There are a few MMU implementation details we can tweak for more efficiency or functionality.

In our simple page-map implementation, the full page map occupies some number of physical pages.

Using the numbers shown here, if each page map entry occupies one word of main memory, we'd need 2^20 words (or 2^10 pages) to hold the page table.

If we have multiple contexts, we would need multiple page tables, and the demands on our physical memory resources would start to get large.

The MMU implementation shown here uses a hierarchical page map.

The top 10 bits of virtual address are used to access a "page directory", which indicates the physical page that holds the page map for that segment of the virtual address space.

The key idea is that the page map segments are in virtual memory, i.e., they don't all have to be resident at any given time.

If the running application is only actively using a small portion of its virtual address space, we may only need a handful of pages to hold the page directory and the necessary page map segments.

The resultant savings really add up when there are many applications, each with their own context.

In this example, note that the middle entries in the page directory, i.e., the entries corresponding to the as-yet unallocated virtual memory between the stack and heap, are all marked as not resident.

.133 So no page map resources need be devoted to holding a zillion page map entries all marked "not resident".

Accessing the page map now requires two access to main memory (first to the page directory, then to the appropriate segment of the page map), but the TLB makes the impact of that additional access negligible.

Normally when changing contexts, the OS would reload the page-table pointer to point to the appropriate page table (or page table directory if we adopt the scheme from the previous slide).

Since this context switch in effect changes all the entries in the page table, the OS would also have to invalidate all the entries in the TLB cache.

This naturally has a huge impact on the TLB hit ratio and the average memory access time takes a huge hit because of the all page map accesses that are now necessary until the TLB is refilled.

To reduce the impact of context switches, some MMUs include a context-number register whose contents are concatenated with the virtual page number to form the query to the TLB.

Essentially this means that the tag field in the TLB cache entries will expand to include the context number provided at the time the TLB entry was filled.

To switch contexts, the OS would now reload both the context-number register and the page-table pointer.

With a new context number, entries in the TLB for other contexts would no longer match, so no need to flush the TLB on a context switch.

If the TLB has sufficient capacity to cache the VPN-to-PPN mappings for several contexts, context switches would no longer have a substantial impact on average memory access time.

Finally, let's return to the question about how to incorporate both a cache and an MMU into our memory system.

The first choice is to place the cache between the CPU and the MMU, i.e., the cache would work on virtual addresses.

This seems good: the cost of the VPN-to-PPN translation is only incurred on a cache miss.

The difficulty comes when there's a context switch, which changes the effective contents of virtual memory.

After all that was the point of the context switch, since we want to switch execution to another program.

But that means the OS would have to invalidate all the entries in the cache when performing a context switch, which makes the cache miss ratio quite large until the cache is refilled.

So once again the performance impact of a context switch would be quite high.

We can solve this problem by caching physical addresses, i.e., placing the cache between the MMU and main memory.

Thus the contents of the cache are unaffected by context switches - the requested physical addresses will be different, but the cache handles that in due course.

The downside of this approach is that we have to incur the cost of the MMU translation before we can start the cache access, slightly increasing the average memory access time.

But if we're clever we don't have to wait for the MMU to finish before starting the access to the cache.

To get started, the cache needs the line number from the virtual address in order to fetch the appropriate cache line.

If the address bits used for the line number are completely contained in the page offset of the virtual address, these bits are unaffected by the MMU translation, and so the cache lookup can happen in parallel with the MMU operation.

Once the cache lookup is complete, the tag field of the cache line can be compared with the appropriate bits of the physical address produced by the MMU.

If there was a TLB hit in the MMU, the physical address should be available at about the same time as the tag field produced by the cache lookup.

By performing the MMU translation and cache lookup in parallel, there's usually no impact on the average memory access time!

Voila, the best of both worlds: a physically addressed cache that incurs no time penalty for MMU translation.

One final detail: one way to increase the capacity of the cache is to increase the number of cache lines and hence the number of bits of address used as the line number.

Since we want the line number to fit into the page offset field of the virtual address, we're limited in how many cache lines we can have.

The same argument applies to increasing the block size.

So to increase the capacity of the cache our only option is to increase the cache associativity, which adds capacity without affecting the address bits used for the line number.

That's it for our discussion of virtual memory.

We use the MMU to provide the context for mapping virtual addresses to physical addresses.

By switching contexts we can create the illusion of many virtual address spaces, so many programs can share a single CPU and physical memory without interfering with each other.

We discussed using a page map to translate virtual page numbers to physical page numbers.

To save costs, we located the page map in physical memory and used a TLB to eliminate the cost of accessing the page map for most virtual memory accesses.

Access to a non-resident page causes a page fault exception, allowing the OS to manage the complexities of equitably sharing physical memory across many applications.

We saw that providing contexts was the first step towards creating virtual machines, which is the topic of our next lecture.