Recall: Four fundamental OS concepts

- Thread
  - Single unique execution context
  - Program Counter, Registers, Execution Flags, Stack
- Address Space with translation
  - Programs execute in an address space that is distinct from the memory space of the physical machine
- Process
  - An instance of an executing program is a process consisting of an address space and one or more threads of control
- Dual Mode operation/Protection
  - Only the "system" has the ability to access certain resources
  - The OS and the hardware are protected from user programs and user programs are isolated from one another by controlling the translation from program virtual addresses to machine physical addresses

Recall: 3 types of Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Interrupt
  - External asynchronous event triggers context switch
  - e.g., Timer, I/O device
  - Independent of user process
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …
- All 3 are an UNPROGRAMMED CONTROL TRANSFER
  - Where does it go?

How do we get the system target address of the “unprogrammed control transfer?”
Interrupt Vector

- Where else do you see this dispatch pattern?

interrupt number (i)

\text{intrpHandler}_i()\
\{\ldots\}

Address and properties of each interrupt handler

Simple B&B: User => Kernel

- How to return to system?

Simple B&B: Interrupt

- How to save registers and set up system stack?

Simple B&B: Save state

- How to save registers and set up system stack?
Process Control Block

(Assume single threaded processes for now)

- Kernel represents each process as a process control block (PCB)
  - Status (running, ready, blocked, …)
  - Registers, SP, … (when not running)
  - Process ID (PID), User, Executable, Priority, …
  - Execution time, …
  - Memory space, translation tables, …

- Kernel Scheduler maintains a data structure containing the PCBs

- Scheduling algorithm selects the next one to run

Recall: give the illusion of multiple processors?

- Assume a single processor. How do we provide the illusion of multiple processors?
  - Multiplex in time!
  - Multiple “virtual CPUs”
- Each virtual “CPU” needs a structure to hold, i.e., PCB:
  - Program Counter (PC), Stack Pointer (SP)
  - Registers (Integer, Floating point, others…?)
- How switch from one virtual CPU to the next?
  - Save PC, SP, and registers in current PCB
  - Load PC, SP, and registers from new PCB
- What triggers switch?
  - Timer, voluntary yield, I/O, other things
**Simultaneous MultiThreading/Hyperthreading**

- Hardware technique
  - Superscalar processors can execute multiple instructions that are independent
  - Hyperthreading duplicates register state to make a second "thread," allowing more instructions to run
- Can schedule each thread as if were separate CPU
  - But, sub-linear speedup!
- Original technique called “Simultaneous Multithreading”
  - SPARC, Pentium 4/Xeon (“Hyperthreading”), Power 5

---

**Scheduler**

```java
if (readyProcesses(PCBs)) {
    nextPCB = selectProcess(PCBs);
    run(nextPCB);
} else {
    run_idle_process();
}
```

- Scheduling: Mechanism for deciding which processes/threads receive the CPU
- Lots of different scheduling policies provide ...
  - Fairness or
  - Real-time guarantees or
  - Latency optimization or ...

---

**Putting it together: web server**

Client → Web Server

- Request
- Reply (retrieved by web server)
Putting it together: web server

1. network socket read
2. copy arriving packet (DMA)
3. kernel copy
4. parse request
5. file read
6. disk request
7. disk data (DMA)
8. kernel copy
9. format reply
10. network socket write
11. kernel copy from user buffer to network buffer
12. format outgoing packet and DMA

Recall: 3 types of Kernel Mode Transfer

- **Syscall**
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall ID and arguments in registers and execute syscall

- **Interrupt**
  - External asynchronous event triggers context switch
  - e.g., Timer, I/O device
  - Independent of user process

- **Trap or Exception**
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero, …

Recall: User/Kernel (Privileged) Mode

Implementing Safe Kernel Mode Transfers

- Important aspects:
  - Separate kernel stack
  - Controlled transfer into kernel (e.g., syscall table)

- Carefully constructed kernel code packs up the user process state and sets it aside
  - Details depend on the machine architecture

- Should be impossible for buggy or malicious user program to cause the kernel to corrupt itself
Need for Separate Kernel Stacks

- Kernel needs space to work
- Cannot put anything on the user stack (Why?)
- Two-stack model
  - OS thread has interrupt stack (located in kernel memory) plus User stack (located in user memory)
  - Syscall handler copies user args to kernel space before invoking specific function (e.g., open)
  - Interrupts (???)

Kernel System Call Handler

- Vector through well-defined syscall entry points!
  - Table mapping system call number to handler
- Locate arguments
  - In registers or on user (!) stack
- Copy arguments
  - From user memory into kernel memory
  - Protect kernel from malicious code evading checks
- Validate arguments
  - Protect kernel from errors in user code
- Copy results back
  - Into user memory

Hardware support: Interrupt Control

- Interrupt processing not visible to the user process:
  - Occurs between instructions, restarted transparently
  - No change to process state
  - What can be observed even with perfect interrupt processing?
- Interrupt Handler invoked with interrupts 'disabled'
  - Re-enabled upon completion
  - Non-blocking (run to completion, no waits)
  - Pack up in a queue and pass off to an OS thread for hard work
    » wake up an existing OS thread

- OS kernel may enable/disable interrupts
  - On x86: CLI (disable interrupts), STI (enable)
  - Atomic section when select next process/thread to run
  - Atomic return from interrupt or syscall
- HW may have multiple levels of interrupt
  - Mask off (disable) certain interrupts, eg., lower priority
  - Certain Non-Maskable-Interrupts (NMI)
    » e.g., kernel segmentation fault
Interrupt Controller

- Interrupts invoked with interrupt lines from devices
- Interrupt controller chooses interrupt request to honor
  - Mask enables/disables interrupts
  - Priority encoder picks highest enabled interrupt
  - Software Interrupt Set/Cleared by Software
  - Interrupt identity specified with ID line
- CPU can disable all interrupts with internal flag
- Non-Maskable Interrupt line (NMI) can’t be disabled

How do we take interrupts safely?

- **Interrupt vector**
  - Limited number of entry points into kernel
- **Kernel interrupt stack**
  - Handler works regardless of state of user code
- **Interrupt masking**
  - Handler is non-blocking
- **Atomic transfer of control**
  - “Single instruction”-like to change:
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode
- **Transparent restartable execution**
  - User program does not know interrupt occurred

Administrivia: Getting started

- **THIS** Friday (1/26) is early drop day! Very hard to drop afterwards…

- Get working on Homework 0 due on Monday by 11:59PM!
  - Get familiar with all the cs162 tools
  - Submit to autograder via **git**

- Participation: Attend section! Get to know your TA!

- Group sign up: now via autograder, then TA form (next week, after EDD)
  - Get finding groups of 4 people ASAP
  - Priority for same section; if cannot make this work, keep same TA