Lecture 21 — Memory: Segmentation & Paging

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 6, 2017

Memory Segmentation

You've been repeatedly told that memory is a linear array of bytes.

You have also been told that there's the stack and the heap, libraries and instructions.

Both are true; they are views of memory at different levels of abstraction.

Each of the elements such as the stack, the heap, the standard C library, et cetera, are known as segments.

Memory Segmentation

Programmers do not necessarily give much thought to whether variables are allocated on the stack or the heap.

Or where program instructions appear in memory.

In many cases it does not matter, though C programmers are well advised to know the difference.

Memory Segments

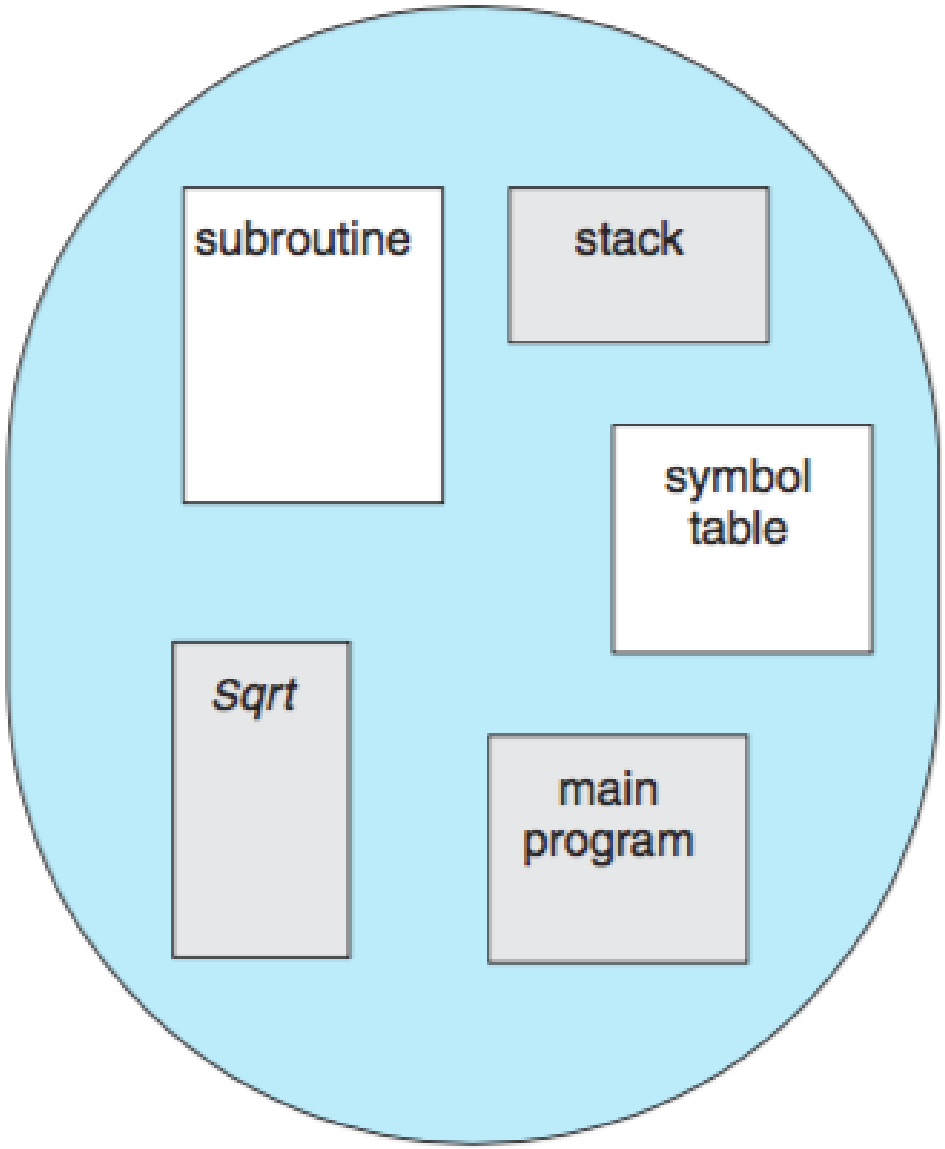
A full program has a collection of segments, which may be different lengths.

