Introduction to Operating Systems

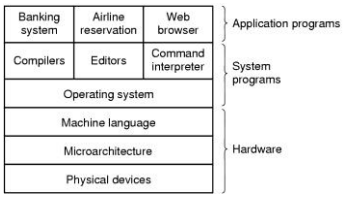
Introduction

- What is the aim of the subject?
- Why are OSs important?
- What is an OS?
- History
- Basic concepts

The Aim of the course

- WILL NOT TEACH YOU HOW TO **USE** AN OPERATING SYSTEM.
- WILL HELP YOU TO ACHIEVE AN UNDERSTANDING OF **HOW AN OPERATING SYSTEM WORKS.**

Computer System Components



- Application Software : Bank automation system, airline reservations, payroll etc.
- **System Software : OS, data base, compilers, editors etc.**

What is SYSTEM SOFTWARE?

- System software provides the environment and the tools to develop or create the application software. So, system software is sort of virtual machine.
- System software is the interface between the hardware and the applications

Why is the OS Important?

- The operating system is the foundation upon which all computing work is performed.
- Knowledge of the internals of an OS is essential to achieve efficiency in
  - building software applications
  - choosing a computing platform

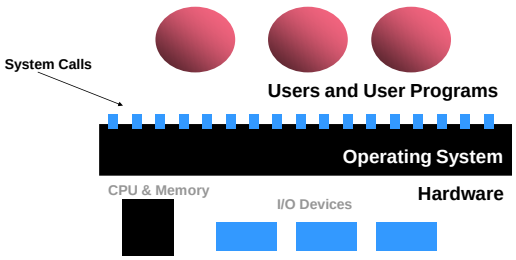
What is an Operating System?

- A big and complex program
- Converts hardware into a useable resource
- Provides efficient use of hardware resources

Life without an OS

- Every programmer would
  - have to know any hardware (!)
  - be able to access the hardware
- Every program would
  - contains code to do the same thing (makes use of hardware)
  - Probably every one would reinvenr America!

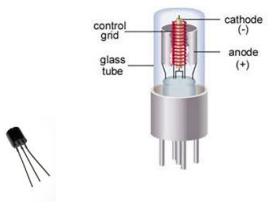
Where does the OS Fit in a system?



History :HW + software
First Generation (1945-1955)

- Vacuum tubes
- No operating system
- Programming is done by wiring a plug board
- Applications are mostly numerical calculations (trajectory computations, computation of tables such as sine, cosine etc.)

Transistor and vacuum tube



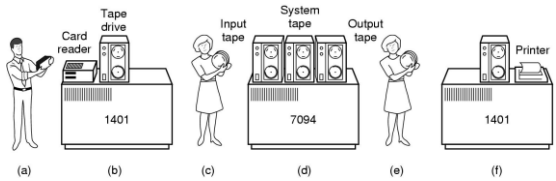
History:
Second Generation (1955-1965)

- Transistors
- Commercially produced computers
- Very expensive and very slow computers compared with your old PC at home
- Batch operation (collect jobs, run in one go, print all outputs)

Second Generation: Efficient I/O

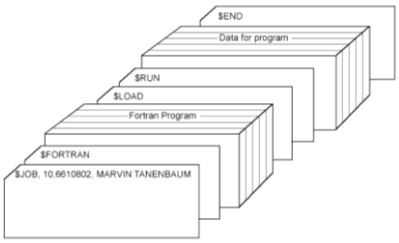
- Spooling (Simultaneous Peripheral Operation On-line)
  - off-line spooling
  - on-line spooling
- Off-line spooling : replace slow I/O devices with I/O dedicated computers so that the main system sees these machines as its I/O devices

Second Generation:Early Batch Systems



- bring cards to 1401
- read cards to tape
- put tape on 7094 which does computing
- put tape on 1401 which prints output

A Deck of Cards (Program)



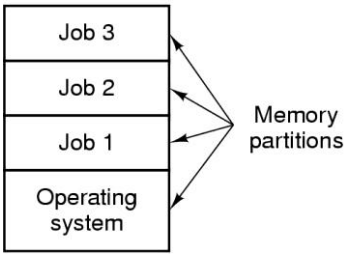
Second Generation: Applications and languages

- Applications were mostly scientific and engineering calculations (eg., solution of partial differential equations)
- High level languages such as FORTRAN COBOL, ALGOL, etc..

History: Third Generation (1965-1980)

- Integrated circuits (small scale) packed as chips
- I/O processors (channels) which can work in parallel with CPU - Multiprogramming

Multiprogramming system - three jobs in memory



Multiprogramming system – more than one jobs in memory

- On-line spooling (using channels)
- Time-sharing (TTY terminals and VDU's)
- Multics OS - original UNIX on Minicomputers
- Minicomputers were cheaper than mainframes but with limited memory anth other hw units (eg. DEC PDPx)

History: Fourth Generation (1980-1990)

- Large scale integration
- Personal computers
- CP/M, MS DOS, Unix operating systems
- Networks

Now!

