# Operating Systems

## SOS Lab 1

# Lab 3 - SOS

- The SOS labs will focus on writing an operating system called SOS
  - For a simulated MIPS machine
- The simulator will call your code
  - When it starts
  - When a user program causes an exception
    - System call, page fault
  - When a device interrupts
- You can’t modify the simulator

# Getting started

- On appropriate (virtual) machines
  - ~/swany/sos-lab1 contains
    - Makefile, exception.c, sos.c, simulator.h
  - ~/swany/sos/lib/main_lab1.o and libsim.a

# SOS

- The operating system initializes through a C-language subroutine that you write called **SOS()**.
- If you were implementing **SOS** for an actual machine, you'd need to define an initial entry point for the hardware to "jump through" at start-up.
- In fact, the "boot up" process is when the hardware loads and executes a pre-defined routine that it finds in a place the hardware specifies (e.g. the boot record).

# SOS

- You will link your code with the two object files, and an executable file will be created.
- When you run the executable, it starts the simulator. Upon instantiation, the simulator calls **SOS()** and your operating system gains control.
- From then on, the interaction between the operating system and the simulator is achieved through well defined communication points.

# Simulator Structure

- The simulator and your code are combined to create an executable that simulates the behavior of a machine and its OS
- Think about assembly instructions that each change a small amount of the machine's state (registers, memory, etc.)
- Think about writing a C program that defines a variable for each piece of machine state and then walks thru an assembly language program one instruction at a time and makes the same state changes that the hardware would have made

# Simulator Structure

- For example, think about some registers and a fictitious instruction “ADD R1 R2 R3”
  - Add the contents of 2 registers and store the results in another
- Imagine defining R{1,2,3} as integers and writing a program in C that keeps track of the register state
- If this is detailed enough, and you read program binaries rather than assembly language strings, then you have a hardware simulator
- libsim.a is just such a thing

# SOS

- The simulator program will call you when the program requires service (which it tells you through an exception) or when a device requires service (through an interrupt)
- You, however, do not return once called
  - You can call **run_user_code()** if you want to run (or go back to running) a user program
  - Or call **noop()** if you want to "idle" the machine
- All exceptions call **exceptionHandler()** passing in an exception type as an argument, and **interruptHandler()**) which gets an interrupt type as an argument.

# The simulated hardware

- 1 CPU with **NumTotalRegs** registers
  - Uses the MIPS instruction set (without floating-point operations)
  - CPU executes in one of two modes: user mode, and "supervisor" (operating system) mode
- To run user code, the OS calls
  - **run_user_code(int registers[NumTotalRegs])**
  - Load registers (including the PC), switch back into user mode and start running at the PC
  - Or, call **noop()** to idle the machine until the next exception or interrupt

# Interrupts and Exceptions

- Both put the OS back in control in the procedures:
  - **exceptionHandler (ExceptionType which)**
  - **interruptHandler (IntType which)**
    - **which** specifies the type of exception or interrupt which are defined in simulator.h
  - You **do not** return from these functions -- you must **run_user_code()** or **noop()**
- Registers are saved when these occur
  - **examine_registers(int buf[NumTotalRegs])**
  - The registers are meaningless when a noop() is interrupted

# Exception Details

- Recall that when a program makes a system call, it is issuing a special instruction called a TRAP
- The OS defines where it expects to find the arguments to the system call when control is transferred via a TRAP
  - Generally some set of registers
- In our case they are r4,r5,r6 and r7
- When the compiler generates code for a system call, it puts the arguments in the appropriate place then calls TRAP
- exception.c says “for system calls, type is in r4, arg1 is in r5, arg2 is in r6, and arg3 is in r7”
  - This is in the exceptionHandler() routine

# Exception Details

- exceptionHandler() doesn't handle many exceptions to begin with -- that's your job
- You can think of this as being handed a skeletal OS that is missing a few modules

# Memory

- A **memory** consisting of **MemorySize** bytes for user programs
  - It is *byte addressable* which means each address is the address of a byte (not an integer or double word)
- The operating system can access this memory starting with the external variable **main_memory**.
  - User programs access this memory starting with address zero. This is very unsophisticated and will be made more realistic as the semester goes on, but for the first lab, it will suffice.
- Initially you will load user programs into memory with
  - **load_user_program(char *filename)**
  - This doesn't execute the program -- it only loads it

# Endianness

- If you use Solaris (on SPARC), you have to be aware that the SPARC and MIPS chips differ in a important way
- The MIPS, and Intel chips are “little endian” and the SPARC is “big endian”
  - This refers to which end of an integer comes first
- Thus the integer 1 (0x00000001) on one looks like 16777216 (0x01000000) on the other
- **examine_registers()** and **run_user_code()** handle this conversion for us
- But, if you set things in main_memory, you should use the functions
  - **int WordToMachine(int i)**
  - **int ShortToMachine(short i)**

# The Console

- The screen and keyboard interact with the CPU through interrupts (recall Chapter 1)
  - In SOS there is a single console device that takes care of both functions
  - The simulator connects this device to **stdin** and **stdout** so you can talk to your simulated system
- Communication uses 2 procedures and interrupts
  - **console_write(char c)** starts a write and causes the interrupt **ConsoleWriteInt** when finished
    - You can't call it again until you get the interrupt
  - When there is a character to read, an interrupt of type **ConsoleReadInt** is generated and the character can be read with **console_read()**
    - If you don't call the read before another interrupt, the char is lost
  - **console_read()** returns -1 when the user types **CNTL-D**.

