Operating Systems
(1DT020 & 1TT802)

Lecture 13

I/O Systems (cont’d)
Protection and Security

May 23, 2008

Léon Mugwaneza

http://www.it.uu.se/edu/course/homepage/os/vt08
Goals for Today

• I/O systems
• Protection & Security

Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne, others from Kubiatowicz - CS162 ©UCB Fall 2007 (University of California at Berkeley)
I/O Systems

• Thousands of devices, each slightly different
  – OS should offer standard interfaces to applications

• Some operational parameters:
  – Byte/Block
    » Some devices provide single byte at a time (e.g. keyboard)
    » Others provide whole blocks (e.g. disks, networks, etc)
  – Also : Sequential/Random
    » Some devices must be accessed sequentially (e.g. tape)
    » Others can be accessed randomly (e.g. disk, cd, etc.)

• Device Rates vary over many orders of magnitude
  – OS should be able to handle this wide range
  – Better not have high overhead/byte for fast devices!
  – Better not waste time waiting for slow devices
How Does User Deal with Timing?

• **Blocking Interface:** “Wait”
  – When request data (e.g. `read()` system call), put process to sleep until data is ready
  – When write data (e.g. `write()` system call), put process to sleep until device is ready for data

• **Non-blocking Interface:** “Don’t Wait”
  – Returns quickly from read or write request with count of bytes successfully transferred
  – Read may return nothing, write may write nothing

• **Asynchronous Interface:** “Tell Me Later”
  – When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
  – When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user
Main components of Intel Chipset: Pentium 4

• **Northbridge:**
  - Handles memory
  - Graphics

• **Southbridge: I/O**
  - PCI bus
  - Disk controllers
  - USB controllers
  - Audio
  - Serial I/O
  - Interrupt controller
  - Timers
How does the processor actually talk to the device?

- CPU interacts with a **Controller**
  - Contains a set of *registers* that can be read and written
  - May contain memory for request queues or bit-mapped images

- Regardless of the complexity of the connections and buses, processor accesses registers in two ways:
  - **I/O instructions**: in/out instructions
    » Example from the Intel architecture: `out 0x21, AL`
  - **Memory mapped I/O**: load/store instructions
    » Registers/memory appear in physical address space
    » I/O accomplished with load and store instructions
Example: Memory-Mapped Display Controller

- **Memory-Mapped:**
  - Hardware maps control registers and display memory into physical address space
    - Addresses set by hardware jumpers or programming at boot time
  - Simply writing to display memory (also called the “frame buffer”) changes image on screen
    - Addr: 0x8000F000—0x8000FFFF
  - Writing graphics description to command-queue area
    - Say enter a set of triangles that describe some scene
      - Addr: 0x80010000—0x8001FFFF
  - Writing to the command register may cause on-board graphics hardware to do something
    - Say render the above scene
      - Addr: 0x0007F004

- **Can protect with page tables**
Transfering Data To/From Controller

• **Programmed I/O:**
  – Each byte transferred via processor in/out or load/store
  – Pro: Simple hardware, easy to program
  – Con: Consumes processor cycles proportional to data size

• **Direct Memory Access:**
  – Give controller access to memory bus
  – Ask it to transfer data to/from memory directly

• **Sample interaction with DMA controller (from book):**

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion

DMA/bus/ interrupt controller

CPU

cache

CPU memory bus

memory

buffer

PCI bus

IDE disk controller

disk

disk

disk

disk

5/23/08
# A Kernel I/O Structure

![Diagram of a kernel I/O structure](image)

**Diagram Description:**
- **Software:**
  - kernel
  - kernel I/O subsystem
  - SCSI device driver
  - keyboard device driver
  - mouse device driver
  - ... (additional device drivers)
  - PCI bus device driver
  - floppy device driver
  - ATAPI device driver

**Hardware:**
- SCSI device controller
- keyboard device controller
- mouse device controller
- ... (additional device controllers)
- PCI bus device controller
- floppy device controller
- ATAPI device controller

