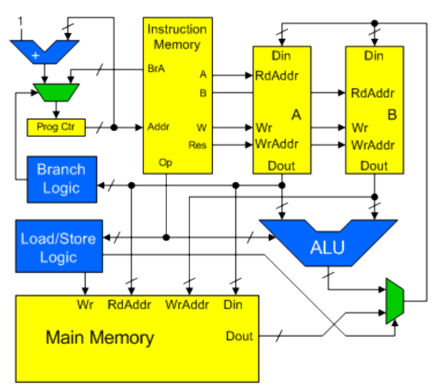
CS24: Introduction to Computation

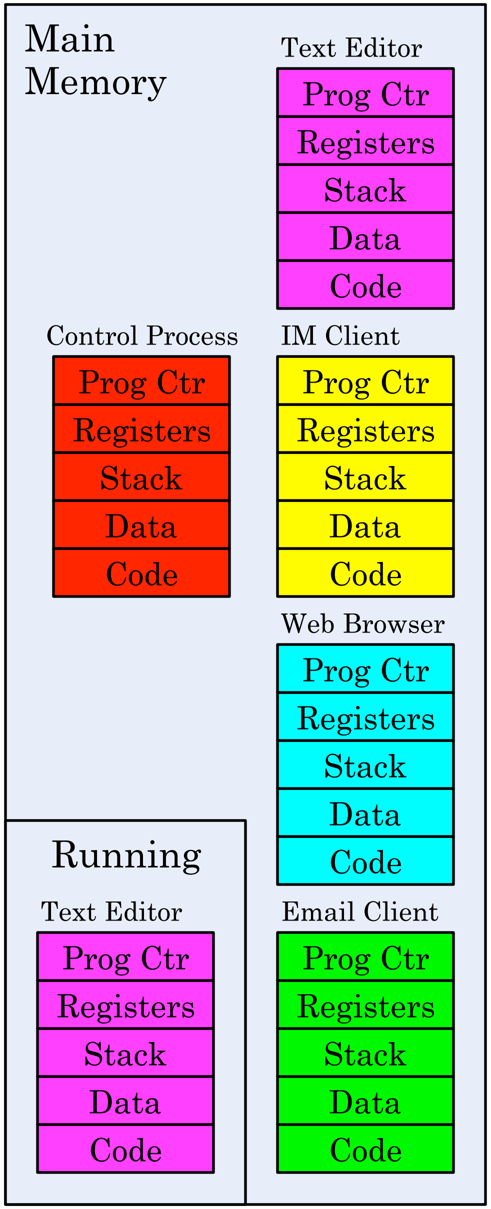
Spring 2015

Lecture 18

Last Time: Overview

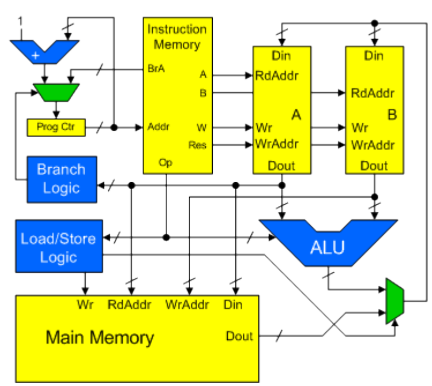
- Expanded on our process abstraction
- A special *control process* manages all other processes
  - Uses the same process abstraction as other processes, but it can access and manipulate everything else
- Controller performs context-switches
  - Suspend the currently running process, copy context to memory
  - Copy another process' context into "running" area, then resume it

