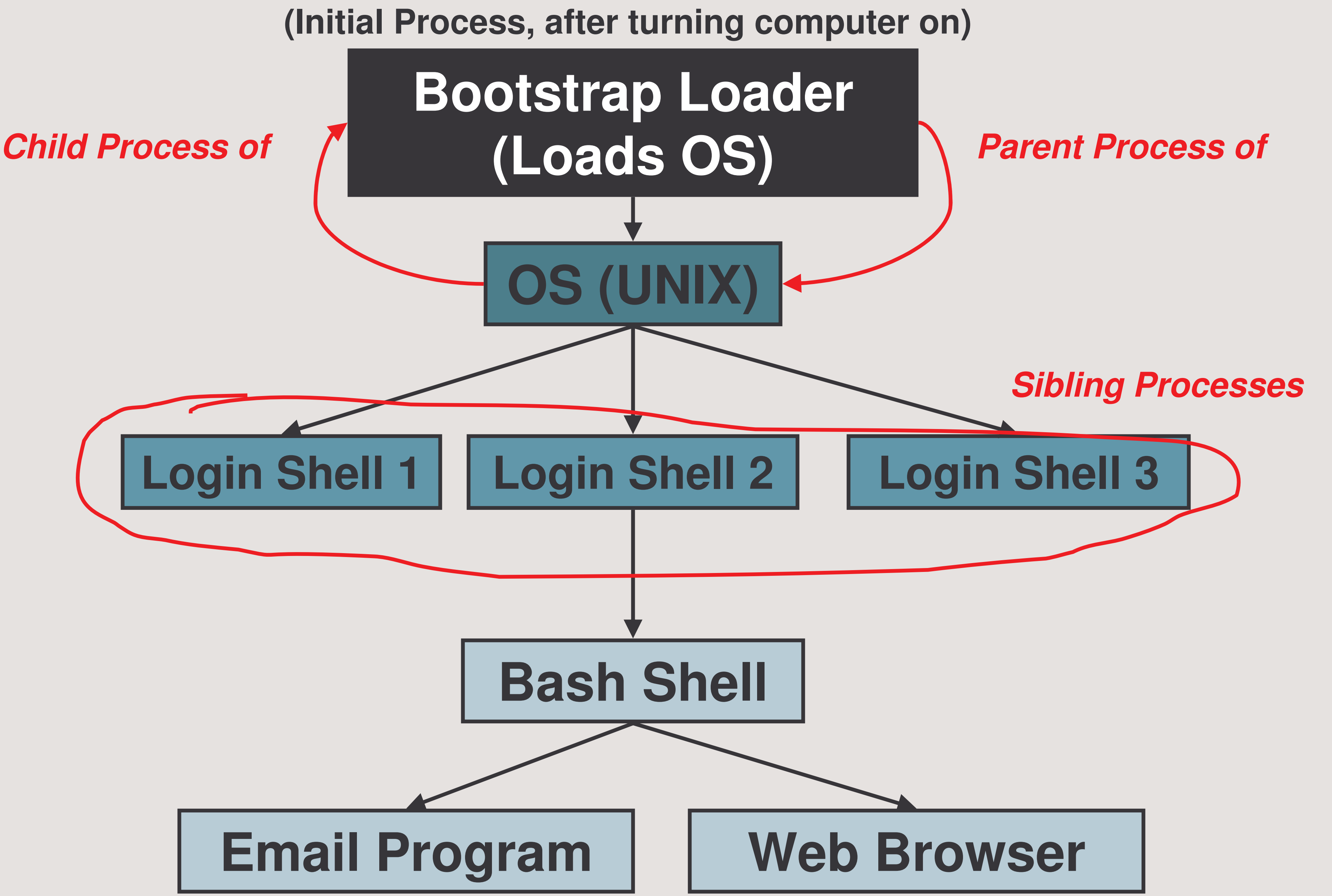
Operating System Organization

COMP 229 (Section PP) Week 7

Prof. Richard Zanibbi
Concordia University
Feb. 27, 2006

Last Week...

