CS 170, 1.22.25

Ryan Wenger

PART 1 – KOS

Origins of KOS

- KOS is the name of the kernel you will be building over the course of this quarter
- It's based on ULTRIX, a flavor of UNIX that was built and shipped in the 1990's by Digital Equipment Corporation (DEC) for their line of MIPS workstations, called the DECstation (though it was later ported to DEC's in-house VAX architecture)
- Crucially, ULTRIX and early Linux were around at the same time—resulting in the existence of a 32-bit Linux-hosted cross compiler (decstation-ultrix-gcc/g++/ld/etc.) that can build C programs to run on ULTRIX
  - This is GCC version 2.95, which supports nothing newer than the ANSI C89 standard. If your programs are behaving weirdly or you're getting nonsensical compiler errors, make sure you aren't doing anything too fancy in your test code.
  - GNU Binutils v2.9.1 are also present in the same directory as GCC (objdump, gas, ar, etc.)
- By implementing a subset of the ULTRIX system calls, you will be able to run such programs on our simulator to test the functionality of your own KOS kernel
  - See the provided Makefile in the lab handout

PART 2 – THE SIMULATOR

The simulator and your KOS kernel are the *same* program.

- When you run KOS on CSIL, you are executing a regular x86 Linux program. Linux sees no difference between your kernel and e.g. or firefox, or any C++ program you run yourself (other than the fact it's 32-bit).
- The simulator that we provide you is nothing more than a collection of .o files (some of which are packaged into .a archives). Inside of one of these is a main() function that eventually calls KOS()—the entry point of the kernel, which you should write.
- In a way, you can think of your own KOS code as library dependency needed to run the simulator.
- Understanding this is crucial to understanding the most fundamental difference between KOS and any real-world kernel.

A Closer Look

Simulator

- Kernel does not run on hardware—in fact, your kernel is compiled using normal Linux gcc (in 32-bit mode)
- This means you can #include <stdlib.h> and call, for example, malloc()
- Helpfully, this means you can use printf() and GDB to debug your kernel like a regular program
  - Don't use printf/scanf to bypass the simulator's ConsoleRead/Write in your submission—you won't get credit and your kernel will behave incorrectly

Real Machine

- Kernel is responsible for managing hardware, and interfacing directly with firmware
- A real kernel must implement its own memory allocator—no code is accessible other than what is contained in the program

Key Things to Recognize in your Mental Model

- Abstractly, the goal of any kernel is to take the mismash collection of asynchronous hardware devices that make up a machine and unite them into some synchronous-looking interface that can be used by a user's programs.
  - This interface is the kernel's System Call API
  - Luckily, you don't have to design this API, you just have to implement the one used by Linux/UNIX/ULTRIX/POSIX/etc.
- The simulator calls your code in 3 ways:
  - At startup
  - When an exception occurs (system call, segfault, etc)
  - On a device interrupt
- Crucially, on real hardware, the latter 2 of these can occur at any moment in the kernel's execution. **That is not the case with KOS**.
  - Because the simulator simply makes "library" calls out to your kernel, it can only call your exception and interrupt handlers when the kernel returns control to it via run_user_code()
  - Thus, it is impossible for your code to jump from the middle of a system call routine into an interrupt handler (unless you explicitly do so).

PART 2.5 – KT THREADS

Quick Summary of Using KT Threads

- If you're familiar with many languages' concept of async/await or "coroutines," then you have a working knowledge of kt threads:
  - kt_fork = spawn async task
  - kt_join = await given task completion and work on other in-progress tasks in the meantime
  - kt_yield = let other tasks make progress until the scheduler decides its my turn again
- Kt provides concurrency but *not* parallelism (i.e., only one kt thread actively runs on the CPU at a given time, but kt scheduler multiplexes between them such that multiple can still make progress)
- Like pthreads, can be created, joined, suspended, and completed (kt_{fork,join,yield,exit})
- Unlike pthreads, are 100% cooperatively scheduled: a kt thread can *only* be suspended at explicit scheduling points, i.e. by calls to kt_fork(), kt_join(), kt_yield(), etc. **as well as** P()/V() operations on kt_sem variables.
  - **No preemption can occur**—your code will not halt in middle of one thread and transition to another unless you explicitly invoke a scheduling point.

