

# CSC 2/456: Operating Systems

Instructor: Sandhya Dwarkadas

TA: Zhuan Chen

10/20/2010

CSC 2/456

1

## General Course Information

- Course Web page:
  - [www.cs.rochester.edu/~sandhya/csc256](http://www.cs.rochester.edu/~sandhya/csc256)
- Course-related announcement/correspondence:
  - Blackboard Discussion Board and e-mail list
- Texts
  - Tanenbaum, "Modern Operating Systems"
  - Silberschatz et al, "Operating System Concepts"

10/20/2010

CSC 2/456

2

## 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

10/20/2010

CSC 2/456

3

## Perspectives of the Computer



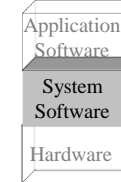
cut save send print



(a) End User View



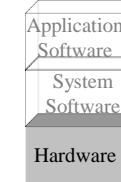
malloc() open() fork()



(b) Application Programmer View



read-disk start-printer push-bits-into-NIC



(c) System Programmer View

10/20/2010

CSC 2/456

4

## System Software

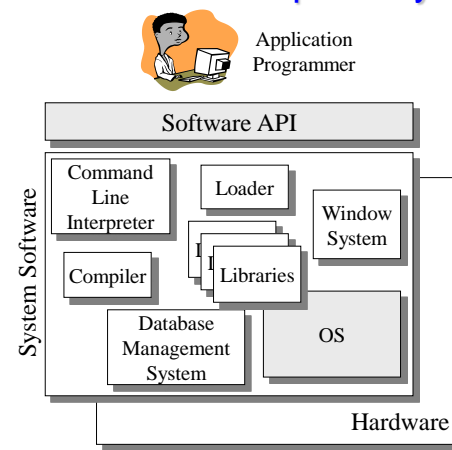
- A general piece of software with common functionalities that support many applications
- Examples
  - C compiler and library functions
  - Shell - command line interpreter
  - A window system
  - A database management system
  - The operating system
    - A thin layer of software that operates directly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

10/20/2010

CSC 2/456

5

## The Structure of Computer Systems



10/20/2010

CSC 2/456

6

## 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

10/20/2010

CSC 2/456

7

## Resource Abstraction

```

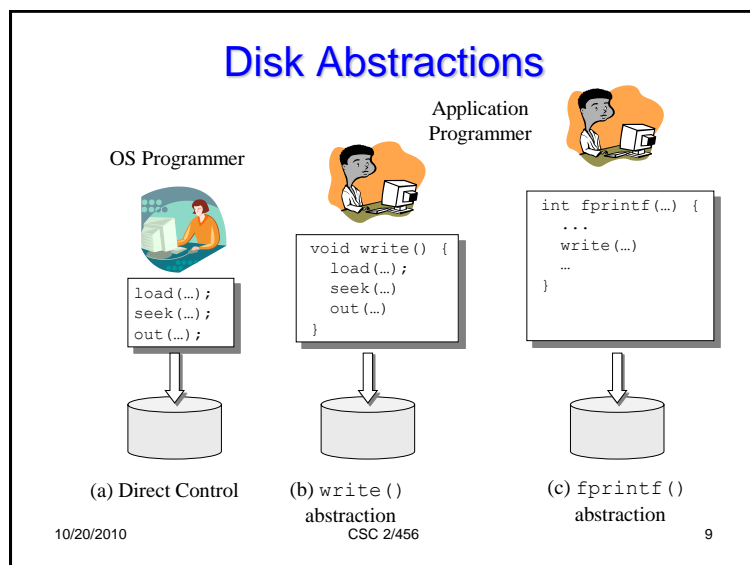
load(block, length, device);
seek(device, track);
out(device, sector)
-----
write(char *block, int len, int device,
      int track, int sector) {
    ...
    load(block, length, device);
    seek(device, track);
    out(device, sector);
    ...
}
-----
write(char *block, int len, int device, int addr);
-----
fprintf(fileID, "%d", datum);

