**CS 61C Lecture Notes Week 8**

Topic: Input/output

**Reading:** Patterson & Hennessy, Chapter 8 and Appendix A.7-8

Early computers had special instructions for input and output, with names like `datai` (data in) and `datao` (data out). These instructions transferred information through wires directly connecting the processor with each i/o device.

In most computer designs today, the i/o device controllers are attached to the same system bus that connects the processor to the memory. In effect, each device controller pretends to be a very small memory.

For example, we'll be studying a simple keyboard controller, called a *receiver*, that has two registers: a data register, holding the most recently typed character, and a control register, with two flag bits whose purpose we'll see shortly. Each of these registers acts like a word of memory; in this case, the control register is at address `0xffff0000` and the data register is at `0xffff0004`. Other devices have different addresses.

This means that the operating system software can interact with the device using ordinary `lw` and `sw` instructions with these specific addresses. The advantage of this approach over the old way is software simplicity, and also that the same i/o devices continue to work on newer computer models as long as the bus design is unchanged.

Before Unix, all operating systems were written in assembly language and only worked for one computer architecture. The use of ordinary load and store instructions for i/o devices was one thing that made it possible to write an operating system in C. Here's how you read the receiver data register:

```
int *rcvData;
int char;

rcvData = (int *)0xffff0004;
char = *rcvData;
```

It's a convention that all i/o device registers are full words (as determined by the bus width), even though, in this case, only one byte of the register has useful information. You should always read and write device registers with `lw` and `sw`, not with `lb` or `sb`. In particular, since the MIPS can be either little-endian or big-endian, you don't know whether the byte containing the actual character is at byte address `0xffff0004` or at `0xffff0007`. But loading the entire word will work either way.

Although the device registers pretend to be words of memory, they don't behave like real memory. The value in a word of memory doesn't change unless the processor changes it with a store instruction. But the value in a device register changes whenever something happens at the device; in the case of the receiver data register, it changes whenever the user types a key on the keyboard. We express this by saying that the device registers are *volatile*. The C language has a `volatile` keyword that can be used to tell the compiler that some word might change value unexpectedly, so the optimizer can't eliminate redundant load instructions. So I really should have said

```
(volatile int) *rcvData;
```

**Timing is everything**

The fundamental fact about i/o is that it's slow. The processor mustn't try to read data from an input device until there is some data to read; it mustn't try to write data to an output device until the device has finished dealing with the previous chunk of data.

Therefore, we need some way for the i/o device to tell the processor when it's ready. There are two main possibilities: *polling* and *interrupts*. We'll talk about polling first, and interrupts later.

The idea of polling is that when the processor wants to send or receive data from a device, it executes a loop essentially like this:

```
while (device-not-ready)
   ;
```

Polling is wasteful of processor time; ideally the processor should be doing other work while waiting for the device. Most i/o is done using interrupts, whereby the device can grab the processor's attention when it's ready. But polling is still useful for certain purposes, such as following the mouse in a drawing program.

How does the processor check whether the device is ready? That's what the control register is for. In the receiver, the rightmost bit of the control register is the "ready bit"; the device turns that bit on when the user types a character and the character is ready to read in the data register. The device turns the ready bit off when the processor actually reads the character. So we can write a polling loop like this:

```
loop:   lw   $8, 0xffff0000
        andi $8, $8, 1         # turn off all but ready bit
        beqz $8, loop          # not ready yet
        lw   $8, 0xffff0004    # ready, get the character
        sb   $8, somewhere
```

**Bus masters and slaves**

The receiver controller we've been describing is an entirely passive device; the processor has to do all the work of pulling characters out of it. This is appropriate for a slow device, and the receiver is especially slow because it is limited by the speed of a human typist.

But that approach wouldn't make sense for a mass storage device such as a disk. If the processor had to read each byte of a disk file explicitly, it wouldn't have time to do anything else. So the disk controller is more complicated. It still has registers that pretend to be memory, but the controller can also pretend to be a processor, making requests to transfer information to and from memory itself.

So if the program wants to read a block from the disk into a memory buffer, the processor stores the buffer's address, the location on the disk of the desired information, and the length of the block in disk controller registers. It then stores a Read request into another register. At that point, the disk controller reads the data from the disk, and stores it directly into the desired memory addresses without any more help from the main processor. Only when the entire transfer is finished does the controller indicate that it's Ready for another request.

A passive device like the keyboard receiver is called a *bus slave*; an active one like the disk controller is called a *bus master*.

