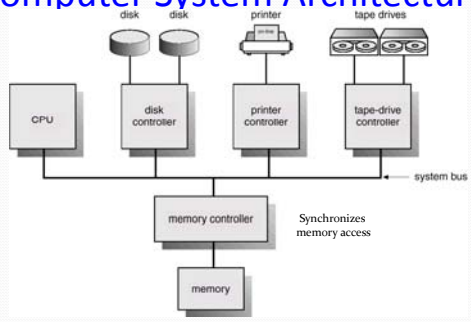
Architecture Review

Everything you were supposed to have learned in CS3610…

Computer System Architecture



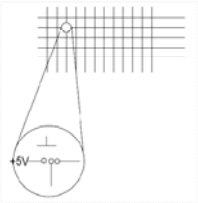
I/O operations

- I/O devices and the CPU can execute concurrently.
- I/O is moving data between device & controller's buffer
  - CPU moves data between controller's buffer & main memory
- Each device controller is in charge of certain device type.
  - May be more than one device per controller
    - SCSI can manage up to 7 devices
  - Each device controller has local buffer, special registers
- A device driver for every device controller
  - Knows details of the controller
  - Presents a uniform interface to the rest of OS

Accessing I/O Devices

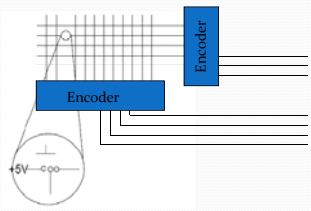
- Memory Mapped I/O
  - I/O devices appear as regular memory to CPU
  - Regular loads/stores used for accessing device
  - This is more commonly used
- Programmed I/O
  - Also called "channel" I/O
  - CPU has separate bus for I/O devices
  - Special instructions are required
- Which is better?

Building a Keyboard Controller



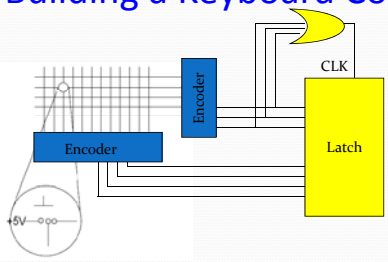
Mesh detects which key is pressed

Building a Keyboard Controller



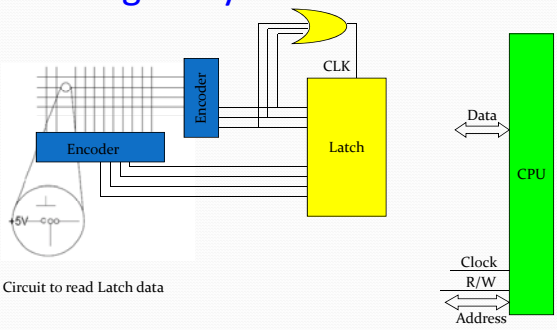
Encoders convert lines to binary

Building a Keyboard Controller



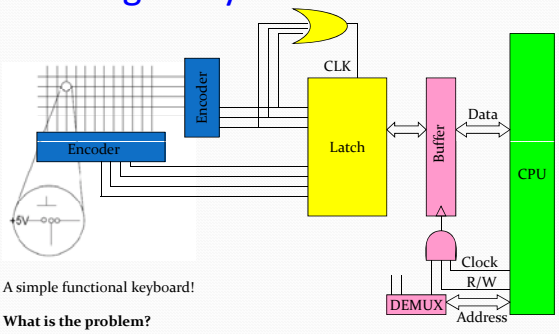
Latch stores encoding of pressed key

Building a Keyboard Controller



Circuit to read Latch data

Building a Keyboard Controller

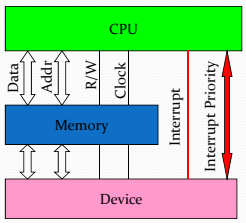


A simple functional keyboard!

What is the problem?

