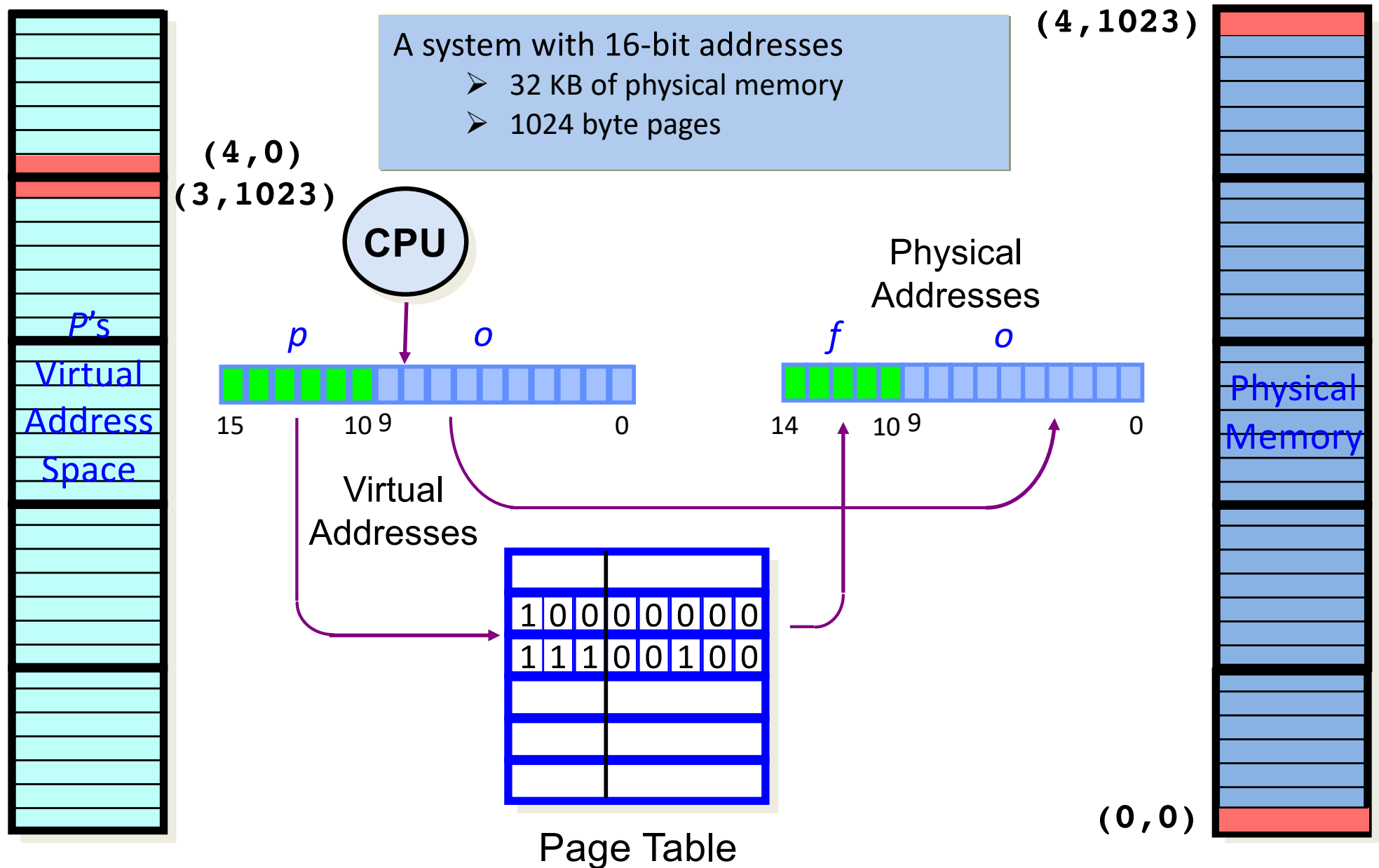


Improving Performance in Time

Address Translation: Current Picture



Virtual Address Translation:

Text Description

Steps to Virtual-Physical Memory Translation:

