CPS311 Lecture: Memory Hierarchies

Last Revised October 21, 2019

Objectives:

1. To introduce cache memory
2. To introduce logical-physical address mapping
3. To introduce virtual memory

Materials

1. Memory System Demo
2. Files for demo: CAMCache1ByteLine.parameters, CAMCache8ByteLine.parameters, DMCache.parameters, SACache.parameters, WBCache.parameters, NonVirtualNoTLB.parameters, NonVirtualWithTLB.system, Virtual.parameters, Full.parameters
3. AMAT Calculation example handout

I. Introduction

A. In the previous lecture, we looked at the basic building blocks of memory systems: the individual devices. We now focus on complete memory systems.

B. Since every instruction executed by the CPU requires at least one memory access (to fetch the instruction) and often more, the performance of memory has a strong impact on system performance. In fact, the overall system cannot perform better than its memory system.

1. Note that we are distinguishing between the CPU and the memory system on a functional basis. The CPU accesses memory both when fetching an instruction, and as a result of executing instructions that reference memory (such as `lw` and `sw` on MIPS).

2. We will consider the memory system to be a logical unit separate from the CPU, even though it is often the case that, for performance reasons, some portions of it physically reside on the same chip as the CPU.

C. At one point in time, the speed of the dominant technology used for memory was well matched to the speed of the CPU. This, however, is not the case today (and has not been the case - in at least some portions of the computer system landscape - for decades).

1. Consider the situation as it exists today.

    a) The dominant technology used for building main memories is dynamic RAM chips. However, DRAM chips have a typical access time of 60-80 nanoseconds, and a cycle time about twice that. Thus, if all instructions were contained in such memory, the rate at which instructions could be fetched from memory would be less than 20 million per second.

    b) But today's CPU's are capable of executing instructions at rates in excess of 1 billion per second - a 50 to 1 (or worse) speed mismatch! If a main memory based on DRAM were the only option, there would have been no reason to build CPU's faster than we had in the early 1990's!

2. With present technologies, it does turn out to be possible to build very fast memories (that can deliver instructions as fast as the CPU can execute them), but only of very limited size.

    For example: static RAM on the same chip as the CPU can function at the same speed as other CPU components. However, static RAM consumes a lot of power, since one side of the flip-flop for each bit is always on, And this generates a lot of heat; therefore, the amount of static RAM that can be put on the CPU chip is relatively small (typically < 100 KB).

3. OTOH, it is also possible to build very large memories, but using technologies that are quite slow compared to that of the CPU.

    For example hard disks can store 100's of GB of data for minimal cost. But hard disk is slow - a typical access time of about 10 ms, which is 10^7 times as long as a clock cycle on a 1 GHz CPU!

4. This speed versus capacity and cost tradeoff has been true throughout most of the history of computer technology, even though specific devices have varied. [If it ever became possible to produce memory that was both very fast and very large, this lecture topic would go away!]

D. We will see that memory systems are seldom composed of just one type of memory; instead, they are HIERARCHICAL systems composed of a mixture of technologies aimed at achieving a good tradeoff between speed, capacity, cost, and physical size.

1. The goal is to produce an overall system that exhibits the speed that is close to that of the fastest technology and the capacity of the largest technology.

2. This goal is achievable because, at any point in time, a program is typically only accessing a very small subset of the total memory it uses - a principle known as LOCALITY.

   a) TEMPORAL LOCALITY: most of a programs references to memory are to locations it has accessed recently.

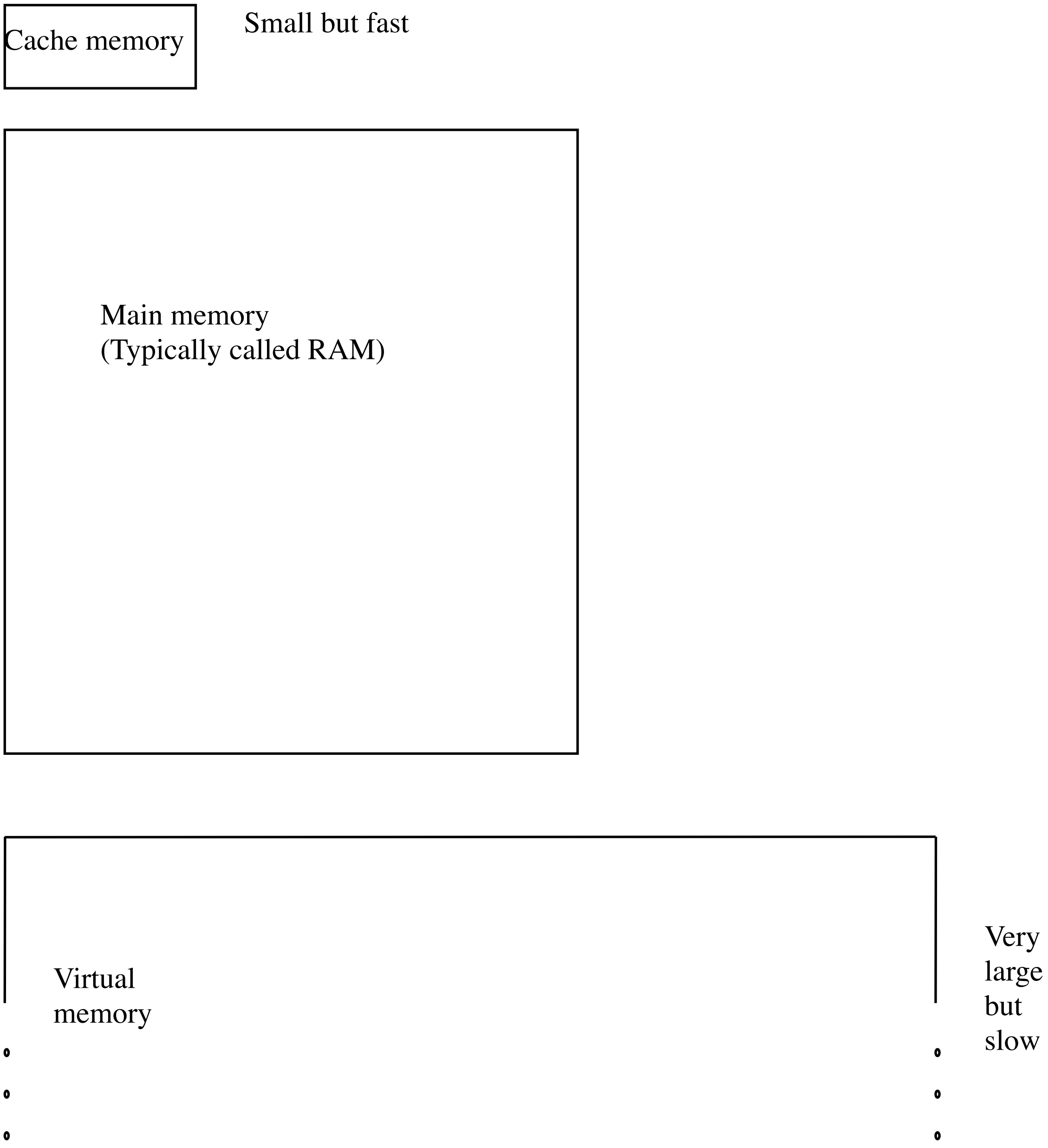
      This is because a program spends most of its time executing various loops. While executing a loop, the program keeps executing a relatively small group of instructions over and over, and typically accesses a small set of data items over and over as well.

   b) SPATIAL LOCALITY: if a program references a location in memory, it is likely to reference other locations that are physically near it.

      This arises because instructions occur sequentially in memory, and data structures such as objects and arrays occupy sequential locations in memory.

3. Thus, the instructions and data the program is currently using are typically small enough to be kept in the fastest kind of memory.

E. A typical computer system may have three basic kinds of memory, which constitute different levels in the hierarchy



1. Different technologies are used for each level.

    a) Main memory is typically implemented using DRAM

b) Cache memory is typically implemented using SRAM - sometimes on the same chip as the CPU; sometimes on separate chips.

Today's systems often have two or even three levels of cache, (called L1, L2, and sometimes L3 cache). (However, we will develop some of our examples using just a single level of cache for simplicity)

c) Virtual memory makes use of disk (hard disk or SDD).

