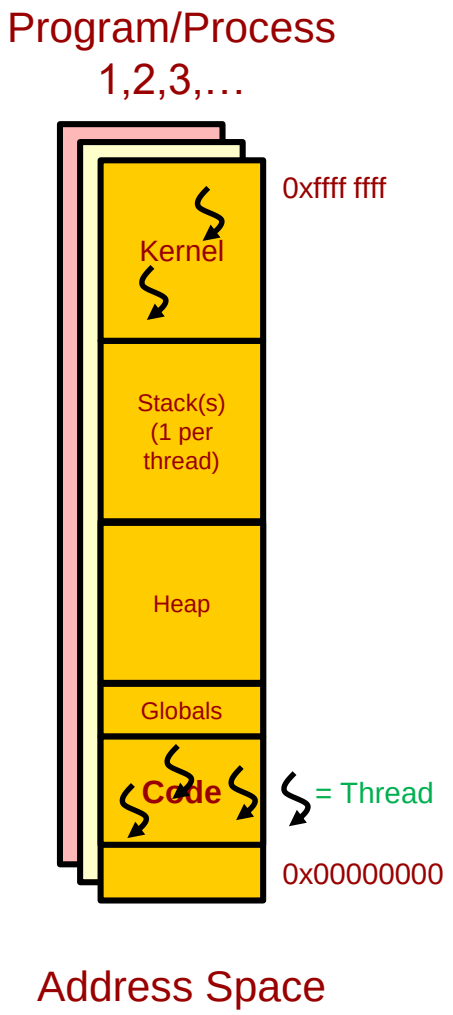
CSCI 350
Ch. 2 – The Kernel Abstraction & Protection

Mark Redekopp

PROCESSES & PROTECTION

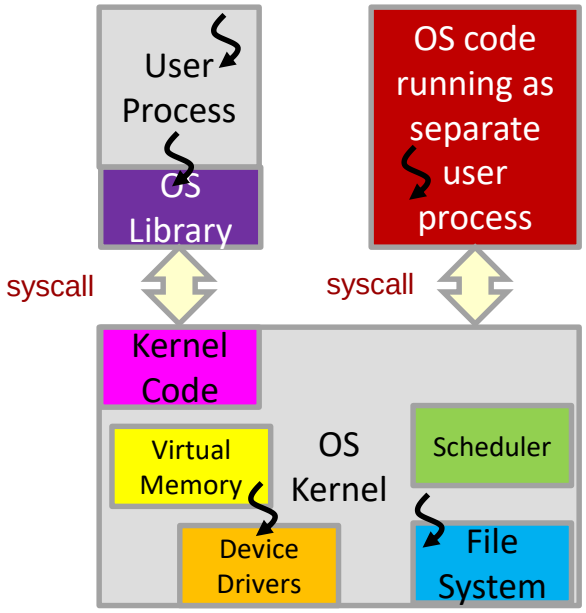
Processes

- **Process**
  - **(def 1.) Address Space + Threads**
    - 1 or more threads
  - **(def 2.) : Running instance of a program that has limited rights**
    - Memory is protected: HW + kernel use address translation (VM) to ensure no access to any other processes' memory
    - CPU is protected: Process can be pre-empted (context-switched)
    - I/O is protected: Processes execute in user-mode (not kernel mode) which generally means direct I/O access is disallowed instead requiring **system calls** into the kernel
- Kernel is not considered a "process"
  - Has access to all resources and much of its code is invoked under the execution of a user process thread (i.e. during a system call)
  - Thought it can have its own independent threads



The Kernel

- Kernel is trusted and has access to everything else
  - The manager of HW & processes
- Kernel is in charge of protection
- Provides access to services via **syscalls**



REVIEW OF USER VS. KERNEL MODE

User vs. Kernel Mode

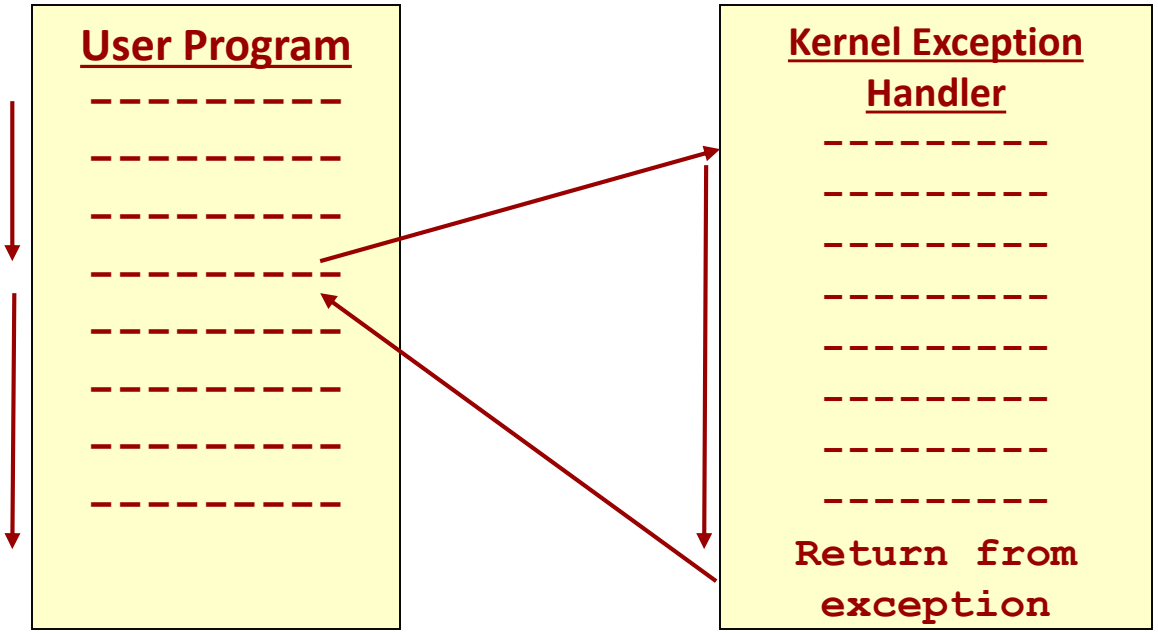
- Kernel mode is a special mode of the processor for executing trusted (OS) code
  - Certain features/privileges are only allowed to code running in kernel mode
  - OS and other system software should run in kernel mode
- User mode is where user applications are designed to run to limit what they can do on their own
  - Provides protection by forcing them to use the OS for many services
- User vs. kernel mode determined by some bit(s) in some processor control register
  - x86 Architecture uses lower 2-bits in the CS segment register (referred to as the Current Privilege Level bits [CPL])
  - 0=Most privileged (kernel mode) and 3=Least privileged (user mode)
    - Levels 1 and 2 may also be used but are not by Linux
- **On an exception, the processor will automatically switch to kernel mode**

