System calls:
A Nachos approach

Dr. Daniel Andresen

CIS520 Operating Systems

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 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.

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

OpenFileId Open(char *name);
```

- Open() 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. By convention, the compiler puts this into register 4. So, the OS looks in R4 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.

Interrupts vs. Exceptions

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.