02. Protection

9th ed: Ch. 2.7+, 14, 15, 16
10th ed: Ch. 2.7+, 16, 17, 19
Objectives

• To describe the evolution of the operating system
• To understand how the OS protects itself from user programs
• To understand how the OS protects user programs from each other
• To know some different ways the OS can be structured
• To be aware of some security considerations
Outline

• OS evolution
• Kernels
• Security
Outline

• OS evolution
  • Single-tasking
  • Dual-mode operation

• Kernels

• Security
Operating system evolution

• **Open shop**: One machine, one CPU, one user, one program – the user is the programmer is the operator, all programming is in machine code
  - E.g., EDSAC, 1947—1955

• **Batch systems**: tape drives collate and run a set of programs in a batch, increasing efficiency
  - Spooling allowed overlap of I/O with computation

• **Multiprogramming**: one machine, one CPU, one running program but many loaded programs
  - **Job scheduling**: select jobs to load and then which resident job to run

• **Timesharing**: switching jobs so frequently that users have the illusion many jobs are running simultaneously
  - **CPU scheduling**: select which job to run from many that are ready
  - Enables interactive computing
Single-tasking OS: MS-DOS

• Command interpreter receives input from user
  • Program is loaded, overwriting much of the command interpreter
  • Instruction pointer set to the start of program
• Once finished, termination causes command interpreter stub to reload command interpreter
  • Exit error code available to user
Dual-mode operation

• Allows OS to stop malicious or buggy code from doing bad things
• Use hardware – a **mode bit** – to distinguish (at least) two modes of operation
  • **User mode** when executing on behalf of a user (i.e. application programs)
  • **Kernel mode** when executing on behalf of the OS
  • Some instructions designated as privileged, only executable in kernel mode

• Increasingly CPUs support multi-mode operations
  • i.e. virtual machine manager (VMM) mode for guest VMs

• Often “nested” e.g., x86 rings 0—3; further inside can do strictly more
  • Not ideal, but disjoint/overlapping permissions is complex
Outline

• OS evolution

• Kernels
  • System calls
  • Microkernels
  • Virtualisation

• Security
Kernels

• Protection prevents applications doing I/O – kernel does it for them
  • Thus we need an unprivileged instruction to transition from user to kernel mode
  • Generally called a **trap** or a **software interrupt** since operates similarly to (hardware) interrupt

• OS services are accessible via **system calls**
  • Invoked by a trap with OS having vectors to handle
  • Vector enforces code run when mode switch occurs
  • Prevents application from switching to kernel mode and then just doing whatever it likes

• Alternative is for OS to emulate for application, and check every instruction before execution as used in some virtualisation systems, e.g., QEMU
System calls

- Provide a (language agnostic) standard interface to the OS services
- Accessed via a high-level (language specific) Application Programming Interface (API) rather than called directly
  - E.g., glibc

```
#[inline(always)]
pub unsafe fn syscall4(mut n: usize,
    a1: usize,
    a2: usize,
    a3: usize,
    a4: usize)
-> usize
{
    llvm_asm!("int $$0x80"
        : "+(eax)"(n)
        : "+(ebx)"(a1) "+(ecx)"(a2) "+(edx)"(a3) "+(esi)"(a4)
        : "memory" "cc"
        : "volatile");
    n
}
```

Raw system calls in Rust
https://github.com/strake/system-call.rs/
System call invocation

• Typically each system call is associated by a number that indexes a **system call table**
  • Invoked by putting the relevant number and any required parameters in the right places and trapping
  • Return status and any values made available to application in user space

• Usually managed by run-time support library, a set of functions built into libraries automatically linked by your compiler
System call parameters

• Three main ways to pass parameters:
  1. Load into registers
  2. Place onto stack for the kernel to pop off
  3. Place into a block of memory and put the block’s address into a register

• One of the latter two usually preferred
  • Registers limited in number and size

```c
int open(const char *path, int oflag, ...);

ssize_t read(int fildes, void *buf, size_t nbyte);
```
Microkernels

- OS interfaces must be extremely stable
  - Makes them difficult to extend with new calls
  - Even more difficult to remove calls
- Alternative is **microkernels**
  - Move OS services into local, sometimes privileged, servers
  - Increases modularity and extensibility
- **Message passing** used to access servers
  - Replaces trapping so must be extremely efficient
- Many common OSs blur the distinction between kernel and microkernel, e.g.,
  - Linux has kernel modules and some servers
  - Windows NT 3.5 originally a microkernel but performance concerns caused NT 4.0 to move some services back into the kernel
Virtualisation

- More recently, trend towards encapsulating applications differently
  - Make the system appear to be supporting just one application
  - Particularly relevant when building systems using microservices
  - Protection, or isolation at a different level
- **Virtualisation**: allows operating systems to be run alongside each other above a **hypervisor**
  - Type 1 runs directly on the host hardware, possibly using hardware extensions (VT-x)
  - Type 2 runs above a full OS kernel
  - Can support cross-architecture using **emulation** (slow) or **interpretation** (if not natively compiled)
Virtual machines vs Containers

• Virtual Machines encapsulate an entire running system, including the OS, and then boot the VM over a hypervisor
  • E.g., Xen, VMWareESX, Hyper-V

• Containers expose functionality in the OS so that each container acts as a separate entity even though they all share the same underlying OS functionality
  • E.g., Linux Containers, FreeBSD Jails, Solaris Zones