2. Only one of these levels is actually necessary - main memory.

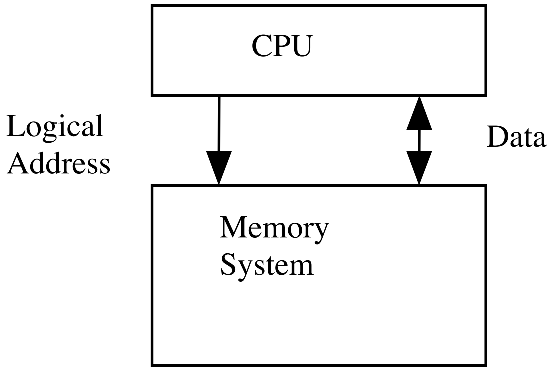
a) Historically, at one time personal computers needed only this.

b) Today, embedded systems often have only this.

c) In fact, from an interface standpoint, the memory system is made to look to the CPU as if it were entirely main memory. (Cache speeds things up, and virtual memory makes the memory system appear larger than main memory physically is - but the overall system looks like main memory to the CPU).

3. Each item has a logical address, whose size is dictated by the ISA of the CPU. (For example, if the CPU is 32 bit, then the logical address of an item is a 32 bit number; if it is 64 bit, then the logical address may be 64 bits or may be a smaller number like 48 bits).

The logical address, then, represents the location in main memory where the item exists (or would exist if main memory were large enough)



a) If the memory system consisted only of one level, then the physical address of an item in main memory would be the same as its logical address. (And a logical address which did not correspond to any physical address would be an error - e.g. if a certain system had only 1 GB of main memory, then any address greater than 0xf3ffffff would cause an exception.)

b) However, in a multi-level hierarchy, some addresses will not be present in all levels.

(1)For example, if a system uses a 32 bit address (corresponding to 4 GB of logical addresses), but it has - say - 128 MB of cache and 1 GB of main memory, then fully 3/4 of the possible logical addresses cannot correspond to any location in main memory, and 31/32 of the possible logical addresses cannot correspond to any location in cache.

(2)Thus, we must distinguish between the <u>logical address</u> of an item and the <u>physical address</u> where an item might be stored in some level of the hierarchy. Moreover, a given item that is present in multiple levels might be stored at different physical addresses in the different levels, though its logical address is the same in all three.

(3)A given item may reside in one, two, or three of the levels of the memory system. Typically, though the system is designed in such a way that, if an item is present in cache, it is also present in main memory; and often, if is present in main memory it is also present on disk as well.

Note that we sometimes speak of "moving" an item from one level of the hierarchy to another. This is really a misnomer - what is done is to copy the item, causing it to exist in both places.

c) It is also possible that a logical addresses may be totally invalid - i.e.it may not correspond to any item present at any level - in which case using it to try to access an item would result in an exception.

F. We speak of the series of memory references generated by the CPU as the reference string. The reference string consists of a logical address plus the operation begin performed (read or write)

1. Example: suppose we were executing a block of code like the following (which corresponds to `x ++` in C)

   ```
   lw $2, x
   addi    $2, $2, 1
   sw $2, x
   ```

   In this case, the CPU's reference string might look like the following (if the program were at memory locations 0x1000 and up, and x were at 0x3000)

   ```
   Read 0x00001000  # instruction fetch
   Read 0x00003000  # read original value of x
   Read 0x00001004  # instruction fetch
   Read 0x00001008  # instruction fetch
   Write 0x00003000 # write updated value of x
   ```

2. Observe that he majority of accesses to a memory system are reads. This is true because <u>every</u> instruction involves one read (to fetch the instruction), and load instructions (or their equivalent) are more common than store instructions. Typically, on the order of 80-90% of the accesses to memory are reads, rather than writes.

   a) When the CPU attempts to read an item - by specifying its logical address, the memory system checks to see if the item is in cache.

      (1)If so, the requested item is delivered from cache without checking the slower but larger levels. (If there are several levels of cache, each is tried in turn, starting with the fastest)

(2)If the desired item is not present in cache, but is in main memory, it is copied to cache (“bumping” some other item if necessary to make room) and is delivered to the CPU. (Often the delivery of the item to the CPU is done in parallel with the updating of cache).

(3)If the desired item is not present in main memory either, then it is either present on disk, or the address is invalid. If it is present on disk, it is copied to main memory (“bumping” some other item if necessary to make room) and also to cache.

b) Writes are a bit more complicated. Typically, an item is written to the fastest cache.

(1)If no item with that address is present, some item is “bumped” to make room for it.

(2)Eventually, an item that has been written to cache must also be written to main memory and to the swap file - but whether this is done immediately or at some future time varies (we’ll talk about this later). (If the cache itself has multiple levels, the item must eventually be written to all levels.) A consequence of the possibility of deferring a write to a lower level is that lower levels of the hierarchy may contain a “stale” version of some data item.

c) This strategy ensures that the items accessed most recently are present in the fastest cache, from which it will be fast to access them again (e.g. if the access occurs in a loop.)

d) Memory systems are typically designed to optimize reads, while not penalizing writes.

II. Cache Memory

A. One technique used to improve overall memory system performance is CACHE MEMORY.

1. At one time, cache memory was a feature generally found only in higher-end computer systems. However, as CPU speeds have continued to increase while memory speeds have not, cache memory has become a necessity on desktop and laptop computer systems as well.

   a) About 15-20 years ago, common CPU clock speeds were on the order of 4-16 MHz, and DRAM cycle times on the order of 70-80 ns. Under these circumstances, it would be possible to perform a memory access every 1-2 clock cycles.

   b) Today, CPU clocks have gone above 1GHz, while DRAM access time has improved only slightly, to about 60 ns. Thus, an access to main memory takes on the order of 60 clocks or more!

   c) Since a pipelined CPU is designed to execute one or more instruction per clock, and since each instruction must be fetched from memory, execution of one (or more) instructions per clock is critically dependent on the use of cache memory.

2. Cache memory is a small, high-speed memory, logically located between the CPU and the rest of the memory system.

   a) At one point in time, cache memory was usually separate from the CPU.

   b) Today's high-speed CPU's depend on having cache memory on the CPU chip that operates at the same speed as the CPU itself. This has been made possible by improved chip manufacturing techniques that allow more transistors per chip.

c) Actually, most systems now use a two or three level cache, some cache on the CPU chip and a larger, separate (and slower) secondary or tertiary cache.

d) It is also common to find - at least at the L1 cache level - that separate caches are used for instructions and data. This facilitates having separate paths to memory for the instruction fetch unit and the data memory access unit of a pipelined machine.

3. Cache memory works because of the phenomenon of locality of reference.

a) Each memory read is first tried against the cache. If the data is found there (a "hit"), the processor can proceed at maximum speed.

b) Otherwise, we have a cache miss and the processor must wait for a slower access to secondary cache or (if there is a miss there too) to main memory.

4. To function effectively, a caches must hit most of the time. The percentage of memory references that are found in the cache at any level - rather than going to the next level - is called the HIT RATE. To see why this is important, suppose we have a 2 GHz CPU.

a) Theoretically, the time to execute an instruction (if the CPU is fully pipelined) is as little as a fraction of a ns. It must be able to reference memory (to fetch and instruction or read/write a data item) in that amount of time. (In our examples below, we will assume a time of 0.5 ns.)

b) Suppose, however, that main memory requires 100 ns to do an access (including bus overhead). Suppose further that there is a a single, on-chip cache that has a hit rate of 90%. Then 90% of memory references can be done in 0.5 ns, but 10% require 100 ns. (We call 100 ns the miss penalty) So the average time per reference becomes

0.9 x 0.5 ns + 0.1 x 100 ns = 10.45 ns

which is equivalent to reducing the clock rate by a factor of over 10!

c) With a 95% hit rate, the average time drops to

0.95 x 0.5 ns + (0.05) x 100 ns = 5.48 ns - almost twice as good, but still not close to what we would expect knowing only the clock rate.