**Device details**

The rest of what there is to know about i/o has to do with the details of particular devices. We'll only deal practically with two in this course: the receiver, which we've already discussed, and the transmitter, which displays characters on the screen. It, too, has a control register and a data register. Of course in this case the program stores characters into the data register, instead of loading characters from it. Also, the Ready bit in the control register is set to 1 when the device is ready for another character; it's set to 0 when the processor stores into the data register, and stays 0 until that stored character is on the screen.

The transmitter control register is at `0xffff0008` and its data register is at `0xffff000c`.

The receiver and transmitter we've described are appropriate for a computer with a single keyboard, a single screen, and character display only. Real computers differ in one of two ways: Small computers have screens that are controlled by the computer dot by dot, rather than character by character. For these screens, there is a block of pseudo-memory into which the processor can store information. For a black-and-white screen there is one bit per dot. (The official name for a dot is a *pixel*, for "picture element.") For a color screen, each pixel may require 24 bits, to specify 256 intensity levels for each of three colors (red, green, and blue).

Larger computers may have many terminals attached, not just one. In these systems there will be a terminal *multiplexor* that treats several terminals as one device. Whenever a user types a key at any of the attached terminals, the multiplexor turns on its Ready bit. The data register, in this case, will contain both the character and a number specifying which keyboard generated the character. Similarly, character output can be multiplexed if the processor stores a character and a terminal number in the transmitter data register.

**Interrupts**

Until now we talked about i/o through polling. This technique is simple, but it requires the processor to waste time checking and re-checking the device controller's ready bit. We'd like the processor to be able to request service from a device, and then go do something else until the device is ready. This happens by means of interrupts.

An interrupt is essentially an *involuntary procedure call.* That is, an interrupt temporarily diverts the computer's attention from one procedure to another, like a procedure call, but this diversion happens at an unexpected moment, not in response to a `jal` instruction in the program.

Because an interrupt is like a procedure call, it has many of the same needs. It must save and restore the program counter; it must use temporary registers; it may require a stack frame; and it must have something like arguments and return values to communicate with its client.

But the involuntary, *asynchronous* (happening unexpectedly) nature of the interrupt causes some differences from ordinary procedure calls. Remember that much of the operation of a procedure call depends on conventions; the caller and the callee agree about things like who is allowed to use which registers. Since the system is not expecting an interrupt at any particular moment, the interrupt procedure can't change the value of *any* registers that the caller might be using. There is no caller, so the interrupt procedure can't expect arguments in registers, and can't return a value in a register. (A particularly tricky aspect of this problem is that the interrupt routine can't change the value in `$1`, the register that the assembler uses to implement some MAL instructions in terms of multiple machine instructions. It's easy to get this wrong, because you don't realize you're using that register.)

These problems are solved by a combination of special hardware and special software conventions. For example, there is no `jal` instruction to indicate where the interrupt procedure is, and the machine can't use `$31` to hold the return address. (What if an interrupt happens just after a `jal` instruction? The interrupt's return address would overwrite the procedure call's return address.) So when a device interrupts, the machine stores the old PC (the return address) in a special register called the EPC (Exception PC). (Interrupts are a kind of *exception*; the other kind is processor errors like dividing by zero.) The processor then starts executing instructions from location `0x80000080`. In other words, all exceptions are handled by a single procedure, part of the operating system; that procedure must figure out the particular kind of exception and jump to a device-specific procedure.

Since the interrupt procedure can't assume that even the temporary registers `$8`–`$15` are available, how can it get started? The MIPS convention is that registers 26 and 27 are reserved for use *only* by exception handlers. If those two registers aren't enough, then the interrupt procedure must save and restore any others that it wants to use.

**Coprocessors**

The MIPS design allows for optional circuitry to be connected to the main processor. The most common example is the floating point arithmetic processor, which is optional on some MIPS models. (It's another integrated circuit mounted next to the main processor on the main circuit board of the machine.) There can be up to four coprocessors, numbered 0 through 3. Each coprocessor has its own set of 32 registers.

But coprocessor 0 is not really separate or optional at all; it's part of the main processor, and it helps in handling interrupts. The EPC, for example, is register 14 of coprocessor 0. We'll see what some of the other registers are for in a moment.

The MIPS has two instructions to transfer information between the central processor and a coprocessor. They're called `mfc` (Move From Coprocessor) and `mtc` (Move To Coprocessor). Here's an example:

```
mfc0  $26, $14
```

This instruction says to copy the information in coprocessor 0 (that's the 0 in the instruction name) register 14 (in other words, the EPC) into general register 26. To copy information from the central processor register to the coprocessor register, you'd say

```
mtc0  $26, $14
```

The general register number comes first, whether it's the source or the destination.

