Virtual Memory (II)

CS439: Principles of Computer Systems

March 9, 2015

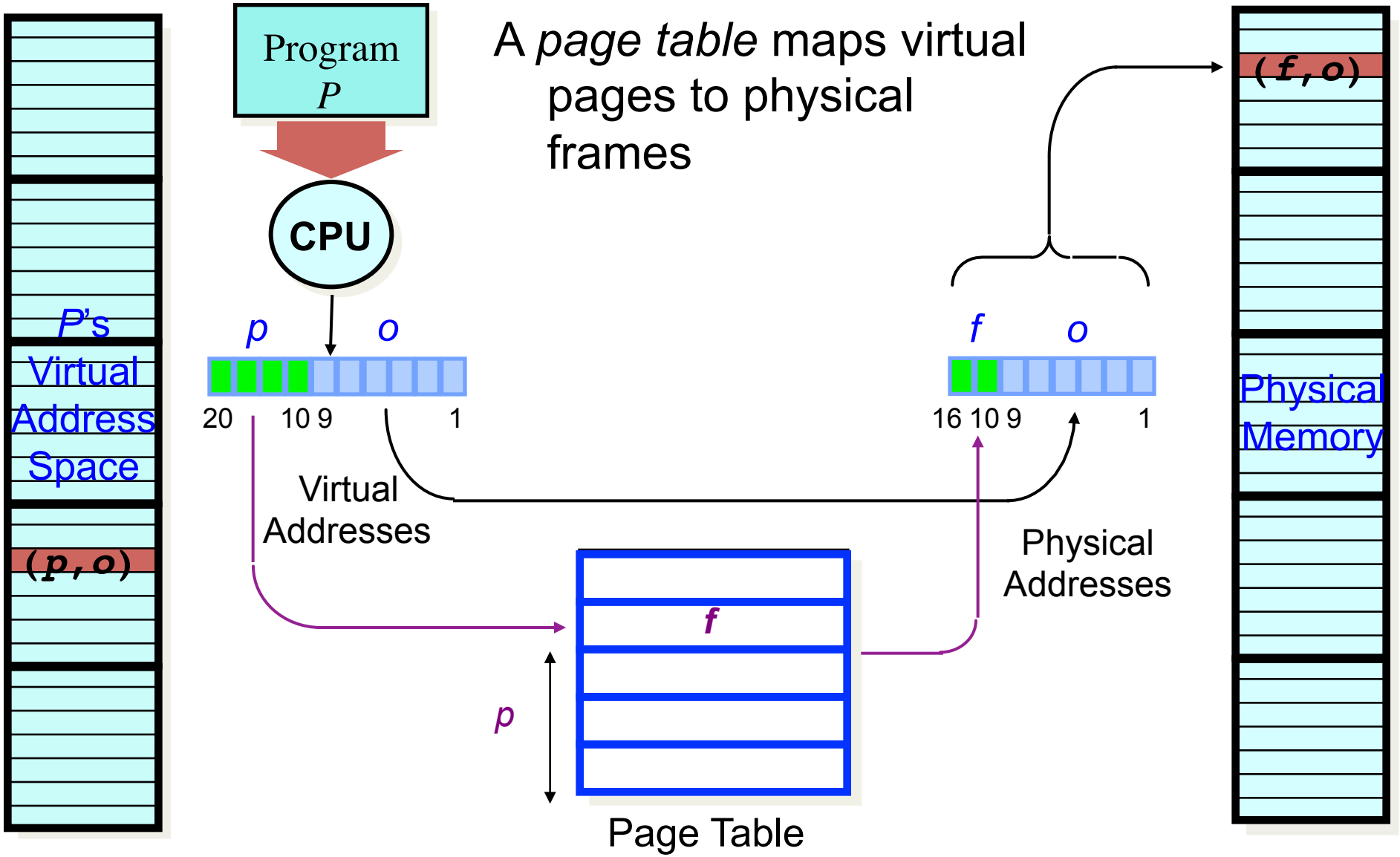
Last Time

- Overlays
- Paging
  - Pages
  - Page frames
  - Address translation
  - Forward-mapped page tables
  - Multi-level page tables
  - Inverted/Hashed page tables