5. Multi-level caches are useful when - as is the case here - there is a wide variation between speeds of the different levels of memory

   For example, if we add a 10 ns secondary cache that hits 90% of the primary cache misses, we get an average reference time of

   0.95 x 0.5 ns + (0.05) x (0.9) x 10 ns + (0.05) x (0.1) x 100 ns = 0.48 ns + 0.45 ns + 0.5 ns = 1.43 ns

   (which is still like more than cutting the clock speed in half)

6. As these examples show, CPU clock rate alone tells you little about overall execution speed without some knowledge of how the caches perform!

B. In principle, a cache is an a content addressable (associative) memory containing pairs of the form:

tag (location in main memory)	value

1. A content addressable memory is one in which items are looked up based on their content, rather than the location where they reside. In a cache, the tag represents the logical address - i.e. the location in <u>main memory</u> where the entry resides - which generally has nothing to do with the location in cache where it resides.

DEMO: Load CAMCache1ByteLine.parameters

Access 1000, then 1100, then 1000 again. Note how second access is much faster

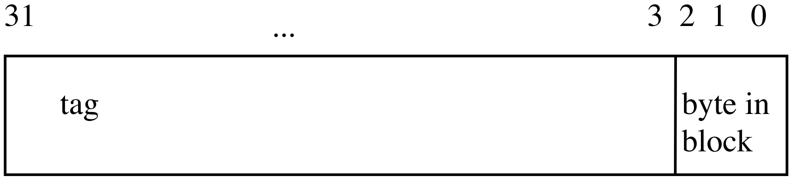
Note that we do not attempt to lookup entries based on their location in the cache - e.g. we do not need to support an operation such as “what is the entry in slot 2?” - rather we look up entries based on the stored tag value.

2. Usually, the “value” part of a cache entry holds more than just one byte, even on a byte-addressable machine. To see why, consider a system that uses a 32-bit logical address. Each cache entry, then, would need a 32 bit tag. If each entry held just one byte of data, then the total size of an entry would be 40 bits - of which just 20% would be useful data and 80% would be overhead.

   Instead, it is common to have each entry in the cache contain several successive bytes - called a line or block. For example, many systems use a cache based on entries holding 8 consecutive bytes of data.

   a) In this case, the tag would consist of the upper 29 bits of the logical address. (The lower 3 bits are implied by the position of each byte in the line.)

   b) A logical address generated by the CPU could be interpreted by the cache as follows:



3. Another benefit of using a multi-byte block size in the cache is that it allows a memory reference to a word to be done in one operation.

DEMO: Load CAMCache8ByteLine.parameters

DEMO Accessing (as a word) 1000, then 1004

(Actually, for Level 2 cache it is common to use very large block sizes - e.g. 128 bytes. This is done to take advantage of spatial locality. When a reference to an item not present in Level 2 cache occurs, an entire block of (say) 128 bytes is copied from main memory to cache - including the desired item and many of its neighbors - in the hope that one or more of the neighbors will be needed soon and will already be in the cache. (Note: the transfer of data from main memory to the cache may require multiple bus cycles, but this is actually done in parallel with the computation using the originally-requested item))

4. Caches are designed to support reads. Writes may still involve an access to main memory.

   a) If the location to be written is currently in the cache, it can be updated there - but some provision for updating main memory must also be made. The simplest strategy is called write through.

      DEMO: Writing (as a long) the content of 1000.

      However, in practice it is often possible to do the write through in parallel with the original operation (and subsequent operations that do not write through).

      (Note, in demo, how write through time is reported separately)

   b) When writing through, an entire line must be written.

      DEMO: Writing (as a byte) the content of 1000

   c) If the line is not in the cache, and an entire line is being written, it can simply be added to the cache and written through to main memory

      DEMO: Writing (as a long) the content of 1010

d) But if less than a full line is being written, and the line is not in the cache, it is actually necessary to first get the full line, then change the desired portion, then write the full line back - since data is transferred to/from cache in full-line units only.

DEMO: Writing (as a byte) the content of 1020

In this case, it apppears that the use of the cache actually increases the time needed for the operation, since the fetch of the original line must be done <u>before</u> the cache can be updated, and then an additional write through must also be done afterward. However, if the write operation can be completed in parallel with further computation, this need not create a delay.

C. In principle, a cache is a large content-addressable memory as in the examples; however, a large fully content addressable (associative) memory is impractical. Algorithms such as binary tree search are useless for cache since we must get the answer in one step. For example, in a cache of 10,000 entries, the tag portion of each address emitted by the CPU would have to be compared to the tags of all 10,000 entries at the same time. Even if the required number of comparators could be economically built, the incoming address would have to drive 10,000 logic loads. This would require several layers of buffering (since a typical gate output can drive about 10 others), which would inject intolerable delays. Therefore one of several approximations to associative memory is used.

D. Direct mapping cache.

1. Entries in the cache are triples of the form

   valid   tag   value (typically 8-128 bytes - assume 8 for this example)

   where valid is a single bit that is true (1) if the entry contains valid data and false (0) if it does not

2. The number of entries is always a power of 2. Suppose it is 2^n.

3. Consider the entries to be numbered 0 .. 2^n - 1. Then a logical address (assume 32 bits) is interpreted by the cache as follows:

31	n+3 n+2	3 2 1 0
tag (32-n-3 bits)	entry number (n bits)	byte in block

that is, any given logical address, if it occurs in the cache at all, must occur in entry number (logical address / 8) % 2^n.

4. When it is desired to look up a logical address to see if it is present in the cache, an entry number is calculated as (logical address / 8) % 2^n. This one entry is checked.

   a) If the tag stored there matches the tag portion of the desired address, and the valid bit is set, then there is a cache hit, and the value in the entry contains the data sought.

   b) If the entry is invalid, or the tags do not match, then there is a cache miss. In this case, a complete block of data is obtained from memory. It is stored (along with the tag portion of its address) in the cache for future use, replacing the entry currently there.

5. A consequence of this scheme is that the address in main memory that corresponds to some entry in the cache is determined by combining its tag with the location in the cache where it occurs - e.g. if a cache of 64 entries uses 8-byte blocks, and the entry in slot 2 has tag 18, then its value corresponds to the contents of main memory locations:

```
00000000000000000011000 000010 000 ...
00000000000000000011000 000010 111

 = 00003010 .. 00003017
```

DEMO: DMCache.parameters - access byte at 3010, 3017

6. Of course, this scheme implies that at any time at most one entry with any given pattern in its n entry-number bits can be in the cache.

   This is usually not a problem, because successive locations in memory map either to different bytes in the same block, or to different entries in the cache - e.g.

   Suppose we have a direct mapping cache like the above. Then:

   logical addresses 0x0 .. 0x7 map to entry 0 of the cache
   logical addresses 0x8 .. 0xf map to entry 1 of the cache
   logical addresses 0x10 .. 0x17 map to entry 2 of the cache
   ...
   logical addresses 0x3f8 .. 0x3ff map to entry 63 of the cache

   DEMO Accessing 0x0, 0x3f8

   So, unless we're executing a very large loop, all of the code for the body of a given loop can easily fit in the cache without collisions (especially given that caches are typically a lot bigger than this demo!)

7. However, not only do logical addresses 0x0-0x7 map to entry 0 of the cache, but so do 0x400 .. 0x407, 0x800 .. 0x807 ...

   DEMO Accessing 0x400

   a) It is easily possible that an instruction in a loop might collide in the cache with some data item accessed by the loop of which it is a part (i.e the code and data differ in address by some multiple of <total cache size>.) (This can be avoided by using separate caches for instructions and data, as is often the case with Level 1 caches, though not with other levels).

   b) Or suppose a loop calls a procedure whose address is some multiple of <total cache size>. away from the loop body, or if there are accesses in a loop to two data structures whose addresses are some multiple of <total cache size>. apart. Such problems cannot be avoided by using separate instruction and data caches.

c) In a case where there is a collision like this, the replacement policy of the cache could actually hurt performance - e.g. if we alternately reference two items that collide, we might get something like this:

Item A referenced - moved to cache
Item B referenced - replaces A in cache
Item A referenced - replaces B in cache
...

DEMO: Alternate accesses to 0x0, 0x400

E. Set Associative Cache

1. This is an improvement on direct mapping. It can address the problem we just discussed.

2. The cache entries are divided into sets - typically involving 2 or 4 entries. (Note that this means that the number of sets is - say - 1/2 or 1/4 the total number of entries.) Each entry has the same form as before.