```

10/20/2010

CSC 2/456

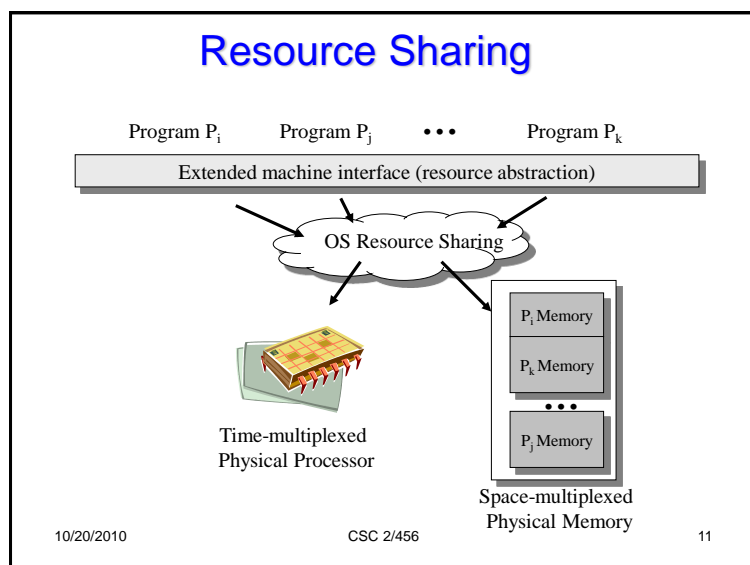
8



### Under the Abstraction

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

10/20/2010CSC 2/45610



### Objectives of Resource Sharing

- Efficiency
- Fairness
- Security/protection

10/20/2010CSC 2/45612

## 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)

10/20/2010

CSC 2/456

13

## History

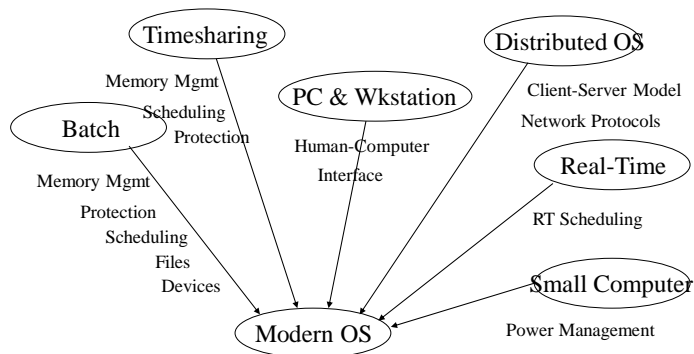
- Machine language
- Batch systems (mainframes)
- Multiprogramming and time sharing
- Graphical user interfaces, virtual memory, protection, network/distributed operating systems

10/20/2010

CSC 2/456

14

## Evolution of Modern OS



10/20/2010

CSC 2/456

15

## Examples of Modern OS

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

10/20/2010

CSC 2/456

16

## 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 efficient/correct application code

10/20/2010

CSC 2/456

17

## Operating Systems Concepts

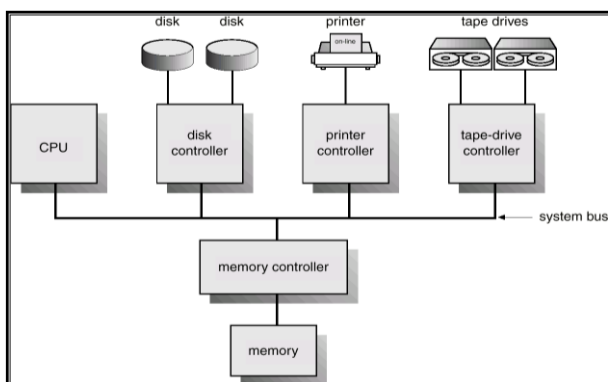
- Processes
- Memory management
- File systems
- Device management
- Security/protection

10/20/2010

CSC 2/456

18

## Computer-System Architecture



10/20/2010

CSC 2/456

19

## 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

10/20/2010

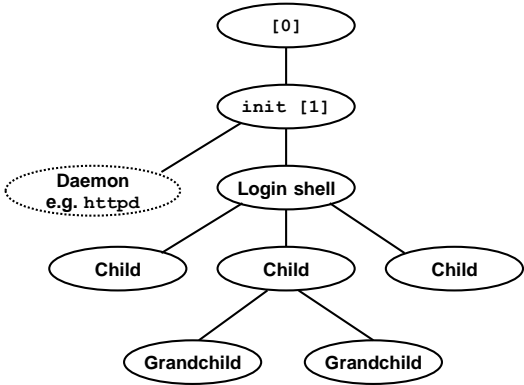
CSC 2/456

20

### Processes

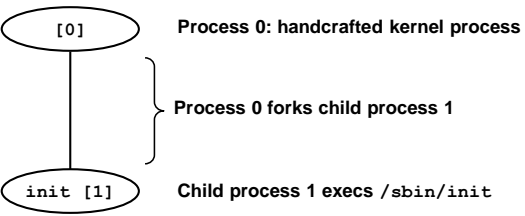
- Def: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science.
  - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - 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

