Memory

Chapter 9

Background on Main Memory

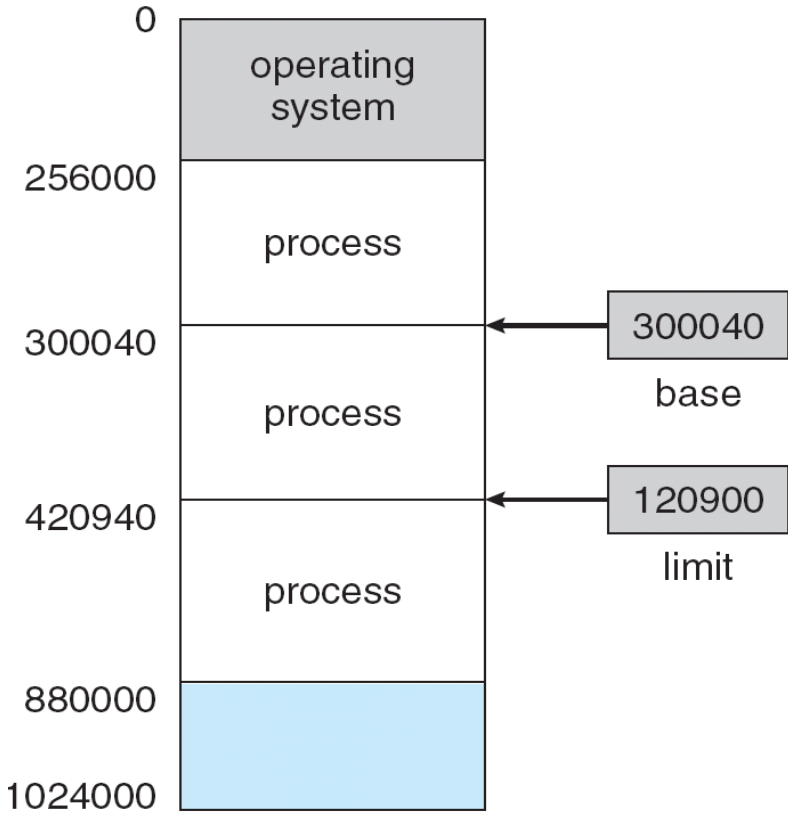
- Memory is a large(ish) *Linearly Addressable* array of *words*
  - Typically, memory is byte addressable, but it could be structured differently
  - Think of main memory as being big, but smaller than you'd like it to be
- Registers and main memory are the only storage the CPU can directly access
  - So, a program must be brought into main memory (typically from disk) before it can be run.
  - Registers are fast, main memory is slower, cache helps to make up the difference

Memory Layout for the OS

- Pretend main memory is divided into two regions
  - Space for a *resident operating system*
    - Maybe in low memory addresses
    - With structures like the interrupt vector table and memory-mapped I/O
  - Space for multiple *user processes* in the rest of memory
- Want to fit in as many user processes as we can

A Simple System for Memory Protection

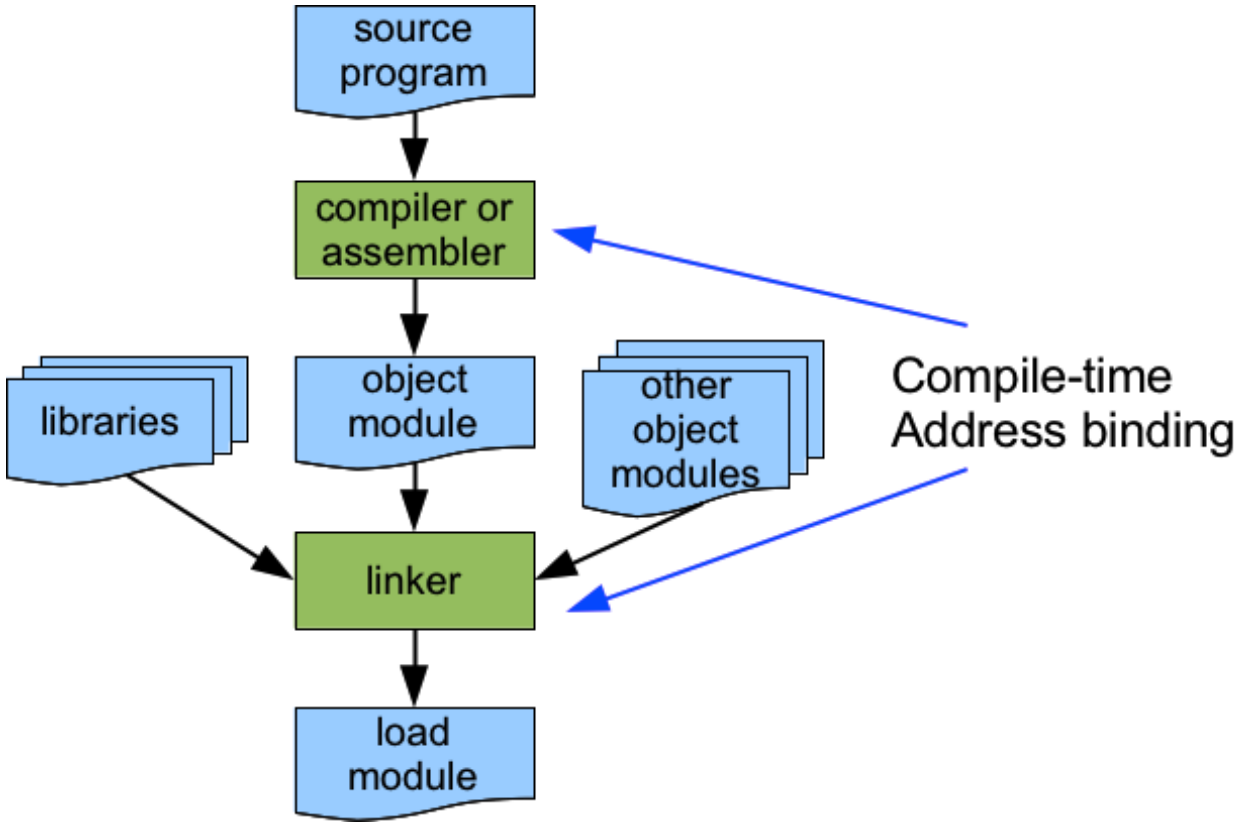
- A pair of registers, *base* and *limit*



Address Binding

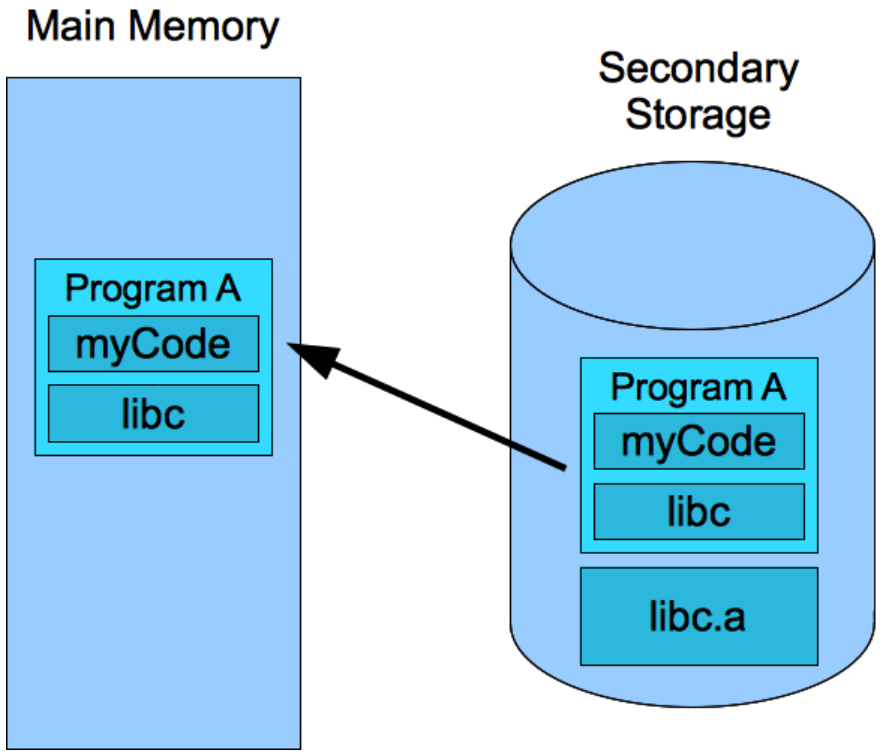
- *Address Binding* : that's how we choose physical memory addresses where a program will run
  - Really, the addresses for each program symbol
  - *Compile time*: compiler chooses an address when it builds the program
    - This builds *absolute code*, must be run in a particular location and rebuilt if it's going to run elsewhere
  - *Load time*: Choose memory address when the program starts running
    - Instruction set may make it easy to build *relocatable code*, will run wherever you put it, but you can't move it after it starts.
  - *Execution time*: Can move after it's started running
    We're probably going to need more help from the hardware.

Building a User Program



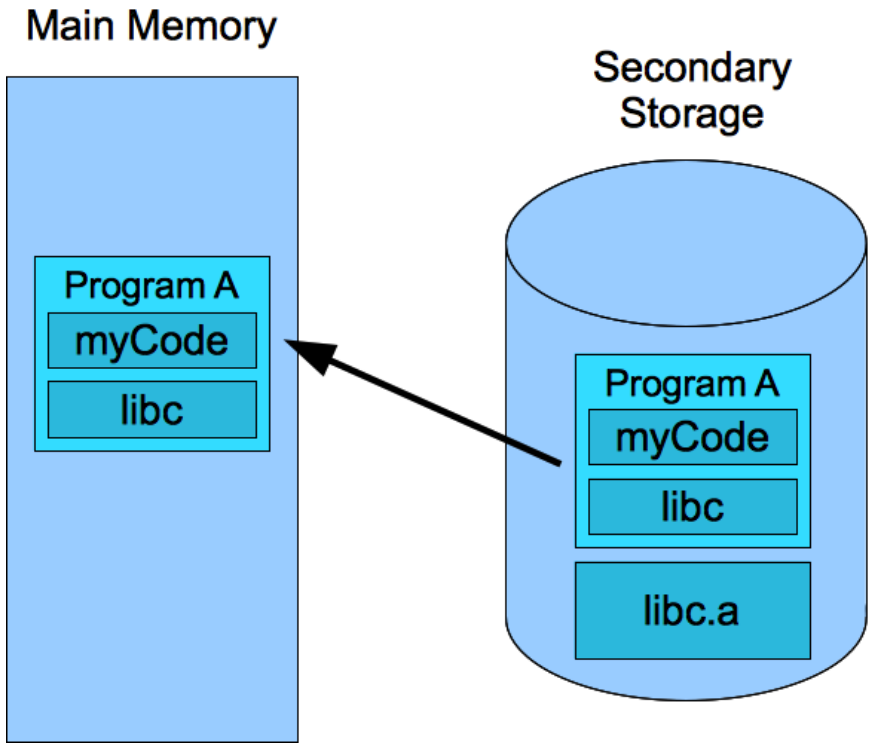
Static Linking

- *Static Linking*
  - Do all linking at build time
  - Application code and libraries linked into a single load image
  - Ready to execute, just copy into memory and run.



Static Linking

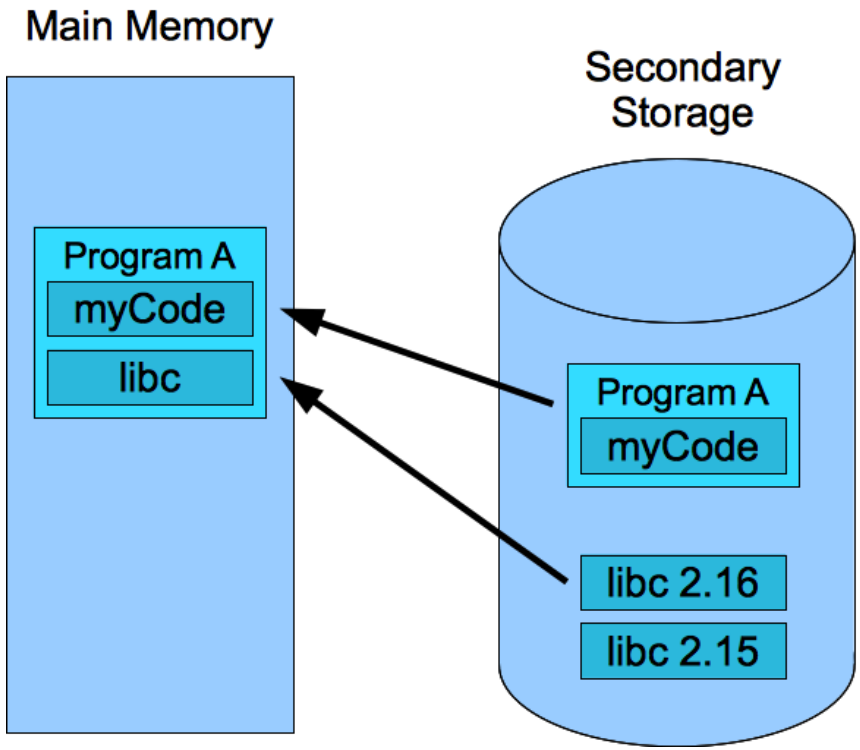
- Executable includes a copy of needed libraries, but just the parts it needs
- So, somewhat reduced memory footprint




```
$ ls -l libc.a
3153906 bytes
$ gcc -static Hello.c
$ ls -l a.out
615997 bytes
```

Dynamic Linking

- *Dynamic Linking*
  - Link with library code at load time
  - Get the latest (compatible) version of every library
  - Smaller executable image

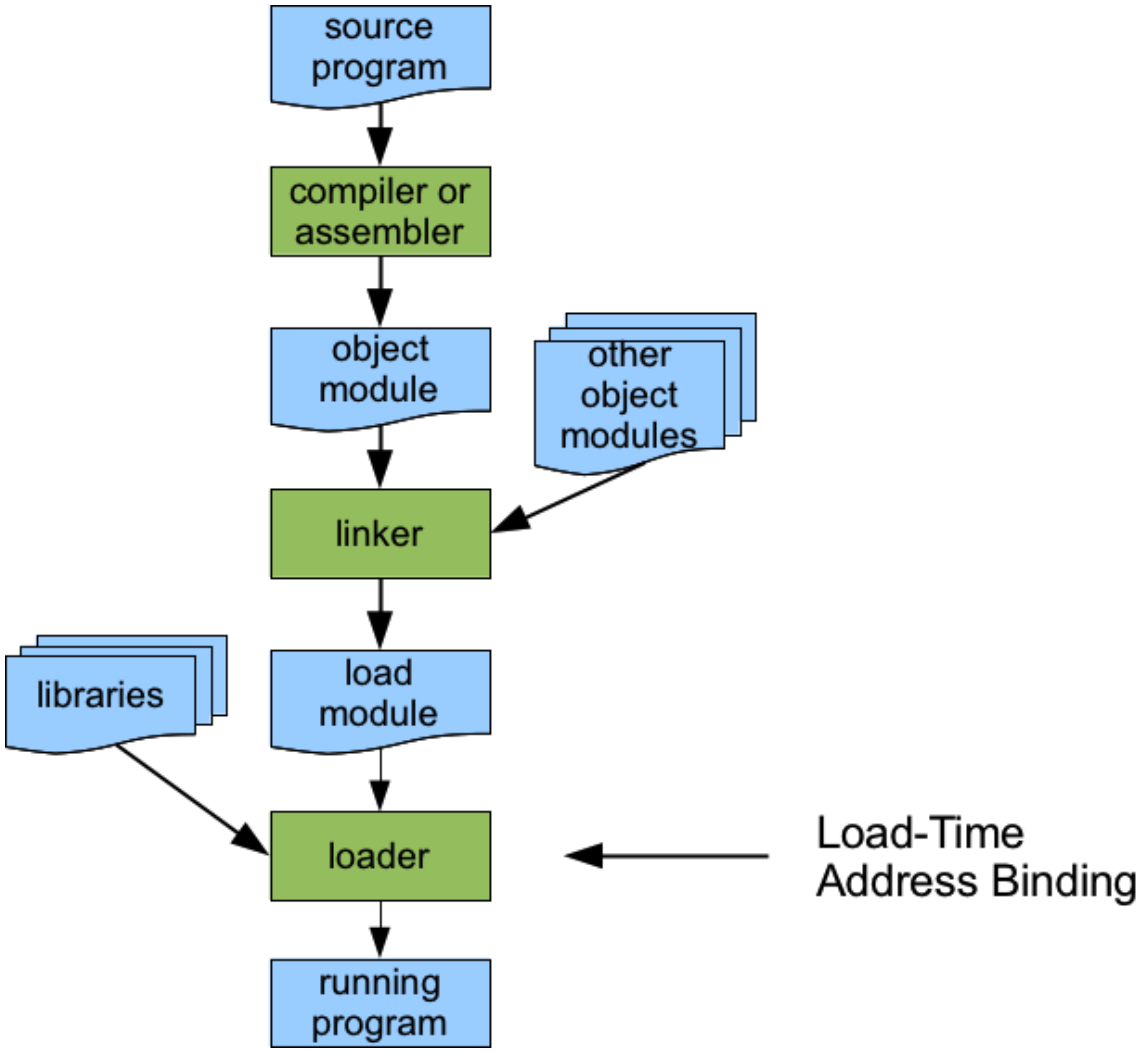



```
$ gcc Hello.c
$ ls -l a.out
7096 bytes
```

Dynamic Linking

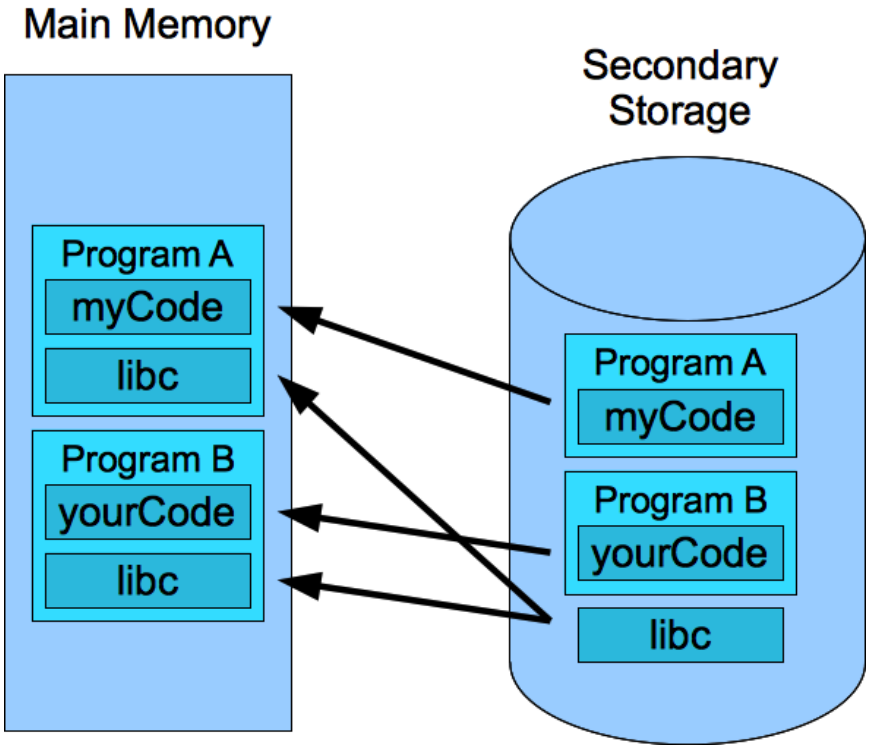
- Linking postponed until load- or execution-time
- Maybe, no special help from the OS needed
  - Initially, a small piece of code, a *stub*, gets called instead of the subroutine
  - On the first call, the stub finds and replaces itself with the actual subroutine
- Particularly useful for libraries

Starting a User Program



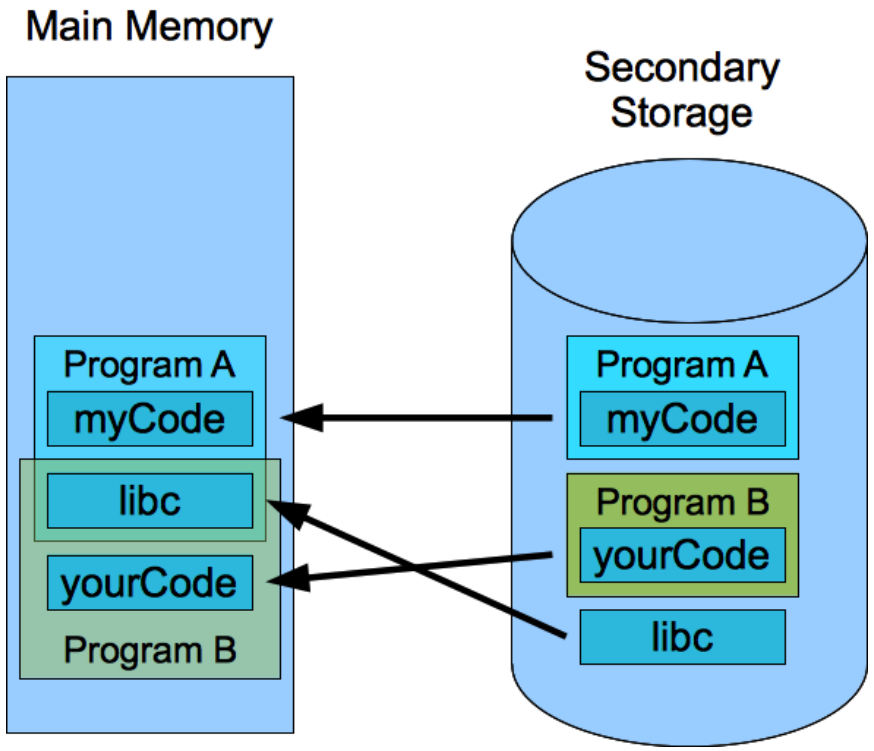
Dynamic Linking

- Programs share common code on disk,
- But, not necessarily in memory.

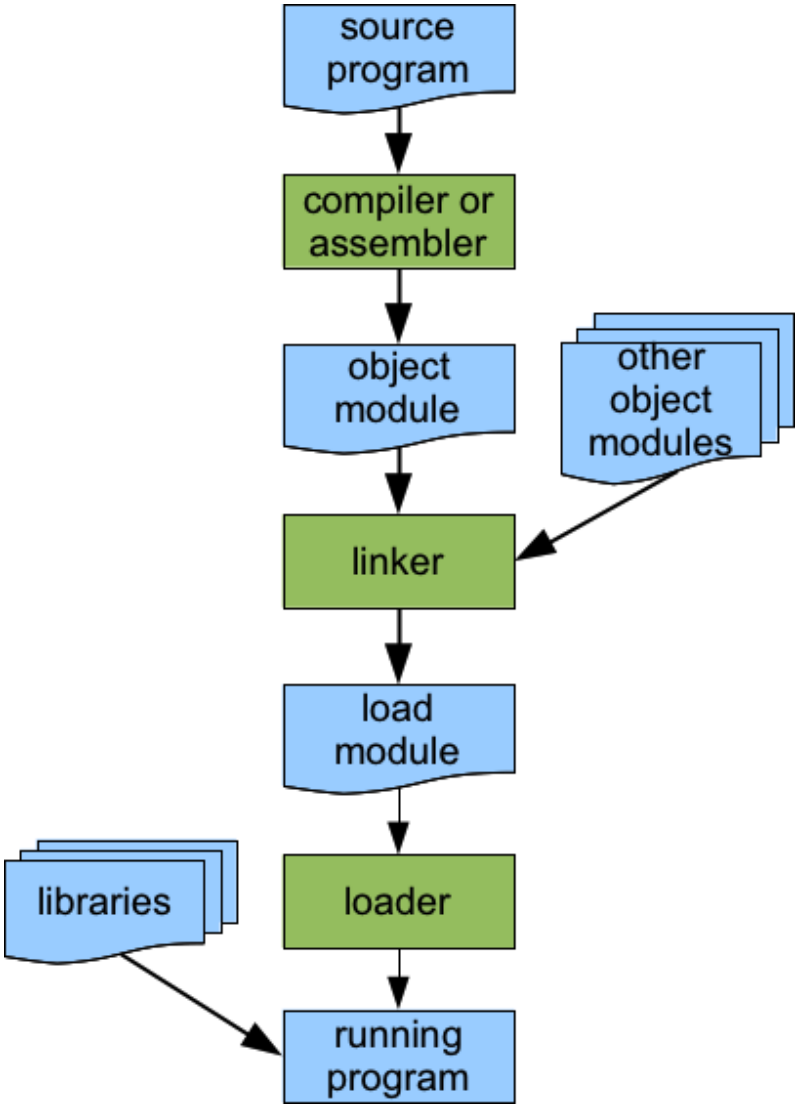


Shared Libraries

- *Shared Dynamically Linked Libraries*
  - Just one copy of the library on disk,
  - … and in memory.
  - Significantly smaller memory footprint
  - Definitely requires help from the OS.
  - And, we're going to need more than base/limit registers



Dynamic Loading



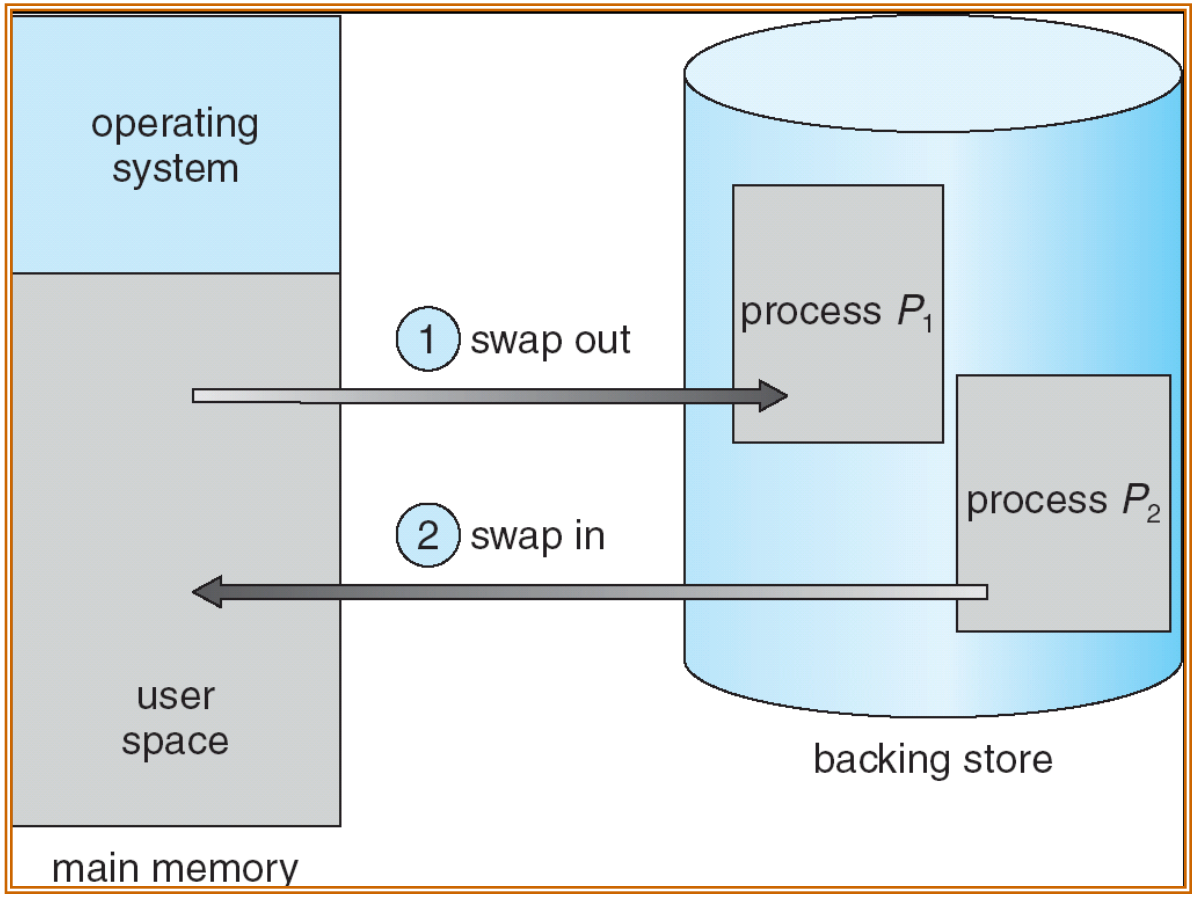
Dynamic Loading

- We want the most out of a limited physical memory
- *Dynamic Loading* : Load parts of the program when they are first needed
  - e.g., load a subroutine when its first called
  - *Overlays:* same region of memory used for several subroutines, one after another
- Useful if large amounts of code are needed later in execution (or maybe not at all)
  - Faster start-up times
  - Reduced process memory footprint (maybe)
- No help from the OS required … but it might be nice.