- **Classic Processes** **(e.g. in "Old" UNIX)**
- **Multithreaded Processes** **(e.g. in Linux, Windows)**
- **File (e.g. `open()`) and Process (e.g. `fork()`, `CreateThread()`) System Calls**
- **Creating Processes and Threads**
- **Resource Management**
- **Resource Descriptors (*file* and *process*)**
- **Overview of Writing Concurrent Programs**
- **Objects and Operating Systems**
- **Why do shells run commands and programs in child processes?**


(Initial Process, after turning computer on)
Bootstrap Loader (Loads OS)
Child Process of
Parent Process of
OS (UNIX)
Sibling Processes
Login Shell 1
Login Shell 2
Login Shell 3
Bash Shell
Email Program
Web Browser

This week...
Operating System Organization

Basic OS Functions (pp. 89-94)

1. Device Management
2. Process and Resource Management
3. Memory Management
4. File Management

Implementation Issues (pp. 94-103)

Processor Modes
Kernels
Requesting OS Services
Software Modularization

Contemporary OS Kernels (pp. 104-110)

UNIX Kernels
Windows NT Executive Kernel

Basic OS Functions

(pp. 89-94, Part II)

Basic OS Functions

As Approximate Notion....

- There is no formal definition of the functions of an operating system.
- OS designs influenced by engineering issues, workplace/marketplace demands
  - Is the OS for a cell phone or desk-top?
  - For a research lab or home user?
  - "What will help us get a grant/sell copies"?

A Traditional View of Basic Functions:

1. Process, thread, and resource management
2. Memory Management
3. File Management
4. Device Management

1. Device Management

Concered With:

- Allocation, isolation, and sharing of 'generic devices' (i.e. *not* including the processor and memory)
- Most OS's treat generic devices roughly the same
- Applies policies chosen by the designer or system administrator

Device-Dependent Component

Contains device drivers (unique for a device)

Implement device functionality, software interface

- e.g. drivers for monitors, keyboards, mp3 players...

Device-Independent Component

A software environment in which device-dependent drivers execute; provided in the basic OS

- Includes system call interface to read/write any registered device
- Also include a mechanism to forward calls to devices
- Small part of the device manager

Example

See Figure 3.2

- Note that “Device Dependent Clouds” are drivers
- Example devices: keyboard, monitor, mp3 player

Adding a Device to a System Using the Device Manager (Simplified Version)

1. Inform device-independent component to **register** the new device
2. Device-independent component **makes the device system call interface available** for the new device
3. Device Driver is **installed** (loaded onto a storage device in the system)
4. When system call for the new device is later made, the **driver implements the system call interface** actions

2. Process, Thread, and Resource Management

Implements Abstract Machines

...including scheduling processes/threads and managing their associated resources

- usually implemented as a single module, defining the abstract machine environment as a whole
- If OS allows threaded processes, then processes and threads must be managed separately

Resource Manager

- Allocates and book-keeps available and utilized resources
- Change in resource often related to change in a process: so resource management often discussed as part of process management

Process, Thread, and Resource Management, Cont'd

Process Manager

- Provides multiple abstract machines (execution contexts)
- Allows multiple users, processes and/or threads to share the processor

Key issue: how will processor time be allocated between processes?

See Fig. 3.3

Abstract Resource example: a File

3. Memory Management

Functions

- Cooperates with Process Manager to allocate main ("primary") memory to processes
- Applies allocation policy given by system administrator

Resource Isolation

- Enforces resource isolation for memory
- Provides means to bypass isolation so processes can share memory

Memory Management, Cont’d

Virtual Memory

- an Abstract Resource which allows abstract machines to request more memory than is physically available
- Achieved by writing/retrieving memory blocks on storage devices (e.g. disk)

Distributed Shared Memory Abstraction

- allows a thread on a machine to access and share physical memory on another machine
- Achieved using message-passing over a network

See Figure 3.4

- Remember that the Process Manager and Memory Manager co-operate (i.e. communicate) extensively: in class, there was an error added to indicate this.