There are three coprocessor 0 registers used in handling exceptions. You already know about the EPC, `$14`. Another one is the *cause* register, `$13`. When an exception happens, the hardware puts a number in this register that indicates what kind of exception it is. Interrupts from i/o devices are one kind of exception; since we won't be handling any other kind, we can ignore the cause register.

Finally, a very important register is the *status* register for the processor, which is coprocessor 0 register `$12`. This register has several uses, but the most important right now is that it includes an *interrupt enable* bit to allow or disallow interrupts.

The reason this is so important is that when an interrupt happens, we'd better be sure that another one, from some other device perhaps, doesn't happen right away. If that happened, the EPC would be set to an instruction in the interrupt handler itself, and the return address in the user program would be lost. Therefore, when the processor gets an interrupt, it immediately turns off the IE bit. (We'll see shortly how it remembers the old value of the bit.)

**User and kernel modes**

On toy computers like Windows PCs and pre-OS-X Macs, an error in an application program often causes the entire system to crash. This is unacceptable in a multi-user system. The system must also protect users' privacy by restricting access to disk files, etc. What this means is that the operating system has to be able to do things that ordinary user programs can't do.

The MIPS operates in one of two modes, *user* mode and *kernel* mode. In user mode, there are restrictions on what a program can do. In particular, there are two restrictions that are important for our purposes. First, user-mode programs can only refer to memory addresses in the first half of the address space, i.e., less than `0x80000000`. This protects the operating system itself from the user, since the operating system text and data are in the top half of the address space, but it also prevents user-mode programs from controlling i/o devices, since the device registers are all in the top half of the address space. Second, the operating system can specify which coprocessors may be used by user-mode programs. Users aren't simply prevented from using *all* coprocessors because then they couldn't take advantage of floating point hardware, which is in coprocessor 1. But the operating system will prevent users from manipulating the registers in coprocessor 0.

**Status bits**

The mode is determined by another bit in the status register, `$12` in coprocessor 0. Here are the relevant bit numbers. (Bit 0 is the rightmost bit; bit 31 is the leftmost bit.)

```
 0   Interrupt Enable  (1 if enabled)
 1   User Mode  (1 if user mode)
 2   Previous IE
 3   Previous User
 4   Old IE
 5   Old User

 8   Enable software interrupt level 0
 9   Enable software interrupt level 1
10   Enable software interrupt level 2
11   Enable hardware interrupt level 0
12   Enable hardware interrupt level 1
13   Enable hardware interrupt level 2
14   Enable hardware interrupt level 3
15   Enable hardware interrupt level 4
```

The rightmost six bits are a sort of stack for status bits, with three sets of values. Whenever an exception happens, the hardware copies the Previous values to the Old bits, and copies the current values to the Previous bits. (In other words, bits 0–3 are shifted left two positions into bits 2–5.) The current bits, the two rightmost ones, are both set to zero. This means that the exception handler runs in kernel mode, with interrupts disabled.

(If interrupts are disabled, why is this status stack needed? How can there be a need to remember two old sets of values? One answer is that not all exceptions are interrupts; the exceptions generated by the processor itself are always enabled. Another answer is that the interrupt handler may choose to re-enable interrupts once it has saved the necessary hardware status into memory.)

When the operating system is finished handling an interrupt, it can restore the previous state of these status bits by executing the instruction `rfe` (Return From Exception). This instruction uses no operands; all it does is shift bits 2–5 of the status word rightward into bits 0–3. This is the last thing that the interrupt handler does before returning to the interrupted program.

Bits 8–15 concern the fact that some interrupts are more urgent than others, because some devices are faster than others. During the processing of an interrupt from a slow device, we want to be able to handle interrupts from faster devices, which need immediate attention. But a slow device shouldn't interrupt the handling of an interrupt from a fast device. Therefore, interrupt handlers can selectively enable or disable particular *kinds* of interrupts. We won't use this feature in the MIPS labs; we'll just make sure that receiver and transmitter interrupt levels are always allowed.

**System calls**

In the MIPS project you'll do regarding interrupts, the simulated machine will always be in kernel mode. But in a real system, the situation is a little more complicated.

Let's say your program wants to print some text. You probably say something like

```
printf("Some text.\n");
```

`Printf` is a C library procedure that knows how to convert numbers to printable form and things like that. But ultimately the `printf` code will say

```
write(1, "Some text.\n", 11);
```

The 1 indicates the "standard output," which generally means your screen. The 11 is the length of the string you're printing. Unlike `printf`, `write` is a *system call.* It works by asking the operating system to do some work on your behalf. In particular, the `write` procedure in the C library executes a `syscall` machine instruction. This instruction causes an exception, so the operating system starts running. Before doing the `syscall`, the `write` procedure puts the number that identifies this particular system call in `$2`, so the kernel can figure out what you want it to do. (The arguments to `write` remain in `$4` and so on, which the kernel can read.)