• Use cases include
  • Laptops and desktops running multiple OSes for exploration or compatibility
  • Developing apps for multiple OSes without having multiple systems
  • QA testing applications without having multiple systems
  • Executing and managing compute environments within datacenters
Outline

• OS evolution
• Kernels

• Security
  • Principle of least privilege
  • Domain of protection
  • Access matrix
  • Access Control Lists (ACLs)
  • Capabilities
  • Authentication
Security

• Defence of the system against internal and external attacks
  • Huge range of attacks, including denial-of-service, worms, viruses, identity theft, theft of service

• Systems generally first distinguish among users, to determine who can do what
  • User identities (user IDs, security IDs) include name and associated number, one per user
  • User ID then associated with all files, processes of that user to determine access control
  • Group identifier (group ID) allows set of users to be defined and controls managed, then also associated with each process, file

• Privilege escalation allows user to change to effective ID with more rights
Principle of least privilege

• Objects should be given just enough privileges to perform their tasks
  • **Hardware objects** (e.g., devices) and **software objects** (e.g., files, programs, semaphores)

• Properly set permissions can limit damage if object has a bug and gets abused
  • Can be **static** (during life of system, during life of process)
  • Or **dynamic** (changed by process as needed) by domain switching, privilege escalation

• Compartmentalization a derivative concept regarding access to data
  • Process of protecting each individual system component through the use of specific permissions and access restrictions
  • More granular, more complex, more protective

• **Covert channels** leak information using side-effects
  • Hardware include wire tapping or receiving electromagnetic radiation from devices
  • Software include page fault statistics or input-dependent timing
  • E.g., lowest layer of recent OCaml TLS library had to be written in C to avoid the garbage collector becoming a covert channel
Domain of protection

- Domain limits access to (and operations on) objects
  - \textit{access-right} = \textit{< object-name, rights-set >} where \textit{rights-set} is a subset of all valid operations that can be performed on \textit{object-name}
  - A domain is then a set of \textit{access-rights}
  - In UNIX a domain is a user id

\[
\begin{align*}
D_1 & : <O_3, \{\text{read, write}\}> \\
D_2 & : <O_2, \{\text{write}\}> \\
D_3 & : <O_4, \{\text{print}\}> \\
\end{align*}
\]
Access matrix

• A matrix of **domains** (**subjects, principals**) against **objects**
  • Rows represent domains, columns represent objects
  • Operations a process in domain can invoke on object
  • Operations can include adding/deleting entries in matrix

• Example of separation of policy from mechanism

<table>
<thead>
<tr>
<th>domain</th>
<th>object</th>
<th>$F_1$</th>
<th>$F_2$</th>
<th>$F_3$</th>
<th>laser printer</th>
<th>$D_1$</th>
<th>$D_2$</th>
<th>$D_3$</th>
<th>$D_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_1$</td>
<td>read</td>
<td>read</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_2$</td>
<td></td>
<td></td>
<td>print</td>
<td></td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
<td>switch control</td>
</tr>
<tr>
<td>$D_3$</td>
<td></td>
<td></td>
<td></td>
<td>execute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$D_4$</td>
<td>write</td>
<td>write</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>switch</td>
<td></td>
</tr>
</tbody>
</table>
Implementing the access matrix

• The access matrix is a table of triples < domain, object, rights-set >
  • For a domain to invoke an operation on an object involves searching to see if
    that operation is in any rights-set for the pair < domain, object >

• Table is large so may not fit in memory – but sparse

• Two common representations
  1. By object, storing list of domains and rights with each object – Access
     Control List (ACL)
  2. By domain, storing list of objects and rights with each domain – Capabilities
Access Control Lists (ACLs)

• Each column is an access list for one object
  • Results in a per-object ordered list of $<\text{domain, rights-set}>$

• Often used in storage systems
  • System naming scheme provides for ACL to be inserted in naming path, e.g., files

• If ACLs stored on disk, check is in software so use only on low duty cycle – for higher duty cycle must cache results of check
  • E.g., ACL checked when file opened for read or write, or when code file is to be executed

• In (e.g.) UNIX access control is by program allowing arbitrary policies
Capabilities

• Each row is a capability for one domain, indicating the permitted operations on a set of objects
• To execute operation $M$ on object $O$, process requests operation and passes the capability as parameter
  • Possession of capability means operation is allowed
  • Capability is a protected object, maintained by the OS and unmodifiable by the application – like a “secure pointer”
• Hardware capabilities, e.g., CHERI
  • Have special machine instructions to modify (restrict) capabilities
  • Support passing of capabilities on procedure (program) call
• Software capabilities are protected by encryption
  • Nice for distributed systems
Authentication

• User to system: required as protection systems depend on user ID
  • Typically established through use of *password* (or passphrase or key)
  • Need to be managed, kept secure
  • Hashed with a salt (easy to compute, hard to invert)
  • Multi-factor authentication adds a second (or more) component
  • Failed access attempts usually logged

• System to user: avoid user talking to the wrong computer / program
  • In the old days with directly wired terminals, make login character same as terminal attention, or always do a terminal attention before trying login
  • E.g., Windows NT’s Ctrl-Alt-Del to login — no-one else can trap it
  • (When your bank phones, how do you know it’s them?)
Summary

- OS evolution
  - Single-tasking
  - Dual-mode operation
- Kernels
  - System calls
  - Microkernels
  - Virtualisation

- Security
  - Principle of least privilege
  - Domain of protection
  - Access matrix
  - Access Control Lists (ACLs)
  - Capabilities
  - Authentication