Exceptions

- Any event that causes a break in normal execution
- Asynchronous exceptions
  - Hardware Interrupts / Events
    - Handling a keyboard press, mouse moving, USB data transfer, etc.
    - We already know about these so we won't focus on these again
- Synchronous exceptions
  - Error Conditions
    - **Page fault**, Invalid address, Arithmetic/FP overflow/error
  - System Calls / Traps
    - User applications calling OS code services switches to kernel mode
- General idea: When these occur, automatically call some subroutine (a.k.a. "handler") in kernel mode to handle the issue, then resume normal processing

Exception Processing

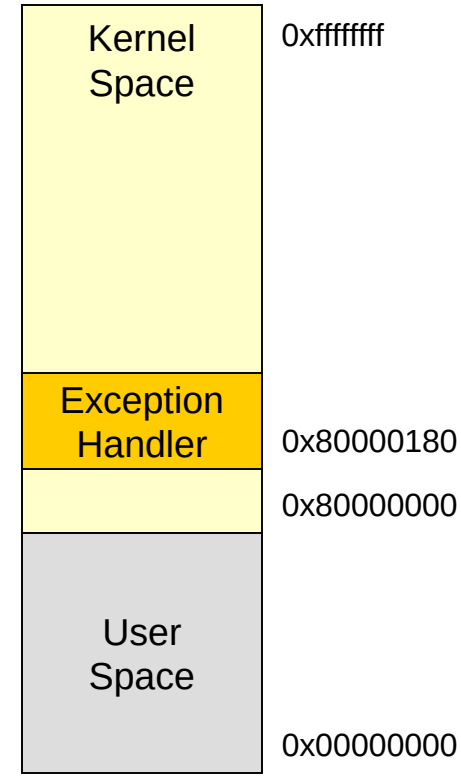
- Where will you be in your program code when an interrupt occurs?
- An exception can be…
  - Asynchronous (due to an interrupt or error)
  - Synchronous (due to a system call/trap)
- Must save PC of offending instruction, program state, and any information needed to return afterwards
- Restore upon return



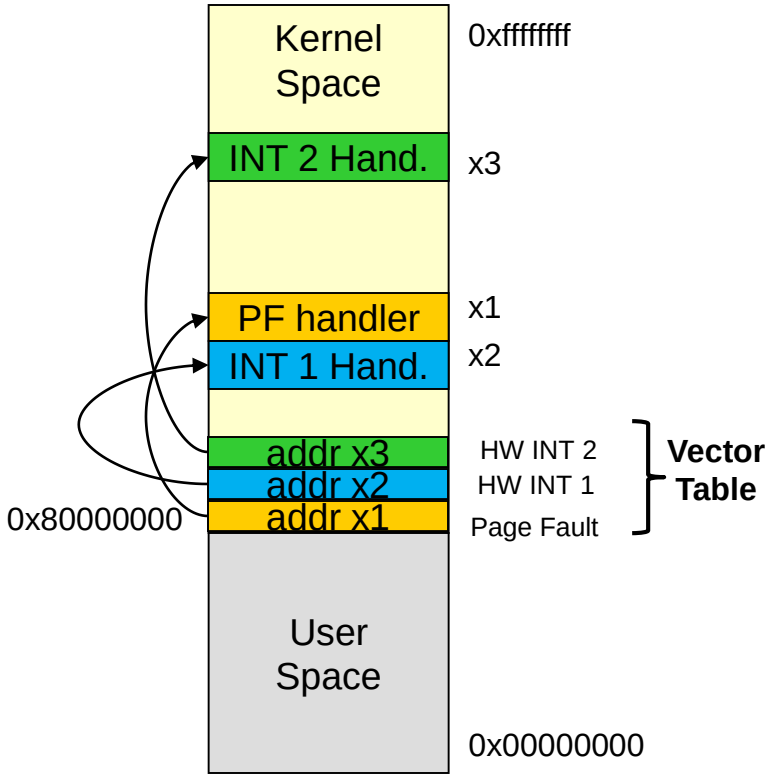
Exception Processing

- Now that you know what causes exceptions, what does the hardware do when an exception occurs?
- Save necessary state to be able to restart the process
  - Save PC of current/offending instruction
- Change to KERNEL MODE if not already
- Call an appropriate “handler” routine to deal with the error / interrupt / syscall
  - Handler identifies cause of exception and handles it
  - May need to save more state
- Restore state (and previous mode) and return to offending application (or kill it if recovery is impossible)

Handler Calling Methods



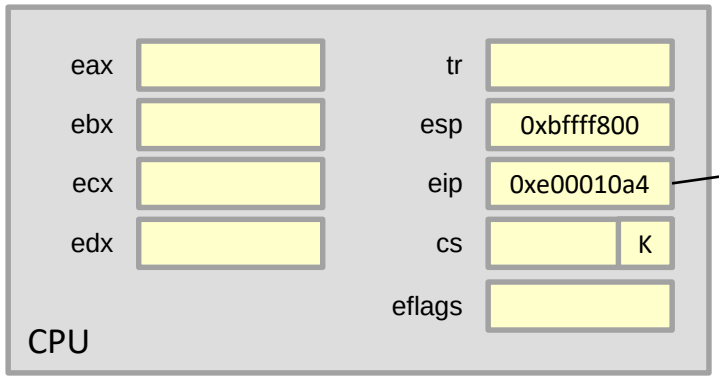
Hardwired Handler Address

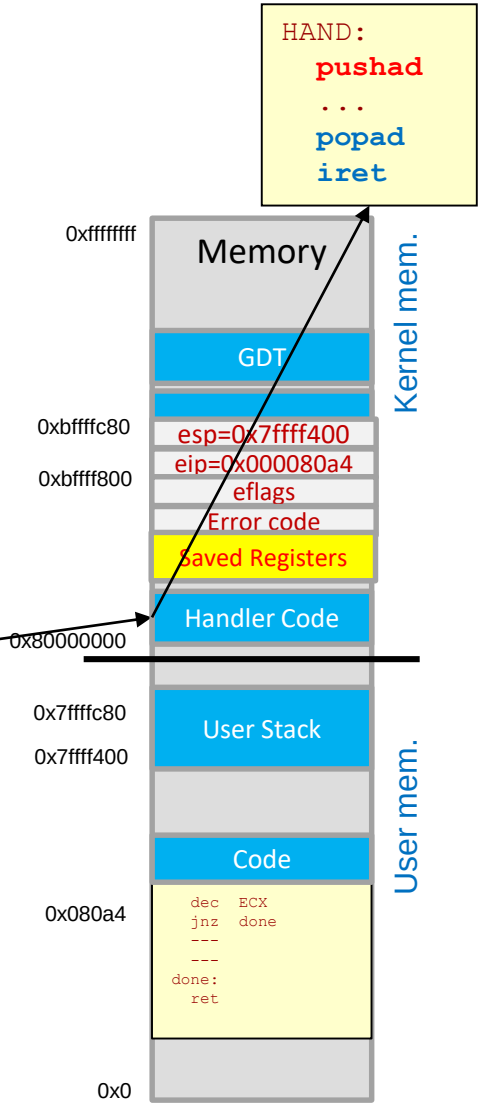


Interrupt / Vector Table

Transition from User to Kernel Mode

- Recall on an interrupt or any exception
  - HW changes to kernel mode, saves some registers & pushes them on kernel stack
  - Vector table is used to look up handler and start execution
  - Handler saves more state then executes
  - Restores registers from kernel stack and returns to user mode