4. File Manager

Files as Abstract Resource

- Recall: files allow storage devices to be represented in a simple, uniform way
- Memory is volatile, and may be overwritten as new processes have their turn on the processor
- File manager implements the file system (e.g. directory tree), used to store memory when it needs to be stored

File Manager

- Interacts with the device manager and memory manager
- Modern era: may be distributed over a network of machines, to allow systems to read/write files on remote machines

Implementation Issues

(pp. 94-103, Part II)

Implementation Mechanisms

Recurring Issues

1. Performance (“speed”, maximize use of resources)
2. Resource Isolation
   - esp. to allow processes to save data without a risk it will be corrupted

Implementation Mechanisms

1. Processor modes: *user* vs. *supervisor* modes
2. Kernels (trusted software module used to support all other software)
3. Invoking system services: either through system functions, or sending messages to a system process

Abstraction vs. Overhead

OS-provided Abstractions

Simplify programming and system management

Use resources (e.g. “takes time”)

OS Design Trade-offs

- Designers need to balance the benefit of OS features for users/programmers against the resources needed to implement the feature
- As hardware improves, more abstraction becomes reasonable (e.g. graphics, windows)

Resource Isolation

Purpose (again)

Roughly, we want to prevent processes from interfering with one another (e.g. overwriting one another's data in memory or on disk)

Protection Mechanisms, Security Policies

- OS provides protection mechanisms (e.g. file access policies) for implementing a *security policy* on a machine
- Security policy is chosen by sys. admin.

Kernel

- Implements all secure operations (and is *trusted*)
- Provides a barrier between between "trusted" routines in the kernel that manage system operation, and all other system and application software
- All software outside of the kernel is *untrusted,*

Processor Modes

Mode Bit

- Provided in modern processors to determine which instructions and memory locations are available to an executing program
- Allows resource isolation to be enforced

Trusted Software (e.g. Kernel)

- Uses processor in supervisor mode (all operations)
- Operations run only in supervisor mode: *supervisor, privileged, or protected*
  - e.g. I/O operations are protected
- Can access all memory (system space)

Untrusted Software

- Uses processor in user mode (restricted operation set)
  - e.g. to perform I/O, applications must request the OS to perform the operation
- Can access restricted portion of memory (user space)

Examples

See Fig 3.5

- processor is using register accessible only to the executing process
- Only a privileged instruction run by a program running in supervisor mode can load the contents of object pointer register shown

See Fig 3.6

Memory accessible in user mode (“user space”) is a subset of the supervisor space (“system space”)

Mode Bits, cont'd

Older Systems (e.g. 8088/8086)

- Did not have a supervisor mode
- Made it hard to provide robust isolation (*lots* of rebooting)

User Programs Requesting OS Services from the Kernel: Trap Instructions

Trap Instructions

- similar to a hardware interrupt
- used by programs running in user mode to request OS services
- OS services in the kernel are run in supervisor mode

Only kernel functions have access to kernel data structures (e.g. resource descriptors)

Two Methods for Using Traps (see Fig 3.7)

System Calls (e.g. Figures 3.8, 3.9)

– User process issues a “trap” instruction; results in a reference into a trap table

– “Stub” functions are provided to hide some details of the trap call (e.g. fork())

Message Passing

– User process issues a send() system call that communicates with an OS process

– OS process executes the trap instruction in supervisor mode, returns a message to the calling user process

Basic Operating System Organization

See Figure 3.10

Notice that the only two OS managers we’ve looked at that *don’t* communicate extensively are the Memory and Device managers.

Interaction of Basic OS Components: Modularity vs. Performance

Modularity of Components

Would allow modules to keep relevant data structures separate from other modules

Performance

Using additional functions to modify OS data structures slows performance down

Traditionally, components implemented in one module (e.g. the Unix kernel)

Monolithic Kernel

All components implemented in one module (e.g. Unix)

Microkernels

- Only operations that must be trusted are implemented in the kernel (e.g. thread scheduling, hardware device management)
- Goal: minimize trusted code size
- Results in many system calls into the micokernel

Contemporary OS Kernels

(pp. 104-110, Part II)

***** See Figures 3.11, 3.12***

Note that the Windows NT/2000/XP Kernel is similar to a microkernel

Next Week...

Process and Resource Management

read pages 199-228 (Part II, Chapter 6)