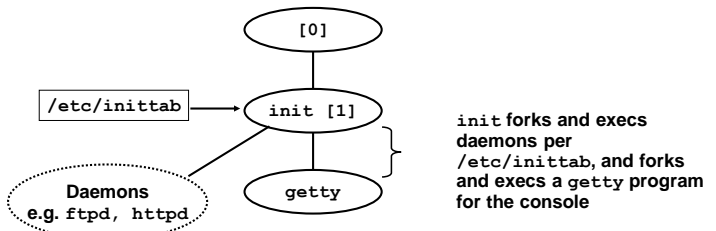


### Unix Startup: Step 1

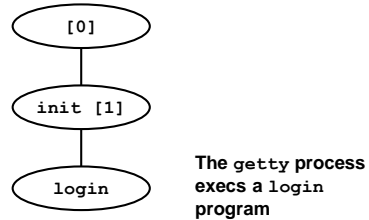
1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., /boot/vmlinux)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process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



### Unix Startup: Step 2



### Unix Startup: Step 3

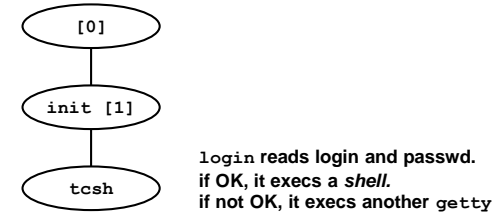


10/20/2010

CSC 2/456

25

### Unix Startup: Step 4



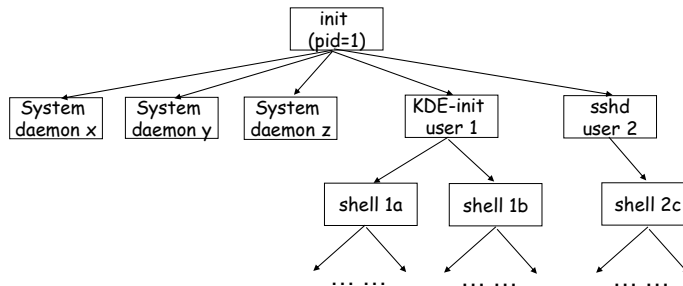
10/20/2010

CSC 2/456

26

### 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

10/20/2010

CSC 2/456

27

### 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
  - User mode - execution of user programs
    - untrusted
    - not allowed to have complete/direct access to hardware resources
  - Kernel mode (also system mode or monitor mode) - execution of the operating system
    - trusted
    - allowed to have complete/direct access to hardware resources
- Hardware support is needed for such protection

10/20/2010

CSC 2/456

28

## 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)

10/20/2010

CSC 2/456

29

## 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*

10/20/2010

CSC 2/456

30

## 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);
}
```

10/20/2010

CSC 2/456

31

## 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

10/20/2010

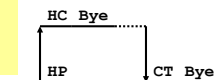
CSC 2/456

32

## 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();
}
```



10/20/2010

CSC 2/456

33

## Waitpid

- waitpid(pid, &status, options)

- Can wait for specific process
- Various options

```
void forkll()
{
    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();
}
```

10/20/2010

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

10/20/2010

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).

10/20/2010

CSC 2/456

38

## Problem with Simple Shell Example

- Shell correctly waits for and reaps foreground jobs.
- But what about background jobs?
  - Will become zombies when they terminate.
  - Will never be reaped because shell (typically) will not terminate.
  - Creates a memory leak that will eventually crash the kernel when it runs out of memory.
- Solution: Reaping background jobs requires a mechanism called a *signal*.

10/20/2010

CSC 2/456

39

## Signals

- A *signal* is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID's
  - The only information in a signal is its ID and the fact that it arrived.