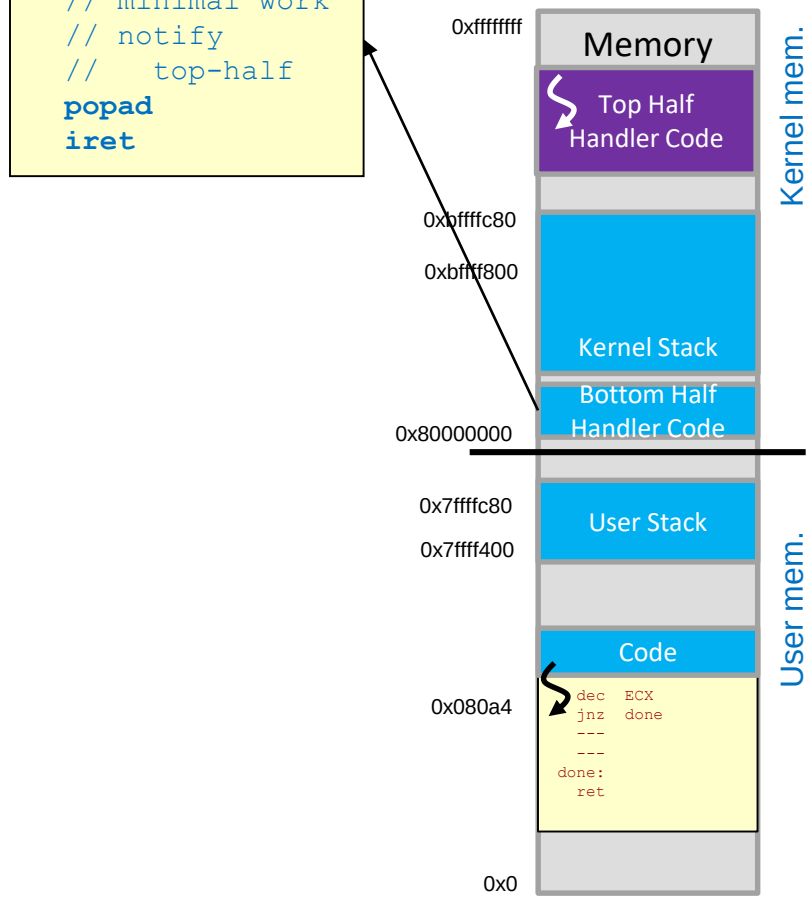


- **Question**: What's the difference between a mode switch and a context switch?

Interrupts

- Most systems don't allow new interrupts while currently handling an interrupt
- Important: Get in and out of an interrupt handler quickly
- Common interrupt handler architecture: bottom- and top-half
  - Bottom-half: actual interrupt handler
    - Do minimal work needed to deal with the HW issue
    - Signal or queue-up work for the top half
  - Top-half: Executed in separate thread from bottom-half
    - Can perform work and itself be interrupted

```
HAND:
  pushad
  // minimal work
  // notify
  //   top-half
  popad
  iret
```



Interrupts in Pintos

```
/* Interrupt stack frame. */
struct intr_frame
  {
    /* Pushed by intr_entry in intr-stubs.S.
       These are the interrupted task's saved registers. */
    uint32_t edi;               /* Saved EDI. */
    uint32_t esi;               /* Saved ESI. */
    uint32_t ebp;               /* Saved EBP. */
    uint32_t esp_dummy;         /* Not used. */
    uint32_t ebx;               /* Saved EBX. */
    uint32_t edx;               /* Saved EDX. */
    uint32_t ecx;               /* Saved ECX. */
    uint32_t eax;               /* Saved EAX. */
    uint16_t gs, :16;           /* Saved GS segment register. */
    uint16_t fs, :16;           /* Saved FS segment register. */
    uint16_t es, :16;           /* Saved ES segment register. */
    uint16_t ds, :16;           /* Saved DS segment register. */

    /* Pushed by intrNN_stub in intr-stubs.S. */
    uint32_t vec_no;            /* Interrupt vector number. */

    /* Sometimes pushed by the CPU,
       otherwise for consistency pushed as 0 by intrNN_stub.
       The CPU puts it just under `eip', but we move it here. */
    uint32_t error_code;        /* Error code. */

    /* Pushed by intrNN_stub in intr-stubs.S.
       This frame pointer eases interpretation of backtraces. */
    void *frame_pointer;        /* Saved EBP (frame pointer). */

    /* Pushed by the CPU.
       These are the interrupted task's saved registers. */
    void (*eip) (void);         /* Next instruction to execute. */
    uint16_t cs, :16;           /* Code segment for eip. */
    uint32_t eflags;            /* Saved CPU flags. */
    void *esp;                  /* Saved stack pointer. */
    uint16_t ss, :16;           /* Data segment for esp. */
  };
```

Pintos: threads/interrupt.h

```
/* The Interrupt Descriptor Table (IDT).
   The format is fixed by the CPU.  See
   [IA32-v3a] sections 5.10 "Interrupt
   Descriptor Table (IDT)",
   5.11 "IDT Descriptors",
   5.12.1.2 "Flag Usage By
   Exception- or Interrupt-Handler Procedure". */
static uint64_t idt[INTR_CNT];

  /* Initialize IDT. */
  for (i = 0; i < INTR_CNT; i++)
    idt[i] = make_intr_gate (intr_stubs[i], 0);
```

Pintos: threads/interrupt.c

```
/* All the stubs. */
STUB(00, zero) STUB(01, zero) STUB(02, zero) STUB(03, zero)
STUB(04, zero) STUB(05, zero) STUB(06, zero) STUB(07, zero)
...
STUB(f8, zero) STUB(f9, zero) STUB(fa, zero) STUB(fb, zero)
STUB(fc, zero) STUB(fd, zero) STUB(fe, zero) STUB(ff, zero)

intr_entry:
  /* Save caller's registers. */
  pushl %ds
  pushl %es
  pushl %fs
  pushl %gs
  pushal /* Saves %eax,%ecx,%edx,%ebx,%esp,%ebp,%esi,%edi */
  ...
```

