Let's create a new abstraction called a "process" to capture the notion of a running program.

A process encompasses all the resources that would be used when running a program including those of the CPU, the MMU, input and output devices, etc.

Each process has a "state" that captures everything we know about its execution.

The process state includes * the hardware state of the CPU, i.e., the values in the registers and program counter.

* the contents of the process' virtual address space, including code, data values, the stack, and data objects dynamically allocated from the heap.

Under the management of the MMU, this portion of the state can be resident in main memory or can reside in secondary storage.

* the hardware state of the MMU, which, as we saw earlier, depends on the context-number and page-directory registers.

Also included are the pages allocated for the hierarchical page map.

* additional information about the process' input and output activities, such as where it has reached in reading or writing files in the file system, the status and buffers associated with open network connections, pending events from the user interface (e.g., keyboard characters and mouse clicks), and so on.

As we'll see, there is a special, privileged process, called the operating system (OS), running in its own kernel-mode context.

The OS manages all the bookkeeping for each process, arranging for the process run periodically.

The OS will provide various services to the processes, such as accessing data in files, establishing network connections, managing the window system and user interface, and so on.

To switch from running one user-mode process to another, the OS will need to capture and save the *entire* state of the current user-mode process.

Some of it already lives in main memory, so we're all set there.

Some of it will be found in various kernel data structures.

And some of it we'll need to be able to save and restore from the various hardware resources in the CPU and

MMU.

In order to successfully implement processes, the OS must be able to make it seem as if each process was running on its own "virtual machine" that works independently of other virtual machines for other processes.

Our goal is to efficiently share one physical machine between all the virtual machines.

Here's a sketch of the organization we're proposing.

The resources provided by a physical machine are shown at the bottom of the slide.

The CPU and main memory form the computation engine at heart of the system.

Connected to the CPU are various peripherals, a collective noun coined from the English word "periphery" that indicates the resources surrounding the CPU.

A timer generates periodic CPU interrupts that can be used to trigger periodic actions.

Secondary storage provides high-capacity non-volatile memories for the system.

Connections to the outside world are important too.

Many computers include USB connections for removable devices.

And most provide wired or wireless network connections.

And finally there are usually video monitors, keyboards and mice that serve as the user interface.

Cameras and microphones are becoming increasing important as the next generation of user interface.

The physical machine is managed by the OS running in the privileged kernel context.

The OS handles the low-level interfaces to the peripherals, initializes and manages the MMU contexts, and so on.

It's the OS that creates the virtual machine seen by each process.

User-mode programs run directly on the physical processor, but their execution can be interrupted by the timer, giving the OS the opportunity to save away the current process state and move to running the next process.

Via the MMU, the OS provides each process with an independent virtual address space that's isolated from the actions of other processes.

The virtual peripherals provided by the OS isolate the process from all the details of sharing resources with other processes.

The notion of a window allows the process to access a rectangular array of pixels without having to worry if some pixels in the window are hidden by other windows.

Or worrying about how to ensure the mouse cursor always appears on top of whatever is being displayed, and so on.

Instead of accessing I/O devices directly, each process has access to a stream of I/O events that are generated when a character is typed, the mouse is clicked, etc.

For example, the OS deals with how to determine which typed characters belong to which process.

In most window systems, the user clicks on a window to indicate that the process that owns the window now has the keyboard focus and should receive any subsequent typed characters.

And the position of the mouse when clicked might determine which process receives the click.

All of which is to say that the details of sharing have been abstracted out of the simple interface provided by the virtual peripherals.

The same is true of accessing files on disk.

The OS provides the useful abstraction of having each file appear as a contiguous, growable array of bytes that supports read and write operations.

The OS knows how the file is mapped to a pool of sectors on the disk and deals with bad sectors, reducing fragmentation, and improving throughput by doing read look-aheads and write behinds.

For networks, the OS provides access to an in-order stream of bytes to some remote socket.

It implements the appropriate network protocols for packetizing the stream, addressing the packets, and dealing with dropped, damaged, or out-of-order packets.

To configure and control these virtual services, the process communicates with the OS using supervisor calls (SVCs), a type of controlled-access procedure call that invokes code in the OS kernel.

The details of the design and implementation of each virtual service are beyond the scope of this course.

If you're interested, a course on operating systems will explore each of these topics in detail.

The OS provides an independent virtual machine for each process, periodically switching from running one process to running the next process.

Let's follow along as we switch from running process #0 to running process #1.

Initially, the CPU is executing user-mode code in process #0.

That execution is interrupted, either by an explicit yield by the program, or, more likely, by a timer interrupt.

Either ends up transferring control to OS code running in kernel mode, while saving the current PC+4 value in the XP register.

We'll talk about the interrupt mechanism in more detail in just a moment.

The OS saves the state of process #0 in the appropriate table in kernel storage.

Then it reloads the state from the kernel table for process #1.

Note that the process #1 state was saved when process #1 was interrupted at some earlier point.

The OS then uses a JMP() to resume user-mode execution using the newly restored process #1 state.

Execution resumes in process #1 just where it was when interrupted earlier.

And now we're running the user-mode program in process #1.

We've interrupted one process and resumed execution of another.

We'll keep doing this in a round-robin fashion, giving each process a chance to run, before starting another round of execution.

The black arrows give a sense for how time proceeds.

For each process, virtual time unfolds as a sequence of executed instructions.

Unless it looks at a real-time clock, a process is unaware that occasionally its execution is suspended for a while.

The suspension and resumption are completely transparent to a running process.

Of course, from the outside we can see that in real time, the execution path moves from process to process, visiting the OS during switches, producing the dove-tailed execution path we see here.

Time-multiplexing of the CPU is called "timesharing" and we'll examine the implementation in more detail in the following segment.