3. A memory address is interpreted as follows. (The example assumes a 2-way set associative cache with 2^n entries and hence 2^{n-1} sets.)

31	n+2 n+1.	3 2 1 0
tag (32-n-2 bits)	set number (n-1 bits)	byte in block

a) The middle bits of the logical address select not one entry in the cache, but a set of entries. The tags for each entry in the set are compared in parallel with the desired logical address, and if one matches (and is valid) there is a hit. (Parallel comparison of a few tags does not pose the kind of problem that made pure content addressable memory impractical).

b) When a reference is not found in the cache, one of the entries in the set must be replaced. This is may be done either in first-in first-out fashion, or in least recently used fashion, or randomly (which actually works fairly well!).

DEMO: SACache.parameters - accesses to 0x0, 0x400, 0x0

4. An issue in the design of set associative caches is set size

a) A set size of two is commonly used, because it is simpler to build.

b) A set size of four has been found experimentally to give marginally better hit/miss performance. Note that going from a set size of 2 to 4 - for a given total size cache - results in there being half as many sets, and hence twice the possibility of collision between two items. However, it takes 5 items colliding before replacement gets in the way.

c) Experimental evidence suggests set sizes greater than 4 produce no significant gain.

F. Issues with regard to caches

1. Validity of cache items.

a) When the system is first started up, or when there is a change of user in a multiprogrammed environment, the cache will not contain valid data until a sufficient number of reads have been done. At such times, the valid bit for each entry in the cache is cleared, to be set later when an entry is copied from main memory.

Of course, in a set associative cache each member of a set must have its own valid bit

DEMO: Highlight this aspect

b) There may be a provision for the operating system to invalidate cache entries wholesale when a context change to a new user is done.

2. Write-through versus write-back

a) We've already looked at one approach to the problem that arises when a memory access is a write rather than a read? In a write-through cache, the data is written both to main memory and to cache at the same time. This slows the system down some (though the CPU can go on to the next instruction while the memory write occurs). The slowdown is not drastic since writes are proportionally rare.

b) An alternative strategy is called write-back. In a write-back cache, the data is written only to the cache. Each cache entry includes a "written in" (dirty) bit, which is set to indicate that the cache entry contains a more recent value than main memory.

When a cache entry is selected for replacement by a new entry, it is then written to main memory, if the written in bit is set. This avoids doing multiple writes to a single location in main memory.

DEMO: WBCache.parameters. (This one is direct mapping again)

Write 0x0 - note no write-through, but cache line is now dirty.

Now write 0x400 - note write back.

c) Of course, with write back a cache line must be also be written back if it is invalidated by the operating system during a context change to a new user or as part of bringing in a new page, as we shall discuss soon.

d) There is a potential problem if a DMA IO device is to access data that has not yet been written back. This can be handled by forcing the cache to be flushed to main memory before a DMA operation is initiated.

G. Caching as a general principle in CS

1. We have been discussing the concept of caching in terms of memory systems - which is where the idea originated.

2. But the same principle is used in many places in computer systems - i.e. the idea of keeping a copy of something you expect to access again soon in a place where it can be accessed more quickly. Examples?

   ASK

   a) Caching of web pages by a browser

   b) Caching of web pages by an ISP

   c) Caching of disk blocks by a disk controller

3. A problem that arises from caching wherever it is used is the possibility of having the cached copy of an item be inconsistent with the "official copy". (This occurs if another system has write access to the "official copy".

   a) In the case of memory caches, problems can arise if physical memory can be written by more than one device - e.g. by the CPU and a DMA controller, or by two CPUs. In this context, it is known as the cache coherency problem.

      One approach to the cache coherency problem is to use what is called a "snoopy cache" - which is simply a cache that monitors the address lines for the memory it is caching and invalidates an entry it contains if the corresponding location is written to by some other device.

      (We will not discuss this topic further here).

b) Other places where the problem can occur?

ASK

Example: The possibility that an update to a web page may not be seen because stale copies are cached by a browser or ISP.

(This creates an interesting issue when developing code contained on a page - e.g. JavaScript)

III.Memory Mapping

A. Consider a computer system where two or more programs are running at the same time - perhaps two programs serving the same user, or programs serving multiple users. Obviously, each program must have its own memory separate and distinct from the other programs. How can this be accomplished?

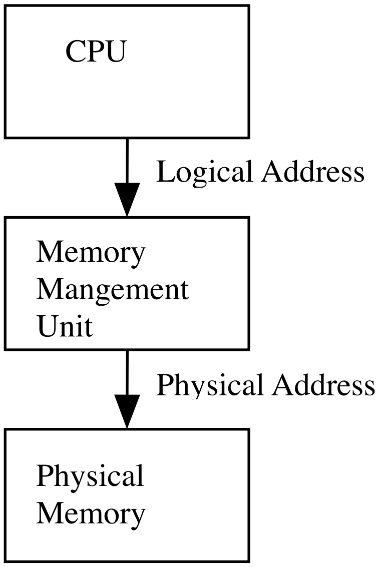
1. The earliest solution was to require each program that coexisted with others to use different memory addresses. This was of limited utility.

a) Most viable in cases where there was a fixed set of programs that would be running all the time, each of which was assigned to some fixed region of memory.

b) Not at all useful for a multi-user system, where several people might run the same program on different data.

Example: in various labs, all of you have been using a single mips system, and all of your programs have used the same addresses, but obviously writes by one program did not affect a different program.

2. The solution that has proved viable for many years is to distinguish between <u>logical addresses</u> generated by a program and <u>physical addresses</u> in memory, and to insert a hardware memory management unit (MMU) between the CPU and the rest of the memory system that performs translation between one and the other.



a) In a situation like this, we can distinguish the LOGICAL ADDRESS SPACE and the PHYSICAL ADDRESS SPACE.

(1)The logical address space is the range of logical addresses that the CPU can generate, dictated by the ISA.

Example: For a 32-bit CPU, the logical address space is generally 0 .. 0xFFFFFFFF.

(2)The physical address space is the range of physical addresses, dictated by the amount of physical memory actually installed.

Example: If a system has 1 GB of physical memory, its physical address space may be 0 .. 0x3FFFFFFF

(3)In general, the two address spaces will not be of the same size.

b) With this arrangement, it becomes possible for all programs to be created as if they used the same range of logical addresses; but the memory management unit would map the same logical address to different physical addresses for different programs/users.

c) A by-product of this is that it provides an easy way to protect users from one another; if user A's program uses some physical address, but no logical address in user B's program maps to that physical address, then nothing that user B's program can do can have any effect on user A.

d) Another by-product of this is that it is not necessary for physical memory to consist of contiguous addresses.

Memory modules are generally designed in such a way that, if the module contains 2^n bytes, then the physical addresses it corresponds to must be of the form xxxx (n zeroes) .. xxxx (n ones) - where the upper part of the address (xxxx) is the same for all locations in the module. It a memory system is constructed of modules of two different sizes, this can make assigning contiguous addresses difficult.

Example: suppose a memory system consists of a 1 GB module. Presumably, this would be configured to correspond to physical addresses 0 .. 0x3FFFFFFF Now suppose the user adds a 2 GB module. What addresses should be assigned to this module?

The contiguous addresses would be 0x40000000 .. 0xBFFFFFFF. But this would not fit the preferred way of assigning addresses to a module. (The upper bits of the address are 00 for some addresses, 01 for others, and 10 for still others.)

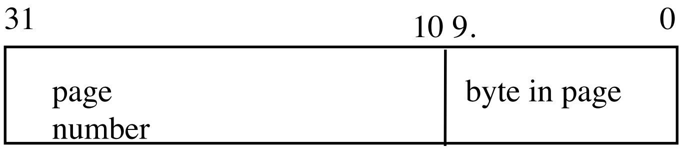
It would be better to leave a hole in the physical address space and assign the new module the addresses 0x80000000 .. 0xFFFFFFFF.

Memory mapping makes this possible.

