Interrupts

- Forcibly change normal flow of control
- Enters the kernel at a specific point; the kernel then figures out which *interrupt handler* should run
- Many different types of interrupts

Types of Interrupts

- Synchronous versus asynchronous
- Asynchronous
  - From external source, such as I/O device
  - Not related to instruction being executed
- Synchronous (also called *exceptions*)
  - Programming errors or requests for kernel intervention
  - *Faults* — correctable; offending instruction is retried
  - *Traps* — often for debugging; instruction isn't retried

Interrupts and Hardware

- I/O devices have (unique or shared) *Interrupt Request Lines* (IRQs)
- Complex mechanisms to pass IRQs to CPU
- Interrupts can have varying priorities
- PICs and APICs map IRQs to *interrupt vectors*, and pass the latter to the CPU
- Priority and load-balancing scheme used on multiprocessors

Interrupt Masking

- Two different types: global and per-IRQ
- Global — delays all interrupts
- Selective — individual IRQs can be masked selectively
- Selective masking is usually what's needed — interference most common from two interrupts of the same type

Dispatching Interrupts

- Each interrupt has to be handled by a special device- or trap-specific routine
- *Interrupt Descriptor Table* (IDT) has *gate descriptors* for each interrupt vector
- Hardware locates the proper gate descriptor for this interrupt vector, and locates the new context
- A new stack pointer, program counter, CPU and memory state, etc., are loaded
- Global interrupt mask set
- The old program counter, stack pointer, CPU and memory state, etc., are saved on the new stack
- The specific handler is invoked

Returning From an Interrupt

- Load old program counter, stack pointer, CPU and memory state, etc., from the interrupt handler's stack
- Branches back to previous program; no change should be noticeable
- Note: CPU state generally unmasks interrupts

Nested Interrupts

- What if a second interrupt occurs while an interrupt routine is excuting?
- Generally a good thing to permit that — is it possible?
- And why is it a good thing?

Maximum Parallelism

- You want to keep all I/O devices as busy as possible
- In general, an I/O interrupt represents the end of an operation; another request should be issued as soon as possible
- Most devices don't interfere with each others' data structures; there's no reason to block out other devices

Portability

- Which has a higher priority, a disk interrupt or a network interrupt?
- Different CPU architectures make different decisions
- By not assuming or enforcing any priority, Linux becomes more portable

Nested Interrupts

- As soon as possible, unmask the global interrupt
- As soon as reasonable, re-enable interrupts from that IRQ
- But that isn't always a great idea, since it could cause re-entry to the same handler
- IRQ-specific mask is not enabled during interrupt-handling

First-Level Interrupt Handler

- Often in assembler
- Perform minimal, common functions: saving registers, unmasking other interrupts
- Eventually, undoes that: restores registers, returns to previous context
- Most important: call proper second-level interrupt handler (C program)

Exception Handling

- Three broad categories: debugging, virtual memory, error
- We’re not going to discuss program trace or breakpoints in this class
- Virtual memory is a topic for later
- What about error exceptions?

Error Exceptions

- Most error exceptions — divide by zero, invalid operation, illegal memory reference, etc. — translate directly into signals
- This isn't a coincidence. . .
- The kernel's job is fairly simple: send the appropriate signal to the current process
- That will probably kill the process, but that's not the concern of the exception handler

Interrupt Handling Philosophy

- Do as little as possible in the interrupt handler,
- Defer non-critical actions till later
- Again — want to do as little as possible with IRQ interrupts masked
- *No process context available*

No Process Context

- Interrupts (as opposed to exceptions) are not associated with particular instructions
- They're also not associated with a given process
- The currently-running process, at the time of the interrupt, as no relationship whatsoever to that interrupt
- Interrupt handlers cannot refer to **`current`**
- Interrupt handlers cannot sleep!

Interrupt Stacks

- When an interrupt occurs, what stack is used?
- The *kernel stack* of the current process, whatever it is, is used
- (There's always some process running — the "idle" process, if nothing else)
- It's only 8K bytes — we'd better not have too-deep nesting of interrupts

Finding the Proper Interrupt Handler

- First differentiator is the interrupt vector
- On modern hardware, multiple I/O devices can share a single IRQ and hence interrupt vector
- Each device's *interrupt service routine* (ISR) for that IRQ is called; the determination of whether or not that device has interrupted is device-dependent

Allocating IRQs to Devices and Drivers

- IRQ assignment is hardware-dependent.
- Sometimes it's hardwired, sometimes it's set physically, sometimes it's programmable
- Linux device drivers request IRQs when the device is opened
- Note: especially useful for dynamically-loaded drivers, such as for USB or PCMCIA devices
- Two devices that aren't used at the same time can share an IRQ, even if the hardware doesn't support simultaneous sharing

