CSCI-GA.2250-001

Operating Systems

Lecture 1.5 (somewhere between 1 and 2):
Structure of Operating Systems

Hubertus Franke
frankeh@cims.nyu.edu
Recap: What is an OS?

• **Code that:**
  – Sits between programs & hardware
  – Sits between different programs
  – Sits betweens different users

• **Job of OS:**
  – Manage hardware resources
    • Allocation, protection, reclamation, virtualization
  – Provide services to app. **How? → system call**
    • Abstraction, simplification, standardization
A peek into Unix/Linux

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

User space/level

Kernel space/level

- User/kernel modes are supported by hardware
- Some systems do not have clear user-kernel boundary
Unix: Application

Application (E.g., emacs)

Libraries

Portable OS Layer

Machine-dependent layer

Written by programmer
Compiled by programmer
Uses function calls
Unix: Libraries

Application

Libraries (e.g., stdio.h)

Portable OS Layer

Machine-dependent layer

Provided pre-compiled
Defined in headers
Input to linker (compiler)
Invoked like functions
May be “resolved” when program is loaded
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

system calls (read, open..)
All “high-level” code
Typical Unix OS Structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

Bootstrap
System initialization
Interrupt and exception
I/O device driver
Memory management
Kernel/user mode switching
Processor management
System Call

- Invoked via non-priviliged instruction
  - TRAP
  - Treated often like an interrupt, but its “somewhat” different

- Synchronous transfer control

- Side-effect of executing a trap in userspace is that an exception is raised and program execution continues at a prescribed instruction in the kernel → syscall_handler()
Steps in Making a System Call

Example:
read (fd, buffer, nbytes)
System Calls (POSIX)

- System calls for process management
- Example of fork used in simplified shell program

```c
#define TRUE 1
while(TRUE) {
    type_prompt();
    read_command(command, parameters);
    if (fork()! = 0) {
        /* some code*/
        waitpid(-1,&status, 0);
    } else {
        /* some code*/
        execve(command, parameters,0);
    }
}
```

Portable Operating System Interface for Unix (IEEE standard 90’s)
System Calls (POSIX)

• System calls for file/directory management
  – \texttt{fd=open(file,how,....)}
  – \texttt{n=wride(fd,buffer,nbytes)}
  – \texttt{s=rmdir(name)}

• Miscellaneous
  – \texttt{s=kill(pid,signal)}
  – \texttt{s=chmod(name,mode)}
System Calls (Windows Win32 API)

• **Process Management**
  – `CreateProcess` - new process (combined work of fork and `execve` in UNIX)
    • In Windows - no process hierarchy, event concept implemented
  – `WaitForSingleObject` - wait for an event (can wait for process to exit)

• **File Management**
  – `CreateFile`, `CloseHandle`, `CreateDirectory`, ...
  – Windows does not have signals, links to files, ..., but has a large number of system calls for managing GUI
List of important syscalls

<table>
<thead>
<tr>
<th>Posix</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Process Management</strong></td>
</tr>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Clone current process</td>
</tr>
<tr>
<td>execv</td>
<td>CreateProcess</td>
<td>Replace current process</td>
</tr>
<tr>
<td>wait(pid)</td>
<td>WaitForSingleObject</td>
<td>Wait for a child to terminate.</td>
</tr>
<tr>
<td>exit</td>
<td>ExitProcess</td>
<td>Terminate process &amp; return status</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>File Management</strong></td>
</tr>
<tr>
<td>open</td>
<td>CreateFile</td>
<td>Open a file &amp; return descriptor</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close an open file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read from file to buffer</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write from buffer to file</td>
</tr>
<tr>
<td>lseek</td>
<td>SetFilePointer</td>
<td>Move file pointer</td>
</tr>
<tr>
<td>stat</td>
<td>GetFileAttributesEx</td>
<td>Get status info</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Directory and File System Management</strong></td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove <em>empty</em> directory</td>
</tr>
<tr>
<td>link</td>
<td>(none)</td>
<td>Create a directory entry</td>
</tr>
<tr>
<td>unlink</td>
<td>DeleteFile</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>mount</td>
<td>(none)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>umount</td>
<td>(none)</td>
<td>Unmount a file system</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Miscellaneous</strong></td>
</tr>
<tr>
<td>chdir</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>chmod</td>
<td>(none)</td>
<td>Change permissions on a file</td>
</tr>
<tr>
<td>kill</td>
<td>(none)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>time</td>
<td>GetLocalTime</td>
<td>Elapsed time since 1 jan 1970</td>
</tr>
</tbody>
</table>
OS Service Examples

