Virtual Memory (I)

CS439: Principles of Computer Systems

March 4, 2015

Last Time

- Physical addresses in memory
- Virtual/logical addresses in process space
- Relocation
- Algorithms for fitting processes into memory
  - First Fit
  - Best Fit
  - Worst Fit
  - All had problems with external fragmentation
- Swapping and compaction
  - Reducing external fragmentation
  - Allowing more processes to run than would fit in memory

Today's Agenda: Virtual Memory

- Overlays (briefly)
- Paging
  - Page Tables
  - HUGE Page Tables
- Initializing Memory
  - Demand Paging

(More) History

- Problem: Processes too large to fit into memory
- First Solution: Overlays
  - Program divided into pieces, or *overlays*
  - Overlay manager swaps overlays in and out
  - Programmer had to manually divide the program into pieces
  - Ugh! Automate this, and you get…

Virtual Memory

- Process's view of memory
- Can be much larger than the size of physical memory
- Only portions of the virtual address space are in physical memory at any one time
- Automatically divides process address space into *pages*—a technique known as *paging*

Paging

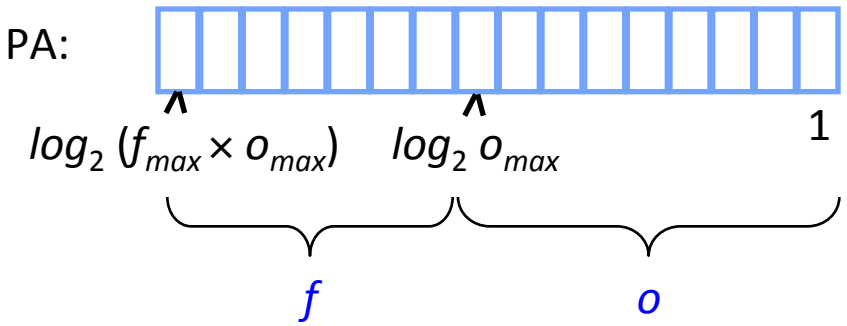
1. Divides a process's virtual address space into fixed-sized *pages*
2. Stores a copy of that address space on disk
3. Views the physical memory as a series of equal-sized *page frames* (or *frames)*
4. Moves the pages into frames in memory
   - *when* they are moved is a policy decision
5. Manages the pages in memory
   - Is a page still in use? is it written? …

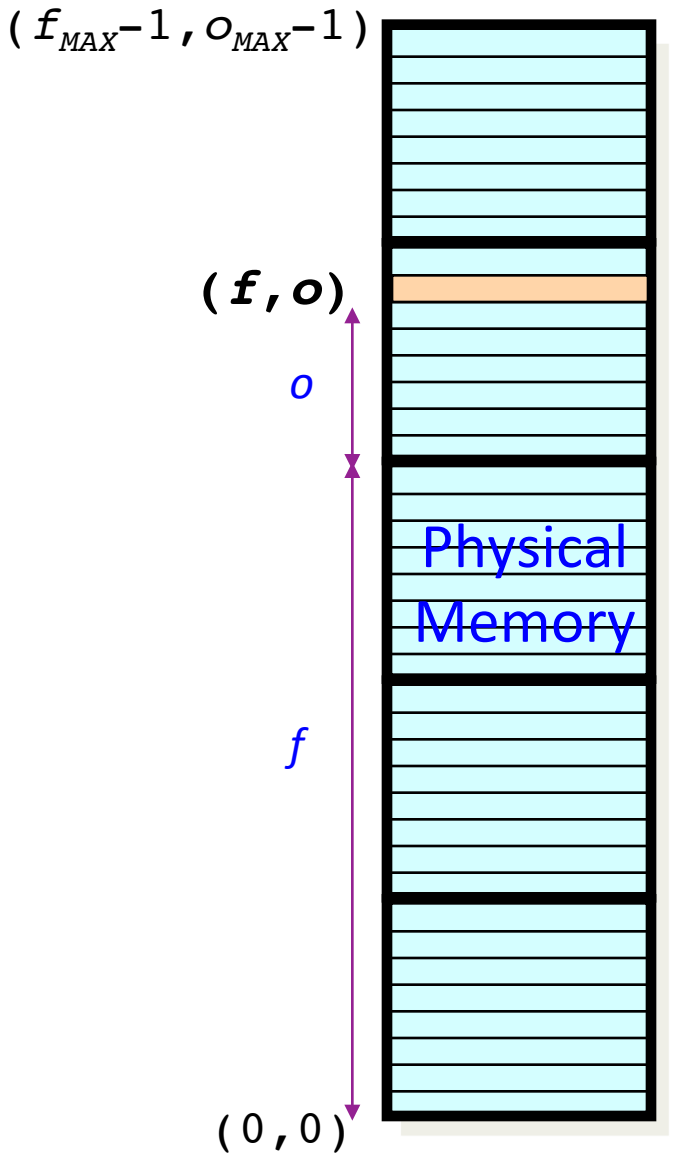
Page Frames

Physical memory partitioned into equal sized *page frames*

A memory address is a pair (*f*, *o*)

- *f* — frame number (f_{max} frames)
- *o* — frame offset (o_{max} bytes/frames)
- Physical address = $o_{max} \times f + o$

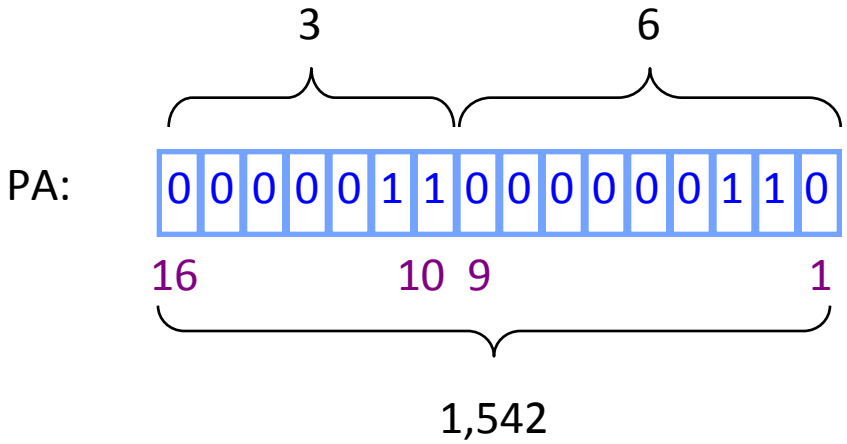


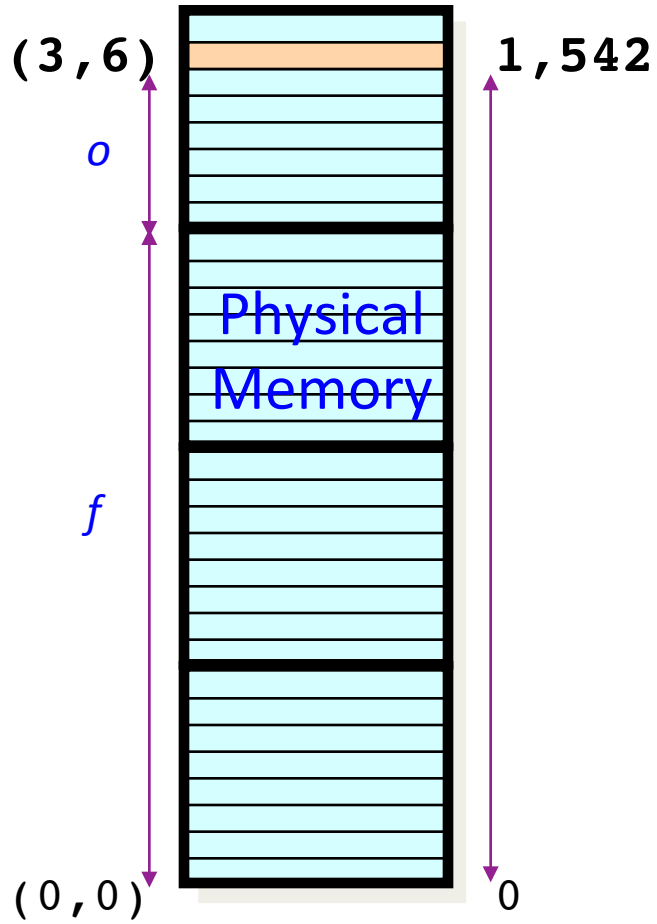


Physical Addresses (Frame/Offset Pairs)

Example: A 16-bit address space with (o_{max} =) 512-byte page frames

Addressing location (3, 6) = 1,542





Paging Implementation

- Physical memory is *viewed* as *page frames*
- In physical addresses:
  - high-level bits are *interpreted* as frame number
  - low-level bits are *interpreted* as offset

Virtual Memory: Pages

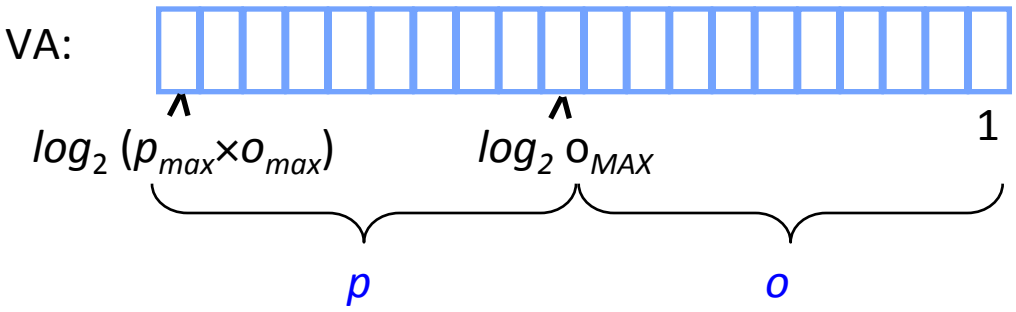
A process's virtual address space is partitioned into equal sized *pages*.