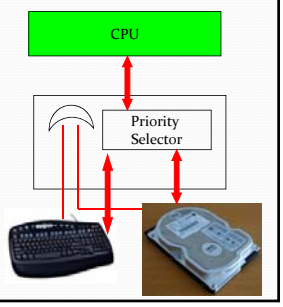
Interrupts

- Mechanism required for device to *interrupt* the CPU
  - Alternatively, CPU could poll. But this is inefficient
- Implementing interrupts
  - A line to interrupt CPU
  - Set of lines to specify priority



Handling Multiple Devices

- Interrupt controller
  - OR all interrupt lines
  - Highest priority passed to CPU
  - Can remap priority levels



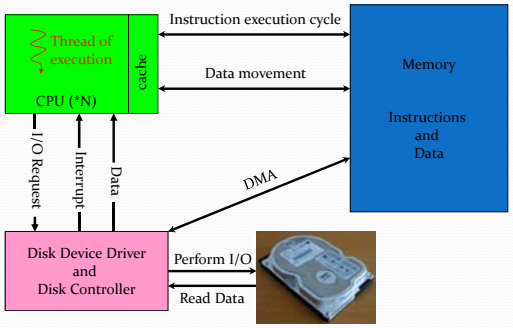
How does I/O work?

- Device driver loads controller registers appropriately
- Controller examines registers, executes I/O
- Controller signals I/O completion to device driver
  - Using interrupts
- High overhead for moving bulk data (i.e. disk I/O)

Direct Memory Access (DMA)

- Transfer data directly between device and memory
  - No CPU intervention
- Device controller transfers blocks of data
- Interrupts when block transfer completed
  - As compared to when byte is completed
- Very useful for high-speed I/O devices

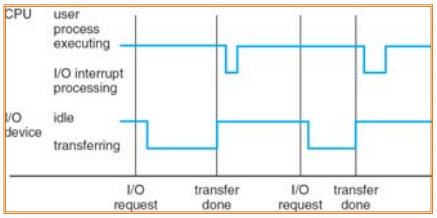
Example I/O



Interrupts (very important!)

- Notification from interface that device needs servicing
  - Hardware: sends trigger on bus
  - Software: uses a **system call**
- Steps followed on receiving an interrupt:
  - Stop kernel execution
  - Save machine context at interrupted instruction
  - Commonly, incoming interrupts are disabled
  - Transfer execution to Interrupt Service Routine (ISR)
    - Mapping done using the Interrupt Vector (faster)
  - After ISR, restore kernel state and resume execution
- Most operating systems are interrupt-driven

Interrupt Timeline



Traps and Exceptions

- Software generated interrupt
  - Exception: user program acts silly
    - Caused by an error (div by 0, or memory access violation)
    - Just a performance optimization
  - Trap: user program requires OS service
    - Caused by system calls
- Handled similar to hardware interrupts:
  - Stops executing the process
  - Calls handler subroutine
  - Restores state after servicing the trap

Why Protection?

- Application programs could:
  - Start scribbling into memory
  - Get into infinite loops
- Other users could be:
  - Gluttonous
  - Evil
  - Or just too numerous
- Correct operation of system should be guaranteed

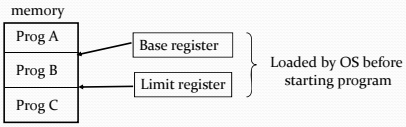
  $\Rightarrow$ A protection mechanism

Preventing Runaway Programs

- Also how to prevent against infinite loops
  - Set a timer to generate an interrupt in a given time
  - Before transferring to user, OS loads timer with time to interrupt
  - Operating system decrements counter until it reaches 0
  - The program is then interrupted and OS regains control.
- Ensures OS gets control of CPU
  - When erroneous programs get into infinite loop
  - Programs purposely continue to execute past time limit
- Setting this timer value is a privileged operation
  - Can only be done by OS

Protecting Memory

- Protect program from accessing other program's data
- Protect the OS from user programs
- Simplest scheme is base and limit registers:



- Virtual memory and segmentation are similar

Virtual Memory / Segmentation

- Here we have a table of base and limit values
- Application references some address x
  - CPU breaks this into a page number and an offset (or a segment number and offset)
  - (More common) Pages have fixed size, usually 512 bytes
  - (Now rare) Segments: variable size up to some maximum
  - Looks up the page table entry (PTE) for this reference
  - PTE tells the processor where to find the data in memory (by adding the offset to the base address in the PTE)
  - PTE can also specify other things such as protection status of page, which "end" it starts at if the page is only partially filled, etc.