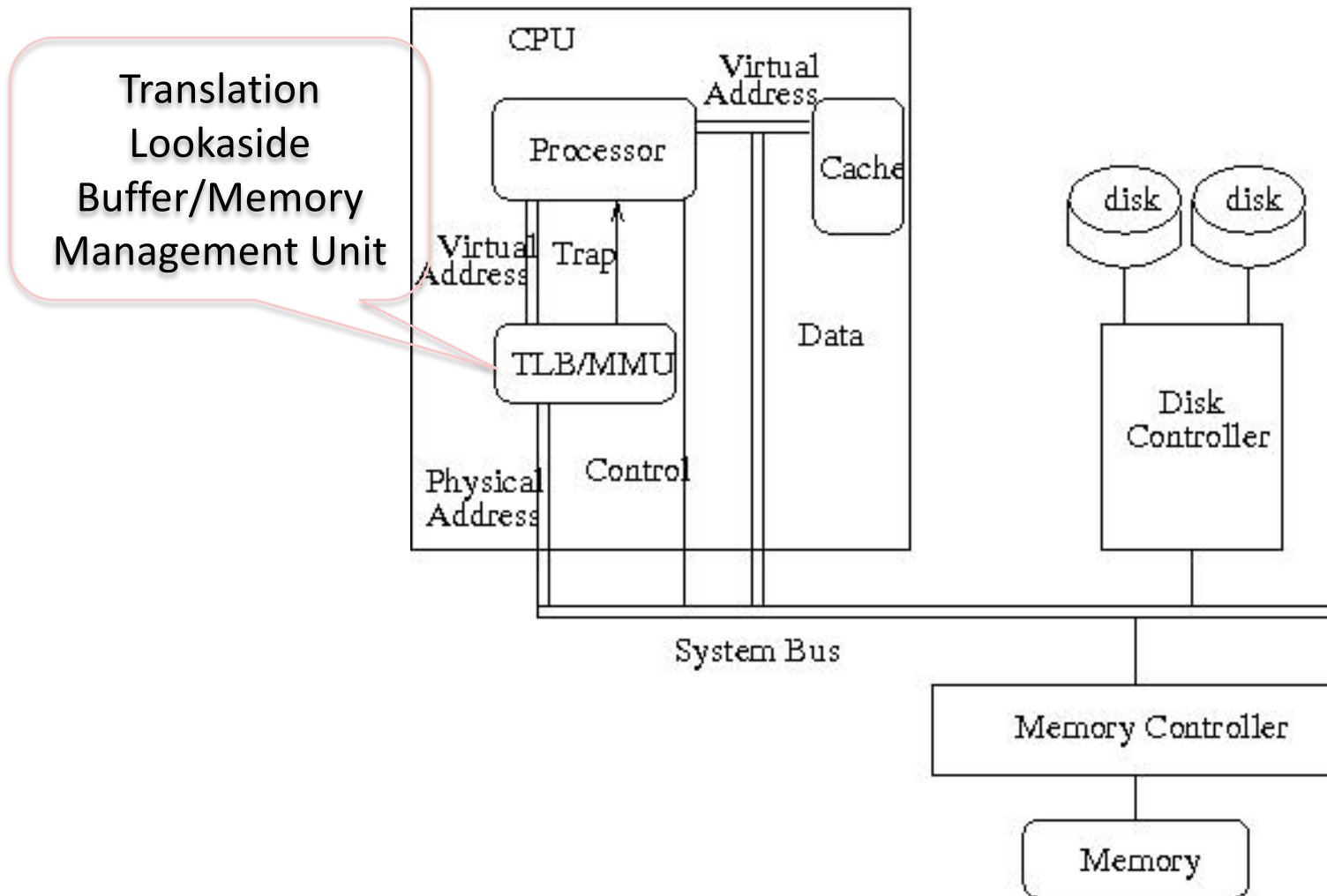
- The program gives a virtual address to the CPU to translate
- The MMU splits the virtual address into its page and offset numbers
- Since the offset is the same in Virtual and Physical memory it is sent along with no change
- The page number is translated into a frame number
 - Look into page table 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

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:

