CSC 2/456: Operating Systems

Instructor: Sandhya Dwarkadas

TA: Brandon Shroyer

General Course Information

- Course Web page:
  - www.cs.rochester.edu/~sandhya/csc256
- Course-related announcement/correspondence:
  - Blackboard Discussion Board
- Texts
  - Tanenbaum, "Modern Operating Systems"
  - Silberschatz et al, "Operating System Concepts"

General Course Information (cont.)

- Assignments and grading
  - six programming assignments (total 50%)
  - midterm and final (40%)
  - homework/other (10%)
  - Other: participation in class discussions, presentations
- "CSC456 Part" in assignments
- C programming
- Class presentation and end-of-term survey paper for CSC456 students

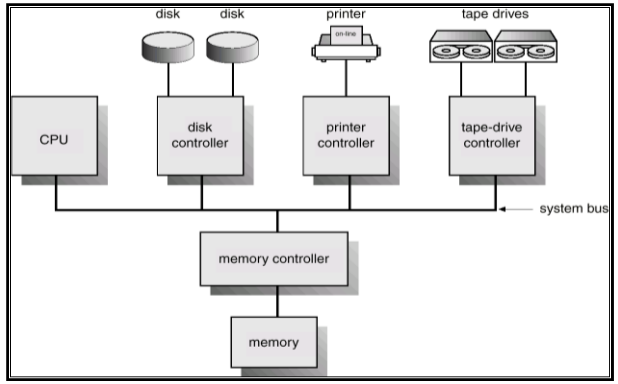
What is an Operating System?

- Software that abstracts the computer hardware
  - Hides the messy details of the underlying hardware
  - Presents users with a resource abstraction that is easy to use
  - Extends or virtualizes the underlying machine
- Manages the resources
  - Processors, memory, timers, disks, mice, network interfaces, printers, displays, …
  - Allows multiple users and programs to share the resources and coordinates the sharing

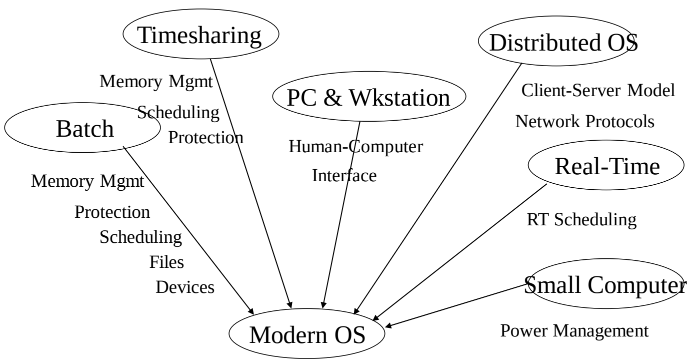
Why Study Operating Systems?

- Learn to design an OS or other computer systems
- Understand an OS
  - Understand the inner workings of an OS
  - Enable you to write <u>efficient/correct</u> application code

Computer-System Architecture



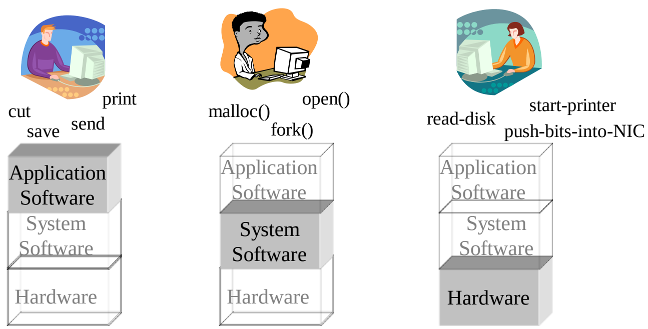
Evolution of Modern OS



Examples of Modern OSes

- UNIX variants (e.g., Solaris, Linux) -- have evolved since 1970
- Windows 7/NT/2K -- has evolved since 1989
- Smartphone Oses: Android, iOS, …
- Other OSes –
  - microkernel
  - extensible OS
  - virtual machines
  - sensor OS
  - Software isolated processes
  - special-purpose OS – for highly concurrent Internet servers
  - still evolving …