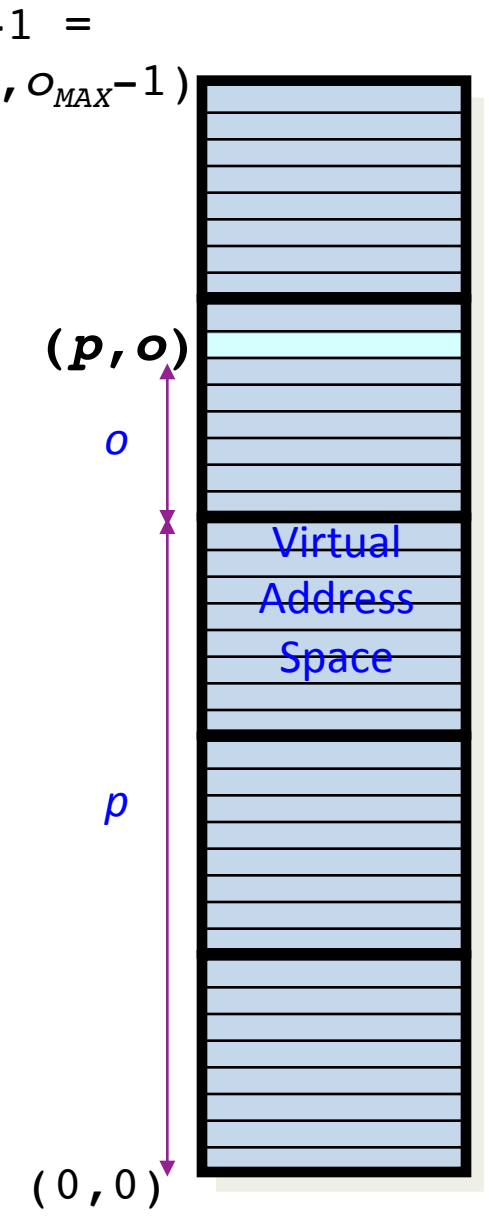
$|page| = |page\ frame|$

The system's page size is equivalent to its frame size.

A virtual address is a pair (*p*, *o*)

- *p* — page number (p_{max} pages)
- *o* — page offset (o_{max} bytes/pages)
- Virtual address = $o_{max} \times p + o$





Paging Implementation

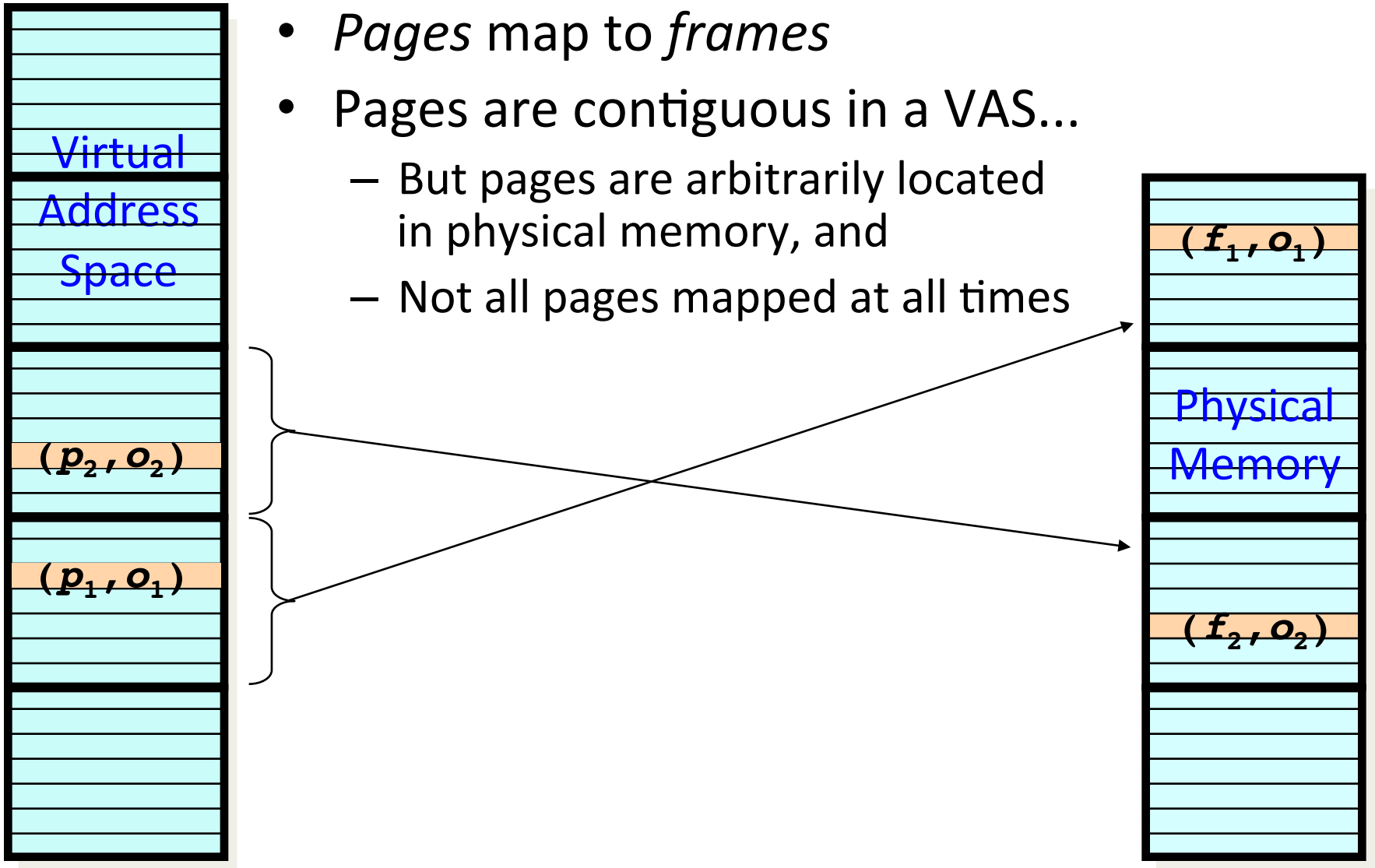
- Virtual address space (aka virtual memory) *viewed* as *pages*
- In virtual addresses:
  - high-level bits are *interpreted* as page number
  - low-level bits are *interpreted* as the offset

iClicker Question

Paging has what advantage over relocation?