Normally the compiler is responsible for constructing the various segments.

Some possible segments:

1. The code (instructions).
2. Global variables.
3. The heap.
4. The stack (one per thread).
5. The standard C library.

Segments from the Programmer's Perspective



Thinking about Segments

Rather than thinking about memory as just a pure address, we can think of it as a tuple: $<segment, offset>$.

Given that, we need an implementation to map these tuples into memory addresses.

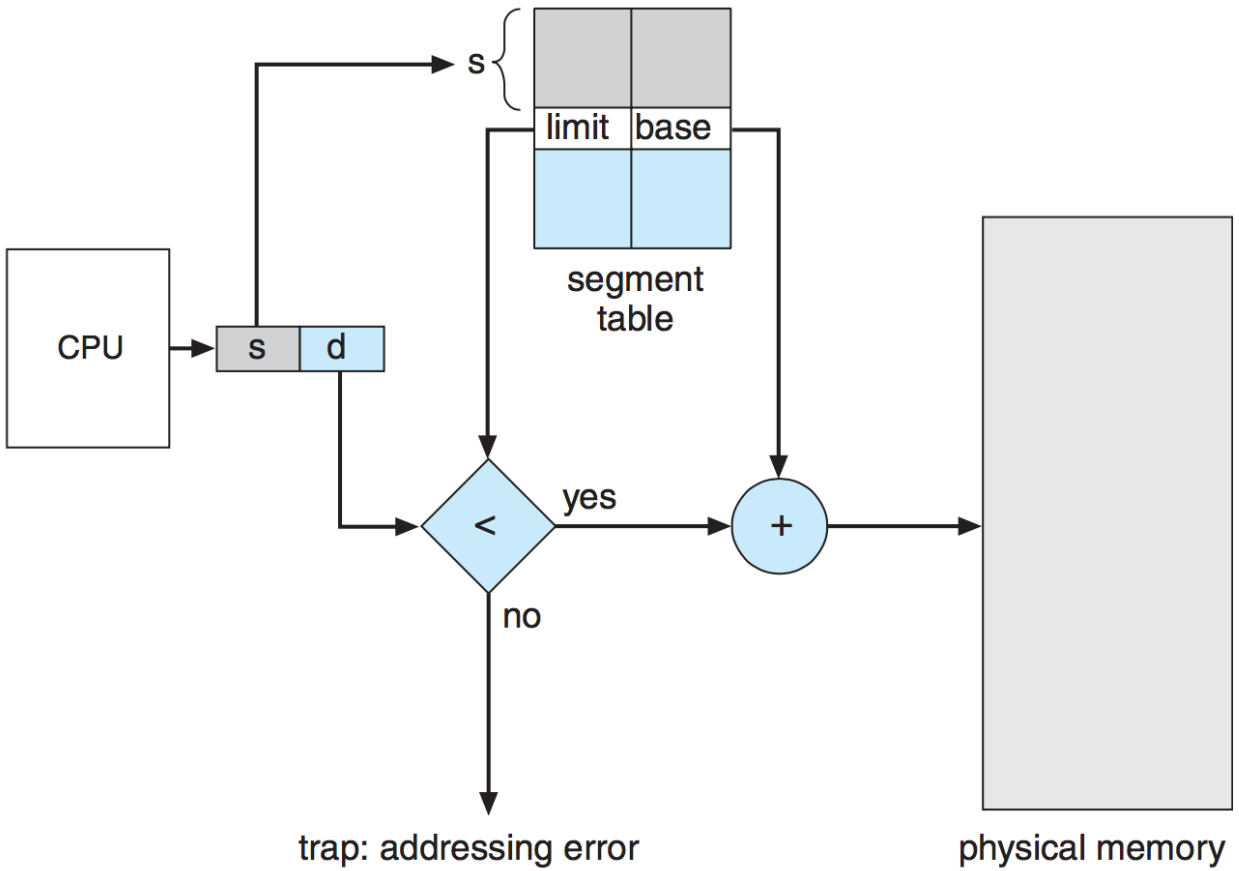
The mapping has a segment table.