Swapping

- When running lots of processes, memory may fill up
  - For example my laptop is running 238 processes right now
  - But, most of them are idle most of the time
- Solution, temporarily remove an entire process from memory
  - *Backing store*, fast, reasonably large storage (usually disk) used to hold memory contents for processes that don't reside in main memory right now
  - Bring process' memory image back into physical memory when there's more room

What's Swapping Look Like?

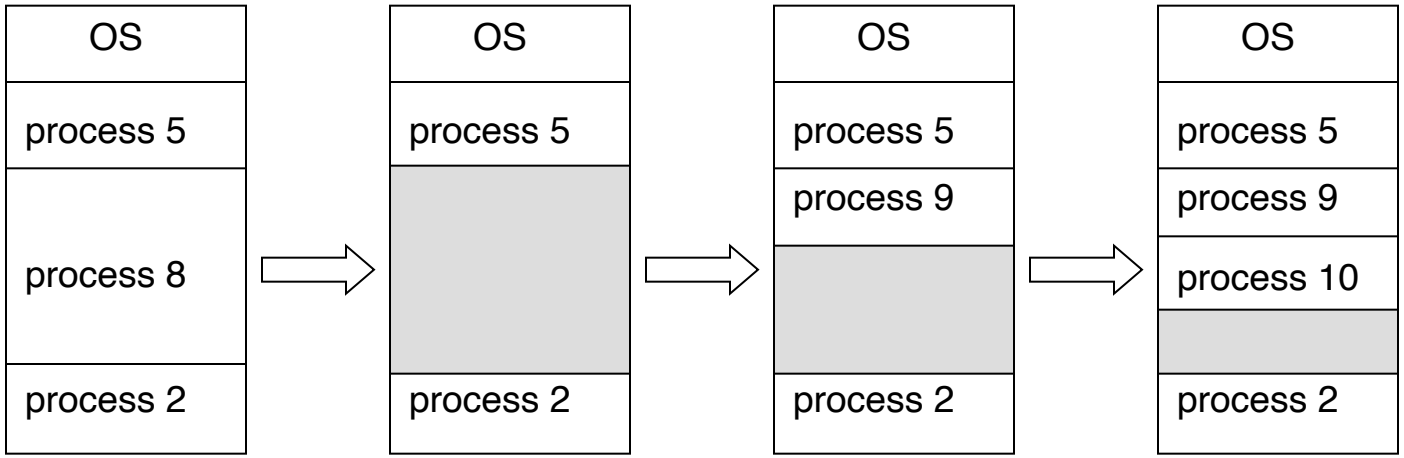


Swapping

- This is called *swapping*
  - Process PCB stays in the process table
  - Really, we are preempting a process' memory
  - So, this is a different way a process can block
- Major cost of swapping is *transfer time*
  - Total transfer time is proportional to process memory size

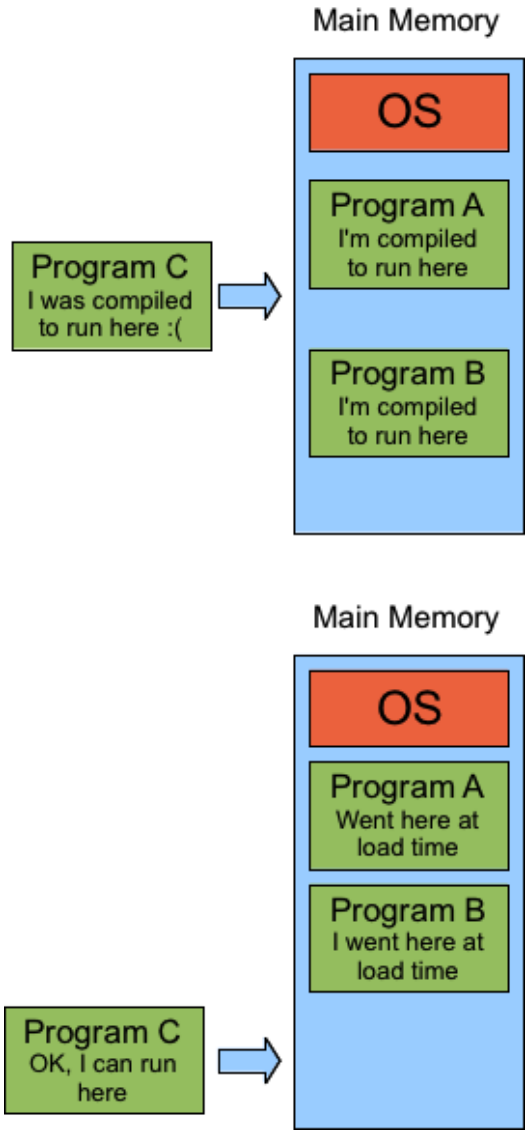
Making Room for Processes

- It's like a game, the OS has to find room for all the processes
  - What are the rules?
  - For now, processes require *contiguous allocation*
    - All memory for a process must be in one big block
- Are you going to run multiple processes? That's *multiple partition allocation*
  - As processes arrive and leave, the OS must find room for the new ones



Making Room for Processes

- Address binding matters
- With compile-time address binding
  - We have to put a process in the memory it was compiled for
- With load-time address binding
  - We can load a process into any region of memory
  - As long as there's enough



Dynamic Storage Allocation

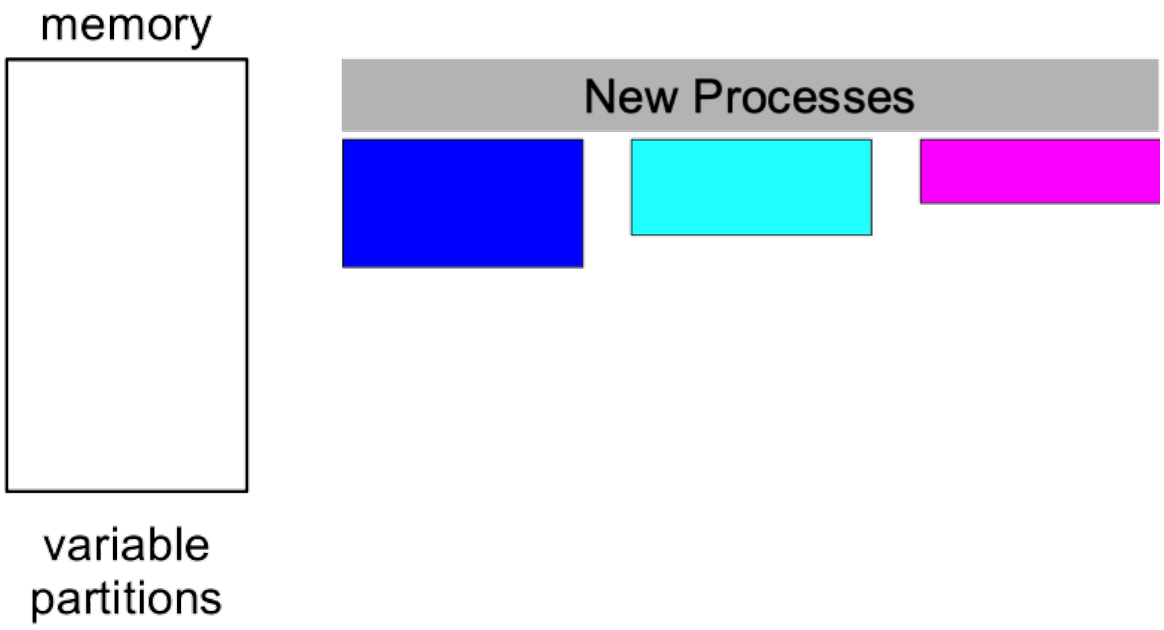
- This is a general problem
  - OS has to find space for processes as they start and exit
  - Just like malloc()/new must find space for new program structures
  - OS manages all of physical physical memory
  - malloc()/new manages the block of heap memory inside a process
- Both must:
  - Maintain a list of allocated regions
  - Maintain a list of free memory *holes*
  - Find sufficiently large holes when a new requests arrive
  - Re-claim memory when it's no longer in use
  - Efficiently coalesce adjacent holes into larger ones

Dynamic Storage-Allocation Problem

- We have a list of allocated regions and a list of *holes* (free blocks between them)
- A request for a size-*n* block comes along
  - *First-fit* : walk down the list, choose the first hole as large as *n*
  - *Best-fit* : find the smallest hole that's as large as n
    - Maybe a good idea, less wasted space
    - But, it could take a little longer
    - ... and, the leftover hole will probably be useless
  - *Next-fit* : get the first sufficiently large block after the last one allocated
    - May promote locality
  - *Worst-fit* : find the largest hole and use it
    - Maybe the left-over hole will be big enough to be useful
    - Maybe more efficient to implement

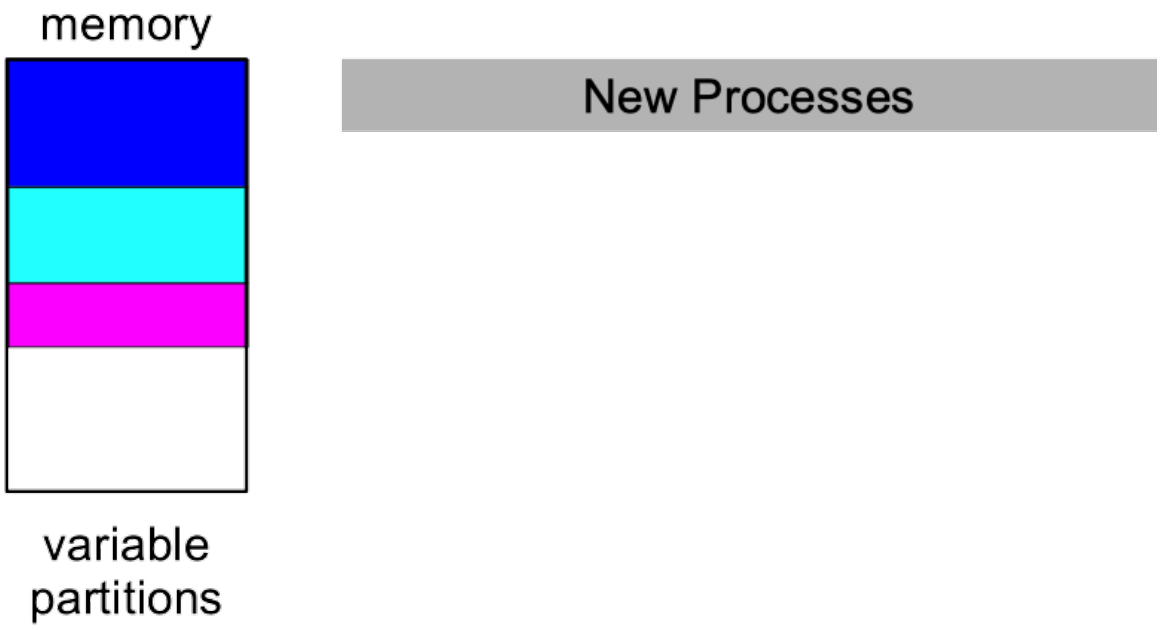
Memory Partitioning for User Processes

- Variable partition allocation
  - Find room for each new process based on its size.
  - We can pack them into memory one after another, without any gaps.



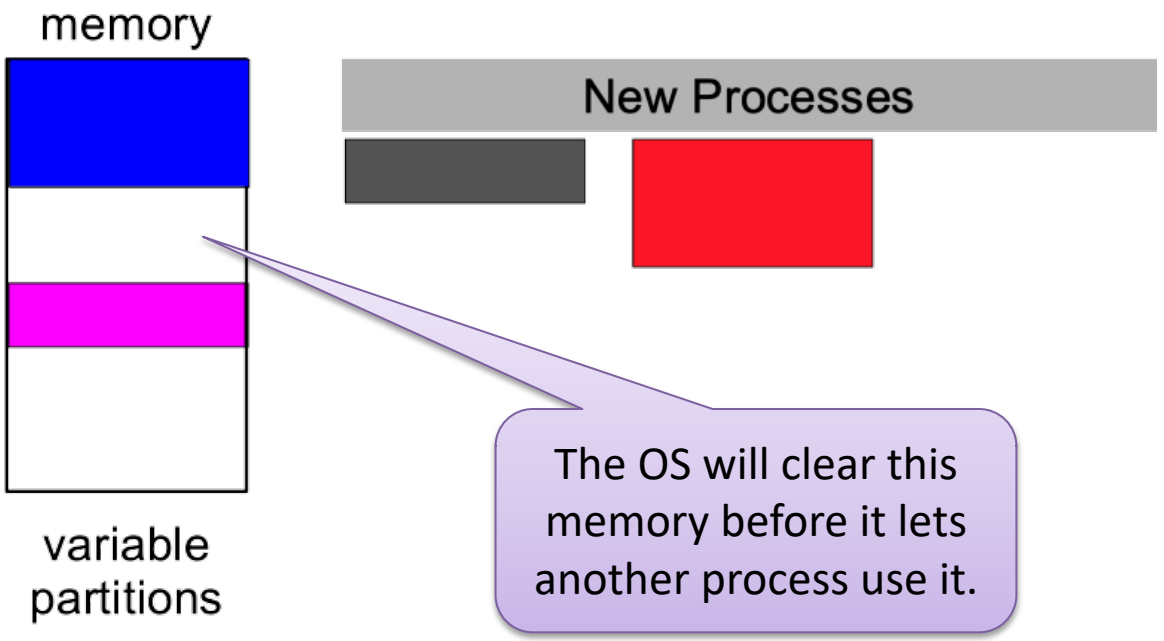
Memory Partitioning for User Processes

- Variable partition allocation
  - We can pack them into memory one after another, without any gaps.



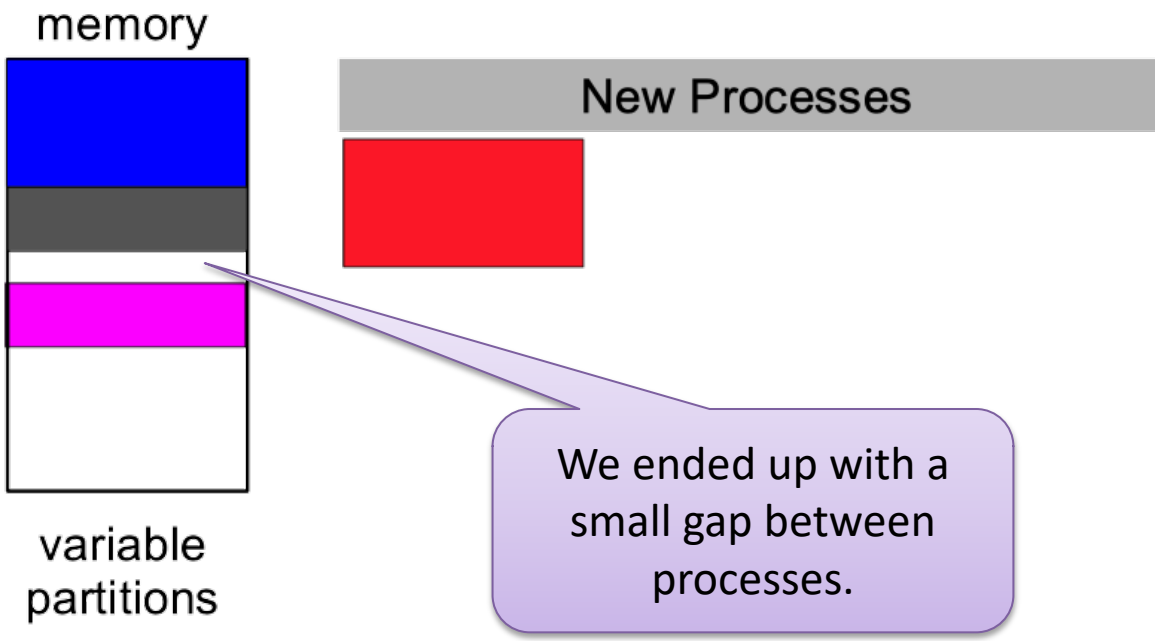
Memory Partitioning for User Processes

- Processes will finish up
- … and new ones will start up and re-use their memory.



Memory Partitioning for User Processes

- May get some *external fragmentation*

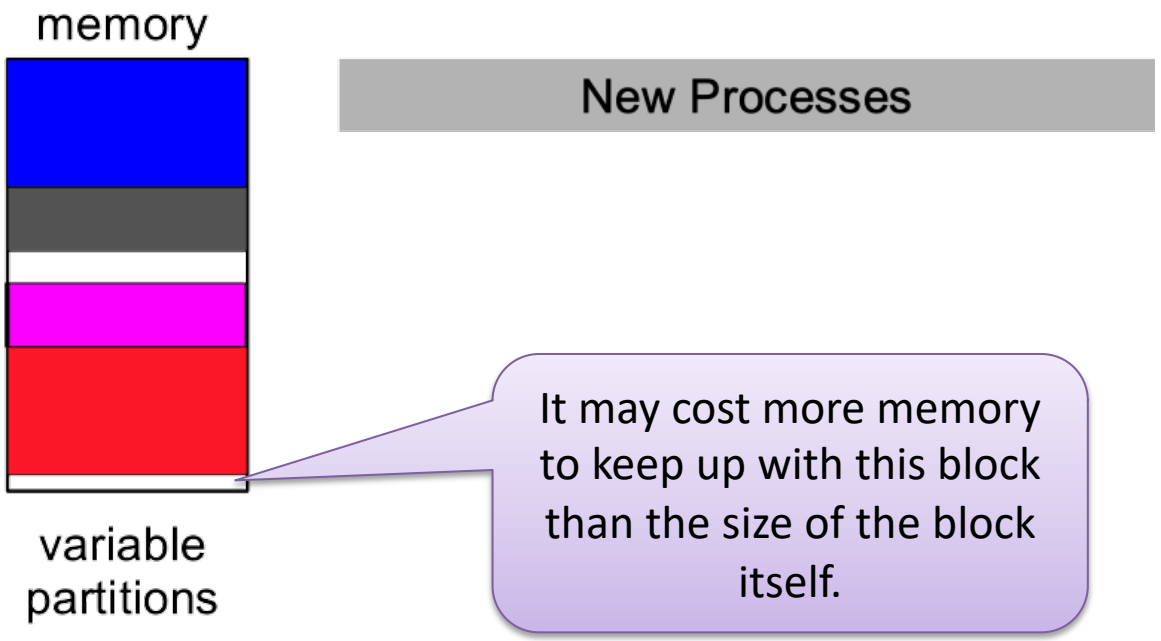


Fragmentation

- Some memory is going to be wasted
- *External Fragmentation*
  - There may be some wasted memory between allocated regions
  - Lots of holes, but all too small to use
- *50-percent rule*
  - With first-fit, if a total of *n* bytes of memory have been allocated, another ½ *n* will be lost to fragmentation

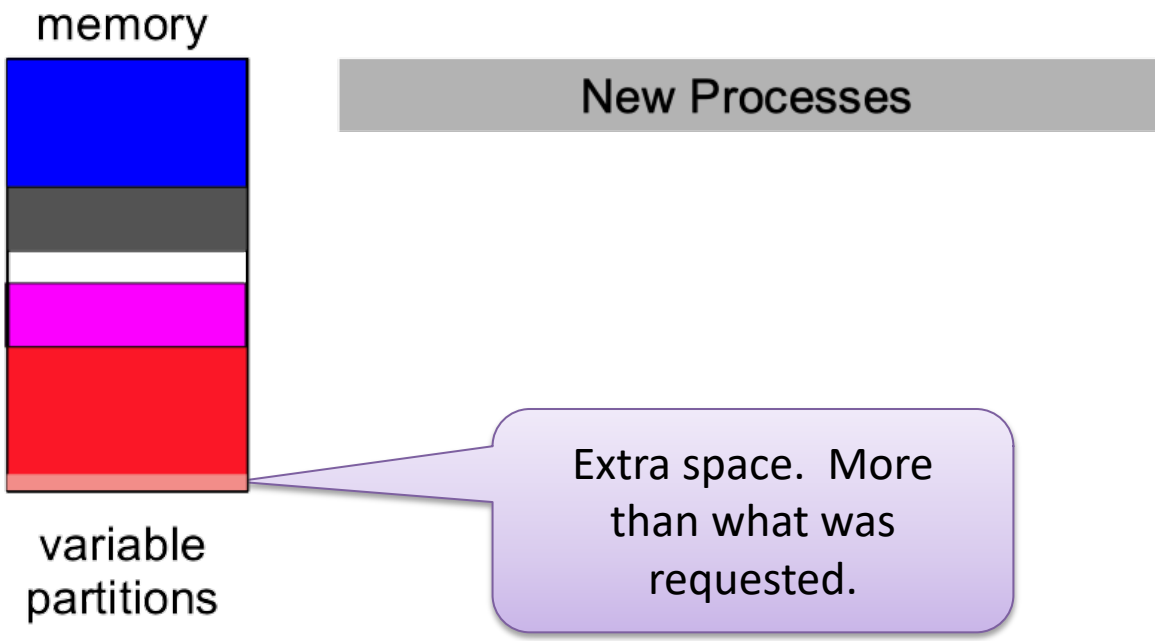
Memory Partitioning for User Processes

- If a hole is very close to the size of a memory request
  - It may not be practical to keep up with the left-over memory.



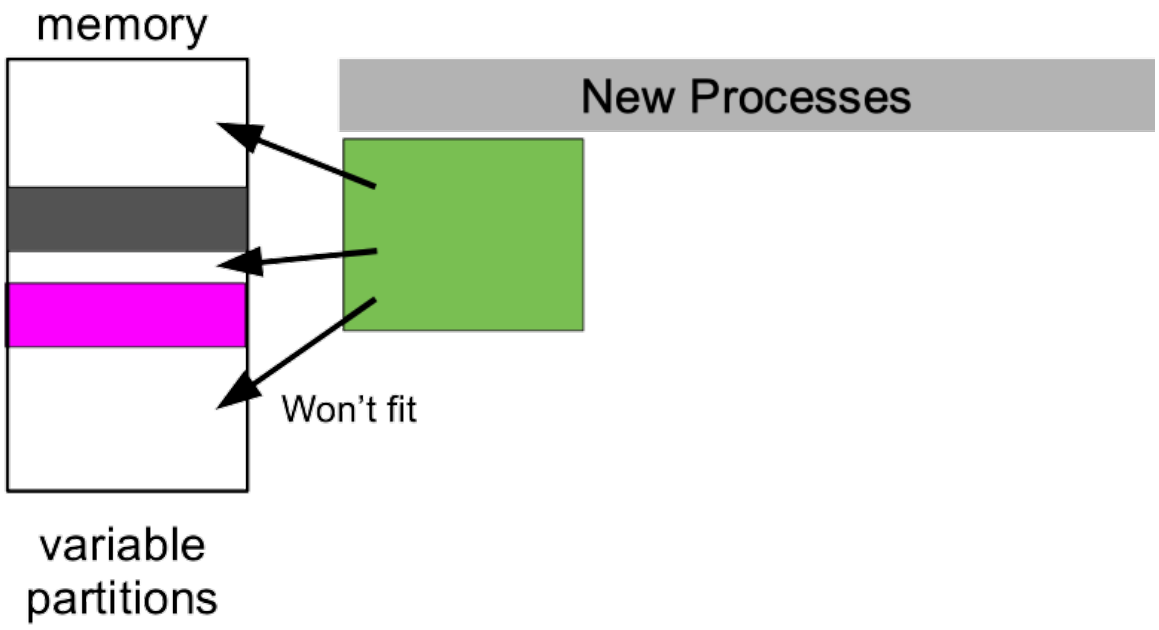
Memory Partitioning for User Processes

- We may give a little more memory than was needed
  - to simplify memory bookkeeping.
  - That's called *internal fragmentation*



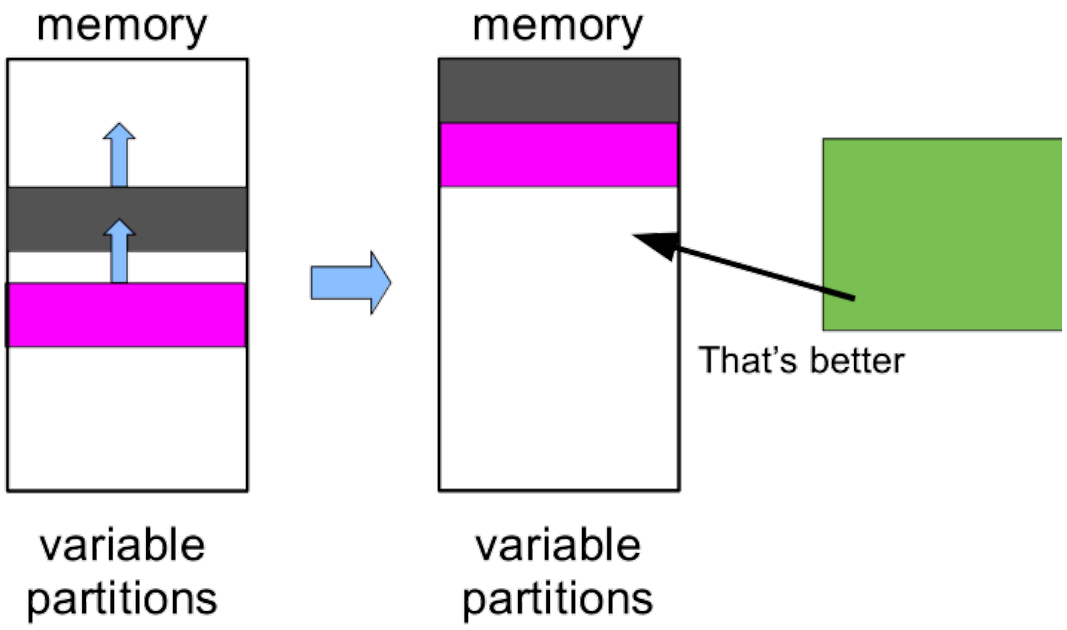
Memory Partitioning for User Processes

- Eventually, we may have a lot of memory, that's too fragmented to use.



