System Software for Program Execution: User-level View of Operating Systems

COMP 229 (Section PP) Week 6

Prof. Richard Zanibbi
Concordia University
February 13, 2006

Last Week...

Supports Programming,
Efficient System Use

System Software for Program Execution (pp. 1-18)

Systems and application software (again)

Resource Abstraction

Resource Sharing

Applies System Software
to a Problem

Operating System Strategies (pp. 18-39)

A brief history of computing and operating systems

(batch, timesharing, personal, SCC's, embedded, networks)

including
BIOS

The modern OS as integration/evolution of previous systems

Goals:

1. Learn the function and operation of operating systems
2. Learn how to exploit OS design during programming (for program execution)

"Hacker notes"

Resource Abstractions

Files

Represent data on storage devices (e.g. disk)

Represent data structured as a list

Abstract Machines

Represent resources for executing programs

Program running on abstract machine: a “process”

Good Abstractions for Programmers:

Easy to understand

Provide necessary control (not “too abstract”)

Resource Sharing

Resource Sharing

- **Time-multiplexed:** share time on resource (“take turns”, e.g. processor)
- **Space-multiplexed:** share space on resource (e.g. memory)

Transparent Resource Sharing

- Sharing “invisible” to programmer (done by OS)
- e.g. simulating multiple abstract machines (multiprogramming)

Explicit Resource Sharing

- Programmer requests resource sharing
- e.g. sharing memory between programs

Resource Isolation

- OS permits one process (abstract machine) access to a resource at a time

Security and Consistency Issues

- Prevent problematic or malicious access to resources

Example

(Multiprogramming "best case" performance, Fig. 1.8)

Operating System Strategies

OS Strategy

General characteristics of the programmer's abstract machine

Early Days (approx. pre-1960)

One program at a time, basically no OS

Strategies

1. **Batch systems:** executed non-interactively, multiprogramming
2. **Timesharing systems:** multiple users interacting with the system; extended multiprogramming, security
3. **Personal Computers, and Workstations:** single-user machines; multiprogrammed, minimize wait time rather than max. hardware use (*many had no file system (!); provided by CP/M, MS-DOS)*
4. **Embedded systems:** guaranteed response times ("real time")
5. **Small, Communicating Computers (SCC's):** new resource management, power management, device management
6. **Network Technology**: handle resources, information on networks

This week...
User-Level View of Operating Systems

The Programmer's Abstract Machine (pp. 42-47)

- Sequential Computation ("single thread")
- Multithreaded Computation

Process Resources (pp. 47-52)

- esp. files (under UNIX, Windows)

More on Processes and Threads (pp. 52-58)

Writing Concurrent Programs (pp. 58-72)

- Executing Computations
- Executing Commands in UNIX and Windows

Objects (pp. 72-74)

The Programmer's Abstract Machine

(pp. 42-52)

Sequential Computation

Algorithm

A process or set of rules for calculation or problem-solving, esp. with a computer (CA Oxford Dictionary)

Sequential algorithms

Describe a *sequence* of operations

Fundamental abstraction for computation

Examples

Sorting and Search Algorithms

(e.g. Vol.3,"The Art of Computer Programming" (Donald Knuth))

Algorithms May be Defined Using

- Natural Language (e.g. English or French)
- Pseudocode ("informal" operation sequence)
- Mathematics
  - may not provide an explicit operation sequence
- As a Source Program in a Programming Language (e.g. C/C++, Java)
  - explicit sequence of operations (*unambiguous*)

System Calls

Binary Program

- Machine-readable translation of source program (e.g. “load module”, or executable program (.exe))
- Represents explicit machine instructions used to implement an algorithm (e.g. op. sequence)

System Calls in Binary Programs

- Procedure calls to the system call interface invoke OS features (e.g. stop a program, read data from a device)
- System calls allow the programmer to take advantage of resource abstractions and sharing built into the OS

What is POSIX?

Portable Operating System Interface

Developed to standardize UNIX system call interfaces

POSIX.1: published 1988

Now used by a large number of UNIXes, including Linux

- Linus Torvalds, Univ. Helsinki, Finland, first announced Linux on the Minix newsgroup in 1991

Sequential Computation and Multiprogramming

Running a Program

1. Translate source to binary program (and possibly link)
2. Provide program input data, parameters
3. Direct OS to start executing instructions from the transfer address ("main entry point") (includes OS calling program loader)
4. Program halts when last statement is executed or system call for halt (`exit()`) is made

Execution Engine (part of a process)

Used by OS to implement multiprogramming

Status of abstract machine running the process

Copy of the runtime stack for the process

Runtime Stack

Used to implement "scopes" in languages (among other things)

Contains local variables, return addresses, etc.