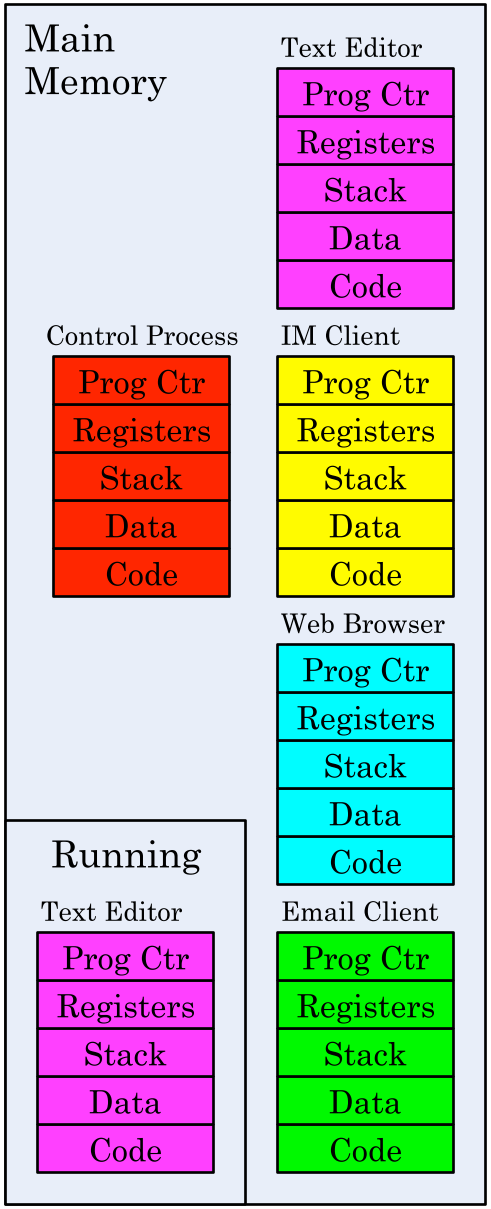




Last Time: Operating Modes

- Introduced processor *operating modes*
  - Enforce difference between “user” processes and the control process
- Applications run in “user mode” or “normal mode”
  - Can perform a limited subset of operations supported by the processor
- Control process runs in “kernel mode” or “protected mode”
  - Can use <u>all</u> processor capabilities
- Introduced the concept of an *operating system*
  - Intermediates between programs and hardware





Exceptional Control Flow

- Each process has a *logical control flow*
  - Sequence of instructions that the program executes
- *Exceptional control flow* is when logical flow is interrupted
  - Transfer control to an exception handler
- May or *may not* return to the interrupted program:

Interrupt	Signal from IO 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

- Enables several capabilities:
  - Periodically, interrupt the current process and let the controller context-switch to another process
  - If a process misbehaves, the control process can step in and handle the situation (e.g. terminate the process)

IA32 and Exceptional Control Flow

- IA32 supports 256 different kinds of exceptions
  - 0-31 are architecture-defined interrupts and exceptions
  - 32-255 are user-defined exceptions, typically used for operating system entry-points, hardware support
- When an exception occurs, handler is looked up in the Interrupt Descriptor Table
  - Stores address, privilege info for exception handler
  - Allows transfer from user-mode to kernel-mode code
- Can manually invoke an exception with **`int`** ***`n`***
  - e.g. **`int $0x80`** causes exception 0x80 (128) handler to be invoked
  - For *NIX operating systems on IA32, exception 0x80 is operating-system entry-point

IA32 Interrupt Operation

- IA32 **`int`** operation is similar to a **`call`**, but is *much* more involved
  - Processor saves return-address onto the stack
  - Processor also saves **`eflags`** register onto stack
    - Hardware interrupts also clear the Interrupt-Enable flag
- In IA32, each privilege level has its own stack
  - If changing privilege levels, must also change to a different stack
    - Info about caller's stack is also saved onto handler's stack
  - Reduces potential for stack-corruption attacks, and ensures system calls will have sufficient memory
- Saves all info we need to restore execution at the caller, but must do lots of extra work to return!
  - IA32 **`iret`** instruction performs inverse of these steps

Invoking Operating System Calls

- Operating system code runs at a different protection level than application code
- Use exceptional control flow to make system calls
  - Invoke a <u>trap</u> to perform the operation
- On IA32, UNIX operating system calls are available through **`int $0x80`** (interrupt 128)
- Very difficult to pass arguments to exception handler on the stack!
  - System uses a different stack than the application!
- Easy solution:
  - Pass arguments through general-purpose registers
  - (Limits the total number of arguments to system calls, but in practice this is not a problem.)

OPERATING SYSTEM CALLS (2)

- Specify the operation to perform in **`%eax`**
  - Numeric value indicates operation to perform
- Other registers specify arguments to system call
- List of system operations and their IDs:
  - MacOS X: **`/usr/include/sys/syscall.h`**
  - Linux: **`/usr/include/asm/unistd.h`**
  - **Note:** Many system-call IDs vary across platform! ☹
- Simple example: Get the current process ID
  - UNIX API call: **`int getpid()`**
  - Set **`eax`** = 20. No other arguments. Return-value in **`eax`**.
  - Assembly code:

```
movl $20, %eax
int  $0x80       # Invoke system call.
# Kernel stores process ID into eax.
```

Operating System Calls (3)