Pintos: threads/intr-stubs.s

Register Handlers

```
void
exception_init (void)
{
  /* These exceptions can be raised explicitly by a user program,
     e.g. via the INT, INT3, INTO, and BOUND instructions.  Thus,
     we set DPL==3, meaning that user programs are allowed to
     invoke them via these instructions. */
  intr_register_int (3, 3, INTR_ON, kill, "#BP Breakpoint Exception");
  intr_register_int (4, 3, INTR_ON, kill, "#OF Overflow Exception");
  intr_register_int (5, 3, INTR_ON, kill,
                     "#BR BOUND Range Exceeded Exception");

  /* These exceptions have DPL==0, preventing user processes from
     invoking them via the INT instruction.  They can still be
     caused indirectly, e.g. #DE can be caused by dividing by
     0.  */
  intr_register_int (0, 0, INTR_ON, kill, "#DE Divide Error");
  intr_register_int (1, 0, INTR_ON, kill, "#DB Debug Exception");
  intr_register_int (6, 0, INTR_ON, kill, "#UD Invalid Opcode Exception");
  intr_register_int (7, 0, INTR_ON, kill,
                     "#NM Device Not Available Exception");
  intr_register_int (11, 0, INTR_ON, kill, "#NP Segment Not Present");
  intr_register_int (12, 0, INTR_ON, kill, "#SS Stack Fault Exception");
  intr_register_int (13, 0, INTR_ON, kill, "#GP General Protection Exception");
  intr_register_int (16, 0, INTR_ON, kill, "#MF x87 FPU Floating-Point Error");
  intr_register_int (19, 0, INTR_ON, kill,
                     "#XF SIMD Floating-Point Exception");

  /* Most exceptions can be handled with interrupts turned on.
     We need to disable interrupts for page faults because the
     fault address is stored in CR2 and needs to be preserved. */
  intr_register_int (14, 0, INTR_OFF, page_fault, "#PF Page-Fault Exception");
}
```

Pintos: userprog/exception.c

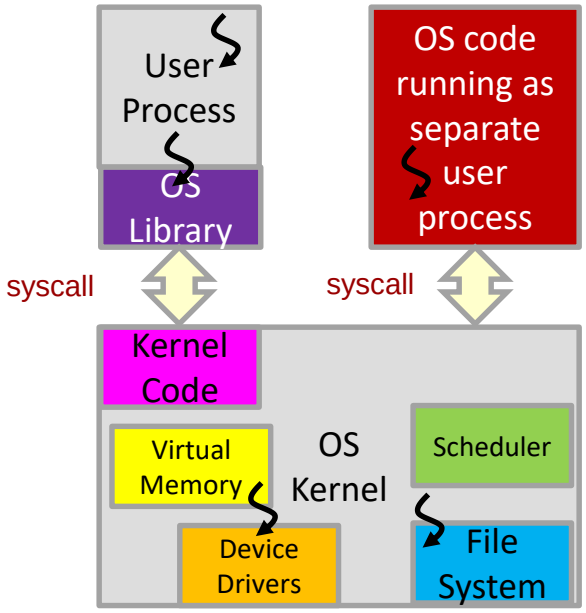
```
/* Sets up the timer to interrupt TIMER_FREQ times per second,
   and registers the corresponding interrupt. */
void
timer_init (void)
{
  pit_configure_channel (0, 2, TIMER_FREQ);
  intr_register_ext (0x20, timer_interrupt, "8254 Timer");
}

/* Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
}
```

Pintos: devices/timer.c

SYSCALL INTRO

Mode Switches

- What causes user to kernel mode switch?
  - An exception: interrupt, error, or syscall
- What causes a kernel to user mode switch?
  - Return from exception:
    - Interrupt
    - Voluntary/Blocking context switch
  - Upcalls/signals (more on this later)



Syscalls

- Provide a controlled method for user mode applications to call kernel mode (OS) code
  - OS will define all possible system calls available to user apps.
  - Generally defined by number and necessary arguments
- Syscalls are an exception and thus switch to kernel mode
- MIPS Syntax: syscall
- x86 Syntax: INT 0x80 (value between 0-255 or 0x00-0xff)
  - Service num. placed in EAX or on stack
  - Pintos uses INT 0x30 with arguments on the stack
    - 1st argument is the syscall number since INT 0x30 serves ALL syscall requests
    - Remaining arguments are specific to desired syscall

```
/* System call numbers. */
enum
 {
  /* Projects 2 and later. */
  SYS_HALT,                   /* 0 = Halt the operating system. */
  SYS_EXIT,                   /* 1 = Terminate this process. */
  SYS_EXEC,                   /* 2 = Start another process. */
  SYS_WAIT,                   /* 3 = Wait for a child process to die. */
  SYS_CREATE,                 /* 4 = Create a file. */
  SYS_REMOVE,                 /* 5 = Delete a file. */
  SYS_OPEN,                   /* 6 = Open a file. */
  SYS_FILESIZE,               /* 7 = Obtain a file's size. */
  SYS_READ,                   /* 8 = Read from a file. */
  SYS_WRITE,                  /* 9 = Write to a file. */
  ...
  };
```

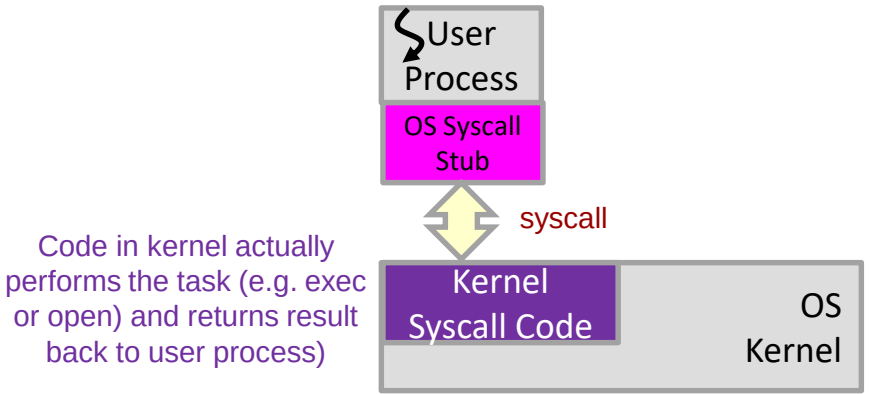
Pintos: lib/syscall-nr.h

User Stub & Kernel Side

- Performing a system call requires pushing arguments on stack and then executing the INT or SYSCALL instruction
- To abstract some of this, the OS usually provides a user-level library of ***stubs*** giving a nice API to the programmer
  - The stubs just invokes the syscall
- The syscall is then treated as an exception and transitions to kernel code
- The kernel then examines the stack and calls the desired operation

```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)                                          \
        ({                                                              \
          int retval;                                                   \
          asm volatile                                                  \
            ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
               : "=a" (retval)                                          \
               : [number] "i" (NUMBER),                                 \
                 [arg0] "g" (ARG0)                                      \
               : "memory");                                             \
          retval;                                                       \
        })

/* Nice API for Applications to call */
pid_t
exec (const char *file)
{
  /* Really just invokes the INT 0x30 instruction */
  return (pid_t) syscall1 (SYS_EXEC, file);
}

int
open (const char *file)
{
  return syscall1 (SYS_OPEN, file);
}
```

