CS 3305A
System Calls
Lecture 7
Interface to the OS

- All operating systems have an interface to OS that is accessible by users/user programs
- We had a discussion of shells which allows a user to interface with the operating system through the command line
  - A second strategy is through a graphical user interface (GUI)
- We had seen how system functions (such as fork()) can communicate with OS
A View of Operating System Services

user and other system programs

<table>
<thead>
<tr>
<th>GUI</th>
<th>batch</th>
<th>command line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>user interfaces</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

system calls

- program execution
- I/O operations
- file systems
- communication
- resource allocation
- accounting
- error detection
- protection and security

operating system

hardware
Interface to the OS

- **System calls** provide an interface to OS services
  - Program passes relevant information to OS
  - OS performs the service if
    - The OS is able to do so
    - The service is permitted for this program at this time
- **System calls** are typically written in C and C++
  - Tasks that require hardware to be accessed directly may be written using assembly language
Application Programmer Interface (API)

- Programmers call a function (**system function**) in a library which invokes system calls
- The programmer only needs to understand the system function by understanding its parameters and results
- Example
  - Programmer API: `count = read(fd, buf, nbytes)`
  - System calls Used: `sys_read()`
  - System call code is part of the kernel (**Core OS**)
Examples: Other System Calls

- Linux Examples:
  - `sys_fork`, `sys_pipe()`

- Note: We have been using system call loosely
  - Could be referring to the system function
  - System function and System call are two different entities
    - System function: used by user / programmer (API)
    - System call: Part of OS Kernel
## Some System Functions For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pid = fork()</code></td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td><code>pid = waitpid(pid, &amp;statloc, options)</code></td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td><code>s = execve(name, argv, environp)</code></td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td><code>exit(status)</code></td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>
## Some System Functions For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = open(file, how, ...)</code></td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td><code>s = close(fd)</code></td>
<td>Close an open file</td>
</tr>
<tr>
<td><code>n = read(fd, buffer, nbytes)</code></td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td><code>n = write(fd, buffer, nbytes)</code></td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td><code>position = lseek(fd, offset, whence)</code></td>
<td>Move the file pointer</td>
</tr>
<tr>
<td><code>s = stat(name, &amp;buf)</code></td>
<td>Get a file’s status information</td>
</tr>
</tbody>
</table>
Some System Functions For Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s = mkdir(name, mode)</code></td>
<td>Create a new directory</td>
</tr>
<tr>
<td><code>s = rmdir(name)</code></td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td><code>s = link(name1, name2)</code></td>
<td>Create a new entry, name2, pointing to name1</td>
</tr>
<tr>
<td><code>s = unlink(name)</code></td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td><code>s = mount(special, name, flag)</code></td>
<td>Mount a file system</td>
</tr>
<tr>
<td><code>s = umount(special)</code></td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>
APIs

- Let’s say that a user program has the following line of code: `read(fd, buf, nbytes)`
- This program needs the operating system to access the file and read from it.
- Some issues to be addressed:
  - How are parameters passed?
  - How are results provided to the user program?
  - How is control given to the system call and the operating system?
System Call Parameter Passing

- Three general methods used to pass parameters to the OS
  - Registers: Pass the parameters in registers
    - In some cases, there may be more parameters than registers
  - Block: Parameters stored in a block, or table, in memory, and address of block passed as a parameter in a register
    - This approach taken by Linux and Solaris
  - Stack: Parameters placed, or pushed, onto the stack by the program and popped off the stack by the operating system

- Block and stack methods do not limit the number or length of parameters being passed
Linux: Parameter passing

- System calls with fewer than 6 parameters passed in registers
  - If 6 or more arguments
    - Pass pointer to block structure
A system call number is associated with each system call.

The OS maintains a system call handler table which is indexed according to the system call numbers.

Entry in table points to code.

- sys_read code
- sys_write code
System Calls and Traps

- TRAP switches CPU to supervisor (kernel) mode
  - The state of the user process is saved so that the OS instructions needed can be executed (system call)
  - When the system handler finishes execution then the user process can execute
Making a System Call

- System function call:
  
  \[
  \text{count} = \text{read (fd, buffer, length)}
  \]

- Step 1: The input parameters are passed into registers or to a block

- Step 2: TRAP (execution of system call) is executed
  
  - The state of the user process is saved T
  
  - System call number for read() is sent to system call handler
  
  - This code/number tells the OS what system call handler (kernel code) to execute

  - This causes a switch from the user mode to the kernel mode
Making a System Call

- Step 3: System call handler code is executed
- Step 4: After execution control returns to the library procedure (system function)
System Call

- The system call handler will have to actually wait for data from the disk.
- Reading data from disk is much slower than memory.
- We do not want the CPU to be idle while waiting for data from the disk.
  - Most operating systems allow for another executing program to use the CPU.
  - This is called **multiprogramming** - more later.
- How does a process find out about reading being completed?
  - Interrupt.
Interrupt Handlers

- **Interrupt handler**
  - Saves processor state: CPU registers of interrupted process are saved into a data structure in memory
  - Runs code to handle the I/O
  - Restores processor state: CPU registers values stored in memory are reloaded
  - PC is set back to PC of interrupted process