HW2: Tiny-UNIX on the SAPC

1. Introduction

The objective of this assignment is to implement a tiny-UNIX with 3 services: read, write, and exit on the SAPC. The students are asked to convert the read, write functions in hw1's I/O package to system calls. In addition, they are asked to write the new service exit. The prototypes for the services are:

```
int read(int dev, char *buf, int nchar);     /* read nchar bytes into buf from dev */
int write(int dev, char *buf, int nchar);    /* write nchar bytes from buf to dev */
int exit(int exitcode);                      /* exit will return back to Tutor */
```

The testio program in hw1 can be modified to test the 3 services. Use a script file to capture the outputs.

2. Discussions

This assignment builds on your standalone I/O package from hw1. If you prefer, you can use the hw1 solution, which is included in the provided hw2 directory. In any case, fix yours, if necessary, up to spec first.

To make system calls, the user program needs to execute a trap instruction to transfer control to the kernel and the kernel's trap handler will perform the service. The x86 Linux syscall linkage is as follows:

--int 0x80 is the syscall instruction
--the syscall # is in eax
--the syscall args are in ebx (first), ecx (second), and edx (third).

Here are the files you need ($pclibsrc = /home/cheungr/serl/tutor-linux/libc):

i) <u>Shared between user and kernel:</u>

tty_public.h: device numbers
tsyscall.h: syscall numbers (see a sample system call numbers in [Linux/i386 system calls])

ii) <u>Kernel files:</u>

ioconf.h, tty.h: i/o headers used only by kernel
tsystm.h: syscall dispatch, kernel function prototypes (a copy is posted)
startup0.s: this file is provided. Sets up stack, calls into startup1.c
startup.c: this file is provided, but you need to modify it to call your kernel initialization routine instead of main.
tunix.c: has kernel init routine. It needs to call ioinit, call set_trap_gate(0x80,&syscall), and possibly other inits. tunix.c also has the code for syscallc and sys call exit. The code for set_trap_gate has been provide in libc/cpu.c and built into the libc library

syscallc: first write it with a big switch statement over the various different syscalls. If you have time, upgrade it to use a sysent dispatch table.

sysentry.s: Trap handler's assembler envelope routine syscall for the trap resulting from a system call--needs to push eax, ebx, ecx, edx on stack, where they can be accessed from C, call syscallc, then pops, iret. Use $pclibsrc/irq4.s as an example and modify it to fit your needs.

io.c: rename "read" to "sysread", etc. to avoid linking problems, since "read" is a now user-level call.

ioconf.c, ioconf.h: from hw1.

tty.c, tty.h, tty_public.h: tty driver from hw1, unchanged

iii) <u>User-level files:</u>

tunistd.h: prototypes for user mode system calls (like UNIX /usr/include/unistd.h)
test1.c: has main(). Easily extended to multiple user files, or user assembler sources, as long as they follow the syscall rules and have a main entry point. First example is
main() { write(TTY1,"hi!\n",4);}.
Work back to testio.c from hw1.
ulib.s: library set-ups for syscalls: read, write, exit. Provided for write, you add read and exit.
crt0.s: user-level "C startup module" sets up stack, calls main, does exit syscall. Entry point ustart.

iv) <u>hw1 solution files not directly used in hw2</u>:
io_public.h: like tunistd.h above, but also lists ioinit(), and not exit().
testio.c: remove ioinit() call here to turn into proper user program.(and note that the kprintf's are only for debugging)

v) <u>Make file</u>

makefile: The provided makefile can make a hw2 system by "make U=test1" to use test1.c as a user program. The default user program is uprog.c and it can be build by entering “make”.

In the hw2 directory, empty files (file with length =0) are provided and you can use them to test the makefile. (Empty files are valid .c and .s programs and make treats them as regular files). If you try "make U=test1" with the provided files, you should see compiles followed by a load with an error as follows:

```
…
io.c: In function `write':
/home/cheungr/cs444/hw2/io.c:50: multiple definition of `write'
ulib.o(.text+0x0): first defined here
….
```

This happens because both ulib.s and io.c define global symbols named "write". You need to change write to syswrite in io.c to fix this. We want to use "write" for the user-level system call, so the kernel needs another name for its function. syswrite is the Linux kernel name for its write-implementing function, so let's adopt that name.

3) The Finished Program

The idea here is that each user program to be run on the SAPC has to be separately built with tunix, downloaded and run. Startup0 executes first, and transfers control to the kernel initialization in tunix.c, which sets up the system and starts the user code at ustart (calling ustart will do the trick). The C user startup module reinitializes the stack and calls main. The syscalls in the user code (in ulib.s, called from test1.c) cause execution of the system call handler in tunix.c (and functions called from there), returning to the user code in ulib.s at the iret. Finally the user does a syscall exit. The kernel gets control, and finishes up.

4) Suggested Steps

There are lots of little pieces to this system. Here is a suggested sequence to follow:

1. Create a simple test1.c and get a system built. Change io.c's write to syswrite, read to sysread, fill out startup0.s and startup.c, write a tiny tunix.c init function that calls ioinit, then calls main (cheating for now--later it should call ustart), and then returns to startup, shutting down. At this point, test1.c just has a main that kprintf's a message. Now it should build and run, but does no syscalls.

2. Make test1.c do a simple write. ulib.s is set up for write already, so do the write syscall first. Set the trap vector up in kernel init in tunix.c, and write sysentry.s—have it push registers on the stack and then call into tunix.c. In tunix.c, access the pushed registers (themselves syscall arguments from the user) via args to the C function, and call syswrite. We'll go over the trick about the syscall args in class.

3. Next add an exit to test1.c and implement syscall exit.

4. Write the proper user startup module crt0.s to reinitialize the stack and call into main, then when that returns, it does an exit syscall. It has entry point ustart. Change the call to main in kernel initialization to call ustart now. Try a user program without its own exit syscall. Modify the final test1.c to perform the same operations as in testio.c and create a typescript file that captures its execution. Please include the lnx file and the typescript file as a part of your deliverables.