**Devices:**
- SCSI devices
- keyboard
- mouse
- ... (additional devices)
- PCI bus
- floppy-disk drives
- ATAPI devices (disks, tapes, drives)
Device Drivers

- **Device Driver**: Device-specific code in the kernel that interacts directly with the device hardware
  - Supports a standard, internal interface
  - Same kernel I/O system can interact easily with different device drivers
  - Special device-specific configuration supported with the `ioctl()` system call

- **Device Drivers typically divided into two pieces**:
  - Top half: accessed in call path from system calls
    » Implements a set of **standard, cross-device calls** like `open()`, `close()`, `read()`, `write()`, `ioctl()`, `strategy()`
    » This is the kernel’s interface to the device driver
    » Top half will **start** I/O to device, may put thread to sleep until finished
  - Bottom half: run as interrupt routine
    » Gets input or transfers next block of output
    » May wake sleeping threads if I/O now complete
I/O Device Notifying the OS

• The OS needs to know when:
  – The I/O device has completed an operation
  – The I/O operation has encountered an error

• I/O Interrupt:
  – Device generates an interrupt whenever it needs service
  – Handled in bottom half of device driver
    » Often run on special kernel-level stack
  – Pro: handles unpredictable events well
  – Con: interrupts relatively high overhead

• Polling:
  – OS periodically checks a device-specific status register
    » I/O device puts completion information in status register
    » Could use timer to invoke lower half of drivers occasionally
  – Pro: low overhead
  – Con: may waste many cycles on polling if infrequent or unpredictable I/O operations

• Actual devices combine both polling and interrupts
  – For instance: High-bandwidth network device:
    » Interrupt for first incoming packet
    » Poll for following packets until hardware empty
Protection vs Security

- **Protection:** one or more mechanisms for controlling the access of programs, processes, or users to resources
  - Page Table Mechanism
  - File Access Mechanism

- **Security:** use of protection mechanisms to prevent misuse of resources
  - Misuse defined with respect to policy
    - E.g.: prevent exposure of certain sensitive information
    - E.g.: prevent unauthorized modification/deletion of data
  - Requires consideration of the external environment within which the system operates
    - Most well-constructed system cannot protect information if user accidentally reveals password

[A short to introduction protection and security]
Protection : Dual-Mode Operation

- Multiprogramming goals
  - Isolate processes and kernel from one another
  - Allow flexible translation that allows easy sharing between processes
  - User cannot change mode to kernel mode or modify page table mapping
  - Limited access to memory: cannot adversely effect other processes
    » Side-effect: Limited access to memory-mapped I/O operations
  - Limited access to interrupt controller
  - What else needs to be protected?

- To Assist with Protection, Hardware provides at least two modes: “Kernel” mode (or “protected”) and “User” mode
  - Mode set with bits in control register only accessible in kernel-mode
  - Some instructions only available in kernel mode (Privileged instructions)

- Intel processor actually has four “rings” of protection:
  - PL (Privileged Level) from 0 – 3 (PL0 has full access, PL3 has least)
  - Privilege Level set in code segment descriptor (CS)
  - Mirrored “IOPL” bits in condition register gives permission to programs to use the I/O instructions
  - Typical OS kernels on Intel only use PL0 (“kernel”) and PL3 (“user”)

- A couple of issues:
  - How to share CPU between kernel and user programs?
  - How do programs interact?
  - How does one switch between kernel and user modes?
How to get from Kernel→User

• What does the kernel do to create a new user process?
  – Allocate and initialize address-space control block
  – Read program off disk and store in memory
  – Allocate and initialize translation table
    » Point at code in memory so program can execute
    » Possibly point at statically initialized data
  – Run Program:
    » Set machine registers
    » Set hardware pointer to translation table
    » Set processor status word for user mode
    » Jump to start of program

• How does kernel switch between processes?
  – Same saving/restoring of registers as before
  – Save/restore hardware pointer to translation table
User→Kernel (System Call)

- How does the user program get back into kernel?