Pintos: User-side **stub** in lib/user/syscall.c



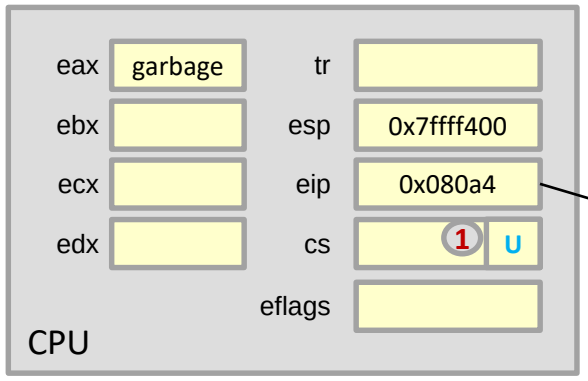
Code in kernel actually performs the task (e.g. exec or open) and returns result back to user process)

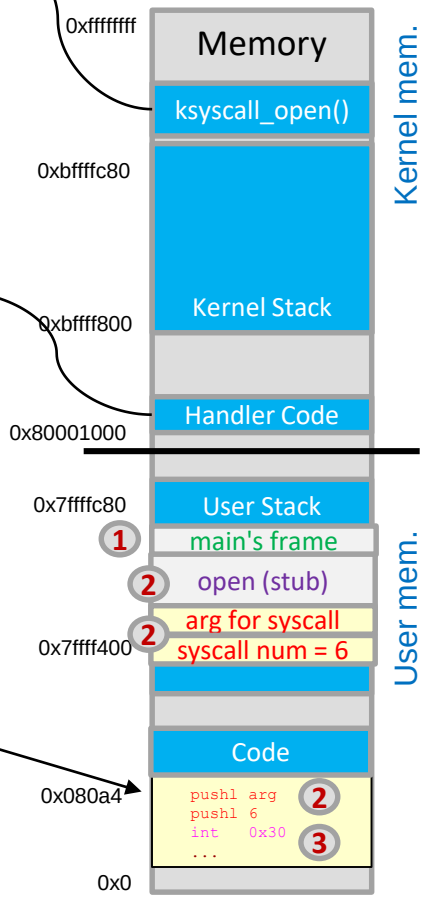
Syscall Mechanism 1

1. Suppose a user process (thus in user mode) executing **main()** in wants to open a file
2. It will first call the user-level stub for open which will push the argument and the syscall number on the stack
3. Then it will execute the INT 0x30 instruction

```
int ksyscall_open(...)
{
  /* extract arguments */
  /* perform task */
  return value;
}
```

```
HAND30:
  pushad
  /* Extract syscall num    */
  /* If syscall num == 0 ...*/
  /* ...                    */
  /* If syscall num == 6    */
  call ksyscall_open
  /* store retval in %eax */
  ...
  popad
  iret
```



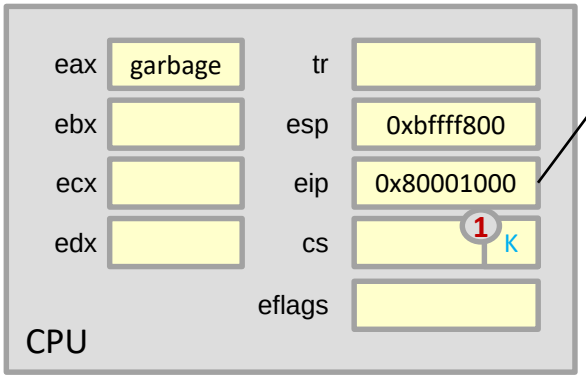


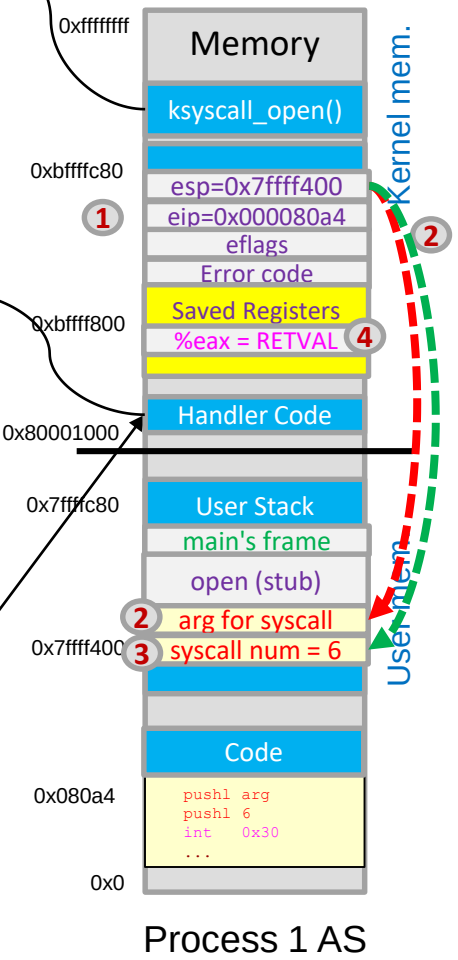
Syscall Mechanism 2

1. The HW will enter kernel mode and along with the first portion of the handler will save the user-level registers onto the kernel stack
   - Note the old value of %eax (whatever garbage it is) will be saved
2. The handler must now extract the syscall number from the stack.
   - How?
   - Using the user-level saved %esp
3. By looking at the syscall number we can know to call the real kernel implementation for open() (i.e. ksyscall_open)
   - Can also extract the argument from the user stack
4. Return value must be communicated back in %eax so we place it in the saved stack version of %eax

```
int ksyscall_open(...)
{
  /* extract arguments */
  /* perform task */
  return value;
}
```