B. There are a number of ways to actually carry out this mapping. We'll only discuss one in detail - called PAGING.

1. The logical address space (range of logical addresses) is divided into pages of some fixed, power of 2 size.

   Example: if a byte-addressable system uses a 32 bit address, its logical address space is 0 .. 0xffffffff = 4 GB. Suppose we use a page size of 1 KB. Then the logical address space can be thought of as being composed 4 mega (2^{22}) pages of 1024 bytes each. A logical address generated by the CPU is interpreted as follows:



2. Physical memory is divided into page frames, each of which is of the same size as a page. Thus, any logical page can be stored in any physical frame.

3. The memory management unit uses a page table, with one entry per page. Each entry contains a frame number where the page is mapped, plus a valid bit (false means this logical page is not actually mapped and an address that references it is illegal)

   Example: Given the following page table:

Page number	Valid	Frame Number
0	0	N/A
1	1	7
2	1	5
3	1	12
4	0	N/A
5	1	8
...		

   The logical address 0x82A is interpreted follows:

   0000 0000 0000 0000 0000 10 00 0010 1010

   page number = 2   byte on page = 0x2A which is mapped to frame 5 by the page table, meaning that the corresponding physical address is

```
0000 0000 0000 0000 0001 01 00 0010 1010
```

i.e. `0x142A`

4. The memory management unit makes use of distinct page tables for each process (program running on the behalf of some user). Thus, while two different processes might generate the same logical address, the page table would generally specify a different mapping to a physical address for each. (Some systems allow sharing of common libraries between processes so as to only have one copy residing in memory.)

C. Some issues with regard to paging

1. In addition to the frame number and a "valid" bit, the page table entry may also contain bits specifying protection for the page - e.g. a page table entry may specify that a certain page is read only [as might be the case if the same physical page is shared by two or more processes].

2. Where is the page table stored?

   a) If the number of pages is small (memory is small, page size large), it may be possible to keep the page table in special registers in the memory management unit. (This has been done on some systems)

   b) More typically, the page table is itself kept in main memory.

      (1)In this case, the MMU contains a single register that contains the physical address of the start of the page table in memory.

(2)The MMU performs a logical to physical address translation as follows:

(a)The page number is extracted from the logical address.

(b)The page number is multiplied by the size of a page table entry (often 4 bytes - in which case shifting left 2 places can be used)

(c)The result is added to the base address of the page table, contained in an MMU register.

(d)The resultant location in memory is read, and is used as a page table entry to calculate the physical address corresponding to the original logical address [or to cause an exception to be thrown if not valid].

DEMO: NonVirtualNoTLB.parameters - Read 1000

c) In this case, a context switch from one process to another requires simply loading this register with the address of the page table for the new process.

3. With a large logical address space, the page table itself can be very large - for example, with a 4 GB logical address space and a page size of 1 KB, the page table needs 4 million entries. If each entry is 4 bytes, that requires 16 MB for the page table!

a) To avoid this, it is common for the MMU to also include a "limit" register which specifies the actual size of the page table. When the MMU performs a mapping, it first checks the page number extracted from the logical address against the limit register. If the page number is too large, the address is treated as invalid.

b) There are now two registers in the MMU that are part of the context of a process - the base address of the page table, and its size.

DEMO - Read 4000

c) It would seem, at this point, that we may have eliminated the need for a “valid” bit in each page table entry. But actually, we haven’t - for two reasons:

(1)There might be a good reason for having a page number that is in the allowed range still be invalid.

Example: it is quite common to find that page 0 - though obviously in range - is made invalid, with no physical memory assigned to it. Why?

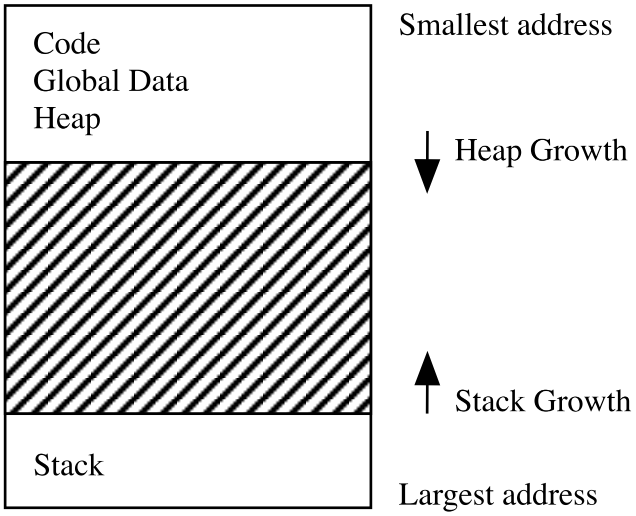
ASK

Page 0 corresponds to logical addresses in the range 0 .. page-size - 1. If we use 0 as a null pointer, then any attempt to dereference null will result in a memory management error. Thus, by the expedient of “wasting” one entry in the page table (e.g 4 bytes), we get automatic hardware detection of attempting to use a null pointer.

DEMO: Read 0

(2)Virtual memory will also make use of the ability to flag pages as invalid, as we shall see shortly.

4. Now we get another problem, though. We have previously said that process address space is often structured this way:



a) But if we have a limit register whose value is based on the number of pages in low memory, then any logical address on the stack is necessarily invalid.

b) To handle this, it is possible to actually have two page tables per process - one used for mapping addresses in code/global data/ heap space, and one used for mapping addresses in stack space.

(1)The MMU uses a high order bit in the logical address to decide which one to use

(2)The limit register for the first page table specifies a <u>maximum</u> permissible page number; that for the second specifies a <u>minimum</u> permissible page number.

5. It is also possible to have an additional page table for use by system software.

6. An interesting consequence of using address mapping is that the size of logical address space and physical memory can differ.

a) For example, with a 64 bit logical address on a byte addressable 64-bit CPU, it is possible to specify 2^{64} (16 quintillion) different addresses - far beyond the amount of physical memory such a system has now, or will be likely to have in the forseeable future.

(1)Actually, throughout much of computer system history, it has been the case - for most systems - that physical memory has been smaller than the range of permissible logical addresses.

(2)This is not a problem on a system using memory mapping - the maximum permissible size of memory mapped by all the page tables is limited to the amount of physical memory actually installed.

b) There have been cases where the reverse has been true: physical memory has been larger than the size dictated by the size of a logical address. (E.g. this was often the case with minicomputers using a 16 bit logical address). This is not a problem - the frame number can be larger than a page number, allowing the generation of a physical address that is longer than a logical address.

(1)Of course, no one process could generate enough different logical addresses to access all of physical memory.

(2)However, on a multi-user system, the sum total of all the memory allocated to all the processes might make use of all the physical memory.

D. We'll note briefly that there are a couple of other ways to do memory management other than paging.

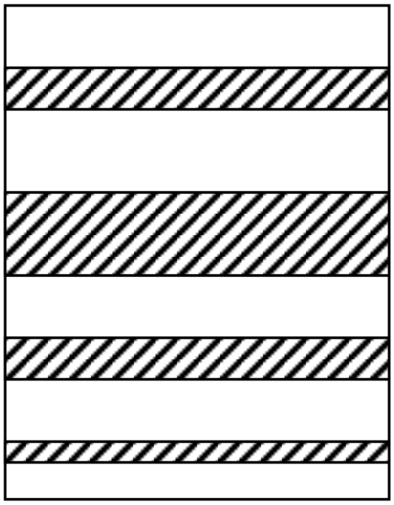
1. One approach is called SEGMENTATION. It differs from paging in that, while paging uses pages and frames of some fixed size, segmentation uses segments of variable size.

a) A maximum segment size is established and can be used to split a logical address into a segment number and offset in segment.

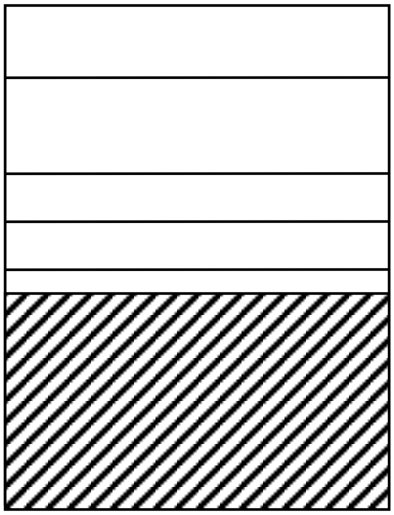
b) A segment table (which is analogous to a page table) stores, for each segment, its base address in physical memory plus its size.

c) Mapping a logical to a physical address involves looking up the mapping information for the segment and adding the offset in segment to the base address for the segment - after first ensuring that the offset in segment does not exceed the segment size.

d) Pure segmentation has never worked well - because varying sized segments tend to produce "checkerboarding" of physical memory - e.g. one could end up with a situation like this, where the hashed regions represent memory that is unused.