- Normally you don't invoke system calls this way!
  - C standard library provides very helpful wrapper functions for making system calls… use them! ☺
  - Also, C **`syscall(int number, ...)`** function does the hard work of invoking a system call for you

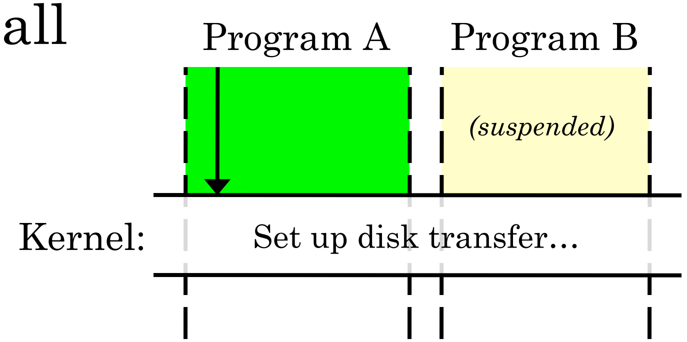
- Aside:
  - Invoking system calls via IA32 trap can be slow…
    - Pentium IV was *much* slower than Pentium III…
  - IA32 also has **`syscall`** and **`sysret`** instructions
    - Allows for *significantly faster* invocation of system calls
  - Linux 2.5+ supports both **`int $0x80`** and **`syscall`**
    - **`syscall`** can be ½ to ¼ time of equivalent **`int $0x80`** call!
  - Will revisit when we discuss virtual memory

Some UNIX System Calls

- File IO operations:
  - **`open()`**, **`close()`** – open or close a file
  - **`read()`**, **`write()`** – read or write data to a file
  - **`lseek()`** – set position of next read or write operation
- Directory operations:
  - **`mkdir()`** – create a directory
  - **`chdir()`** – change current directory
  - **`rmdir()`** – remove directory
  - **`link()`** – add a filename to an existing file
  - **`unlink()`** – remove filename from a file (possibly deleting)
- Process operations:
  - **`fork()`** – start a new child-process
  - **`execve()`** – start running a new program in this process
  - **`exit()`** – terminate this process
  - **`kill()`** – send a signal to another process
  - **`wait()`** – wait for a process to change state

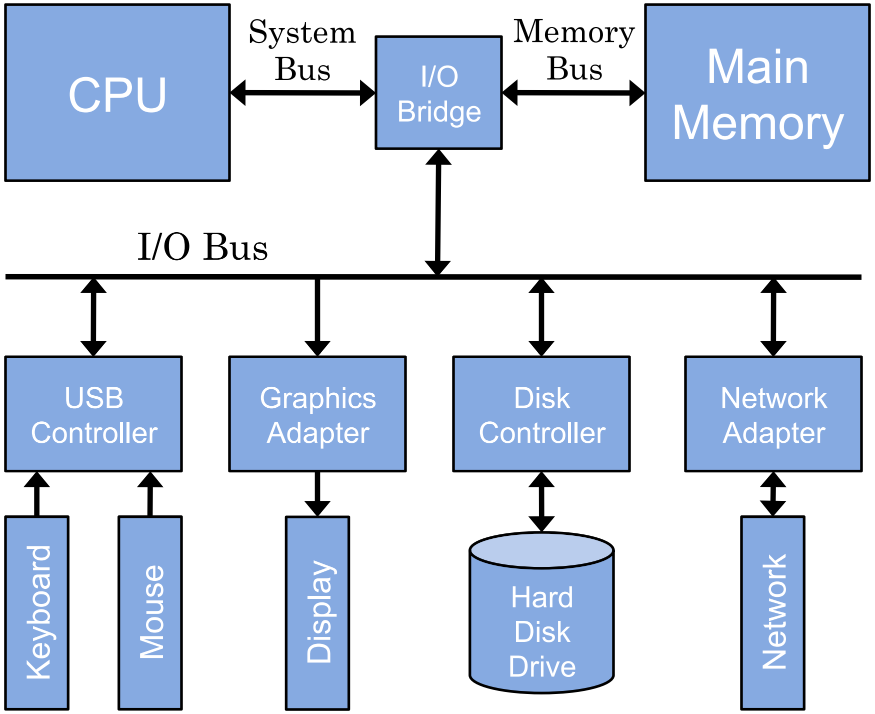
UNIX System Calls (2)