- Client/Server computation
- Clients : PCs, workstations running under Windows NT and UNIX like operating systems
- Servers : workstations systems runinig under UNIX and Windows NT
- Internet and intranet networking (WWW)
- Cloud computing

Important Points

- OS should provide
  - a simpler, more powerful interface
  - higher level services
- OS services only accessed via system calls
- Users and programs can't directly access the hardware

Set of System Calls (APIs) is what programs think the operating system is.

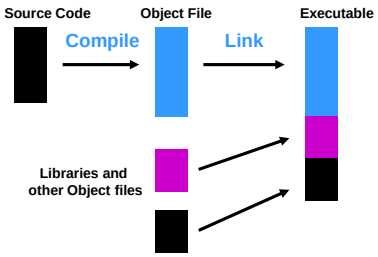
UNIX like OS Concepts

- Kernel
  - The main OS program. Contains code for most services. Always in primary memory
- Device Drivers
  - Programs that provide a simple, consistent interface to I/O devices
  - Typically part of the kernel

Some OS Concepts

- Program
  - A static machine code program on a disk
- Process
  - A program in execution.
  - The collection of OS data structures and resources owned by a program while it is running.

Producing an Executable under an OS

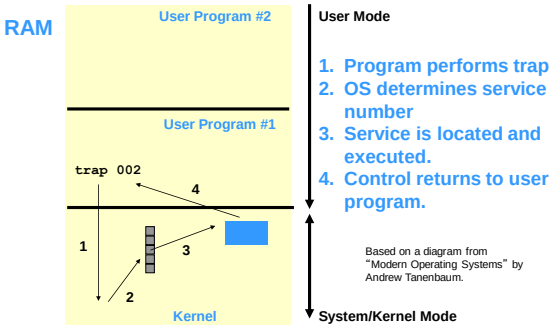


Use of system calls:
A Simple Program to print a directory

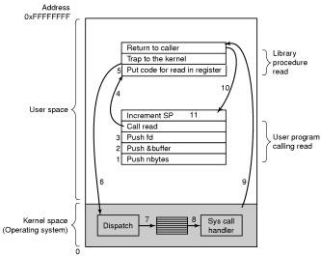
```
#include    <sys/types.h>
#include    <dirent.h>
#include    "ourhdr.h"

int main(int argc, char *argv[])
{
    DIR                 *dp;
    struct dirent   *dirp;

    if (argc != 2)
      err_quit("a single argument (the directory name) is required");
    if ( (dp = opendir(argv[1])) == NULL)
      err_sys("can't open %s", argv[1]);

    while ( (dirp = readdir(dp)) != NULL)
      printf("%s\n", dirp->d_name);

    closedir(dp);
    exit(0);
}
```

Functions supplied by system libraries.
These functions will contain a trap instruction.

What is a Trap? Passing control to OS



Steps in Making a System Call



There are 11 steps in making the system call :
Count=read (fd, buffer, nbytes)
Unsuccessful call will set count to -1.

System call execution

- If the system call cannot execute the returned value is set to -1 and a errno is put in a global variable, so always check the the returned value…
- Calling program pushes the parameters onto a stack
- Then the lib procedure is called, which puts the corresponding system call number in a table the OS expects and then call a TRAP instruction to switch from user mode to kernel mode
- Then the kernel executes the system call handler, after which the control may be passed back to the user program…
- The kernel then clears the stack from the remains of the system call which has just been completed

System call execution

- **POSIX-Portable Operating System Interface** for **Unix** has about 100 system calls
- MSFT decouples the system calls and application interface, known as Win32 API (Application Program Interface). Win32 API has thousands of calls, some of which may call system calls, most of which execute in user mode…

Some System Calls For Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some System Calls For File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

Some System Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some System Calls For Miscellaneous Tasks

Miscellaneous

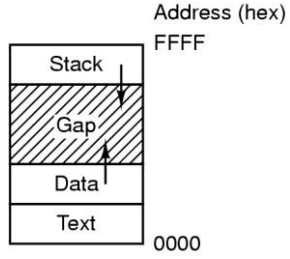
Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

A System Call Example

- A stripped down shell:

```
• /* shell  is the command interpreter of OS */
#define TRUE 1
while (TRUE) {                    /* repeat forever, TRUE is set to 1 */
   type_prompt( );                /* display prompt */
   read_command (command, parameters)   /* input from terminal */

if (fork() != 0) {                /* fork off child process */
   /* Parent code */
   waitpid( -1, &status, 0);      /* wait for child to exit */
} else {
   /* Child code */
   execve (command, parameters, 0);     /* execute command */
 }
}
```


System Calls: Process execution structure



- Processes have three segments: text, data, stack

System Calls: a short list UNIX system calls

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

Some Win32 API calls with corresponding UNIX system calls