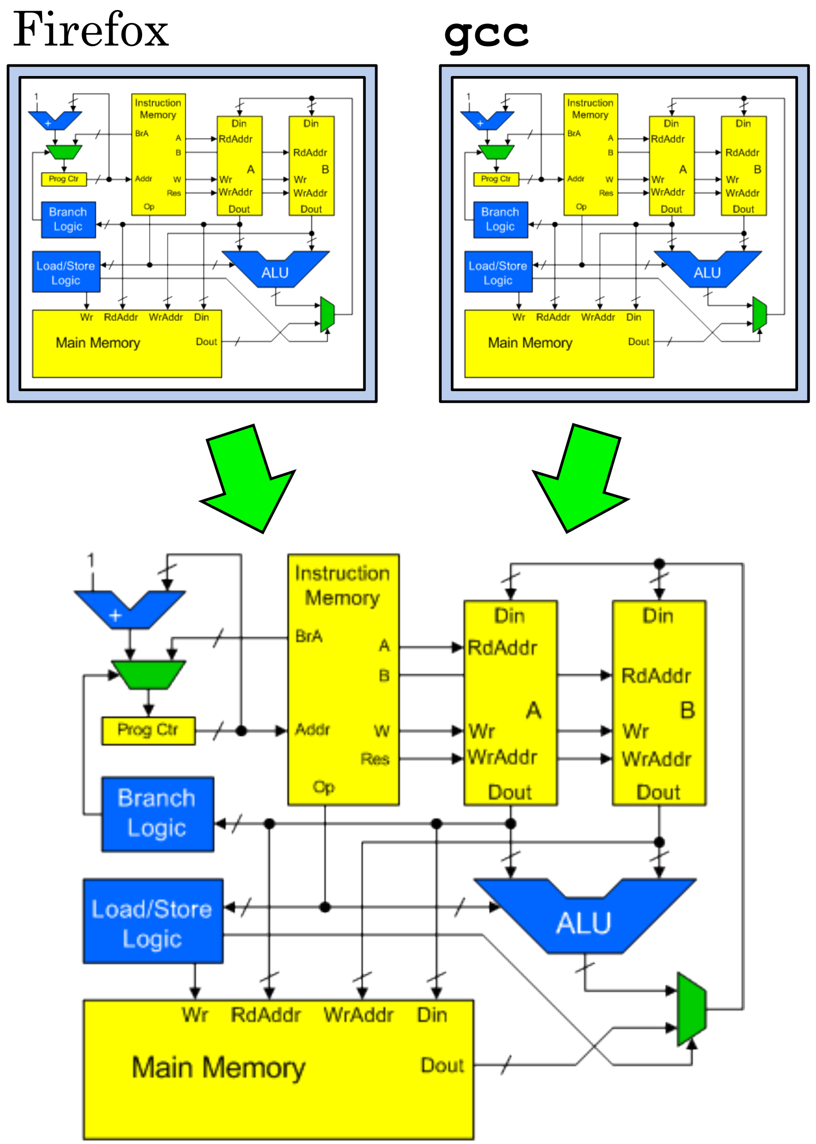
CS24: Introduction to Computing Systems

Spring 2015

Lecture 17

LAST TIME

- Introduced virtualization
  - Present an *abstraction* of the processor and memory to programs
  - Each program runs as if it has sole access to the computer hardware
- A running program plus its context is called a process
- A process' context includes:
  - All register state, including the stack pointer, program counter, and the flags register
  - Contents of all memory that the program is using
    - Program code, stack, heap, etc.