# Other Stuff

- To model real devices, you can only read or write a character to/from the console about every 100 operations
- For the first lab, there is no timer interrupt in the simulator
- Due to issues in the simulator, don’t use the top 8 bytes of memory
- Also, the MIPS simulator assumes all 4 byte words are aligned
- The GTYPE in the Makefiles and paths refers to the architecture string from GNU Autoconf config.guess (e.g. `i686-pc-linux-gnu`)

# User programs for SOS

- The simulator simulates a DEC Ultrix machine and can run programs compiled for that system
- To create these programs, we have a cross-compiler in /home/swany/xcomp/$GTYPE
  - A version of gcc that runs on Linux and can generate binaries for the DEC MIPS system
- There are example programs and a Makefile in /home/swany/sos/test_execs
- There is a post-processing step that converts the COFF binaries into the simulator object file format called NOFF
  - This is called coff2noff and an example of how to use it to build programs is in the Makefile

# The Environment

- You must link with libsim.a and main_lab1.o
  - Already compiled for Linux
- They combine to provide
  - a machine instruction simulator for the MIPS and DEC Ultrix
  - a simple machine console
  - a way to load a MIPS/DEC Ultrix binary into an array (called **main_memory**)
  - an interface that allows the simulator and your code to interact
- Thus, you can take a simple program, compiled for a DEC MIPS machine, and load it into main_memory, set the PC to the first instruction and the simulator can run the program

# A word on compilation

- A potential point of confusion is the difference between compiling your simulator/OS and compiling the programs that run on it
- You compile your SOS with the system compiler (gcc)
  - The simulator is compiled this way
- There is a special version of gcc that you use to generate the binaries that your SOS will run
  - You can probably just use the provided binaries at this point, but you don’t have to

# Assignment Details - 1

- Look at the initial files given in /home/swany/sos-lab1 -- in particular sos.c
  - It is a basic SOS implementation that loads a.out into user memory and executes it
  - The a.out must be very simple since we haven't implemented anything yet.  It can only call _exit()
- Copy the files into your home area and compile them.  Test the *sos* with some user programs like *halt* and *cpu* (which must be named *a.out*)

# Assignment Details - 2

- Change sos.c to support restricted versions of the system calls read() and write()
  - read() should read from the console when the first argument is 0 (stdin)
  - write() should write to the console when the first argument is 1 (stdout) or 2 (stderr)
  - Any other first argument should return an error
- The system call driver works like this: if you return a negative value to a system call, it will return -1 to the user program and set the errno value to the return value times -1.
- This is how you can return different errors.
  - See /usr/include/sys/errno.h for standard error numbers.
- Remember that an address of X in the user program refers to that location in user memory, not the SOS binary

# Assignment Details - 3

- Your SOS should be ready for any arguments to read() and write() and should deal with them gracefully
  - Return with an error and don't leave the system in a bad state
- The next step is to change SOS so that it will call a given file with a given set of arguments.
- These may be compiled into the SOS, but in an easy to change location (globals in sos.c)
- Then SOS should load the file and execute it, but it should also set up memory so that the program sees the arguments as **argc** and **argv**

# Assignment Details - 4

- In summary, your SOS should run cross-compiled programs like cat, cat1, cat80, hw, argtest and allow the programs to read and write stdin, stdout and stderr.
- Also, it should allow a.out to take arguments that are hard-coded into sos.c

# Simulator notes

- If you do something incorrect, the simulator will exit
  - Not all the error messages are terribly helpful, though
- Remember: you can't just return from an interrupt or exception -- you must call run_user_code() or noop()
- Don't try to write a second character until the interrupt occurs indicating the first has been written
  - Trying to do so will give this error: "`Assertion failed: line 107, file "machine/console.c`"
- There are various flags that sos will accept.
  - *sos help* lists them

# Threads

- libmt is the user-level threads package mythreads
- SOS should make use of this and you are provided an implementation to use

# Assignment Tasks

- There are really four independent tasks for lab3 (=sos-lab1)
  - OS initialization
  - implement the **write()** system call which will only allow you to write the console
  - implement the **read()** system call which will only allow you to read the console
  - initialize the user code's argc and argv[] parameters
- They are separate tasks, but doing them in this order helps build the skills for the next ones

# OS Initialization

- The key thing here to realize is that the simulator is expecting your code (the OS kernel) to do its business, store off anything it will need to remember when the next exception or interrupt occurs, and then call **run_user_code()**. When **run_user_code()** executes, your code is done. Anything you store in a global variable (like the ready queue) will be preserved, any blocked threads will still be blocked, but the currently running thread dies
- Thus:
  - Exception called … -> mt_fork the task -> exception handler does a mt_joinall and eventually calls run_user_code()
- Thus when there is no more work for the kernel to do, your code sends the simulator back into user mode
  - Remember joinall just waits for *runnable* threads (or sleeping ones)
  - Thus it waits until threads have called exit or blocked on a semaphore

# Writing the console

- A good approach might be a semaphore for characters being written
- Also, you need a mutex for exclusive access to the console
- Why?

