Secure Operating System Design and Implementation

Jon A. Solworth

Dept. of Computer Science
University of Illinois at Chicago

January 12, 2012

Part I

What is an Operating System?

What is an Operating System?

1. Traditionally, an OS is the first layer of software
2. (All other software on the computer depend on the OS)
3. It allows safe sharing of the computer
4. It provides services such as file systems, networking, memory management, process creation, authentication and authorization.

System software

1. The OS is part of the system software
2. The system software is layered for increasingly high level abstractions
3. Ending ultimately with applications.
4. The OS is difficult to build because it is constructed on bare metal, and hence does not have the software facilities available to applications.
5. In addition:
    1. OS is designed to run continuously (must not fail, must release systematically storage no longer is use)
    2. OS must deal with errors (hardware, out of memory)
    3. OS must deal with hardware design (devices, privileged instructions, etc.)

Layering

- Layering of software provides abstractions
- Each layer provides new abstractions to the level above it
- OS enables multi-tasking
  (multiple things to be done concurrently)
- The OS abstractions should be:
  - General purpose
  - Efficient
  - Secure
- Typical layering (from lowest to highest)
  - OS Kernel (executes privileged instructions)
  - OS processes (unprivileged instructions)
  - Application processes (unprivileged instructions)

  Note: unprivileged = user space
- OS Kernel is itself layered

Layering of user space

Everything in user space is a process. Processes are layered

Programming Language	the language in which a program is written
Libraries	optional, general purpose code, typically written in the same language as application code
Application program	code specific to solving a particular problem

Note that there are processes which perform OS functions (OS processes) which are structured in the same way.

Layering of the OS

- Can layer within the OS kernel (monolithic kernel) or
- Can layer outside the OS kernel with OS processes (microkernel)
- In either event, all privileged instructions are executed in the kernel
- Monolithic kernels treat the OS as one big program
- Monolithic kernels are more efficient
- Monolithic kernels have a single address space for OS code, a bug in one component can effect other components.
- Microkernels divide the OS into smaller programs
- Are less efficient, use multiple address spaces
- Popular OSs are monolithic

Part II

What is this course all about?

What is this course all about?

- Goal is to learn how to write an OS
- Of course, there isn't time to build a whole OS in this course
- We'll start from some assembly language and work our way up.
- Building the layers of abstraction in an OS
- This will be invaluable experience
- OS hackers need to be more precise than application hackers
- We'll talk about techniques which will improve your programming
- Thus increasing your skill even if you never again hack an OS

What background do you need?

- experience and knowledge in writing C programs
- understand computer architecture including assembly language programming
- understand deadlock, starvation, synchronization
- you'll be expected to use make, gdb, gcc, ld, ...
- I'll teach you the rest
- but since I don't know what you know
- so you have to ask questions

What background do you need?

- experience and knowledge in writing C programs
- understand computer architecture including assembly language programming
- understand deadlock, starvation, synchronization
- you'll be expected to use make, gdb, gcc, ld, ...
- I'll teach you the rest
- but since I don't know what you know
- so you have to ask questions
- Lots of questions

OS coding

- OS is a difficult environment
- Its large, modular, and highly interconnected
- Many low-level issues need to be contended with
  - Storage allocation/Typing
  - Concurrency issues
  - Failures
- Bugs typically crash the kernel
- OS kernel is trusted code, failures violate security

OS coding (cont'd)

- You need to know what you're doing
- You need to be conservative
- You need to realize that all your skill is not enough
- "A poor workman blames her tools"
- "A good workman has sharp tools"
- You need to separate learning about things
- ... from coding
- So that your coding is as clean as possible
- A program is to be read (and analyzed) by humans
- Execution is a minor issue

Systems programmers

- There are many things you need to get right at the same time
- You must try to learn as much as you can
- System programmers re-implement and hone OSs
- And think very carefully about small issues, in an attempt to reduce errors

Course project

- Normally OS development
  - Computer for OS development (hack and compile)
  - Computer for OS target (run the OS)
  - Serial cable between them
  - Run a debugger front end on development computer
  - Run a debugger back end on target computer
  - A bug usually crashes target computer
  - Problem: need to integrate debugger support into target OS, but initially there is no target OS
- Virtual Machine (VM) based OS development
  - target is an OS in a VM
  - development environment is your computer
  - in a crash can freeze the VM
  - some VMs, such as Xen, have their own debugger support
  - so you get it when you start
  - and your OS can't interfere with it much
  - ah, life is good

First assignment

- Install VMware on your computer (free!)
- Install Xen, Fedora in VMware (a VM on top of a VM)
- Fedora will provide your development environment
- And we'll build a guest OS on top of Xen (which is running on top of VMware) which is running on your host which can be Windows or OSX or Linux
- If you screw up anything, damage is limited to VM

What are my related research interests?

- I'm building a new security oriented OS called Ethos which is intended to make it far easier to write applications which withstand attack.
- How should an OS interface be designed to make systems more secure?
- What programming languages effect on construction of an OS?
- What kind of tools can be built to make OS more reliable?
- What should the system software look like on top of the OS kernel?

Part III

Operating System Overview

Overview

Definition

An Operating System (OS) provides a process abstraction.

Definition

A process is a program in execution; it is guaranteed to make finite progress.