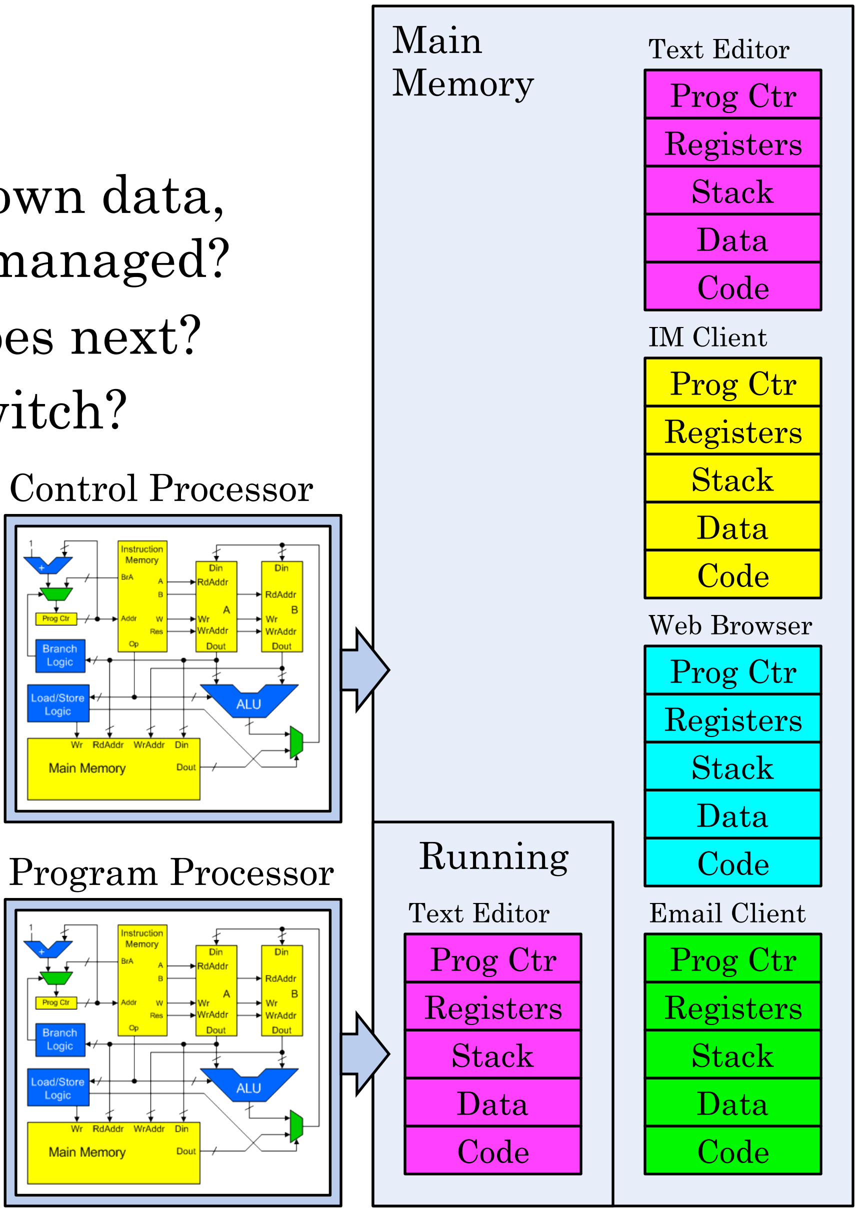


"But what about…"

- We still have some big issues to solve!
- Who manages all the processes?
  - How do we ensure that processes can't see each other, but that the manager can see everything?
- How do we interrupt a running program, in order to perform a context switch?
  - How do we choose which process should run next?
- What if a program crashes?
  - Must not bring down the whole system!
  - How do we find out that the program died, and what do we do in this case?

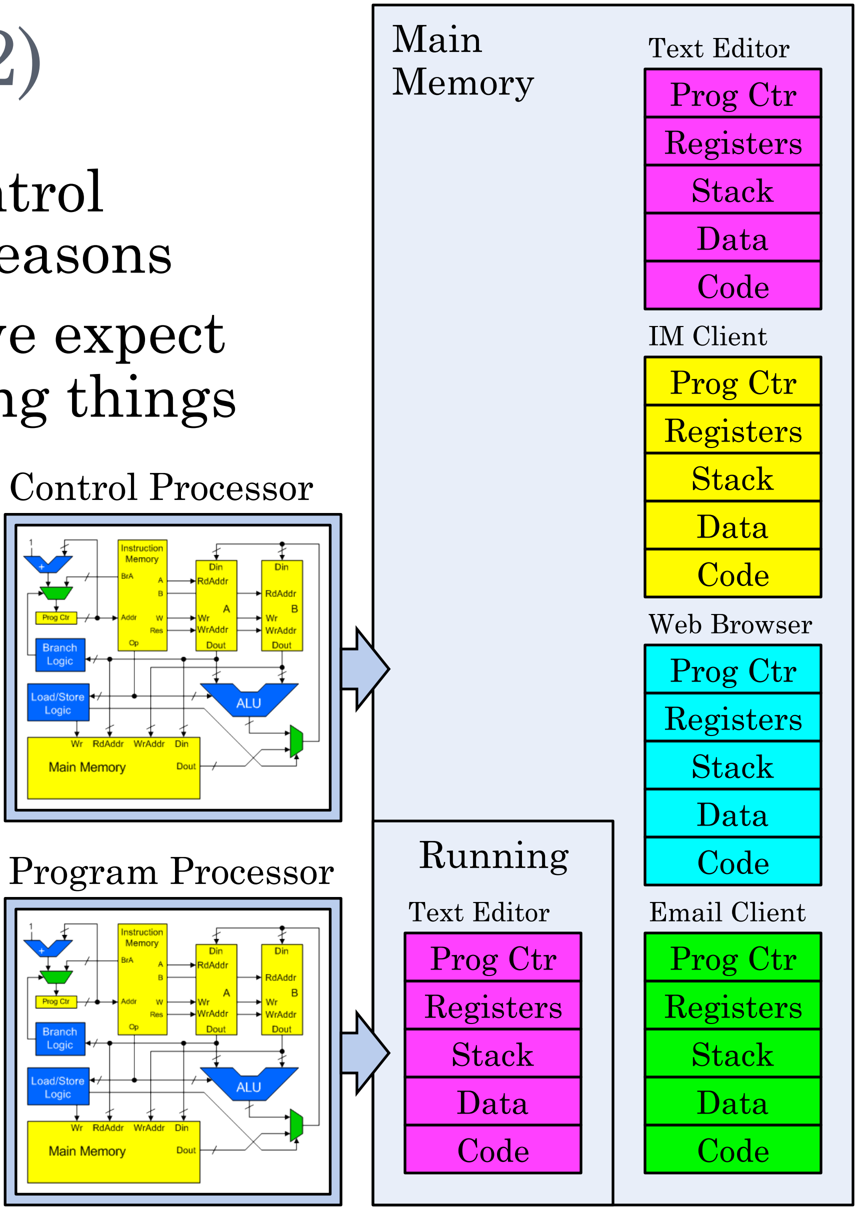
Process Manager?

