Lecture Topics

• Today: Operating System Overview
  (Stallings, chapter 2.1-2.4, 2.8-2.10)

• Next: continued

Announcements

• Consulting hours posted
• Self-Study Exercise #1 posted
• Self-Study Exercise #2 posted
• Project #1 posted
What is an operating system?

- **Software which:**
  - Controls the resources of a computer
  - Provides an interface between processes and hardware

- **Advantages over working with the bare machine:**
  - Programmer productivity
  - Better utilization of resources
  - Fair access to resources
  - Correctness and security

- **Disadvantage:** OS consumes resources

Limited Direct Execution

- A modern OS does not monitor or interpret the execution of individual instructions in a user process

- Instead, the OS relinquished control of the CPU to user processes (user processes run directly on the hardware)

- Potential problem: OS must have ability to regain control of the CPU (via interrupts)

- Called *limited direct execution*
Virtual Memory

- When a process reads and writes to computer memory, the addresses it uses are not physical memory addresses.
- Rather, the OS and architecture collaborate to map user's logical addresses to physical addresses.
- Called virtual memory.

Major OS Functions

- Process and thread management
  - process control: creation, termination, suspension, resumption, scheduling, resource allocation, etc.
- Memory management
  - allocation/deallocation of computer memory, implementation of virtual memory, demand paging
- Interprocess (or interthread) communication
  - signals, pipes, messages, events, semaphores, shared memory, etc.
Major OS Functions (cont.)

• I/O and device drivers
  – character and block device management, handling
    interrupts, direct memory access (DMA), etc.

• File management
  – file system implementation, memory-resident data
    structures, buffer cache, device files, etc.

• Network management
  – sockets or equivalent, TCP/IP protocols, etc.

Major Advances (Denning)

• Processes
• Memory management
• Information protection and security
• Scheduling and resource management
Processes

Memory Management
Scheduling and Resource Management

Operating System

- Service Call from Process
- Interrupt from Process
- Interrupt from I/O
- Long-Term Queue
- Short-Term Queue
- I/O Queues
- Short-Term Scheduler (code)
- Pass Control to Process

Linux Overview

- Applications
- DBs, all server types, ...
- System Libraries
- System Call Interface
- VFS
- ext3/…
- ZFS
- LVM
- Block Device Interfaces
- Sockets
- TCP/UDP
- Scheduler
- IP
- Ethernet
- Virtual Memory
- Device Drivers
- I/O Bus
- I/O Bridge
- Expander Interconnect
- I/O Controller
- Disk
- Disk Transports
- Network Controller
- Port
- Port
- CPU 1
- Memory Bus
- DRAM
System Call Interface

- The system call interface is the programming mechanism that a running process uses to request service from the Linux kernel.
- In Linux 3, there are 300+ system calls.
- The set of system calls has evolved over time (new ones added to every new version of Linux).
System Calls (partial list)

- Each system call is identified by a number
- Each system call number mapped to specific function
- Each function expects a specific set of arguments

<table>
<thead>
<tr>
<th>Number</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>sys_read</td>
</tr>
<tr>
<td>1</td>
<td>sys_write</td>
</tr>
<tr>
<td>2</td>
<td>sys_open</td>
</tr>
<tr>
<td>3</td>
<td>sys_close</td>
</tr>
<tr>
<td>4</td>
<td>sys_stat</td>
</tr>
<tr>
<td>5</td>
<td>sys_fstat</td>
</tr>
<tr>
<td>6</td>
<td>sys_lstat</td>
</tr>
<tr>
<td>7</td>
<td>sys_poll</td>
</tr>
<tr>
<td>8</td>
<td>sys_lseek</td>
</tr>
<tr>
<td>9</td>
<td>sys_mmap</td>
</tr>
</tbody>
</table>

System Call Implementation

- To invoke one of the system calls, a process places the arguments (including the number of the system call) in registers, then issues a specific machine language instruction:
  - Intel: syscall
  - ARM: svc
  - SPARC: ta

- The machine instruction turns on kernel mode and calls the corresponding function
System Call Implementation

System call:
- turns on kernel mode
- uses call number as index into table of function pointers
- jumps to actual function

Return:
- turns off kernel mode
- jumps back to process with integer return value (status code)

Cumbersome to invoke system calls directly, so "wrapper" functions developed:

```c
int read( int fd, void *buf, size_t count );
int write( int fd, void *buf, size_t count );
int open( const char *pathname, int flags );
int close( int fd );
```

Those functions preprocess the arguments, invoke the appropriate system call, and postprocess the return value.
System Call Example

Function "write" is declared as:

```
int write( int fd, void *buf, size_t count );
```

It copies up to "count" bytes from the buffer "buf" to a file; it returns the actual number of bytes which were copied.