A. Easier to manage transfer of pages from disk
B. Requires less hardware support
C. No external fragmentation

From Virtual to Physical

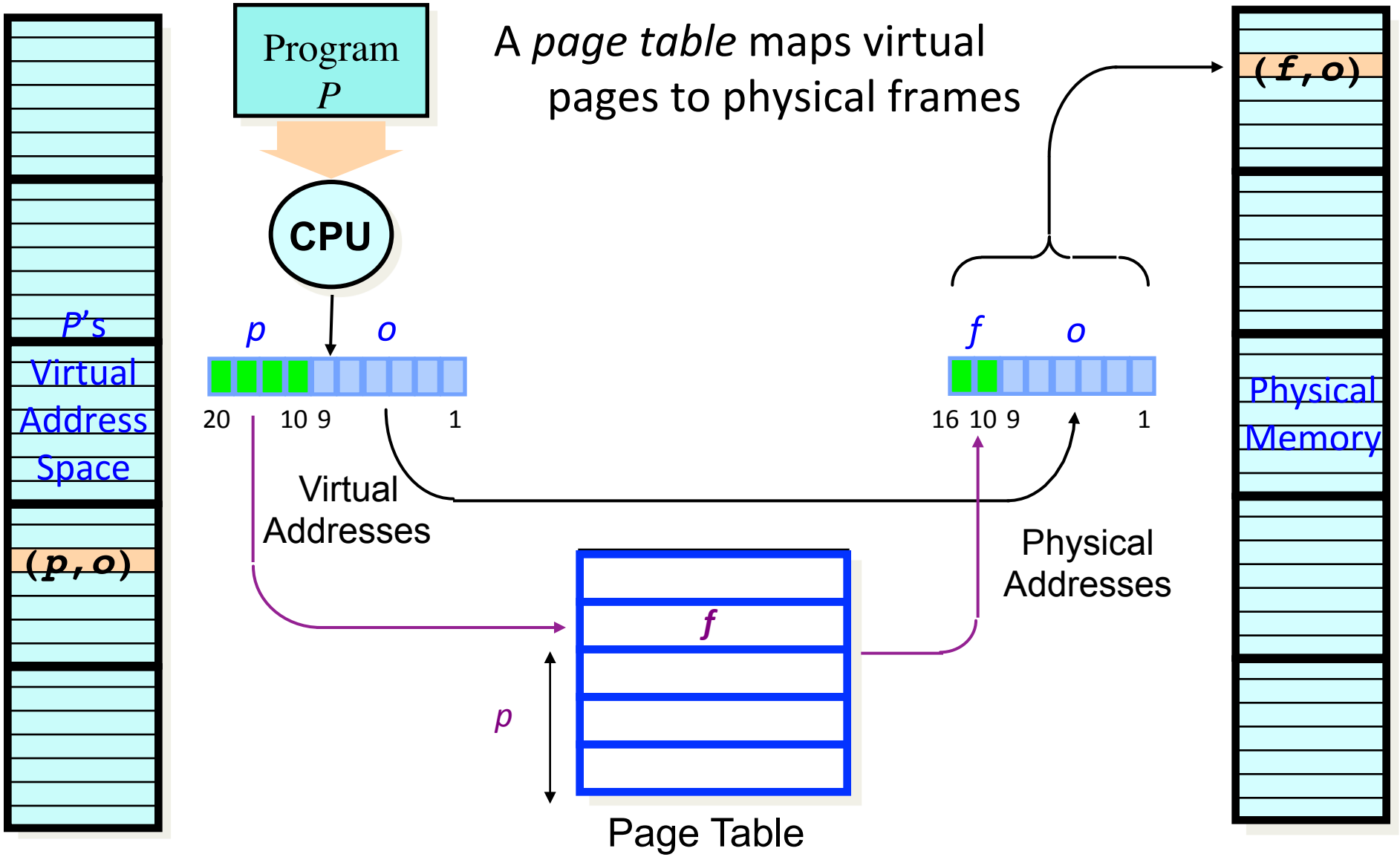


- *Pages* map to *frames*
- Pages are contiguous in a VAS...
  - But pages are arbitrarily located in physical memory, and
  - Not all pages mapped at all times

Ummmm…

- Problem: How do we find addresses when pages are not allocated contiguously in memory?
- Solution: A *page table* keeps track of the mapping of pages to page frames
  - You can think of the page table as a set of relocation registers, one for each frame
  - Mapping is invisible to the process
  - Protection is provided with the same mechanisms as used in dynamic relocation

Virtual Address Translation



Virtual Address Translation: Text Description

Steps to Virtual-Physical Memory Translation:

- The program gives a virtual address to the CPU to translate
- The MMU splits the virtual address into its page and offset numbers
- Since the offset is the same in Virtual and Physical memory it is sent along with no change
- The page number is translated into a frame number
  - Look into page table to find out if page exists in physical memory
    - The page number is the index of the correct entry in the page table (much like indexing an array)
  - If the page exists in memory the frame number is sent along
  - If it doesn't exist in memory but exists on disk, the corresponding page is moved from disk into physical memory and its frame number is recorded
- Append offset to end of frame number to get the full physical address

So… How are we doing?

Wanted:

- reduce or eliminate external fragmentation
- the ability to grow processes easily
- to allow processes to use more memory than that which is physically available
- easy to allocate and de-allocate memory to/from processes
- ability to easily share memory between processes
- protection

Advantages of Paging: Sharing

- Paging allows shared memory
  - the memory used by a process no longer needs to be contiguous
- A shared page may exist in different parts of the virtual address space of each process, but the virtual addresses map to the same physical address

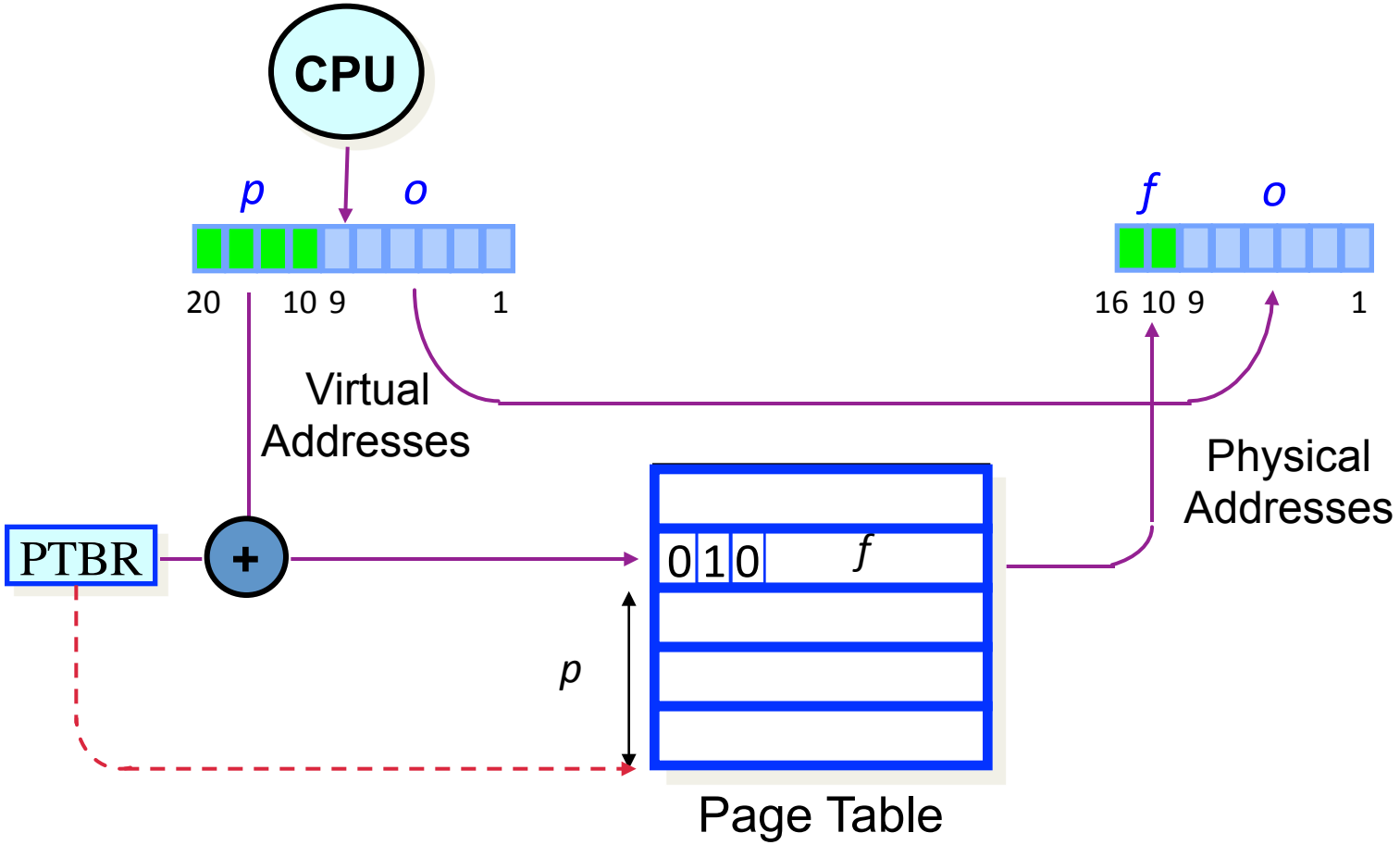
Virtual Address Translation: Protection and Consistency

1 table per process

Part of process's state

Page Table Entry Contents:

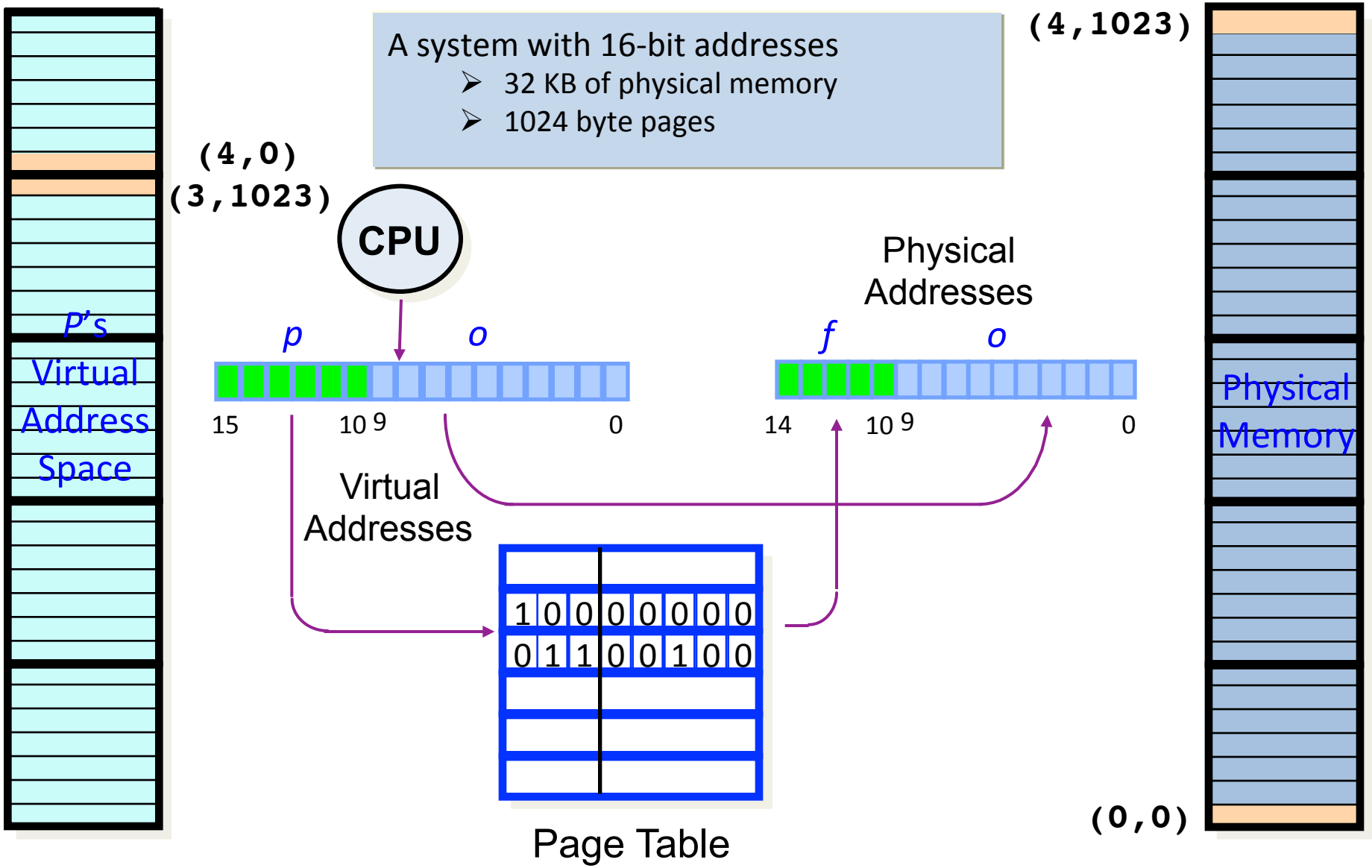
- Flags: dirty bit, resident bit, clock/ reference bit
- Frame number



Virtual Address Translation: Protection and Consistency Text Description

- There is only 1 page table per process. It is part of the process' state. This keeps different processes from corrupting each other's page tables and being able to access each other's memory.
- Page Table Entry Contents:
  - Flags: dirty big, resident bit, and clock/reference bit. Flag stored at beginning of entry, aka the first few bits
  - Frame number aka where that page lives in physical memory. Stored in the remaining bits of the entry.
- There is a page table base register (PTBR) that holds the base address of each page table
  - A page number is added to the PTBR to get the address of the corresponding entry in the page table for that page (again, like indexing into an array)

Virtual Address Translation: Current Picture



Virtual Address Translation: Current Picture Text Description

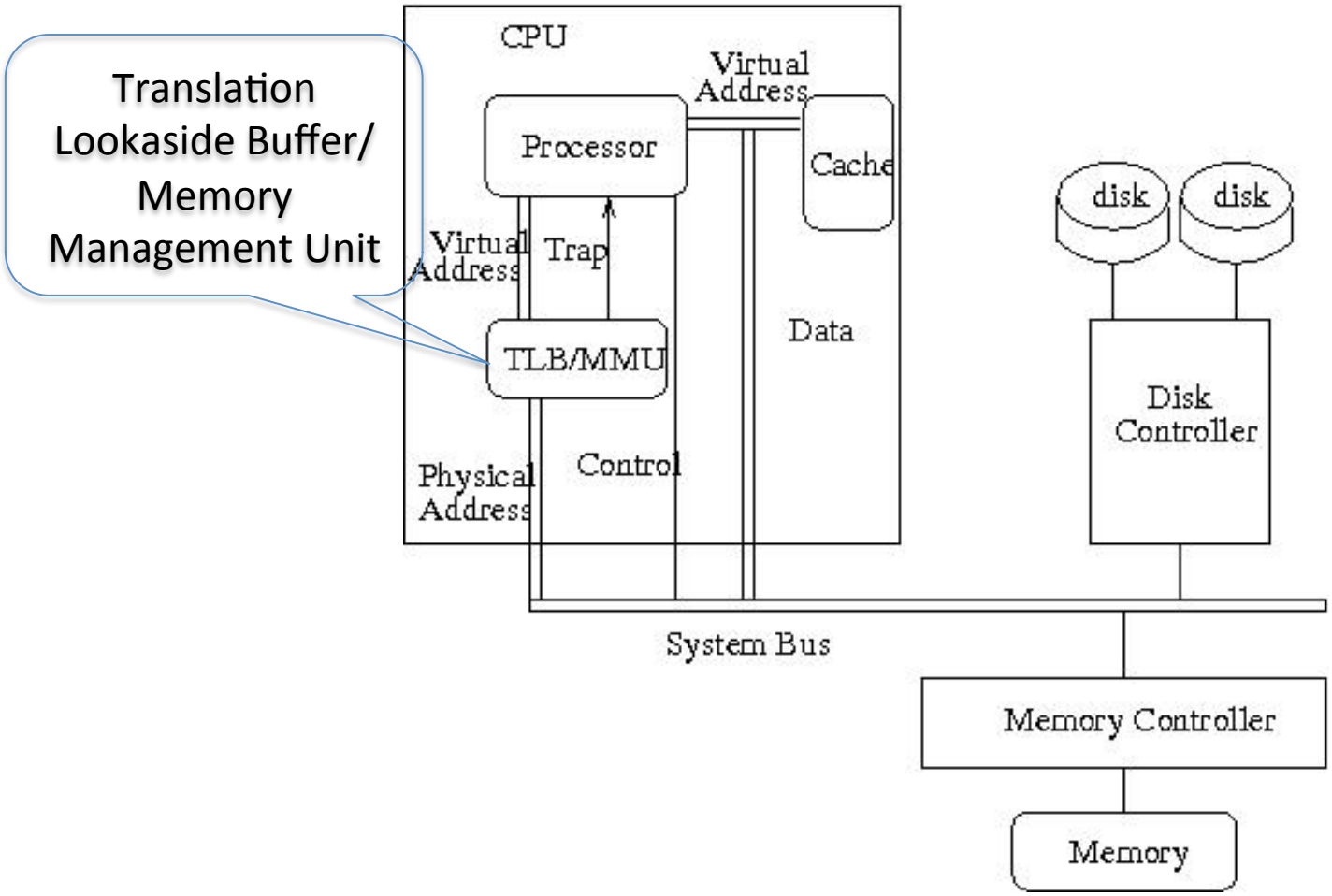
- We have a system with 16-bit addresses
  - has 32KB of physical memory
  - 1024 byte pages
- The number of bits in a physical address is 15. We can get this from the fact that there are about 32k bytes in the system. Log base 2 of 32k is 15 so we need 15 bits to represent all those bytes.
- The number of bits a virtual address is 16. We know this because the system has 16-bit addresses.
- The number of bits in the offset is 10. This is the same in both virtual and physical memory. We can calculate this by looking at the number of bytes per page. We have 1024 byte pages and log base 2 of 1024 is 10, so we need 10 bits to represent all the bytes in a page. (If we had the number of bytes in a frame, then that would also tell us the number of bytes for an offset.)
  - There are 5 bits for each frame number, 15 - 10 = 5
  - There are 6 bits for each page number, 16 - 10 = 6
- Review of Virtual-Physical Translation
  - offset is the same in both the page and the frame so it is transferred directly over
  - use page table to get the frame number from the page number
  - append offset to end of frame number to get full physical address

Performance Issues: Speed

Problem: Virtual memory references require two memory references!