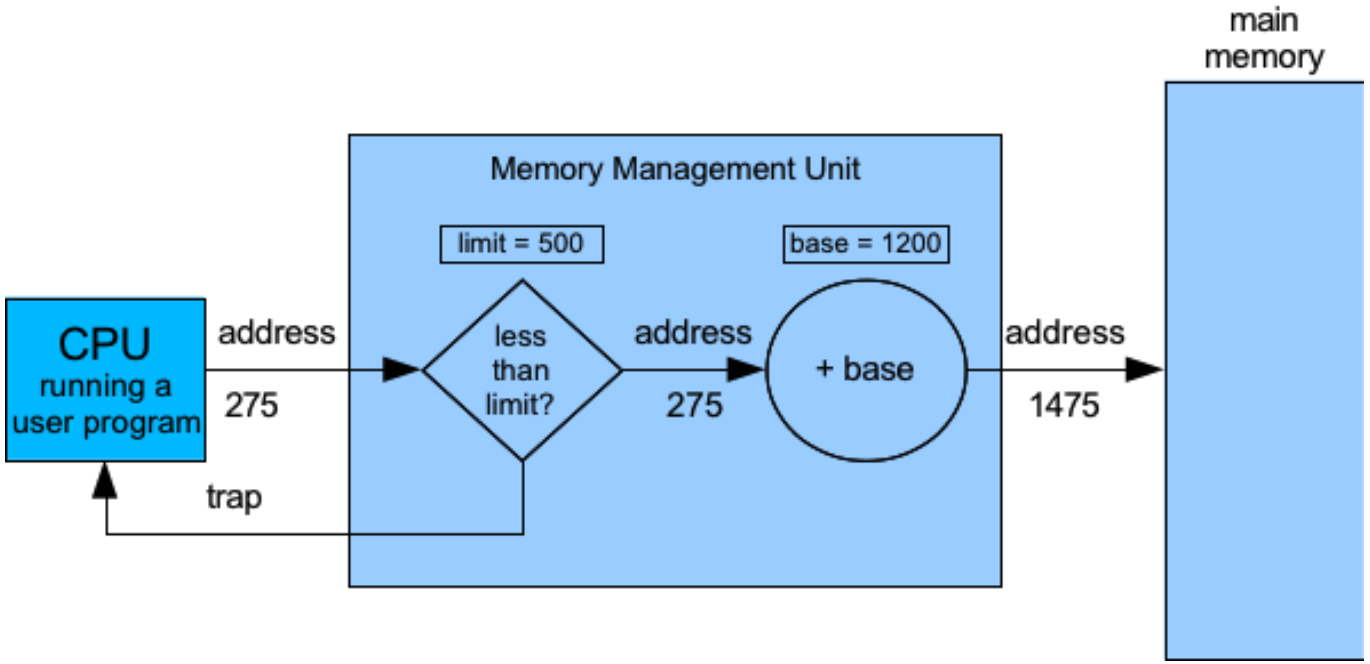
Compaction

- Can we move allocated blocks around to squeeze them together?
  - That's *compaction*, lots of little fragments become one big fragment
  - Can we do this with running programs? We can if we have *execution-time address binding*



Making Programs Relocatable

- We can use the base and limit register to make programs *relocatable*
- Just add the base register to every address



Logical vs. Physical Address Space

- Pretend we're using this base/limit technique to make programs relocatable
  - When your process uses a pointer with the value, say, 0xC3B5, what memory does this pointer really access?
  - Depends on where your program resides in memory

Logical vs. Physical Address Space

- Surprise! There are two ways of talking about of memory addresses
  - *Logical address* : what the user program sees (also called a *virtual address*)
  - *Physical address* : address that's meaningful to hardware, an address of a word in physical memory
  - *Address translation* : run-time conversion from logical to physical addresses
- You (well, your programs) have been lied to all along
- But, it's been a good lie

Logical vs. Physical Address Space

- For compile-time or load-time address binding
  - We don't need a distinction between logical and physical addresses
- For execution-time address binding
  - We must make this distinction
  - So our programs can still be happy even if they sometimes move around in physical memory

The Memory-Management Unit (MMU)

- A hardware device that handles mapping logical to physical addresses
- Typically, part of the CPU
- For example, we can use *base* as a *relocation register*
  - The MMU will add the base register to every logical address
  - Automatically, whenever the CPU is in user mode
  - Fast, simple, easy to implement this kind of MMU

The Memory-Management Unit (MMU)

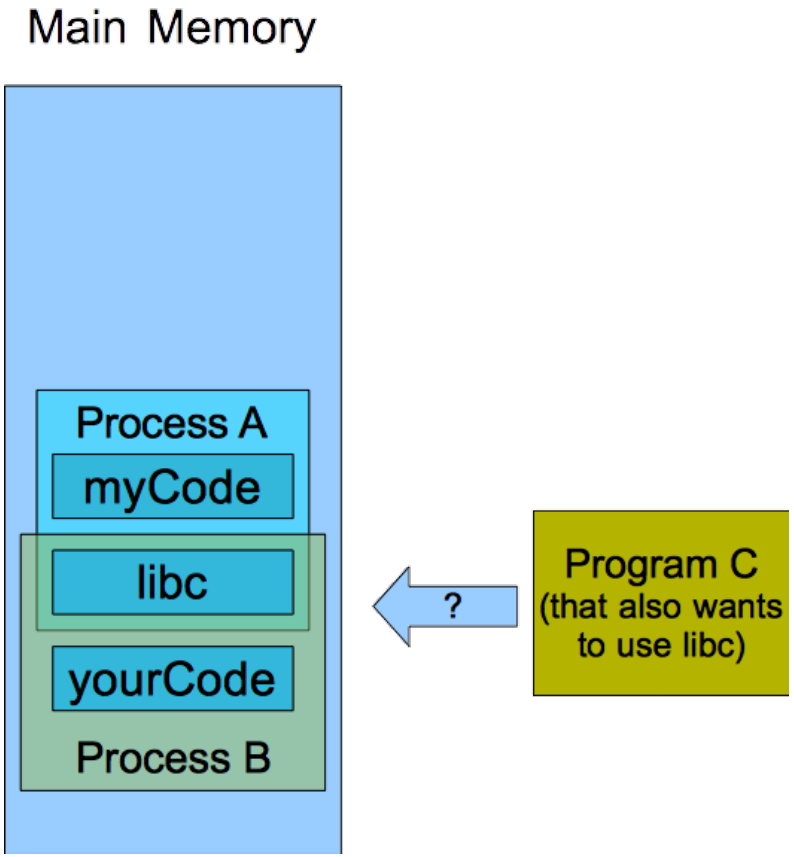
- The user program only sees *logical addresses*
  - It never sees the real physical addresses
  - Really, why would you want to?
- Of course, the OS still has to think about physical addresses (and each process' logical address space).
  - Consider a call like:
    read(fd, buffer, 100);

The process gives us a logical address.

But, we'll need to give the device controller a physical address.

Address Translation Requirements

- However, just a base and limit register won't be enough to let us implement
  - Shared memory
  - Shared libraries
  - Other things we'll talk about later

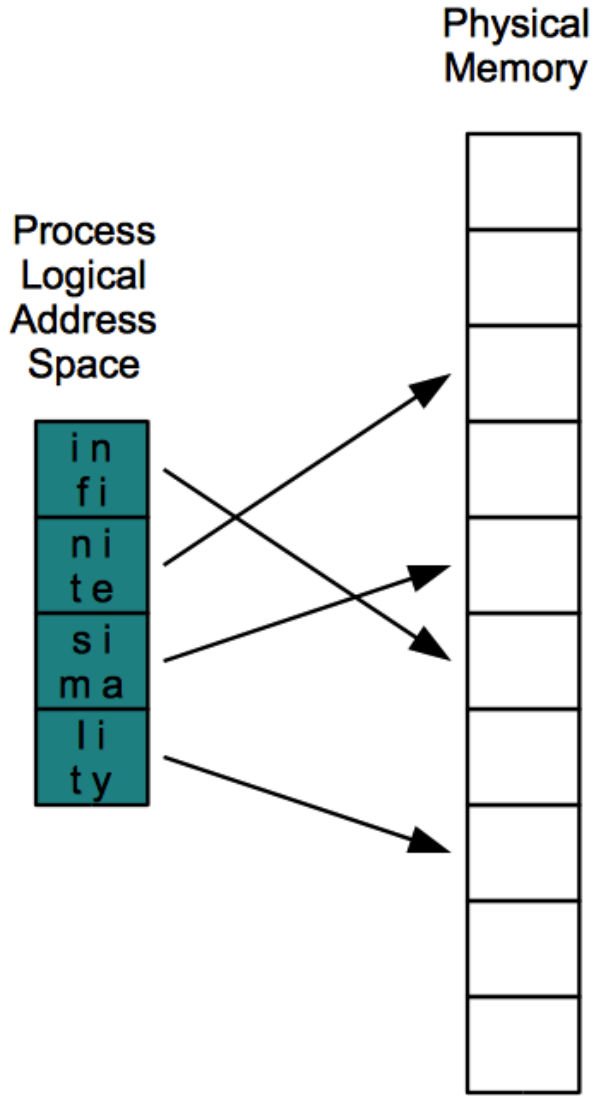


Paged Memory

- Memory allocation would be easier if processes memory didn't need to be **contiguous**
  - Can we do this? Let processes use a little bit of physical memory here, and a little bit there?
  - Process still needs to be able to see its logical address space as contiguous (why?)
  - We certainly need this for shared libraries
- OK, we'll allocate process memory as multiple fixed-sized blocks, called *pages*
- Process memory = a sequence of pages
  - Divide the process' logical address space into fixed-sized *pages*
- Physical memory = a sequence of page-sized *frames*
  - Divide physical memory into frames , each able to hold a page
  - Size typically a power of 2, between 512 bytes and 8,192 bytes

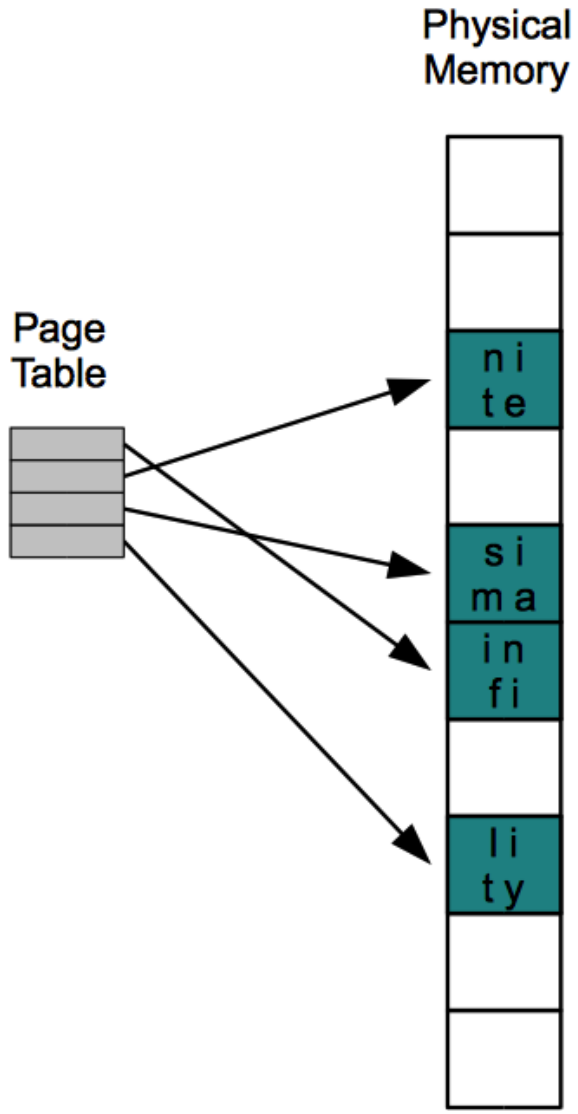
Paged Memory

- To run a program of size *n* pages, we just need *n* free frames (anywhere in memory)
  - We'll tolerate some internal fragmentation
  - But, we'll virtually eliminate external fragmentation



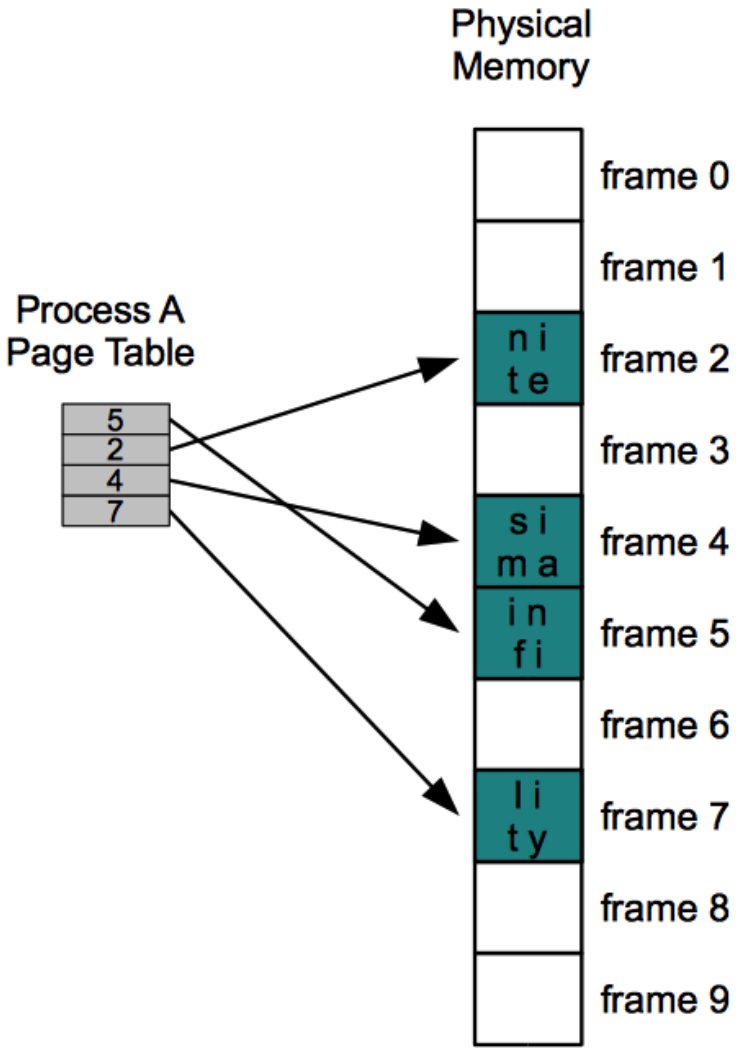
Paged Memory

- But, if process pages could be scattered all over physical memory
  …
  - How do we know how to access a particular byte of a process' memory?
  - We need a table that says where process pages are.
  - We need a *page table*

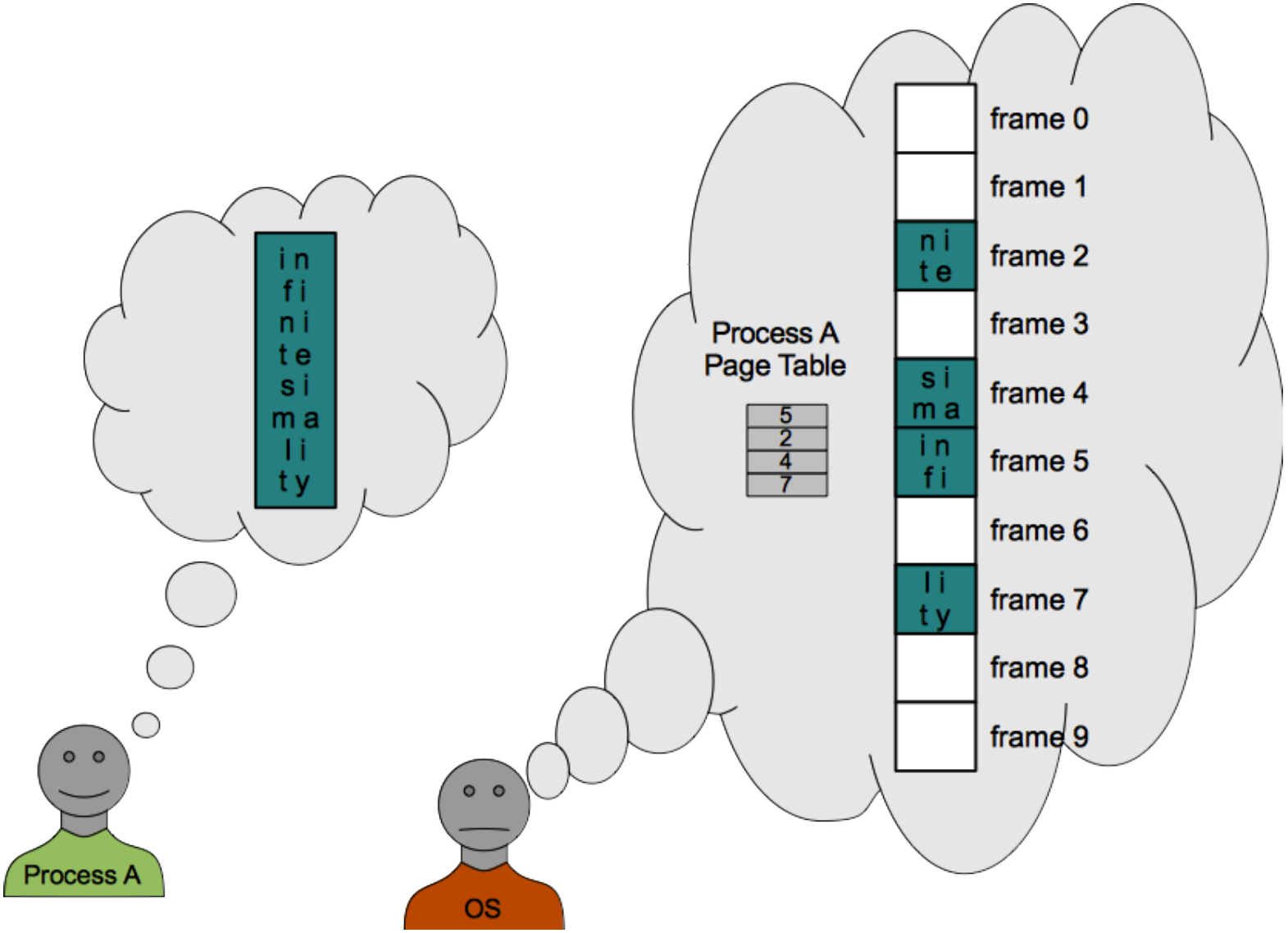


Page Table Organization

- Line 0 says what frame holds page 0
- Line 1 says what frame holds page 1
- And so on
- It's an array of frame numbers (indexed by page)



Logical and Physical views of Memory



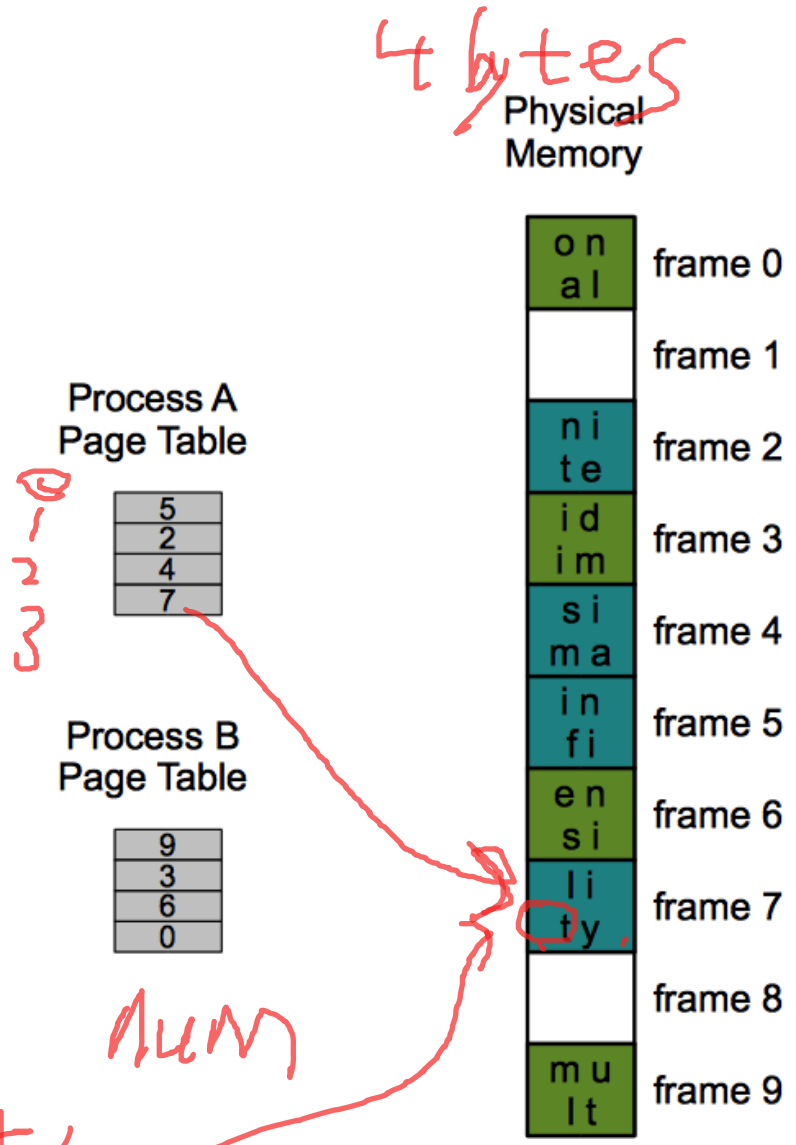
Multiple Processes and Page Tables

- Each process has its own page table

- Fun with Address Translation
  - Byte 3 of process A
  - Byte 8 of process B
  - Byte 14 of process A

14//4 = 3 = Page Num

14%04 = 2 ← offset



Paged Memory Address Translation

- Let *psize* be the length of a page (and a frame)
- Problem: given logical address a_l find physical address a_p

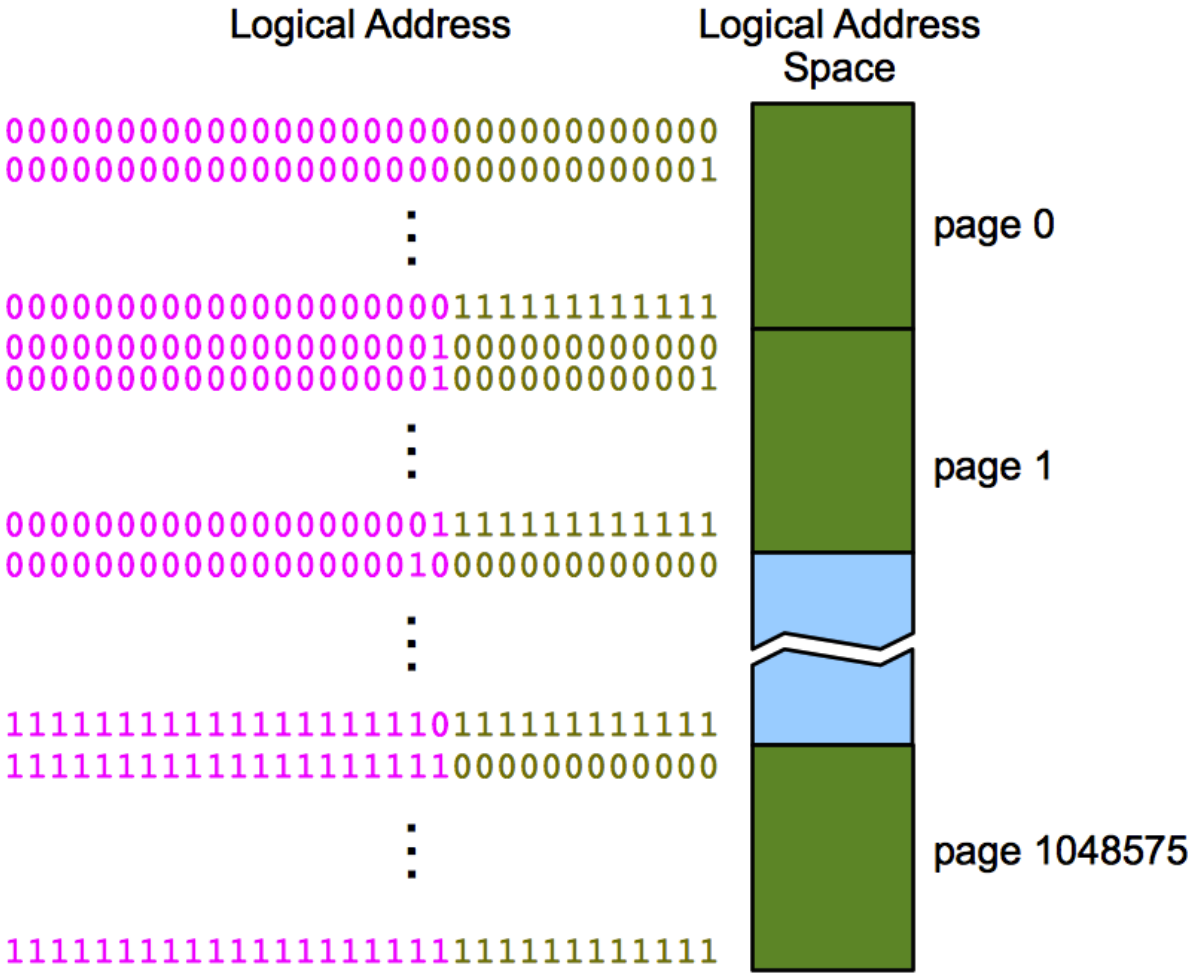
  - First, figure out what page we need.
    Page number: $p = a_l$ div *psize*
  - Then, figure out offset within that page.
    Page offset: $d = a_l$ mod *psize*
  - Figure out memory frame where that page resides
    Frame number: $f = pageTable[\ p\ ]$
  - Build a physical address, offset past start of that frame
    Physical Address: $a_p = f * psize + d$

  - But, is it realistic to expect the hardware to do all this math … on every memory access?