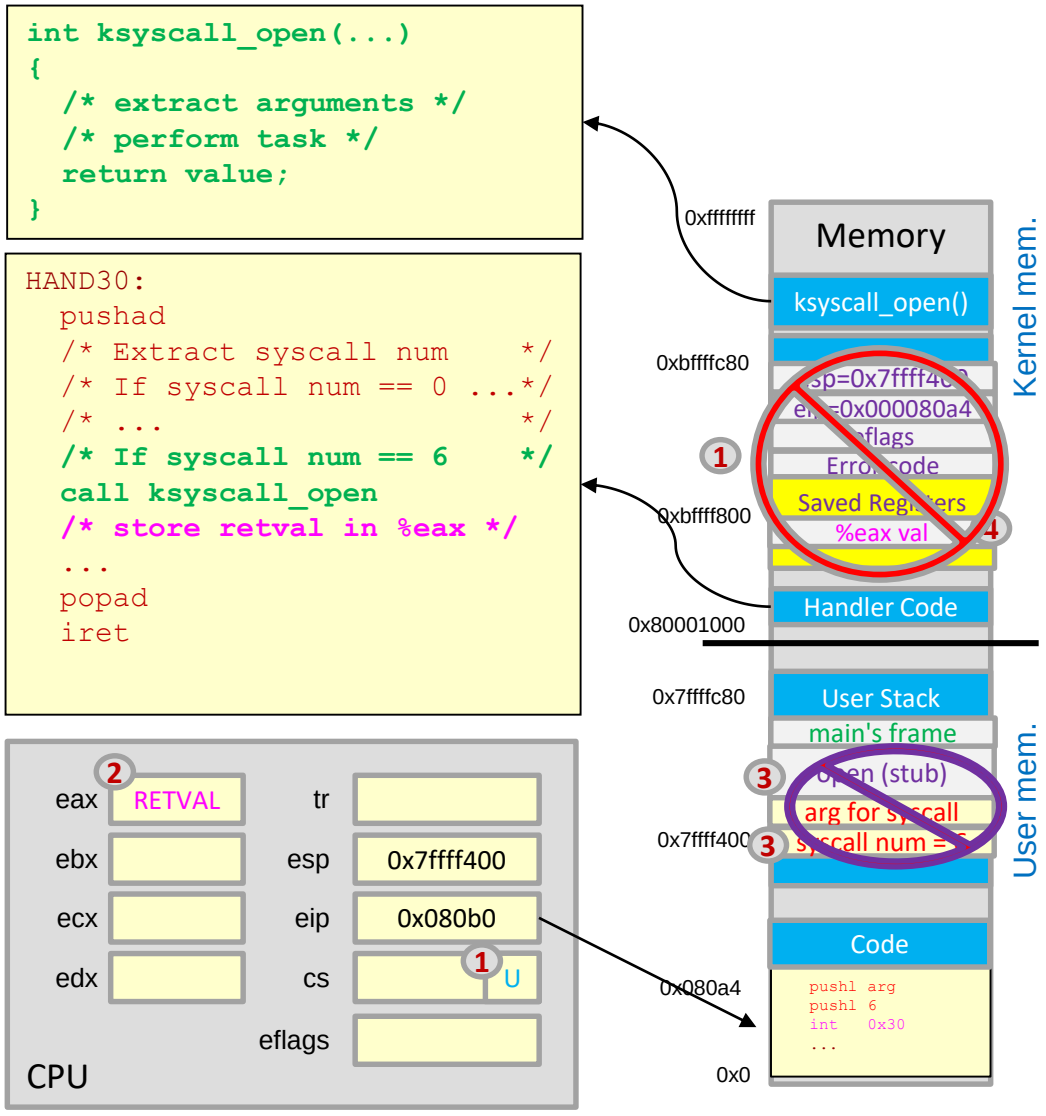
```
HAND30:
  pushad
  /* Extract syscall num     */
  /* If syscall num == 0 ...*/
  /* ...                     */
  /* If syscall num == 6     */
  call ksyscall_open
  /* store retval in %eax */
  ...
  popad
  iret
```





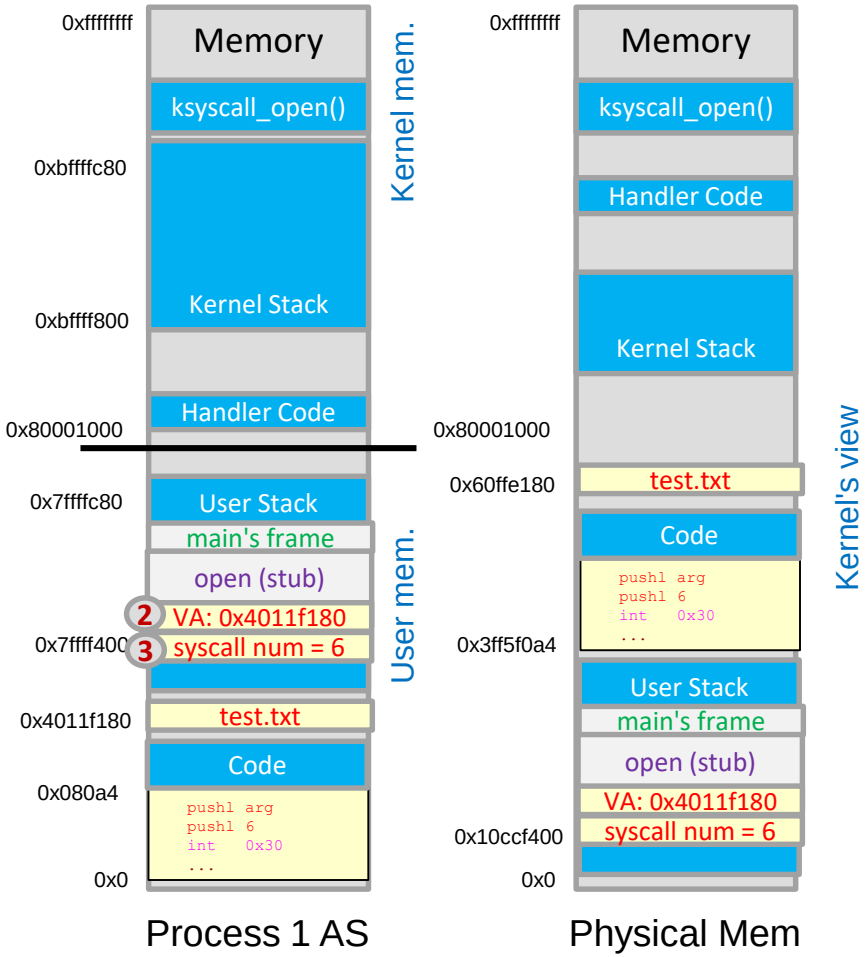
Syscall Mechanism 3

1. When done, the handler then restores all the registers and state
   - User mode is restored
2. The return val is in %eax
3. The user level stub can now clean up the stack and return the value back to main()



Notes

- This is the subject of project 2 in Pintos
- You will implement the syscall handling
- One note: Any pointers passed from the user process on the user stack are virtual addresses and must be translated to the kernel's address space