Each entry in the table contains two values: the base and the limit.

So there will be some addition involved as well as a comparison to see if the address lies within that range.

Segment Table: Hardware

Memory accesses are such a common operation that we will need another rescue from the hardware folks to make this not painfully slow.



Segmentation: Contiguous

With segmentation, memory need no longer be contiguous.

We can allocate different parts of the program in different segments.

Different segments can be located in different areas of memory.

(Want to find out where something is? Use the `&` [address-of] operator...)

Paging

Fixed & variable sized partitions suffer from fragmentation, external or internal.

Divide memory up into small, fixed-size chunks of equal size, called frames.

Divide each process's memory into chunks the same size as a frame: pages.

Then a page can be assigned to a frame.

A frame may be empty or may have exactly one page in it.

Paging: Picture Frame

Imagine, as an analogy, a simple picture frame.

The frame may be empty or it may contain a picture.

If the picture frame is empty, all that is necessary is to put a picture in it.

Paging: Picture Frame

To put in a different picture, first, take out the picture that is already there.

Taking out the picture to empty the frame is allowed, too.

A picture is always aligned to be completely in a frame; not half in and half out.

Now expand this scheme by having a very long row of picture frames.

Each frame can contain one picture at a time, at most.

A picture can be in at most one frame at a time.

Paging: Process Start

A process starts, is loaded into memory, and has initial memory requirements.
(e.g., the stack and global variables).

The number of pages can & will change over time as memory is allocated/freed.

A process may also be swapped out to disk, but to run it will need to be swapped back in.

A process will take up a certain number of pages in memory at any given time.

