The page map provides the context for interpreting virtual addresses, i.e., it provides the information needed to correctly determine where to find a virtual address in main memory or secondary storage.

Several programs may be simultaneously loaded into main memory, each with its own context.

Note that the separate contexts ensure that the programs don't interfere which each other.

For example, the physical location for virtual address 0 in one program will be different than the physical location for virtual address 0 in another program.

Each program operates independently in its own virtual address space.

It's the context provided by the page map that allows them to coexist and share a common physical memory.

So we need to switch contexts when switching programs.

This is accomplished by reloading the page map.

In a timesharing system, the CPU will periodically switch from running one program to another, giving the illusion that multiple programs are each running on their own virtual machine.

This is accomplished by switching contexts when switching the CPU state to the next program.

There's a privileged set of code called the operating system (OS) that manages the sharing of one physical processor and main memory amongst many programs, each with its own CPU state and virtual address space.

The OS is effectively creating many virtual machines and choreographing their execution using a single set of shared physical resources.

The OS runs in a special OS context, which we call the kernel.

The OS contains the necessary exception handlers and timesharing support.

Since it has to manage physical memory, it's allowed to access any physical location as it deals with page faults, etc.

Exceptions in running programs cause the hardware to switch to the kernel context, which we call entering "kernel mode".

After the exception handling is complete, execution of the program resumes in what we call "user mode".

Since the OS runs in kernel mode it has privileged access to many hardware registers that are inaccessible in

user mode.

These include the MMU state, I/O devices, and so on.

User-mode programs that need to access, say, the disk, need to make a request to the OS kernel to perform the operation, giving the OS the chance to vet the request for appropriate permissions, etc.

We'll see how all of this works in an upcoming lecture.

User-mode programs (aka applications) are written as if they have access to the entire virtual address space.

They often obey the same conventions such as the address of the first instruction in the program, the initial value for the stack pointer, etc.

Since all these virtual addresses are interpreted using the current context, by controlling the contexts the OS can ensure that the programs can coexist without conflict.

The diagram on the right shows a standard plan for organizing the virtual address space of an application.

Typically the first virtual page is made inaccessible, which helps catch errors involving references to uninitialized (i.e., zero-valued) pointers.

Then come some number of read-only pages that hold the application's code and perhaps the code from any shared libraries it uses.

Marking code pages as read-only avoids hard-to-find bugs where errant data accesses inadvertently change the program!

Then there are read-write pages holding the application's statically allocated data structures.

The rest of the virtual address space is divided between two data regions that can grow over time.

The first is the application's stack, used to hold procedure activation records.

Here we show it located at the lower end of the virtual address space since our convention is that the stack grows towards higher addresses.

The other growable region is the heap, used when dynamically allocating storage for long-lived data structures.

"Dynamically" means that the allocation and deallocation of objects is done by explicit procedure calls while the application is running.

In other words, we don't know which objects will be created until the program actually executes.

As shown here, as the heap expands it grows towards lower addresses.

The page fault handler knows to allocate new pages when these regions grow.

Of course, if they ever meet somewhere in the middle and more space is needed, the application is out of luck - it's run out of virtual memory!