Syscall Protection

- The kernel must protect itself against errant or malicious user arguments to the syscalls
- Examples:
  - Validate pointers: Ensure they point at legal memory for that user process
  - Validate formats: Check that a C-string is null-terminated or stop before going off valid memory regions
  - Copy arguments to kernel space to avoid TOCTOU (Time of check vs. Time of Use) attacks
    - Some other thread, process or device modifying the argument after it is check but before it is used

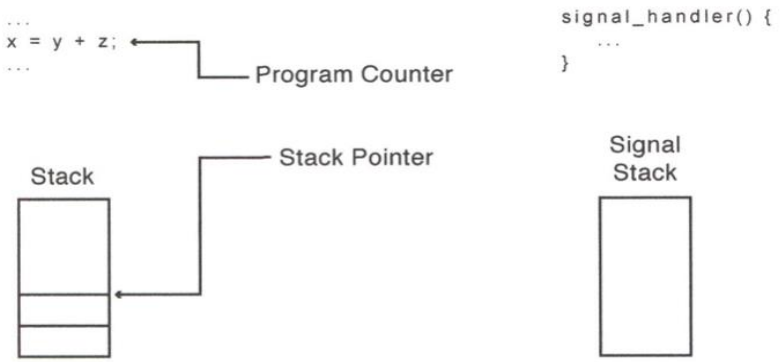
UPCALLS (SIGNALS)

What is an Upcall

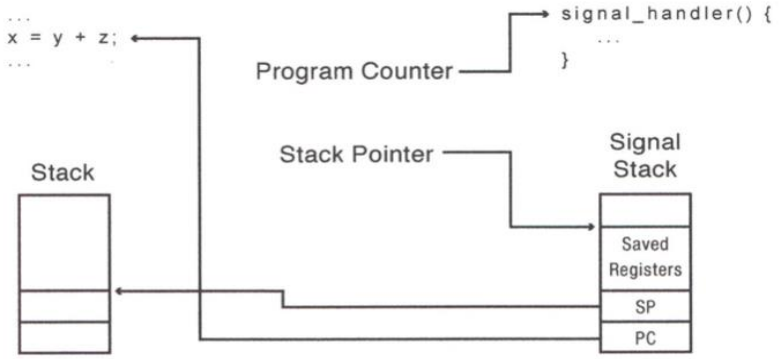
- An upcall or signal is like a user-level interrupt (interrupt to a user-process)
  - The kernel delivers some kind of event to the user process
  - The user process may or may not be executing
  - An upcall or signal is NOT a response to a synchronous request by the user processor
- Examples
  - Asynchronous I/O: The process requested some I/O but did not want to wait, instead asking to be notified upon completion
  - Interprocess communication: Another process has sent data or notification to this process requiring immediate attention
    - Example: Debugger restarting a process, Logout/shutdown event to applications telling them to save data and exit
  - User-level exception processing
    - Many OS's give applications the chance to respond to an error or other even
    - SIGINT = Interrupt = Ctrl+C
    - SIGSEGV = Segmentation Violation = The dreaded seg-fault
    - SIGFPE = Floating-point exception

Supporting Upcalls/Signals

- Signals are handled like kernel exceptions but at the user level
  - Recall the kernel had to save the current state on its own stack
  - It could then execute a kernel handler
- Thus, the user process may have a separate signal stack
  - When a the kernel triggers a signal, the current process state can be saved on the signal stack and then the handler (registered in advance) can be called
  - When a signal handler ends it will restore the state from the signal stack



Just Before the signal

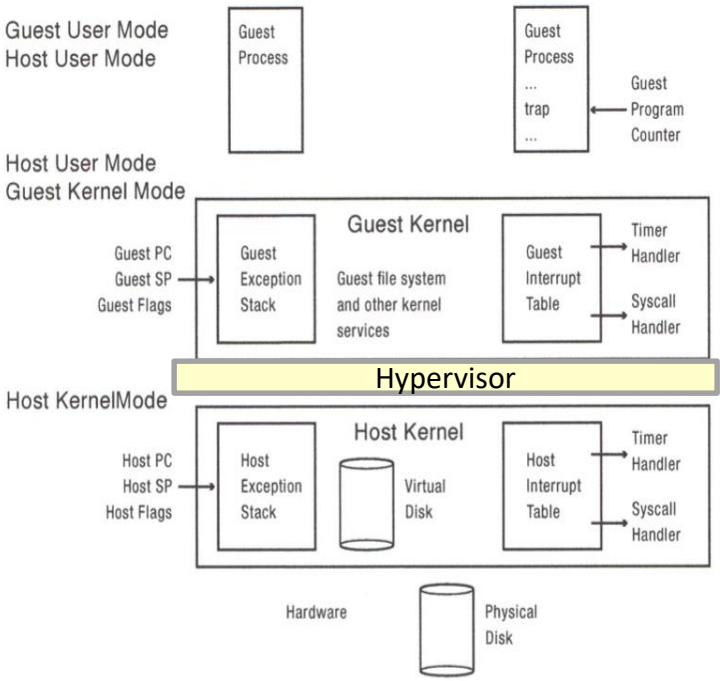


Just After the signal

VIRTUAL MACHINES

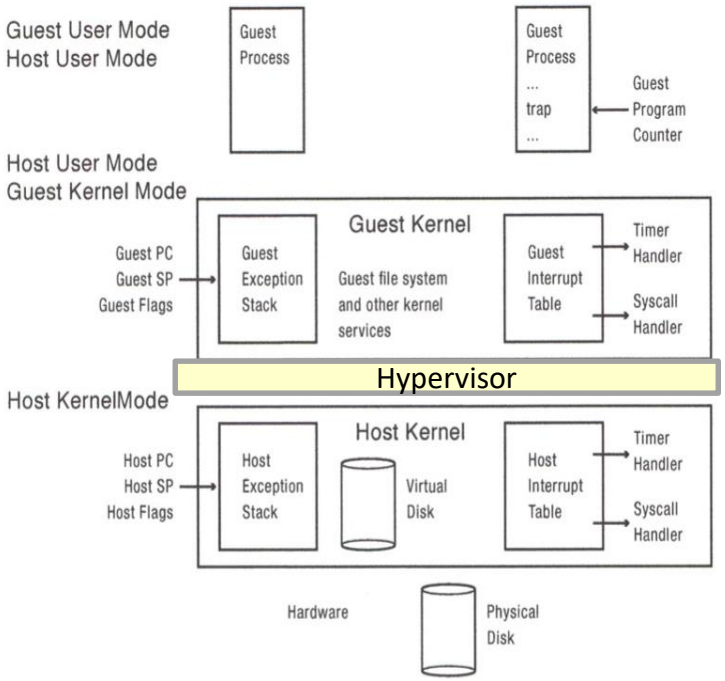
Virtual Machine Definitions

- Definitions
  - Host OS: OS running on the bare HW
  - Guest OS: OS running in a virtual environment
  - Hypervisor: Layer that often interfaces the Guest and Host OS to each other