**System call**: Voluntary procedure call into kernel

- Hardware for controlled User→Kernel transition
- Can any kernel routine be called?
  - No! Only specific ones.
- System call ID encoded into system call instruction
  - Index forces well-defined interface with kernel
System Call Continued

• What are some system calls?
  – I/O: open, close, read, write, lseek
  – Files: delete, mkdir, rmdir, truncate, chown, chgrp, ...
  – Process: fork, exit, wait
  – Network: socket create, set options
  – Operations on shared memory segments, semaphores, other IPC

• Are system calls constant across operating systems?
  – Not entirely, but there are lots of commonalities
  – Also some standardization attempts (POSIX)

• What happens at beginning of system call?
  » On entry to kernel, sets system to kernel mode
  » Handler address fetched from table/Handler started

• System Call argument passing:
  – In registers (not very much can be passed)
  – Write into user memory, kernel copies into kernel memory
    » User addresses must be translated!
    » Kernel has different view of memory than user
  – Every argument must be explicitly checked!
User→Kernel (Exceptions: Traps and Interrupts)

- A system call instruction causes a synchronous exception (or “trap”)
  - In fact, often called a software “trap” instruction
- Other sources of **Synchronous Exceptions**:
  - Divide by zero, Illegal instruction, undefined instruction, Bus error (bad address, e.g. unaligned access)
  - Segmentation Fault (address out of range)
  - Page Fault (for illusion of infinite-sized memory)
- **Interrupts are Asynchronous Exceptions**
  - Examples: timer, disk ready, network, etc....
  - Interrupts can be disabled, traps cannot!
- **On system call, exception, or interrupt:**
  - Hardware enters kernel mode with interrupts disabled
  - Saves PC, then jumps to appropriate handler in kernel
  - For some processors (x86), processor also saves registers, changes stack, etc.
- **Actual handler typically saves registers, other CPU state, and switches to kernel stack**
Communication

• Now that we have isolated processes, how can they communicate?
  – Shared memory: common mapping to physical page
    » As long as they place objects in shared memory address range, threads from each process can communicate
    » Note that processes A and B can talk to shared memory through different addresses
    » In some sense, this violates the whole notion of protection that we have been developing
  – If address spaces don’t share memory, all inter-address space communication must go through kernel (via system calls)
    » Byte stream producer/consumer (put/get): Example, communicate through pipes connecting stdin/stdout
    » Message passing (send/receive): another kind of process communication and synchronization tool
      • Blocking vs non blocking send
      • Blocking vs non blocking receive
      • Naming: the other process, a mail box (which can be shared), a private channel
    » File System (read/write): File system is shared state!
Security: Preventing Misuse

• Types of Misuse:
  – Accidental:
    » If I delete shell, can’t log in to fix it!
    » Could make it more difficult by asking: “do you really want to delete the shell?”
  – Intentional:
    » Doesn’t help to ask if user wants perform action

• Three Pieces to Security
  – Authentication: who the user actually is
  – Authorization: who is allowed to do what
  – Enforcement: make sure people do only what they are supposed to do

• Loopholes in any carefully constructed system:
  – Log in as super-user and you’ve circumvented authentication
  – Log in as self and can do anything with your resources; for instance: run program that erases all of your files
  – Can you trust software to correctly enforce Authentication and Authorization?
Authentication: Identifying Users

• How to identify users to the system?
  – Passwords
    » Shared secret between two parties
    » Since only user knows password, someone types correct password ⇒ must be user typing it
    » Very common technique
      • Encrypt passwords to help hid them
      • Force them to be longer/not amenable to dictionary attack
      • Use one-time passwords
  – Smart Cards
    » Electronics embedded in card capable of providing long passwords or satisfying challenge → response queries
    » May have display to allow reading of password
    » Or can be plugged in directly; several credit cards now in this category
  – Biometrics
    » Use of one or more intrinsic physical or behavioral traits to identify someone
    » Examples: fingerprint reader, palm reader, retinal scan
    » Becoming quite a bit more common
Authorization: Who Can Do What?

