In the last lecture we introduced the notion of virtual memory and added a Memory Management Unit (MMU) to translate the virtual addresses generated by the CPU to the physical addresses sent to main memory.

This gave us the ability to share physical memory between many running programs while still giving each program the illusion of having its own large address space.

Both the virtual and physical address spaces are divided into a sequence of pages, each holding some fixed number of locations.

For example if each page holds 2^12 bytes, a 32-bit address would have 2^32/2^12 = 2^20 pages.

In this example the 32-bit address can be thought of as having two fields: a 20-bit page number formed from the high-order address bits and a 12-bit page offset formed from the low-order address bits.

This arrangement ensures that nearby data will be located on the same page.

The MMU translates virtual page numbers into physical page numbers using a page map.

Conceptually the page map is an array where each entry in the array contains a physical page number along with a couple of bits indicating the page status.

The translation process is simple: the virtual page number is used as an index into the array to fetch the corresponding physical page number.

The physical page number is then combined with the page offset to form the complete physical address.

In the actual implementation the page map is usually organized into multiple levels, which permits us to have resident only the portion of the page map we're actively using.

And to avoid the costs of accessing the page map on each address translation, we use a cache (called the translation look-aside buffer) to remember the results of recent vpn-to-ppn translations.

All allocated locations of each virtual address space can be found on secondary storage.

Note that they may not necessarily be resident in main memory.

If the CPU attempts to access a virtual address that's not resident in main memory, a page fault is signaled and the operating system will arrange to move the desired page from secondary storage into main memory.

In practice, only the active pages for each program are resident in main memory at any given time.

Here’s a diagram showing the translation process.

First we check to see if the required vpn-to-ppn mapping is cached in the TLB.

If not, we have to access the hierarchical page map to see if the page is resident and, if so, lookup its physical page number.

If we discover that the page is not resident, a page fault exception is signaled to the CPU so that it can run a handler to load the page from secondary storage.

Note that access to a particular mapping context is controlled by two registers.

The context-number register controls which mappings are accessible in the TLB.

And the page-directory register indicates which physical page holds the top tier of the hierarchical page map.

We can switch to another context by simply reloading these two registers.

To effectively accommodate multiple contexts we’ll need to have sufficient TLB capacity to simultaneously cache the most frequent mappings for all the processes.

And we’ll need some number of physical pages to hold the required page directories and segments of the page tables.

For example, for a particular process, three pages will suffice hold the resident two-level page map for 1024 pages at each end of the virtual address space,

providing access to up to 8MB of code, stack, and heap, more than enough for many simple programs.

The page map creates the context needed to translate virtual addresses to physical addresses.

In a computer system that’s working on multiple tasks at the same time, we would like to support multiple contexts and to be able to quickly switch from one context to another.

Multiple contexts would allow us to share physical memory between multiple programs.

Each program would have an independent virtual address space, e.g., two programs could both access virtual address 0 as the address of their first instruction and would end up accessing different physical locations in main memory.

When switching between programs, we’d perform a “context switch” to move to the appropriate MMU context.

The ability to share the CPU between many programs seems like a great idea!

Let’s figure out the details of how that might work…