It is quite conceivable that a process might need a segment which, while smaller than the sum total of all available memory, is still bigger than any one available chunk of memory. This could necessitate a costly compaction of memory to move all the free space to a single block.



e) The Intel x86 chip family provides support for segmentation (with segment sizes limited to 64 KB), but modern versions of Windows do not use it.

2. Another approach is called SEGMENTATION WITH PAGING. In this approach, <u>each</u> segment has its own page table which is actually used for mapping logical addresses to physical addresses.

a) A logical address consists of a segment number, page in segment, and offset in page.

b) A logical address is mapped by using its segment number to locate the correct page table, and then mapping the rest of the address as with paging.

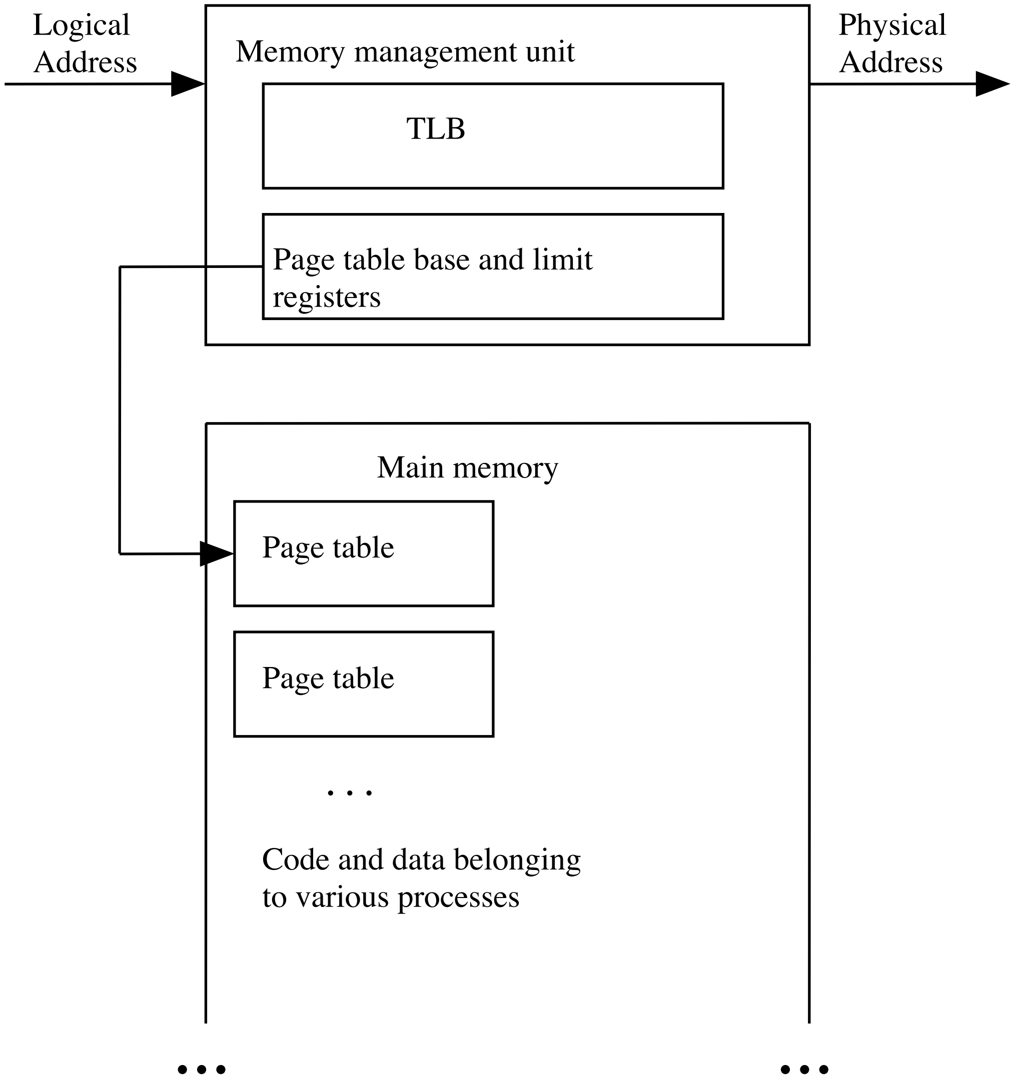
c) Variable segment size is handled by using different size page tables for each segment.

E. If you have been thinking carefully about what we've said about mapping, it has probably occurred to you that, if the page and/or segment table is itself kept in memory, then any attempt to access main memory actually requires two references - one to look up the translation, and one to do the actual access.

1. This would have the effect of halving the speed of main memory - not good.

2. The situation could be even worse when using segmentation with paging, which could require three references to memory (segment table, page table for the proper segment, actual location in memory)!

3. To reduce the impact of this problem, most virtual memory systems make use of a special kind of cache called a TRANSLATION LOOK-ASIDE BUFFER (TLB) which stores page table entries that have been used recently. That is, a configuration like the following is used:



a) When a logical address needs to be translated, the page number is extracted, and then the TLB is checked to see if it contains the translation (physical frame number) for this page. If it does, the translation is done, and only the actual reference to the page in main memory is needed.

DEMO: (Note: complete system - not just parameters)
NonVirtualWithTLB.system - read 1000

b) If the TLB does not contain the translation, the page table in main memory is accessed to obtain the translation. At the same time, a copy is stored in the TLB for future use (replacing some other entry).

DEMO: Read 2000

c) Because pages typically contain multiple instructions / data objects, it is usually the case that most of a program's references to memory in an interval of time involve just a few distinct pages, and therefore most of the needed translations can be found in TLB.

d) In effect, a TLB serves as a cache for <u>page translations.</u> Note that a TLB can be quite small - therefore, it may be practical to implement it as a fully content addressable (associative) memory, rather than using one of the techniques like direct mapping or set associative that we used for cache per se.

IV. **Virtual Memory**

A. Any of the memory mapping schemes we have discussed can be easily extended to yield VIRTUAL MEMORY. We'll use paging for our examples.

1. In a paging system, when the MMU attempts to translate a logical address, one of the following outcomes will occur:

a) A valid translation

b) The page number is outside the range of valid pages - i.e. there is no entry in the page table at all corresponding to that page number.

c) The address translates correctly, but the access violates the protection specified in the page table (i.e. attempting to write to a read-only page).

d) The page is invalid - i.e. no physical memory is actually allocated for that page.

Any one of b-d is an error which typically results in an exception that terminates the offending process.

2. Now suppose - as is sometimes the case - that the sum total of the memory needs of all the processes on the system exceed the amount of physical memory available. In this case, we cannot assign a frame to every page.

   a) However, it is typically the case that many of the pages that a program uses are either never used at all (e.g. specialized error handling code) or are not needed at some point during execution of the program (e.g. initialization code is not needed once the program gets into mainstream execution; termination code is not needed until the program finishes.)

   b) Therefore, we could adopt the expedient of only assigning physical frames to some of the pages needed by a program, and flagging the page table entries for the rest as invalid. (Sometimes we may refer to this bit as “resident” to reflect this change in interpretation, but the hardware behavior hasn’t changed.) Copies of pages that are legitimate but not currently resident in memory can be kept on disk.

   c) Now, if the program accesses a page that is inside the range of valid page numbers, but marked not present, there are two possibilities:

(1)The reference really is invalid, and this should be treated as an error.

(2)The reference is to a page that would be valid if we had enough physical memory - and hence, there is a copy of the desired page on disk.

This latter case is called a PAGE FAULT.

d) In the case of a page fault, we can resolve the problem as follows:

(1)Find an available page frame in physical memory - perhaps by “bumping” the page that is currently in to disk. Moving a page to disk to make room for another is called PAGE REPLACEMENT.

