Let's turn our attention to how the operating system (OS) deals with input/output devices.

There are actually two parts to the discussion.

First, we'll talk about how the OS interacts with the devices themselves.

This will involve a combination of interrupt handlers and kernel buffers.

Then we'll discuss how supervisor calls access the kernel buffers in response to requests from user-mode processes.

As we'll see, this can get a bit tricky when the OS cannot complete the request at the time the SVC was executed.

Here's the plan!

When the user types a key on the keyboard, the keyboard triggers an interrupt request to the CPU.

The interrupt suspends execution of the currently-running process and executes the handler whose job it is to deal with this particular I/O event.

In this case, the keyboard handler reads the character from the keyboard and saves it in a kernel buffer associated with the process that has been chosen to receive incoming keystrokes.

In the language of OSes, we'd say that process has the keyboard focus.

This transfer takes just a handful of instructions and when the handler exits, we resume running the interrupted process.

Assuming the interrupt request is serviced promptly, the CPU can easily keep up with the arrival of typed characters.

Humans are pretty slow compared to the rate of executing instructions!

But the buffer in the kernel can hold only so many characters before it fills up.

What happens then?

Well, there are a couple of choices.

Overwriting characters received earlier doesn't make much sense: why keep later characters if the earlier ones have been discarded.

Better that the CPU discard any characters received after the buffer was full, but it should give some indication that it's doing so.

And, in fact, many systems beep at the user to signal that the character they've just typed is being ignored.

At some later time, a user-mode program executes a ReadKey() supervisor call, requesting that the OS return the next character in R0.

In the OS, the ReadKey SVC handler grabs the next character from the buffer, places it in the user's R0, and resumes execution at the instruction following the SVC.

There are few tricky bits we need to figure out.

The ReadKey() SVC is what we call a "blocking I/O" request, i.e., the program assumes that when the SVC returns, the next character is in R0.

If there isn't (yet) a character to be returned, execution should be "blocked", i.e., suspended, until such time that a character is available.

Many OSes also provide for non-blocking I/O requests, which always return immediately with both a status flag and a result.

The program can check the status flag to see if there was a character and do the right thing if there wasn't, e.g., reissue the request at a later time.

Note that the user-mode program didn't have any direct interaction with the keyboard, i.e., it's not constantly polling the device to see if there's a keystroke to be processed.

Instead, we're using an "event-driven" approach, where the device signals the OS, via an interrupt, when it needs attention.

This is an elegant separation of responsibilities.

Imagine how cumbersome it would be if every program had to check constantly to see if there were pending I/O operations.

Our event-driven organization provides for on-demand servicing of devices, but doesn't devote CPU resources to the I/O subsystem until there's actually work to be done.

The interrupt-driven OS interactions with I/O devices are completely transparent to user programs.

Here's sketch of what the OS keyboard handler code might actually look like.

Depending on the hardware, the CPU might access device status and data using special I/O instructions in the ISA.

For example, in the simulated Beta used for lab assignments, there's a RDCHAR() instruction for reading keyboard characters and a CLICK() instruction for reading the coordinates of a mouse click.

Another common approach is to use "memory-mapped I/O", where a portion of the kernel address space is devoted to servicing I/O devices.

In this scheme, ordinary LD and ST store instructions are used to access specific addresses, which the CPU recognizes as accesses to the keyboard or mouse device interfaces.

This is the scheme shown in the code here.

The C data structure represents the two I/O locations devoted to the keyboard: one for status and one for the actual keyboard data.

The keyboard interrupt handler reads the keystroke data from the keyboard and places the character into the next location in the circular character buffer in the kernel.

In real life keyboard processing is usually a bit more complicated.

What one actually reads from a keyboard is a key number and a flag indicating whether the event is a key press or a key release.

Knowing the keyboard layout, the OS translates the key number into the appropriate ASCII character, dealing with complications like holding down the shift key or control key to indicate a capital character or a control character.

And certain combination of keystrokes, e.g., CTRL-ALT-DEL on a Windows system, are interpreted as special user commands to start running particular applications like the Task Manager.

Many OSes let the user specify whether they want "raw" keyboard input (i.e., the key numbers and status) or "digested" input (i.e., ASCII characters).

Whew!

Who knew that processing keystrokes could be so complicated!

Next, we'll figure out how to code the associated supervisor call that lets user programs read characters.