PART 3 – ARCHITECTING YOUR KERNEL

The Scheduler

- Your kernel's scheduler is the heart of the entire system—it decides what MIPS code gets run, when it gets run, and (later on) how long to run it for.
- The key thing to recognize is that, after KOS initially starts up, it doesn't actually *do* anything until a user space program invokes it (whether by system call or segfault/etc)
  - Thus, once it's done handling whatever actions have been generated by the user's program, it just waits for something else to happen.
- In future labs, the scheduler will have to juggle multiple processes (programs)—some of which may be waiting for I/O interrupts and thus cannot yet continue, some of which can continue and are waiting for their turn to execute, and (at most) exactly one who is currently executing MIPS code in the simulator

The Console

- The simulator's only means of sending information to/from your terminal ("Console") is the console_read() and console_write() functions. Each operates on 1 character at a time.
- Each of these is associated with an interrupt: a ConsoleReadInt occurs when there is data ready to be read by console_read(); ConsoleWriteInt occurs when the previous call to console_write() has completed and the console is ready to put out another character.
- Have ReadInt handler wake a dedicated kt thread which calls console_read() and pushes result to a queue. A semaphore can track the amount of data in queue: V() on push and P() on pop. read() syscall then pops from queue, waiting for data if needed. To guard case where queue becomes full, either use a 2nd kt_sem to track # of free slots or make buffer size very large.
  - (hint: having a good implementation of this kind of semaphore-controlled queue data structure will come in very handy during lab 3)

System Calls

- Because KOS must be able to handle multiple in-progress system calls (i.e. one process makes a syscall while another is still waiting for its own to complete), every system call handler should execute in its own kt thread.
  - This means arguments must be marshalled to and from each handler invocation. Consider writing a macro to streamline this, as you'll eventually end up with 19 different calls, and copy/pasting the same marshall/unmarshall chunk can get annoying

- Here's part of a set of macros my partner and I used when we took 170. If you're curious about them, come see me in office hours, or feel free to try and fill in the missing ones yourself.

```
/* Use this instead of an actual prototype in system call definitions
 * It matches with the auto declarations in the header so syscalls
 * are compatible with `kt_fork(void * (*) (void *))` and auto unmarshalls
 * the arguments into local variables. It also creates a local variable
 * called `caller` of type `PCB *`.
 *
 * ex: `kssize_t read(int fd, void *buf, ksize_t count)`
 *      becomes
 *      `SYSCALL_DEFINE(read, kssize_t, int fd, void *buf, ksize_t count)`
 */

#define SYSCALL_DECL(name)      void *sys_##name(void * __caller)

#define SYSCALL_DEFINE(name, ret_t, ...)      \
        SYSCALL_DECL(name) \
        {                   \
                PCB *caller = __caller; \
                __VA_OPT__(__SYSCALL_ARG_UNMARSHALL(5, __VA_ARGS__))
```

PART 4 – TESTING

TEST TEST TEST TEST TEST TEST TEST TEST

- testing KOS is absolutely imperative
- There are a handful of programs provided in Rich's cs170 directory on CSIL (~rich/cs170/test_execs), but you will *need* to write your own if you want to ensure a good score on the labs
- Even though KOS runs as an x86 Linux process, the simulator simulates a DECstation machine with a 32-bit MIPS R3000 CPU. You should use and/or modify the provided Makefile.xcomp in the handout to build your test programs
  - This will ensure all of the linker options and transformations necessary to run code on the simulator are applied to the output of the decstation-ultrix-gcc compiler
    - Note: the <unistd.h> header file, which typically contains UNIX system call declarations, is missing from the toolchain; however, the cross compiler "knows" about these functions, so you can call them without including any header files
    - Highly recommend compiling your test programs with decstation-ultrix-gcc and with the Linux host '/usr/bin/gcc –m32 –std=gnu89' so you can compare behavior between your kernel and Linux. Including the '-std=c89' or '-std=gnu89' flag will tell modern GCC to adhere to the same C language standard (from 1989) that the MIPS cross compiler uses, which can be very helpful in catching pesky syntax-related errors that would cause the older compiler to silently emit garbage (e.g: use of "//" style comments).

Final Tips

- There is a “cookbook” (“Heloise's Helpful Hints”) link on the lab handout page that provides nearly everything you need to design and implement the labs.
- For the forward-thinking of you: future labs will have your kernel managing multiple concurrent user processes. Don’t let yourself worry about scheduling yet (kt threads will do most of the heavy lifting anyway), but for now it’s a good idea to encapsulate a process into its own “class” (PCB in the cookbook) that the rest of your code supports multiple instances of.
- When in doubt, ask “what would Linux do?” The manual pages for system calls (e.g. ‘man 2 <system call>’ in terminal or Google) precisely document how they’re expected to behave in edge cases as well as when to return which errors. Compile a test C program for both Linux and KOS, then compare the behavior.
  - The strace(1) command on Linux traces and prints system calls made by a process and may be helpful in some cases as well.
- Go to class. Rich is one of the most knowledgeable and passionate lecturers I’ve ever had during my 6 years at UCSB—don’t put him to waste. He’ll cover specifics about the behavior of every system call you’ll have to implement for the labs this quarter and save you a lot of time digging through documentation or prodding ChatGPT.