- If each process can only see its own data, how are the processes actually managed?
  - Who decides what process goes next?
  - Who performs the context switch?
- Could introduce a separate "control processor" that manages the processes…
  - Can access all of main memory, including the internal state of processes
- Good idea?



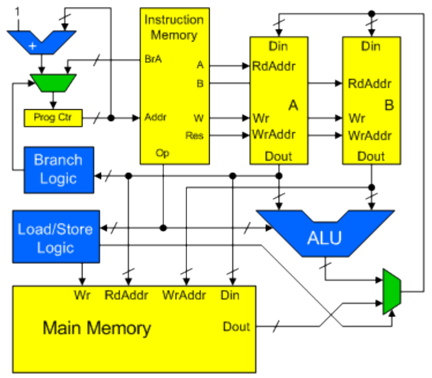
PROCESS MANAGER? (2)

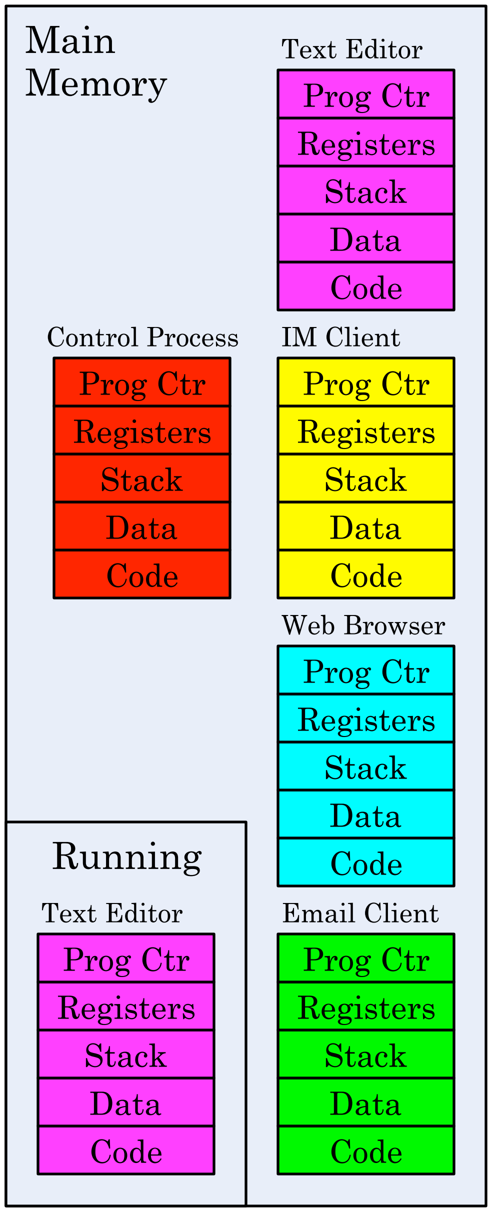
- Implementing a separate control processor is bad for several reasons
- Separate processor implies we expect to spend lots of time managing things
  - *Ideally*, most clock cycles are spent running our actual programs!
  - Really not enough work to justify a separate control processor
- Also, <u>severely</u> curtails ability to upgrade/debug control processor's services
  - Would have to fabricate a new processor!