Using Protection

- We can use the protection and virtualization mechanisms provided by the hardware in our favor
- When a guest user process performs a syscall, it will trap into the host OS, which can then redirect it to the guest OS
- When the guest kernel (which doesn't know it is running in user mode) executes a privileged instruction or hardware access, an exception will be generated to the host OS which can then perform the desired guest OS operation and restart it
  - Example: When the Guest OS tries to read from disk it will generate an exception, allowing the Host OS to read normally from a file that "acts-like" the disk for the Guest system

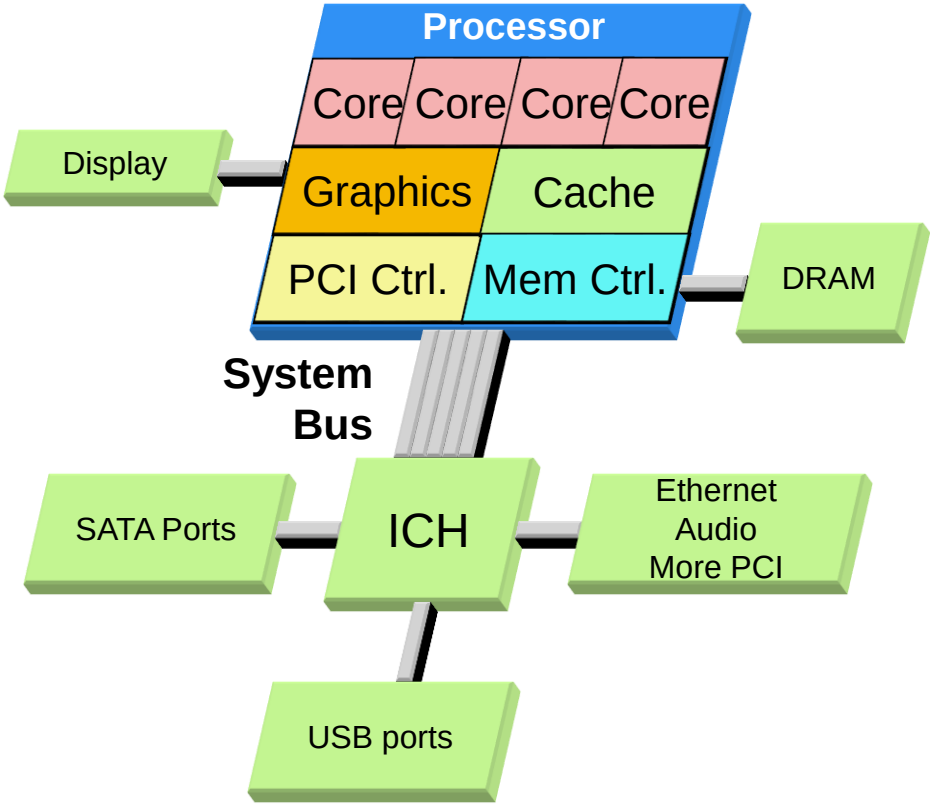


Let's have fun by understanding how a modern system even boots to an OS…

PC ARCHITECTURE AND BOOT PROCESS

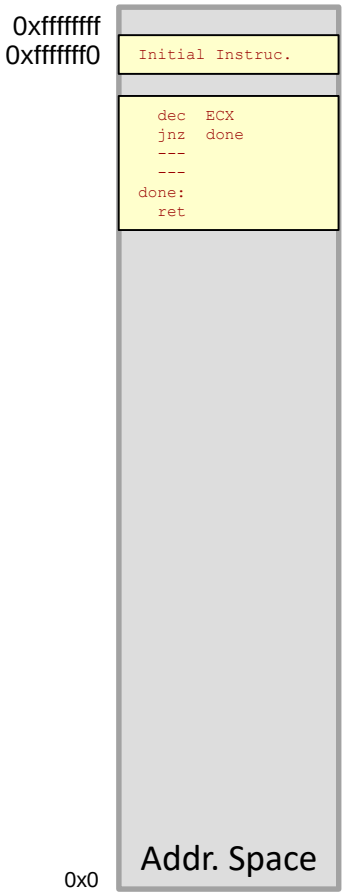
Modern PC Architecture

- Moore's Law has allowed greater integration levels
  - Multiple cores and greater levels of cache
  - Memory controller, graphics, and high-speed I/O are integrated onto the processor die



Intel Boot Process

- On power on, each core races for a lock bit
  - First one executes the boot sequence
  - Others wait for Start-Up Inter-Processor Interrupt
- On power on, fetch instruction at 0xFFFFFFF0
  - Address corresponds to ROM/Flash
  - Jump to initialization code (BIOS) elsewhere
- Bootstrap
  - Choose mode: Real, Flat protected, Segmented protected



https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Intel Boot Process

- Determine basic memory map and DRAM configuration (but DRAM still not functional)
- Enable Interrupt Controllers for Protected Mode operation
  - Setup data-structures and base address registers (BARs) needed to support interrupts (i.e. descriptor tables [IDT, GDT], and Task-State Segment [TSS])
- Questions
  - Where is code right now?
  - Can this code be written using functions?
- Configure cache to be used as RAM for stack and place it NEM (No Eviction Mode)

https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Intel Boot Process

- Initialize DRAM
  - Write to all memory and ensure ECC flags match data (either via BIOS or HW mechanism)
- Copy BIOS code into DRAM and take processor out of special cache mode (and flush cache)
- Initialize other cores (sending them an initial EIP/PC)
- Discover and initialize various I/O devices
  - Timers, Cache, PCI bus, SATA (hard drive access)
  - Determine address ranges (memory map)
- Load Master Boot Record from first sector of boot device
  - Points to where OS is located and how to load code into memory
  - Transfer is now transferred to the OS

https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf
http://www.drdobbs.com/parallel/booting-an-intel-architecture-system-par/232300699

Windows Boot Process

- OS Loader
  - Loads system drivers and kernel code
  - Reads initial system registry info
- OS Initialization
  - Kernel Init
    - Init kernel data structures & PnP manager
  - Session SMSSInit (SMSS = Session Manager)
    - Initializes registry, starts other devices/drivers, and start other processes
  - Winlogon Init
    - User logon screen, service control manager, and policy scripts
  - Explorer Init
    - DWM (Desktop Window Manager), shell
- Post Boot phase
  - Other services are started (tray icons, etc.)
- Other good reference:
  - http://www.cs.fsu.edu/~zwang/files/cop4610/Fall2016/windows.pdf

https://social.technet.microsoft.com/wiki/contents/articles/11341.the-windows-7-boot-process-sbsl.aspx