In this lecture we return to the memory system that we last discussed in Lecture 14 of Part 2.

There we learned about the fundamental tradeoff in current memory technologies: as the memory's capacity increases, so does it access time.

It takes some architectural cleverness to build a memory system that has a large capacity and a small average access time.

The cleverness is embodied in the cache, a hardware subsystem that lives between the CPU and main memory.

Modern CPUs have several levels of cache, where the modest-capacity first level has an access time close to that of the CPU, and higher levels of cache have slower access times but larger capacities.

Caches give fast access to a small number of memory locations, using associative addressing so that the cache has the ability to hold the contents of the memory locations the CPU is accessing most frequently.

The current contents of the cache are managed automatically by the hardware.

Caches work well because of the principle of locality: if the CPU accesses location X at time T, it's likely to access nearby locations in the not-too-distant future.

The cache is organized so that nearby locations can all reside in the cache simultaneously, using a simple indexing scheme to choose which cache location should be checked for a matching address.

If the address requested by the CPU resides in the cache, access time is quite fast.

In order to increase the probability that requested addresses reside in the cache, we introduced the notion of "associativity", which increased the number of cache locations checked on each access and solved the problem of having, say, instructions and data compete for the same cache locations..

We also discussed appropriate choices for block size (the number of words in a cache line), replacement policy (how to choose which cache line to reuse on a cache miss), and write policy (deciding when to write changed data back to main memory).

We'll see these same choices again in this lecture as we work to expand the memory hierarchy beyond main memory.

We never discussed where the data in main memory comes from and how the process of filling main memory is managed.

That's the topic of today's lecture..

Flash drives and hard disks provide storage options that have more capacity than main memory, with the added benefit of being non-volatile, i.e., they continue to store data even when turned off.

The generic name for these new devices is "secondary storage", where data will reside until it's moved to "primary storage", i.e., main memory, for use.

So when we first turn on a computer system, all of its data will be found in secondary storage, which we'll think of as the final level of our memory hierarchy.

As we think about the right memory architecture, we'll build on the ideas from our previous discussion of caches, and, indeed, think of main memory as another level of cache for the permanent, high-capacity secondary storage.

We'll be building what we call a virtual memory system, which, like caches, will automatically move data from secondary storage into main memory as needed.

The virtual memory system will also let us control what data can be accessed by the program, serving as a stepping stone to building a system that can securely run many programs on a single CPU.

Let's get started!

Here we see the cache and main memory, the two components of our memory system as developed in Lecture 14.

And here's our new secondary storage layer.

The good news: the capacity of secondary storage is huge!

Even the most modest modern computer system will have 100's of gigabytes of secondary storage and having a terabyte or two is not uncommon on medium-size desktop computers.

Secondary storage for the cloud can grow to many petabytes (a petabyte is 10^15 bytes or a million gigabytes).

The bad news: disk access times are 100,000 times longer that those of DRAM.

So the change in access time from DRAM to disk is much, much larger than the change from caches to DRAM.

When looking at DRAM timing, we discovered that the additional access time for retrieving a contiguous block of words was small compared to the access time for the first word, so fetching a block was the right plan assuming we'd eventually access the additional words.

For disks, the access time difference between the first word and successive words is even more dramatic.

So, not surprisingly, we'll be reading fairly large blocks of data from disk.

The consequence of the much, much larger secondary-storage access time is that it will be very time consuming to access disk if the data we need is not in main memory.

So we need to design our virtual memory system to minimize misses when accessing main memory.

A miss, and the subsequent disk access, will have a huge impact on the average memory access time, so the miss rate will need to be very, very small compared to, say, the rate of executing instructions.

Given the enormous miss penalties of secondary storage, what does that tell us about how it should be used as part of our memory hierarchy?

We will need high associativity, i.e., we need a great deal of flexibility on how data from disk can be located in main memory.

In other words, if our working set of memory accesses fit in main memory, our virtual memory system should make that possible, avoiding unnecessary collisions between accesses to one block of data and another.

We'll want to use a large block size to take advantage of the low incremental cost of reading successive words from disk.

And, given the principle of locality, we'd expect to be accessing other words of the block, thus amortizing the cost of the miss over many future hits.

Finally, we'll want to use a write-back strategy where we'll only update the contents of disk when data that's changed in main memory needs to be replaced by data from other blocks of secondary storage.

There is upside to misses having such long latencies.

We can manage the organization of main memory and the accesses to secondary storage in software.

Even it takes 1000's of instructions to deal with the consequences of a miss, executing those instructions is quick compared to the access time of a disk.

So our strategy will be to handle hits in hardware and misses in software.

This will lead to simple memory management hardware and the possibility of using very clever strategies implemented in software to figure out what to do on misses.