- Common theme across many system calls:
  - A significant number involve slow disk accesses
  - Some involve interacting with other processes
- The kernel frequently performs context-switches when system calls are made
- Example: two programs running concurrently
  - Program A executes a **`read()`** call
    - Read a block of data from disk
    - Will be waiting 8+ ms for data!
  - Program A transfers control to the kernel…
    - Kernel initiates the disk read, then goes to do other stuff in the meantime
    - *…but how do we "go do other stuff in the meantime"?*



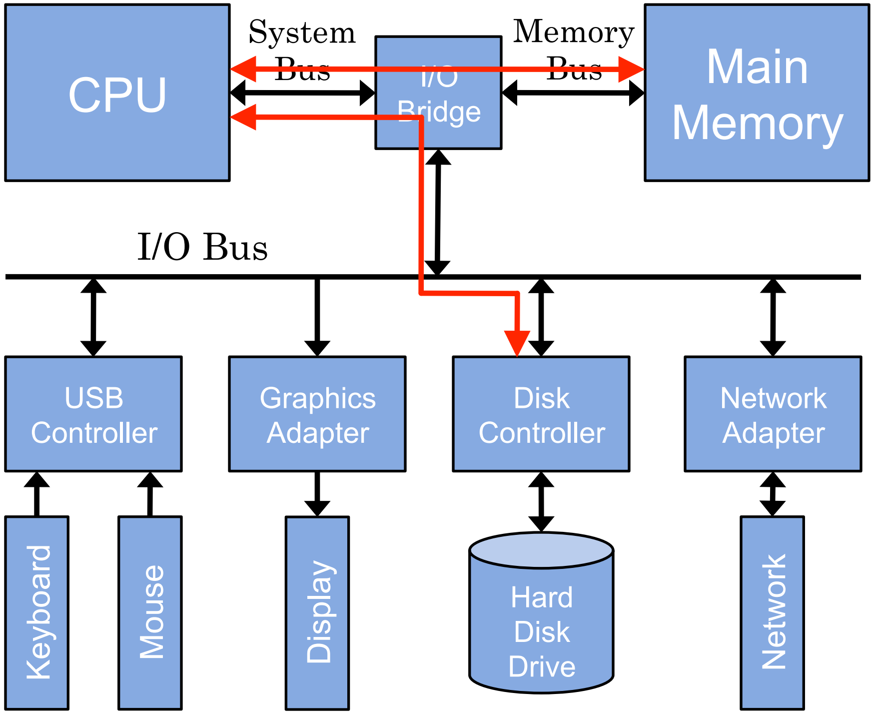
Computer System Buses

- Would like to be able to do other stuff while data is moved from disk into main memory
  - Don’t want the processor to be tied up by the disk access!
- Typical computer layout:
  - CPU, memory, and peripherals connected via an I/O Bridge
- System and memory buses are fast
- IO bus is slower, but supports a wide range of devices
  - e.g. PCI bus: “Peripheral Component Interconnect”



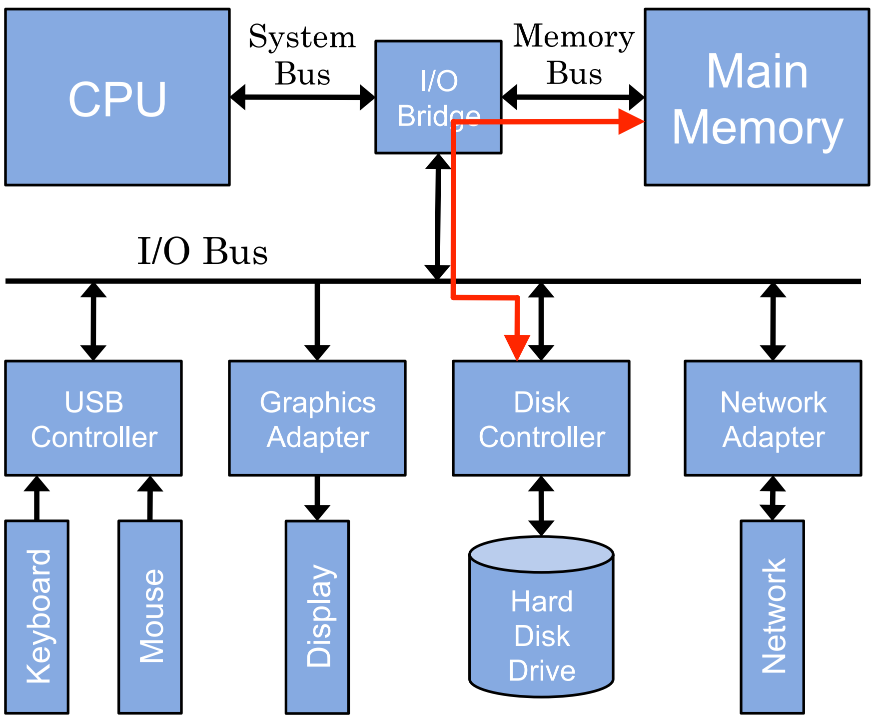
Computer System Buses (2)