Process Manager (2)

- Instead, we can virtualize the control processor as well
- Now, have another interesting problem!
- The control process is special:
  - Needs to see all data for all programs
  - Must be able to perform special context-switch operations
- Application processes:
  - Definitely shouldn't be able to do these things!
  - Should still have a limited view of the world





Operating Modes

- The processor can provide multiple <u>operating modes</u>
  - Physically enforce differences between different processes
- Kernel mode:
  - Program can do everything the processor supports
  - Access all of memory, use special instructions, etc.
  - Also known as "protected mode" or "privileged mode"
- User mode:
  - Program has a restricted view of the world
  - Can only access its own memory
  - Some instructions are disallowed
    - e.g. ones that set the processor mode
  - Also called "normal mode"
- Control process provides essential, trusted features
  - Run control process in kernel mode
  - Application processes are always run in user mode

Computer Operating System

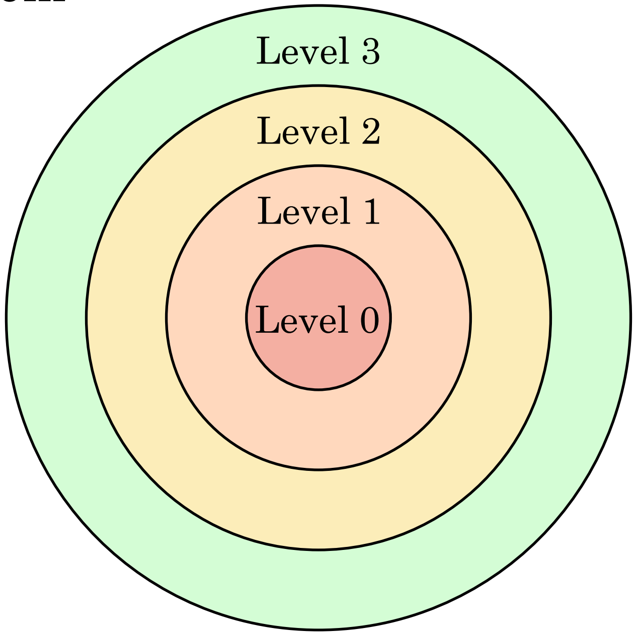
- The control process provides services to all other processes…
- This program becomes the core or *kernel* of an *operating system* for our computer
- The operating system has several purposes:
  - Manage computer hardware on behalf of programs
  - Provide an abstraction of the processor to support concurrently executing processes
    - Isolate concurrent processes from each other
    - Terminate and clean up after programs that exit or crash
  - Provide other common facilities that programs need
    - e.g. memory management, file IO, networking, etc.
    - Provide a unified API for working with these facilities

Computer Operating System (2)

- Operating system extends our abstraction hierarchy
- Computer hardware (lowest level):
  - Provides basic facilities for executing programs
  - Processor, main memory (plus caches!), IO devices, etc.
- Operating system:
  - Mediates use of hardware among various programs
  - Provides simple, efficient APIs for sophisticated features that most programs will need
- Application programs:
  - Solve specific problems that users need to solve
  - Compilers, databases, email clients, web browsers, etc.
- Users: people, other computers, etc.
- Each level of abstraction hierarchy only has to interact with the next lower level

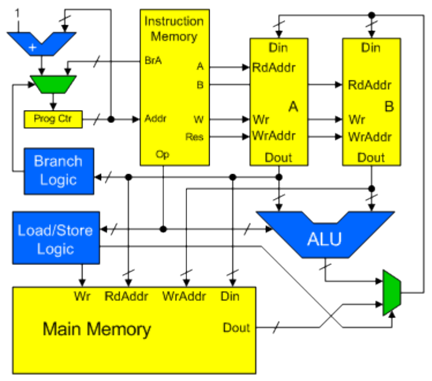
IA32 Operating Modes (2)