Address Translation Simplified

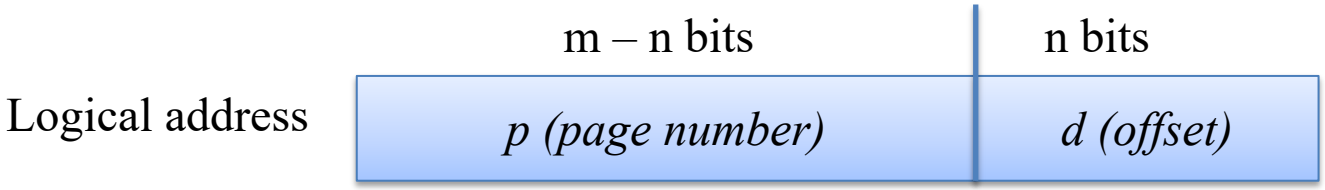
- That's why page size is a power of two
  - The math on the previous slide works out nicely
  - For a logical address space of 2^m words and a page size of 2^n words
  - **Page number :** $p = a_l \text{ div } 2^n$
    That's just the high-order m-n bits
  - **Page offset :** $d = a_l \text{ mod } 2^n$
    That's just the low-order n bits
  - **Frame number:** $f = pageTable[\ p\ ]$
  - **Physical address :** $a_p = f * 2^n + d$
    The multiply here is just an n-bit left shift

Thinking About Logical Addresses

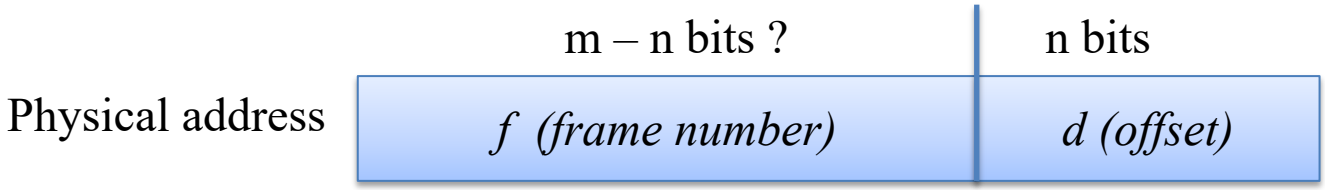


Address Translation Simplified

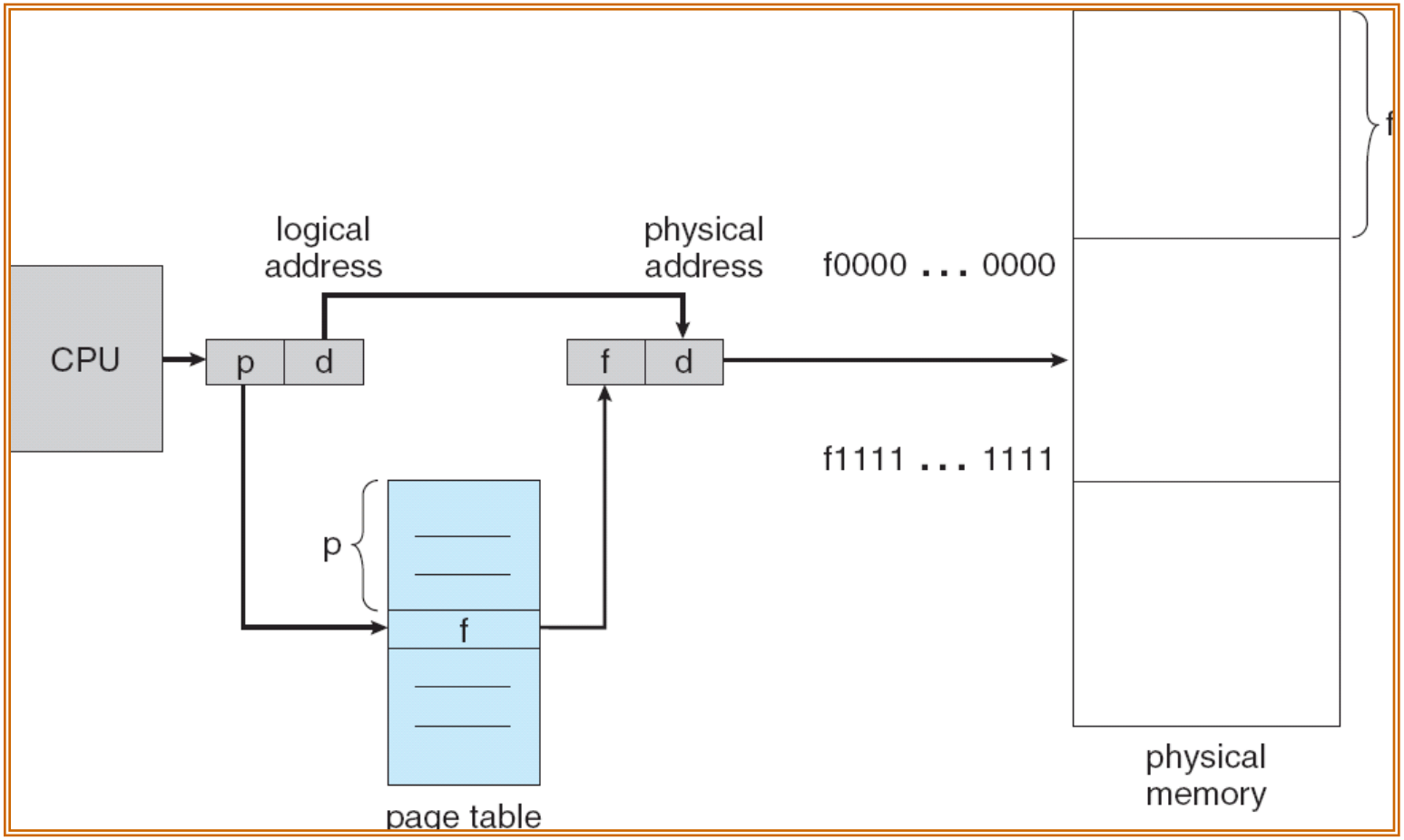
- So, page number and offset are just bit fields in the logical address



- Look up the frame number *f* on line *p* of the page table
- Then, build the physical address:

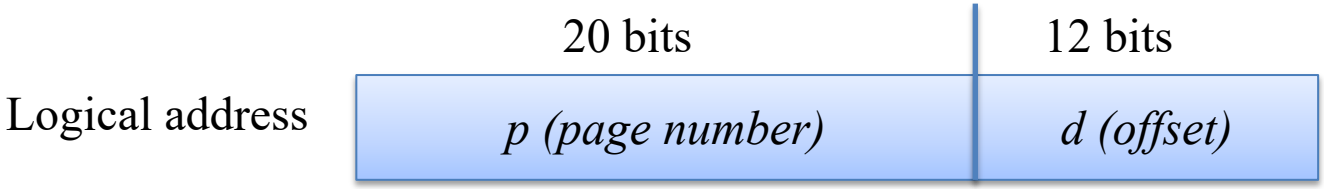


Paged Memory Hardware



Page Table Example

- Consider a byte-addressable computer system with:
  - A 32-bit logical address (i.e., 2^{32} addresses)
  - A 32-bit physical address
  - A page size of 4 KB (2^{12} bytes)
- So, a logical address would look like:



- So, how many entries would the page table have?

Implementation of the Page Table

- We have to store the page table somewhere, how about main memory
  - That means we need a little more memory for each process
- Memory translation is done in hardware
  - We need to tell the hardware where the (current) page table is located
  - We need a *Page Table Base Register* (PTBR)
  - OS will need to save/restore this during a context switch
  - But, we don't need base/limit registers anymore (why?)

Storing The Page Table

- Recall, a 32-bit logical & physical address, 4-KB page size

- How should we store a line of the page table?
  - It's like a pointer to the start of a frame in memory
  - So, we could store it in 4 bytes, like we would store any 32-bit pointer
  - But, here, we could only have 2^{20} frames, so we only need 20 bits of each page table line
  - So, we'll have some bits left over, what could we do with these?

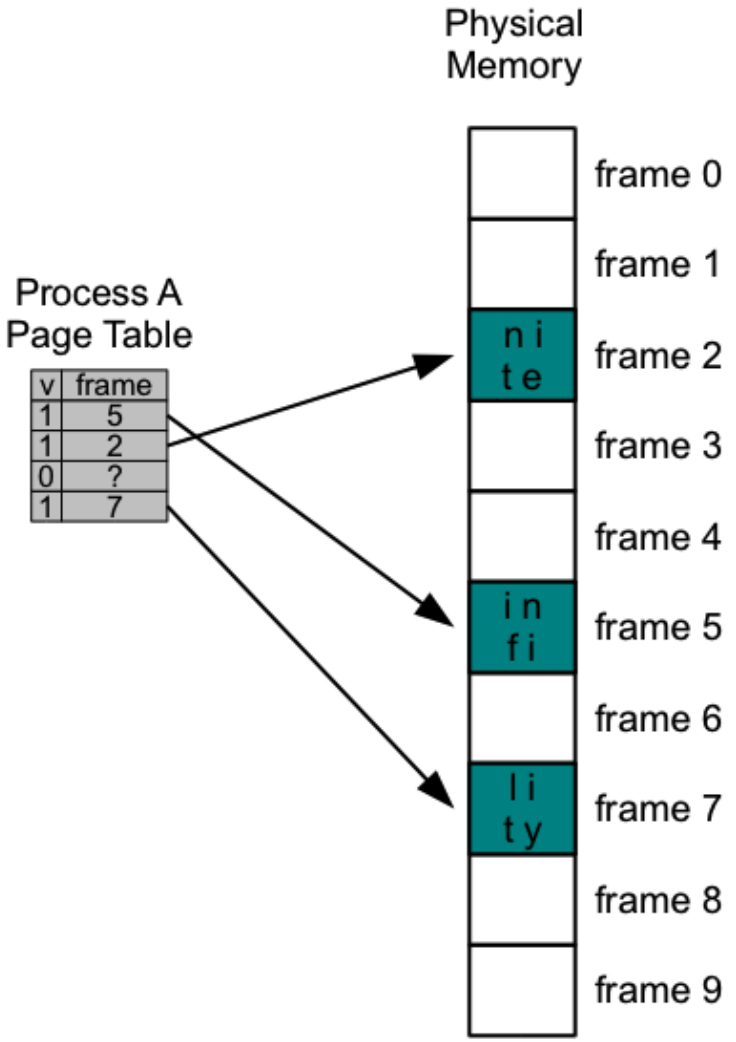
But, 20 bits only requires 3 bytes. Should we store the page table as 3 bytes per entry? … No. Bad idea.

Extra Bits in each Page Table Entry

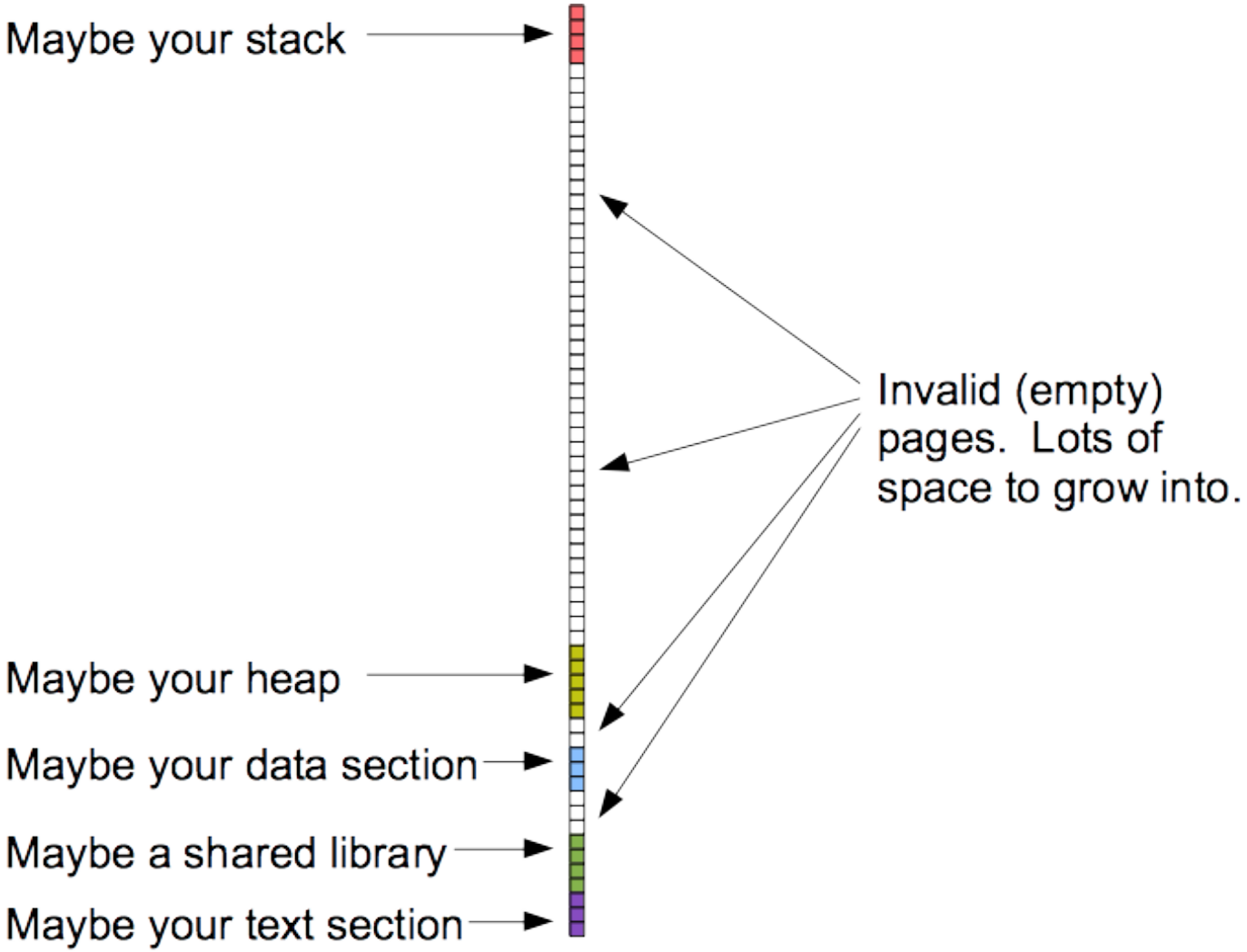
- What can we do with extra bits in each page table line?
- With help from the hardware, we'll get a lot of value out of these
- Typically, we'll count on:
  - **Valid/invalid bit**
    Set if the page table line really points to a frame
    Otherwise, the process can't use that page

Implementing a Sparse Logical Address Space

- Using the valid/invalid bit:
  - We can omit pages the process doesn't need
  - Just a few pages for a small process
  - Give the process room to grow



Your Logical Address Space … is a Very Big Place



Plenty of Extra Page Table Bits

- Typically, we'll expect the hardware to provide
- **Valid/invalid bit**
- **Read-Only bit**
  - True if process can't write to this page
- **Modified bit** (dirty bit)
  - Set by the hardware whenever the page is written to
- **Reference bit**
  - Set by the hardware whenever the page is referenced (read or written)
- **No Execute bit**
  - True if a process can't execute code on the page
  - A security mechanism, your stack is typically not executable.

Be the MMU

- Let's try some memory references
- 8-bit logical addresses, 32-byte page size
- Read 01001010
- Write 11100111
- Read 00101110
- Write 10010011

V	Rd	M	Re	Frame
1	0	0	1	11001
1	1	0	0	00101
0	0	0	0	01101
0	0	0	0	00000
1	0	0	0	11011
1	1	0	0	00001
0	0	0	0	10100
1	1	0	1	00010

- Notice, physical and logical address spaces don't have to be the same size
- How much memory should you use to store a line of this page table?

Respecting the Valid and Read-Only Bits

- What should we do if:
  - A process tries to use an invalid page?
  - A process tries to write to a read-only page?

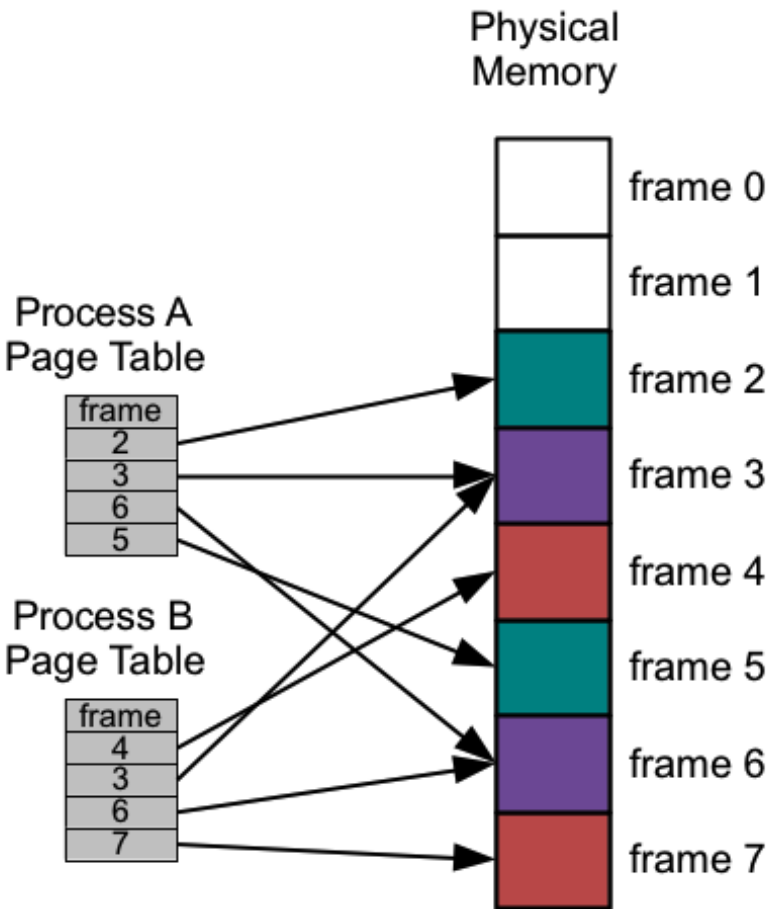
- OS can do more interesting things than just kill the process

Paged Memory Tricks

- We got to the idea of paged memory via concerns over memory allocation and fragmentation
- But, paged memory is really about creating a nice execution environment for processes
- We can use paged memory to implement:
  - Sparsely populated logical address space, with room for our programs to grow
  - Shared memory
    For IPC, for shared libraries or for shared program code
  - Copy-on-write
    For efficient POSIX fork()

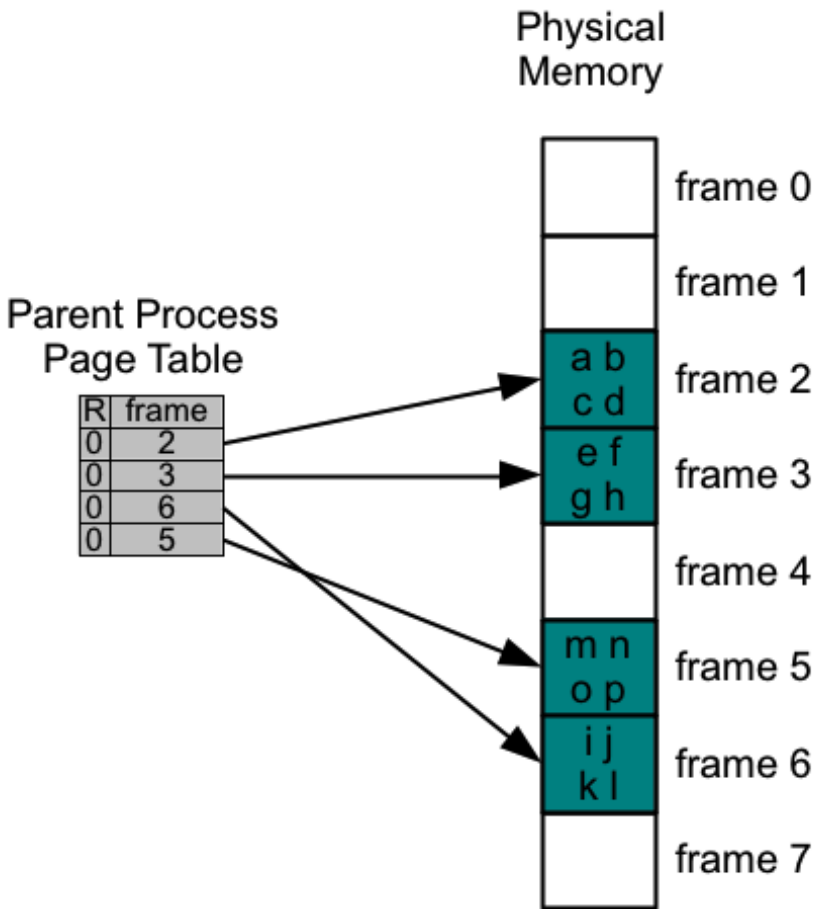
Shared Memory via Paged Memory

- We just need to let parts of the page tables point to the same frames
- In page-sized blocks, any number of processes can share any regions of memory



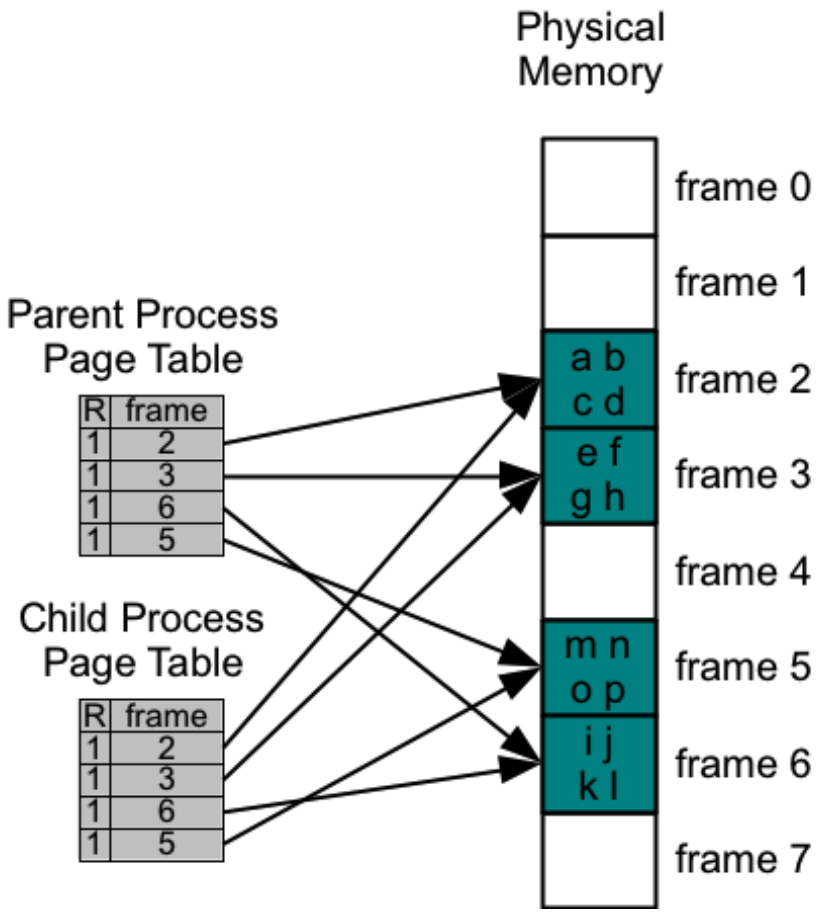
Copy On Write

- Imagine, we have a nice happy process, permitted to write to any of its pages



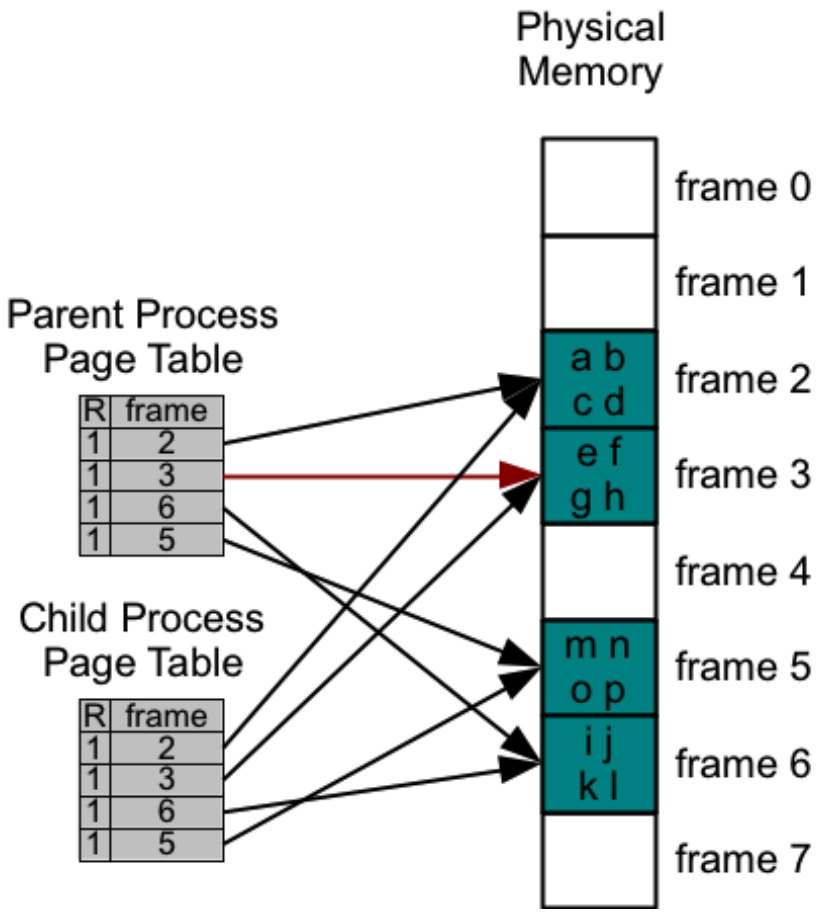
Copy On Write

- The process calls fork, now we have a child that needs a copy of the parent's memory
- Quick, we need a copy of all the parent's memory.
- But, we're lazy, let's just let it use the same memory frames.
- But, mark them as read-only (for now)



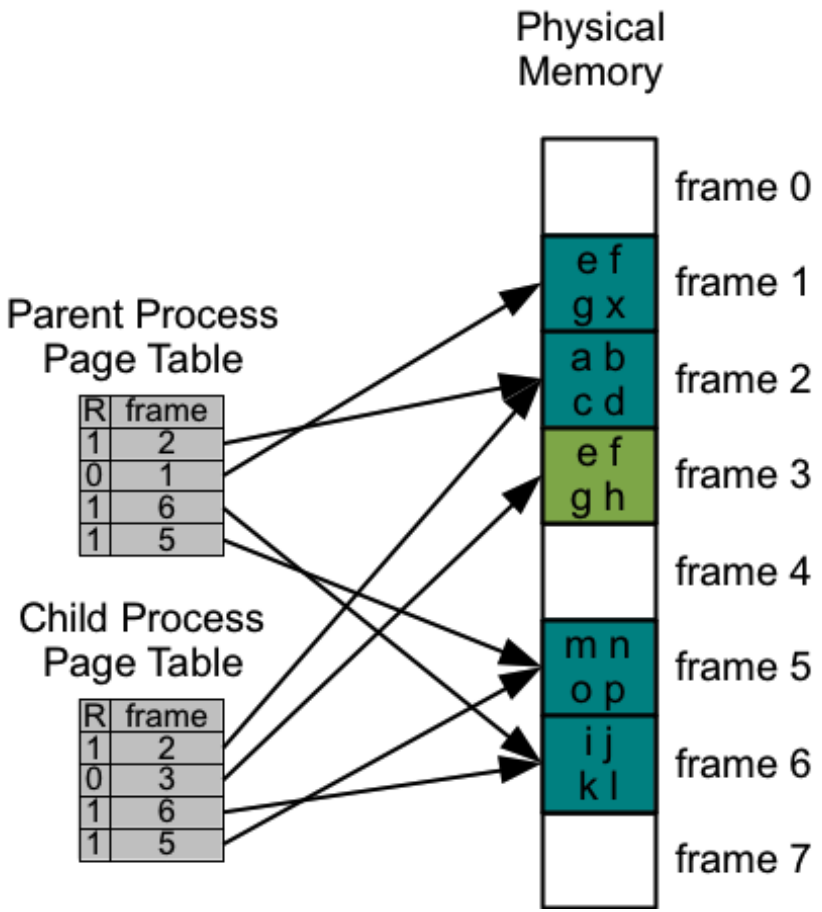
Copy On Write

- What if either process tries to write to its memory?
- It should be able to, but this will trap to the kernel



Copy On Write

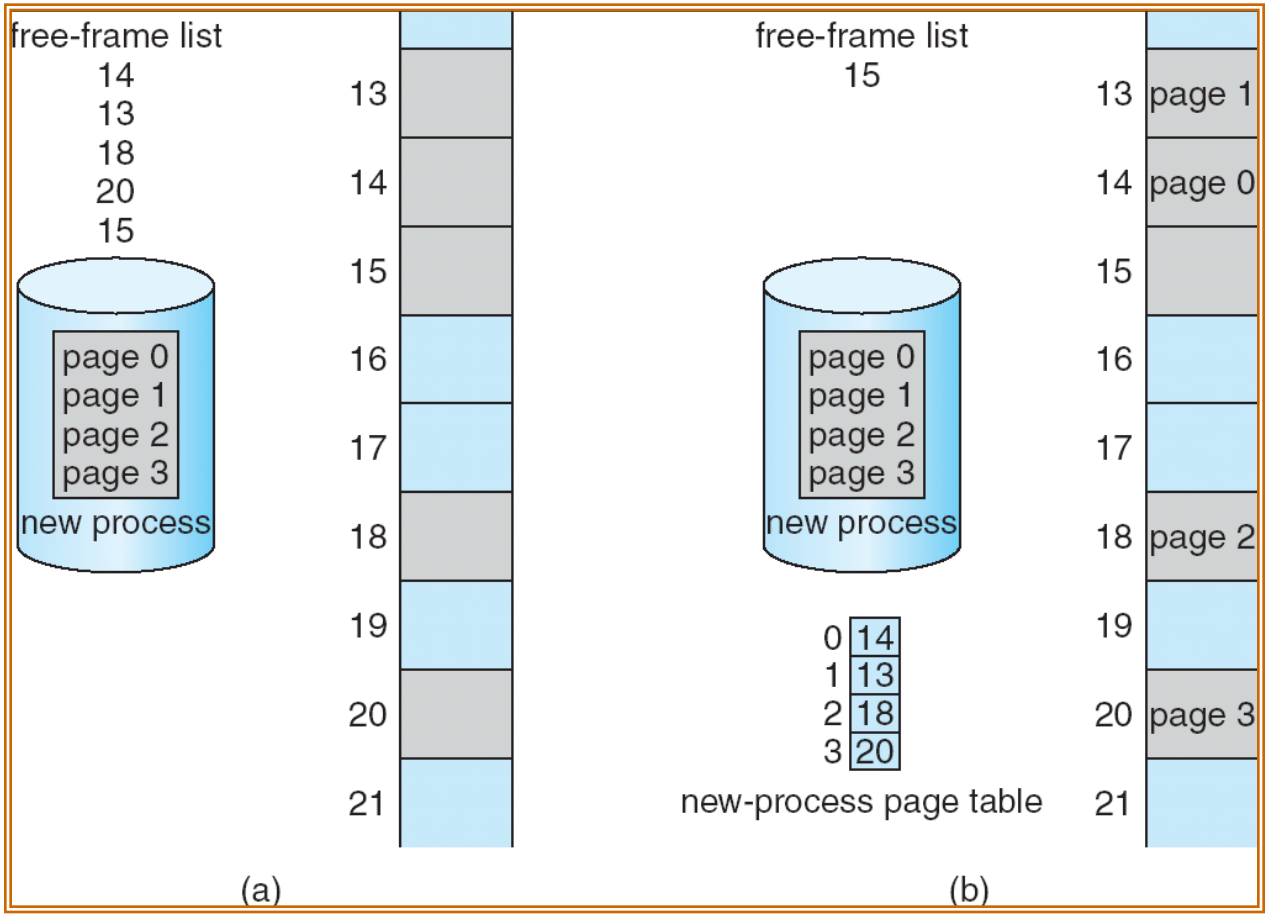
- Great. The kernel can make things right.
- We'll copy the frame, and then let both processes write to their copy.
- This is *like* a system call, but the process didn't really intend to make it.
- And, we only pay to copy the frames we need to.