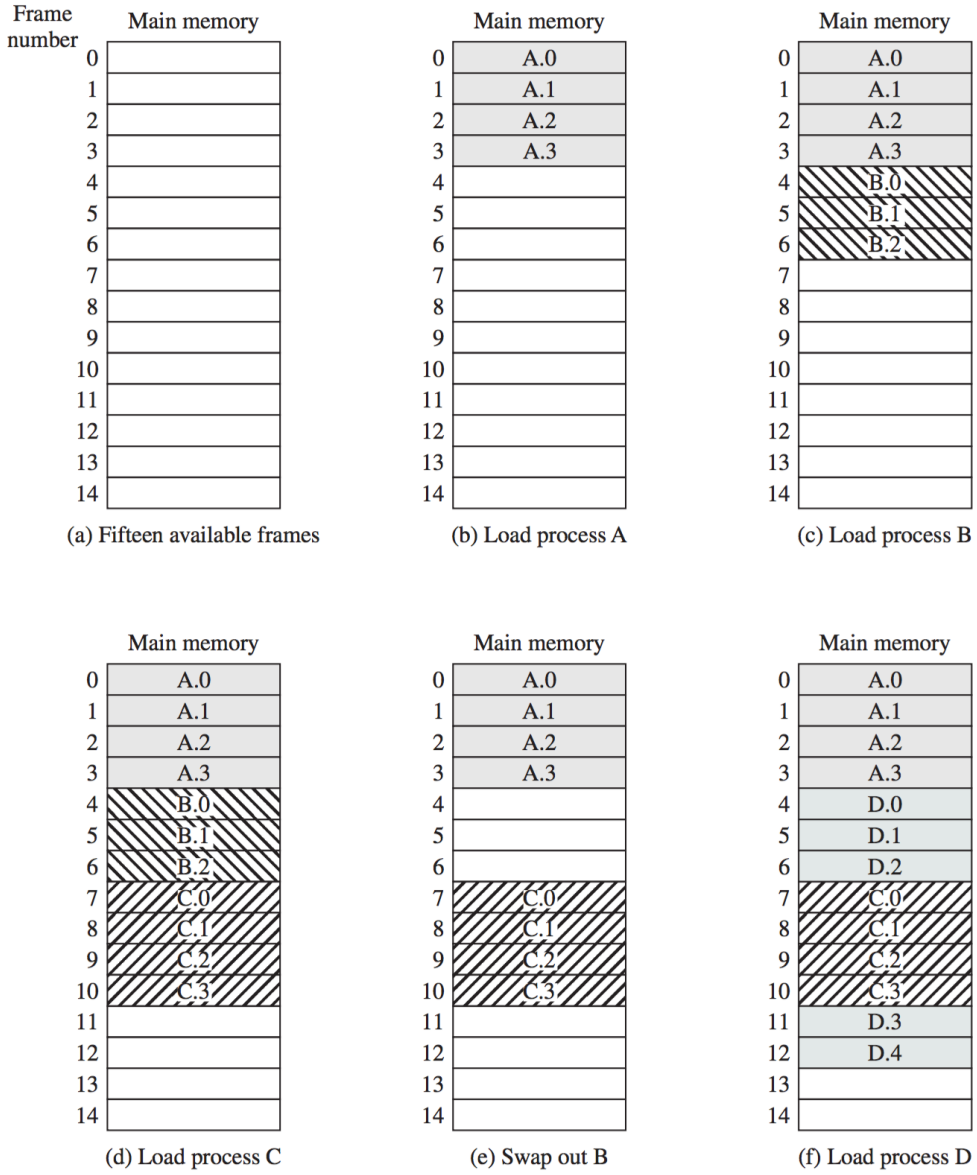
Page Benefits

Pages provide the benefit of separating the logical address from the physical address.

Programmers may pretend the address space of the computer is 2^{64} bytes.

Rather than however many GB of memory are in the physical machine.

Page Example



(a) Fifteen available frames

(b) Load process A

(c) Load process B

(d) Load process C

(e) Swap out B

(f) Load process D

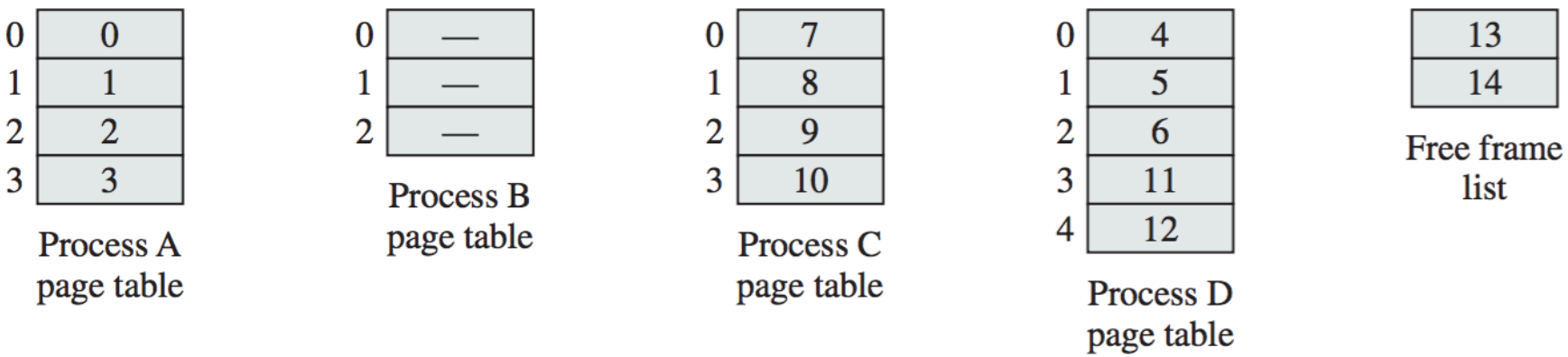
Page Tables

Now that we have multiple segments for each process and they are no longer contiguous, it is insufficient to have a base address and a limit.