- One access to get the page table entry
- One access to get the data

Back to Architecture



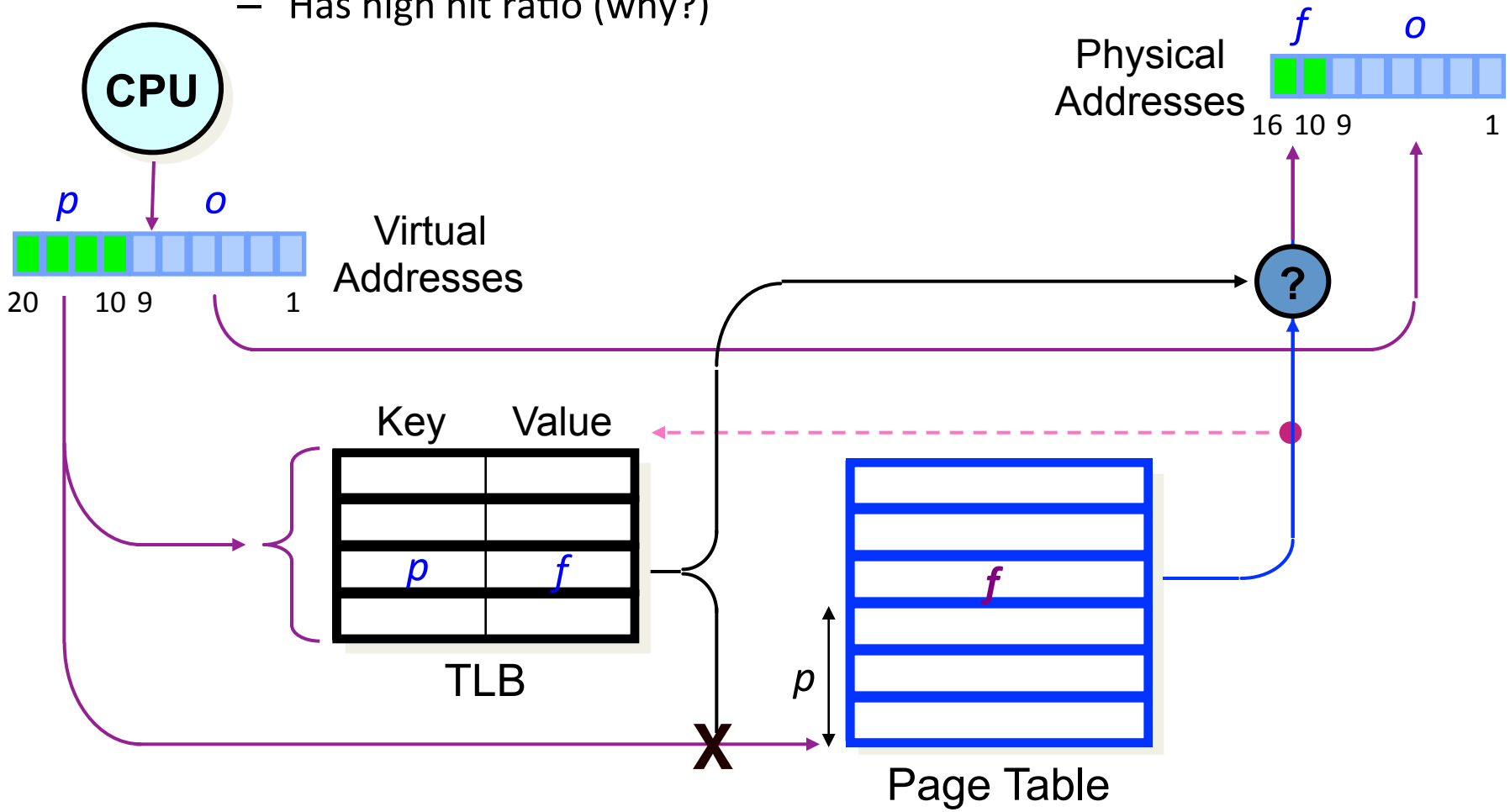
Back to Architecture: Text Description

- There is a system bus that connects to the CPU, the Disk Controller, and the Memory Controller. It controls the flow of information between them.
- The Disk Controller connects to disks and allows the System Bus to interface with them. It creates the abstraction between the system and the disk hardware.
- The Memory Controller connects to memory and controls the abstraction between the system and the memory hardware.
- The System Bus interacts with the CPU: it gets a Physical Address from the TLB/MMU and sends/receives data from the Processor and/or Cache.
- The TLB (Translation Lookaside Buffer)/ MMU (Memory Management Unit) are an intermediary between the System Bus and Processor for Physical Addresses. The Processor sends Virtual Addresses to the TLB/ MMU, they are translated into physical addresses and then sent along the System Bus. The TLB is a cache for these translations, as we will see next.

Improving Speed: Translation Lookaside Buffers (TLBs)

Cache recently accessed page-to-frame translations in a TLB

- For TLB hit, physical page number obtained in 1 cycle
- For TLB miss, translation is updated in TLB
- Has high hit ratio (why?)



Improving Speed: Translation Lookaside Buffers (TLBs) Text Description

- The TLB adds an extra step to getting a frame number from a page number.
  - The TLB holds recently used frame/page pairings.
  - TLBs have high hit ratio because they exploit locality within the system
  - For a TLB hit, the translation can be completed in 1 cycle
- The system simultaneously sends the page number to both the TLB and the page table looking for a frame number. If there is a hit in the TLB then it stops looking in the page table and sends the frame number along. If there is a miss in the TLB then it finds the frame number in the page table, sends the frame number along, and updates the TLB.

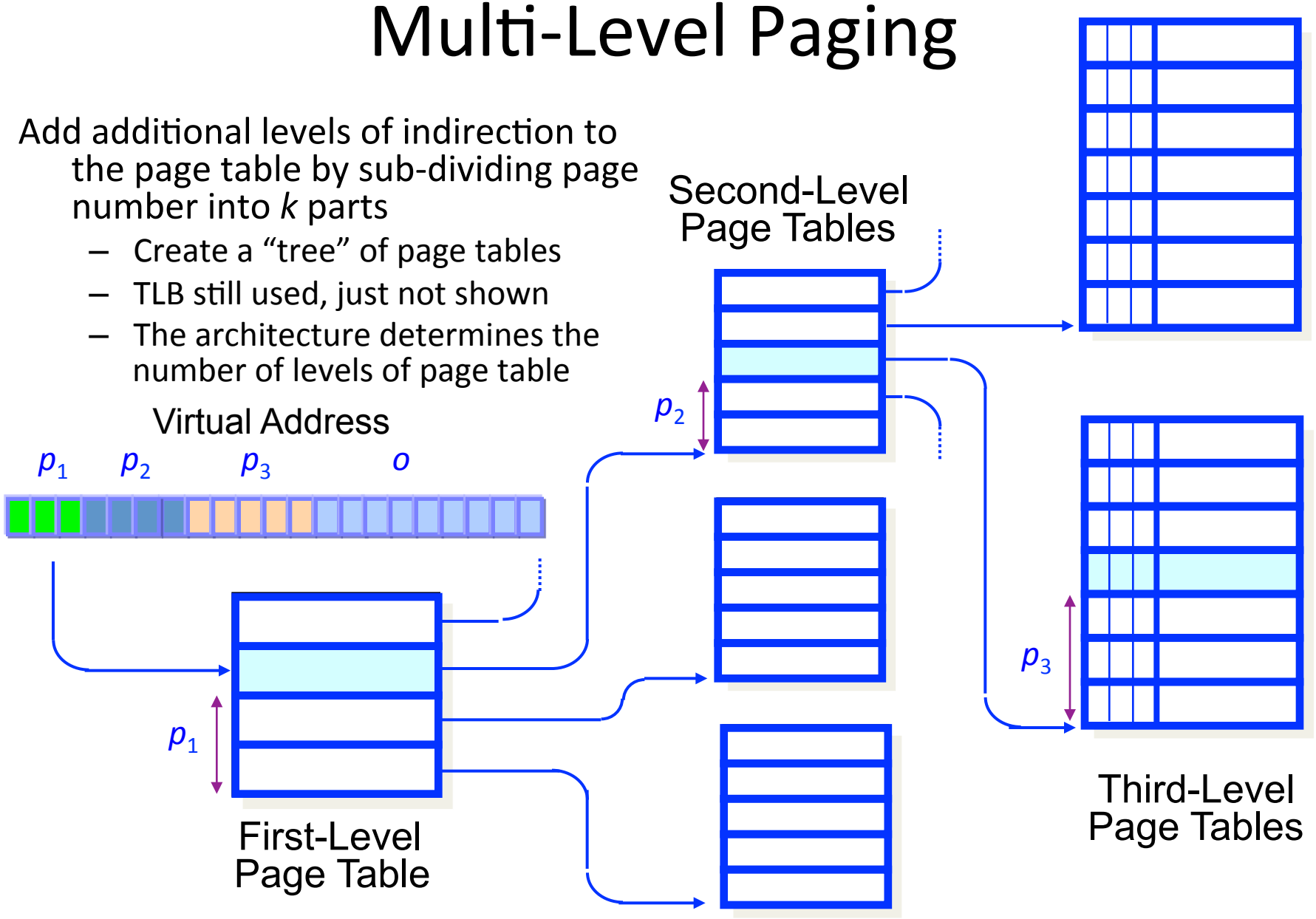
Performance Issues: Space

- Two sources:
  - data structure overhead (the page table)
  - fragmentation (*How large should a page be?*)
- Page table can be very large!
  - 32-bit addresses, 4096-byte pages… 1 million pages!
    - 32-bit addresses imply a VAS of 2^32 bytes
    - Divide VAS into pages (2^32/2^12)
    - 1 million pages (2^20)