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt from keyboard (ctl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated

10/20/2010

CSC 2/456

40

## Signal Concepts

- Sending a signal
  - Kernel *sends* (delivers) a signal to a *destination process* by updating some state in the context of the destination process.
  - Kernel sends a signal for one of the following reasons:
    - Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
    - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.

10/20/2010

CSC 2/456

41

## Signal Concepts (cont)

- Receiving a signal
  - A destination process *receives* a signal when it is forced by the kernel to react in some way to the delivery of the signal.
  - Three possible ways to react:
    - Ignore the signal (do nothing)
    - Terminate the process.
    - *Catch* the signal by executing a user-level function called a *signal handler*.
      - Akin to a hardware exception handler being called in response to an asynchronous interrupt.

10/20/2010

CSC 2/456

42

## Signal Concepts (cont)

- A signal is *pending* if it has been sent but not yet received.
  - There can be at most one pending signal of any particular type.
  - Important: Signals are not queued
    - If a process has a pending signal of type *k*, then subsequent signals of type *k* that are sent to that process are discarded.
- A process can *block* the receipt of certain signals.
  - Blocked signals can be delivered, but will not be received until the signal is unblocked.
- A pending signal is received at most once.

10/20/2010

CSC 2/456

43

## Signal Concepts

- Kernel maintains *pending* and *blocked* bit vectors in the context of each process.
  - *pending* – represents the set of pending signals
    - Kernel sets bit *k* in *pending* whenever a signal of type *k* is delivered.
    - Kernel clears bit *k* in *pending* whenever a signal of type *k* is received
  - *blocked* – represents the set of blocked signals
    - Can be set and cleared by the application using the `sigprocmask` function.

10/20/2010

CSC 2/456

44

## Process Groups

- Every process belongs to exactly one process group

- `getpgrp()` – Return process group of current process
- `setpgid()` – Change process group of a process

10/20/2010 CSC 2/456 45

## Sending Signals with `kill` Program

- `kill` program sends arbitrary signal to a process or process group

```
linux> ./forks 16
linux> Child1: pid=24818 pgrp=24817
Child2: pid=24819 pgrp=24817
```

```
linux> ps
PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24818 pts/2        00:00:02 forks
24819 pts/2        00:00:02 forks
24820 pts/2        00:00:00 ps
linux> kill -9 -24817
linux> ps
PID TTY          TIME CMD
24788 pts/2        00:00:00 tcsh
24823 pts/2        00:00:00 ps
linux>
```

- Examples
  - `kill -9 24818`
    - Send SIGKILL to process 24818
  - `kill -9 -24817`
    - Send SIGKILL to every process in process group 24817.

10/20/2010 CSC 2/456 46

## Sending Signals from the Keyboard

- Typing `ctrl-c` (`ctrl-z`) sends a SIGINT (SIGTSTP) to every job in the foreground process group.
  - `SIGTERM` – default action is to terminate each process
  - `SIGTSTP` – default action is to stop (suspend) each process

10/20/2010 CSC 2/456 47

## Receiving Signals

- Suppose kernel is returning from exception handler and is ready to pass control to process  $p$ .
- Kernel computes `pnb = pending & ~blocked`
  - The set of pending nonblocked signals for process  $p$
- If (`pnb == 0`)
  - Pass control to next instruction in the logical flow for  $p$ .
- Else
  - Choose least nonzero bit  $k$  in `pnb` and force process  $p$  to receive signal  $k$ .
  - The receipt of the signal triggers some *action* by  $p$
  - Repeat for all nonzero  $k$  in `pnb`.
  - Pass control to next instruction in logical flow for  $p$ .

10/20/2010 CSC 2/456 48

## Default Actions

- Each signal type has a predefined *default action*, which is one of:
  - The process terminates
  - The process terminates and dumps core.
  - The process stops until restarted by a SIGCONT signal.
  - The process ignores the signal.

10/20/2010

CSC 2/456

49

## Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - SIG\_IGN: ignore signals of type `signum`
  - SIG\_DFL: revert to the default action on receipt of signals of type `signum`.
  - Otherwise, `handler` is the address of a *signal handler*
    - Called when process receives signal of type `signum`
    - Referred to as "*installing*" the handler.
    - Executing handler is called "*catching*" or "*handling*" the signal.
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

10/20/2010

CSC 2/456

50

## 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).

10/20/2010

CSC 2/456

51