Each process needs a page table, to keep track of which pages are located where in memory.

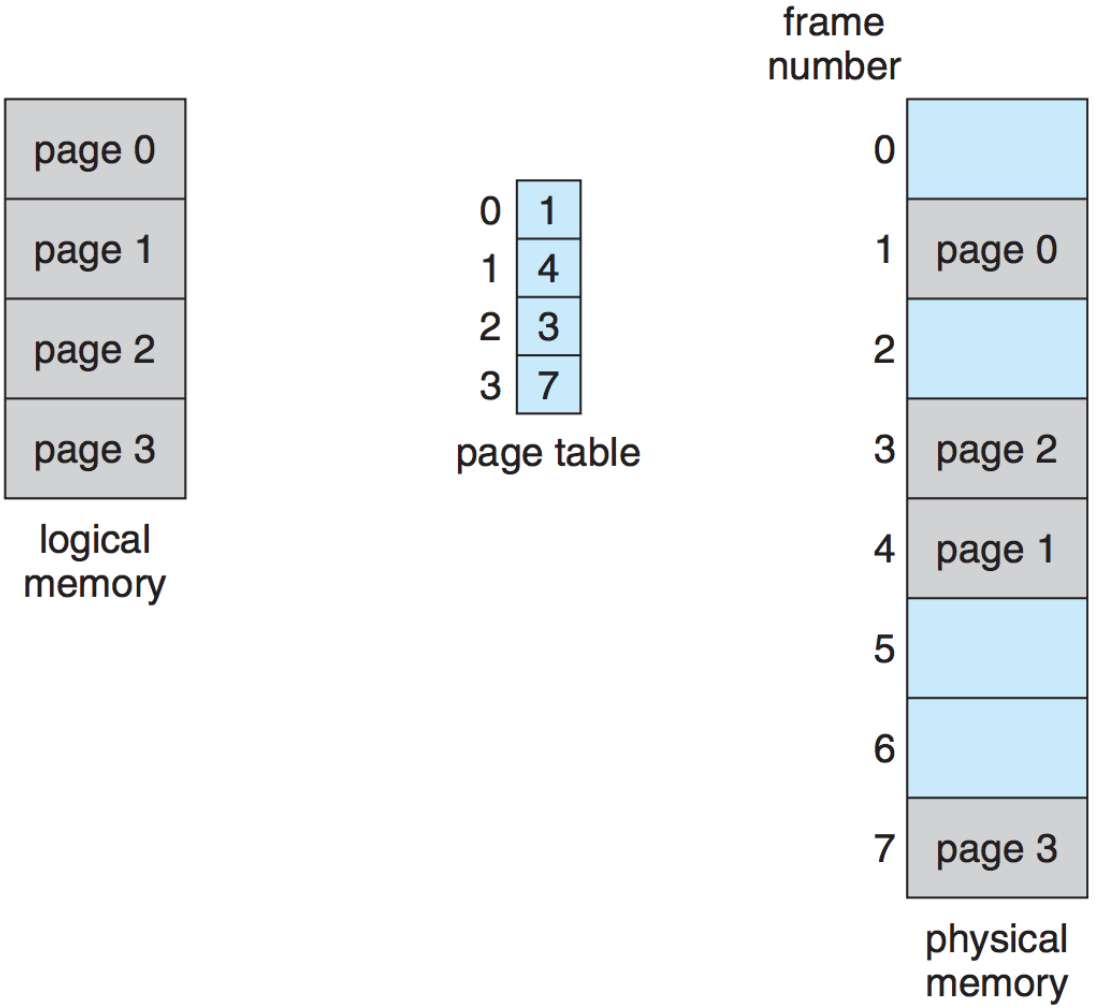
A list of free frames is also necessary

Page Table Diagram



Page Table

The page table is used to map logical memory to physical memory.



