User-mode programs need to communicate with the OS to request service or get access to useful OS data like the time of day.

But if they're running in a different MMU context than the OS, they don't have direct access to OS code and data.

And that might be bad idea in any case: the OS is usually responsible for implementing security and access policies and other users of the system would be upset if any random user program could circumvent those protections.

What's needed is the ability for user-mode programs to call OS code at specific entry points, using registers or the user-mode virtual memory to send or receive information.

We'd use these "supervisor calls" to access a well-documented and secure OS application programming interface (API).

An example of such an interface is POSIX (https://en.wikipedia.org/wiki/POSIX), a standard interface implemented by many Unix-like operating systems.

As it turns out, we have a way of transferring control from a user-mode program to a specific OS handler - just execute an illegal instruction!

We'll adopt the convention of using illegal instructions with an opcode field of 1 to serve as supervisor calls.

The low order bits of these SVC instructions will contain an index indicating which SVC service we're trying to access.

Let's see how this would work.

Here's our user-mode/kernel-mode diagram again.

Note that the user-mode programs contain supervisor calls with different indices, which when executed are intended to serve as requests for different OS services.

When an SVC instruction is executed, the hardware detects the opcode field of 1 as an illegal instruction and triggers an exception that runs the OS IllOp handler, as we saw in the previous segment.

The handler saves the process state in the temporary storage area, then dispatches to the appropriate handler based on the opcode field.

This handler can access the user's registers in the temporary storage area, or using the appropriate OS subroutines can access the contents of any user-mode virtual address.

If information is to be returned to the user, the return values can be stored in the temporary storage area, overwriting, say, the saved contents of the user's R0 register.

Then, when the handler completes, the potentially-updated saved register values are reloaded into the CPU registers and execution of the user-mode program resumes at the instruction following the supervisor call.

In the previous segment we saw how the illegal instruction handler uses a dispatch table to choose the appropriate sub-handler depending on the opcode field of the illegal instruction.

In this slide we see the sub-handler for SVC instructions, i.e., those with an opcode field of 1.

This code uses the low-order bits of the instruction to access another dispatch table to select the appropriate code for each of the eight possible SVCs.

Our Tiny OS only has a meagre selection of simple services.

A real OS would have SVCs for accessing files, dealing with network connections, managing virtual memory, spawning new processes, and so on.

Here's the code for resuming execution of the user-mode process when the SVC handler is done: simply restore the saved values for the registers and JMP to resume execution at the instruction following the SVC instruction.

There are times when for some reason the SVC request cannot be completed and the request should be retried in the future.

For example, the ReadCh SVC returns the next character typed by the user, but if no character has yet been typed, the OS cannot complete the request at this time.

In this case, the SVC handler should branch to I_Wait, which arranges for the SVC instruction to be re-executed next time this process runs and then calls Scheduler() to run the next process.

This gives all the other processes a chance to run before the SVC is tried again, hopefully this time successfully.

You can see that this code also serves as the implementation for two different SVCs!

A process can give up the remainder of its current execution time slice by calling the Yield() SVC.

This simply causes the OS to call Scheduler(), suspending execution of the current process until its next turn in the round-robin scheduling process.

And to stop execution, a process can call the Halt() SVC.

Looking at the implementation, we can see that "halt" is a bit of misnomer.

What really happens is that the system arranges to re-execute the Halt() SVC each time the process is scheduled, which then causes the OS to schedule the next process for execution.

The process appears to halt since the instruction following the Halt() SVC is never executed.

Adding new SVC handlers is straightforward.

First we need to define new SVC macros for use in user-mode programs.

In this example, we're defining SVCs for getting and setting the time of day.

Since these are the eighth and ninth SVCs, we need to make a small adjustment to the SVC dispatch code and then add the appropriate entries to the end of the dispatch table.

The code for the new handlers is equally straightforward.

The handler can access the value of the program's R0 by looking at the correct entry in the UserMState temporary holding area.

It just takes a few instructions to implement the desired operations.

The SVC mechanism provides controlled access to OS services and data.

As we'll see in a few lectures, it'll be useful that SVC handlers can't be interrupted since they are running in supervisor mode where interrupts are disabled.

So, for example, if we need to increment a value in main memory, using a LD/ADDC/ST sequence, but we want to ensure no other process execution intervenes between the LD and the ST, we can encapsulate the required functionality as an SVC, which is guaranteed to be uninterruptible.

We've made an excellent start at exploring the implementation of a simple time-shared operating system.

We'll continue the exploration in the next lecture when we see how the OS deals with external input/output devices.