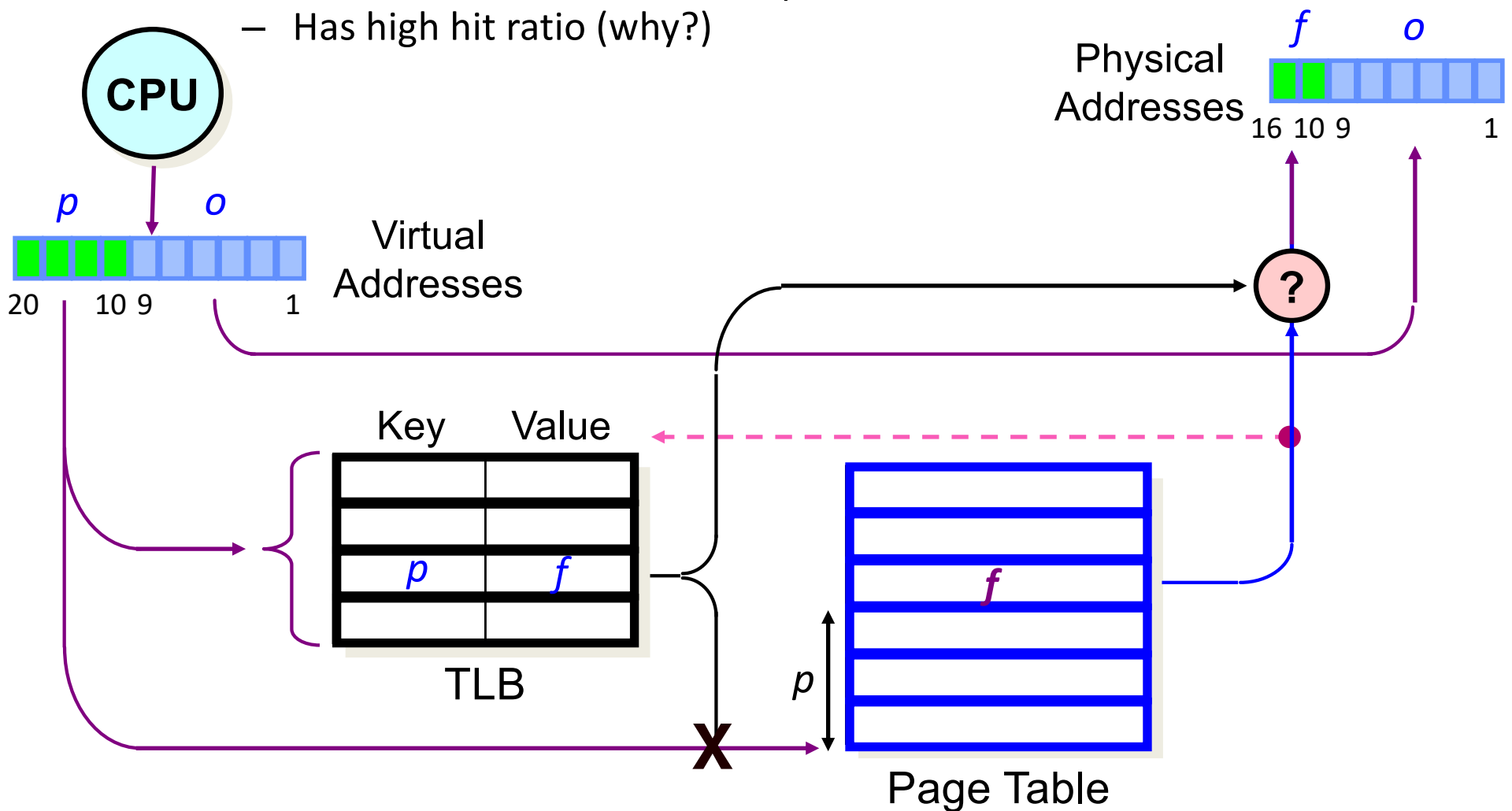
Text Description

- 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:

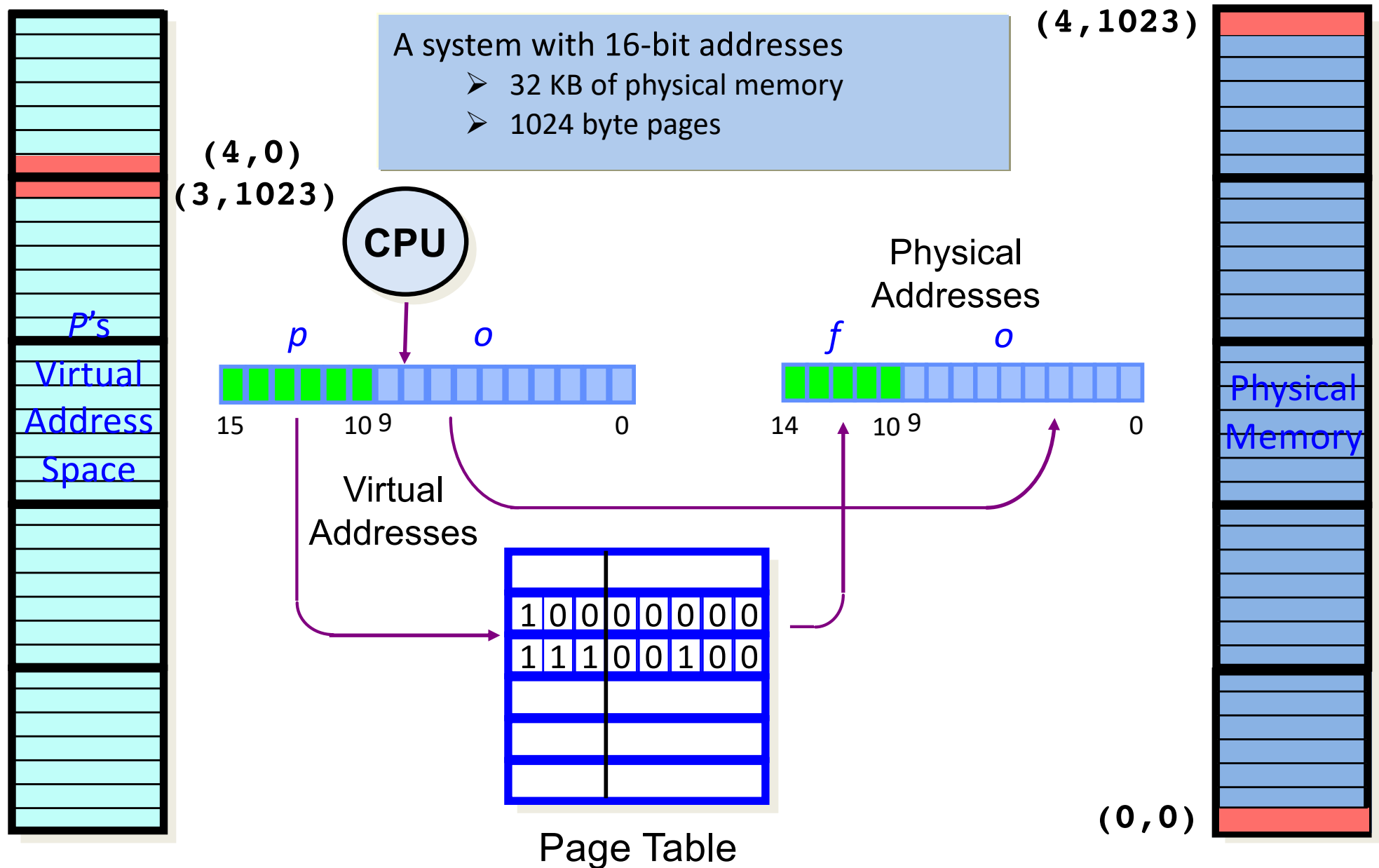
Translation Lookaside Buffers (TLBs)

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.