Perspectives of the Computer



(a) End User View

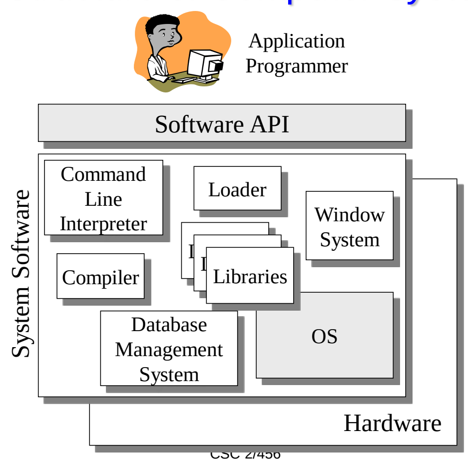
(b) Application Programmer View

(c) System Programmer View

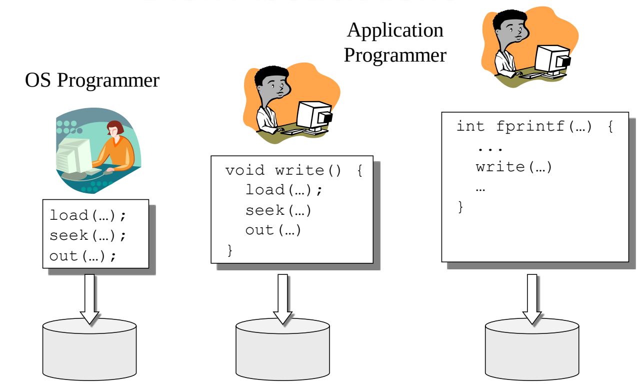
System Software

- A general piece of software with common functionalities that support many applications
- Examples
  - C compiler and library functions
  - Shell – command line interpreter
  - A window system
  - A database management system
  - The operating system
    - A thin layer of software that operates directly on the raw hardware

The Structure of Computer Systems



Disk Abstractions



(a) Direct Control

(b) `write()` abstraction

(c) `fprintf()` abstraction

System Calls and Interfaces/Abstractions

- Examples: Win32, POSIX, or Java APIs
- Process management
  - fork, waitpid, execve, exit, kill
- Exceptions, interrupts (events)
  - signals
- File management
  - open, close, read, write, lseek
- Directory and file system management
  - mkdir, rmdir, link, unlink, mount, umount
- Inter-process communication
  - sockets, ipc (msg, shm, sem)

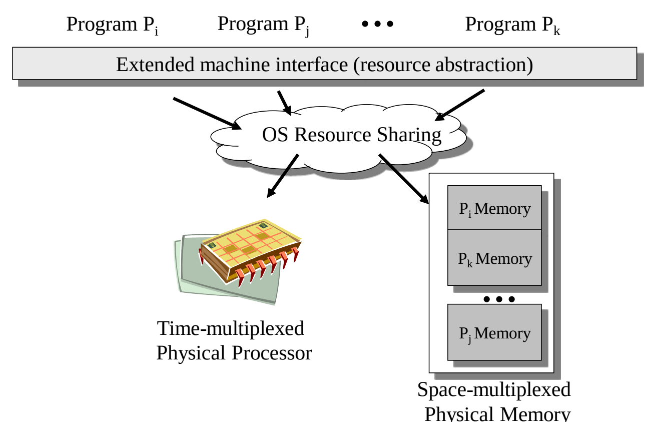
Procfs: The /proc filesystem [Killian'84]

- Processes as files: a pseudo-file system
  - a file system interface to kernel in-memory data structures
- Linux implementation originated with Bell Labs' Plan 9
- Hierarchical file system
  - Each live process has its own directory (numbered with pid)
  - Non-process-related system information in named files: e.g., cpuinfo, meminfo

Under the Abstraction

- *functional complexity*
- *a single abstraction over multiple devices*
- *replication → reliability*
- *resouce sharing*

Resource Sharing



Objectives of Resource Sharing

- Efficiency
- Fairness
- Security/protection

History

- Machine language
- Batch systems (mainframes)
- Multiprogramming and time sharing
- Graphical user interfaces, virtual memory, protection, network/distributed operating systems

Operating Systems Concepts

- Processes
- Memory management
- File systems
- Device management
- Security/protection

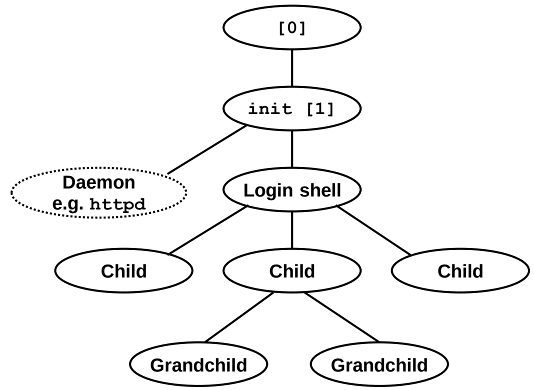
System Boot

- How does the hardware know where the kernel is or how to load that kernel?
  - Use a *bootstrap* program or loader
  - Execution starts at a predefined memory location in ROM (read-only memory)
  - Read a single block at a fixed location on disk and execute the code from that boot block
  - Easily change operating system image by writing new versions to disk

Processes

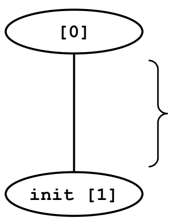
- Def: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private address space
    - Each program seems to have exclusive use of main memory.
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

Unix Process Hierarchy



Unix Startup: Step 1

1. **Pushing reset button loads the `PC` with the address of a small bootstrap program.**
2. **Bootstrap program loads the boot block (disk block 0).**
3. **Boot block program loads kernel binary (e.g., `/boot/vmlinux`)**
4. **Boot block program passes control to kernel.**
5. **Kernel handcrafts the data structures for process 0.**

