your own solution if you're using it for hw3. Note that you can add your own entries in the log to figure out what's happening.

2. Write trivial user programs that just kprintf a message, and the crt0's for them that call mainx (where x=1, 2, or 3), and then do an exit syscall. Write a fake scheduler that just loops through the process entry table until it finds a status = RUN process and then calls it at ustart1, 2 or 3. If there aren't any processes with status=RUN left, bring down the system. Inside sysexit, call the scheduler after marking the process ZOMBIE. This simple system will work because each process only needs to run once. They use the same stack area one after another. Make sure your makefile is right: see that it rebuilds the right things after each edit.

3. Now use the supplied asmswtch code to start each process. For each user process, you need to initialize the PEntry's saved-pc to ustartx ((where x=1, 2, or 3), and the saved-eflags to allow interrupts, and the saved-esp to the proper stack. Also the saved-ebx should be 0. In your new fake scheduler, loop through the process entry table until you find a RUN user process (1, then 2, then 3) and then call asmswtch.s with old-process = process 0, new-process = proc 1, to switch from proc 0 to proc 1, thus running uprog1. Later, when proc 1 exits, sysexit calls the scheduler again, which finds proc 2, and calls asmswtch.s with pointers for (proc1, proc2) to switch from proc 1 to proc 2, and that runs uprog2. When uprog2 exits, sysexit calls the scheduler... Eventually the scheduler calls asmswtch.s with (proc3, proc0) to switch back to proc 0 when no more RUN processes exist. Then you can shut down the system using process 0. (Of course you can do just part of this to start, then write a more finished product.) Make sure the user processes are each using their own stack.

4. Add a debug_log call to the scheduler to report on each process switch, for example, "|(2z-3)" for a switch from process 2, a zombie, to process 3.With just kprintf's in your user programs, your debug log should look like this now:

|(0-1)|(1z-2)|(2z-3)|(3z-0)

5. Add writes of one or two chars each to the three user programs and get this working--this output all fits in the 6-char queue, so no blocking is needed. This should work even though you haven't yet edited tty.c. Make sure the kernel waits for the output to drain while it's shutting down. Also make sure the user processes are running with interrupts enabled and using their appointed stack. You can check IF with get_eflags(), for example. Since the three writes return immediately, all the output will be done in the final kernel wait-for-output, and the debug log will look something like this (for 2 output chars for each process):

|(0-1)~|(1z-2)~|(2z-3)|(3z-0)~~~~~s

(Possibly more ~s will come earlier.)

6. Write the real scheduler after the class on this subject. Get it working in this easy user program environment. Change the code in tty.c to block and unblock at the right moments. Change to a 7-char output from one user program as soon as you want to try out blocking. When that works, try two, then all three with over-6 chars. Then try it with the provided uprog's.

The following files are the deliverables for this assignment:
README--authorship
tsystm.h--possibly new prototypes here
tunix.c--kernel initialization, shutdown, process 0 added
sched.c and sched.h--scheduler
tty.c--calls to sleep and wakeup added
crt01.s, uprog1.c
crt02.s, uprog2.c
crt03.s, uprog3.c