Fig. 1.2: "Classic" processes (e.g. UNIX)

Multiprocessing vs. Multithreading

"Classic" Process

Sequential program "run" alone on its own abstract machine

Multiprocessing

Concurrent execution of processes

Processes contain separate resource abstractions, binary program copies

Multithreading

Thread: another name for "execution engine"

- "Classic" processes have a single thread

Multithreading: multiple threads within a single process

Threads within a process share resources, including binary program

Example: Fig. 2.6 (Multithreaded process, as found in Windows NT/XP, Linux with POSIX threads extension (2.2+))

Example: Fig 2.3 ("accountant example")

Implementing Threads

Example: Java

Java Virtual Machine (JVM) supports threads

`Thread` base class supports threads

Create subclass of `Thread` (e.g. `MyThread`): instances define a new thread

Executing Threads

– If multiple processors are available, threads may be executed in parallel

– Otherwise (usually) threads share the processor (through time-multiplexing)

Process Resources

(pp. 47-52)

OS Management of Resources Used by Processes

Resource Allocation

Threads access resources through system calls

(*insures resource isolation*)

1. Threads must request a resource before use
2. After request, thread is suspended until resource is allocated to it
   - while suspended, other threads/processes allocated processor time
3. After a resource is allocated to a thread, all threads in its associated process may share access to the resource (the process "owns" the resource)

Memory and Processors

Traditionally distinguished from other resources

- Processes/threads (implicitly) request processor when ready
- When user process is started, loader requests memory from the OS to run the program (e.g. user login -> shell -> interaction)

Resource Descriptors

- Data structures used to represent physical and abstract resources managed by the OS
- Indicates if resource is available
- Includes record of which processes are currently blocked waiting for the resource

e.g. file descriptors, process descriptors

Files and File Descriptors

Files

- Named, linear stream of bytes stored on a device
- Common (default) method for representing data storage
- Forms the basis of many other device abstractions
- OS maintains record of which files are currently open

File Lock

OS preventing processes other than currently allocated from accessing a file (e.g. for writing)

File Descriptor

OS's internal representation of file attributes (for file resources)

- e.g. Indicates if file is locked, or read-only

UNIX: a non-negative integer (file descriptor (*inode)* number)

Windows: "`HANDLE`": reference to internal data structure

(Internal) OS Representation of Byte Stream Files

File Pointer

On file open:
(default) pointer set to 0

Read/Write K bytes:
Advance pointer by K

Setting File Pointer:
`lseek()` (POSIX)
`SetFilePointer()` (Win)

No "Type" for Bytes:
Treated as "raw" bytes

Name: Test

Byte	Value	(ASCII)
0	0100 0001	A
1	0100 0010	B
2	0100 0011	C
3	0100 0100	D
4	0100 0101	E
5	0100 0110	F
6	0100 0111	G

Example: POSIX File System Calls

System Call	Effect
`open()`	Open file for read or write. OS creates internal representation, optionally **locks** the file. Returns a file descriptor (integer), or -1 (error)
`close()`	Close file, releasing associated locks and resources (e.g. internal representation)
`read()`	Read bytes into a buffer. Normally blocks (suspends) a process until completion. Returns # bytes read, or -1 (error)
`write()`	Write bytes from a buffer. Normally blocks (suspends) a process until completion. Returns # bytes written, or -1 (error)
`lseek()`	Set the file pointer location
`fcntl()`	("File Control"): various options to set file attributes (locks, thread blocking, ...)

