A key technology for timesharing is the periodic interrupt from the external timer device.

Let's remind ourselves how the interrupt hardware in the Beta works.

External devices request an interrupt by asserting the Beta's interrupt request (IRQ) input.

If the Beta is running in user mode, i.e., the supervisor bit stored in the PC is 0, asserting IRQ will trigger the following actions on the clock cycle the interrupt is recognized.

The goal is to save the current PC+4 value in the XP register and force the program counter (PC) to a particular kernel-mode instruction, which starts the execution of the interrupt handler code.

The normal process of generating control signals based on the current instruction is superseded by forcing particular values for some of the control signals.

PCSEL is set to 4, which selects a specified kernel-mode address as the next value of the program counter.

The address chosen depends on the type of external interrupt.

In the case of the timer interrupt, the address is 0x80000008.

Note that PC[31], the supervisor bit, is being set to 1 and the CPU will be in kernel-mode as it starts executing the code of the interrupt handler.

The WASEL, WDSEL, and WERF control signals are set so that PC+4 is written into the XP register (i.e., R30) in the register file.

And, finally, MWR is set to 0 to ensure that if we're interrupting a ST instruction that its execution is aborted correctly.

So in the next clock cycle, execution starts with the first instruction of the kernel-mode interrupt handler, which can find the PC+4 of the interrupted instruction in the XP register of the CPU.

As we can see the interrupt hardware is pretty minimal: it saves the PC+4 of the interrupted user-mode program in the XP register and sets the program counter to some predetermined value that depends on which external interrupt happened.

The remainder of the work to handle the interrupt request is performed in software.

The state of the interrupted process, e.g., the values in the CPU registers R0 through R30, is stored in main

memory in an OS data structure called UserMState.

Then the appropriate handler code, usually a procedure written in C, is invoked to do the heavy lifting.

When that procedure returns, the process state is reloaded from UserMState.

The OS subtracts 4 from the value in XP, making it point to the interrupted instruction and then resumes user-mode execution with a JMP(XP).

Note that in our simple Beta implementation the first instructions for the various interrupt handlers occupy consecutive locations in main memory.

Since interrupt handlers are longer than one instruction, this first instruction is invariably a branch to the actual interrupt code.

Here we see that the reset interrupt (asserted when the CPU first starts running) sets the PC to 0, the illegal instruction interrupt sets the PC to 4, the timer interrupt sets the PC to 8, and so on.

In all cases, bit 31 of the new PC value is set to 1 so that handlers execute in supervisor or kernel mode, giving them access to the kernel context.

A common alternative is provide a table of new PC values at a known location and have the interrupt hardware access that table to fetch the PC for the appropriate handler routine.

This provides the same functionality as our simple Beta implementation.

Since the process state is saved and restored during an interrupt, interrupts are transparent to the running user-mode program.

In essence, we borrow a few CPU cycles to deal with the interrupt, then it's back to normal program execution.

Here's how the timer interrupt handler would work.

Our initial goal is to use the timer interrupt to update a data value in the OS that records the current time of day (TOD).

Let's assume the timer interrupt is triggered every 1/60th of a second.

A user-mode program executes normally, not needing to make any special provision to deal with timer interrupts.

Periodically the timer interrupts the user-mode program to run the clock interrupt handler code in the OS, then

resumes execution of the user-mode program.

The program continues execution just as if the interrupt had not occurred.

If the program needs access to the TOD, it makes the appropriate service request to the OS.

The clock handler code in the OS starts and ends with a small amount of assembly-language code to save and restore the state.

In the middle, the assembly code makes a C procedure call to actually handle the interrupt.

Here's what the handler code might look like.

In C, we find the declarations for the TOD data value and the structure, called UserMState, that temporarily holds the saved process state.

There's also the C procedure for incrementing the TOD value.

A timer interrupt executes the BR() instruction at location 8, which branches to the actual interrupt handler code at CLOCK_H.

The code first saves the values of all the CPU registers into the UserMState data structure.

Note that we don't save the value of R31 since its value is always 0.

After setting up the kernel-mode stack, the assembly-language stub calls the C procedure above to do the hard work.

When the procedure returns, the CPU registers are reloaded from the saved process state and the XP register value decremented by 4 so that it will point to the interrupted instruction.

Then a JMP(XP) resumes user-mode execution.

Okay, that was simple enough.

But what does this all have to do with timesharing?

Wasn't our goal to arrange to periodically switch which process was running?

Aha!

We have code that runs on every timer interrupt, so let's modify it so that every so often we arrange to call the OS'

Scheduler() routine.

In this example, we'd set the constant QUANTUM to 2 if we wanted to call Scheduler() every second timer interrupt.

The Scheduler() subroutine is where the time sharing magic happens!

Here we see the UserMState data structure from the previous slide where the user-mode process state is stored during interrupts.

And here's an array of process control block (PCB) data structures, one for each process in the system.

The PCB holds the complete state of a process when some other process is currently executing - it's the long-term storage for processor state!

As you can see, it includes a copy of MState with the process' register values, the MMU state, and various state associated with the process' input/output activities, represented here by a number indicating which virtual user-interface console is attached to the process.

There are N processes altogether.

The variable CUR gives the index into ProcTable for the currently running process.

And here's the surprisingly simple code for implementing timesharing.

Whenever the Scheduler() routine is called, it starts by moving the temporary saved state into the PCB for the current process.

It then increments CUR to move to the next process, making sure it wraps back around to 0 when we've just finished running the last of the N processes.

It then loads reloads the temporary state from the PCB of the new process and sets up the MMU appropriately.

At this point Scheduler() returns and the clock interrupt handler reloads the CPU registers from the updated temporary saved state and resumes execution.

Voila!

We're now running a new process.

Let's use this diagram to once again walk through how time sharing works.

At the top of the diagram you'll see the code for the user-mode processes, and below the OS code along with its data structures.

The timer interrupts the currently running user-mode program and starts execution of the OS' clock handler code.

The first thing the handler does is save all the registers into the UserMState data structure.

If the Scheduler() routine is called, it moves the temporarily saved state into the PCB, which provides the long-term storage for a process' state.

Next, Scheduler() copies the saved state for the next process into the temporary holding area.

Then the clock handler reloads the updated state into the CPU registers and resumes execution, this time running code in the new process.

While we're looking at the OS, note that since its code runs with the supervisor mode bit set to 1, interrupts are disabled while in the OS.

This prevents the awkward problem of getting a second interrupt while still in the middle of handling a first interrupt, a situation that might accidentally overwrite the state in UserMState.

But that means one has to be very careful when writing OS code.

Any sort of infinite loop can never be interrupted.

You may have experienced this when your machine appears to freeze, accepting no inputs and just sitting there like a lump.

At this point, your only choice is to power-cycle the hardware (the ultimate interrupt!) and start afresh.

Interrupts are allowed during execution of user-mode programs, so if they run amok and need to be interrupted, that's always possible since the OS is still responding to, say, keyboard interrupts.

Every OS has a magic combination of keystrokes that is guaranteed to suspend execution of the current process, sometimes arranging to make a copy of the process state for later debugging.

Very handy!