Improving Performance in Space

Address Translation: Current Picture



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

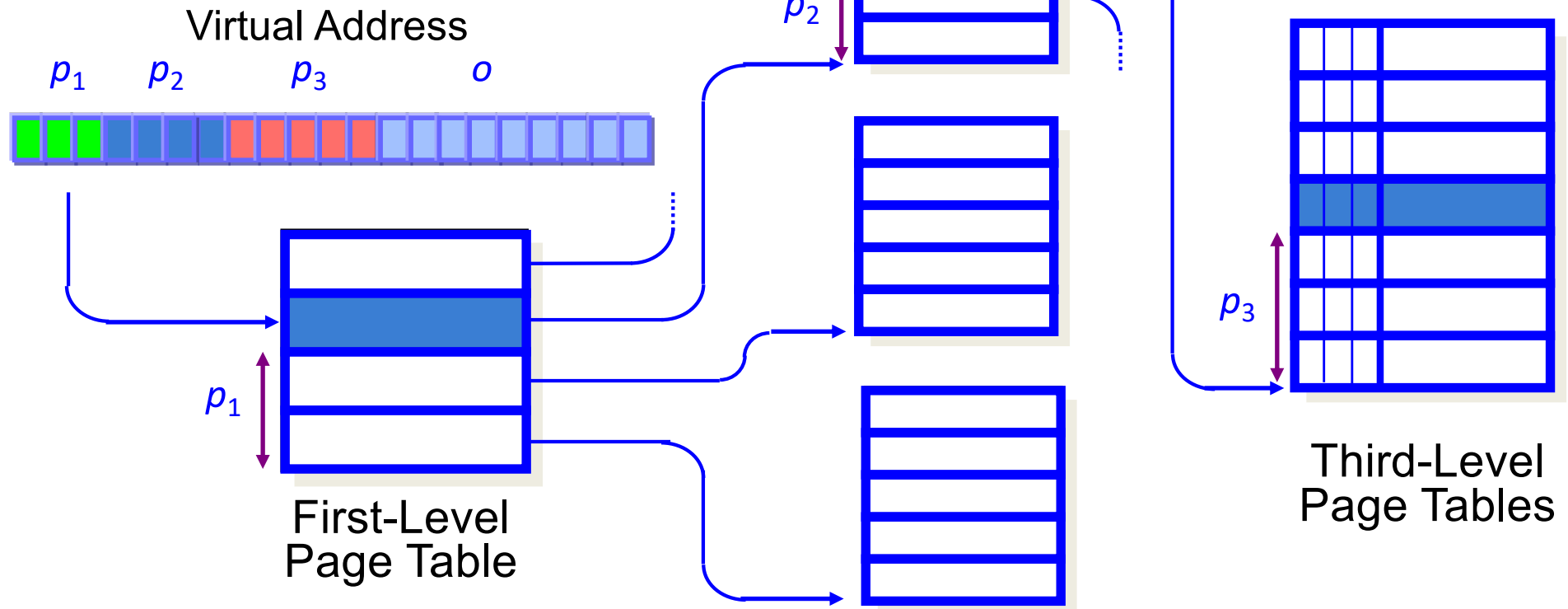
Performance Issues: Space

- 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

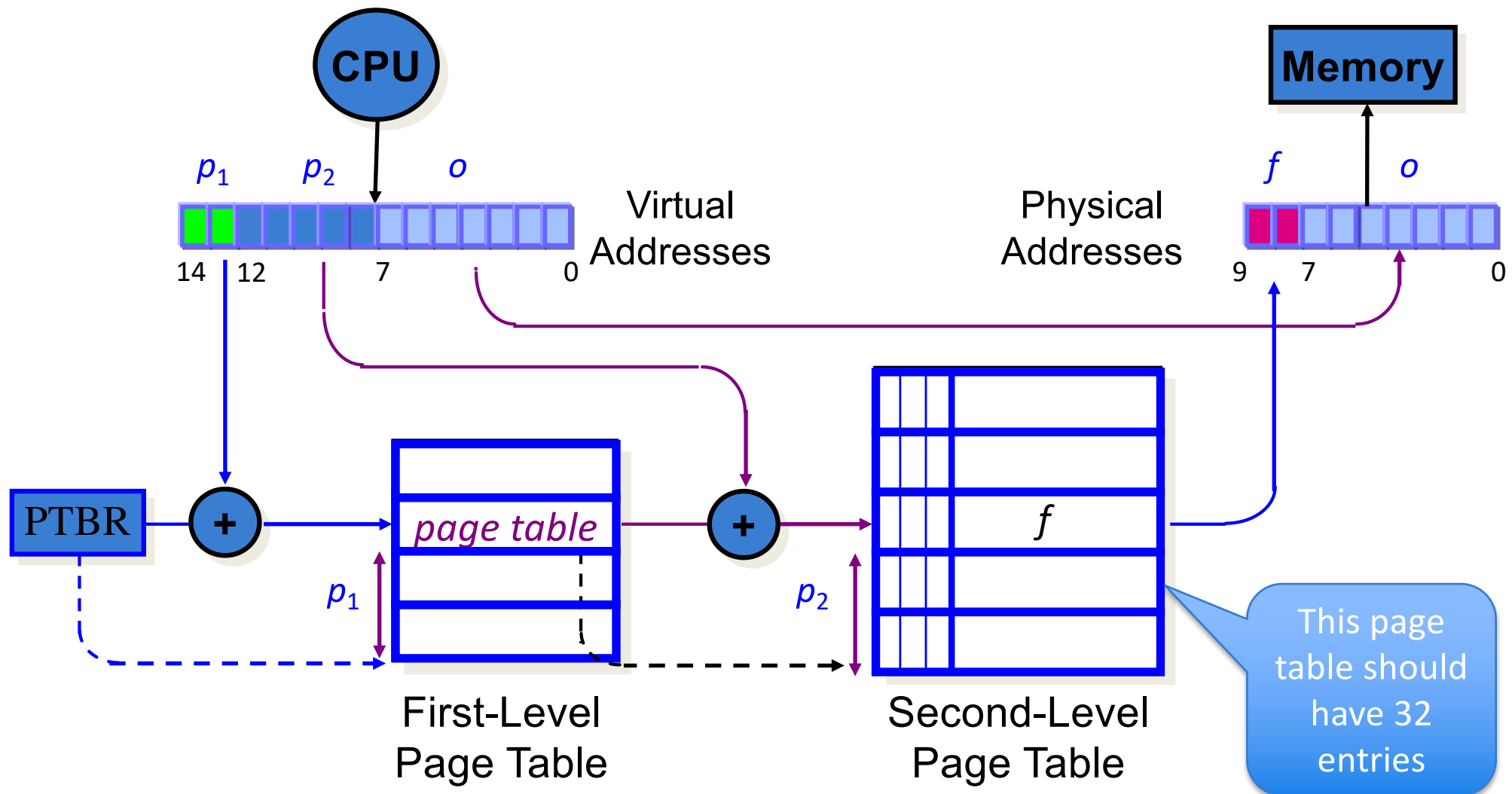


Dealing with Large Page Tables: Multi-Level Paging Text Description

- 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.

