Operating Systems Design (CS 423)

Elsa L Gunter
2112 SC, UIUC
http://www.cs.illinois.edu/class/cs423/

Based on slides by Roy Campbell, Sam King, and Andrew S Tanenbaum

History of Operating Systems

- Two major forces in OS history
  - Equipment was very expensive, steadily becoming cheaper
    - People viewed as increasingly expensive
  - Systems, including OS growing steadily in complexity
- Original OS Structure
  - Simple, library of services, single "thread" operation
  - Gave poor utilization of system

Mainframes and Single Operator Computers

- Three groups interacted with computers
  - Computer designers / developers configure computers for specific application
  - Operators collects program, runs it on the computer, collects output and returns it to the user
  - Programmers write programs, submits it to the operator, waits for output, looks for errors, repeat
- Wastes both computer and human time

Batch Computers

- Decouple Input, Execution, Output
  - One device reads in a collection of programs
  - Second device (main computer) executes program batch and outputs results to a "relatively fast" storage (magnetic tape)
  - Third device outputs results from tape to line feeder (basically a dot matrix printer)
- Allows for parallel processing of input, execution, output
- Executing multiple programs; protection starts to become an issue

Main Time Sharing

- Allow multiple humans to interact at the same time
- Main Idea: Human are very slow I/O
- Switch / interleave other activities while waiting for human
- OS much more complicated
  - Multiple sources of input
  - Multiple jobs running in the same time interval
  - Protection needs much more serious

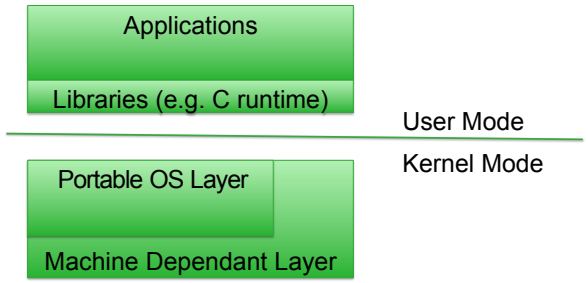
Personal Computers

- Computer hardware much much cheaper
- Single user
- Simple single threaded OS, basically library routines
- No time sharing between jobs
  - Wait for printer
- No protection
- Gradually migrated to full multi-user OS

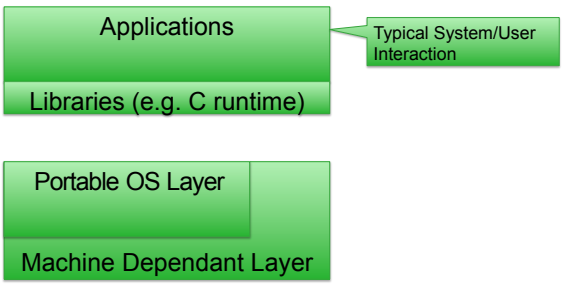
Today's Computer

- Everywhere – microwaves, sewing machines, washing machines, cars …
- Single User computers: Cell phones, gaming devices, TiVo. Etc
- Many start with modern complex OS (eg Linux)