(2)Copy the desired page from disk to this frame.

(3)Modify the page table to show the correct mapping, and set the resident bit to true.

(4)Re-start the instruction that failed.

(5)Note that this is a fairly complex and time-consuming process, especially given that disk accesses take on the order of 10 ms (about 100,000 times as long as main memory accesses).

Thus, it is typically performed by software - specifically by code that is part of the operating system. During the time that a process is unrunnable due to waiting for a needed page, some other process can be run instead.

(This stands in contrast to reading data from main memory into a cache, which must be handled by hardware because the time frame needs to be very short).

DEMO: Virtual.parameters - access 1000, then 1010

3. From a hardware standpoint, virtual memory is not terribly different from simple mapping.

   a) From a hardware standpoint, a page fault may look just like an invalid page. The operating system decides what is actually the case.

   b) Since the page table entry for an invalid or not resident page is mostly unused (the resident bit says everything that needs to be said), the operating system may use these bits to store information it needs to handle a page fault - e.g. an indicator as to where to find the page on disk - or an indication that the page really is invalid.

   c) Virtual memory does impose some additional hardware requirements.

      (1)The ability to re-start an instruction that failed due to a page fault after it has been resolved.

      (2)Several other abilities we will discuss shortly.

4. From a terminological standpoint, what we have been calling a "logical address" is often called a "virtual address" when virtual memory is used.

B. The performance of a virtual memory system is critically dependent on the frequency of page faults.

1. The rate of page faults is called the page fault rate. It is vital that this be kept very low.

2. To see how important this is, recall the example we did earlier concerning a memory system with a two-level cache having the following parameters

Memory access time for level 1 cache: 0.5 ns
95% hit rate for level 1 cache
Memory access time for level 2 cache: 10 ns
90% hit rate for level 2 cache (that is, 90% of the 10% of references that are tried in Level 2 at all)
Memory access time for main memory: 100 ns

If virtual memory were not used (and hence there were no page faults), the average access time for this memory would be:

0.95 x 0.5 ns + (0.05) x (0.9) x 10 ns + (0.05) x (0.1) x 100 ns = 1.43 ns

Now suppose we use virtual memory with the following parameters:

Access time for disk: 10 ms (recall that 1ms = 10^6 ns)
Page fault rate 1% (1% of the accesses to main memory result in a page fault)

Now the average memory access time is

0.95 x 0.5 ns + (0.05) x (0.9) x 10 ns + (0.05) x (0.1) x (0.99) x 100 ns + (0.05) x (0.1) x (0.01) x 10 x 10^6 ns = 1.42 ns + 500 ns = 501.42 ns

almost a 500 fold increase over the time without paging!

3. On the other hand, if we can get the page fault rate down to .001%, we can get an average memory access time of

   0.95 x 0.5 ns + (0.05) x (0.9) x 10 ns + (0.05) x (0.1)x(0.99999) x 100 ns + (0.05) x (0.1) x (0.00001) x 10 x 10^6 ns = 1.42 ns + 0.5 ns = 1.92 ns

4. Several factors contribute to a low page fault rate

   a) Appropriate selection of the page to be replaced when a frame needs to be replaced is critical. We will discuss this below.

   b) Paging systems take advantage of spatial locality by transferring data between disk and memory in units of whole pages. This means that, if there has been a page fault for some item, other items physically near it (e.g. successive instructions or different fields in the same data structure) will likely also be brought into memory and will not themselves cause a page fault.

This, in turn, argues for a larger page size where possible.
[Typically, the page size is never greater than the block size of the disk - we don't want to have to do multiple disk accesses to transfer a page!]

c) The scheme we have described is called DEMAND PAGING - a page is brought into memory when it is demanded. It is also possible to do some ANTICIPATORY PAGING - bringing a page into memory before it is actually needed because it is expected that it will be needed.

(1)If the blocks are clustered on the disk in such a way as to allow multiple blocks to be transferred in virtually the same time as one, several pages may be brought in from disk at once to take further advantage of spatial locality.

(2)When a program first starts up, its code pages may be loaded enmasse, rather than being faulted for individually.

(Actually, if this is not done, the system software can get the pages upon demand directly from the executable image on disk, rather than first transferring them to the paging file.)

(3)Anticipatory paging is really a software issue, rather than a hardware issue, though.

d) A virtual memory system can run programs whose total memory requirements exceed the amount of physical memory available. The fact that the memory requirements of all current processes exceeds the available memory is called OVERALLOCATION.

Empirically, at some point excessive overallocation can lead to a situation in which pages are replaced and then needed again soon - which leads to a high page fault rate and poor performance. This condition - called thrashing - can result in a system becoming painfully slow.

To prevent thrashing, a system may totally swap out of memory a process if necessary to keep the overallocation for current processes within bounds.

C. Issues in the design of a virtual memory system

1. In our discussion of the resolution of a page fault above, we talked about the need for page replacement - periodically moving a page out of memory to make room for another.

a) The choice of the page to replace will have a significant impact on the overall performance of the system. If a page is replaced and then is needed again, it must be reloaded into memory from disk - a costly operation.

b) It can be proved that the optimal page replacement policy would be to replace the page whose next reference is furthest in the future.

Of course, if there is a page that is never going to be referenced again (e.g. it contains code that won't be executed again), then replacing it would be optimal by this criterion (next reference infinitely far in the future).

c) Unfortunately, the optimal policy is generally not practical, since we don't have the necessary knowledge. Instead, a page replacement algorithm is chosen that gives the best possible approximation to optimal performance.

(1) There are many possible page replacement algorithms. This is actually a topic in operating system design we cannot pursue here.

(2)However, we can note that many of the algorithms rely on the page table incorporating an additional bit called the "referenced bit" in each entry. (The textbook calls this the "used" bit - same meaning - different term) When a mapping is performed, if the page table entry has its referenced bit off, it is turned on. (If it is already on, it is left on.) This bit provides information about recently used pages that various algorithms use because they rely on the principle of temporal locality to help them make a good choice.

2. Virtual memory systems always use a "write-back" policy, in the sense that writes are done to a page in memory, but are not done immediately to disk (since the time overhead of doing so would be huge).

   a) A consequence of this is that we need to keep track of whether a page in memory has been modified by one or more write operations since it was copied from disk, so that the updated copy can be written to disk when the page is replaced by some other. (If a page has not been modified since it was read from disk, there is no reason to write a copy back, given the computational cost of doing so.)

   b) For this reason, the page table entry will contain a bit sometimes called the "modified bit" or the "dirty bit" which is set false when a page is copied in from disk, but true by the memory management hardware when a write operation is done to any location on the page.

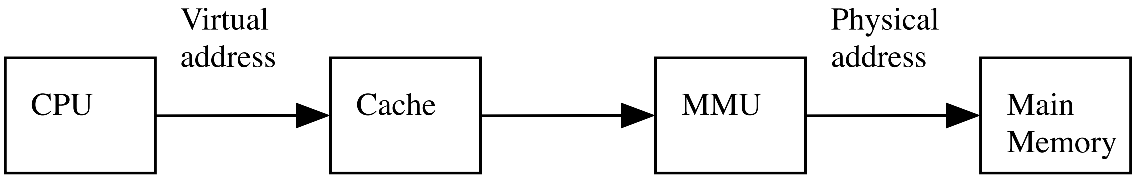
   c) Page replacement algorithms may give preference to replacing a page that is not dirty, rather than one that is, because the latter requires a write operation. Of course, such a preference cannot be absolute - otherwise, a dirty page will never be replaced, which would not be a good idea if the page was not needed again.

V. **Putting it All Together**

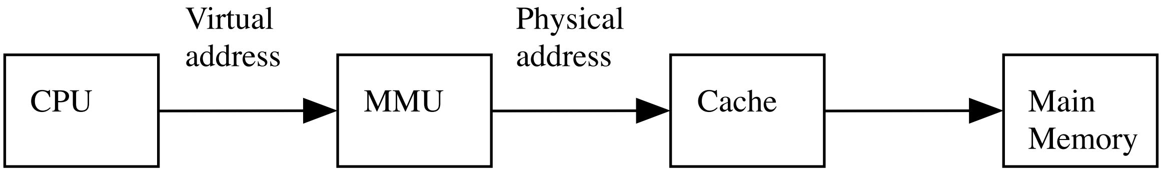
A. Of course, almost all current computer systems use both caching and memory mapping.