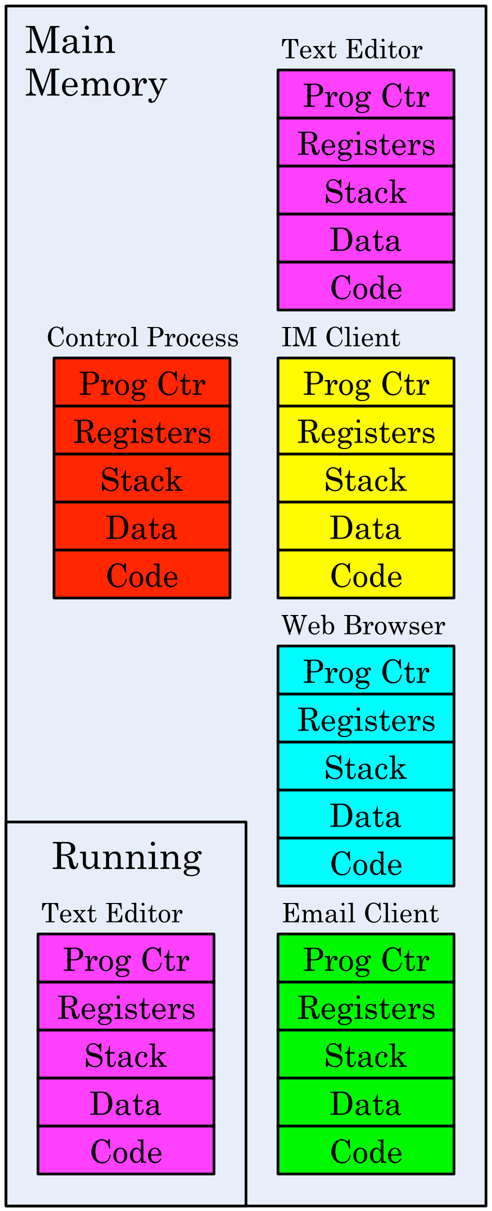
- Some processors provide more than just two operating modes
- IA32 provides four different operating modes
- Reason:
  - Some software components need more privileged access to the processor, but they don't need to access *everything*…
  - Device drivers, specific operating system services, etc.
  - Partition OS code into privilege levels
- Modes form a security hierarchy:
  - Lower number = higher privileges
  - Each privilege level has its own stack and memory areas
- OS kernel runs at level 0
- OS services run at levels 1 and 2
- Applications run at level 3



Interruptions…

- Next question: how do we actually trigger the context switch?
  - How does the control process interrupt the currently running program?
- Also, what if the running program misbehaves?
  - e.g. inadvertently tries to manipulate another program’s memory, or runs an invalid instruction
- Somehow, we need to transfer control back to the control process in these cases.





Exceptional Control Flow

- Programs implement a flow of control
  - The sequence of instructions that are executed by the program
  - Loops, conditionals, subroutine calls – for processing data, handling different scenarios, or performing common operations
  - This is the logical control flow that is executed within a process
- Frequently, computer must also handle various interruptions that occur
  - Hardware events, timer events, program crashes, etc.
  - To handle the event, must jump to a very different location, often not even in the current process' code
  - This is called exceptional control flow

Exceptional Control Flow (2)

- Several major causes of exceptional control flow
- Interrupts
  - Caused by hardware signaling to the processor
  - e.g. external I/O devices that are ready to do stuff
  - Usually <u>not</u> caused by execution of a specific instruction
  - (Software can also invoke an interrupt handler manually, if desired…)
- Exceptions
  - Caused by a program executing an instruction
  - Exception can be intentional, to perform a task…
  - Or, it may be unintentional, if an error occurred

Software and Hardware Exceptions

- In languages like C++ and Java, code that throws an exception simply stops executing!
  - Java: "abnormal termination" or "abrupt completion"
  - Control transfers to exception handler, and doesn't return back to the code that caused the exception
- Hardware exception handling is quite different!
  - <u>Frequently</u> have exceptions where we *want* to return to the instruction that caused the exception
- Four classes (kinds) of exceptions:

Interrupt	Signal from hardware device	Always returns to next instruction
Trap	Intentional exception	Always returns to next instruction
Fault	Potentially recoverable error	Might return to current instruction
Abort	Nonrecoverable error	Never returns

Exception Classes

- **Interrupts** are caused by hardware
  - Example: a periodic timer interrupt
  - Handler can respond to the hardware interrupt
  - Then, control returns back to interrupted program
- **Traps** are intentional exceptions caused by programs
  - Frequently used to implement calls to operating system
  - Caller specifies the requested service when invoking the exception
  - Processor switches from user-mode to kernel-mode when jumping to the exception handler
  - Operating system can provide the requested service…
  - Then, control returns back to interrupted program

Exception Classes (2)

- **Faults** are unintentional exceptions caused by software
  - Faults represent error conditions that might be recoverable
- Example: virtual memory that is paged to disk
  - Program accesses a page that isn't in memory
  - Processor causes a *page fault*, which invokes the page fault handler
  - Handler loads requested page from disk into memory
  - Execution occurs with the instruction that caused the fault (<u>not</u> the next instruction!)
    - …now the instruction will presumably succeed.
- If a fault handler cannot recover from a fault, the program is usually terminated

Exception Classes (3)

- **Aborts** are unrecoverable fatal errors
  - Frequently used to handle hardware errors
  - Example: IA32 Machine-Check exception is an abort
  - Handler never returns to the interrupted program
- In fact, entire system may grind to a halt!
  - Windows "Blue Screen of Death" and Linux kernel-panic can both occur because of an abort

IA32 Exceptions

- IA32 processors support 256 different kinds of exceptions
  - Each is assigned an integer from 0 to 255
- Exception types 0 to 31 are IA32 architecture-defined interrupts and exceptions
- Some examples:
  - Exception 0 is a divide-by-zero fault
  - Exception 13 is a general protection fault
    - Frequently caused when programs use an invalid pointer
    - Segmentation faults! ☺
  - Exception 18 is a machine-check abort
    - Called when a hardware error of some kind is detected

IA32 Exceptions (2)

- Exception types 32-255 are user-defined exceptions
  - Can be assigned to hardware devices, used by the operating system, etc.
  - On UNIX platforms, exception 128 (0x80) is a trap used for making operating system calls
    - (more on this next time!)

