CIS520 – Operating Systems
Handout 7
OS Potpourri

- When does a process need to access OS functionality? Here are several examples
  - Reading a file. The OS must perform the file system operations required to read the data off of disk.
  - Creating a child process. The OS must set stuff up for the child process.
  - Sending a packet out onto the network. The OS typically handles the network interface.

  Why have the OS do these things? Why doesn't the process just do them directly?
  - Convenience. Implement the functionality once in the OS and encapsulate it behind an interface that everyone uses. So, processes just deal with the simple interface, and don't have to write complicated low-level code to deal with devices.
  - Portability. OS exports a common interface typically available on many hardware platforms. Applications do not contain hardware-specific code.
  - Protection. If give applications complete access to disk or network or whatever, they can corrupt data from other applications, either maliciously or because of bugs. Having the OS do it eliminates security problems between applications. Of course, applications still have to trust the OS.
- How do processes invoke OS functionality? By making a system call. Conceptually, processes call a subroutine that goes off and performs the required functionality. But OS must execute in a different protection domain than the application. Typically, OS executes in supervisor mode, which allows it to do things like manipulate the disk directly.
- To switch from normal user mode to supervisor mode, most machines provide a system call instruction. This instruction causes an exception to take place. The hardware switches from user mode to supervisor mode and invokes the exception handler inside the operating system. There is typically some kind of convention that the process uses to interact with the OS.
- Let's do an example - the `Open` system call. System calls typically start out with a normal subroutine call. In this case, when the process wants to open a file, it just calls the `Open` routine in a system library someplace.

```
/* Open the Nachos file "name", and return an "OpenFileId" that can
 * be used to read and write to the file.
 */
OpenFileId Open(char *name);
```

- Inside the library, the `Open` subroutine executes a `syscall` instruction, which generates a system call exception.

```
Open:
        addiu $2,$0,SC_Open
        syscall
        j       $31
        .end Open
```

By convention, the `Open` subroutine puts a number (in this case `SC_Open`) into register 2. Inside the exception handler the OS looks at register 2 to figure out what system call it should perform.

- The `Open` system call also takes a parameter - the address of the character string giving the name of the file to open. By convention, the compiler puts this parameter into register 4 when it generates the code that calls the `Open` routine in the library. So, the OS looks in that register to find the address of the name of the file to open.
- More conventions: succeeding parameters are put into register 5, register 6, etc. Any return values from the system call are put into register 2.
- Inside the exception handler, the OS figures out what action to take, performs the action, then returns back to the user program.
- There are other kinds of exceptions. For example, if the program attempts to deference a NULL pointer, the hardware will generate an exception. The OS will have to figure out what kind of exception took place and handle it accordingly. Another kind of exception is a divide by 0 fault.
- Similar things happen on a interrupt. When an interrupt occurs, the hardware puts the OS into supervisor mode and invokes an interrupt handler. The difference between interrupts and exceptions is that interrupts are generated by external events (the disk IO completes, a new character is typed at the console, etc.) while exceptions are generated by a running program.
- Object file formats. To run a process, the OS must load in an executable file from the disk into memory. What does this file contain? The code to run, any initialized data, and a specification for how much space the uninitialized data takes up. May also be other stuff to help debuggers run, etc.
- The compiler, linker and OS must agree on a format for the executable file. For example, Nachos uses the following format for executables:

```
#define NOFFMAGIC       0xbadfad        /* magic number denoting Nachos
                                         * object code file
                                         */
typedef struct segment {
  int virtualAddr;              /* location of segment in virt addr space */
  int inFileAddr;               /* location of segment in this file */
  int size;                     /* size of segment */
} Segment;
typedef struct noffHeader {
   int noffMagic;               /* should be NOFFMAGIC */
   Segment code;                /* executable code segment */
   Segment initData;            /* initialized data segment */
   Segment uninitData;          /* uninitialized data segment --
                                 * should be zero'ed before use
                                 */
} NoffHeader;
```

- What does the OS do when it loads an executable in?
  - Reads in the header part of the executable.
  - Checks to see if the magic number matches.
  - Figures out how much space it needs to hold the process. This includes space for the stack, the code, the initialized data and the uninitialized data.
  - If it needs to hold the entire process in physical memory, it goes off and finds the physical memory it needs to hold the process.
  - It then reads the code segment in from the file to physical memory.
  - It then reads the initialized data segment in from the file to physical memory.

  - It zeros the stack and unintialized memory.

- How does the operating system do IO? First, we give an overview of how the hardware does IO.
- There are two basic ways to do IO - memory mapped IO and programmed IO.
  - Memory mapped IO - the control registers on the IO device are mapped into the memory space of the processor. The processor controls the device by performing reads and writes to the addresses that the IO device is mapped into.
  - Programmed IO - the processor has special IO instructions like IN and OUT. These control the IO device directly.
- Writing the low level, complex code to control devices can be a very tricky business. So, the OS encapsulates this code inside things called device drivers. There are several standard interfaces that device drivers present to the kernel. It is the job of the device driver to implement its standard interface for its device. The rest of the OS can then use this interface and doesn't have to deal with complex IO code.
- For example, Unix has a block device driver interface. All block device drivers support a standard set of calls like open, close, read and write. The disk device driver, for example, translates these calls into operations that read and write sectors on the disk.
- Typically, IO takes place asynchronously with respect to the processor. So, the processor will start an IO operation (like writing a disk sector), then go off and do some other processing. When the IO operation completes, it interrupts the processor. The processor is typically vectored off to an interrupt handler, which takes whatever action needs to take place.
- Here is how Nachos does IO. Each device presents an interface. For example, the disk interface is in disk.h, and has operations to start a read and write request. When the request completes, the "hardware" invokes the HandleInterrupt method.
- Only one thread can use each device at a time. Also, threads typically want to use devices synchronously. So, for example, a thread will perform a disk operation then wait until the disk operation completes. Nachos therefore encapsulates the device interface inside a higher level interface that provides synchronous, synchronized access to the device. For the disk device, this interface is in synchdisk.h. This provides operations to read and write sectors, for example.
- Each method in the synchronous interface ensures exclusive access to the IO device by acquiring a lock before it performs any operation on the device.
- When the synchronous method gets exclusive access to the device, it performs the operation to start the IO. It then uses a semaphore (P operation) to block until the IO operation completes. When the IO operation completes, it invokes an interrupt handler. This handler performs a V operation on the semaphore to unblock the synchronous method. The synchronous method then releases the lock and returns back to the calling thread.