In effect, a system call is a procedure call to the kernel.

The kernel saves the characters you want to print. If the transmitter device is ready, the kernel can send it the first character. But it doesn't wait until that one is completed in order to send the next one! Instead, it remembers all your characters in a *buffer* (a block of memory allocated for this purpose) and immediately returns to your program.

When the transmitter is ready, it interrupts. The interrupt handler knows where the transmitter buffer is, so it can extract the next character and send it to the transmitter data register to be printed.

It's important to understand that at the moment of the interrupt, your program may not be running! Some other user process might be running at that moment. In fact, your program may not even be in memory at all. (We'll see later how memory is managed.) That's why the system call copies your entire string into a buffer in the kernel's memory; the interrupt routine can't assume that your own copy of the string is available.

The buffer we've been describing serves as an implicit argument to the interrupt routine; as mentioned earlier, that routine can't have explicit arguments because it doesn't have an explicit caller.

Although system calls and interrupts are both exceptions, they have quite different characteristics. The system call is a *voluntary* procedure call to the kernel; the user program can put arguments in registers and collect a return value from a register just as it could for an ordinary procedure call. But the interrupt is involuntary and asynchronous; it can't make any assumptions about registers.

One specific example of this general point is that an interrupt handler shouldn't assume that `$29` points to available stack space. It's entirely possible that the current user process has used up all its stack space. (Remember that the interrupt is probably on behalf of an entirely different process.) In the interrupt project in this course you can use the stack, but a real system has to be more careful.

**Ring buffers**

So far you may have the impression that the `write` system call fills up a buffer, and the interrupt handler empties it, and then maybe another `write` fills it up again. In reality, it's quite possible for a second `write` to happen while some characters are still remaining from the first one.

So the buffer has to be a *queue*; you should remember this abstract data type from 61A. The operations on a queue are to add a new datum at the end, to check whether the queue is empty, and to remove the datum at the front if it's nonempty. We could implement a queue using linked nodes, as in 61A, but in practice a simpler approach is taken. We decide ahead of time the maximum number of characters that we'll allow in the queue, and preallocate a fixed array of that size. (If a user program asks to `write` into the buffer when it's full, that user process has to wait until space is available. In the input direction, if users type so fast that the input buffer fills up before anyone reads the input characters, then some of what the user types is just lost.)

This block of memory is treated as if it were a circle. That is, after a character is inserted into the last byte of the array, the next character goes into the first byte of the array.

The system maintains two pointers into the array, one that points to the first free byte (used to insert into the queue) and one that points to the first occupied byte (used to remove from the queue). For an output buffer, the pointer to the first free byte is used by the `write` system call handler to copy your characters in, while the pointer to the first occupied byte is used by the transmitter interrupt handler to find the next character to print.

If the buffer is empty, the two pointers point to the same byte. If you think about it, you'll see that the two pointers will also point to the same byte if the buffer is *full.* To avoid this possible confusion, we declare that there must always be at least one unused byte in the array. That is, the capacity of the queue is one less than the size of the array. The queue is full if the insertion pointer is one less than the removal pointer.

**Enabling device interrupts**

In addition to the systemwide interrupt enable bit (which is used to avoid the problem of overlapping interrupts losing information) and the interrupt level enable bits (used to implement a priority system for interrupts), there is an interrupt enable bit for every i/o device. In the devices we're using (the transmitter and receiver), the IE bit is bit 1 of the device status register (the next to the rightmost bit). The device status register has only two useful bits, the Ready bit, which we've already discussed, and the IE bit.

The Ready bit is set by the device hardware and read by the operating system code. The IE bit is the other way around; the OS code sets it, and the device reads it. (When you `sw` into the device status register, only the IE bit matters; the others can have any value you want.)

If both the Ready bit and the IE bit are on, the device causes an interrupt. (The interrupt is delayed if either the systemwide IE bit or the interrupt level bit for this device's level is off.) The device keeps sending the interrupt signal until either the device becomes not ready (because the OS reads or writes the data register) or the OS turns off the IE bit.

An input device, such as the keyboard receiver, is Ready only if there is some input data ready that the OS hasn't yet read. Ordinarily (when the user hasn't typed anything recently) the device is not ready, and so it's okay to leave IE on all the time.