• Services that need to be provided at kernel level
  – System calls: file open, close, read and write
  – Control the CPU so that users won’t stuck by running
    while ( 1 ) ;
  – Protection:
    • Keep user programs from crashing OS
    • Keep user programs from crashing each other

• Services that can be provided at user level
  – Read time of the day
Is Any OS Complete?
(Criteria to Evaluate OS)

Portability
Security
Fairness
Robustness
Efficiency
Interfaces
Operating Systems Structure (Chapter 1)

Monolithic systems - basic structure
1. A main program that invokes the requested service procedure.
2. A set of service procedures that carry out the system calls.
3. A set of utility procedures that help the service procedures.
Monolithic Systems

By far the most common OS organization
A simple structuring model for a monolithic system.
Layered Systems

- Layer-n services are comprised of services provided by Layer-(n-1).
- Structure of the THE operating system (Dijkstra 1968)

<table>
<thead>
<tr>
<th>Layer</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>The operator</td>
</tr>
<tr>
<td>4</td>
<td>User programs</td>
</tr>
<tr>
<td>3</td>
<td>Input/output management</td>
</tr>
<tr>
<td>2</td>
<td>Operator-process communication</td>
</tr>
<tr>
<td>1</td>
<td>Memory and drum management</td>
</tr>
<tr>
<td>0</td>
<td>Processor allocation and multiprogramming</td>
</tr>
</tbody>
</table>

- THE used this approach as a design aid
- Multics Operating System relied on Hardware Protection to enforce layering
Microkernels

- Microkernels move the layering boundaries between kernel and userspace
- Move only most rudimentary services to kernel
- Move other services to Userspace
- Higher Overhead, but more flexibility, higher robustness
  - Minix, L4, K42,
  - Minix (Tanenbaum is only 3200 lines of C and 800 lines assembler)
Client-Server Model

- Assumes generic network model (network, bus)
- Communication via message passing
Virtual Machines (1)

- VM/370: Timesharing system should be comprised of:
  - Multiprogramming
  - Extended machine with more interface than bare HW
  - Completely separate these two functions
- Provides ability to “self-virtualize”
- Beginning of “modern day” virtualization technology
Virtual Machines (2)

- A type 1 hypervisor (like virtual machine monitor)
  - Privileged instructions are trapped and “emulated”

- A type 2 hypervisor (runs on top of a host OS)
  - Unmodified (trapped)
  - Modified (paravirtualization)
Other areas of virtual machines usage

- Java virtual machines
- Dynamic scripting languages (e.g. Python)

- Typically define a instruction set that is “interpreted” by the associate virtual machine
  - JVM, PVM
  - Modern system then JIT (Just In Time) compile the VM instructions into native code.
History of the UNIX Operating System
(source: wikipedia)

Bell Labs: Thomson, Richie, Kernigan et.al
Lecture 2:
Processes and Threads - Part 1

Hubertus Franke
frankeh@cims.nyu.edu
OS Management of Application Execution

• Resources are made available to multiple applications
• The processor is switched among multiple applications so all will appear to be progressing
• The processor and I/O devices can be used efficiently
When the processor begins to execute the program code, we refer to this executing entity as a *process*.
What Is a Process?

An abstraction of a running program

Program Counter
The Process Model

• A process is an instance of an executing program and includes
  – Program counter
  – Registers
  – Variables
  – …

• A process has a program, input, output, and state.

If a program is running twice, does it count as two processes? or one?
Process: a running program

- A process includes
  - Address space
  - Process table entries (state, registers)
    - Open files, thread(s) state, resources held

- A process tree
  - A created two child processes, B and C
  - B created three child processes, D, E, and F
Process Element

• While the program is executing, this process can be uniquely characterized by a number of elements, including:

Where do we store all this info?
Process Control Block

- Contains the process elements
- It is possible to interrupt a running process and later resume execution as if the interruption had not occurred
- Created and managed by the operating system
- Key tool that allows support for multiple processes

Figure 3.1 Simplified Process Control Block
Address Space

- Defines where sections of data and code are located in 32 or 64 address space
- Defines protection of such sections
- ReadOnly, ReadWrite, Execute
- Confined “private” addressing concept
  - requires form of address virtualization
Multiprogramming

• One CPU and several processes
• CPU switches from process to process quickly
What Really Happens

What We Think It Happens
If we run the same program several times, will we get the same execution time?
Small program that switches the processor from one process to another (also called Scheduler)
Process Creation