- Normally, CPU interacts directly with main memory and with peripherals
- If CPU has to manually move each block of data from disk to memory…
- Can't do anything else while we are doing this!
  - Clearly need a better way to do this…



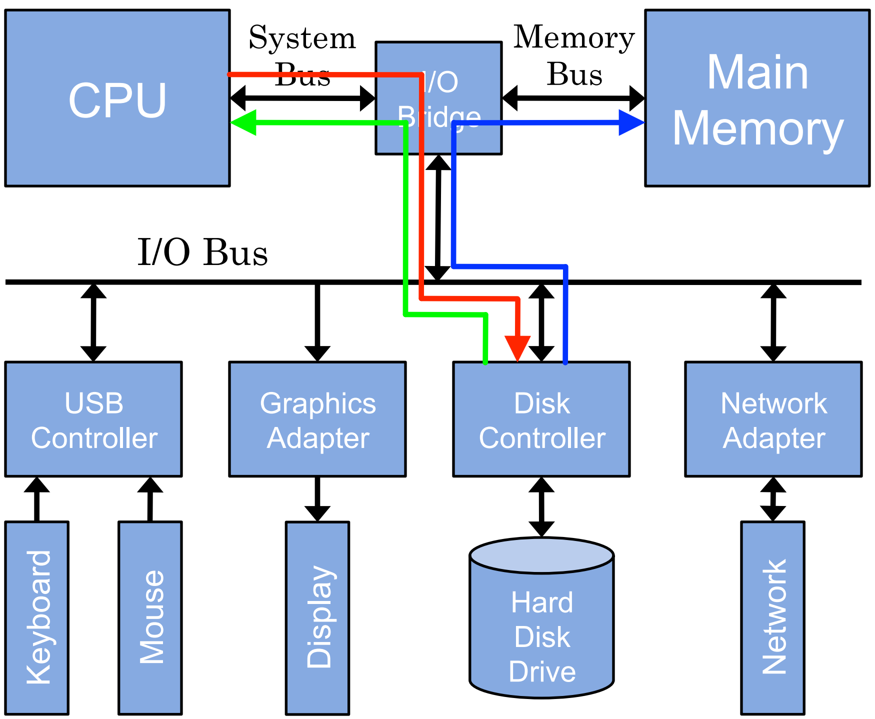
Direct Memory Access

- Some peripherals can also access memory directly
- Example:
  - CPU tells disk controller to transfer a disk block to main memory
  - Disk controller moves data to main memory on its own
- Called Direct Memory Access (DMA)
  - Interaction between disk controller and memory is called a <u>DMA transfer</u>
- Frees up CPU to do other things during transfer