Improving Performance in Space AND Time

But wait...

- 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

Inverted Page Tables

- 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:

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?

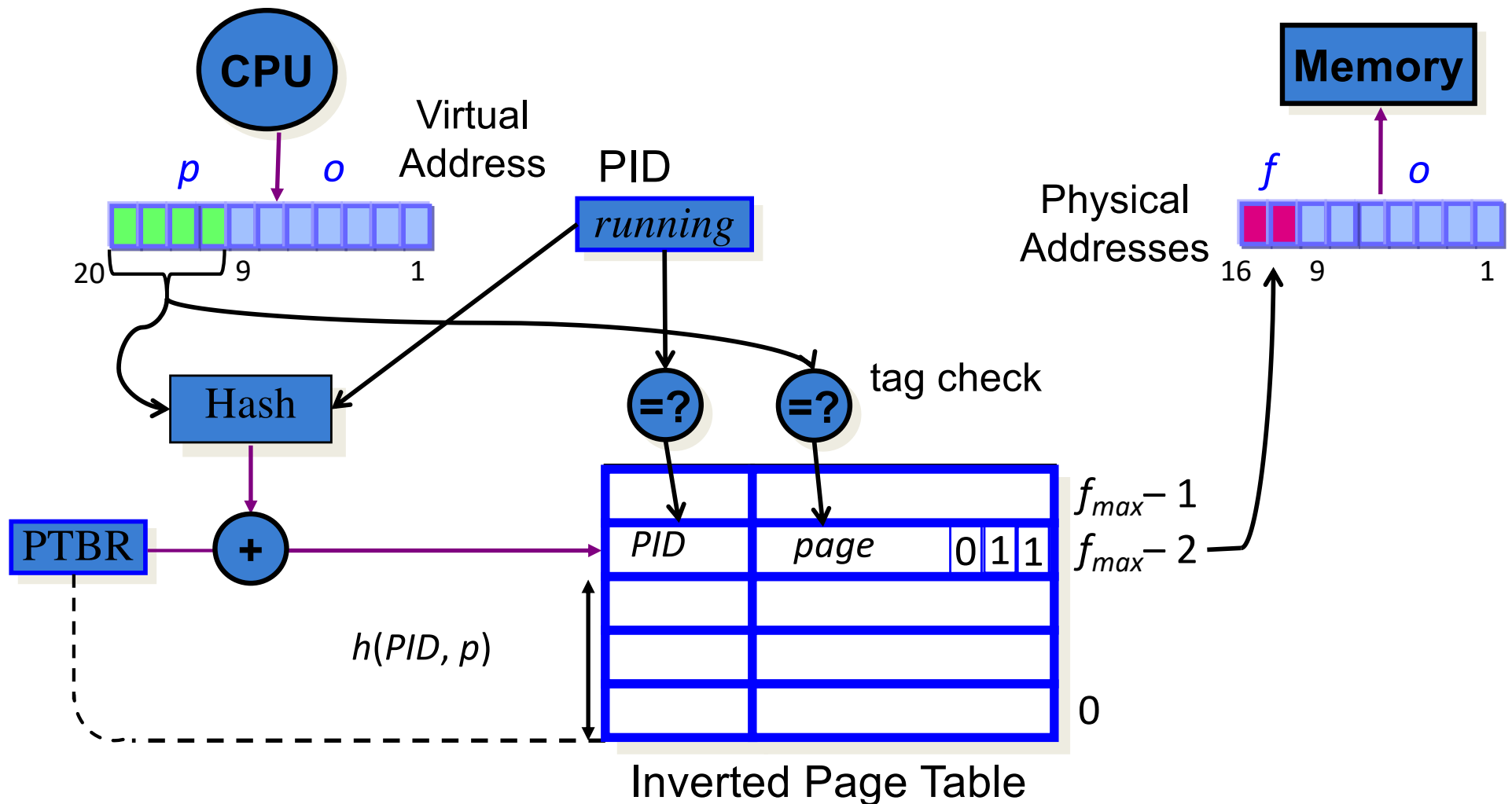
Use a Hash Table!

- 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.

