Let's review what happens when the CPU accesses a non-resident virtual page, i.e., a page with its resident bit set to 0.

In the example shown here, the CPU is trying to access virtual page 5.

In this case, the MMU signals a page fault exception, causing the CPU to suspend execution of the program and switch to the page fault handler, which is code that deals with the page fault.

The handler starts by either finding an unused physical page or, if necessary, creating an unused page by selecting an in-use page and making it available.

In our example, the handler has chosen virtual page 1 for reuse.

If the selected page is dirty, i.e., its D bit is 1 indicating that its contents have changed since being read from secondary storage, write it back to secondary storage.

Finally, mark the selected virtual page as no longer resident.

In the "after" figure, we see that the R bit for virtual page 1 has been set to 0.

Now physical page 4 is available for re-use.

Are there any restrictions on which page we can select?

Obviously, we can't select the page that holds the code for the page fault handler.

Pages immune from selection are called "wired" pages.

And it would very inefficient to choose the page that holds the code that made the initial memory access, since we expect to start executing that code as soon as we finish handling the page fault.

The optimal strategy would be to choose the page whose next use will occur farthest in the future.

But, of course, this involves knowledge of future execution paths and so isn't a realizable strategy.

Wikipedia provides a nice description of the many strategies for choosing a replacement page, with their various tradeoffs between ease of implementation and impact on the rate of page faults - see the URL given at the bottom of the slide.

The aging algorithm they describe is frequently used since it offers near optimal performance at a moderate implementation cost.

Next, the desired virtual page is read from secondary storage into the selected physical page.

In our example, virtual page 5 is now loaded into physical page 4.

Then the R bit and PPN fields in the page table entry for virtual page 5 are updated to indicate that the contents of that virtual page now reside in physical page 4.

Finally the handler is finished and execution of the original program is resumed, re-executing the instruction that caused the page fault.

Since the page map has been updated, this time the access succeeds and execution continues.

To double-check our understanding of page faults, let's run through an example.

Here's the same setup as in our previous example, but this time consider a store instruction that's making an access to virtual address 0x600, which is located on virtual page 6.

Checking the page table entry for VPN 6, we see that its R bit 0 indicating that it is NOT resident in main memory, which causes a page fault exception.

The page fault handler selects VPN 0xE for replacement since we've been told in the setup that it's the least-recently-used page.

The page table entry for VPN 0xE has D=1 so the handler writes the contents of VPN 0xE, which is found in PPN 0x5, to secondary storage.

Then it updates the page table to indicate that VPN 0xE is no longer resident.

Next, the contents of VPN 0x6 are read from secondary storage into the now available PPN 0x5.

Now the handler updates the page table entry for VPN 0x6 to indicate that it's resident in PPN 0x5.

The page fault handler has completed its work, so program execution resumes and the ST instruction is re-executed.

This time the MMU is able to translate virtual address 0x600 to physical address 0x500.

And since the ST instruction modifies the contents of VPN 0x6, its D bit is set to 1.

Whew!

We're done :) We can think of the work of the MMU as being divided into two tasks, which as computer scientists, we would think of as two procedures.

In this formulation the information in the page map is held in several arrays: the R array holds the resident bits, the D array holds the dirty bits, the PPN array holds the physical page numbers, and the DiskAdr array holds the location in secondary storage for each virtual page.

The VtoP procedure is invoked on each memory access to translate the virtual address into a physical address.

If the requested virtual page is not resident, the PageFault procedure is invoked to make the page resident.

Once the requested page is resident, the VPN is used as an index to lookup the corresponding PPN, which is then concatenated with the page offset to form the physical address.

The PageFault routine starts by selecting a virtual page to be replaced, writing out its contents if it's dirty.

The selected page is then marked as not resident.

Finally the desired virtual page is read from secondary storage and the page map information updated to reflect that it's now resident in the newly filled physical page.

We'll use hardware to implement the VtoP functionality since it's needed for every memory access.

The call to the PageFault procedure is accomplished via a page fault exception, which directs the CPU to execute the appropriate handler software that contains the PageFault procedure.

This is a good strategy to pursue in all our implementation choices: use hardware for the operations that need to be fast, but use exceptions to handle the (hopefully infrequent) exceptional cases in software.

Since the software is executed by the CPU, which is itself a piece of hardware, what we're really doing is making the tradeoff between using special-purpose hardware (e.g., the MMU) or using general-purpose hardware (e.g., the CPU).

In general, one should be skeptical of proposals to use special-purpose hardware, reserving that choice for operations that truly are commonplace and whose performance is critical to the overall performance of the system.