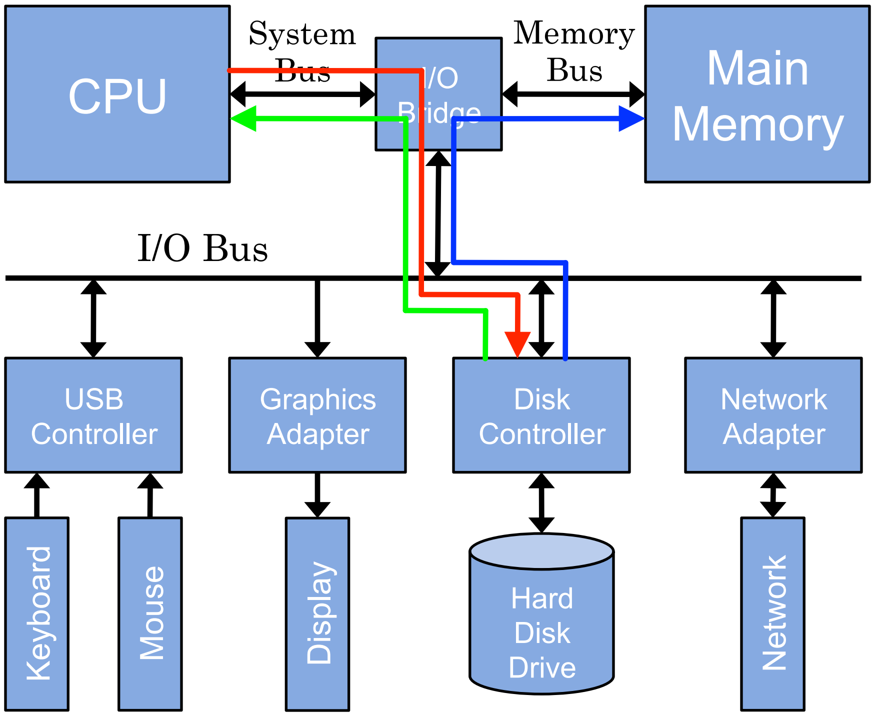
DMA Transfer Sequence

- Step 1: CPU tells disk controller to read a block of data into memory
- Step 2: Disk controller performs a DMA transfer into memory
- Step 3: Disk controller signals an interrupt to inform CPU that transfer is complete
- Result:
  - Operating system can do other work while slow operations take place!



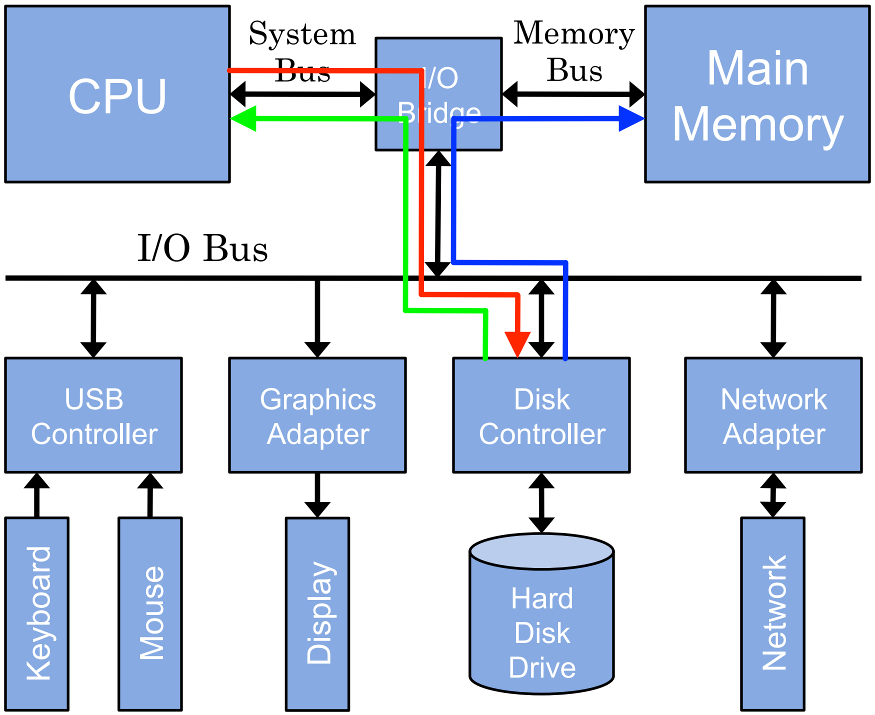
Direct Memory Access Notes

- Direct Memory Access is <u>essential</u> for modern high-performance computing!
  - Used by disk controllers, graphics cards, sound cards, networking cards, etc.
- Buses must support multiple "bus masters"
  - An arbiter must resolve concurrent requests from multiple bus masters
- While DMA transfers take place, CPU access to memory is slower
  - CPU will hopefully be using its caches…



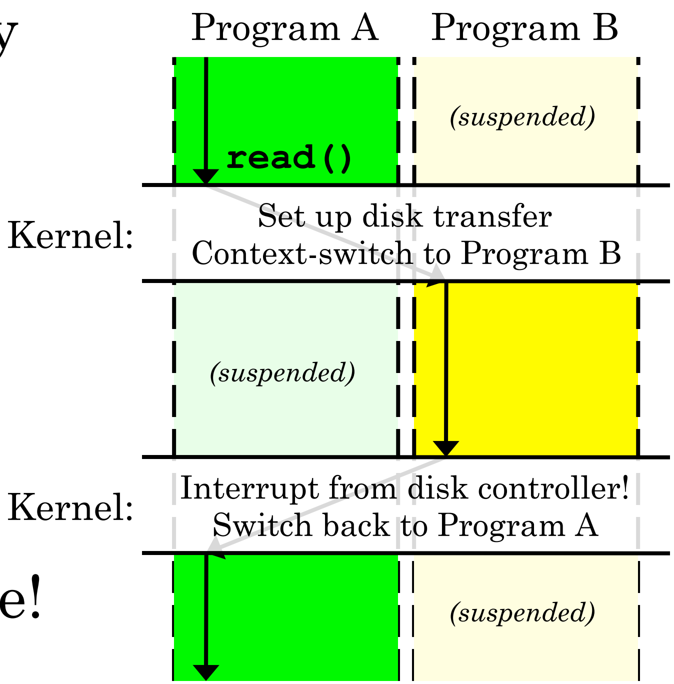
Direct Memory Access Notes (2)