Today's Agenda

- Page Faults
- Paging: Policy
  - Demand paging vs. pre-paging
  - Page Replacement Algorithms
    - FIFO
    - LRU
    - Clock
  - Effect of increased memory
  - Global vs. local
  - Page Sizes
- Multiprogramming
  - Swap

Virtual Address Translation



Virtual Address Translation: Text Description

Steps to Virtual-Physical Memory Translation:

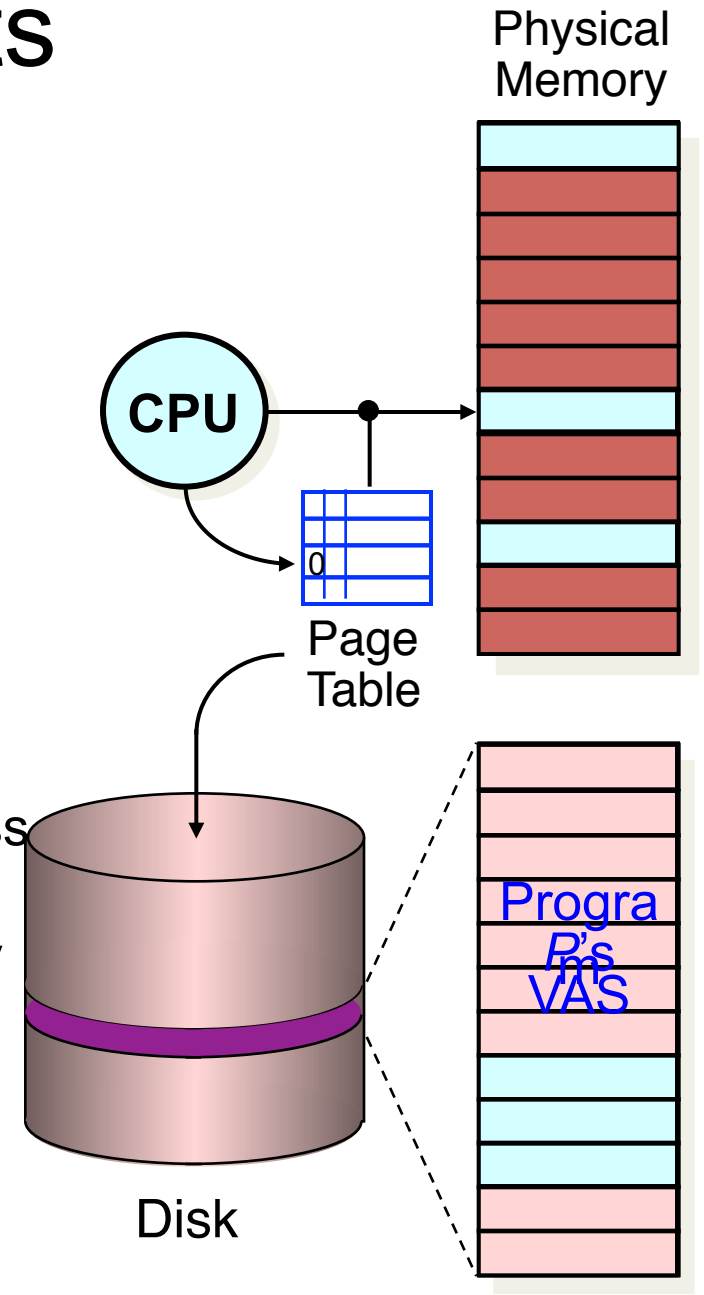
- The program gives a virtual address to the CPU to translate
- The MMU splits the virtual address into its page and offset numbers
- Since the offset is the same in Virtual and Physical memory it is sent along with no change
- The page number is translated into a frame number
  - Look into page table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

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



Paging: The Mechanism

- Physical and virtual memory are partitioned into equal size units (pages and page frames)
- Size of VAS unrelated to size of physical memory
- Virtual pages are mapped to physical frames
- There is no external fragmentation
- Missing pages cause page faults

Paging: The Policy

- 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?