Protected Instructions

Also called privileged instructions. Some examples:

- Direct user access to some hardware resources
  - Direct access to I/O devices like disks, printers, etc.
- Instructions that manipulate memory management state
  - page table pointers, TLB load, etc.
- Setting of special mode bits
- Halt instruction

Needed for:

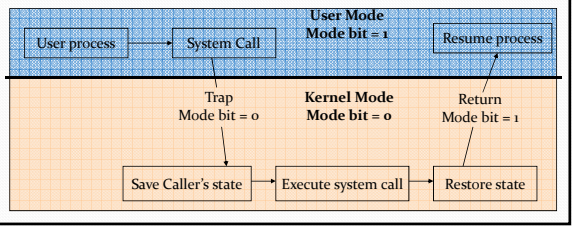
- Abstraction/ease of use and protection

Dual-Mode Operation

- Allows OS to protect itself and other system components
  - *User* mode and *kernel* mode
- OS runs in kernel mode, user programs in user mode
  - OS is god, the applications are peasants
  - Privileged instructions only executable in kernel mode
- Mode bit provided by hardware
  - Can distinguish if system is running user code or kernel code
  - System call changes mode to kernel
  - Return from call using RTI resets it to user
- How do user programs do something privileged?

Crossing Protection Boundaries

- User calls OS procedure for "privileged" operations
- Calling a kernel mode service from user mode program:
  - Using *System Calls* computer switches execution to kernel mode

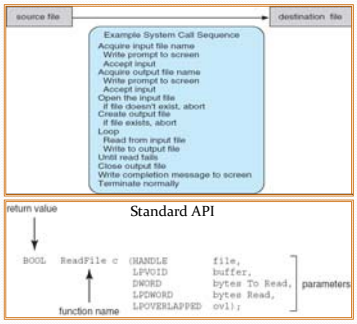


System Calls

- Programming interface to services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs using APIs
- Three most common APIs:
  - Win32 API for Windows
  - POSIX API for POSIX-based systems (UNIX, Linux, Mac OS X)
  - Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?
  - Easier to use

Why APIs?

System call sequence to copy contents of one file to another



Reducing System Call Overhead

- **Problem:** The user-kernel mode distinction poses a performance barrier
  - Crossing this hardware barrier is costly.
  - System calls take 10x-1000x more time than a procedure call
- Solution: Perform some system functionality in user mode
  - *Libraries (DLLs)* can reduce number of system calls,
    - by caching results (getpid) or
    - buffering ops (open/read/write vs. fopen/fread/ fwrite).

SideBar: Real System Have Holes

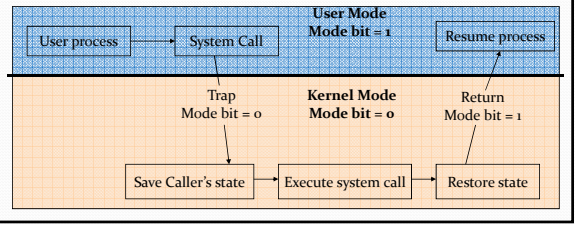
- OSes protect some things, ignore others
  - Most will blow up when you run:
  - Usual response: freeze
  - To unfreeze, reboot
  - If not, also try touching memory

```
int main() {
        while (1) {
                fork();
        }
}
```

- More broadly: O/S isn't able to defend itself against deliberate (or stupid) misuse, attack, etc
  - Goal: make it feasible to write applications that are good citizens
  - Promote effective resource sharing, correctness, security, high performance… but you need to understand the O/S to make effective use of its features