Dealing With Large Page Tables: Multi-Level Paging

Add additional levels of indirection to the page table by sub-dividing page number into *k* parts

- Create a "tree" of page tables
- TLB still used, just not shown
- The architecture determines the number of levels of page table

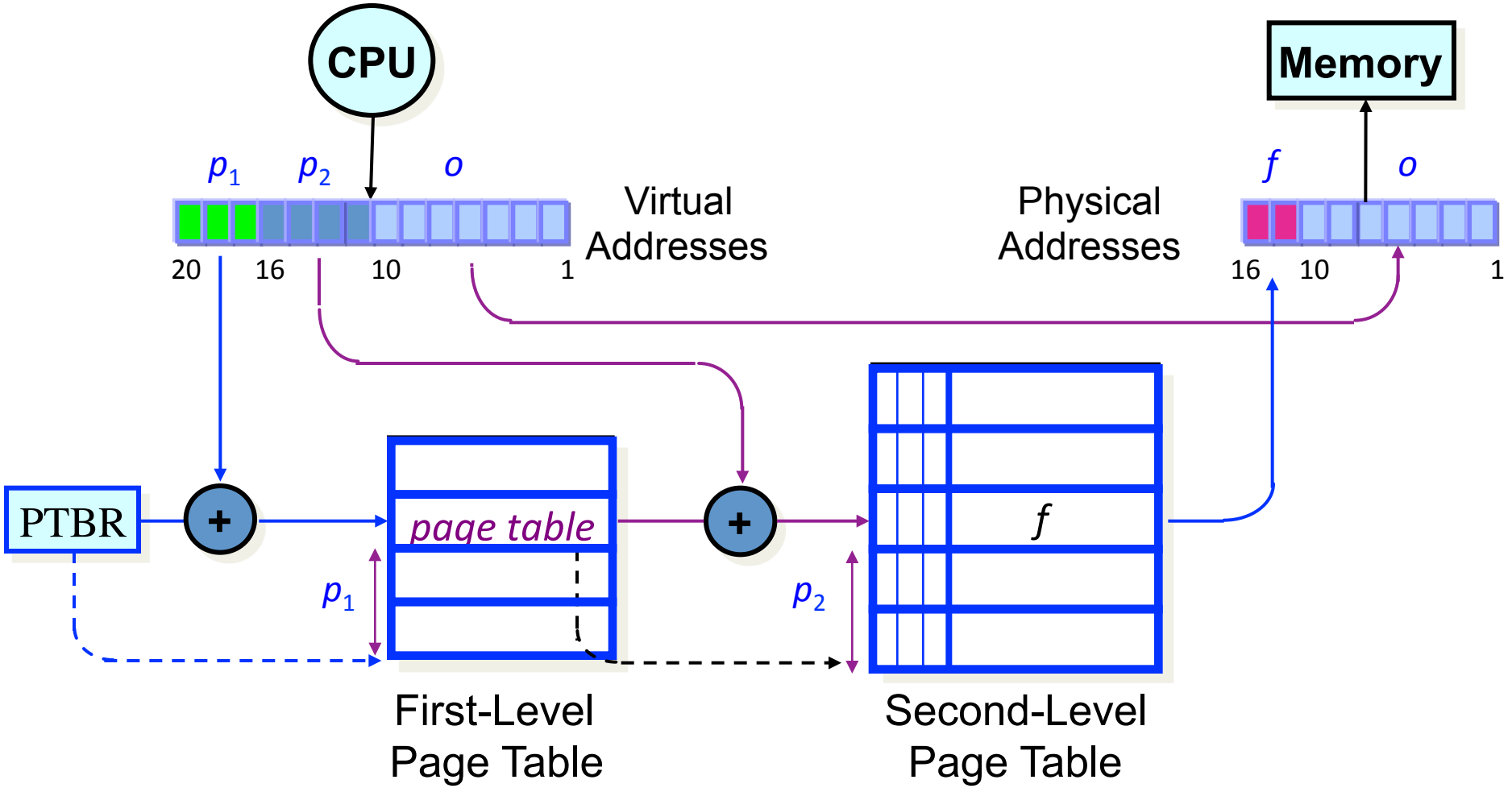


Dealing with Large Page Tables: Multi-Level Paging Text Description

- Add additional levels of indirection to the page table by subdividing page number into k parts. This creates a k-depth "tree" of page tables.
  - The architecture of the system determines k.
- Example: A system with 3 levels of page tables:
  - There are 3 bits for the first level page table
  - There are 4 bits for the second level page table
  - There are 5 bits for the third level page table
  - Note that if this were a single-level paged system, there would be 12 (3+4+5) bits for the page number
- Each entry in the first level page table points to a second level page table, and each entry in the second level page table points to a third level page table. The third level page table entries actually hold the bit/frame number information that is usually associated with a page table entry.
- Multi-Level page tables save space because you only have to allocate space for the page-table levels that you need. For example if you insanely only needed to hold information for 1 page table entry you would only need one first level page table, one second level table and one third level page table. It is important to note that you will always have only one first level page table.

Multi-Level Paging: Example

Example: Two-level paging



Multi-Level Paging: Example Text Description

- This example has two-level paging
- The first 3 bits are for the first level page table. The value of these bits is added to the PTBR (Page Table Base Register) which points to the beginning of the first level page table.
- The next 4 bits are for the second level page table. From the first level page table entry we get the address of the beginning of the second level page table, then we add the value from the second-level bits in the virtual address to this address. This gives us the address of the corresponding page table entry in the second level page table. Now we can access the relevant bits and frame number from the right second level page table entry.
- The TLB is very important now because each lookup from the page table requires a 2 step look up instead of just a single step.

64-bit Addresses

- Multi-level page tables work decently well for 32-bit address spaces… but what happens when we get to 64-bit?
- Too cumbersome (5 levels?)
- Rethink the page table:
  - Instead of making tables proportional to the size of the VAS, make them proportional to the size of the physical address space
  - One entry for each physical page---and hash!

  Result: Hashed/Inverted Page tables

Inverted Page Tables

- Also called *page registers*
- Each frame is associated with an entry
- Entry contains:
  - Residence bit: is the frame occupied?
  - Occupier: page number of the page in the frame
  - Protection bits

Inverted Page Tables: Example

- Physical Memory Size: 16MB
- Page Size: 4096 bytes
- Number of frames: 4096
- Space for page entries: 32 KB (8 bytes/entry)
- Percentage overhead: 0.2%
- Size of virtual memory: irrelevant

Inverted Page Tables: Virtual Address Translation

- As the CPU generates virtual addresses, where is the physical page?
  - Must search entire page table (32KB of Registers!)
  - Uh-Oh
- How can we solve that problem?
  - Cache it! (How did you know?)
  - Use the TLB to hold all the heavily used pages
  - But, um… the TLB is limited in size
  - So what about a miss in the TLB?

