Here's how our virtual memory system will work.

The memory addresses generated by the CPU are called virtual addresses to distinguish them from the physical addresses used by main memory.

In between the CPU and main memory there's a new piece of hardware called the memory management unit (MMU).

The MMU's job is to translate virtual addresses to physical addresses.

"But wait!" you say.

"Doesn't the cache go between the CPU and main memory?" You're right and at the end of this lecture we'll talk about how to use both an MMU and a cache.

But for now, let's assume there's only an MMU and no cache.

The MMU hardware translates virtual addresses to physical addresses using a simple table lookup.

This table is called the page map or page table.

Conceptually, the MMU uses the virtual address as index to select an entry in the table, which tells us the corresponding physical address.

The table allows a particular virtual address to be found anywhere in main memory.

In normal operation we'd want to ensure that two virtual addresses don't map to the same physical address.

But it would be okay if some of the virtual addresses did not have a translation to a physical address.

This would indicate that the contents of the requested virtual address haven't yet been loaded into main memory, so the MMU would signal a memory-management exception to the CPU, which could assign a location in physical memory and perform the required I/O operation to initialize that location from secondary storage.

The MMU table gives the system a lot of control over how physical memory is accessed by the program running on the CPU.

For example, we could arrange to run multiple programs in quick succession (a technique called time sharing) by changing the page map when we change programs.

Main memory locations accessible to one program could be made inaccessible to another program by proper

management of their respective page maps.

And we could use memory-management exceptions to load program contents into main memory on demand instead of having to load the entire program before execution starts.

In fact, we only need to ensure the current working set of a program is actually resident in main memory.

Locations not currently being used could live in secondary storage until needed.

In this lecture and next, we'll see how the MMU plays a central role in the design of a modern timesharing computer system.

Of course, we'd need an impossibly large table to separately map each virtual address to a physical address.

So instead we divide both the virtual and physical address spaces into fixed-sized blocks, called pages.

Page sizes are always a power-of-2 bytes, say 2^p bytes, so p is the number address bits needed to select a particular location on the page.

We'll the use low-order p bits of the virtual or physical address as the page offset.

The remaining address bits tell us which page is being accessed and are called the page number.

A typical page size is 4KB to 16KB, which correspond to p=12 and p=14 respectively.

Suppose p=12.

If the CPU produces a 32-bit virtual address, the low-order 12 bits of the virtual address are the page offset and the high-order 20 bits are the virtual page number.

Similarly, the low-order p bits of the physical address are the page offset and the remaining physical address bits are the physical page number.

The key idea is that the MMU will manage pages, not individual locations.

We'll move entire pages from secondary storage into main memory.

By the principal of locality, if a program access one location on a page, we expect it will soon access other nearby locations.

By choosing the page offset from the low-order address bits, we'll ensure that nearby locations live on the same

page (unless of course we're near one end of the page or the other).

So pages naturally capture the notion of locality.

And since pages are large, by dealing with pages when accessing secondary storage, we'll take advantage that reading or writing many locations is only slightly more time consuming than accessing the first location.

The MMU will map virtual page numbers to physical page numbers.

It does this by using the virtual page number (VPN) as an index into the page table.

Each entry in the page table indicates if the page is resident in main memory and, if it is, provides the appropriate physical page number (PPN).

The PPN is combined with the page offset to form the physical address for main memory.

If the requested virtual page is NOT resident in main memory, the MMU signals a memory-management exception, called a page fault, to the CPU so it can load the appropriate page from secondary storage and set up the appropriate mapping in the MMU.

Our plan to use main memory as page cache is called "paging" or sometimes "demand paging" since movements of pages to and from secondary storage is determined by the demands of the program.

So here's the plan.

Initially all the virtual pages for a program reside in secondary storage and the MMU is empty, i.e., there are no pages resident in physical memory.

The CPU starts running the program and each virtual address it generates, either for an instruction fetch or data access, is passed to the MMU to be mapped to a physical address in main memory.

If the virtual address is resident in physical memory, the main memory hardware can complete the access.

If the virtual address in NOT resident in physical memory, the MMU signals a page fault exception, forcing the CPU to switch execution to special code called the page fault handler.