- Can invoke the handler for any exception type with IA32 instruction **`int`** ***`n`***
  - ***`n`*** is the type of the exception

IA32 Interrupt Descriptor Table

- When an exception occurs, the processor must transfer control to the appropriate handler…
- Using the exception's type, processor looks up the handler to call in the Interrupt Descriptor Table
- Interrupt Descriptor Table is a sequence of 256 entries, each of which is 8 bytes
  - IDT can reside anywhere in memory; address is stored in **`idtr`** register
  - Operating system sets up this table using **`lidt`**/**`sidt`**
- When an exception ***n*** occurs:
  - Processor retrieves the 8-byte descriptor stored at the address **`idtr + 8 * n`**
  - Uses this descriptor to invoke the exception handler

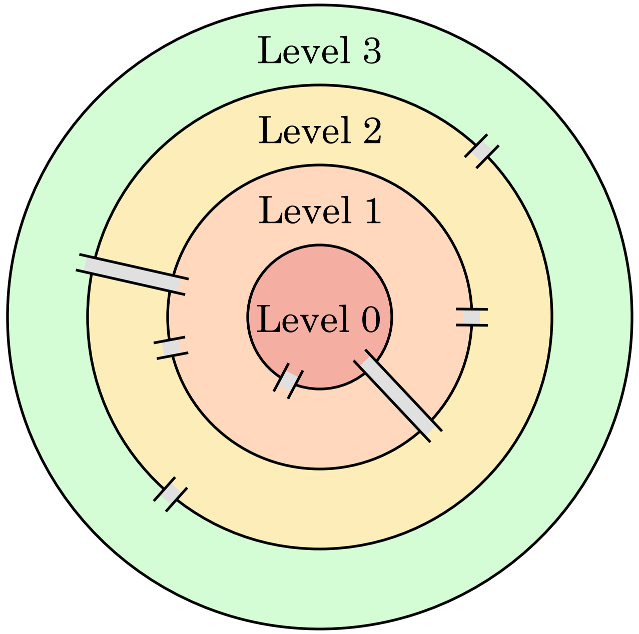
Interrupt Descriptors and Gates

- Interrupt descriptor record encodes both a call address and privilege information
  - Record is called a *gate descriptor*
- IA32 has several kinds of gates:
  - Call gates – for **`call`** or **`jmp`** operations across privilege boundaries
  - Interrupt gates, trap gates – for invoking exception handlers at a different privilege level
  - Task gates – for tasks that can be dispatched by the processor and performed at a different privilege level
- All gates function on same basic principle:
  - "If the caller has at least privilege level A, invoke the handler at privilege level B."

IA32 Gates

- *Why are these things called gates?!*

- Our privilege ring model:
  - Lower levels are more secure than higher levels
  - Can't just cross privilege boundaries! Causes a fault.
- Gates allow us to move from a lower privilege level into a higher privilege level
  - Literally provides a gateway between privilege levels
  - The hardware can verify or disallow the transition

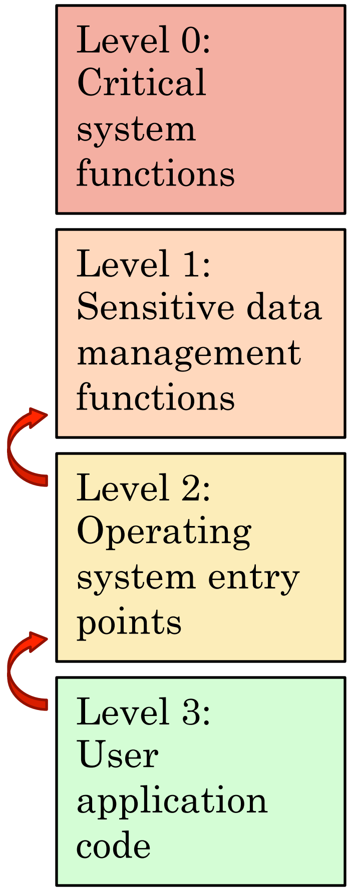


IA32 Gates, Operating-System Calls

- Gates allow operating system to carefully manage privilege levels
- In interrupt, fault, and abort handlers:
  - If handler needs to use privileged instructions (and has been verified to be secure!), simply move to the higher privilege level
  - User program was interrupted, so they can't affect the handler anyway
- In trap handlers (e.g. operating system calls):
  - Can start out at a lower privilege level
  - Examine the request, and the user/process/caller for whether they are allowed to make the request
  - If allowed, handler can perform an inter-privilege call through a call-gate to move to a higher privilege level