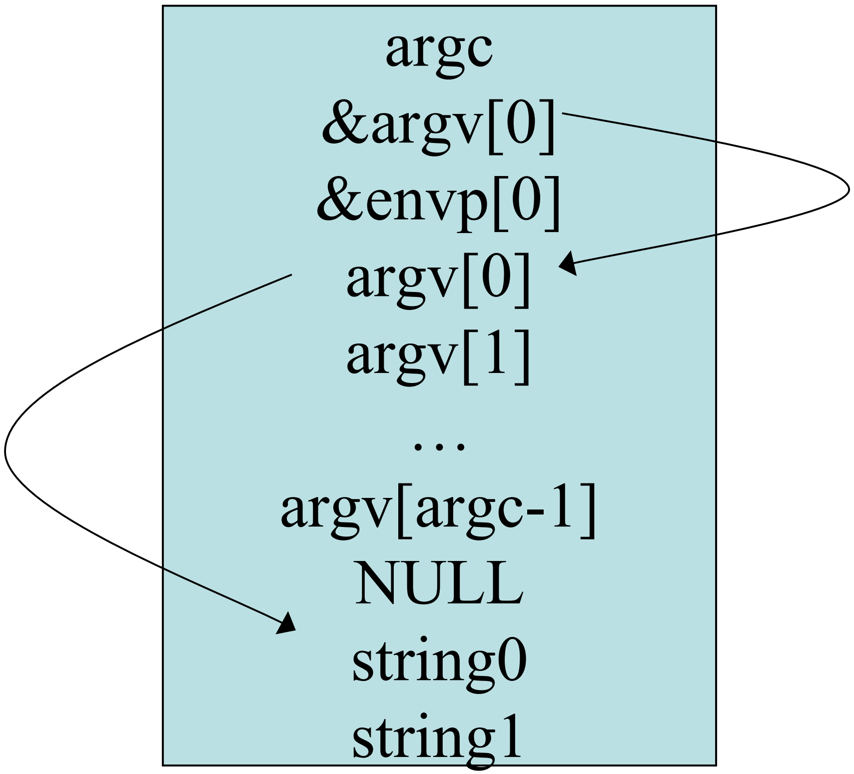
# Reading the console

- This is a little different than writing since the interrupts are *unsolicited*
  - The OS and user programs aren't necessarily expecting them
- Thus, a sensible approach is to consume characters from the console and buffer them in the kernel
  - Many devices work this way as hardware devices often have small buffers (one character in this case)
  - The kernel consumes them and stores them until the user process is ready to read them
- The most elegant solution takes advantage of the ability of the semaphore to count how many "wake ups" have occurred

# Initializing argc and argv

- This is by far the most fun part of the whole adventure
- You must setup the initial stack that contains the arguments to **main()**
  - i.e. argc and argv
- Think carefully about what we know about stacks

# Initializing argc and argv



# SOS

- Link in this order:
  - $(SOSDIR)/lib/$(GTYPE)/libsim.a\ $(SOSDIR)/lib/$(GTYPE)/libkt.a\ $(SOSDIR)/lib/$(GTYPE)/libfdr.a
- mt_joinall will exit when there are no *runnable* threads
- Initial PC is 0
  - Note NextPCReg
  - All instructions are word aligned (+4)

# Handy Tool: Dllist

- Read interface in dllist.h in /home/swany/sos
- Each node has a val, which is a Jval.  Each node also has a key
  - See jval.h for Jval details
- Interfaces:
  - dll_append(Dllist, Jval);
  - dll_prepend(Dllist, Jval);
  - dll_insert_b(Dllist, Jval);
  - dll_insert_a(Dllist, Jval);
  - dll_delete_node(Dllist);
  - int dll_empty(Dllist);

# Handy Tool: JRB

- JRB is an implementation of Red-Black trees, which are based on balanced binary trees
  - Operations take O(log(n)) time
- Create a tree with make_jrb() which returns a pointer to the head node in an empty tree
- Like the Dllist, each node has a val, which is a Jval. Each node also has a key
- For integers, you can insert using jrb_insert_int(JRB tree, int key, Jval val);
  - Returns a JRB (pointer to the new node)

# JRB

- The macros **jrb_first()**, **jrb_last()**, **jrb_prev()** and **jrb_next()**
  - work just like their counterparts in Dllist
- **jrb_find_int()** to find a key
  - Returns a JRB node or NULL
- **jrb_delete_node(JRB node);**