Monitoring Interrupt Activity

- Linux has a pseudo-file system, **`/proc`**, for monitoring (and sometimes changing) kernel behavior
- Run

  ```
  cat /proc/interrupts
  ```

  to see what's going on

/proc/interrupts

```
$ cat /proc/interrupts
           CPU0
  0:  130066609          XT-PIC  timer
  2:          0          XT-PIC  cascade
  3:          0          XT-PIC  uhci_hcd
  5:          0          XT-PIC  uhci_hcd
  8:        436          XT-PIC  rtc
  9:    2431568          XT-PIC  acpi, libata, uhci_hcd, eth0
 10:          0          XT-PIC  ehci_hcd, uhci_hcd
 14:    1170240          XT-PIC  ide0
NMI:          0
ERR:          0
```

Columns: IRQ, count, interrupt controller, devices

Much More in /proc

```
$ cat /proc/pci
PCI devices found:
  Bus  0, device   0, function  0:
    Class 0600: PCI device 8086:2580 (rev 4).
  Bus  0, device   1, function  0:
    Class 0604: PCI device 8086:2581 (rev 4).
      IRQ 11.
      Master Capable.  No bursts.  Min Gnt=2.
  Bus  0, device   2, function  0:
    Class 0300: PCI device 8086:2582 (rev 4).
      IRQ 11.
      Non-prefetchable 32 bit memory at 0xdff00000 [0xdff7fff
      I/O at 0xe898 [0xe89f].

...
```

Soft Interrupts

- We don't want to do too much in regular interrupt handlers:
  - Interrupts are masked
  - We don't want the kernel stack to grow too much
- Instead, interrupt handlers schedule work to be performed later
- Three mechanisms: *softirqs*, *tasklets*, and *work queues*
- Softirqs are used to implement tasklets
- For all of these, requests are queued

Softirqs

- Specified at kernel compile time
- Limited number:

Priority	*Type*
0	High-priority tasklets
1	Timer interrupts
2	Network transmission
3	Network reception
4	SCSI disks
5	Regular tasklets

Running Softirqs

- Run at various points by the kernel
- Most important: after handling IRQs and after timer interrupts
- Softirq routines can be executed simultaneously on multiple CPUs:
  - Code must be re-entrant
  - Code must do its own locking as needed

Rescheduling Softirqs

- A softirq routine can reschedule itself
- This could starve user-level processes
- Softirq scheduler only runs a limited number of requests at a time
- The rest are executed by a kernel thread, which competes with user processes for CPU time

Tasklets

- Similar to softirqs
- Created and destroyed dynamically
- Individual tasklets are locked during execution; no problem about re-entrancy, and no need for locking by the code
- The preferred mechanism for most deferred activity

Work Queues

- Always run by kernel threads
- Softirqs and tasklets run in an interrupt context; work queues have a process context
- Because they have a process context, they can sleep
- However, they're kernel-only; there is no user mode associated with it

System Calls

- System calls are the way in which user programs request actions from the kernel
- Almost always, they represent controlled access to privileged operations
- If something can be done with reasonable efficiency purely at user level, it should not be a system call

Division of Labor

- When a C program writes **`open()`**, the compiled program is *not* issuing a system call directly
- There is a library subroutine named **`open()`**, generally in assembler; it issues the actual system call
- May need to convert from C calling conventions to kernel calling conventions

Entering the Kernel

- The kernel is entered via a *software interrupt*
- This interrupt is handled very much like I/O interrupts or exceptions
- A small assembler first-level interrupt handler calls the appropriate C code to process the system call

Passing Parameters

- Passing parameters to system calls is rather complex
- For ordinary C functions, parameters are passed on the stack
- Interrutps, including software interrupts, switch stacks; copying data between stacks is complex
- Parameters are always passed in registers
- The assembler stub pushes these onto the stack, to emulate the C interface at the kernel end

Rules #1–3 for System Calls

1. Check all parameters carefully
2. Check all parameters carefully
3. Check all parameters carefully

By the way, check all parameters carefully

Copying Data to and from User Space

- Some systems calls (i.e., **`write()`** and **`read()`**) pass a buffer address; data is to be copied to or from the kernel
- It's vital to check that the program only passes valid, legal, user-space addresses
- Users *must not* read or write kernel memory, or reference non-existent memory
- Great care is needed
- First check: make sure that address passed is lower than **`PAGE_OFFSET`**, i.e., not in the kernel

Page Faults and System Calls

- User memory may not exist, or may be paged out
- The virtual memory system will handle any page faults and copy in the page if necessary; this operation could block
- Operation can fail if memory doesn't exist, or if access type is wrong
- Always do such copies via standard subroutines, and check for error returns
- The page fault handler makes sure that kernel page faults come from that section of code
- Page faults from elsewhere in the kernel crash the system!

Adding a System Call

- Write the code
- If it's in a new file, add the filename to the appropriate Makefile
- Routines are generally named **sys**_*xxx*
- Add the syscall number to linux/syscalls.h
- Add the routine in the proper spot in syscall_table.
- Write the C linkage routine

A Reimplementation of getpid()

```
#include <asm/unistd.h>

asmlinkage long sys_mypid(void)
{
        return current->tgid;
}
```

Simple User Linkage

```
#define __NR_mypid  294
__syscall0(long, mypid)

int main() {
        printf("%d\n", mypid());
return 0;
}
```

__syscall0 is for a system call with no arguments; there are also **__syscall1**, **__syscall2**, . . . , **__syscall6**