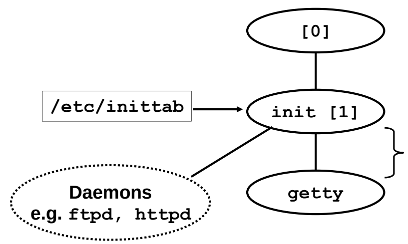


Process 0: handcrafted kernel process

Process 0 forks child process 1

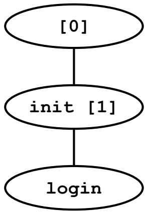
Child process 1 execs `/sbin/init`

Unix Startup: Step 2



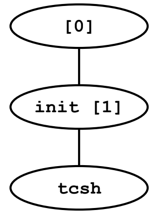
`init` forks and execs daemons per `/etc/inittab`, and forks and execs a `getty` program for the console

Unix Startup: Step 3



The `getty` process execs a `login` program

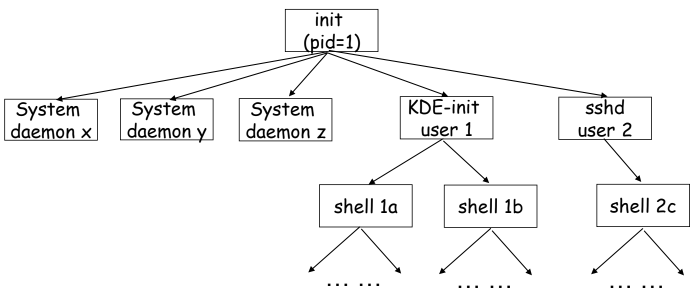
Unix Startup: Step 4



`login` reads login and passwd.
if OK, it execs a *shell.*
if not OK, it execs another `getty`

Process Tree on a Linux System

- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.



Unix: fork, exec; Win32API: CreateProcess

System Protection

- User programs (programs not belonging to the OS) are generally not trusted
  - A user program may use an unfair amount of a resource
  - A user program may maliciously cause other programs or the OS to fail
- Need protection against untrusted user programs; the system must differentiate between at least two modes of operations
  1. *User mode* – execution of user programs
     - o untrusted
     - o not allowed to have complete/direct access to hardware resources
  2. *Kernel mode* (also *system mode* or *monitor mode*) – execution of the operating system
     - o trusted
     - o allowed to have complete/direct access to hardware resources

o Hardware support is needed for such protection

`fork`: Creating new processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's `pid` to the parent process

```
if (fork() == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Fork is interesting (and often confusing) because it is called *once* but returns *twice*

`exit`: Destroying Process

- `void exit(int status)`
  - exits a process
    - Normally return with status 0
  - `atexit()` registers functions to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

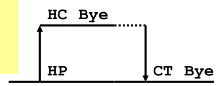

`wait`: Synchronizing with children

- `int wait(int *child_status)`
  - suspends current process until one of its children terminates
  - return value is the `pid` of the child process that terminated
  - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

`wait`: Synchronizing with children

```
void fork9() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



Waitpid

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```
void fork11()
{
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
       if ((pid[i] = fork()) == 0)
           exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
       pid_t wpid = waitpid(pid[i], &child_status, 0);
       if (WIFEXITED(child_status))
           printf("Child %d terminated with exit status %d\n",
                  wpid, WEXITSTATUS(child_status));
       else
           printf("Child %d terminated abnormally\n", wpid);
    }
```

`exec`: Running new programs

- `int execl(char *path, char *arg0, char *arg1, …, 0)`
  - loads and runs executable at `path` with args `arg0`, `arg1`, …
    - `path` is the complete path of an executable
    - `arg0` becomes the name of the process
      - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
    - "real" arguments to the executable start with `arg1`, etc.
    - list of args is terminated by a `(char *)0` argument
  - returns `-1` if error, otherwise doesn't return!

```
main() {
   if (fork() == 0) {
      execl("/usr/bin/cp", "cp", "foo", "bar", 0);
   }
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```

8/29/2012 CSC 2/456 34

User Operating-System Interface

- Command interpreter – special program initiated when a user first logs on
- Graphical user interface
  - Common desktop environment (CDE)
  - K desktop environment (KDE)
  - GNOME desktop (GNOME)
  - Aqua (MacOS X)

8/29/2012 CSC 2/456 35

Assignment #1

- Exclusively outside of the OS
- Part I: observing the OS through the /proc virtual file system
- Part II: building a shell (command-line interpreter)
  - Support foreground/background executions
  - Support pipes

8/29/2012 CSC 2/456 36

Simple Shell `eval` Function

```
void eval(char *cmdline)
{
    char *argv[MAXARGS]; /* argv for execve() */
    int bg;              /* should the job run in bg or fg? */
    pid_t pid;           /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        if ((pid = Fork()) == 0) {   /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }

        if (!bg) {   /* parent waits for fg job to terminate */
           int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else         /* otherwise, don't wait for bg job */
            printf("%d %s", pid, cmdline);
    }
}
```

Disclaimer

- Parts of the lecture slides contain original work from Gary Nutt, Andrew S. Tanenbaum, Dave O'Hallaron, Randal Bryant, and Kai Shen. The slides are intended for the sole purpose of instruction of operating systems at the University of Rochester. All copyrighted materials belong to their original owner(s).