1. One question that arises concerns the relative placement of the cache(s) and the MMU.

a) One possibility:



b) Another possibility:



c) Pros and Cons

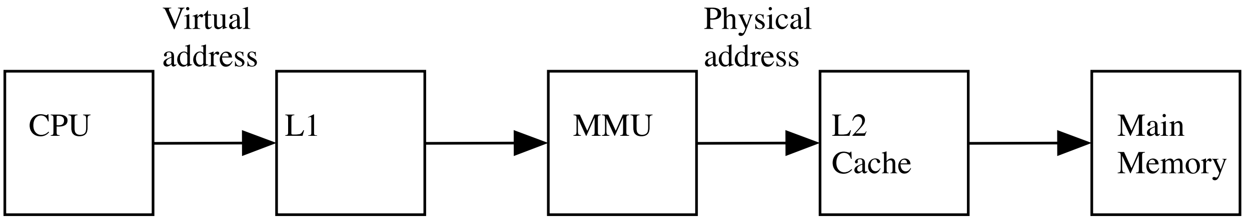
(1) The first arrangement has the advantage that the MMU only has to translate virtual addresses that result from cache misses. As a result, the MMU handles only a small fraction of all memory references.

Since translating an address may require a reference to a page table in memory (doubling the time needed for the reference) this placement reduces the likelihood of this occurring. Indeed, it could even reduce the need for a TLB in the MMU, given that the MMU is invoked relatively infrequently.

(2) The second arrangement has the advantage that entries in the cache are based on physical addresses in main memory,

which makes it easier to achieve cache coherence when multiple devices are accessing the same memory (e.g with DMA IO or on a shared memory multiprocessor)

d) In practice, a third arrangement is often used, in systems where there are two caches (as is almost always the case today)



(In the case of a system with dual/quad core CPU's, each core has its own L1 Cache and MMU; but all cores share L2 Cache).

B. When a system uses two levels of cache plus memory management, a memory access can turn out to be quite complex.

DEMO: Full.parameters

Read 1000 - first reference to page
Read 1004 - found in L1 cache
Read 1020 - found in L2 cache
Read 1080 - found on resident page
Read 2000 - fetches a new page

HANDOUT of AMAT Calculation Example

C. If a system uses cache and/or virtual memory. the pattern of memory accesses by a program may have a significant impact on its performance.

1. Recall that both cache and virtual memory depend on locality of reference to minimize the number of accesses to slower speed memory.

2. Consider the following problem, to be run on a computer that uses a 32 KB L1 data cache with a block size of 8.

a) We have a 1000 page document for which we want to prepare an index (lists of page numbers on which various words occur). We have a list of 1000 index terms (words that will appear in the index).

b) One approach: for each index term, scan the entire book looking for occurrences of it:

```
for (int t = 0; t < 1000; t ++)
for (int p = 1; p <= 1000; p ++)
   if (term[t] occurs on page[p])
      add p to the list of occurrences for term[t];
```

c) An alternate approach: scan the entire book page by page, looking for index terms that occur on that page

```
for (int p = 1; p <= 1000; p ++)
for (int t = 0; t < 1000; t ++)
   if (page[p] contains term[t])
      add p to the list of occurrences for term[t];
```

d) Which approach works better with the data cache?

(1)Since a typical page contains about 2000 characters, it is clear that the entire book is many times bigger than the cache. However, it should be easily possible to hold the entire array of index terms in the cache. (1000 terms @ < 10 characters)

(2)If we use the first approach, each time we encounter a page we will have a series of cache misses for all the text on the page. (The text for the page will long ago have been replaced in the cache by the text for some other page) [Of course, when we miss for one character, we read an entire block into the cache, so the next seven characters will be hits (assuming ASCII-coded text)]. Thus, the total number of misses is about

(1000 iterations of outer loop) * (1000 * 2000 / 8) misses = 250 million misses (+ a small number of misses for the index terms)

(3)If we use the second approach, each page of the book only needs to be scanned only once; and the array of index terms, once it is first read into the cache from main memory, can remain (though some entries may be bumped by text from the book and have to be reread on occasion) Thus, the total number of misses is about

1000 * 2000 / 8 = 250,000 misses (+ a number of misses for the index terms that is slightly bigger than in the last example, though still relatively small)

(4)If the miss penalty is 10 ns, then the difference between the two amounts to over 2 seconds - not a lot of time, but this illustrates a general problem that can arise.

3. As another example, consider the following program, which runs on a system that has a virtual memory system that limits each process to having 10,000 pages of size 4096 bytes resident at any one time.

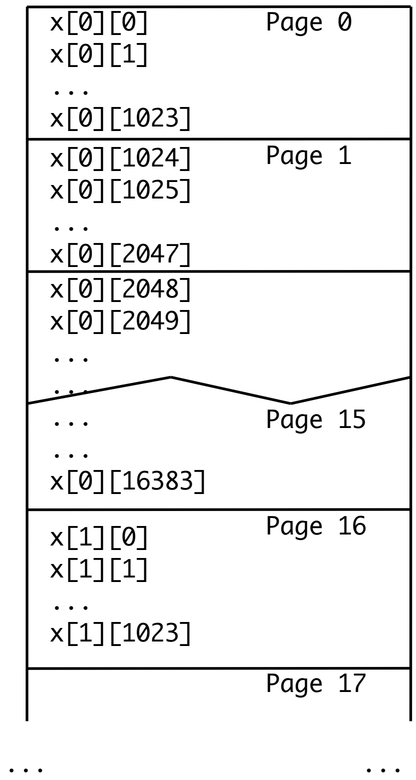
```
// x is a very large 2-dimensional array of reals
// over 268 million elements = 1 GB at 4 bytes/float

float x [ 16384 ] [ 16384 ];

// Calculate the average of elements in x

double sum = 0;
for (int i=0; i < 16384; i ++)
   for (int j=0; j < 16384; j ++)
        sum += x[j][i];
double average = sum / (16384*16384);
```

Now consider how the array x is stored in memory. It will occupy 1 GB / 4096 bytes/page = 256 * 1024 pages, with 1024 array elements on each page. Most programming languages (including C++) arrange the elements are arranged in row major order, looking like this:



This means that successive elements in the same column but different rows are 16 pages apart - e.g. x[0][0] is on page 0, but x[1][0] is on page 16.

This means that 16384 different pages are referenced while processing the first column. If FIFO page replacement is used (or something akin to it), then Page 0 will have been replaced by the time we finish the first iteration of the outer loop. Thus, when we go to process x[0][1], the page on which it resides (which would have been brought into main memory to access x[0][0]) will no longer be resident, and we will have to get it again from disk.

In fact, if no elements of the array are resident in memory to begin with, we will have a page fault for every element of the array - over 268 million faults in all. If each fault requires 10 ms to resolve, the total time just for handling page faults will be over 2,680,000 seconds = more than 745 hours!

4. Suppose, on the other hand, that we just switch the order of the two subscripts of x in the line that accumulates the sum, so we have:

```
sum += x[i][j];
```

5. Now we process all of the elements on Page 0 first, then all of the elements on Page 1 ... In fact, we only process each page once, resulting in a maximum of 16384 page faults - requiring about 164 seconds to handle!

6. The two approaches would also differ somewhat in terms of how efficiently they use the cache, but in this case the difference in terms of virtual memory usage is certainly enough to settle the efficiency question.

7. BTW, most programming languages store two dimensional arrays in row major order - so the order of processing

   ```
   for (int row = 0; row < # of rows; row ++)
      for (int col = 0; col < # of cols; col ++)
           -- process x[row][col]
   ```

   tends to be much more efficient that switching rows and columns.

8. However, FORTRAN - one of the earliest programming languages and one that is still used for a lot of scientific data processing (where large two dimensional arrays are common) uses column major order! Thus, naievely translating code between FORTRAN and Ada/C/C++/Java can produce interesting results!