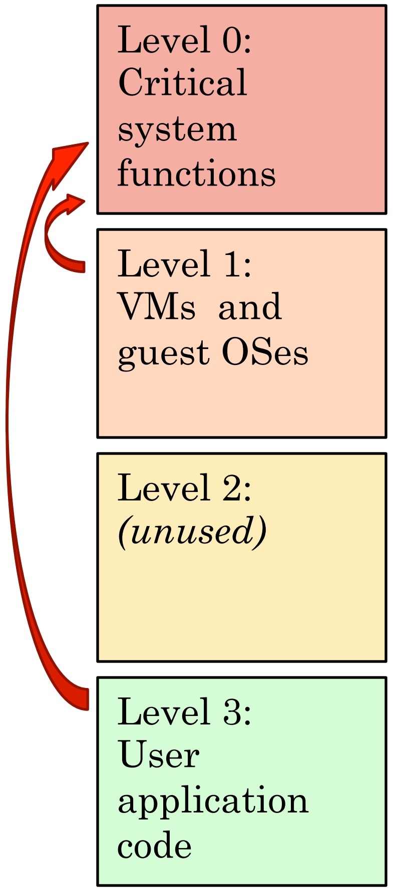
IA32 Gates, System Calls (2)

- Lowest privilege level on IA32 is level 3, highest is level 0
- Can partition operating system code into different levels, based on needs of code
- Example:
  - Application (level 3) makes a system call to modify a user password
  - Trap handler is invoked at level 2 via a trap-gate, and examines caller's identity
  - If caller is allowed to make the change, move to level 1 via a call-gate, and call the code that modifies the password
  - Otherwise, report an access-denied error to the caller


Level 0: Critical system functions
Level 1: Sensitive data management functions
Level 2: Operating system entry points
Level 3: User application code

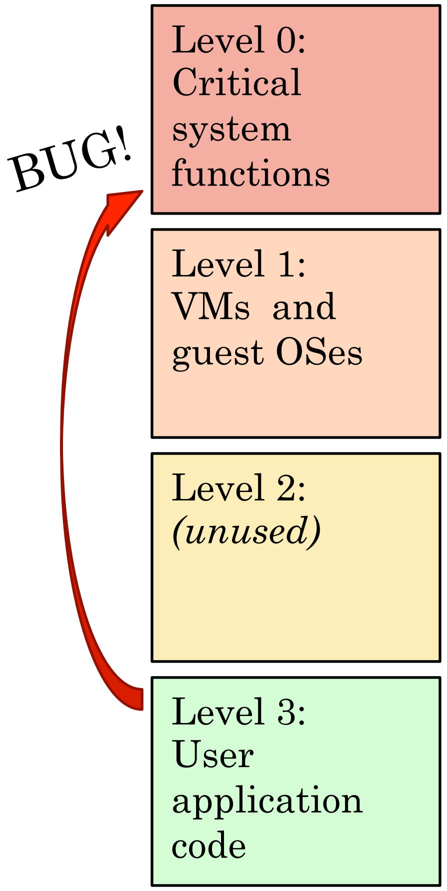
IA32 Gates, System Calls (3)

- In reality, most operating systems only use level 3 and level 0
- Example:
  - Application (level 3) makes a system call to modify a user password
  - Trap handler transitions immediately to level 0, verifies the call, and performs the operation
- Guest operating systems running in virtual machines can be run at level 1
  - Host operating system still retains total control over hardware
  - Guest OS needs to access some hardware functionality, but doesn't have full control


Level 0: Critical system functions
Level 1: VMs and guest OSes
Level 2: *(unused)*
Level 3: User application code

OS Calls and Privilege Escalation

- Clearly, operating system must move between privilege levels very carefully!
  - Code at higher privilege level must also be secure from exploits
- Privilege escalation exploits:
  - Take advantage of a bug in the OS code to perform operations at a higher privilege level than you should have access to!
- If operating system code has buffer-overflow issues (lecture 8), attacker can use this problem to invoke privileged code


BUG!
Level 0: Critical system functions
Level 1: VMs and guest OSes
Level 2: (unused)
Level 3: User application code

IA32 Privilege Levels

- A nice feature of IA32 privilege levels: **Each privilege level has its own stack!**
- Makes it harder for lower-level code to interfere with execution in higher privilege levels
  - Also ensures that higher privilege levels will have sufficient stack space to service requests!
- Also makes it more challenging for caller to pass arguments on the stack, across privilege levels
  - For call gates, a mechanism is provided to pass arguments on the stack
  - For exception gates, simply cannot pass arguments on the stack

IA32 Interrupt Operation

- IA32 interrupt operation very similar to a **`call`**
  - Processor saves return-address onto the stack
  - Processor also saves **`eflags`** register onto stack
    - **Note:** the **`call`** instruction doesn't do this!
- Two reasons that **`eflags`** needs to be saved!
  - Reason 1: Interrupted code might have been in the middle of a comparison operation!
  - Interrupt could be triggered by hardware or software

```
...
cmp 16(%ebp), %esi
jge end_for
...
```

⇦ Timer interrupt!