Frame Allocation

- Now, memory allocation just becomes frame allocation
- When some process P_1 starts
  - OS must find enough frames for P_1
  - OS must build a page table for P_1
- OS needs to keep up with used/available frames
  - OS will maintain a *frame table*, an array of records, one for each frame
  - Records what's in each frame
  - And, whether the frame is free / occupied

Free Frame Management



Before allocation

After allocation

Page Table Performance

- Paged memory sounds expensive
- Every memory reference requires
  - A read from the page table
  - A read/write to the desired page
  - That's two main memory references for every read/write
- Need to speed this up for typical programs
  - Normally, we depend on a cache to speed memory access

Speeding up Memory Access

- Let's have a special cache, just to speed up address translation
- The *Translation Look-aside Buffer* (TLB)
  - A special cache for each CPU core
  - Cache for a subset of the page table
  - Maps page number straight to frame number (really, from page number to a copy of the page table line)

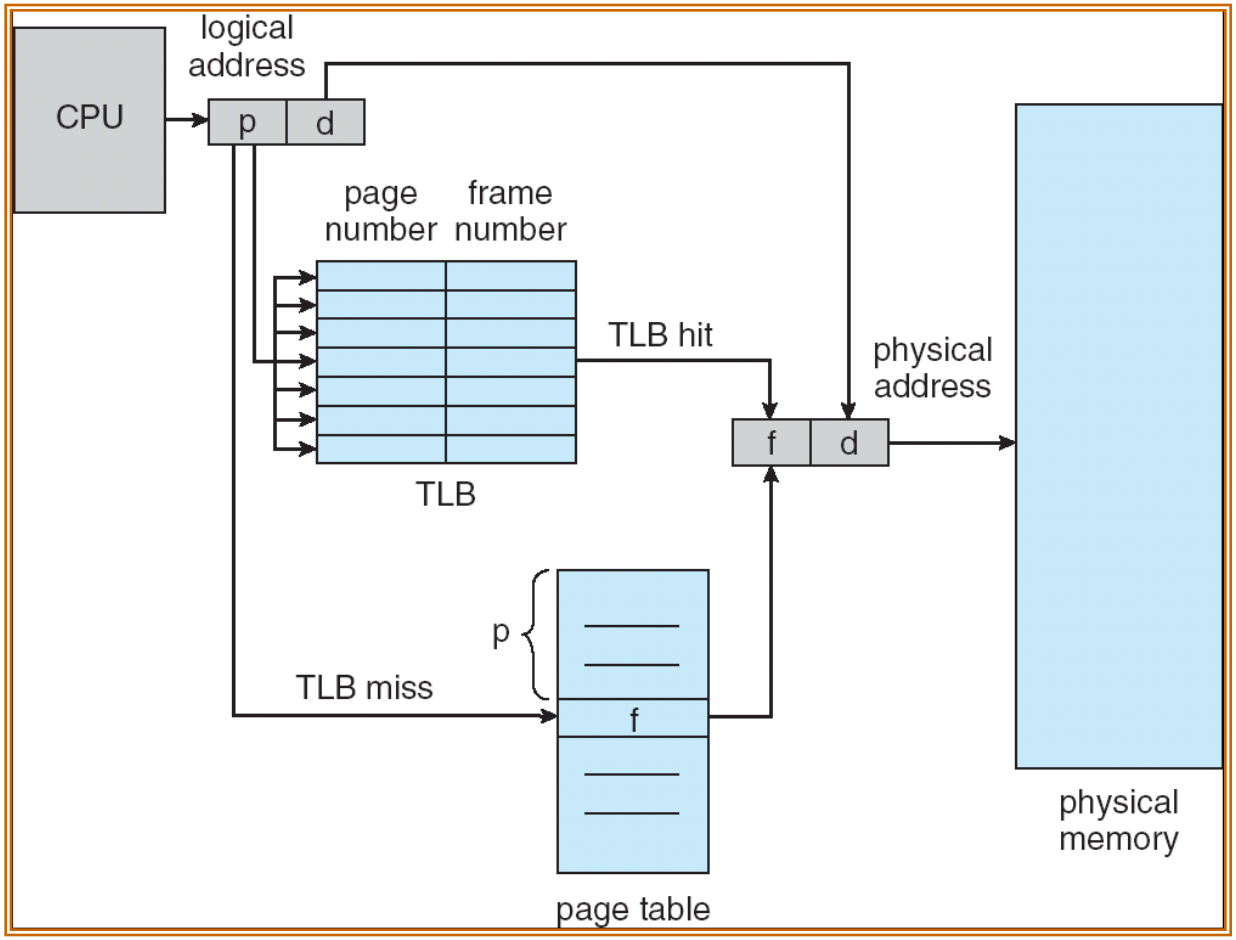
The TLB

- The TLB is an associative memory, with parallel search by page number

Page	Rd	M	Re	Frame
7	1	0	1	27
3	0	1	1	6
15	0	0	0	3
0	0	1	1	9

- Address Translation (p, d)
  - If *p* is in the TLB get the frame number (a TLB hit)
  - Otherwise, lookup the frame number in the page table (a TLB miss)
    - But, add a new entry to the TLB (for the future)
    - We'll need to remove something (normally a hardware choice)

Paging with the TLB



Be the New MMU

- As before, assume a 32-byte page size
- Say we have a 2-entry TLB

- Read 01000101
- Write 00000111
- Write 01000110
- Read 10010011

Page Table

V	Rd	M	Re	Frame
1	0	0	1	11001
1	1	0	0	00101
1	0	0	0	01101
0	0	0	0	00000
1	1	0	0	11011
1	0	0	0	00001
0	0	0	0	10100
0	0	0	0	00010

TLB

Page	Rd	M	Re	Frame
2	0	0	1	01101
0	0	1	1	11001
4	1	0	1	11011

Effective Access Time

- Assume TLB Lookup is 10 ns
- Assume memory cycle time is 100 ns
  - On a hit: 10 + 100 → 110 ns
  - On a miss: 10 + 100 + 100 → 210 ns
- *TLB Hit Ratio* (α): fraction of memory access that get TLB hits
- **Effective Access Time :** $110\ \alpha + 210\ (1 - \alpha)$
- How large can we expect α to be?
  - We'd like it very close to 1
  - With some *locality of reference*, maybe it can be

That's a Big Page Table!

- A page table could be quite large
  - Recall, a 32-bit logical/physical address space, with 4 KB pages
  - With valid/invalid bit, we don't need all the pages
  - But, we still need 2^{20} page table entries, 4-bytes each
    That's 4 MB, just for the page table

With 260 processes on my laptop, that would be about 1/16 of my total memory, just for page tables.

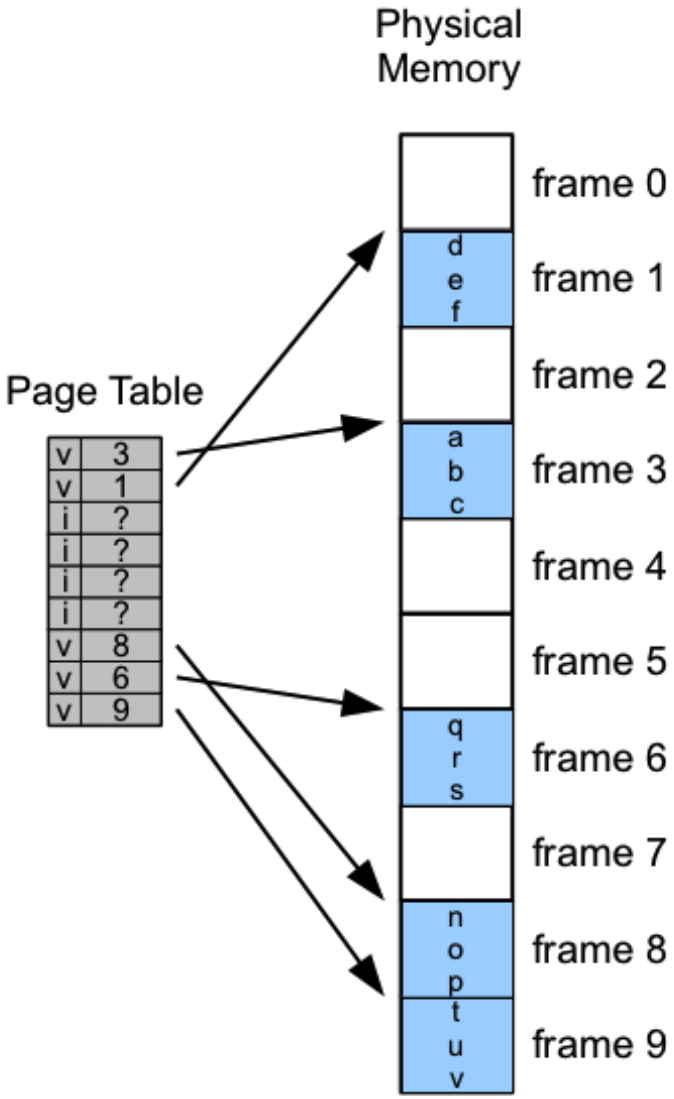
  - But, most processes don't really need an address space that large
- Four techniques for dealing with this:
  - Variable page table size
  - Hierarchical paging
  - Hashed page tables
  - Inverted page tables.

Variable Page Table Size

- Maybe we just need part of the page table
  - We could have a *Page Table Length Register* (PTLR)
  - Tells the hardware the length of the (current) page table
  - So, we just need the first part of a potentially very large page table
  - Saved and restored at context switch time

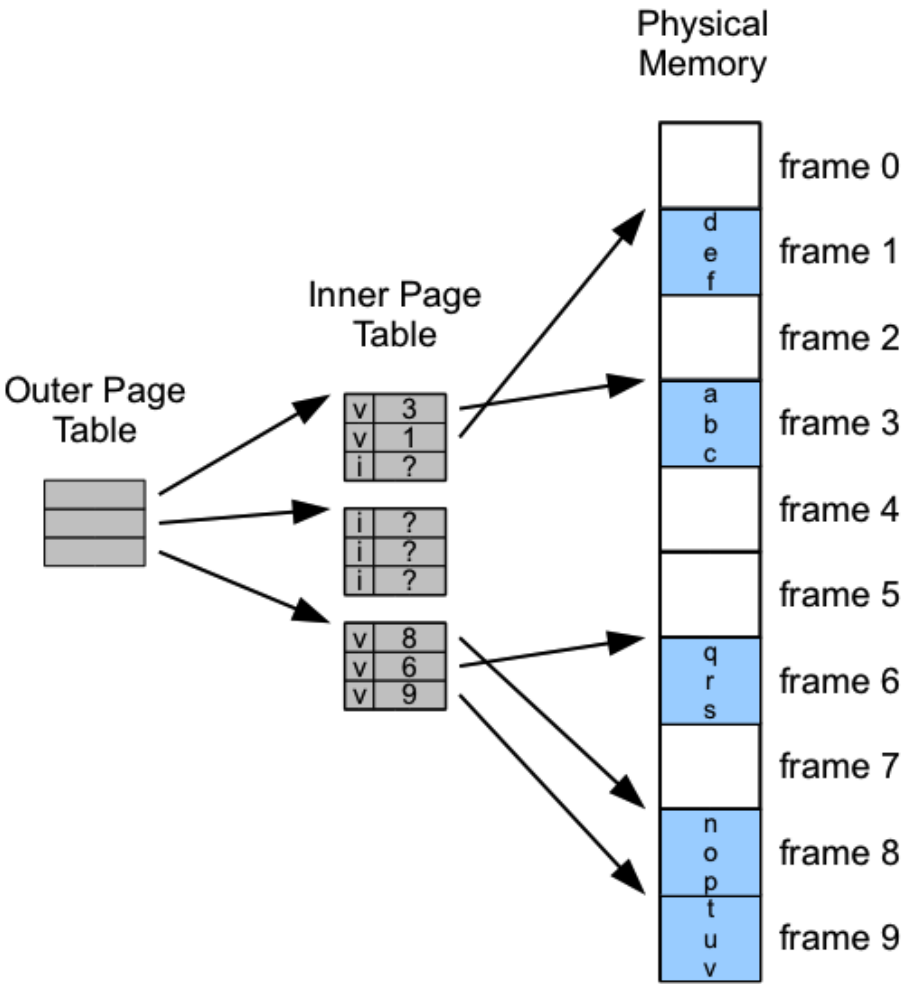
Hierarchical Page Tables

- So, it's hard to find contiguous memory regions for lots of big page tables
  - Great, we already figured out a good way to solve this kind of problem
  - What did we do?



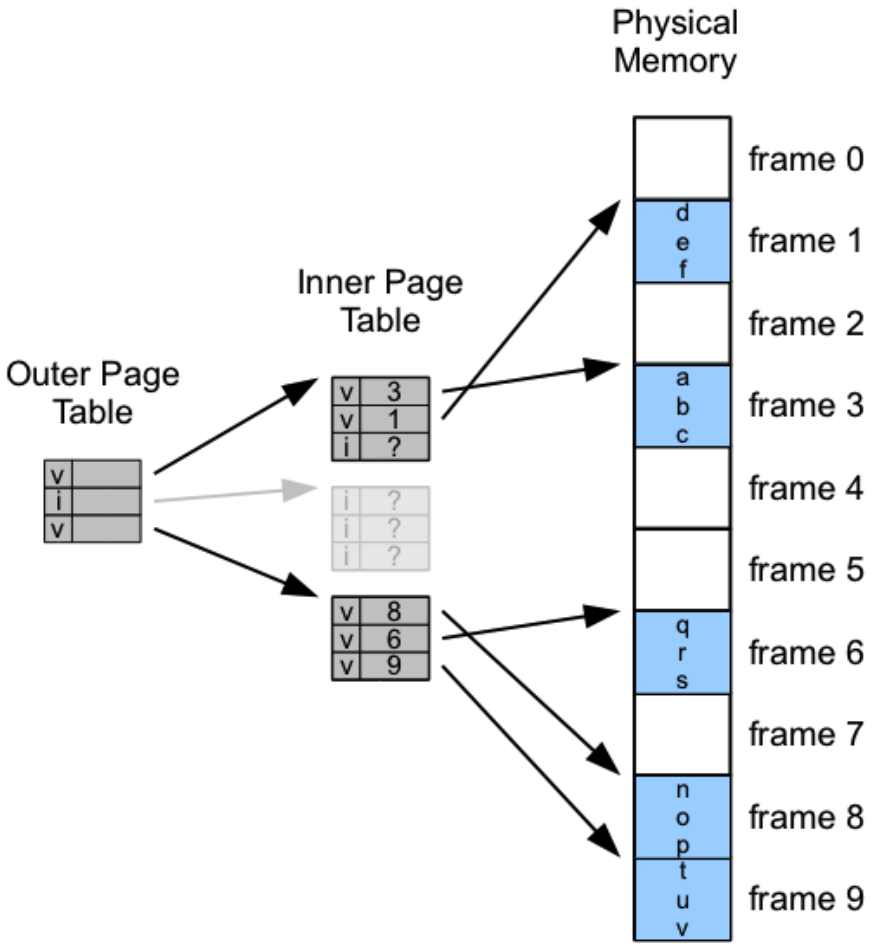
Hierarchical Page Tables

- You guessed it, we'll page the page table
  - So, we'll break the page table into page-sized sections
  - Then, we'll need another, outer page table
  - … to point to the starts of all the pages of the inner page table.



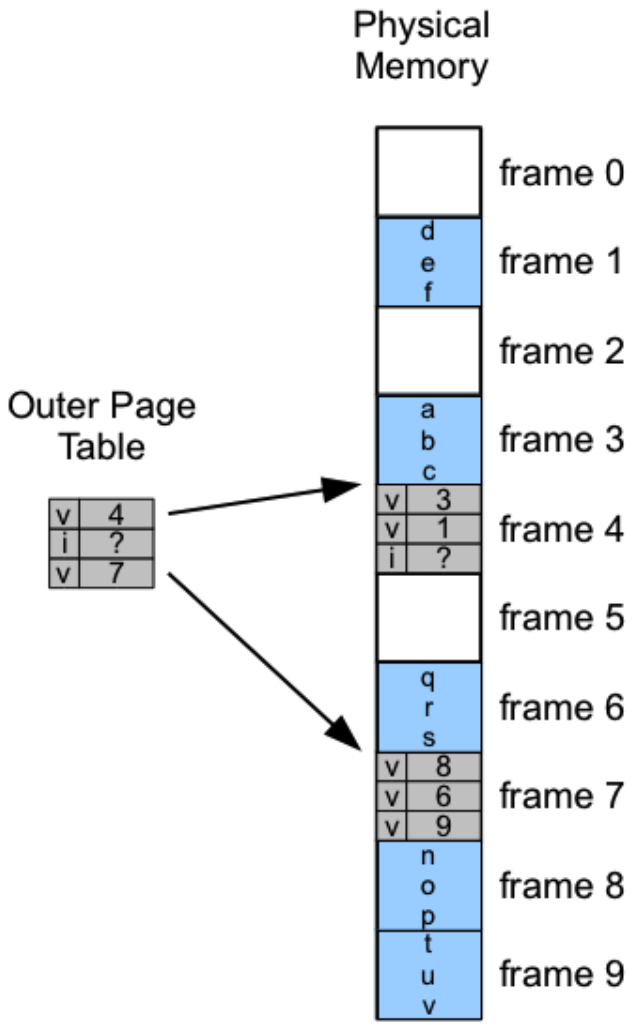
Invalid Page Table Pages

- Lines of the outer page table can be valid/invalid
- This can let us mark pages of the inner page table as invalid
  - Don't need as many pages in the inner table
  - Easily omit large sections of logical address space
  - Then, fill them in later if the process grows.



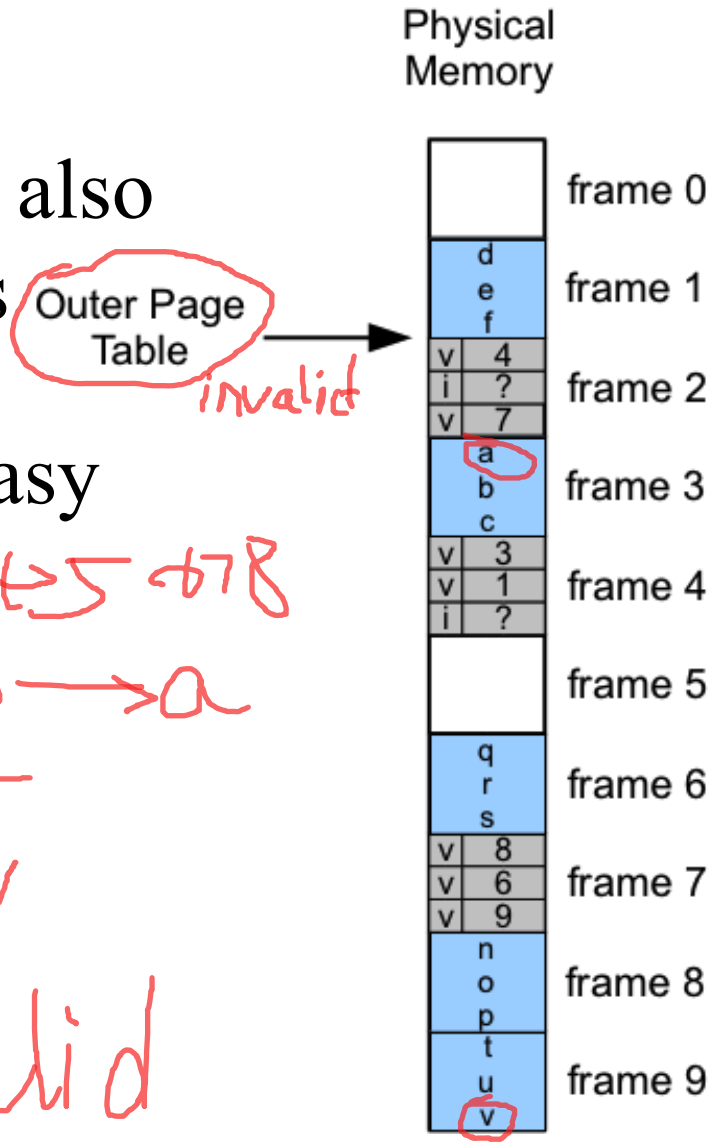
Storing the Inner Page Table

- Of course, pages of the inner page table go into memory frames
- So, it's really more like this picture



Storing the Outer Page Table

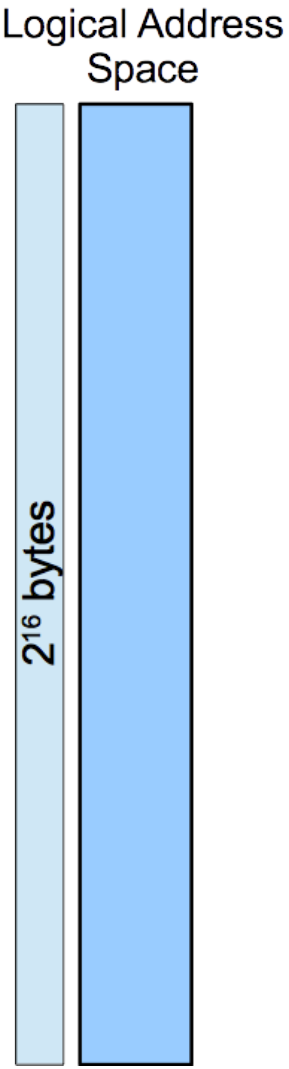
- Of course, the outer page table goes in main memory also
- So, it's really more like this picture
- Let's try it out with some easy examples:
  - read 0 (first byte on the first page)
  - read 26 (last byte on the last page)
  - read 9 (first byte on page 3)



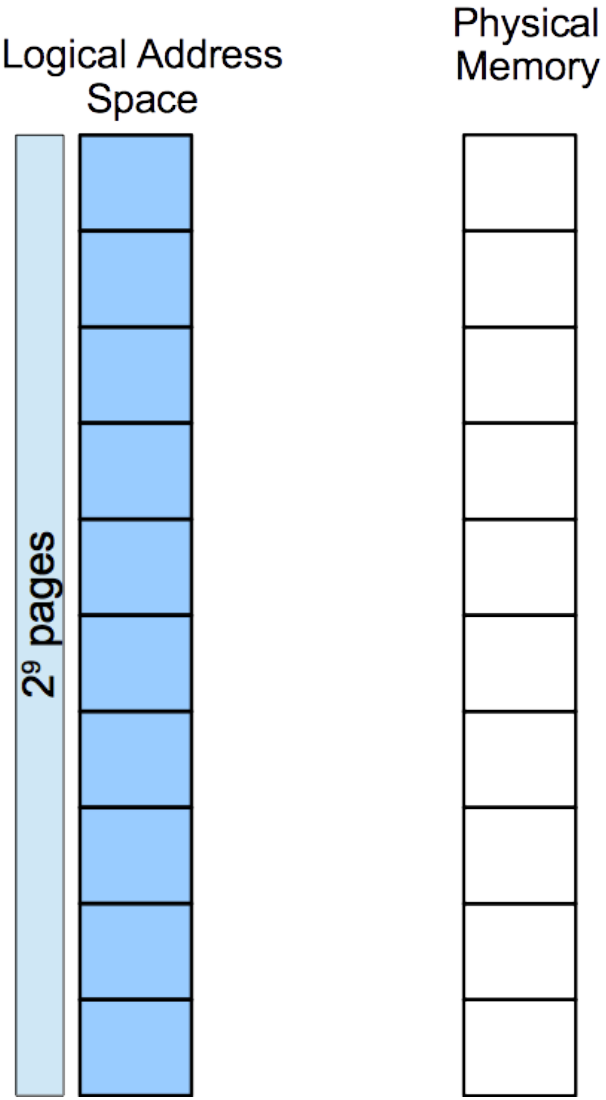
A Larger Example

- Imagine we have a 16-bit logical address
- Assume the page size is 128 bytes (that's 2^7 bytes)
- Assume each page table entry requires 2 bytes.