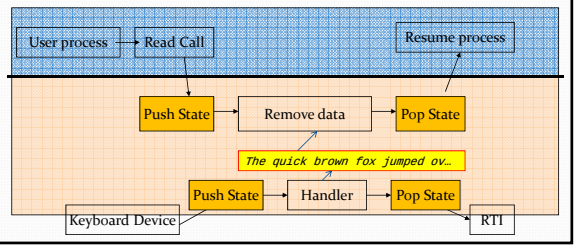
Back to Protection Boundaries

- User calls OS procedure for "privileged" operations
- Calling a kernel mode service from user mode program:
  - Using *System Calls* computer switches execution to kernel mode



Interrupts… same idea

- Device "asks" kernel to call an interrupt handler



How interrupts work

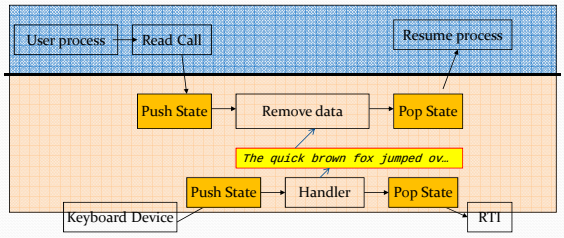
- Hardware requests an interrupt
- At next opportunity, processor pauses and triggers the same sequence we saw for system calls
  - Whatever was running has its state saved (kernel stack)
  - Switch to kernel mode if we weren't in it already
  - Kernel handler wakes up, finds device id
  - Table indexed by device-id gives address of handler
  - Handler looks just like any other function
- But notice that an interrupt can occur while we're already in the kernel executing a system call

Interrupts vs. Procedure Call

- Procedure call
  - Push args on stack
  - "Call" pushes PC, then jumps to function
  - Function saves registers it will use
  - *Execute function*
  - Pop (restore) registers
  - Pop return PC address, jumping to caller
- Interrupt, trap, etc
  - Pause normal execution
  - CPU pushes PC, enters kernel mode, jumps to a standard address
  - Save registers and decide what handler to call
  - *Procedure call to handler*
  - RTI/RTT pops (restore) registers and resumes prior execution

Making it real

- What issues make it tricky to write an interrupt handler or a kernel procedure?



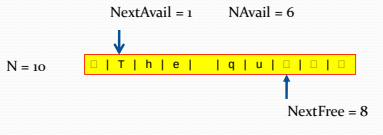
Implementing a queue

- How should we implement the data structure that stores the input characters?
  - O/S doesn't want to risk memory "exhaustion"
  - Best: a fixed maximum-size buffer
- So: the queue is a byte array:
  - byte[N] InputQueue = new byte[N];

A circular structure

- We'll have a pointer into the queue, pointing to the next free slot. It increments from 0..(N-1), then to 0
  - Int NextFree = 0;
  - NextFree = (NextFree+1)%N;
- Similarly for the next available character, NextAvail

A circular structure



N: Length of the InputQueue
NextAvail: "pointer" to the next available character, if NAvail > 0
NAvail: how many characters are currently available
NextFree: pointer to the next free slot, if Navail < N.

Code fragments

```
char ReadInput()
{
   ....
   char c = Buf[NextAvail];
   NextAvail = (NextAvail+1)%N;
   NAvail = NAvail-1;
   ....
   return c;
}
```

```
void Interrupt(char c)
{
   ....
   Buf[NextFree] = c;
   NextFree = (NextFree+1)%N;
   NAvail = Navail+1;
   ....
}
```

Not shown: "block" (make the application wait) if there are no characters when a Read is done. Don't restart the keyboard if there won't be room for the next input character.

A bug!

- User runs the program and types in 34 characters:

```
The quick brown fox jumped over the lazy dog
```

- Application receives 22, and some are wrong:

```
Th11zpjkj[1brow171jznz
```

- How can this be? The code was trivial!

