Another service provided by operating system is dealing properly with the attempt to execute instructions with "illegal" opcodes.

Illegal is quotes because that just means opcodes whose operations aren't implemented directly by the hardware.

As we'll see, it's possible extend the functionality of the hardware via software emulation.

The action of the CPU upon encountering an illegal instruction (sometimes referred to as an unimplemented user operation or UUO) is very similar to how it processes interrupts.

Think of illegal instructions as an interrupt caused directly by the CPU!

As for interrupts, the execution of the current instruction is suspended and the control signals are set to values to capture PC+4 in the XP register and set the PC to, in this case, 0x80000004.

Note that bit 31 of the new PC, aka the supervisor bit, is set to 1, meaning that the OS handler will have access to the kernel-mode context.

Here's some code similar to that found in the Tiny Operating System (TinyOS), which you'll be experimenting with in the final lab assignment.

Let's do a quick walk-through of the code executed when an illegal instruction is executed.

Starting at location 0, we see the branches to the handlers for the various interrupts and exceptions.

In the case of an illegal instruction, the BR(I_IllOp) in location 4 will be executed.

Immediately following is where the OS data structures are allocated.

This includes space for the OS stack, UserMState where user-mode register values are stored during interrupts, and the process table, providing long-term storage for the complete state of each process while another process is executing.

When writing in assembly language, it's convenient to define macros for operations that are used repeatedly.

We can use a macro call whenever we want to perform the action and the assembler will insert the body of the macro in place of the macro call, performing a lexical substitution of the macro's arguments.

Here's a macro for a two-instruction sequence that extracts a particular field of bits from a 32-bit value.

M is the bit number of the left-most bit, N is the bit number of the right-most bit.

Bits are numbered 0 through 31, where bit 31 is the most-significant bit, i.e., the one at the left end of the 32-bit binary value.

And here are some macros that expand into instruction sequences that save and restore the CPU registers to or from the UserMState temporary storage area.

With those macros in hand, let's see how illegal opcodes are handled.

Like all interrupt handlers, the first action is to save the user-mode registers in the temporary storage area and initialize the OS stack.

Next, we fetch the illegal instruction from the user-mode program.

Note that the saved PC+4 value is a virtual address in the context of the interrupted program.

So we'll need to use the MMU routines to compute the correct physical address - more about this on the next slide.

Then we'll use the opcode of the illegal instruction as an index into a table of subroutine addresses, one for each of the 64 possible opcodes.

Once we have the address of the handler for this particular illegal opcode, we JMP there to deal with the situation.

Selecting a destination from a table of addresses is called "dispatching" and the table is called the "dispatch table".

If the dispatch table contains many different entries, dispatching is much more efficient in time and space than a long series of compares and branches.

In this case, the table is indicating that the handler for most illegal opcodes is the UUOError routine, so it might have smaller and faster simply to test for the two illegal opcodes the OS is going to emulate.

Illegal opcode 1 will be used to implement procedure calls from user-mode to the OS, which we call supervisor calls.

More on this in the next segment.

As an example of having the OS emulate an instruction, we'll use illegal opcode 2 as the opcode for the SWAPREG instruction, which we'll discuss now.

But first just a quick look at how the OS converts user-mode virtual addresses into physical addresses it can use.

We'll build on the MMU VtoP procedure, described in the previous lecture.

This procedure expects as its arguments the virtual page number and offset fields of the virtual address, so, following our convention for passing arguments to C procedures, these are pushed onto the stack in reverse order.

The corresponding physical address is returned in R0.

We can then use the calculated physical address to read the desired location from physical memory.

Okay, back to dealing with illegal opcodes.

Here's the handler for opcodes that are truly illegal.

In this case the OS uses various kernel routines to print out a helpful error message on the user's console, then crashes the system!

You may have seen these "blue screens of death" if you run the Windows operating system, full of cryptic hex numbers.

Actually, this wouldn't be the best approach to handling an illegal opcode in a user's program.

In a real operating system, it would be better to save the state of the process in a special debugging file historically referred to as a "core dump" and then terminate this particular process, perhaps printing a short error message on the user's console to let them know what happened.

Then later the user could start a debugging program to examine the dump file to see where their bug is.

Finally, here's the handler that will emulate the actions of the SWAPREG instruction, after which program execution will resume as if the instruction had been implemented in hardware.

SWAPREG is an instruction that swaps the values in the two specified registers.

To define a new instruction, we'd first have to let the assembler know to convert the swapreg(ra,rc) assembly language statement into binary.

In this case we'll use a binary format similar to the ADDC instruction, but setting the unused literal field to 0.

The encoding for the RA and RC registers occur in their usual fields and the opcode field is set to 2.

Emulation is surprisingly simple.

First we extract the RA and RC fields from the binary for the swapreg instruction and convert those values into the appropriate byte offsets for accessing the temporary array of saved register values.

Then we use RA and RC offsets to access the user-mode register values that have been saved in UserMState.

We'll make the appropriate interchange, leaving the updated register values in UserMState, where they'll be loaded into the CPU registers upon returning from the illegal instruction interrupt handler.

Finally, we'll branch to the kernel code that restores the process state and resumes execution.

We'll see this code in the next segment.