Page Sizes

For convenience, page size is usually a power of 2.
(and the actual value is determined by hardware).

The selection of a power of 2 makes translating a logical address into a tuple of the page number and offset easy.

If the logical address space has size 2^m and the page size is 2^n bytes:
The high order $m - n$ bits of the logical address are the page number;
The lower n bits are the page offset.

Pages: External Fragmentation

External fragmentation is eliminated as a problem in this scheme, because pages are all the same size.

That also means that compaction is not an issue.

Compaction, when it's possible, is painful enough in memory.
    It is excruciating to do on disk.

It is therefore desirable to avoid it entirely.

We accept some internal fragmentation because a process gets a whole page.

Pages: Internal Fragmentation

How much internal fragmentation do we have to live with? Not very much.

If the memory required aligns perfectly with a multiple of the page size, then no memory is wasted.

If a new memory allocation comes in, then a new page is allocated and added to the logical memory space of the process.

The last frame, however, may not be completely full.

In the worst case scenario, a full page less one byte is wasted.

Internal fragmentation of one page is not very much overall.

Pages: Does Size Matter?

How big should page sizes be?

If they are smaller, then less memory is wasted in internal fragmentation.

However, having a large number of pages introduces a lot of overhead.

The size of pages has tended to grow along with the size of main memory.

Page Size: Disk Size

The key factor is actually disk: the disk operates on a certain block size and it is most efficient for the size of a page to be equal to a disk read/write size.

That way when a page is to be swapped into or out of memory, it can be done in a single disk read or write.

In a typical modern system, pages are 4 KB, but they can be bigger.

Pages: Size and Protection

Now we finally have a good answer to why the application developer can treat memory as if it is infinitely large and unshared.

The program is scattered across physical memory, but appears to the application developer and running application as if it is all contiguous.

Pages: Size and Protection

We also get protection in this scheme.

A program cannot access any address outside of its memory space.

There is simply no way to make a memory request outside of the logical memory space.

No matter what address is generated, it could only be inside the page table, and the page table has only entries of that process.

Pages: The OS View

The operating system, however, can manage memory of all processes, so it will need another scheme.

The OS will operate on the frame table, a listing of all the frames, indicating which page of which process a frame currently holds, if any.

Shared Pages

Another great advantage of paging is the possibility of sharing of common code.

Users very often have multiple programs open; and sometimes they are duplicates (e.g., notepad, Microsoft Word, et cetera).

In a multiuser system, different users may have some of the same program open (e.g., Skype, Firefox, et cetera).

We could reduce memory consumption if common parts of this program are shared between all instances of that program.

Sharing is Caring: `vi`

Imagine there are 5 users on the system, each of whom wants to use `vi`.

Let's say the program itself uses 10 pages (made up number) on its own, and then some variable number of pages based on what file is being edited.

Without sharing, each copy of `vi` that runs will consume 10 pages, so 50 pages are being used for the executable.

If we can share those 10 pages, we have saved 40 pages worth of memory space.

Sharing Code

Other programs and code can easily be shared, such as compilers, libraries, and operating system utilities.

In fact, any code can be shared as long as it is reentrant (also sometimes called pure or stateless).

This is code that does not change when it is executed.

That means there is no state maintained by the code.