- With DMA, cache coherence is a problem again…
  - CPU SRAM caches all live behind the I/O bridge
  - DMA transfers interact with DRAM main memory
  - Easy to have a cached block that a DMA transfer is modifying!
- Two solutions:
  - Allow external writes to invalidate cache lines
    - (And also, external reads must flush cache lines…)
  - Or, OS must carefully control access to memory used in DMA transfer



BACK TO OUR UNIX SYSTEM CALL…

- Two programs running concurrently
- Program A performs a **`read()`** call
  - Read a block of data from disk
  - Will be waiting 5+ ms for data!
- Prog A transfers control to kernel
  - Trap: **`int $0x80`**
  - Kernel initiates the disk read
  - Sets up a DMA transfer with the disk controller
- Program A can't progress for a while!
  - Kernel can context-switch to another process while Program A waits
- When DMA transfer completes, disk controller signals an interrupt to the processor
  - Processor handles the interrupt
  - Since Program A now has its data, context-switch back to A



Summary: Operating System Calls

- Operating system provides many useful facilities
  - Not just the process abstraction!
  - Interacts with disk hardware, networking hardware, etc.
- Applications use exceptions (specifically, traps) to transfer control to the operating system
  - Changes from user-mode to kernel-mode
- Using hardware features like Direct Memory Access:
  - Kernel can set up long-running tasks to run in background
  - When done, hardware signals the kernel via an interrupt
- Kernel can frequently use system calls as an opportunity to context-switch to other processes
  - Minimize time waiting for tasks to complete
- All of these steps depend on *exceptional flow control!*

User-Mode Exceptional Flow

- So far, exceptional control flow features have been usable only by the operating system
  - All exception handlers run in protected-mode
- Would like similar capabilities for user-mode code
- Already saw one set of functions that provide exceptional control flow:

  `int setjmp(jmp_buf env)`
    - Saves current execution context into **`env`**
    - Context includes **`%esp`** and caller's **`%eip`**, among other things

  `void longjmp(jmp_buf env, int val)`
    - Restores execution context from **`env`**
    - Causes execution to return to where **`setjmp()`** was called!
- Provides non-local (i.e. inter-procedure) goto
  - The only context not saved/restored is process memory…
  - Basically like an intra-process context switch!

User-Mode Exceptional Flow (2)

- With **`setjmp()/longjmp()`**, can implement a form of software exception-handling
  - e.g. C++/Java-style exceptions (lecture 11)
  - When exception is handled, can't return back to code that caused the exception!
- Other situations where exceptional flow control would be useful:
  - Let applications leverage the CPU's timer-interrupt support to provide timer events
  - Perform clean-shutdown operations when a program terminates (e.g. by Ctrl-C, seg-fault, or **`kill`** cmd)
  - Signal a user-mode server to reload its configuration, without having to stop and restart it

SIGNALS

- UNIX operating systems provide <u>signals</u>
  - A higher-level form of exception handling
  - Several hardware- and CPU-level exceptions are exposed to programs via this mechanism
- Some example signals, along with default behaviors:

ID	Name	Description	Default Action
1	SIGHUP	Terminal line hangup	Terminate
2	SIGINT	Keyboard interrupt (Ctrl-C)	Terminate
3	SIGQUIT	Quit from keyboard (Ctrl-\)	Terminate
4	SIGILL	Illegal instruction	Terminate
8	SIGFPE	Floating-point exception	Terminate + dump core
9	SIGKILL	Kill program	Terminate
11	SIGSEGV	Invalid memory access	Terminate + dump core
14	SIGALRM	Timer signal from alarm function	Terminate
10	SIGUSR1	User-defined signal 1	Terminate
12	SIGUSR2	User-defined signal 2	Terminate
20	SIGTSTP	Stop from keyboard (Ctrl-Z)	Suspend until SIGCONT received