When to Load a Page?

- *When a process starts*: Load all the pages
- *Overlays*: application programmer indicates to load and remove pages
- *Demand paging*: OS loads a page the first time it is referenced
- *Pre-paging*: OS guesses in advance which pages the process will need and pre-loads them

Initializing Memory: Pre-Paging

1. Process needing *k* pages arrives
2. If *k* page frames are free, the OS allocates all *k* pages to the free frames. (Otherwise, we'll need to replace…)
3. The OS puts the first page in a frame and puts the frame number in the first entry in the page table, and so on.
4. OS marks all TLB entries as invalid (flushes the TLB).
5. OS starts a process
6. As process executes, OS loads TLB entries as each page is accessed, replacing an existing entry if the TLB is full.

Initializing Memory: Demand Paging

1. A process arrives
2. The OS stores the process's VAS on disk and *does not* put any of it into memory
3. As the process executes, pages are faulted in

Performance of Demand Paging

- Theoretically, a process could access a new page of memory with each instruction (expensive!)
- But processes tend to exhibit *locality of reference*
  - *temporal locality*: if a process accesses an item in memory, it will tend to reference the same item again soon
  - *spatial locality*: if a process accesses an item in memory, it will tend to reference an adjacent item soon

What Happens When a Page is Referenced That is Not in Memory (and Memory is Full)

The OS:

1. selects a page to replace (page replacement algorithm)
2. invalidates the old page in the page table
3. starts loading the new page into memory from disk
4. context switches to another process while I/O is being done
5. gets interrupt that the page is loaded in memory
6. updates the page table entry
7. continues faulting process

Page Replacement Algorithms

- When a process faults and memory is full, some page must be swapped out…
- Reducing the frequency of page faults is critical to OS performance
- A slow page replacement algorithm will also adversely affect OS performance

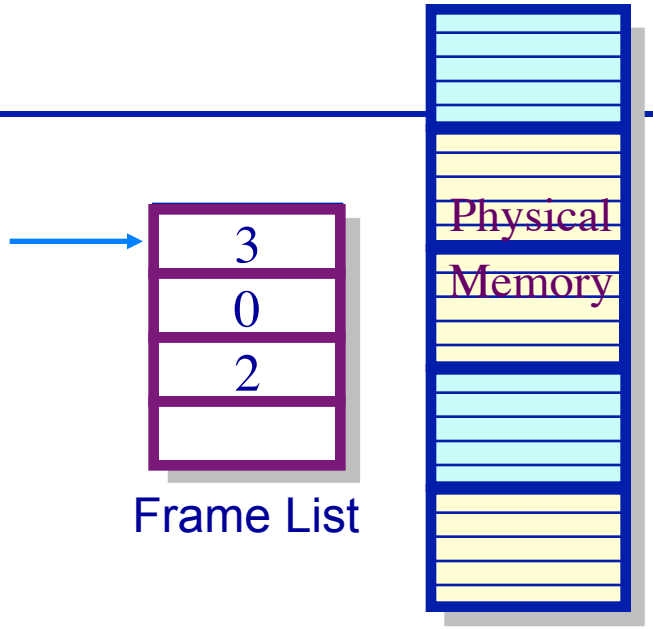
Page Replacement Algorithms: FIFO

First-In, First-Out

- Throw out the oldest page
- Simple to implement
- OS can easily throw out a page that is being accessed frequently

FIFO Replacement

- Implemented with a single pointer
- Performance with 4 page frames:



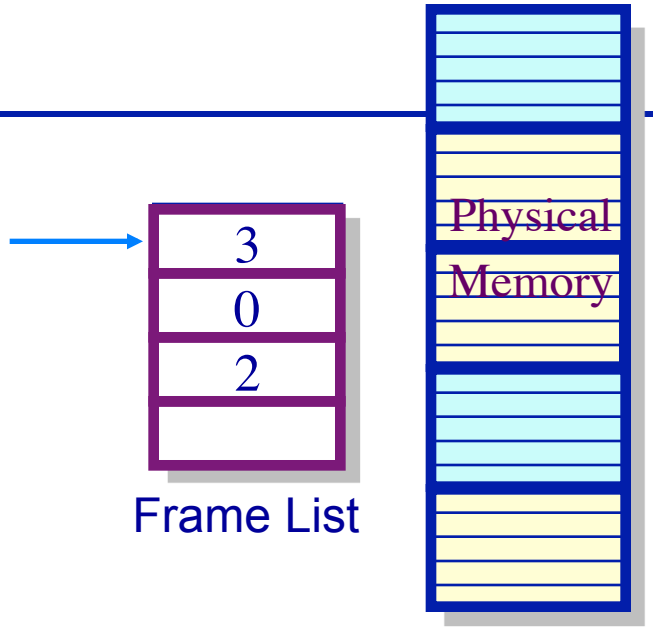
Time	0	1	2	3	4	5	6	7	8	9	10
Requests		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames 0	*a*										
Page Frames 1	*b*										
Page Frames 2	*c*										
Page Frames 3	*d*										
Faults											

FIFO Replacement: Text Description

- Implemented with a single pointer
  - pointer points to the oldest page
- Performance with 4 page frames:
  - keep track of time and requests made at each time
    - requests denoted with letters (a, b, c etc)
  - time shown on the x-axis
  - page frames kept track on the y-axis
  - keep count 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.

FIFO Replacement: Solution

- Implemented with a single pointer
- Performance with 4 page frames:



Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	***e***	*e*	*e*	*e*	*e*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	***a***	*a*	*a*	*a*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	***b***	*b*	*b*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•		•	•	•	•

FIFO Replacement: Solution Text Description

Performance with 4 page frames and 10 requests:

- single letters denote virtual pages
- Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - oldest page is a
- Time 1: request page c, already in memory so no fault
- Time 2: request page a, already in memory so no fault
- Time 3: request page d, already in memory so no fault
- Time 4: request page b, already in memory so no fault
- Time 5: request page e which isn't in memory, causes a page fault
  - puts e where a used to be since a was the oldest page in memory
  - new frame setup: e in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - new oldest page is b
- Time 6: request page b, already in memory so no fault
- Time 7: request page a which isn't in memory, causes a page fault
  - puts a where b used to be since b was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, c in frame 2 and d in frame 3
  - new oldest page is c
- Time 8: request page b which isn't in memory, causes a page fault
  - puts b where c used to be since b was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, b in frame 2 and d in frame 3
  - new oldest page is d
- Time 9: request page c which isn't in memory, causes a page fault
  - puts c where d used to be since d was the oldest page in memory
  - new frame setup: e in frame 0, a in frame 1, b in frame 2 and c in frame 3
  - new oldest page is e
- Time 10: request page d which isn't in memory, causes a page fault
  - puts d where e used to be since e was the oldest page in memory
  - final frame setup: d in frame 0, a in frame 1, b in frame 2 and c in frame 3
- Total number of page faults: 5

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			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*										
	1	*b*										
	2	*c*										
	3	*d*										
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			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	*d*	*d*	*d*	*d*	*d*	***e***	*e*	*e*	*e*	*e*	*e*
Faults							•					•

Time page needed next

- $a = 7$, $b = 6$, $c = 9$, $d = 10$
- $a = 15$, $b = 11$, $c = 13$, $d = 14$

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

Least Recently Used

- Approximation of optimal that 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		*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames 0	*a*										
Page Frames 1	*b*										
Page Frames 2	*c*										
Page Frames 3	*d*										
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			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	***e***	*e*	*e*	*e*	*e*	***d***
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•				•	•
Time page last used						$a = 2$ $b = 4$ $c = 1$ $d = 3$				$a = 7$ $b = 8$ $e = 5$ $d = 3$	$a = 7$ $b = 8$ $e = 5$ $c = 9$	

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

Implementing LRU

Option 1: keep a time stamp for each page representing the last access

Problem: OS must record time stamp for each memory access and search all pages to find one to toss

Option 2: keep a list of pages, where the front of the list is the most recently used and the end is the least recently used. Move page to front on access. Doubly link the list.