Rough Equivalences for POSIX and Win32 API

POSIX	Win32
`open()`	`CreateFile()/ OpenFile()`
`close()`	`CloseHandle()`
`read()`	`ReadFile()`
`write()`	`WriteFile()`
`lseek()`	`SetFilePointer()`

Examples

UNIX file system calls, Fig. 2.4

Windows file system calls, Fig. 2.5

Using File to Represent Other Resources: Devices and Pipes in UNIX

Other Resource Abstractions

e.g. keyboard, display, memory, processors

– The more similar these abstractions are, the easier they are for programmers to use

UNIX Devices

Devices are represented as files in the directory tree

– Devices: located in the **`/dev`** directory (e.g. **`/dev/tty`**)

– Devices have **`open(), close(), read(), write(), seek(), fcntl()`** commands implemented in their drivers (similar to file interface)

UNIX Pipes

Abstract resource allowing processes to communicate by chaining outputs to inputs

e.g. > **`cat schedule.txt | grep deadline`**

– Inputs/outputs are represented using standard files for each process (**standard input** and **standard output**)

More on Processes and Threads

(pp. 52-74)

Heavyweight vs. Lightweight Threads

Heavyweight ("Classic") Process

Processes which may contain exactly one thread (execution engine)

Lightweight Process (= "Thread")

- A thread (execution engine) within a multithreaded process
- "Light" because each thread does not require the overhead of a separate process

Illustration/Example: Fig. 2.6

Thread Representation (Data) in Detail

Runtime Stack

Data private to the thread (e.g. local variables)

Status

OS data structure: all properties unique to the thread

– program counter value
– whether thread is blocked waiting for a resource
– which resource is being waited for
– etc.

The Move Towards Threads

Lightweight Processes (Threads)

Became popular in late 1990's (esp. w. Java)

Motivation: Sharing Resources

- Server managing shared file systems
- Windowing systems using threads to represent individual windows within the physical display

Example: Fig. 2.7 (window threads)

Thread Libraries vs. True Threading

Thread Libraries (Early 1990’s)

- Used to run threads within heavyweight (“classic”) processes (e.g. Mach C thread library)
- OS still implemented classic processes only
- **Problem:** if one “thread” blocks, then all threads block

“True” Threading

- OS implements and manages threads independently of one another
- If a thread in a multithreaded process blocks, the other threads can still execute