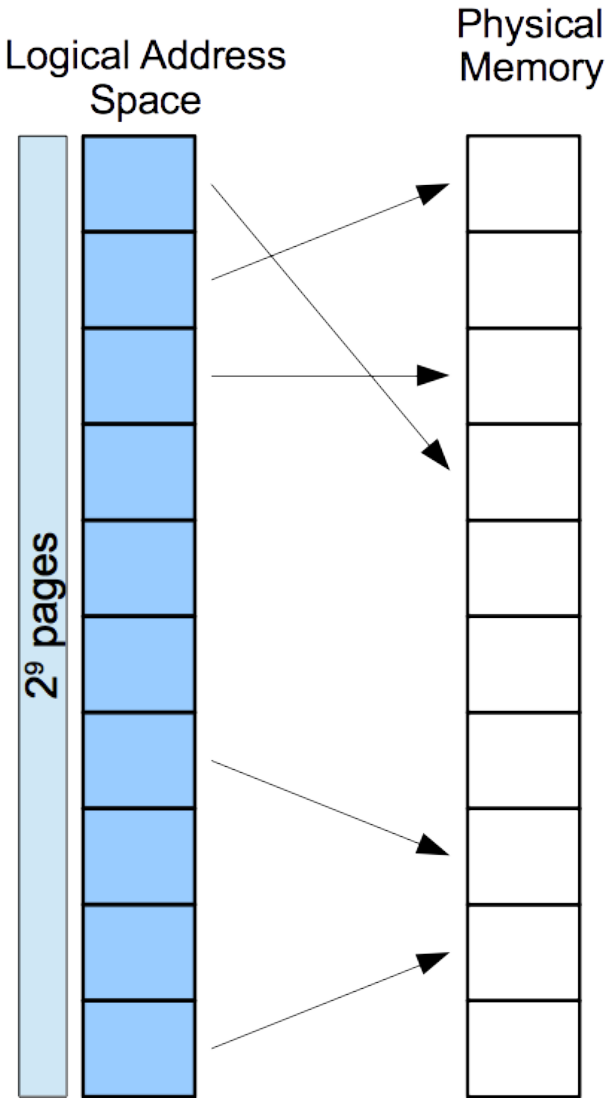
2^{16}-byte logical address space



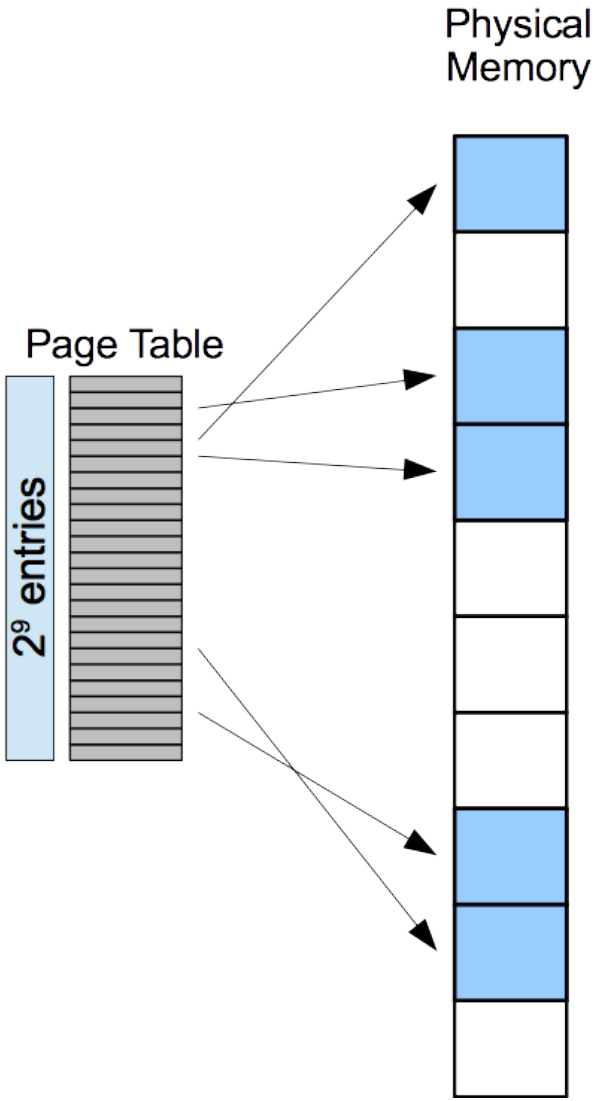
Broken into (up to) 2^9 Pages



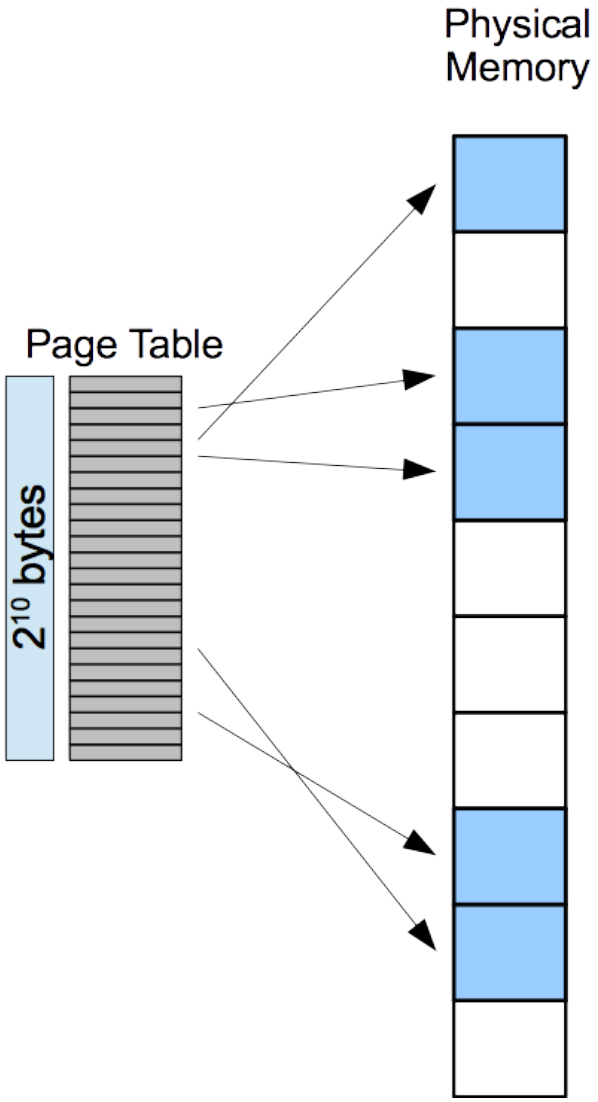
Each Page Occupying an Arbitrary Frame



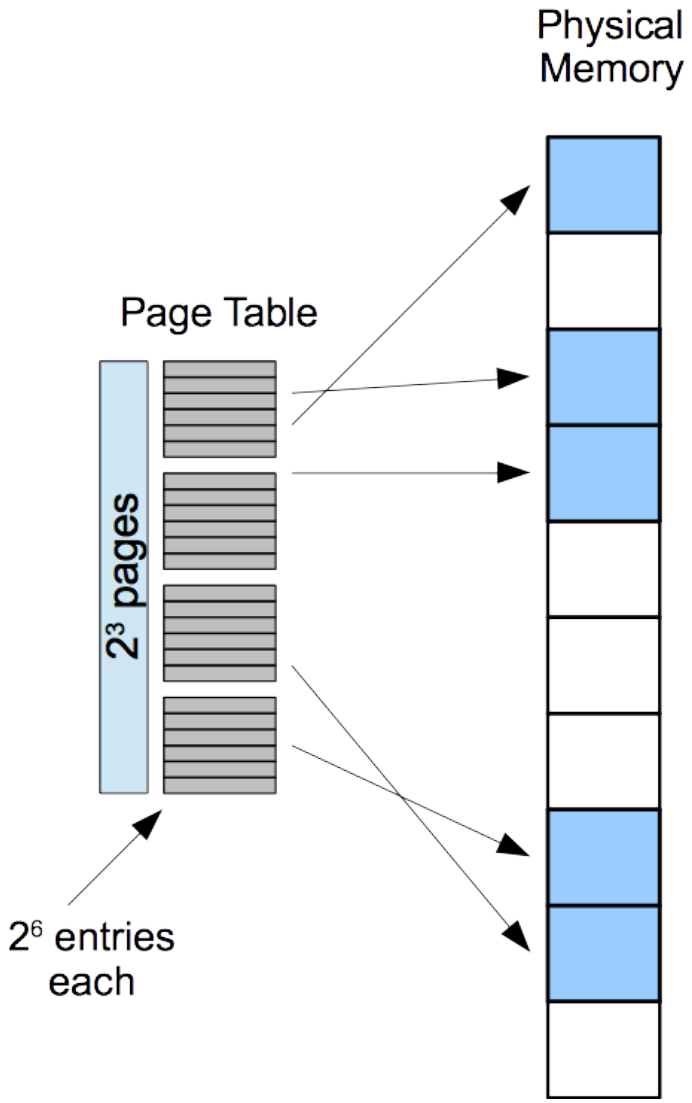
A Page Table, with one Entry per Page



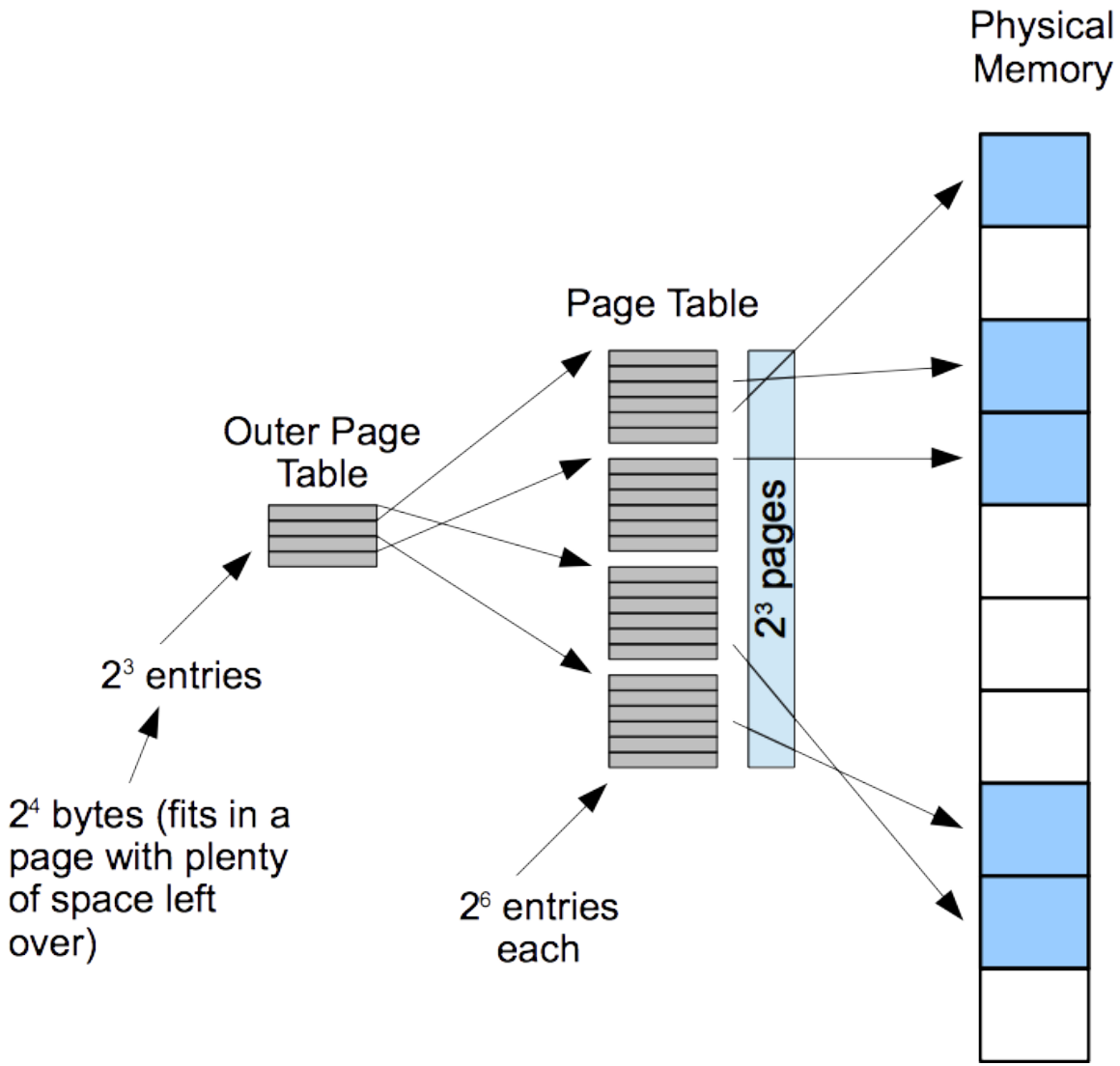
Assume 2 bytes Per Page Table Entry



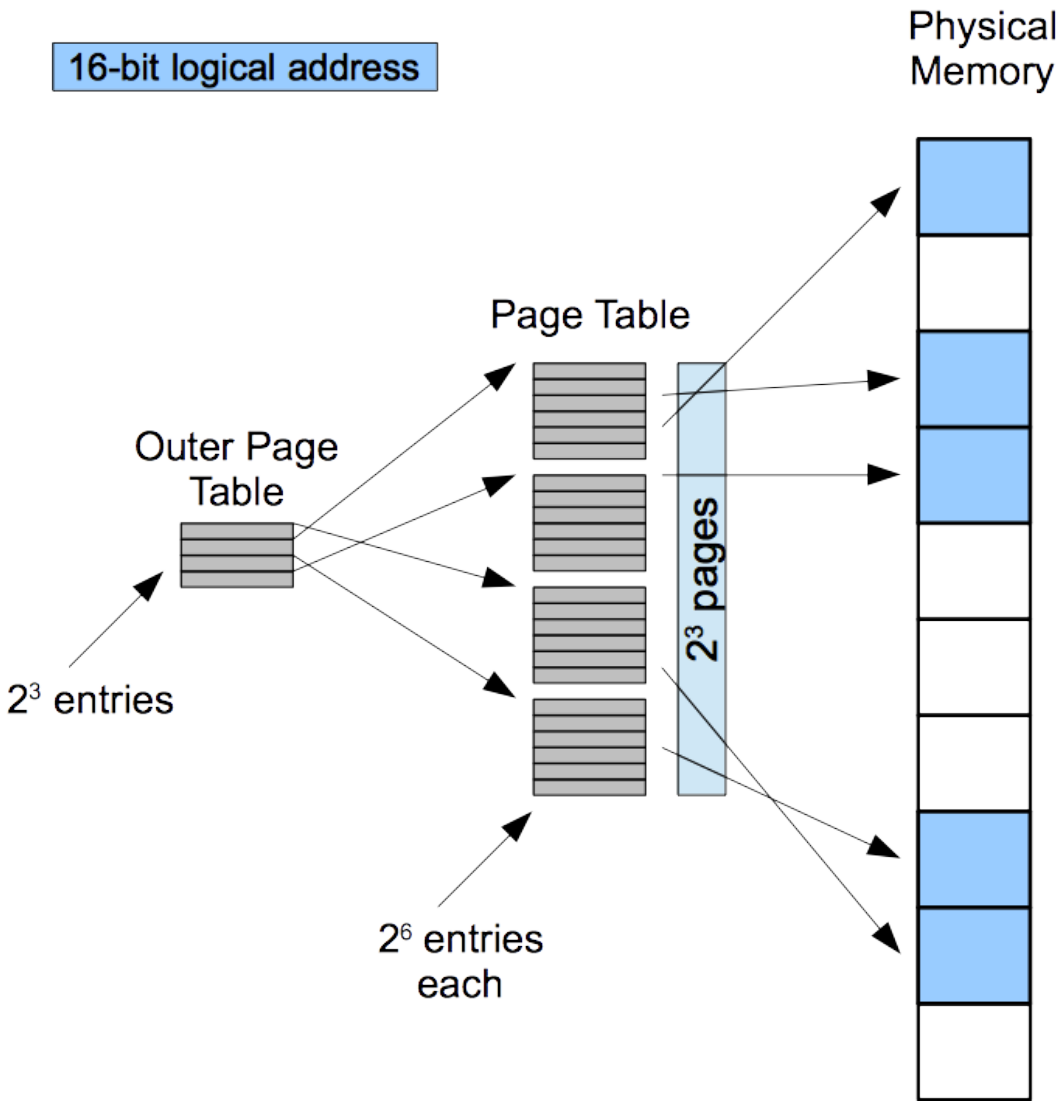
Break The Page Table into Pages



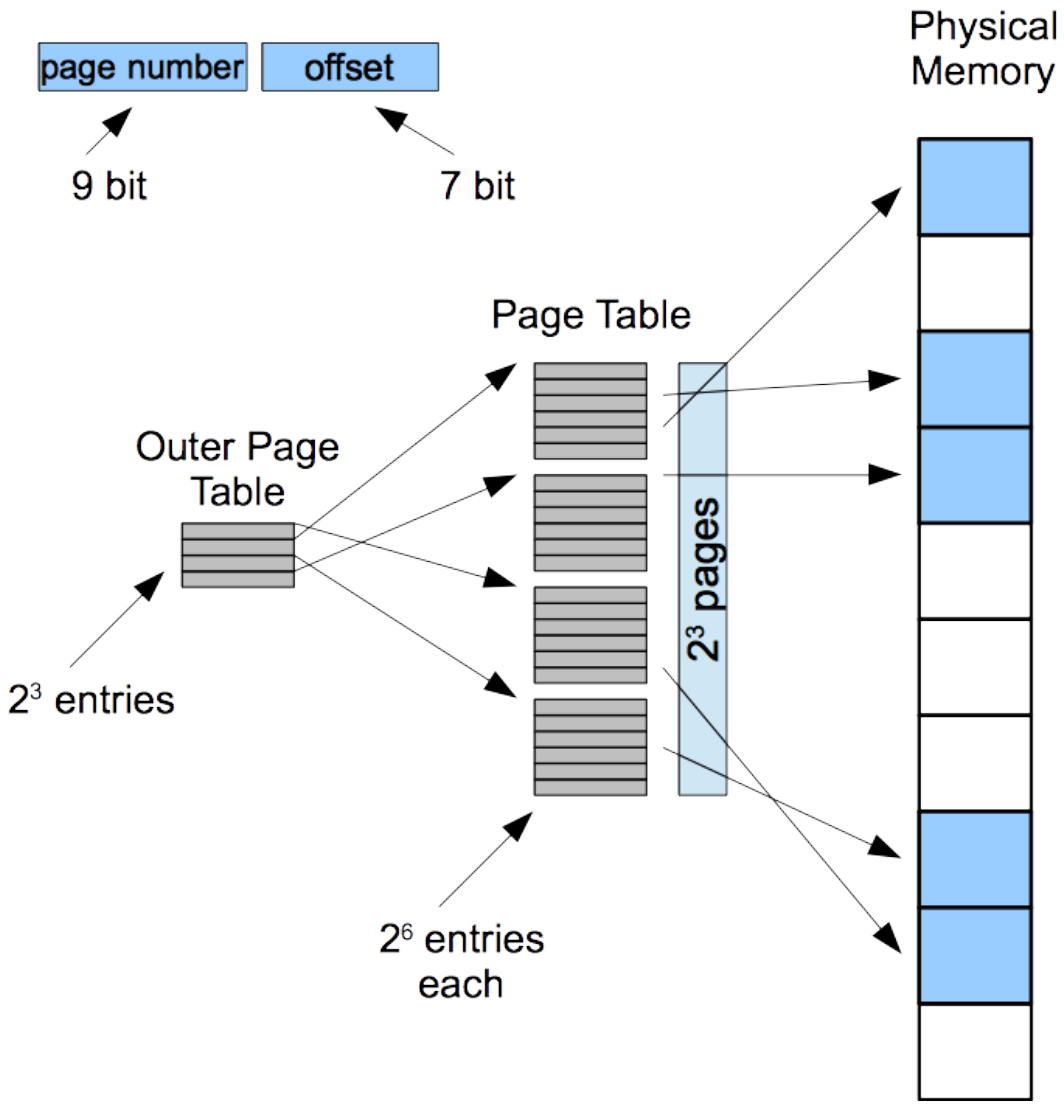
Another Table To Remember Where these Go.



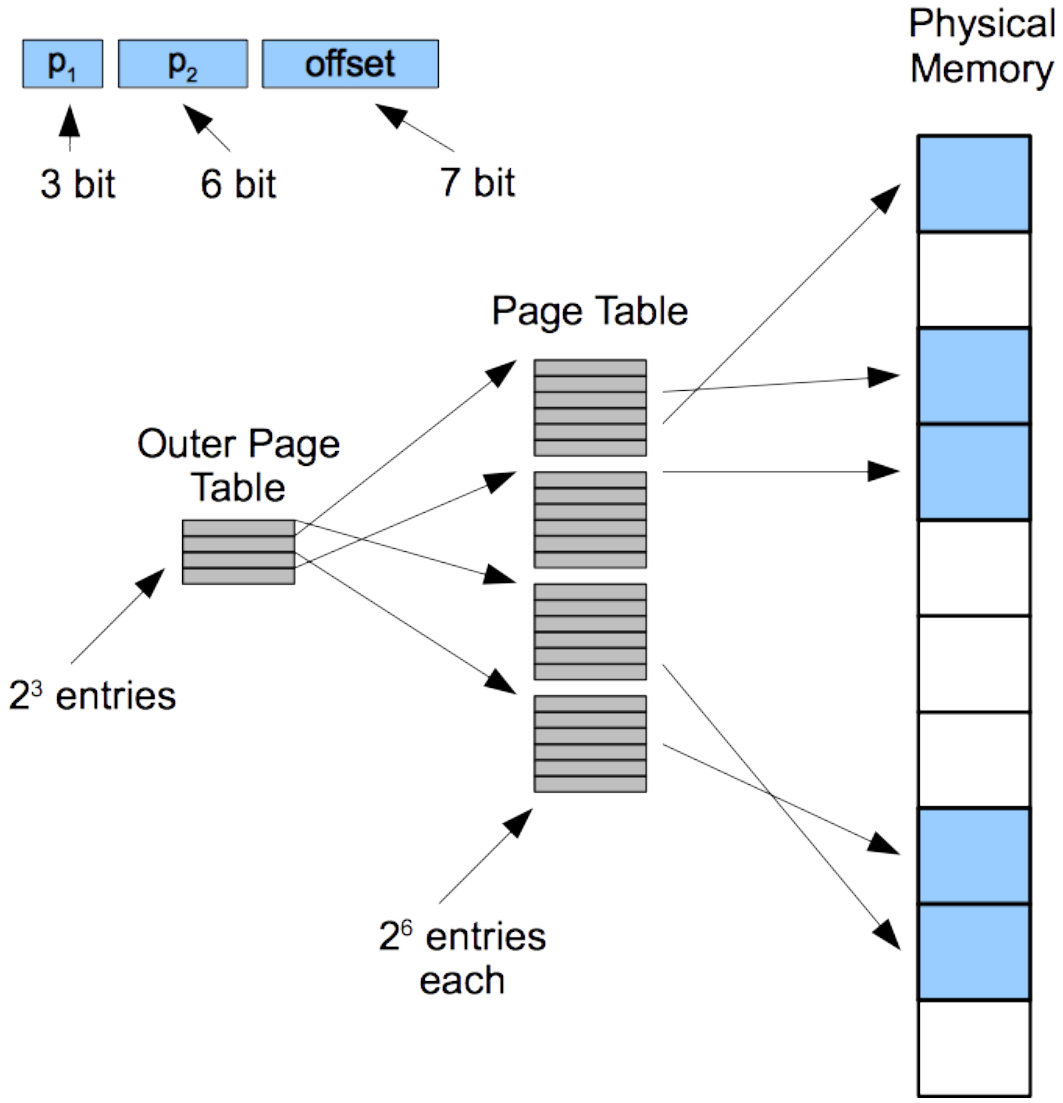
How Do We Translate an Address?



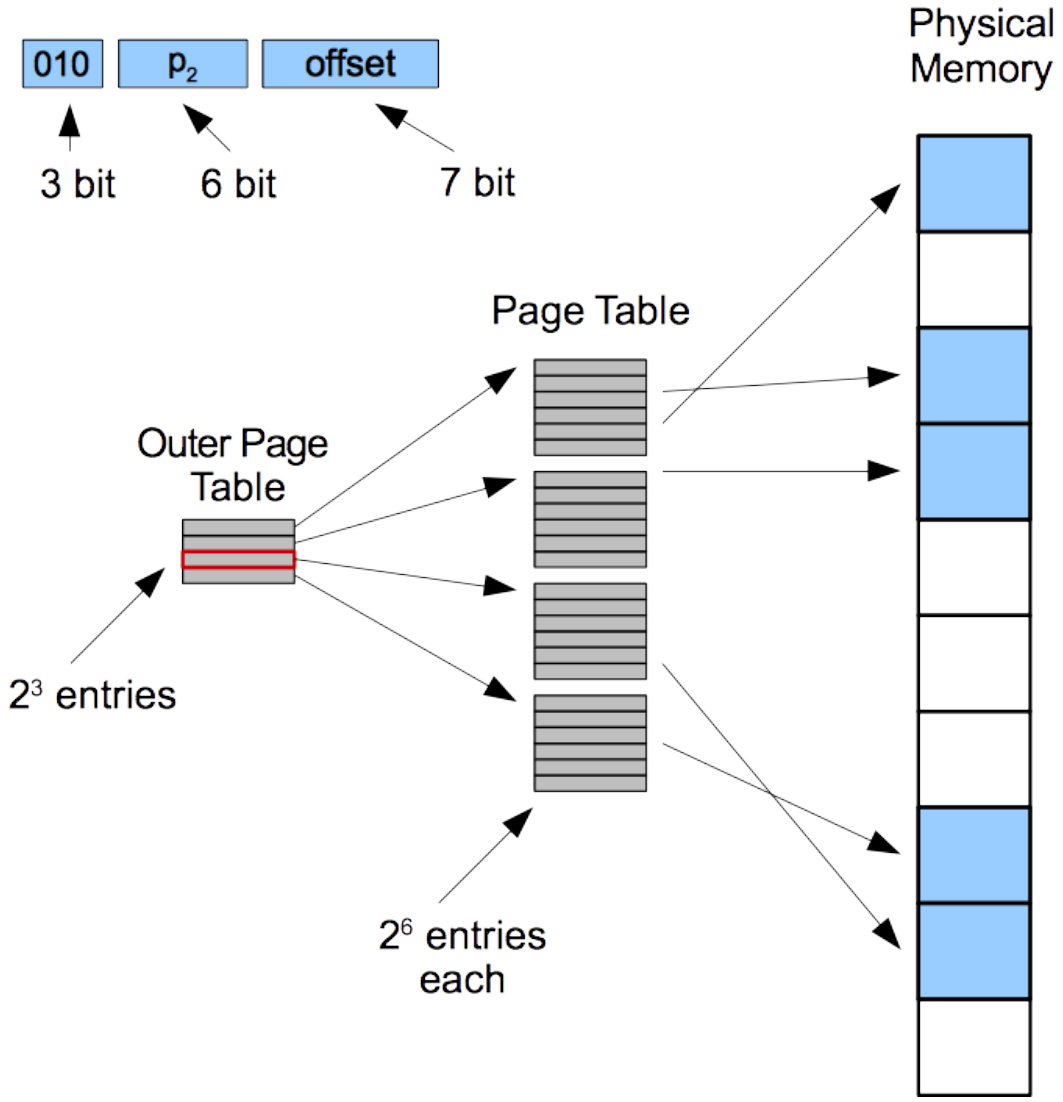
It’s Still Page Number and Offset



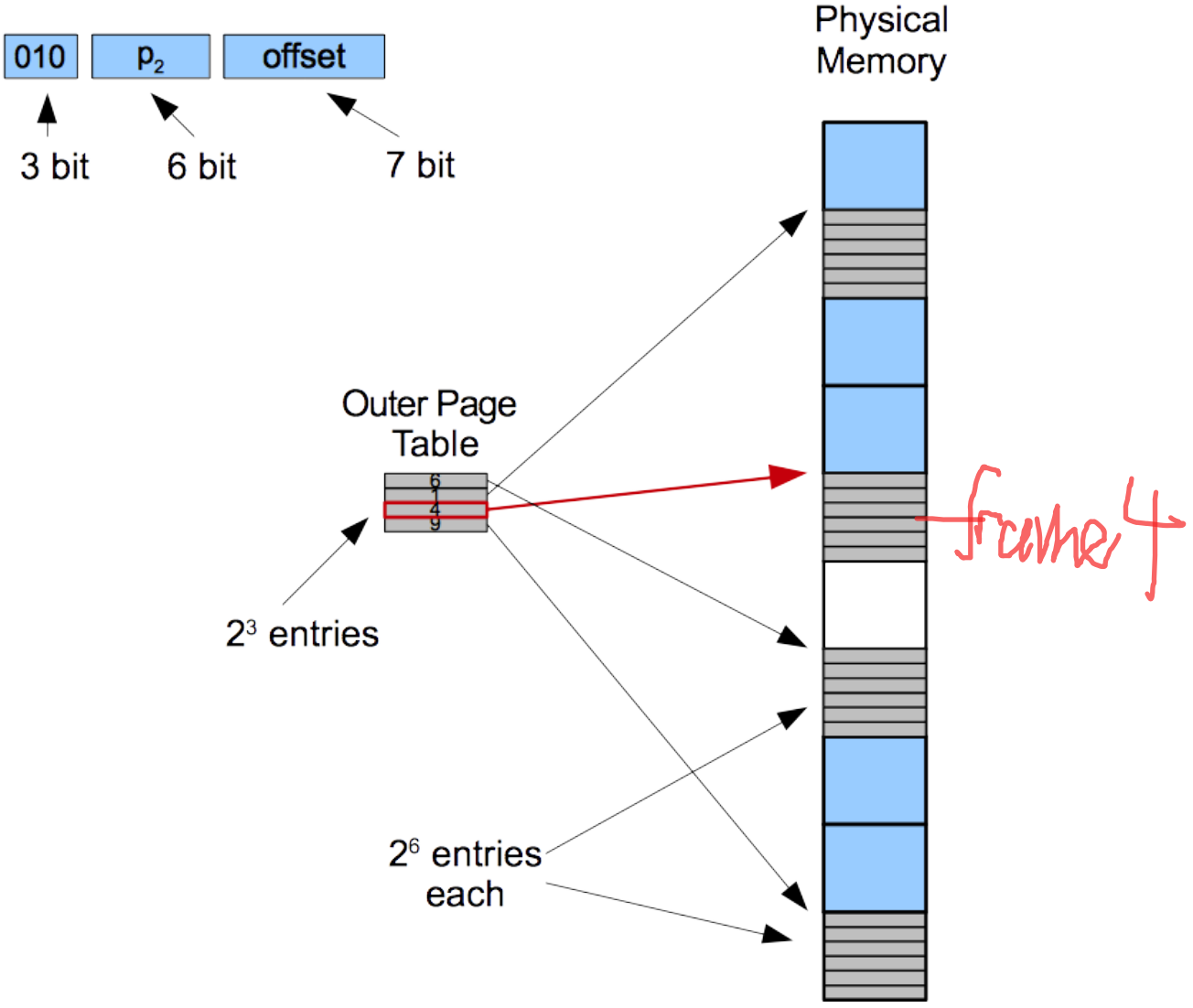
But, we have to find the right page of the page table