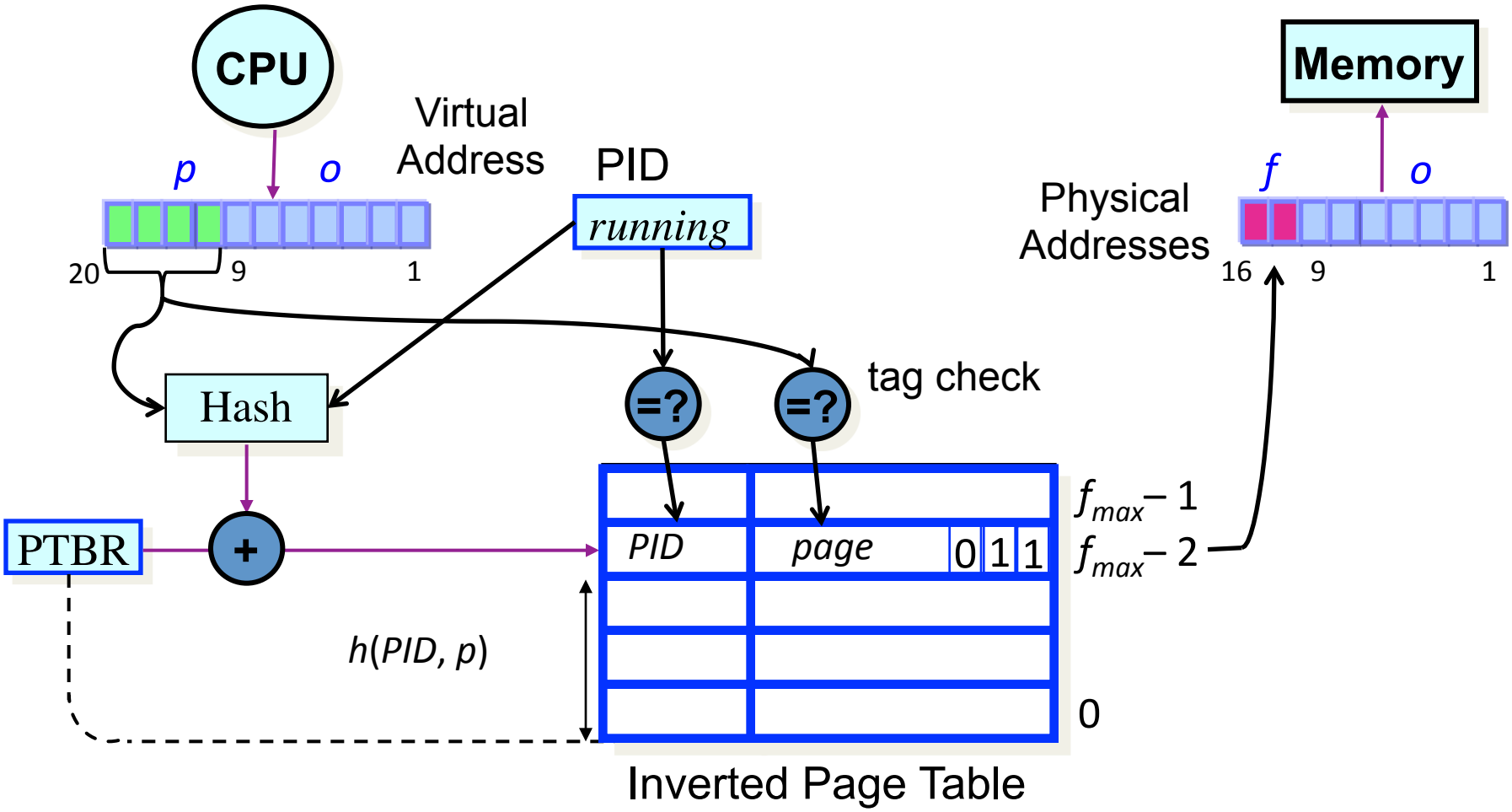
Use a Hash Table!

- Page *i* is placed in slot *f(i)* where *f* is an agreed-upon hash function
  - Deal with collisions using linked list or rehashing
- To look up page *i*, perform the following:
  - Compute *f(i)* and use it as an index into the table of page entries
  - Extract the corresponding page entry

Hashed Inverted Table In Use

Hash page numbers to find corresponding frame number

- Page frame number is not explicitly stored (1 frame per entry)
- Protection, dirty, used, resident bits also in entry



Hashed Inverted Table In Use: Text Description

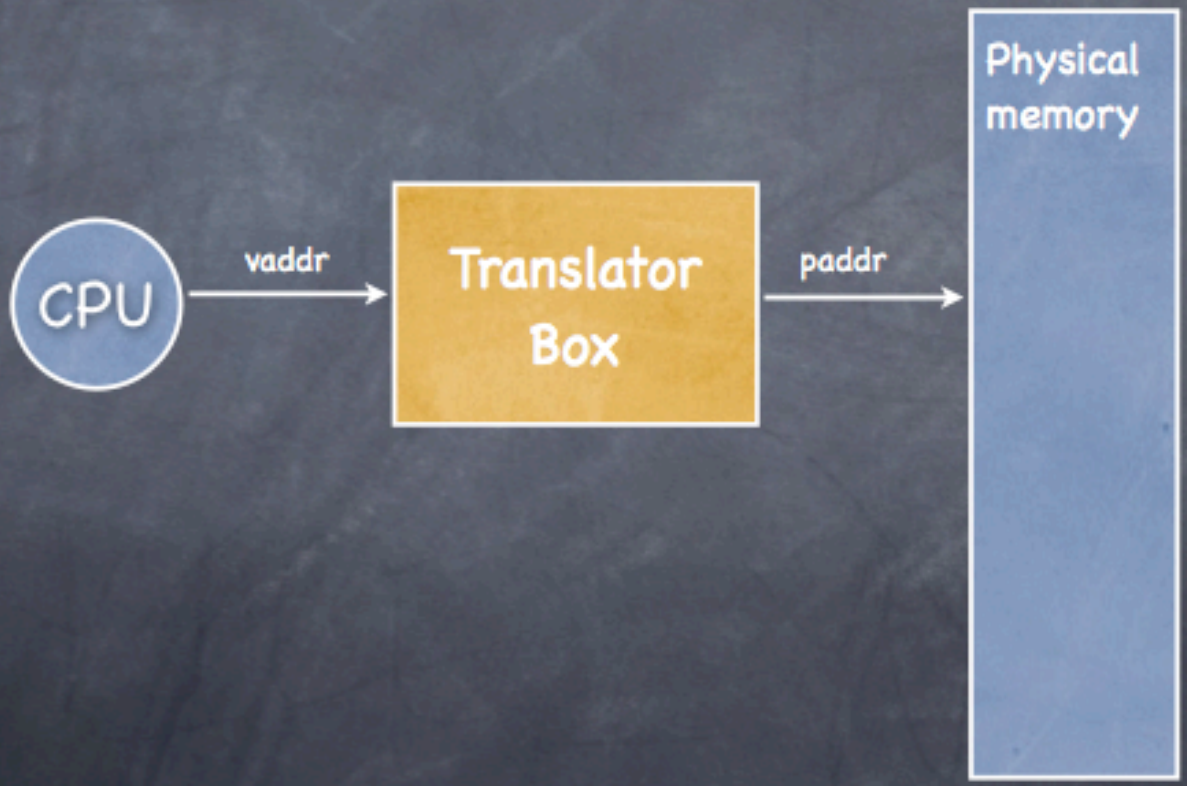
- Hash page numbers to find corresponding frame number
  - page frame number is not explicitly stored (1 frame per entry)
  - protection, dirty, used, and resident bits also in entry
- The hash function takes in the PID of the process and the page number. The answer from this hash function is added to the PTBR to get the address of the right page table entry.
- Once at the right page table entry the PID and page number saved in the entry are checked against the given PID and page number. If they do not match, then the frame isn't holding information for that page and it needs to be swapped in. If they do match then the right frame has been found and the frame number is sent along.

iClicker Question

Why use hashed/inverted page tables?

A. Regular (forward-mapped) page tables are too slow
B. Forward-mapped page tables don't scale to larger virtual address spaces
C. Inverted page tables have a simpler lookup algorithm, so the hardware that implements them is simpler
D. Inverted page tables allow a virtual page to be anywhere in physical memory