- The term process derives from the term processor
- When executed on a uniprocessor, the processor is virtualized so that each process is allocated a virtual processor.
- Process is about isolation, processes have limited interaction with each other.

More definitions

Definition

A logical mechanism is virtualized if it appears to exist, but doesn't actually. (Virtualization occurs in software).

Definition

A mechanism is transparent if it exists, but does not appear to do so.

Definition

A mechanism is real if it exists and is visible.

Operating system structure

A process should be able to run with processor efficiency but must have limited powers. To do this:

- Unprivileged instructions for computations and
- Privilege instructions which are reserved for the OS to manage shared resources
- Process execute only unprivileged instructions
- The OS—in particular, the OS kernel—is the only entity which can execute privileged instruction
- The software thus consists of the OS kernel and processes.

Operating system kernel

- The OS kernel also executes unprivileged instructions
- Processes cannot execute privileged instructions, they rely instead on kernel-exported abstractions (the kernel interface).
- The kernel is a the center of the OS, which together with processes implements the OS abstractions.
- The kernel ensures that those OS abstractions are not bypassed
- The kernel traditionally provided the first level of software abstraction on computers
- And thus this abstraction level affects all software above it.
- In particular, this abstraction has a critical impact on security.

Resource visibility

- All of memory is visible to the OS
- But a process can only see the memory allocated to it
- Which prevent the process from seeing/modify other's data
- Hence, memory protection is part of the separation between processes and kernel
- And modifying memory protection is privileged
- In addition, I/O devices are typically visible in the memory address space
- The same mechanism which protects kernel (and other process's memory) also protects I/O devices

Architecture must enable the OS

- To managed how memory is allocated to processes
- To restrict the memory that a process can use
- To ensure finite progress for processes
- To provide a safe way of entering the kernel
- To provide safe sharing of I/O devices

Memory usage

The primary mechanisms partition memory, so that one or more partitions can be allocated to a process. The fundamental methods differed based on the partition characteristics:

fixed sized	thus implement paging
variable sized	thus implement segmentation

The creating of such partitions must be done by privileged instructions.

Finite progress

- Processes are running on the "bare metal"
- Which means that a process which runs in a computational loop, e.g.,

  ```
  while (1)
      ;
  ```

  will not ever voluntarily enter the OS
- Hence, a means to force an involuntary transfer to the OS is necessary to ensure other processes can run
- That mechanism is a timer interrupt
- The OS sets the timer interrupt before it begins executing the process
- When the timer goes off, the kernel is re-entered

Safe entering of the kernel

- Since a process cannot execute privileged instructions, it needs the OS to perform them.
- To request the OS to perform these operations, a system call is used.
- The system call simultaneously enters the kernel and sets the privilege bit.
- The entry points are defined by a vector, and indexed by system call number.
- This ensures that the entry points are well defined (and hence can be appropriately guarded)
- To return to the process, a return-from-interrupt instruction is executed

---

I/O devices

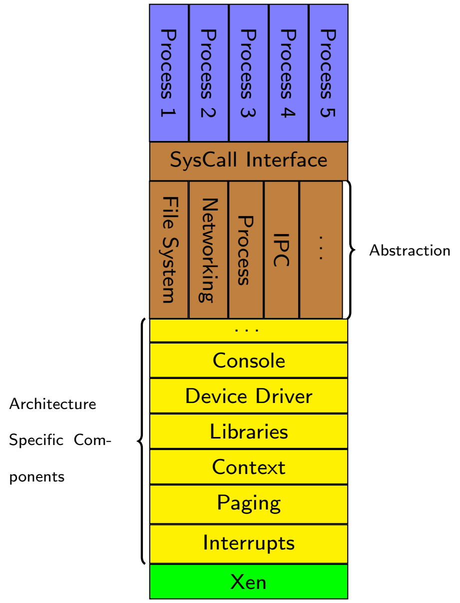
- I/O devices are typically accessed via device registers
- Device registers are mapped to memory addresses
- And hence protecting the memory address space of the I/O device register is sufficient to protect the I/O device.
- I/O devices have long latency associated with them
- So interrupts are used to notify the OS when the device finishes an operation.

Summarizing isolation

Isolation is provided by managing resources

- Memory
- I/O devices
- CPU (for finite progress)
- trap instructions (for safe transitioning from unprivileged to privileged)

OS Layers



OS facilities

In addition to isolation, OS provides abstractions

Process	creation, scheduling, removal
Memory	allocation, de-allocation
Filesystem	read, write, organize, recovery on failure.
Networking	support of Internet Protocol
IPC	interprocess communication
Device I/O	e.g., keyboard, mouse, and display
Authentication	identification of external entities
Authorization	(a.k.a. access controls)

Many abstractions are for controlled sharing.

OS interface

OS interface is critical. It should be:

Flexible	enabling needed semantics to be implemented
Efficient	so that resources are not wasted
Coherent	so that different parts work together well
Long lived	processes depend on the OS, and hence changes to the underlying OS interface can break applications
Minimize error	so that programs are easy to get right
Abstract	so that its easy to program with

Conclusion

- An OS implements a process abstraction
- Processes are isolated from each other
- Processes communicate with the outside world through system calls to the OS kernel
- The OS kernel operates in privileged mode, only software which can execute privileged instructions
- The OS kernel exports a number of abstractions to processes
- These abstractions have a large effect on applications