Here's an old quiz problem we can use to test our understanding of all the factors that went into the final design of our ReadKey() SVC code.

We're considering three different versions (R1, R2, and R3) of the ReadKey() SVC code, all variants of the various attempts from the previous section.

And there are three types of systems (Models A, B, and C).

We've been asked to match the three handlers to the appropriate system.

Looking at R1, we see it's similar to Attempt #2 from the previous section, except it always reads from the same keyboard regardless of the process making the SVC request.

That wouldn't make much sense in a timesharing system since a single stream of input characters would be shared across all the processes.

So this handler must be intended for the Model C system, which has only a single process.

Looking at R2, we see it's similar to Attempt #1 from the previous section, which had the fatal flaw of a potentially infinite loop if attempting to read from an empty buffer.

So this code would only run successfully on the Model B system, which *does* allow device interrupts even when the CPU is running inside an SVC call.

So the keyboard interrupt would interrupt the while loop in R2 and the next iteration of the loop would discover that buffer was no longer empty.

By the process of elimination that leaves the R3 handler to be paired with the Model A system.

R3 is Attempt #3 from the previous section and is designed for our standard system in which the kernel is uninterruptible.

The problem goes on to say that a fumble-fingered summer intern has jumbled up the disks containing the handlers and sent an unknown handler version to each user running one of the three model systems.

To atone for the mistake, he's been assigned the task of reading various user messages sent after the user has tried the new handler disk on their particular system.

Based on the message, he's been asked to identify which handler disk and system the user is using.

The first message says "I get compile-time errors; Scheduler and ProcTbl are undefined!"

On the right of the slide we've included a table enumerating all the combinations of handlers and systems, where we've X-ed the matches from the previous slide

since they correspond to when the new handler would be the same as the old handler and the user wouldn't be sending a message!

The phrase "Scheduler and ProcTbl are undefined" wouldn't apply to a timesharing system, which includes both symbols.

So we can eliminate the first two columns from consideration.

And we can also eliminate the second row, since handler R2 doesn't include a call to Scheduler.

So this message came from a user trying to run handler R3 on a Model C system.

Since Model C doesn't support timesharing, it would have neither Scheduler nor ProcTbl as part the OS code.

Okay, here's the next message: "Hey, now the system always reads everybody's input from keyboard 0.

Besides that, it seems to waste a lot more CPU cycles than it used to."

R1 is the only handler that always reads from keyboard 0, so we can eliminate rows 2 and 3.

So how can we tell if R1 is being run on a Model A or a Model B system?

The R1 handler wastes a lot of cycles looping while waiting for a character to arrive and the implication is that was a big change for the user

since they're complaining that running R1 is wasting time compared to their previous handler.

If the user had been running R2 on a model B system, they're already used to the performance hit of looping

and so wouldn't have noticed a performance difference switching to R1, so we can eliminate Model B from consideration.

So this message came from a user running handler R1 on a model A system.

The final message reads "Neat, the new system seems to work fine.

It even wastes less CPU time than it used to!"

Since the system works as expected with the new handler, we can eliminate a lot of possibilities.

Handler R1 wouldn't work fine on a timesharing system since the user could tell that the processes were now all reading from the same keyboard buffer, so we can eliminate R1 on Models A and B.

And handlers R2 and R3 wouldn't work on a Model C system since that doesn't include process tables or scheduling, eliminating the right-most column.

Finally handler R2 wouldn't work on a Model A system with its uninterruptible kernel since any attempt to read from an empty buffer would cause an infinite loop.

So, the message must have been sent by a Model B user now running R3.

Well, that was fun!

Just like solving the logic puzzles you find in games magazines :)