Interface to the OS

- 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).
- All operating systems have an interface to it that is accessible by users/user programs.
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>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

services

operating system

<table>
<thead>
<tr>
<th>protection and security</th>
</tr>
</thead>
</table>

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.
- System call code is part of the kernel.
- 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()`, `sys_open()`,
  `sys_close()`
Examples: Other System Calls

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

- **Windows Examples:**
  - `Createprocess()`, `Create_Pipe()`

- **Note:** We will often use system call loosely
  - Could be referring to the system function
Some System Functions For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(pid, &amp;statloc, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>s = execve(name, argv, environp)</td>
<td>Replace a process’ core image</td>
</tr>
<tr>
<td>exit(status)</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>
Let's say that a user program has the following line of code:

```c
    count = 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 containing argument list
System Call Table

- A system call number (Read_Code) 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

```
<table>
<thead>
<tr>
<th>sys_read</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_write</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>sys_read code</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_write code</td>
</tr>
</tbody>
</table>
```
System Calls and Traps

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

User Mode

read (file, buffer, n);

_\_read(file, buff, n)

User program

call

read (....)

User Mode

Kernel Mode

sys_read()

call

System Call Handler

System call handler takes the Read_Code to determine where the actual code is located
Making a System Call

- **System function call:**
  
  `read (fd, buffer, length)`

- **Step 1:**
  - The calling program pushes the parameters onto the stack
    - A stack entry for each function call
  - This is done for preparation for calling the `read` library procedure which actually makes the system call

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="#" alt="Diagram" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>read():</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fd = fd'</code></td>
</tr>
<tr>
<td><code>buffer = buffer'</code></td>
</tr>
<tr>
<td><code>length = length</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main function</th>
</tr>
</thead>
<tbody>
<tr>
<td>main() function</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine instructions for main(), read()</th>
</tr>
</thead>
</table>
Making a System Call

- **Step 2:** The input parameters are passed into registers or to a block.

- **Step 3:** TARP is executed
  - The state of the user process is saved.
  - The system call number / read_code 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 4: System call handler code is executed
- Step 5: 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?
  - Polling
  - Interrupt
Polling

- What if we have the CPU periodically check the disk controller to see if it is ready to put read data into memory?
- What if the device is not in use or used infrequently?
- Polling is like picking up your phone every few minutes to see if you have a call

- What if instead you wait to be signaled i.e., wait for an interrupt?
Interrupts

- Give each device a wire (interrupt line) that is used to signal the processor when data transfer is complete
  - When interrupt signaled, processor executes a routine called the interrupt handler to deal with the interrupt
  - No overhead when there no device ready to be serviced
Interrupts and Interrupt Handlers

- Interrupt signal is sent over the bus

- CPU has a table with the entries for each device type which is the address of the first instruction of the code to be executed to handle the interrupt
Interrupts and 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
Summary

- We discussed the implementation of system calls