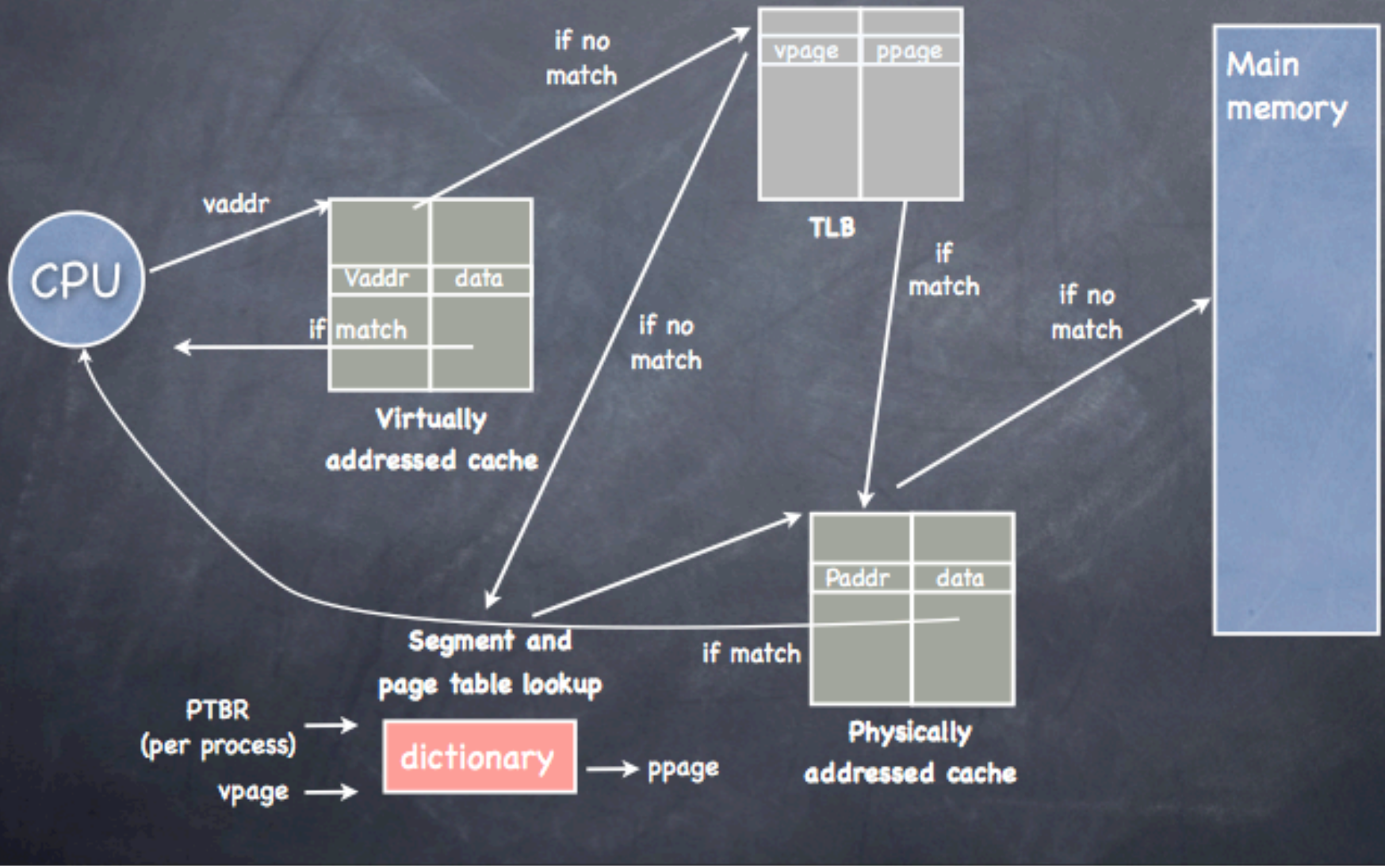
The BIG picture



The BIG Picture (1): Text Description

- The CPU generates virtual addresses
- The address is translated into a physical address
- The physical address is used to reference memory

The BIG picture

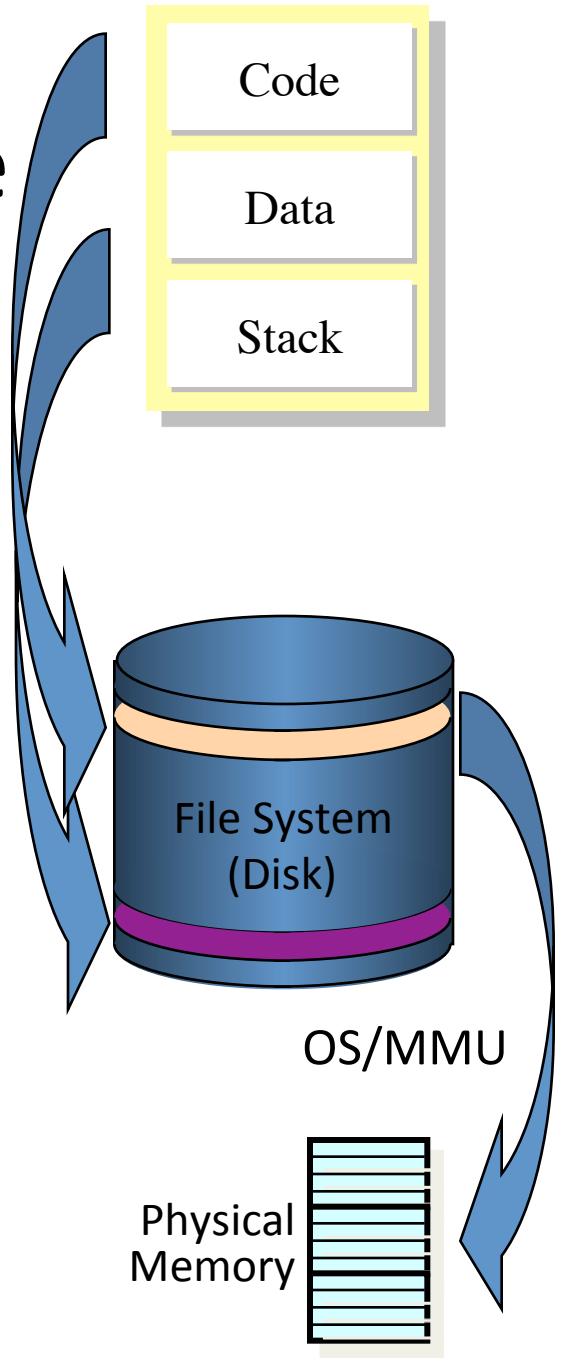


The BIG Picture (2): Text Description

- The CPU generates a virtual address
- That virtual address is used to reference a virtually addressed cache. If there's a match, we're finished
- If there is no match, the address is used to access the TLB, if there is a match, then the resulting physical address is used to access the physically addressed cache.
- If no match, then the translation must from from the page table, which is located using the Page Table Base Register (PTBR) and then indexed using the page number. The resulting frame number is combined with the offset and used to access the physically addressed cache.
- If there is a match in the physically addressed cache, we're finished. Otherwise, the physical address is used to access main memory.

Virtual Memory: The Bigger Picture

- A process's VAS is its context
  - Contains its code, data, and stack
- Code pages are stored in a user's file on disk
  - Some are currently residing in memory; most are not
- Data and stack pages are also stored in a file
  - Although this file is typically not visible to users
  - File only exists while a program is executing
- OS determines which portions of a process's VAS are mapped in memory at any one time

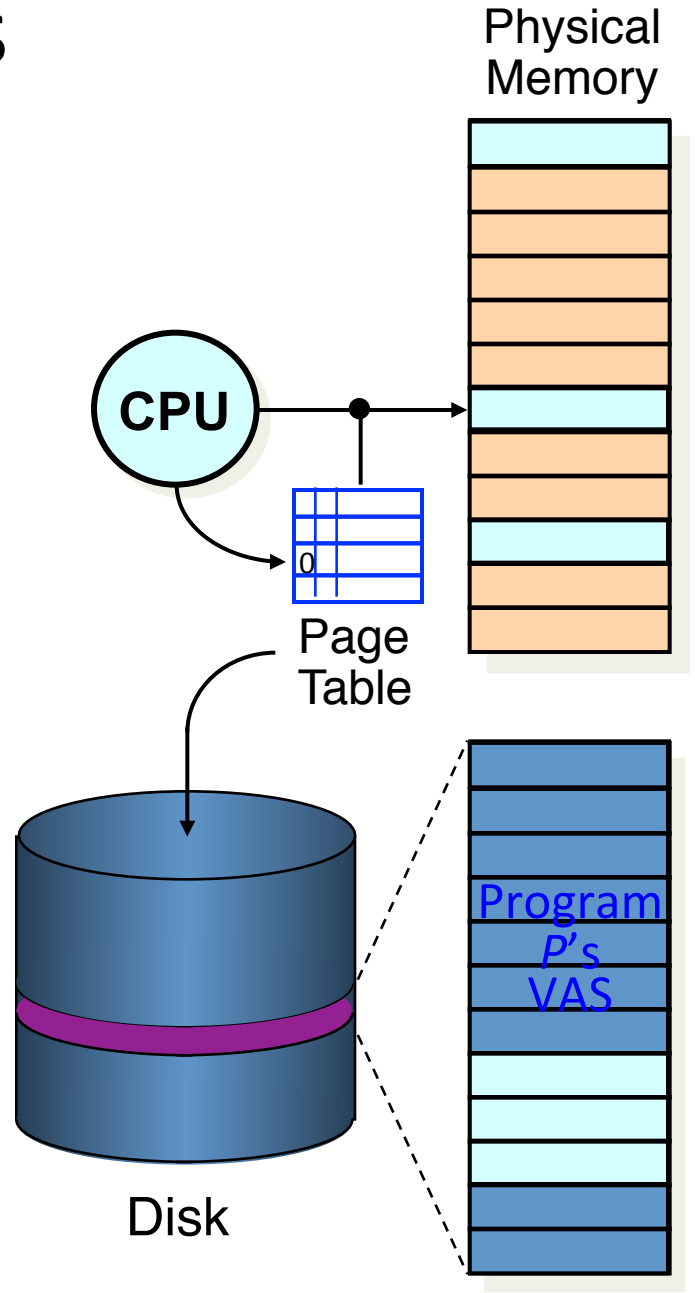


Page Faults

References to non-mapped pages generate a *page fault*

Page fault handling steps:

- Processor runs the interrupt handler
- OS blocks the running process
- OS starts read of the unmapped page
- OS resumes/initiates some other process
- Read of page completes
- OS maps the missing page into memory
- OS restarts the faulting process



Summary

Paging is a commonly used virtual memory mechanism

- Virtual address space is treated as pages
- Physical address space is treated as page frames
- Address translation is necessary
- Page tables are used: TLBs speed them up
  - Multilevel and Inverted

Announcements

- Homework 5 due 8:45a Friday
- Project 2 stack check due Monday
  - Sign up schedule will be posted (watch Piazza or Project page)
  - Be certain to follow instructions and provide all information (CS logins! NOT EIDs!)
- Canvas group sign ups must be completed by tomorrow at noon.