Creating Processes and Threads

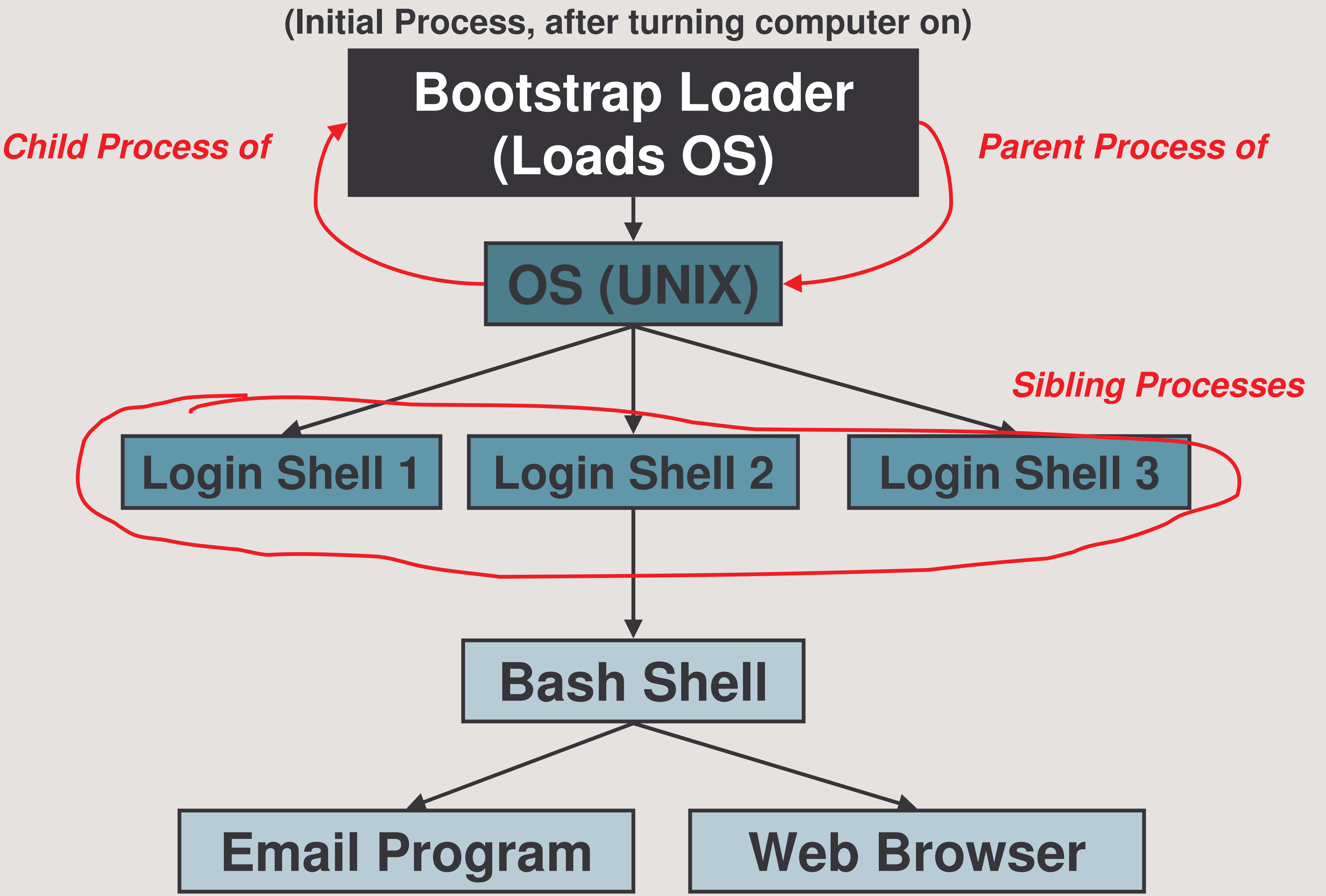
Initial Process

- The first process started when machine is turned on
- Usually the bootstrap loader loading the OS into memory

Creating New Processes and/or Threads

Done using system calls (i.e. another resource request)

- If threads are available, separate system calls are used for creating threads and creating 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

FORK(), JOIN(), and QUIT() System Calls (Conway, 1963; Dennis, Van Horne 1966)

Purpose

Define cooperating sequential processes executing on shared data
Like threads, processes can share resources (e.g. binary program copy)

FORK(label)

- **Creates a child process** that begins execution at “label”
- Parent (calling) process continues execution after child is created
- Parent, child are executed concurrently

QUIT()

Terminates and deallocates (removes) the process issuing the call

JOIN(count)

- **Used to merge processes**
- **count** is an integer variable shared by a parent and its child processes
- Locks processor until system call is finished (prevents inconsistency)

Effect of JOIN(count)

/* Decrement shared variable */

```
count = count – 1;
```

/* QUIT unless this is the last process */

```
if (count != 0) QUIT();
```

(example provided pp. 56-57)

Creating "Classic" (Heavyweight) Processes (~1970)

Classic Process

Single-threaded

Separate resource representation per process

Each has a separate, private address space

Address Space

- Physical resources (mainly memory) that can be referenced by the execution engine (single thread)
- Contains stack, status, data, and program ("text")
- A memory protection mechanism: prevents writing over memory allocated to other processes

Creating "Classic" (Heavyweight) Processes Using `fork()` in POSIX/UNIX

`fork()`

- Creates new process
- Child is given copy of current address space
- Parent and child address spaces are *separate*
  - e.g. changing a variable in the child process does not change the variable's value for the parent

Process Communication In UNIX

- Only permitted through open files

Example: Creating a new process using fork()

1. fork() instruction is executed, causing a system call
2. OS copies data (including address space) from the original process to the child process
3. The child process starts execution at the instruction following the fork() instruction

Creating Modern and (Lightweight) Processes

Creating Modern Process

- Created by system call from a thread
- Child created with separate address space, resources
- A base thread must be created to execute the process

  e.g. `CreateProcess()` in Windows API

Threads Creating Child Threads within a Process

- Again, using a system call from a thread
- Similar to Conway FORK(): child thread runs within the process containing the parent thread
- Child thread has stack and status separate from parent

  e.g. `CreateThread()` in Windows API

Example: Creating a New Thread in a Modern Process

1. **CreateThread(...) is executed**
   - NOTE: unlike fork(), CreateThread() has many parameters
2. **OS creates thread data within the process address space (i.e. within the *same* process)**
3. **OS starts execution of child thread at the indicated starting point**

Writing Concurrent Programs

(pp. 58-72)

Multiple Single-Threaded Processes: the UNIX Model

Unix Process Behaviour Defined By

Text segment (program instructions)