Non-Reentrant Code

Any function that accesses a global or static variable is non-reentrant.

```
int tmp;
void swap( int *x, int *y ) {
  tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Page Table Structure

In the simplest form, the page table is just a standard table.

This structure is simple, but page tables can be very large.

If the system is 32-bit, and page sizes are 4 KB (2^{12}), then the page table has $2^{32}/2^{12} = 2^{20}$ pages, or about 1 million entries.

Page Table Structure

We will examine three strategies for structuring the table:

1. Hierarchical paging.
2. Hashed page tables.
3. Inverted page tables.

Hierarchical Paging

Rather than have one big table, we have multiple levels in the page table.

The page table can be broken up and need not be contiguous in memory.

Suppose we have a two level system.

If the page number is p, the first k bits indicate the outer page.

The outer page contains some information about where the inner pages are.

The remaining $p - k$ bits identify the inner page.

After the inner page is identified, the displacement d is then calculated.

Hashed Page Tables

Instead of the page table being an array of entries, turn it into a hash table.

There is a hash function to assign pages to “buckets” and each bucket is implemented as a linked list.

Then each element of the list is examined to find the matching page.

Inverted Page Tables

For 32-bit virtual addresses, a multilevel page table can work.

But with 64-bit computers, with 4 KB pages, the page table requires 2^{52} entries.

If an entry is 8 bytes, then the table is over 30 million gigabytes (30 PB).

Does your computer have that much memory?
... If so, can I borrow it?

Inverted Page Tables

Inverted page table: one entry per frame, rather than one entry per page.

The entry keeps track of the process and page number.

This saves a huge amount of space.
(1 GB of ram with a 4 KB page size $\rightarrow$ page table requires only 2^{18} entries).

The drawback: no longer possible find physical pages by looking at the address.
Instead, searching (slow).

Paging: Hardware Support

Memory accesses are very frequent and require additions and comparisons.

Recall an operation as simple as adding two numbers requires fetching the add instruction, fetching the operands, and storing the result.

To prevent abysmal performance, modern computers have hardware support.

Hardware is much, much faster than doing these operations in software.

Paging: Hardware Support

The simplest implementation is to use a set of dedicated registers.