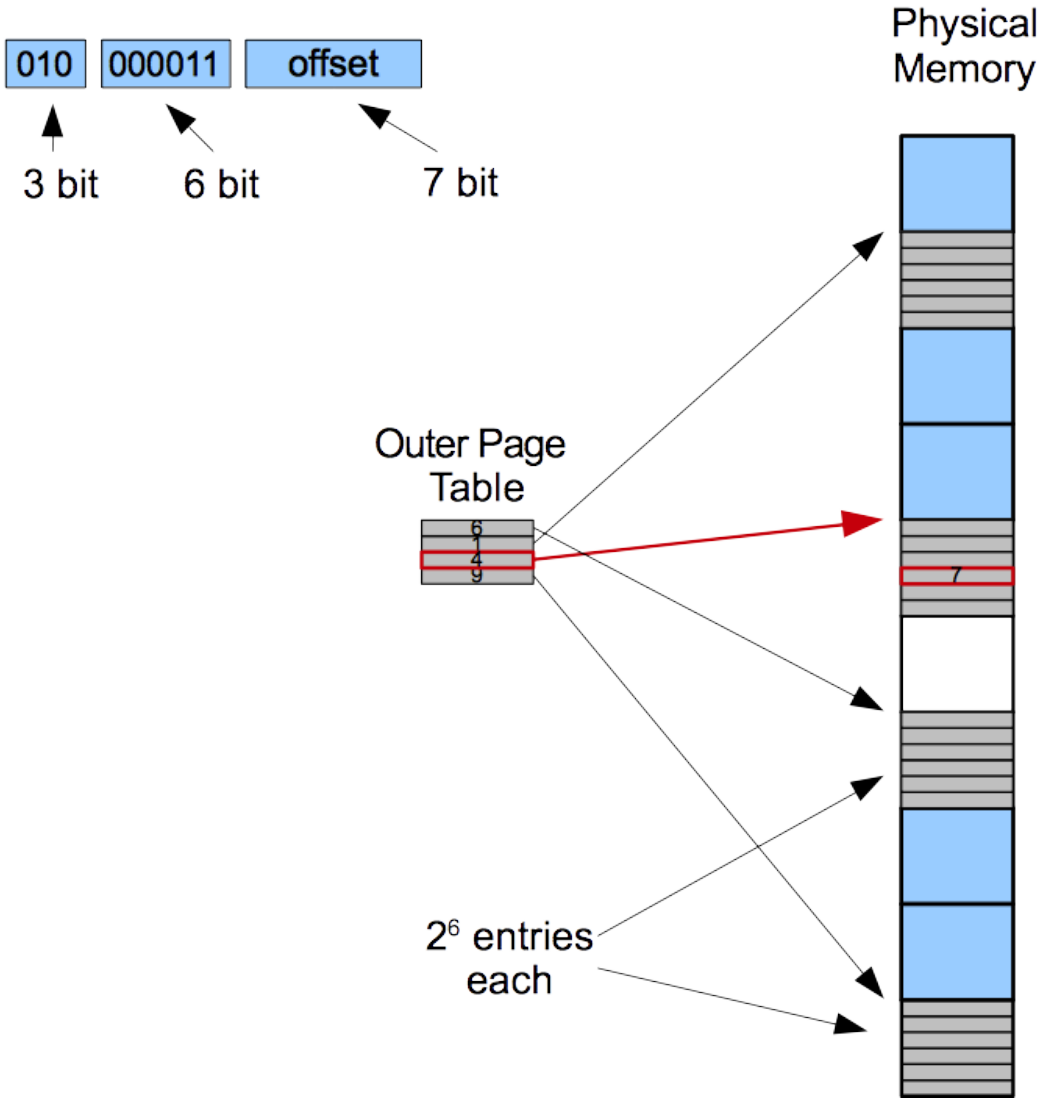
The high-order bits tell us the line of the outer page table we need.



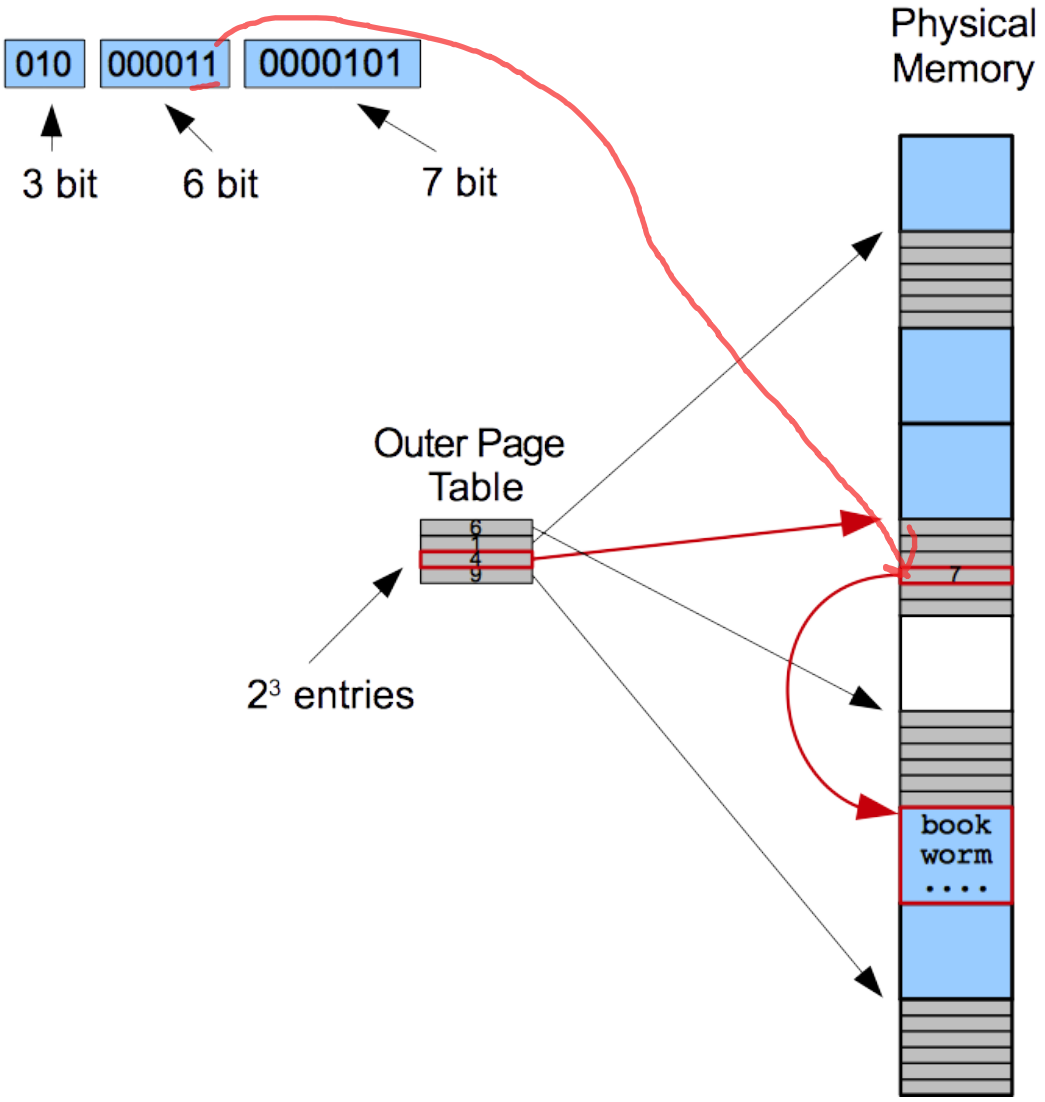
This tells us which frame contains the part of the page table to use.

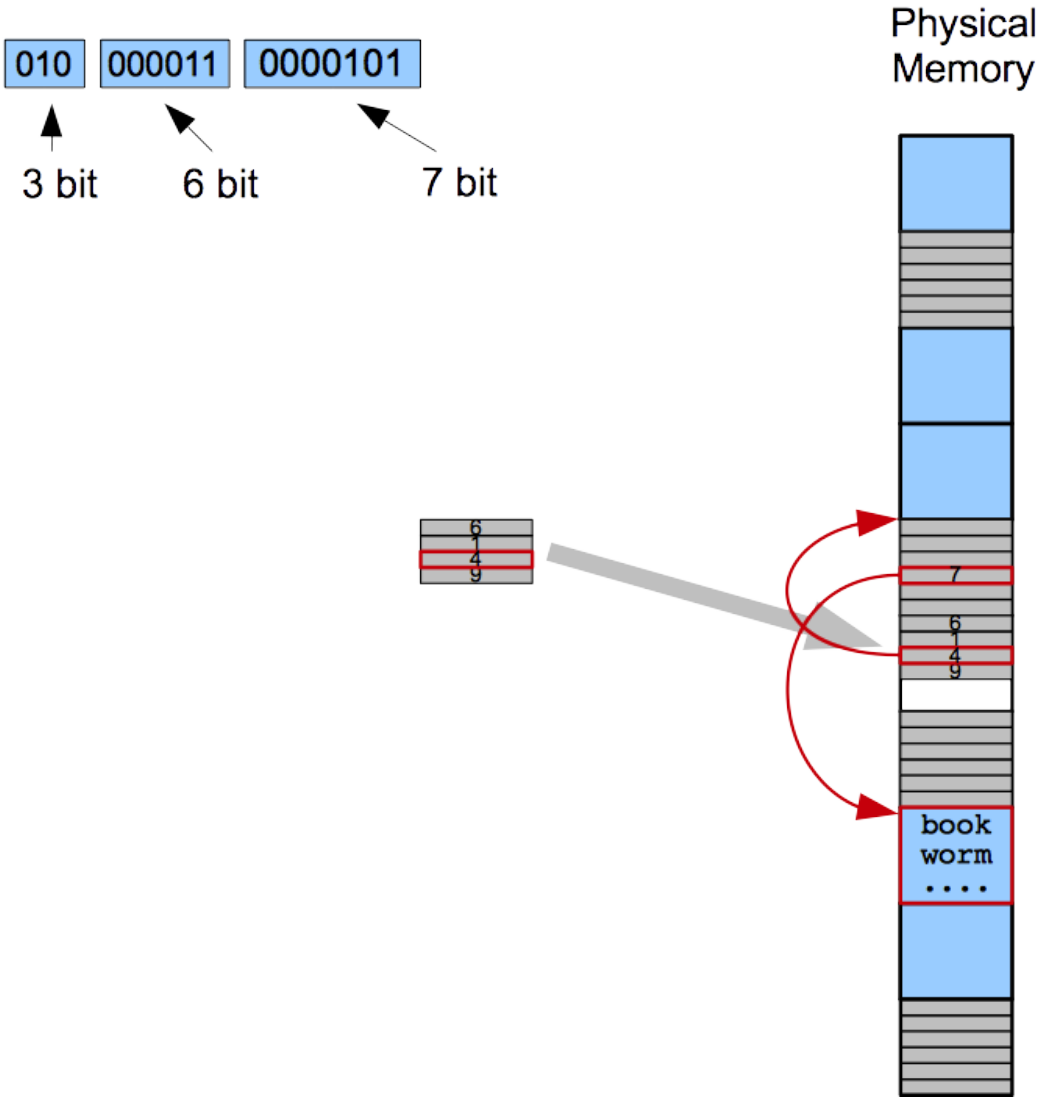


The next group of bits (here 6 bits) tells us which line of entry we need in this part of the page table.



This entry tells us what frame contains the page of the process we want.




010
000011
0000101
3 bit
6 bit
7 bit
Physical Memory
6
1
4
9
7
6
1
4
9
book
worm
....

Where Does It End?

- We could have a 3-level page table, or more
- We'd like to keep adding levels until …?
  - The top-level page table fits in a page
- Consider a typical example:
  - 32-bit logical/physical addresses
  - 4 KB page size

1. How many bits for the offset?

2. How many bits for the page number?

Where Does It End?

- We could have a 3-level page table, or more
- We'd like to keep adding levels until …?
  - The top-level page table fits in a page
- Consider a typical example:
  - 32-bit logical/physical addresses
  - 4 KB page size

1. How many bits for the offset?
   Page size is 2^{12} bytes

2. How many bits for the page number?

Where Does It End?

- We could have a 3-level page table, or more
- We'd like to keep adding levels until …?
  - The top-level page table fits in a page
- Consider a typical example:
  - 32-bit logical/physical addresses
  - 4 KB page size

1. How many bits for the offset?
   Page size is 2^{12} bytes
   so the offset is (the low-order) 12 bits
2. How many bits for the page number?

Where Does It End?

- We could have a 3-level page table, or more
- We'd like to keep adding levels until …?
  - The top-level page table fits in a page
- Consider a typical example:
  - 32-bit logical/physical addresses
  - 4 KB page size

1. How many bits for the offset?
   Page size is 2^{12} bytes
   so the offset is (the low-order) 12 bits
2. How many bits for the page number?
   Logical address is 32 bits

Where Does It End?

- We could have a 3-level page table, or more
- We'd like to keep adding levels until …?
  - The top-level page table fits in a page
- Consider a typical example:
  - 32-bit logical/physical addresses
  - 4 KB page size

1. How many bits for the offset?
   Page size is 2^{12} bytes
   so the offset is (the low-order) 12 bits
2. How many bits for the page number?
   Logical address is 32 bits
   so 20 are left over for the page number. 32 - 12

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?

4. How many bytes for a line of the inner page table?

5. How much total memory for the inner page table?

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,

4. How many bytes for a line of the inner page table?

5. How much total memory for the inner page table?

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,
   so that's 2^{20} entries in the page table.
4. How many bytes for a line of the inner page table?

5. How much total memory for the inner page table?

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,
   so that's 2^{20} entries in the page table.
4. How many bytes for a line of the inner page table?
   We need 20 bits just for the frame number,

5. How much total memory for the inner page table?

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,
   so that's 2^{20} entries in the page table.
4. How many bytes for a line of the inner page table?
   We need 20 bits just for the frame number,
   so let's round up to 4 bytes (2^{32} bits)
5. How much total memory for the inner page table?

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,
   so that's 2^{20} entries in the page table.
4. How many bytes for a line of the inner page table?
   We need 20 bits just for the frame number,
   so let's round up to 4 bytes (2^{32} bits)
5. How much total memory for the inner page table?
   2^{20} entries * 4 bytes per entry

Where Does It End?

- Same example continued
  - 32-bit logical/physical addresses
  - 4 KB page size

3. How many lines in the (inner) page table?
   A process could have 2^{20} pages,
   so that's 2^{20} entries in the page table.
4. How many bytes for a line of the inner page table?
   We need 20 bits just for the frame number,
   so let's round up to 4 bytes (2^{32} bits)
5. How much total memory for the inner page table?
   2^{20} entries * 4 bytes per entry
   That's 2^{22} bytes

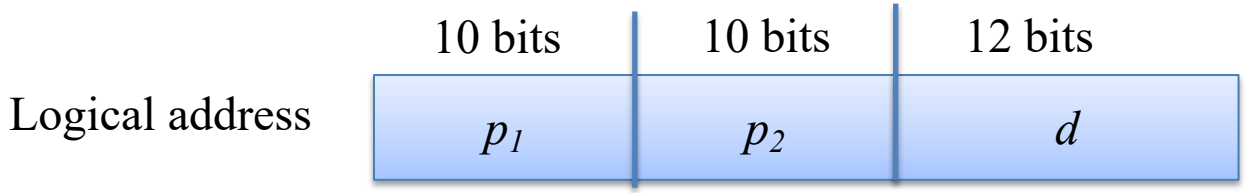
A Typical Hierarchical Paging Example

- Same example continued
  6. How many pages to store the inner page table?

  7. How many lines in the outer page table?

  8. So, how many total bytes for the outer table?

- So, for this example, you can think of a logical address as:

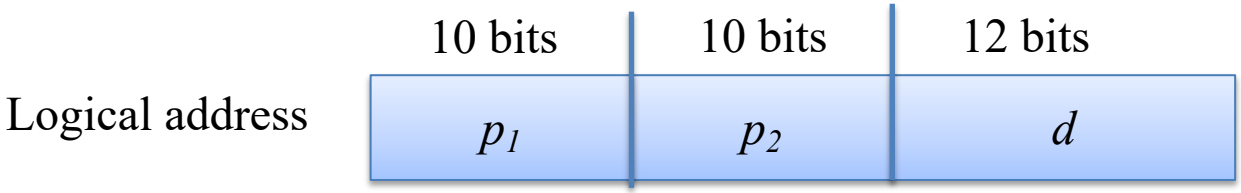


A Typical Hierarchical Paging Example

- Same example continued
  6. How many pages to store the inner page table?
     2^{22} bytes / 2^{12} bytes per page = 2^{10} pages
  7. How many lines in the outer page table?

  8. So, how many total bytes for the outer table?

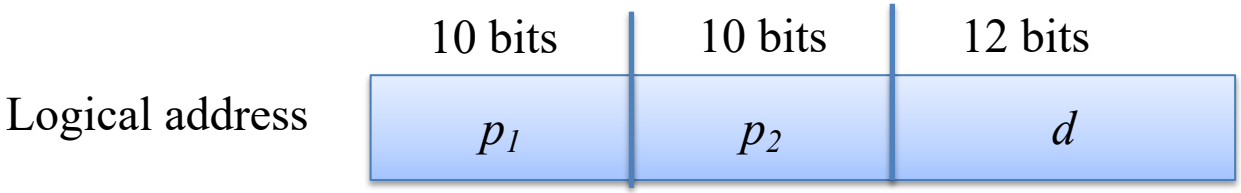
- So, for this example, you can think of a logical address as:



A Typical Hierarchical Paging Example

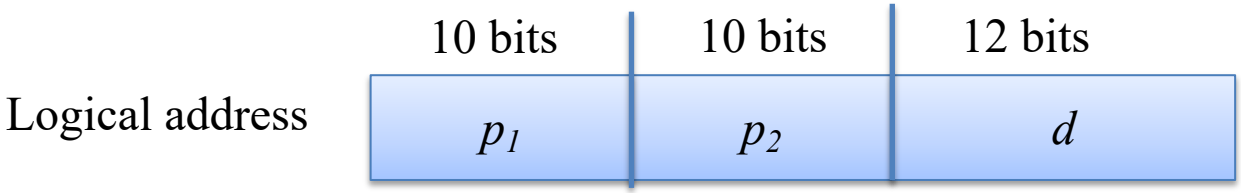
- Same example continued
  6. How many pages to store the inner page table?
     2^{22} bytes / 2^{12} bytes per page = 2^{10} pages
  7. How many lines in the outer page table?
     One for each page of the page table, so 2^{10} lines
  8. So, how many total bytes for the outer table?

- So, for this example, you can think of a logical address as:

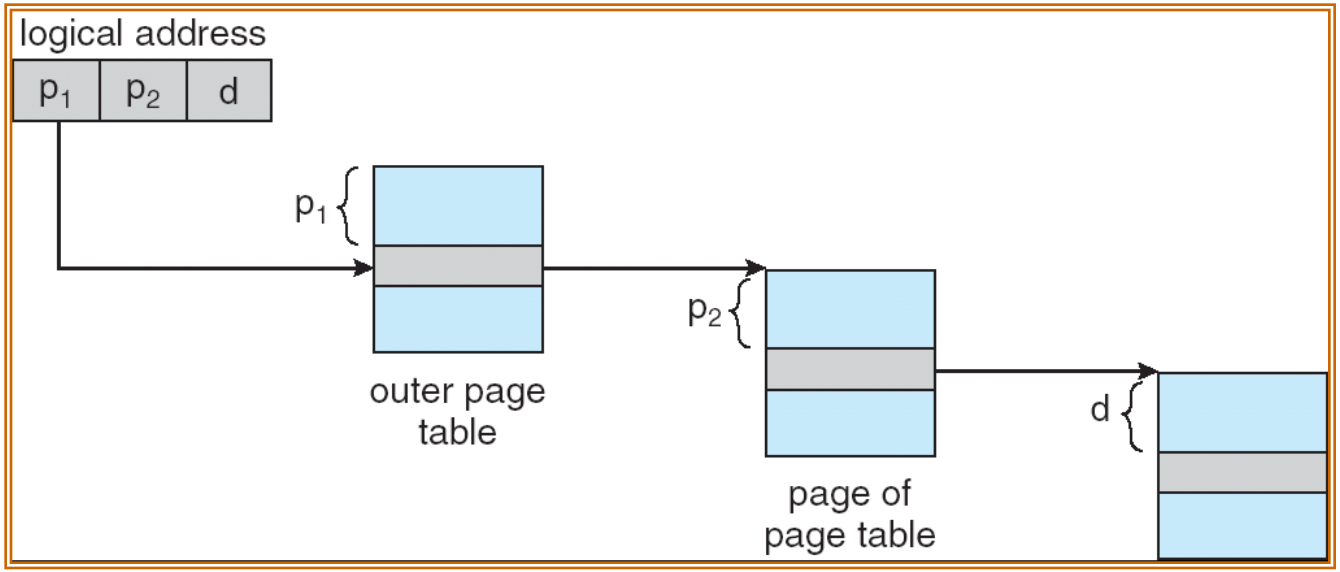


A Typical Hierarchical Paging Example

- Same example continued
  6. How many pages to store the inner page table?
     2^{22} bytes / 2^{12} bytes per page = 2^{10} pages
  7. How many lines in the outer page table?
     One for each page of the page table, so 2^{10} lines
  8. So, how many total bytes for the outer table?
     2^{10} lines * 4 bytes per line = 2^{12} bytes
- So, for this example, you can think of a logical address as:

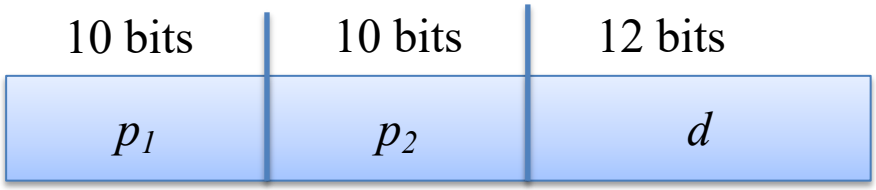


Hierarchical Address Translation



Memory Access Time

- Now, a read/write in a user process could take three memory accesses
  - Read from the outer page table
  - Read from the inner page table
  - Read/Write main memory
- Good thing we have a TLB
  - It will go all the way from the page number to a line of the **innermost** page table
  - Now, a TLB hit can bypass two levels of lookup



TLB				
20-bit page number	Rd	M	Re	Frame
453282	1	1	1	31245
132054	0	0	1	725418

A Multi-Level Page Table

- Consider a system with:
  - 64-bit logical/physical addresses
  - 64 KB page size

1. How many bits for the offset in a logical address?
2. How many bits for the page number?
3. How many lines in the (inner, level-1) page table?
4. Total size of the (inner page table)?
5. How many lines of the page table fit on a page?
6. So, how many lines of a 2^{nd}-level page table?
7. How many lines for a 3^{rd}-level page table?
8. How many levels?

A Multi-Level Page Table

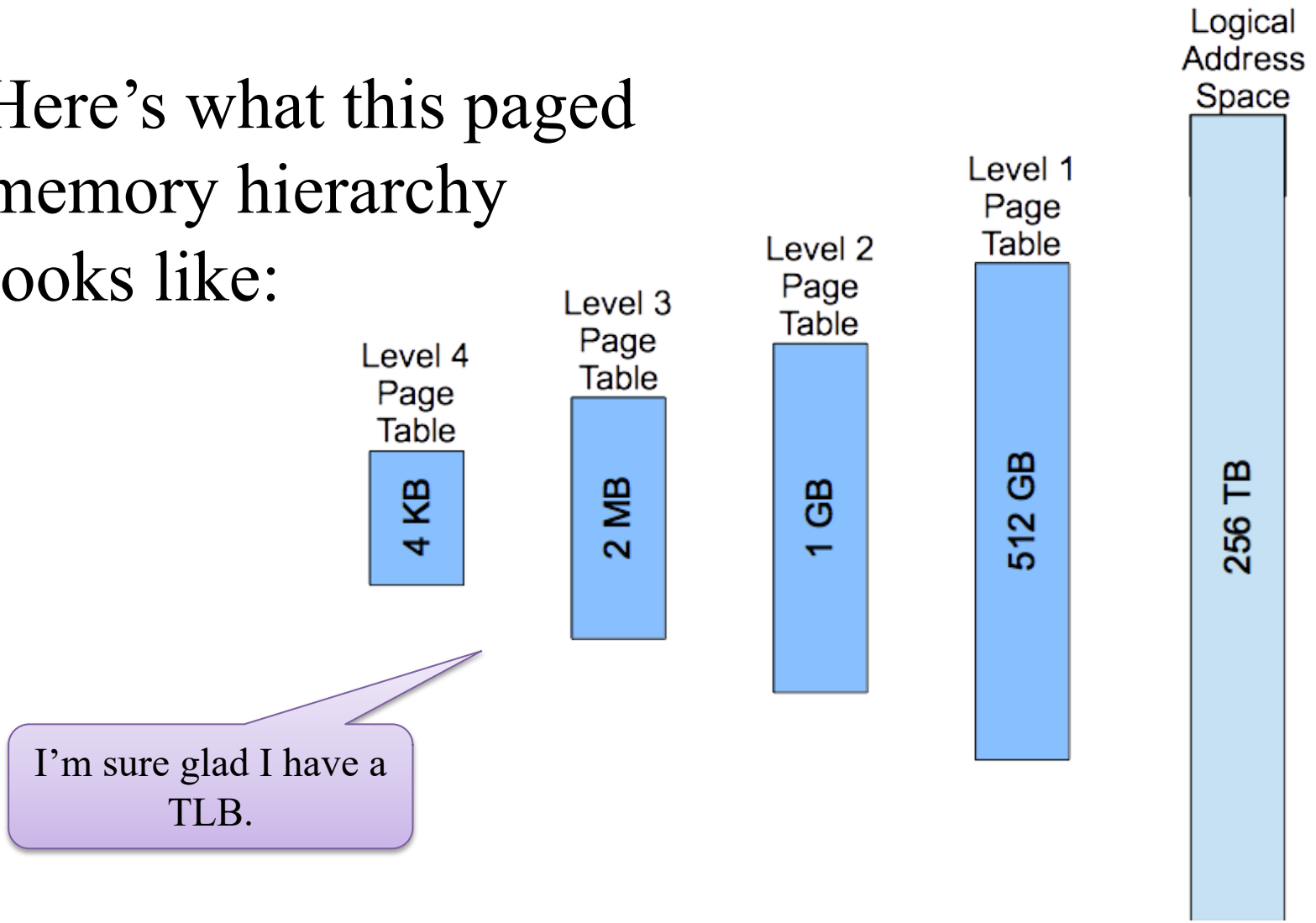
1. How many bits for the offset in a logical address?
   That's the low-order 16 bits.
2. How many bits for the page number?
   That's the high-order 48 bits.
3. How many lines in the (inner, level-1) page table?
   One for each page, so 2^{48}
4. Total size of the (inner page table)?
   If it's 8 bytes per entry, that's $2^{48} * 8 = 2^{51}$
5. How many lines of the page table fit on a page?
   2^{16} page size / 8 entries per page = 2^{13}
6. So, how many lines of a 2^{nd}-level page table?
   $1/2^{13}$ times as many as the (inner) page table, so 2^{35}
7. How many lines for a 3^{rd}-level page table?
   $1/2^{13}$ times as many as the level-2 page table, so 2^{22}
8. How many levels?
   Level-4 page table has 2^9 entries. At 8 bytes each, this fits in a page.