Data segment (static program data)

Stack segment (run-time stack: local variables)

**located in the address space*

UNIX Executable (Binary) Programs

Compiled/Linked programs define the text, data, and stack segments ("`a.out`")

Example: Class UNIX processes (Fig. 2.10)

Note that the address space contains the Status, Stack Segment, Text Segment, and Data Segment

Process Descriptors and Context Switching

Process Descriptor

OS Data Structure representing process attributes, incl.:

- PID: Process Identifier (a reference to the process descriptor)
  - UNIX: PID is integer index for OS-internal table of process descr's
- User that created the process
- Parent, Sibling, and Child processes
- Resources held by process (e.g. file descriptor references)
- Address space for the process
- Threads within process
- State and Stack Location (in memory) *(for classic process)*

Context Switching

Refers to switching the active process and/or thread

May involve swapping a process/thread out of memory and storing it on a device (e.g. disk), among other ops.

UNIX Process Commands

"ps" command

- Lists process identifiers (PID) for the user
- "ps –aux": lists all processes in the system

int fork()

- Returns PID of child process to the parent
- Returns 0 to child process
- Child given copy of text, data, stack segments, access to all open file resources
- Parent process descriptor also copied
- The next process run may be the child, parent, or some other process

fork() Examples

Example 1: What do the parent and child output?

```
theChild = fork()
printf("My PID is %d\n",theChild)
```

Example 2: Directing child process execution

```
childPID = fork()
if (theChild == 0) {
    /* child executes here */
    codeForTheChild();
    exit(0);   // Important!
}
/* Parent executes here */
...
```

Changing Program Data Dynamically: `execve()`

```
int execve(char *path,char *argv[],char *envp[]);
```

program *arguments* *environment variables*

Effect of execve()

- Replaces text, data, and stack areas using the program loader
- After loading, stack is cleared, variables initialized
- Program is started (call does not return)

System Calls to Wait for Child Process to Stop:

wait(): block until any child process terminates

waitpid(P): block until child process with PID *P* terminates

Multiple Modern Processes (e.g. Windows)

Process Creation (`CreateProcess()`)

Flexible: many more parameters than in fork() *(10!)*

Base process can start anywhere in program code

Returns two resource descriptor references:

- one for the new process
- one for the new base thread

Thread Creation (`CreateThread()`)

Again, flexible, this time with 6 parameters

A thread may create a sibling thread

(see text pp. 66-67 for function signatures, discussion)

Executing Commands in UNIX and Windows Shells

Fatal Errors

Unrecoverable process errors

- e.g. trying to write outside a processes' address space

System Shells

Users needs to be able to execute any program, however buggy, but *without crashing the shell*

Solution:

Create a child process to execute each command

- e.g. using **`fork()`** or **`CreateProcess()`**

Objects

(pp. 72-73)

Objects: Another Abstraction

Objects

- Originally used to represent independent “computation units” in simulations; similar to a small process (e.g. workflow simulation)
- Have private data and operations, along with message passing operations (e.g. get/set operations)
- Can be used to represent a coordinated execution of distributed computational units (as was common in simulations)

Classes

- **Simula 67**: introduces notion of classes to define object behaviour (similar to programs for threads/processes)
- First widespread application: interfaces (e.g. InterViews, 1989)
- Java: OS calls into the Java Virtual Machine (JVM), which runs on top of the native OS (*a simulation in an abstract machine!*)
- OS may be designed to support efficient object implementations
- Early attempts at “pure” OO operating systems: efficiency issues

Windows NT/2000/XP Kernel and Objects

Primitive OS Components

Defined as objects

Implemented as C functions: no language inheritance

Threads

- Creating a thread generates a kernel *Dispatcher Object*, which then has additional status fields specific to threads added ("pseudo inheritence")
- This approach allows Kernel operations to be applied uniformly to a broad class of abstract data types

Summary

- Classic Processes (e.g. in “Old” UNIX)
- Multithreaded Processes (e.g. in Linux, Windows)
- Creating Processes and Threads
- Resource Descriptors (*file and process*)
- Resource Management
- File (e.g. `open()`) and Process (e.g. `fork()`, `CreateThread()`) System Calls
- Overview of Writing Concurrent Programs
- Objects and Operating Systems

Next Week...

Operating Systems Organization

pages 89-110

Process Management

pages 197-206