Question

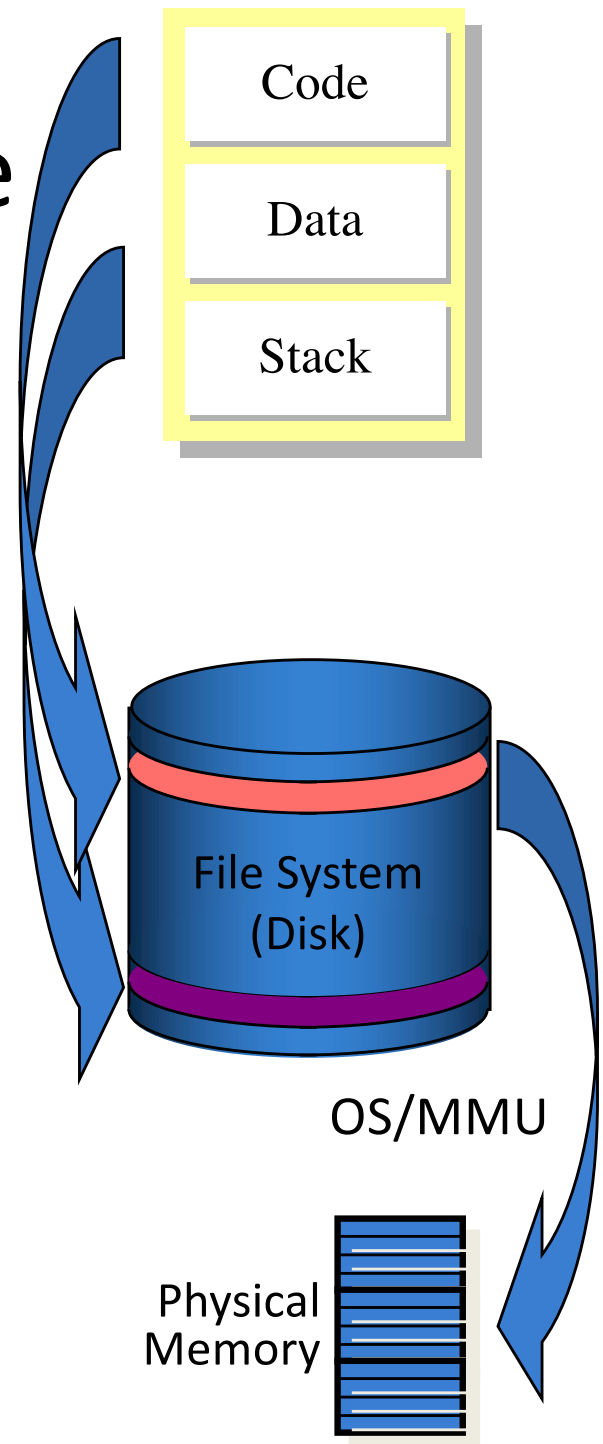
Why use hashed/inverted page tables?

- 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

Swapping

Virtual Memory: The Bigger Picture

- A process's VAS contains its code, data, and stack
- Code pages are stored in a user's file on disk
 - Pages currently in use reside in memory
- Data and stack pages are also stored on disk when not in memory
 - In the swap partition
 - *Anonymous files* that only exist while a program is executing
- OS determines which portions of a process's VAS are mapped in memory at any one time

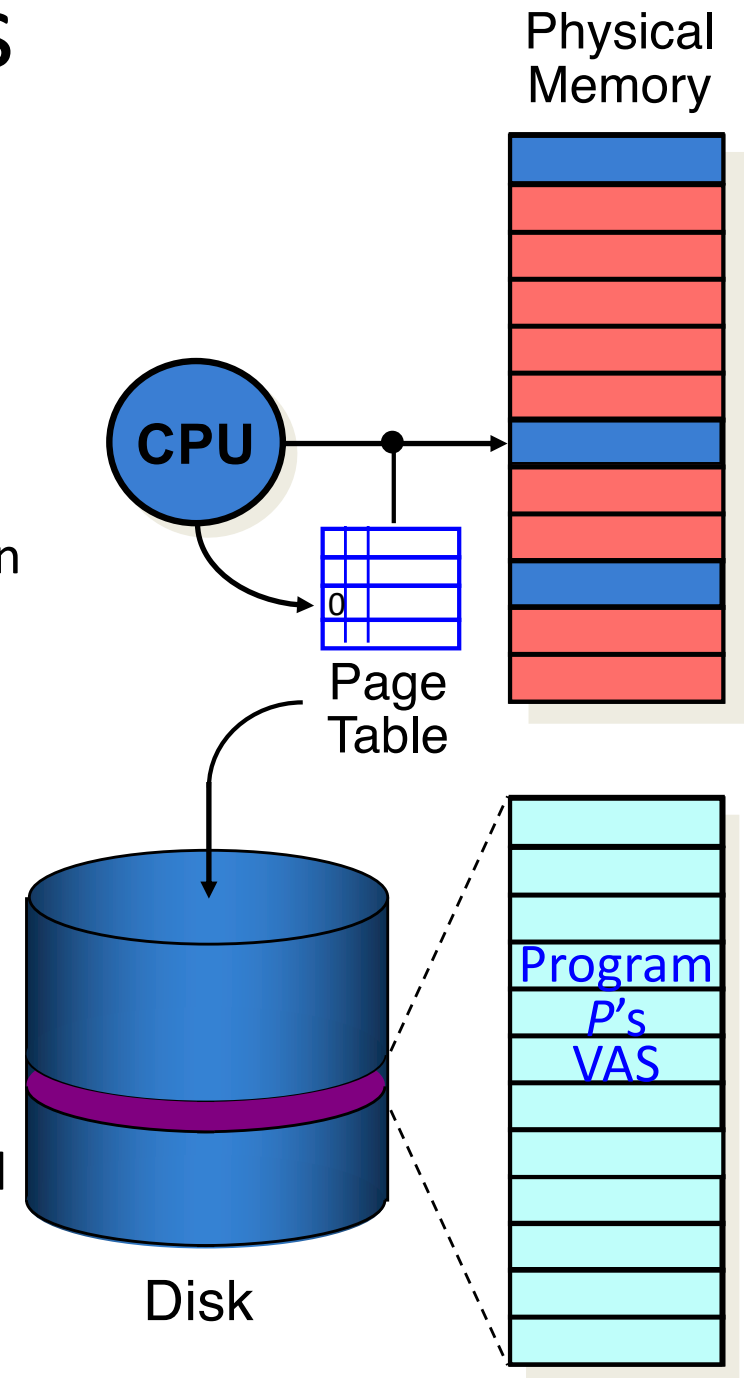


Page Faults

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

Page fault handling steps:

1. Processor runs the interrupt handler
2. Page fault handler determines the reason for the page fault (exception or not resident)
3. If exception, process is killed
4. If not resident, it is a true page fault
5. Handler locates unmapped page and initiates the read of that page
6. Faulting process blocks and another process is scheduled
7. Read of page completes, page is mapped into physical memory
8. Faulting process is unblocked



The BIG Picture

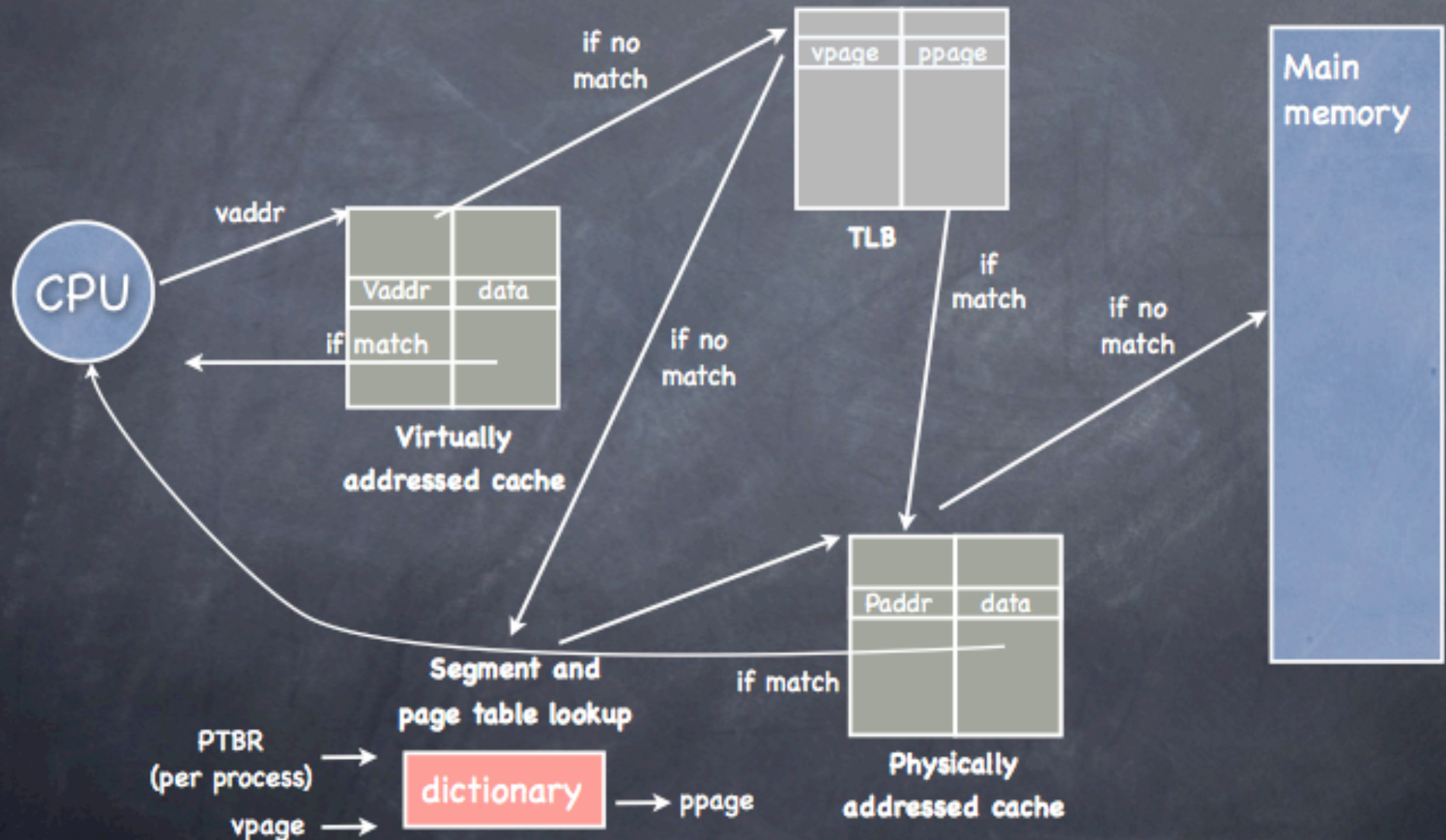
The BIG picture



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 come 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.

Policies

Paging: So Many Questions

- When should a process's pages be loaded into memory?
- If memory is full but a page fault has just occurred (so a process needs another page!), what should happen?
 - If a page should be replaced, which one?
- What is a good page size?
- How many processes are too many?

Loading Pages

- *Demand paging*: A page is loaded the first time it is referenced
- *Pre-paging*: OS guesses in advance which pages the process will need and pre-loads them

Page Replacement Algorithms

What happens when we have a page fault.... but memory is full?