Assembly code

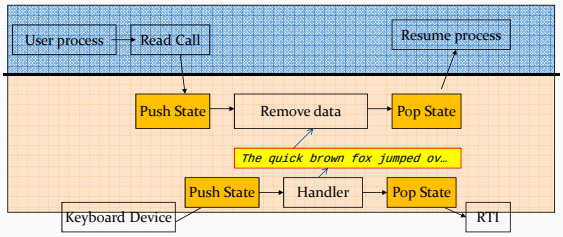
```
char ReadInput()
{
   ....
// NAvail = NAvail -1
   mov  NAvail, R1
   sub  $1,R1
   mov  R1,NAvail
   ....
}
```

```
void Interrupt(char c)
{
   ....
// NAvail = NAvail +1
   mov  NAvail, R1
   add  $1,R1
   mov  R1,NAvail
   ....
}
```

But when does "Interrupt" actually execute?

A bear walks into a bar. Where does it sit?

- An interrupt needs to occur. When does it happen?
  - *Whenever it likes*



Assembly code

```
char ReadInput()
{
   ....
// NAvail = NAvail -1
   mov  NAvail, R1
   sub  $1,R1
   mov  R1,NAvail
   ....
}
```

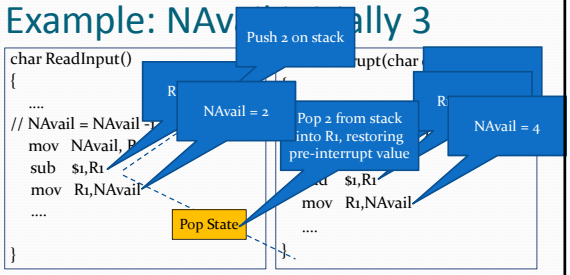
Push State

Pop State

```
void Interrupt(char c)
{
   ....
// NAvail = NAvail +1
   mov  NAvail, R1
   add  $1,R1
   mov  R1,NAvail
   ....
}
```

Push will save the value of R1 on the stack.
Pop restores R1 to its value prior to the interrupt

Example: NAv… ally 3



NAvail should be 3 (old value – 1 + 1)… but we leave it at 2!
A character will vanish from the input buffer

Effect of interrupts?

- NAvail will have the wrong value in it.
  - We can forget that we added a character
  - With other plausible code sequences we could forget that we removed characters
- So… reads could return the wrong number of characters and could read data from parts of the buffer that have junk in them!

An easy fix…

- Interrupt handler can disable interrupts
  - Done with a special hardware instruction accessed from a special kernel function (normal user code can't execute this instruction or call this kernel function – it isn't a system call)
  - While interrupts are disabled, hardware that needs service will be "waiting" and can hurt performance if used too often
- Notice the tension
  - Disabling interrupts makes it easier to write correct code
  - But we prefer not to disable them if possible, because doing so makes our O/S "less responsive" to I/O events
  - Forcing us to think hard: "when *must* I disable them?"

Hardware interrupt "levels"

- Rather than have all interrupts blocked, there are usually a few "levels" of interrupts
  - Each device has its own associated level
  - Illegal instructions, bad memory accesses: top priority
  - Then the system clock/timer
  - Next, disks and the network – high speed DMA devices
  - Then USB stuff: USB memories, keyboard, etc
- So if the keyboard driver disables interrupts, it probably only blocks keyboard interrupts – not others

This illustrates a bigger issue

- Part of the job of the operating system is to maximize concurrency
  - Having more things happening at once means that more work is getting done
  - Overlapping I/O with computation is just one example of concurrency… we'll see many others
- Allowing concurrency creates a risk that concurrent events will interfere with one-another (bugs)
- Can temporarily disallow concurrency, but this will slow the computer down (defeating the purpose)

Definition: A *race condition*

- We say that code contains a *race condition* if:
  - We are executing a fragment of code
  - Correctness of the outcome depends on whether or not some other concurrent event occurs before we finish the fragment
- Our example illustrated a race condition between the **ReadInput** system call and the **Interrupt** handler
  - Disabling interrupts during ReadInput would be one way of eliminating the race condition

Road Map

- Looking at the next few weeks
  - We'll be exploring ways of dealing with problems such as race conditions (there are several such problems)
  - They arise within the O/S, for example in interrupt handlers, but also in *multi-threaded applications*
- To understand these issues, we'll need to understand the "abstraction" of a program running in an O/S
  - First, explore the "process abstraction"
  - Then, revisit race conditions in this context
  - Learn to build safe, concurrent code