Files are represented as "file descriptors" (integer numbers); file descriptor 1 is the standard output stream.

System Call Example

The following program sends "Hello, World!" to the standard output stream:

```
#include <unistd.h>

int main()
{
    write( 1, "Hello, world!\n", 14 );

    return 0;
}
```
System Call Example

Trace of system calls for "example02.c":

execve("./a.out", ["a.out"], [/* 28 vars */]) = 0
uname({sys="Linux", node="cse410", ...}) = 0
brk(0) = 0x16b6000
brk(0x16b6d00) = 0x16b6d00
readlink("/proc/self/exe", "/user/cse410/Examples/a.out", 4096) = 27
brk(0x16d7d00) = 0x16d7d00
brk(0x16d8000) = 0x16d8000
write(1, "Hello, world!\n", 14) = 14
exit_group(0) = ?
+++ exited with 0 +++

Library Functions

The system call "wrapper" functions are very low level, so library functions were developed. The various standard library functions do some processing, then invoke the appropriate system calls.

For example, "printf" handles formatted output and places strings into a buffer. Ultimately, it invokes "write" to copy the contents of the buffer to the standard output stream.
Library Functions

Function "printf" implicitly uses the standard output stream, calls "write":

```c
#include <stdio.h>

int main()
{
    printf( "Hello, world!\n" );

    return 0;
}
```

Library Functions

Trace of system calls for "example03.c":

```c
execve("./a.out", ["a.out"], [/* 28 vars */]) = 0
uname({sys="Linux", node="cse410", ...}) = 0
brk(0) = 0x1a39000
brk(0x1a39d00) = 0x1a39d00
readlink("/proc/self/exe", "/user/cse410/Examples/a.out", brk(0x1a5ad00) = 0x1a5ad00
brk(0x1a5b000) = 0x1a5b000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE
write(1, "Hello, world!\n", 14) = 14
exit_group(0) = ?
+++ exited with 0 +++
```
Linux Utility Programs

The Linux environment includes a rich collection of utility programs which handle common tasks:

- `cp` – copy a file
- `ls` – list the contents of a directory
- `mv` – rename a file
- `mkdir` – create a new directory

Those utility programs use system calls (and library functions) to do the work.
Example: cp

Assume the user enters: cp fileA fileB

- connect to fileA (using "open")
- create fileB (using "open")
- loop to copy contents of fileA to fileB (using "read" and "write")
- disconnect from fileA and fileB (using "close")

The "cp" utility program should also do error checking and handle special cases.

Linux Utility Programs

The source code for the GNU implementation of common utility programs is available ("coreutils"):

- cp – 1206 lines
- ls – 5176 lines
- mv – 493 lines
- mkdir – 296 lines
Linux Shells

One group of utility programs are the shells.

A shell is a utility program which allows the user to enter commands interactively and displays the results on the screen.

```
<34 arctic:~/Examples > pwd
/user/cse410/Examples
<35 arctic:~/Examples > ls
Admin example01 example01.c example02 example02.c example03 example03.c
```

Linux Shells

Two main "families" of shells:

- Bourne family
  - sh (Bourne shell)
  - bash (Bourne again shell)
  - zsh (Z shell)

- C-shell family
  - csh (C shell)
  - tcsh (T shell)
Linux Shells

- A default shell is assigned when each user's account is created
- The default on CSE systems is "tcsh"
- The "chsh" command allows user to select a different shell
- The list of shells is in "/etc/shells"

Linux Shells

- When executed, a shell reads a set of configuration files (usually "dot" files)
- The shell repeatedly:
  - displays a prompt
  - accepts input from the user
  - interprets the command entered by the user
  - displays the results
Command Line Processing

• The shell expects the first "token" on a line to be the name of a command
  – built-in command
  – executable program

• A built-in command is interpreted within the shell itself

• An executable programs is executed as a separate process

Command Line Processing

• If a command is not a built-in command, the shell assumes it is an executable program somewhere in the file system

  /bin/cp
  /user/cse410/bin/mailscore

• It uses the user's $PATH environment variable to locate the executable program
Command Line Processing

• Typical search path

/soft/linux/bin
/usr/local/bin
/usr/bin
/bin
/usr/games
/user/cse410/bin
.

Command Line Processing

• Executable programs can be:
  – machine language programs
  – shell scripts

• File permissions must be set correctly:
  -rwxr-xr-x /bin/cp
  -rwx------ /user/cse410/bin/mailscore
• C/C++ provides access to:
  – command-line arguments
  – environment variables

• Sample programs:
  
  /user/cse410/Examples/example04
  /user/cse410/Examples/example05

Arguments to function "main":

```c
int main( int argc, char* argv[] )
```

  argc – number of command-line arguments
  argv – array of pointers to strings

Array terminated with a null pointer; each string is
terminated with a null byte.