• How do we decide who is authorized to do actions in the system?

• Access Control Matrix: contains all permissions in the system
  - Resources across top
    » Files, Devices, etc…
  - Domains in columns
    » A domain might be a user or a group of permissions
    » E.g. opposite: User D3 can read F2 or execute F3
  - In practice, table would be huge and sparse!

<table>
<thead>
<tr>
<th>domain</th>
<th>object</th>
<th>F_1</th>
<th>F_2</th>
<th>F_3</th>
<th>printer</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_1</td>
<td></td>
<td>read</td>
<td></td>
<td></td>
<td>read</td>
</tr>
<tr>
<td>D_2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>print</td>
</tr>
<tr>
<td>D_3</td>
<td></td>
<td></td>
<td>read</td>
<td>execute</td>
<td></td>
</tr>
<tr>
<td>D_4</td>
<td></td>
<td>read</td>
<td>write</td>
<td></td>
<td>read write</td>
</tr>
</tbody>
</table>

• Important issues:
  - When are access rights checked?
  - How (and when) to revoke authorization?
    » List of revocation attached to objects or processes?
    » Expiration dates? Epoch numbers?
Authorization: Implementation Choices

- **Access Control Lists:** store permissions with object
  - Still might be lots of users!
  - UNIX limits each file to: r,w,x for owner, group, world
  - More recent systems allow definition of groups of users and permissions for each group
  - ACLs allow easy changing of an object’s permissions
    » Example: add Users C, D, and F with rw permissions

- **Capability List:** each process tracks which objects has permission to touch
  - Popular in the past, idea out of favor today
  - Consider page table: Each process has list of pages it has access to, not each page has list of processes …
  - Capability lists allow easy changing of a domain’s permissions
    » Example: you are promoted to system administrator and should be given access to all system files

- **A combination approach:** Users have capabilities (groups or roles), Objects have ACLs
  - ACLs refer to users or groups
  - Change object permission by modifying ACL
  - Change broader user permission via change in group membership
Authorization Continued

• **Principle of least privilege:** programs, users, and systems should get only enough privileges to perform their tasks
  
  – Very hard to do in practice
    » How do you figure out what the minimum set of privileges is needed to run your programs?
  
  – People often run at higher privilege than necessary
    » Such as the “administrator” privilege under Windows

• **One solution: Signed Software**
  
  – Only use software from sources that you trust, thereby dealing with the problem by means of authentication
  
  – Fine for big, established firms such as Microsoft, since they can make their signing keys well known and people trust them
    » Actually, not always fine: recently, one of Microsoft’s signing keys was compromised, leading to malicious software that looked valid
  
  – What about new startups?
    » Who “validates” them?
    » How easy is it to fool them?
Summary (I/O systems)

• I/O Devices Types:
  – Many different speeds (0.1 bytes/sec to GBytes/sec)
  – Different Access Patterns: block, char, net devices
  – Different Access Timing: Non-/Blocking, Asynchronous

• I/O Controllers: Hardware that controls actual device
  – CPU accesses through I/O insts, ld/st to special phy memory
  – Report results through interrupts or a status register polling

• Device Driver: Device-specific code in kernel
Summary (Protection & Security)

• Protection: Prevent unauthorized Sharing of resources
  – Address space protected using translation of addresses through Memory Management Unit (MMU)
    » Every Access translated through page table
    » Changing of page tables only available to kernel
  – Dual-Mode
    » Kernel/User distinction: User restricted
    » User→Kernel: System calls, Traps, or Interrupts
    » Inter-process communication: shared memory, or through kernel (system calls)

• Security: prevent misuse
  – User Identification
    » Passwords/Smart Cards/Biometrics
    » Encrypt password to help hide them
    » Force passwords to be longer/not amenable to dictionary attack
  – Authorization
    » Access Matrix
      • Access lists
      • Capabilities