Page Replacement Algorithms: Goals

Improve OS performance by:

- Reducing the frequency of page faults
- Being efficient

Overarching goal: *Make good decisions quickly*

Page Replacement Algorithms: Optimal

- Look into the future and throw out the page that will be accessed farthest in the future
- Provably optimal

Optimal Page Replacement

Clairvoyant replacement

- ◆ Replace the page that won't be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>										
	1	<i>b</i>										
	2	<i>c</i>										
	3	<i>d</i>										
Faults												
Time page needed next												



Optimal Page Replacement:

Text Description

- Clairvoyant replacement---impossible in real life
- Replace the page that won't be needed for the longest time in the future
- Time and requests on x axis
- Page frames on the y axis
- Keep track of faults as you go
- At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.
- The request string is c, a, d, b, e, b, a, b, c, d.
- Each request occurs at another time tick, so requests are at times 1-10.

Optimal Page Replacement: Solution

Clairvoyant replacement

- ◆ Replace the page that won't be needed for the longest time in the future

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>d</i>
	1	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
	2	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>
	3	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>
Faults							•					•
Time page needed next							<i>a</i> = 7 <i>b</i> = 6 <i>c</i> = 9 <i>d</i> = 10					<i>a</i> = 15 <i>b</i> = 11 <i>c</i> = 13 <i>d</i> = 14

Optimal Page Replacement: Solution

Text Description

- Performance with 4 frames and 10 requests:
 - request order: c a d b e b a b c d
 - we will keep this order in mind as we run the algorithm
 - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
 - Time 1: request page c, in memory so no page fault
 - Time 2: request page a, in memory so no page fault
 - Time 3: request page d, in memory so no page fault
 - Time 4: request page b, in memory so no page fault
 - Time 5: request page e which isn't in memory, causes a page fault
 - look at future requests: b a b c d, see that d is the one farthest in the future
 - put e where d used to be
 - current frame setup: a in frame 0, b in frame 1, c in frame 2 and e in frame 3
 - Time 6: request page b, in memory so no page fault
 - Time 7: request page a, in memory so no page fault
 - Time 8: request page b, in memory so no page fault
 - Time 9: request page c, in memory so no page fault
 - Time 10: request page d which isn't in memory, causes a page fault
 - don't have any future requests so just pick something to kick out
 - put d where a used to be
 - final frame setup: d in frame 0, b in frame 1, c in frame 2 and e in frame 3
 - Total number of page faults: 2

Page Replacement Algorithms: LRU

Page Replacement Algorithms: LRU

Least Recently Used

- Works well when the recent past is a good predictor of the future
- Throw out the page that has not been used in the longest time

Least Recently Used Page Replacement

Use the recent past as a predictor of the near future

- ◆ Replace the page that hasn't been referenced for the longest time

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>										
	1	<i>b</i>										
	2	<i>c</i>										
	3	<i>d</i>										
Faults												
Time page last used												



Least Recently Used Page Replacement Algorithm:

Text Description

- Use the recent past as a predictor of the near future
- Replace the page that hasn't been referenced for the longest time
- Time and requests on x axis
- Page frames on the y axis
- Keep track of faults as you go
- At time 0, pages a, b, c, and d are loaded into frames 0, 1, 2, and 3 respectively.
- The request string is c, a, d, b, e, b, a, b, c, d.
- Each request occurs at another time tick, so requests are at times 1-10.

Least Recently Used Page Replacement: Solution

Use the recent past as a predictor of the near future

- ◆ Replace the page that hasn't been referenced for the longest time

Time	0	1	2	3	4	5	6	7	8	9	10
Requests		<i>c</i>	<i>a</i>	<i>d</i>	<i>b</i>	<i>e</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>
Page Frames	0	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
	1	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>
	2	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>d</i>
	3	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>c</i>	<i>c</i>
Faults						•				•	•
Time page last used					<i>a</i> = 2				<i>a</i> = 7	<i>a</i> = 7	
					<i>b</i> = 4				<i>b</i> = 8	<i>b</i> = 8	
					<i>c</i> = 1				<i>e</i> = 5	<i>e</i> = 5	
					<i>d</i> = 3				<i>d</i> = 3	<i>c</i> = 9	

Least Recently Used Page Replacement: Solution

Text Description

- Performance with 4 frames and 10 requests:
 - request order: c a d b e b a b c d
 - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
 - Time 1: request page c, in memory so no page fault
 - Time 2: request page a, in memory so no page fault
 - Time 3: request page b, in memory so no page fault
 - Time 4: request page d, in memory so no page fault
 - Time 5: request page e which isn't in memory, causes a page fault
 - look back in time and see that c was used least recently
 - past requests: a used at time 2, b used at time 4, c used at time 1 and d used at time 3
 - put e where c used to be
 - current frame setup: a in frame 0, b in frame 1, e in frame 2 and d in frame 3
 - Time 6: request page b, in memory so no page fault
 - Time 7: request page a, in memory so no page fault
 - Time 8: request page b, in memory so no page fault
 - Time 9: request page c which isn't in memory, causes a page fault
 - look back in time and see that d was used least recently
 - past requests: a used at time 7, b used at time 8, e used at time 5 and d used at time 3
 - put c where d used to be
 - current frame setup: a in frame 0, b in frame 1, e in frame 2 and c in frame 3
 - Time 10: request page d which isn't in memory, causes a page fault
 - look back in time and see that e was used least recently
 - past requests: a used at time 7, b used at time 8, e used at time 5 and c used at time 9
 - put d where e used to be
 - current frame setup: a in frame 0, b in frame 1, d in frame 2 and c in frame 3
 - NOTE: replaced d and had to immediately bring it back in. So the past is not a perfect prediction of the future
 - Total number of page faults: 3

Question

What is the goal of a page replacement algorithm?

- A. Reduce the number of page faults
- B. Reduce the penalty for page faults when they do occur
- C. Minimize CPU time for a process