The handler allocates a physical page to hold the requested virtual page and loads the virtual page from secondary storage into main memory.

It then adjusts the page map entry for the requested virtual page to show that it is now resident and to indicate the physical page number for the newly allocated and initialized physical page.

When trying to allocate a physical page, the handler may discover that all physical pages are currently in use.

In this case it chooses an existing page to replace, e.g., a resident virtual page that hasn't been recently accessed.

It swaps the contents of the chosen virtual page out to secondary storage and updates the page map entry for the replaced virtual page to indicate it is no longer resident.

Now there's a free physical page to re-use to hold the contents of the virtual page that was missing.

The working set of the program, i.e., the set of pages the program is currently accessing, is loaded into main memory through a series of page faults.

After a flurry of page faults when the program starts running, the working set changes slowly, so the frequency of page faults drops dramatically, perhaps close to zero if the program is small and well-behaved.

It is possible to write programs that consistently generate page faults, a phenomenon called thrashing.

Given the long access times of secondary storage, a program that's thrashing runs *very* slowly, usually so slowly that users give up and rewrite the program to behave more sensibly.

The design of the page map is straightforward.

There's one entry in the page map for each virtual page.

For example, if the CPU generates a 32-bit virtual address and the page size is 2^12 bytes, the virtual page number has 32-12 = 20 bits and the page table will have 2^20 entries.

Each entry in the page table contains a "resident bit" (R) which is set to 1 when the virtual page is resident in physical memory.

If R is 0, an access to that virtual page will cause a page fault.

If R is 1, the entry also contains the PPN, indicating where to find the virtual page in main memory.

There's one additional state bit called the "dirty bit" (D).

When a page has just been loaded from secondary storage, it's "clean", i.e, the contents of physical memory match the contents of the page in secondary storage.

So the D bit is set to 0.

If subsequently the CPU stores into a location on the page, the D bit for the page is set to 1, indicating the page is "dirty", i.e., the contents of memory now differ from the contents of secondary storage.

If a dirty page is ever chosen for replacement, its contents must be written to secondary storage in order to save the changes before the page gets reused.

Some MMUs have additional state bits in each page table entry.

For example, there could be a "read-only" bit which, when set, would generate an exception if the program attempts to store into the page.

This would be useful for protecting code pages from accidentally being corrupted by errant data accesses, a very handy debugging feature.

Here's an example of the MMU in action.

To make things simple, assume that the virtual address is 12 bits, consisting of an 8-bit page offset and a 4-bit virtual page number.

So there are 2^4 = 16 virtual pages.

The physical address is 11 bits, divided into the same 8-bit page offset and a 3-bit physical page number.

So there are 2^3 = 8 physical pages.

On the left we see a diagram showing the contents of the 16-entry page map, i.e., an entry for each virtual page.

Each page table entry includes a dirty bit (D), a resident bit (R) and a 3-bit physical page number, for a total of 5 bits.

So the page map has 16 entries, each with 5-bits, for a total of 16*5 = 80 bits.

The first entry in the table is for virtual page 0, the second entry for virtual page 1, and so on.

In the middle of the slide there's a diagram of physical memory showing the 8 physical pages.

The annotation for each physical page shows the virtual page number of its contents.

Note that there's no particular order to how virtual pages are stored in physical memory - which page holds what

is determined by which pages are free at the time of a page fault.

In general, after the program has run for a while, we'd expected to find the sort of jumbled ordering we see here.

Let's follow along as the MMU handles the request for virtual address 0x2C8, generated by the execution of the LD instruction shown here.

Splitting the virtual address into page number and offset, we see that the VPN is 2 and the offset is 0xC8.

Looking at the page map entry with index 2, we see that the R bit is 1, indicating that virtual page 2 is resident in physical memory.

The PPN field of entry tells us that virtual page 2 can be found in physical page 4.

Combining the PPN with the 8-bit offset, we find that the contents of virtual address 0x2C8 can be found in main memory location 0x4C8.

Note that the offset is unchanged by the translation process - the offset into the physical page is always the same as the offset into the virtual page.