Registers are the fastest form of storage.

When a process switch takes place, these registers, just as all other registers, are replaced with those of the process to run.

The PDP-11 was an example of a system that had this architecture.
    Addresses were 16 bits and the page size was 8 KB.

Paging: Hardware Support

The page table was therefore 8 entries and kept in fast registers.

This might work if the number of entries in the page table is small (256 entries).

The page table can easily be something like 1 million entries.
Does your CPU have 1 million registers? If so...?

The page table is in main memory. A single register points to the page table.

There's Always a Catch

To access a page from memory, we need to first figure out where it is, so that requires accessing the page table in main memory.

Then after retrieving that, we can look in the page table to find the frame where the desired page is stored.

Then we can access that page.

There's Always a Catch

So two memory accesses are required for every read or write operation.

As far as the CPU is concerned, main memory already moves at a snail's pace.

Doubling the amount of time it takes to do a read or write means it takes roughly forever.

Thus, we will need to find a way to speed this up.

Hardware to the Rescue, Again

A fast cache called the translation lookaside buffer (TLB).

You can think of the TLB as a key-value pair (think HashMap).

The key is the logical address and the value is the physical address.

Translation Lookaside Buffers

To make the search fast, the comparison is done on all items simultaneously.

To prevent this from being extremely expensive, the size of the TLB is limited.

It’s usually something around 32 to 1024 entries in size.

Systems have evolved from having no TLBs to having multiple levels, over time.

Check the TLB

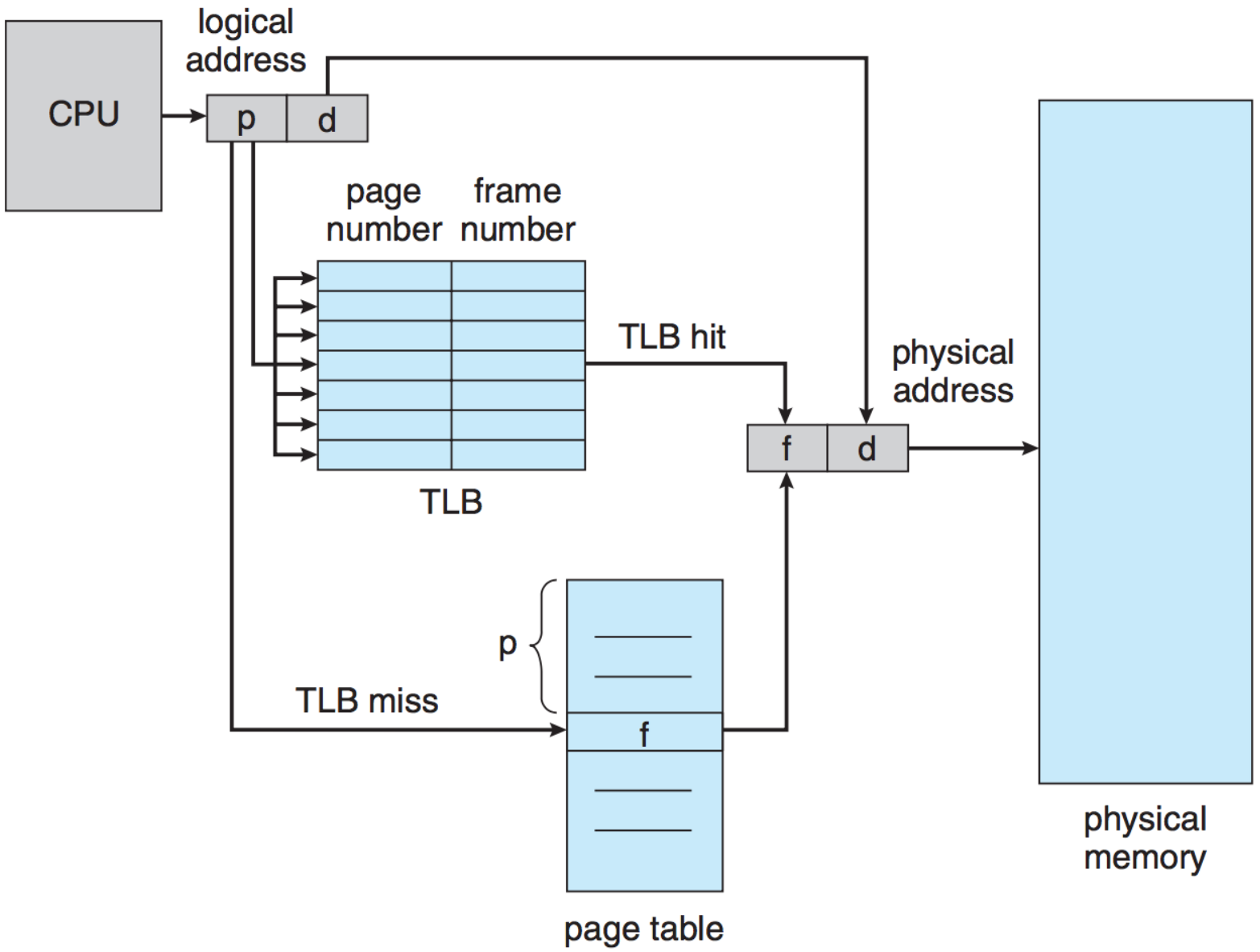
When a memory read/write is issued, the page number is checked.

If it is found in the TLB then the frame number is immediately known.

If the page number is not found in the TLB, this is what we call a TLB miss.

We must look in the full page table, which unfortunately is slower because it requires reading from memory.

TLB Operation



Caching

The TLB idea is a specific instance of the strategy of caching.

Much earlier, when talking about the basics of computer hardware, we mentioned that memory comes at different levels and and different speeds.

Caching is a critical idea in computers and operating systems.

In fact, caching is such an important topic, that it will be the next topic...