Signals and the Kernel

- Many of these signals are received by the kernel!
  - SIGALRM – timer interrupt
  - SIGFPE – floating point exception, divide by zero
  - SIGILL – illegal instruction
  - SIGSEGV – illegal memory access
- Many others are routed through the kernel, if not originating from the kernel itself
  - e.g. SIGHUP (terminal hang-up), SIGINT (Ctrl-C keyboard interrupt), SIGTSTP (Ctrl-Z stop), SIGKILL (kill program)
- As with system calls, the kernel receives signals from hardware and other processes, on behalf of a process
  - Then, kernel forwards the signal to the appropriate process
- *The operating system is a mediator between the computer hardware and the application software.*

Registering Signal Handlers

- Use **`signal()`** to register a signal handler
  - A signal handler takes an integer argument, and returns nothing:

    ```
    typedef void (*handler_t)(int);
    ```

    - Argument is the signal type that was received
  - The **`signal()`** function is a system call (a trap) that registers a handler, and returns the old handler

    ```
    handler_t signal(int signum, handler_t handler)
    ```

  - Can also pass **`SIG_IGN`** to ignore the signal, or **`SIG_DFL`** to use the default handler
- <u>Cannot</u> handle or ignore the **`SIGKILL`** signal!
  - Always forcibly kills the receiving process
  - Used to handle runaway processes

Example Signal Handler

- A trivial example: a program that catches Ctrl-C

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

/* Handler for SIGINT, caused by Ctrl-C at keyboard. */
void handle_sigint(int sig) {
   printf("Caught signal %d\n", sig);
}

int main() {
   signal(SIGINT, handle_sigint);

   while (1) {
      /* System call to wait for a signal to arrive. */
      pause();
   }

   return 0;
}
```

Example Signal Handler (2)

- When run from console:
  - Start program `[user@host:~]> ./noint`
  - Press Ctrl-C **`Caught signal 2`**
  - Press Ctrl-C **`Caught signal 2`**
  - Press Ctrl-\ **`Quit`** *(output by system)*
  - `[user@host:~]>`

- Default signal handler for Ctrl-\ is still in place

Example Signal Handler (3)

- Program main loop:

```
while (1) {
    /* System call to wait for signal to arrive. */
    pause();
}
```

- Use **`pause()`** to keep from pegging the CPU ☺
  - Could leave it out – program will behave the same (although your CPU fan will probably turn on…)
- **Important note:**

```
void handle_sigint(int sig) {
    printf("Caught signal %d\n", sig);
}
```

  - A signal interrupts normal program execution
  - When the signal handler returns, execution resumes exactly where the program was interrupted

A More Complex Example

- This program prints a message every second:

```
/* Print a message, then request another SIGALRM. */
void handle_sigalrm(int sig) {
   printf("Hello!\n");
   alarm(1);  /* Request another SIGALRM in 1 second. */
}

/* User typed Ctrl-C.  Taunt them. */
void handle_sigint(int sig) {
   printf("Ha ha, can't kill me!\n");
}

int main() {
   signal(SIGINT, handle_sigint);
   signal(SIGALRM, handle_sigalrm);
   alarm(1);  /* Request a SIGALRM in 1 second. */

   while (1) pause();  /* Gentle infinite loop. */

   return 0;
}
```

A More Complex Example (2)

- Now we have a more interesting issue!

```
/* Print a message, then request another SIGALRM. */
void handle_sigalrm(int sig) {
   printf("Hello!\n");
   alarm(1);  /* Request another SIGALRM in 1 second. */
}

/* User typed Ctrl-C.  Taunt them. */
void handle_sigint(int sig) {
   printf("Ha ha, can't kill me!\n");
}
```

- Could easily have a situation where one handler is processing its signal, and the other signal occurs!
  - One **printf()** call can interrupt the other **printf()** call
- **printf()** has global state: standard output…

Reentrant Functions

- A signal handler can interrupt *any other code* in the program
  - *…including function calls that are in progress!*
- Signal handlers must only use <u>reentrant functions</u>
  - Functions that can be invoked *multiple times concurrently*, without causing errors
  - Frequently, code in a signal handler will interrupt code in the main program that is using the exact same functions
- Example: **`malloc()`** is <u>not</u> reentrant!
  - Updates large, complex data structures within the heap
  - Two calls to **`malloc()`** can easily stomp on each other!
  - Must not use **`malloc()`** within a signal handler!
    (Or, any other function that calls **`malloc()`**!)

Next Time

- Continue discussion of UNIX signal handlers
- Begin covering the UNIX process model