But an output device, such as the display transmitter, is Ready whenever it's not in the middle of processing output data (in this case, printing a character). So most of the time it's ready. Therefore, the OS must turn the IE bit off most of the time, and turn it on only when it wants to print something. When the system turns IE on, the device will immediately interrupt, and its interrupt handler should send the first character to the device.

**A Handwavy Kernel Code Example**

To make this all a little more concrete, here's some simplified code to run the transmitter device. First we need the device registers and a ring buffer:

```
(volatile int) *XmtStatus = ((volatile int)*)0xffff0008;
(volatile int) *XmtData = ((volatile int)*)0xffff000e;

#define XmtBufSize 1000
char XmtBuf[XmtBufSize];
char *XmtPut = XmtBuf;
char *XmtTake = XmtBuf;

int XmtBufEmpty() {
    return XmtPut == XmtTake;
}

int XmtBufFull() {
    return (XmtTake == XmtPut+1) ||
           (XmtTake == XmtBuf && XmtPut == XmtBuf+XmtBufSize-1);
}
```

In real life we'd define a RingBuffer data type that would include all the pieces as fields, because we'd have lots of devices, each of which would need one.

Here's the procedure to implement a simple system call, by which the user program would ask the kernel to print a single character. (In real systems, since system calls are expensive, we'd allow a single syscall invocation to print a block of characters all at once.)

```
void putchar(char ch) {
    *XmtStatus = 0;        /* enter critical section, disable device */
    while (XmtBufFull()) {
        Sleep(XmtBuf);        /* mark process as waiting */
        *XmtStatus = 2;        /* end critical section */
        Reschedule();        /* let another process run */
    }
    *XmtPut++ = ch;        /* char into buffer */
    if (XmtPut == XmtBuf+XmtBufSize)        /* implement ring */
        XmtPut = XmtBuf;
    *XmtStatus = 2;        /* turn on transmitter */
}
```

The procedures `Sleep()` and `Reschedule()` aren't part of the I/O code, but part of the scheduler, another piece of the kernel that decides which user program to run next. At any time, each user program is either *runnable* or *waiting*; the latter means that it can't run until some condition is satisfied. The `Sleep()` procedure marks the current process as waiting; its argument is used as an arbitrary code to specify the condition that must be satisfied in order to make the process runnable again. `Reschedule()` tells the scheduler to save the state of the currently running program, and pick the next runnable program, restoring its state. That other program then starts running; the call to `Reschedule()` in *this* program doesn't return until someone else calls `Reschedule()`, this program is runnable, and it's this program's turn to run.

This system call handler and the transmitter interrupt handler, which we'll see next, are run as two separate processes that share data, namely the ring buffer and the two pointers into it. So there is a concern about synchronization of the two processes. The calls to `XmtBufFull()` and `Sleep()` are the critical section. Instead of having a mutex to acquire and release, we can take advantage of the fact that one of the competing procedures is an interrupt handler. The system call procedure can disable the device's ability to interrupt temporarily, thereby assuring that it can complete its critical section without interference.

```
void xmtInt() {
    if (XmtBufEmpty()) {
        *XmtStatus = 0;                /* nothing to print, turn device off */
        return;
    }
    *XmtData = (int)(*XmtTake++);        /* print a character! */
    if (XmtTake == XmtBuf+XmtBufSize)        /* implement ring */
        XmtTake = XmtBuf;
    Wake(XmtBuf);                /* allow any waiting user prog to run */
}
```

There's no mention of a critical section here because if the interrupt routine is running, user code isn't running, so there's no danger of a `putchar` system call happening. If the buffer is empty we disable transmitter interrupts, but that's not to mark the beginning of a critical section. It turns off the transmitter and keeps it off, until a user program sends us more characters to print. Then `putchar()` will turn the device back on, to enable printing.

Once we've printed a character, there's at least one empty space in the ring buffer, so we tell the scheduler to look for a process that's waiting for this buffer and mark it runnable. In practice, we wouldn't really do this because rescheduling the process for every character would cause a lot of overhead time to be wasted in the scheduler. We'd defer the `Wake()` call until the buffer is, say, half empty, and then the user program can generate a bunch of output before being blocked again.

Running the keyboard receiver is similar, in that there are two separate procedures, one to handle a `getchar` system call and the other to handle a receiver interrupt. But the roles of the two procedures are swapped, because now it's the interrupt handler that puts characters into the buffer, and the system call handler that takes characters out. Also, as mentioned earlier, we want an idle receiver to be enabled, although an idle transmitter should be disabled.