When a user-mode program wants to read a typed character it executes a ReadKey() SVC.

The binary representation of the SVC has an illegal value in the opcode field, so the CPU hardware causes an exception, which starts executing the illegal opcode handler in the OS.

The OS handler recognizes the illegal opcode value as being an SVC and uses the low-order bits of the SVC instruction to determine which sub-handler to call.

Here's our first draft for the ReadKey sub-handler, this time written in C.

The handler starts by looking at the process table entry for the current process to determine which keyboard buffer holds the characters for the process.

Let's assume for the moment the buffer is *not* empty and skip to the last line, which reads the character from the buffer and uses it to replace the saved value for the user's R0 in the array holding the saved register values.

When the handler exits, the OS will reload the saved registers and resume execution of the user-mode program with the just-read character in R0.

Now let's figure what to do when the keyboard buffer is empty.

The code shown here simply loops until the buffer is no longer empty.

The theory is that eventually the user will type a character, causing an interrupt, which will run the keyboard interrupt handler discussed in the previous section, which will store a new character into the buffer.

This all sounds good until we remember that the SVC handler is running with the supervisor bit (PC[31]) set to 1, disabling interrupts.

Oops!

Since the keyboard interrupt will never happen, the while loop shown here is actually an infinite loop.

So if the user-mode program tries to read a character from an empty buffer, the system will appear to hang, not responding to any external inputs since interrupts are disabled.

Time to reach for the power switch :) We'll fix the looping problem by adding code to subtract 4 from the saved value of the XP register before returning.

How does this fix the problem?

Recall that when the SVC illegal instruction exception happened, the CPU stored the PC+4 value of the illegal instruction in the user's XP register.

When the handler exits, the OS will resume execution of the user-mode program by reloading the registers and then executing a JMP(XP), which would normally then execute the instruction *following* the SVC instruction.

By subtracting 4 from the saved XP value, it will be the SVC itself that gets re-executed.

That, of course, means we'll go through the same set of steps again, repeating the cycle until the keyboard buffer is no longer empty.

It's just a more complicated loop!

But with a crucial difference: one of the instructions - the ReadKey() SVC - is executed in user-mode with PC[31] = 0.

So during that cycle, if there's a pending interrupt from the keyboard, the device interrupt will supersede the execution of the ReadKey() and the keyboard buffer will be filled.

When the keyboard interrupt handler finishes, the ReadKey() SVC will be executed again, this time finding that the buffer is no longer empty.

Yah!

So this version of the handler actually works, with one small caveat.

If the buffer is empty, the user-mode program will continually re-execute the complicated user-mode/kernel-mode loop until the timer interrupt eventually transfers control to the next process.

This seems pretty inefficient.

Once we've checked and found the buffer empty, it would be better to give other processes a chance to run before we try again.

This problem is easy to fix!

We'll just add a call to Scheduler() right after arranging for the ReadKey() SVC to be re-executed.

The call to Scheduler() suspends execution of the current process and arranges for the next process to run when the handler exits.

Eventually the round-robin scheduling will come back to the current process and the ReadKey() SVC will try again.

With this simple one-line fix the system will spend much less time wasting cycles checking the empty buffer and instead use those cycles to run other, hopefully more productive, processes.

The cost is a small delay in restarting the program after a character is typed, but typically the time slices for each process are small enough that one round of process execution happens more quickly than the time between two typed characters, so the extra delay isn't noticeable.

So now we have some insights into one of the traditional arguments against timesharing.

The argument goes as follows.

Suppose we have 10 processes, each of which takes 1 second to complete its computation.

Without timesharing, the first process would be done after 1 second, the second after 2 seconds, and so on.

With timesharing using, say, a 1/10 second time slice, all the processes will complete sometime after 10 seconds since there's a little extra time needed for the hundred or so process switches that would happen before completion.

So in a timesharing system the time-to-completion for *all* processes is as long the worst-case completion time without time sharing!

So why bother with timesharing?

We saw one answer to this question earlier in this slide.

If a process can't make productive use of its time slice, it can donate those cycles to completion of some other task.

So in a system where most processes are waiting for some sort of I/O, timesharing is actually a great way of spending cycles where they'll do the most good.

If you open the Task Manager or Activity Monitor on the system you're using now, you'll see there are hundreds of processes, almost all of which are in some sort of I/O wait.

So timesharing does extract a cost when running compute-intensive computations, but in an actual system where there's a mix of I/O and compute tasks, time sharing is the way to go.

We can actually go one step further to ensure we don't run processes waiting for an I/O event that hasn't yet happened.

We'll add a status field to the process state indicating whether the process is ACTIVE (e.g., status is 0) or WAITING (e.g., status is non-zero).

We'll use different non-zero values to indicate what event the process is waiting for.

Then we'll change the Scheduler() to only run ACTIVE processes.

To see how this works, it's easiest to use a concrete example.

The UNIX OS has two kernel subroutines, sleep() and wakeup(), both of which require a non-zero argument.

The argument will be used as the value of the status field.

Let's see this in action.

When the ReadKey() SVC detects the buffer is empty, it calls sleep() with an argument that uniquely identifies the I/O event it's waiting for, in this case the arrival of a character in a particular buffer.

sleep() sets the process status to this unique identifier, then calls Scheduler().

Scheduler() has been modified to skip over processes with a non-zero status, not giving them a chance to run.

Meanwhile, a keyboard interrupt will cause the interrupt handler to add a character to the keyboard buffer and call wakeup() to signal any process waiting on that buffer.

Watch what happens when the kbdnum in the interrupt handler matches the kbdnum in the ReadKey() handler.

wakeup() loops through all processes, looking for ones that are waiting for this particular I/O event.

When it finds one, it sets the status for the process to zero, marking it as ACTIVE.

The zero status will cause the process to run again next time the Scheduler() reaches it in its round-robin search for things to do.

The effect is that once a process goes to sleep() WAITING for an event, it's not considered for execution again until the event occurs and wakeup() marks the process as ACTIVE.

Pretty neat!

Another elegant fix to ensure that no CPU cycles are wasted on useless activity.

I can remember how impressed I was when I first saw this many years ago in a (very) early version of the UNIX code :)