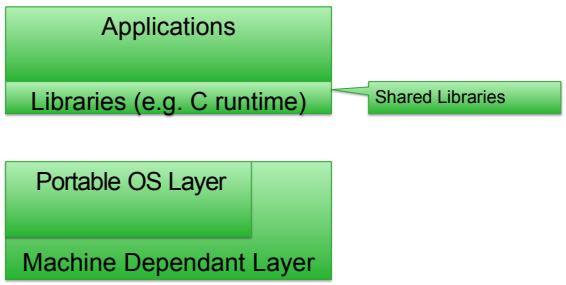
Common Modern OS Architecture



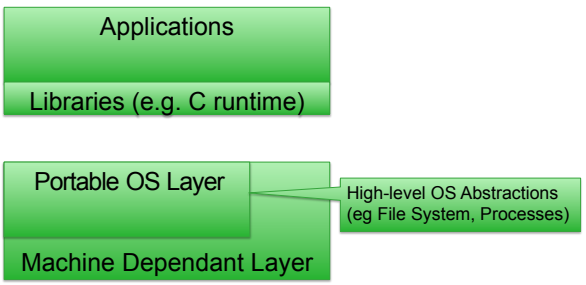
Common Modern OS Architecture



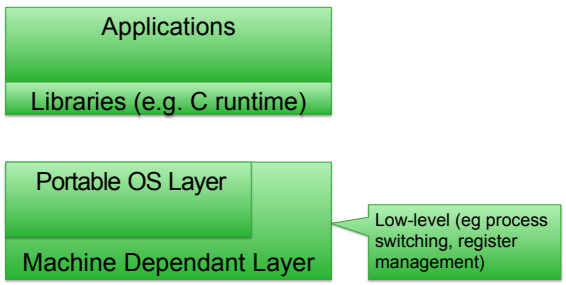
Common Modern OS Architecture



Common Modern OS Architecture



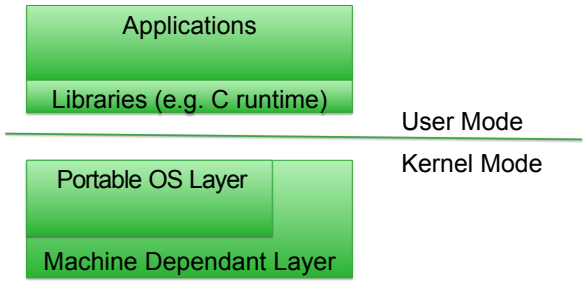
Common Modern OS Architecture



Common Modern OS Architecture

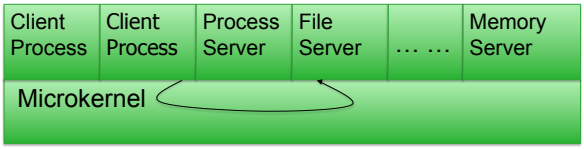
- OS developer need to be paranoid
  - Application software buggy
  - Hardware unreliable
  - Users untrustworthy (naïve / malicious)
- OS developer need to be engineers
  - More is better
    - Use as much memory as possible
  - Faster is better
    - Indirections take time, users don't like to wait

Monolithic OS Architecture



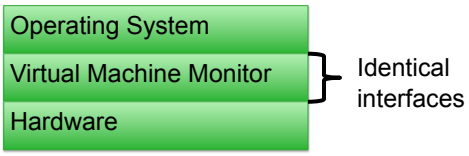
Alternative: Microkernel Architecture

- Protection: How much, what cost?
- Windows NT, Mac OS X microkernel origins
- More common on embedded systems



Alternative: Virtual Machine Monitors

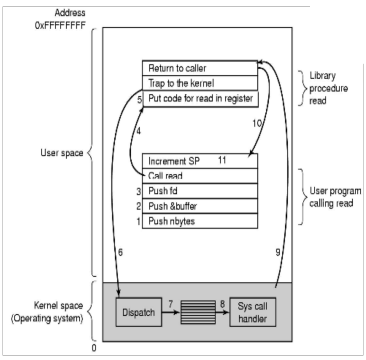
- VMM like microkernel with simple interface
- Performance?
- VMM – Microkernel distinction increasing blurred



Abstractions and Interfaces

- Key Abstractions: file system and process
- Key Interface: system call
- Example:

read(fd, buffer, nbytes)



Unix System Calls

- Process management:
  - fork, exec, wait
- File handling
  - open, read, write, close
- File namespace
  - readdir, link, unlink, rename

Implement a Shell

- Basic behavior
  - Wait for input; parse it to get command
  - Create child process; wait for it to return
  - Repeat
- Tools to use:
  - fork(): makes two processes with same code; returns new pid (of child) to parent, 0 to child
  - exec(argv): replace current process with argv
  - wait(pid): wait for process with pid to return

Implement a Shell

```
while(1) {
  user_input = display_prompt();
  parse_input(user_input,&argv);




}
```


Implement a Shell (approximation)

```
while(1) {
  user_input = display_prompt();
  parse_input(user_input,&argv);
  pid = fork();
  if (pid == 0) //child process
  {exec(argv)}
  else {wait(pid)}  //parent
}
```


Process Illusion

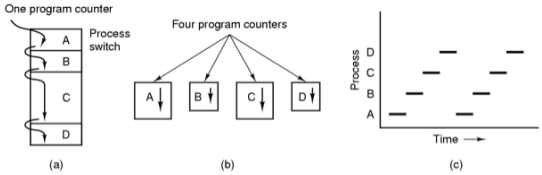
- Illusion: infinite number of processes, exclusive access to CPU, exclusive access to infinite memory
- Reality: fixed process table sizes, shared CPUs, finite shared memory
- Can break illusion: create a process that creates a process that creates a process …
- Will lock up your system, may need to reboot

Fork bomb

- How can we fix this?
- What price do we pay for the fix?
- Is it worth it to fix it?
- Two perspectives: Math and engineering

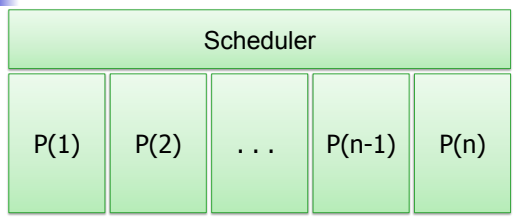
Why Processes?

- Heart of modern (multiprocess) OS



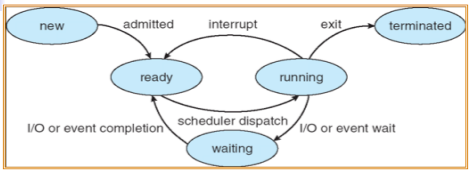
a) Multiprogramming of four programs.
b) Conceptual model of four independent, sequential processes.
c) Only one program is active at once.

Approx Implementation of Processes



- Scheduler interleaves processes, preserving their state
- Abstraction: sequential processes on dedicated CPU

Process States



- As a process executes, it changes *state*
  - new: Process being created
  - ready: Process waiting to run
  - running: Instructions are executed
  - waiting: Process waiting for some event to occur
  - terminated: Process finished execution