The Real World

- AMD64 paged memory
  - 4 KB page size
    (or 2 MB or 1 GB, your choice)
  - A sneaky trick :
    just use the low-order 48 bits of
    the logical address
  - 8 bytes per page table entry.
  - We can figure out the structure of this paging system … let's do that

AMD64 Illustrated

- Here's what this paged memory hierarchy looks like:

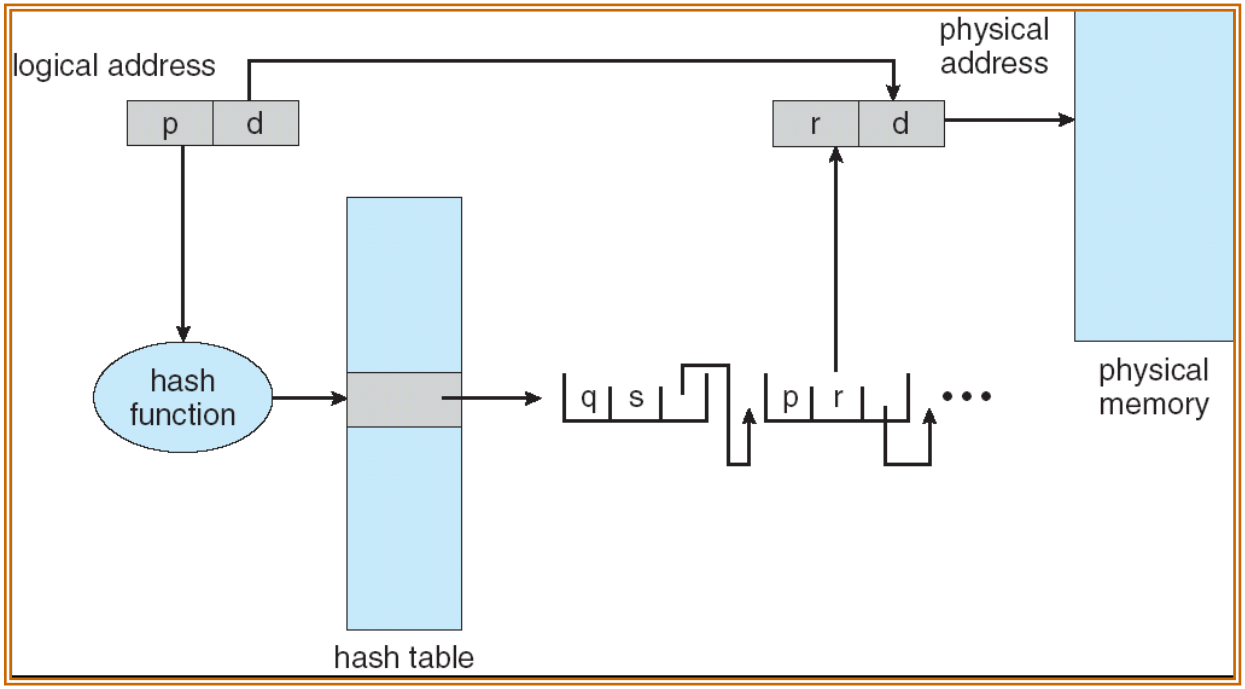


Hashed Page Tables

- More common for large address spaces, greater than 2^{32} words
- Store a sparse mapping from page number to frame number as a hash table
  - Hash the page number to some small integer *i*
  - Store a record with the frame number at line *i* of the hash table
  - You may have collisions, each hash table line has a chain of records for each matching page

Hashed Page Tables

- On lookup:
  - Hash the page number, then look for a matching record



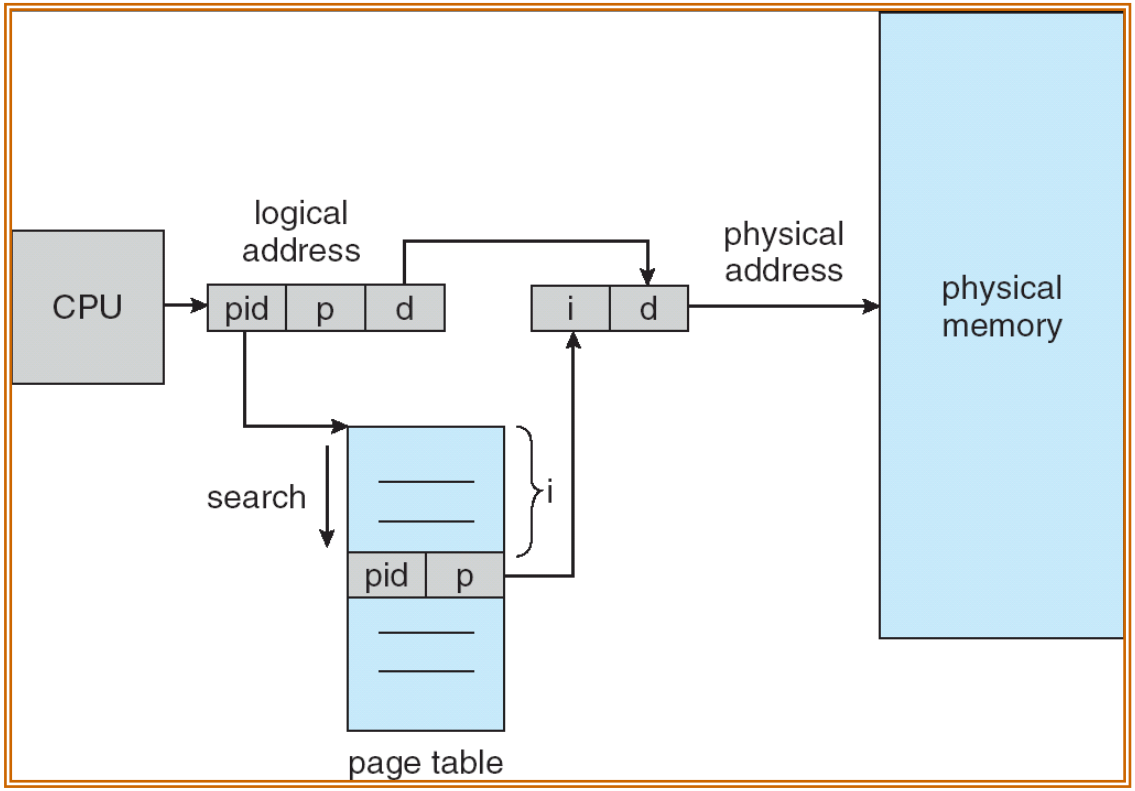
- The point, if the address space is sparse, you don't need a big table.

Inverted Page Table

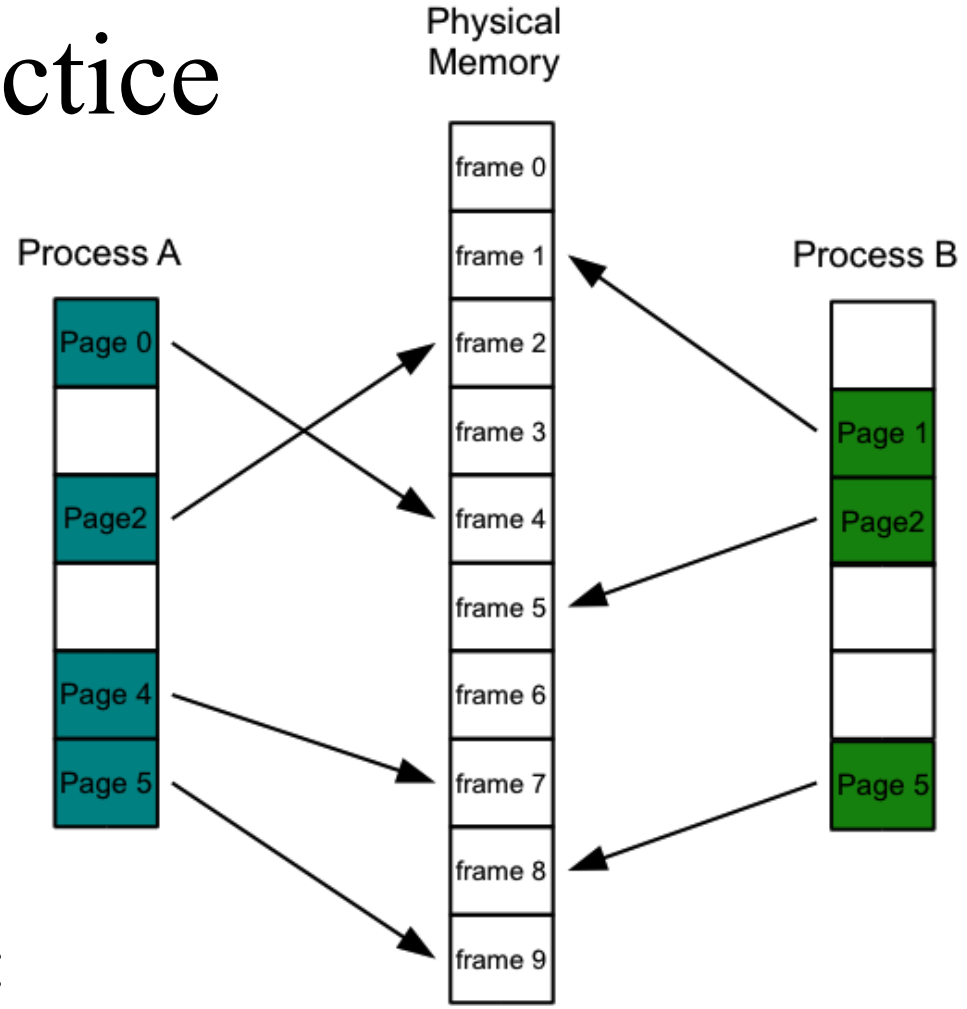
- Consider this
  - Every process has a page table, but …
  - … there are only so many frames of memory
- Let's store address translation by memory frame instead of by process page
  - Just keep one entry for each frame, keeping up with what process and which of its pages occupies it
  - Compared to a regular page table, this is backward
  - But, we can still use it

Inverted Page Table

- Decreased storage overhead
- More complicated page lookup
- Shared Pages?

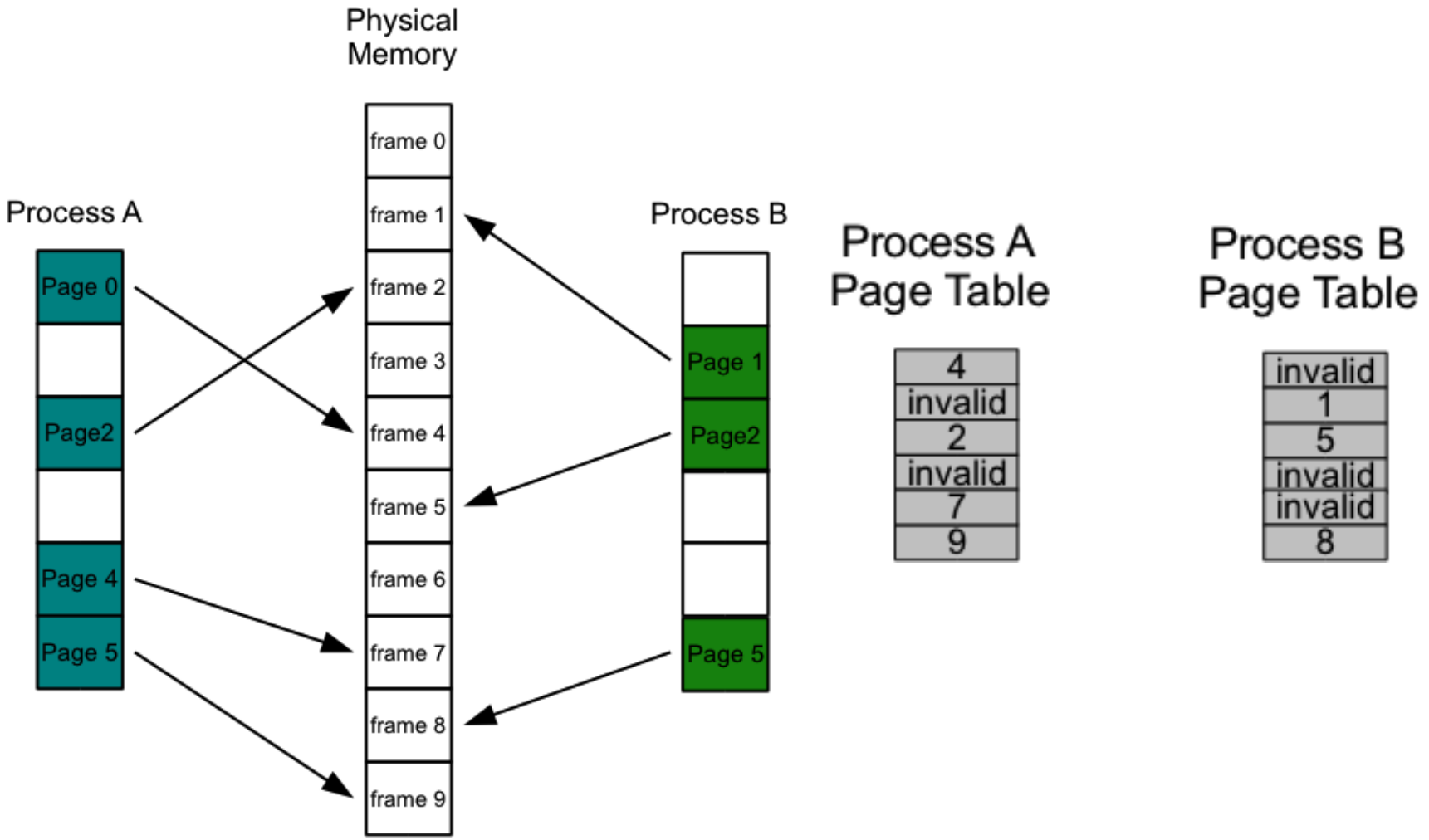


Page Table Practice

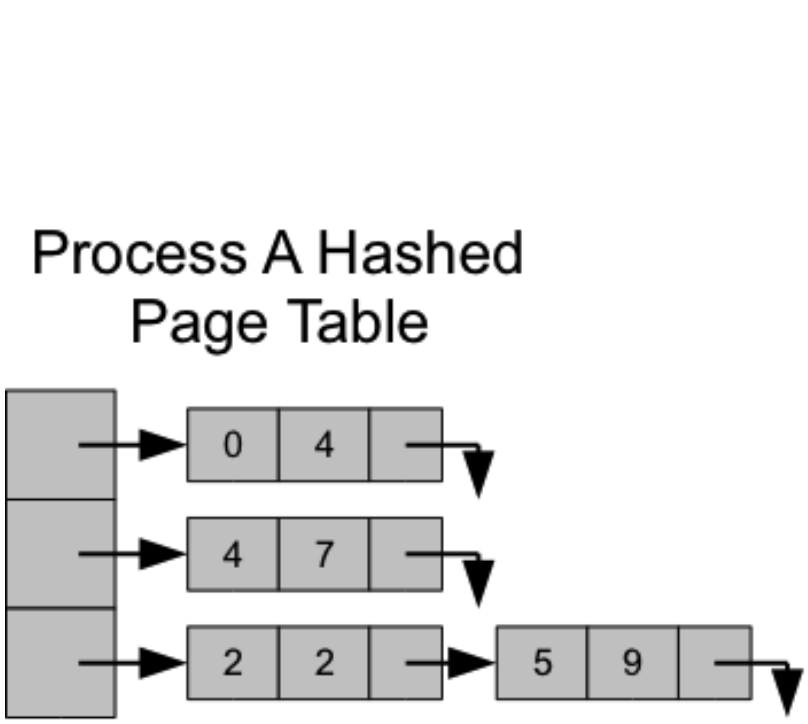


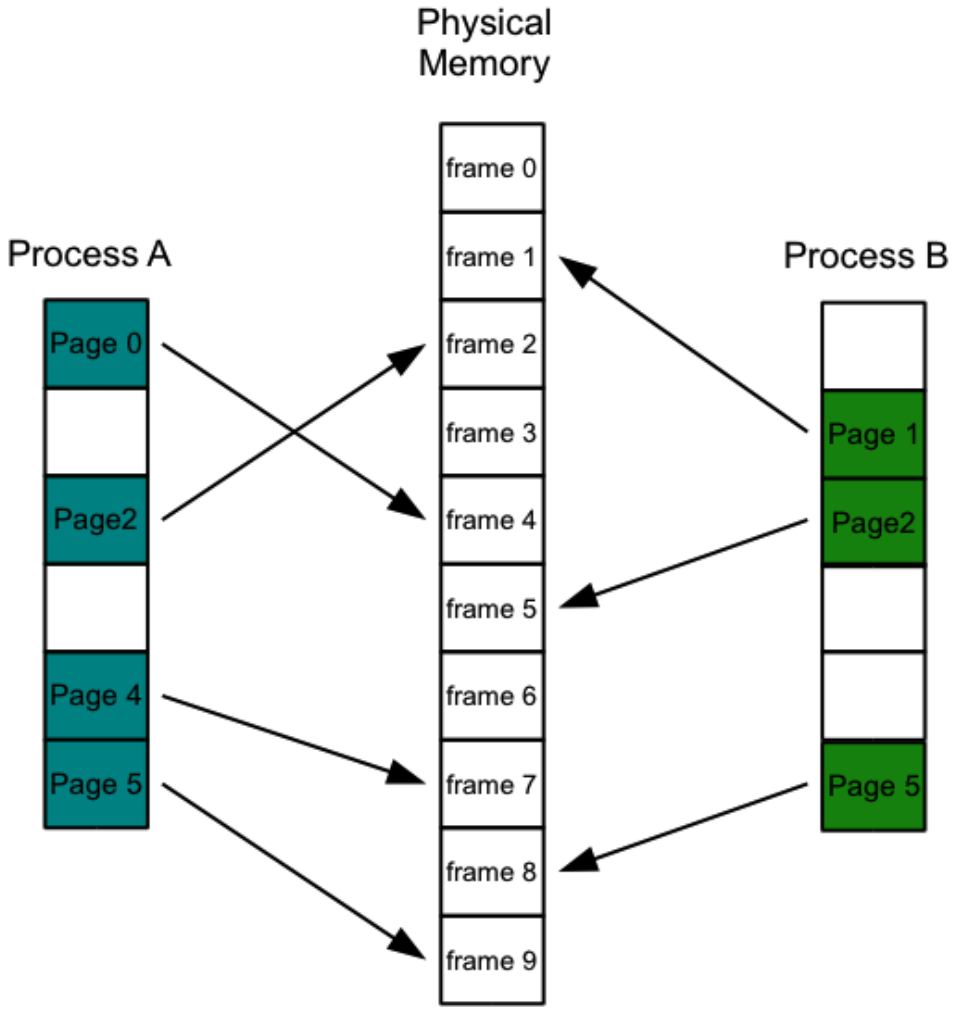
- For this figure, draw:
  - A single-level page table for each process
  - A hashed page table for each, with hash function h(p) = p % 3
  - An inverted page table

Ordinary, Single-Level Page Table for each Process

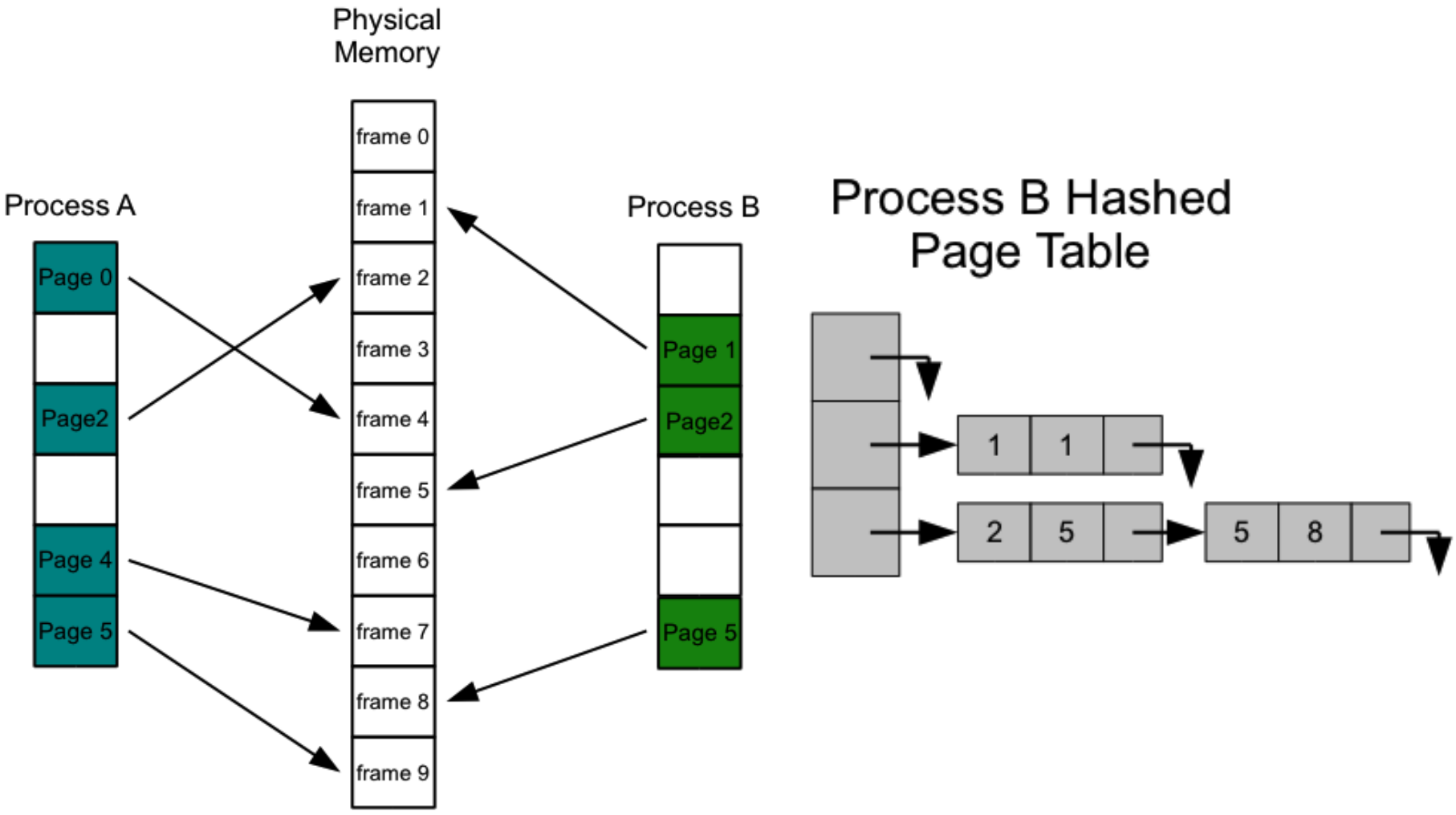


Hashed Page Table for each Process

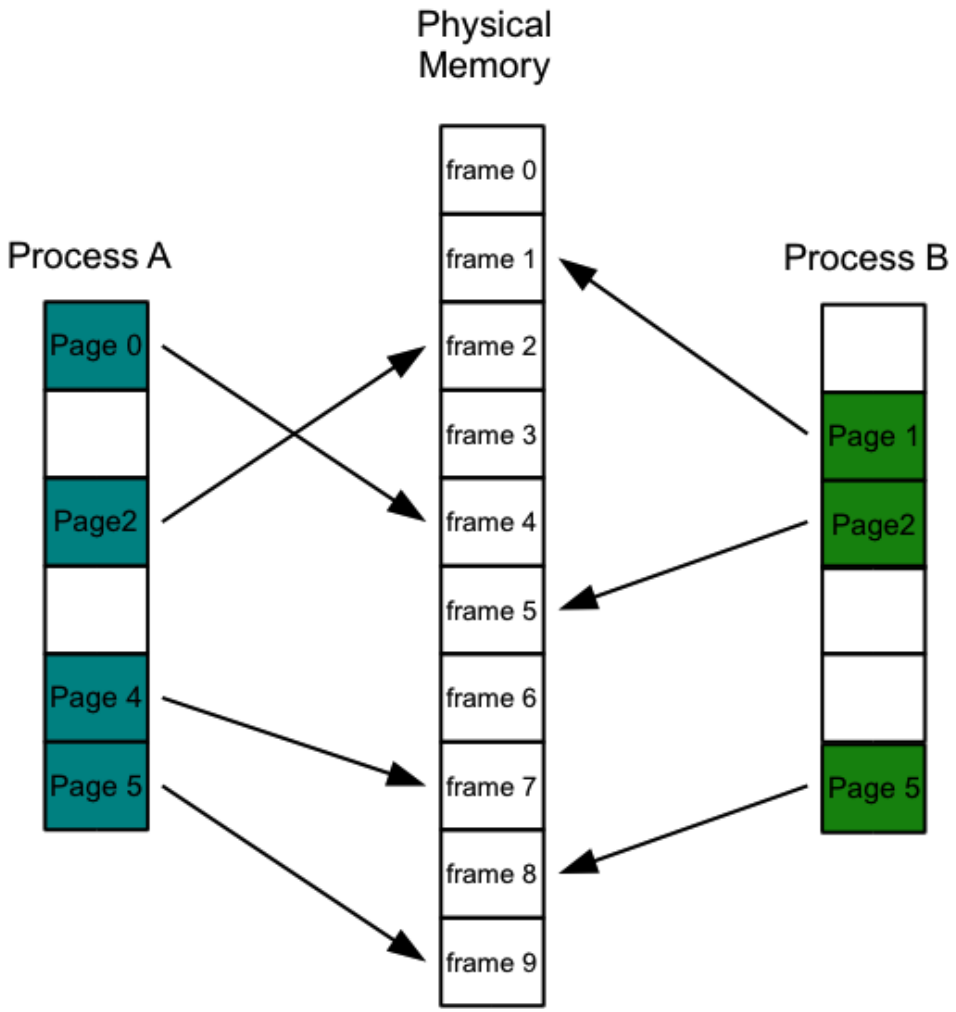




Hashed Page Table for each Process



Inverted Page Table



Inverted Page Table

-	-
B	1
A	2
-	-
A	0
B	2
-	-
A	4
B	5
A	5

What We've Learned

- Load images and memory structure
  - Building load images, getting them into memory, dynamic loading, address binding, etc.
- Dynamic memory allocation and fragmentation
- Paged memory
  - A flexible mechanism for execution-time address binding
  - Structure and use of the page table
  - Paged memory tricks (shared memory, etc)
  - Complex (but necessary) techniques for representing the page table.