• System initialization
  – At boot time
  – Foreground
  – Background (daemons)
• Execution of a process creation system call by a running process
• A user request
• A batch job
• Created by OS to provide a service
• Interactive logon
Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary)
- Killed by another process (involuntary)
## Process Termination: More Scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal completion</td>
<td>The process executes an OS service call to indicate that it has completed running.</td>
</tr>
<tr>
<td>Time limit exceeded</td>
<td>The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time (&quot;wall clock time&quot;), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.</td>
</tr>
<tr>
<td>Memory unavailable</td>
<td>The process requires more memory than the system can provide.</td>
</tr>
<tr>
<td>Bounds violation</td>
<td>The process tries to access a memory location that it is not allowed to access.</td>
</tr>
<tr>
<td>Protection error</td>
<td>The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.</td>
</tr>
<tr>
<td>Arithmetic error</td>
<td>The process tries to perform a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.</td>
</tr>
<tr>
<td>Time overrun</td>
<td>The process has waited longer than a specified maximum for a certain event to occur.</td>
</tr>
<tr>
<td>I/O failure</td>
<td>An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).</td>
</tr>
<tr>
<td>Invalid instruction</td>
<td>The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).</td>
</tr>
<tr>
<td>Privileged instruction</td>
<td>The process attempts to use an instruction reserved for the operating system.</td>
</tr>
<tr>
<td>Data misuse</td>
<td>A piece of data is of the wrong type or is not initialized.</td>
</tr>
<tr>
<td>Operator or OS intervention</td>
<td>For some reason, the operator or the operating system has terminated the process (e.g., if a deadlock exists).</td>
</tr>
<tr>
<td>Parent termination</td>
<td>When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.</td>
</tr>
<tr>
<td>Parent request</td>
<td>A parent process typically has the authority to terminate any of its offspring.</td>
</tr>
</tbody>
</table>
Process State

- Depending on the implementation, there can be several possible state models.

- The Simplest one: Two-state diagram
Process State: Three-State Model

1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available
Process State
Five-State Model

Figure 3.6 Five-State Process Model
Using Queues to Manage Processes

(a) Single blocked queue
Using Queues to Manage Processes

(b) Multiple blocked queues
One Extra State!

Swapped to disk
One Extra State!
Implementation of Processes

- OS maintains a **Control table (also called process table)**
- An array of structures
- One entry per process

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment info</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment info</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment info</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Conceptual view of the tables that OS maintains in order to manage execution of processes on resources.
A Bit About Interrupts

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.
Simple Modeling of Multiprogramming

- A process spends fraction $p$ waiting for I/O
- Assume $n$ processors in memory at once
- The probability that all processes are waiting for I/O at once is $p^n$
- So $\rightarrow$ CPU Utilization $= 1 - p^n$
Multiprogramming lets processes use the CPU when it would otherwise become idle.
Executing the OS Itself

(a) Separate kernel

(b) OS functions execute within user processes

(c) OS functions execute as separate processes
Threads

- Multiple threads of control within a process
- All threads of a process share the same address space
Why Threads?

• For some applications many activities can happen at once
  – With threads, programming becomes easier
  – Benefit applications with I/O and processing that can overlap

• Lighter weight than processes
  – Faster to create and restore
Example 1: A Word Processor
Example 2: Multithreaded Web Server
Processes vs Threads

- Process groups resources
- Threads are entities scheduled for execution on CPU
- No protections among threads (unlike processes) [Why?]
- Thread can be in any of several states: running, blocked, ready, and terminated
<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>
Each thread has its own stack (Why?).
Where to Put The Thread Package?

User space                       Kernel space
Implementing Threads in User Space

- Threads are implemented by a library
- Kernel knows nothing about threads
- Each process needs its own private thread table
- Thread table is managed by the runtime system
Implementing Threads in User Space

Advantages

• Very fast thread scheduling
• Each process can have its own thread scheduling algorithm
• Scale better

Disadvantages

• Blocking system calls can block the whole process
• Page fault blocks the whole process
• No other thread of the process will ever run unless the running thread voluntarily gives up the CPU
Implementing Threads in Kernel Space

- Kernel knows about and manages the threads
- No runtime is needed in each process
- Creating/destroying/(other thread related operations) a thread involves a system call
Implementing Threads in Kernel Space

Advantages

- When a thread blocks (due to page fault or blocking system calls) the OS can execute another thread from the same process

Disadvantages

- Cost of system call is very high
Hybrid Implementation

Multiple user threads on a kernel thread

Kernel

Kernel thread

User space

Kernel space
Conclusions

• Processes is the most central concept in OS
• Process vs Thread
• Multiprogramming vs multithreading