  - When exception handler returns, must be able to pick up where we left off

IA32 Interrupt Operation (2)

- Another scenario:
  - Application code is executing…
  - A hardware interrupt occurs, and execution transfers to the interrupt handler
  - While interrupt handler is executing, the same hardware interrupt occurs again!
    - Can end up in a situation where handler *never* completes
- IA32 **`eflags`** register has an Interrupt Flag
  - When set to 1, maskable interrupts are enabled
    - Hardware signal will cause interrupt handler to be invoked
  - When set to 0, maskable interrupts are disabled
    - Hardware signal doesn't cause handler to be invoked
- When a hardware interrupt occurs, IA32 clears the Interrupt Flag automatically
  - Hardware interrupt handlers can't interrupt themselves

IA32 Interrupt Flag

- Reason 2: The interrupt/exception operation itself usually changes **`eflags`**
  - Therefore, save **`eflags`** onto stack before running handler
  - (Other flags can also be changed by exceptions…)
- When returning from exception handler, **`eflags`** is restored from the stack
  - Also automatically re-enables interrupts
- Not all exceptions cause Interrupt Flag to be cleared!
  - Traps (e.g. **`int $0x80`**) do not disable the Interrupt Flag
- The Nonmaskable Interrupt (NMI) cannot be disabled by the Interrupt Flag
  - For high-priority hardware events that must be handled
  - While an NMI is being handled, processor does ignore other NMIs until NMI handler returns

IA32 Exceptions and Protected-Mode

- Each IA32 operating mode has its own stack…
- IA32 exceptions may also change which stack is currently being used!
- IA32 protected-mode interrupt sequence:
  - If handler's privilege level is different from caller's privilege level, must change to the handler's stack
    - Save location of caller's stack onto the handler's stack
    - If handler is at same privilege level as caller, no stack-save operations are performed
  - Save caller's return-address onto handler's stack
  - Save **`eflags`** onto handler's stack
- All information necessary for resuming execution at caller is now saved. (phew!)

Returning from Exception Handler

- Processor does lots of work when invoking an exception handler!
  - **`int`** operation does a lot of work…
- Returning from a handler is similarly complex
- Handled by the **`iret`** instruction
- In protected-mode, **`iret`** does the following:
  - Restore **`eflags`** register from stack
  - Restore instruction pointer from stack
  - If caller is at a different privilege level, resume using the caller's stack, using info saved on handler's stack
    - If at same privilege level, no need to change the stack in use

Back to the Process Abstraction…

- Needed a way for the control process to manage application processes
  - Perform periodic context-switches between processes
  - If a process misbehaves, intercept the error and terminate the process
- Both tasks become easy with exception handlers
- For context-switches:
  - Create an interrupt handler that is invoked periodically by the processor's timer
  - Set the handler to run in privileged mode, so the controller can access all memory
  - When the timer interrupts the currently running process, control process can suspend the application process and switch the context to another process
  - Return from the handler to the new process to run

Back to the Process Abstraction (2)

- When a process misbehaves, the processor will invoke a fault or abort exception handler
  - e.g. general protection fault, exception 13
- Control process can register handlers for these exceptions:
  - When a process causes a general protection fault, the control process is invoked, and it can terminate the misbehaving process
  - Then, switch to another, more well-behaved process
- Now we can fully implement our virtual process abstraction!

Today: Summary

- Virtualization is central concept in both modern processor design, and in operating system design
  - Virtual processors (processes), virtual memory
- Once we try to run multiple programs on a single processor "at the same time," many issues arise!
  - Want to isolate the address space of different processes from each other
  - Need to ensure that internal memory-layout details of various programs are irrelevant to each other
  - Need to figure out when and how to context-switch between active processes

TODAY: SUMMARY (2)

- Can solve these problems with a combination of hardware and software techniques
- Exceptional control flow allows us to handle errors, other asynchronous notifications easily
- Processor operating modes and privilege levels allows us to write a control process that can manage processes, while keeping application processes at a more restricted privilege level
- First step into what operating systems do for us!
  - Common facilities that all programs can benefit from
  - Software that manages the computer hardware, and provides useful abstractions for our programs