Problem: Still too expensive, since OS must modify up to 6 pointers on memory access

iClicker 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

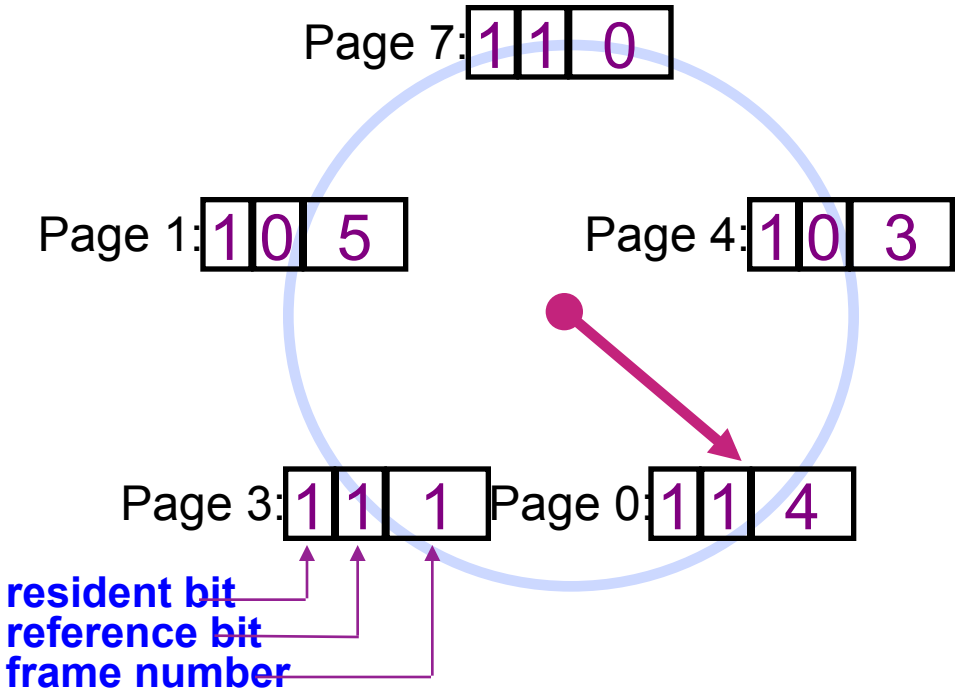
C. Minimize CPU time for an algorithm

Approximating LRU: Clock

- Maintain a circular list of pages in memory
- Maintain pointer (clock hand) to oldest page
- Before replacing a page, check its *reference bit*
  - Remember the reference/clock bit?
  - It tracks whether the page was referenced recently
- If the reference bit is 1 (was referenced recently), clear bit and then check next page
- How does this algorithm terminate?

Clock

- Clock hand points to oldest page
- Clock hand sweeps over pages looking for one with *reference* bit = 0
  - Replace pages that haven't been referenced for one complete revolution of the clock




```
func Clock_Replacement
begin
 while (victim page not found) do
    if(reference bit for cur page = 0)
then
        replace current page
     else
        reset reference bit
      advance clock pointer
  end while
end Clock_Replacement
```

Clock: Text Description

- Pages are arranged in a (logical) circle
- Each page around the clock face keeps track of 3 bits: resident, reference and frame number
- Clock hand points to the oldest page
- Clock hand sweeps over pages looking for one with reference bit that is 0
  - Replace pages that haven't been referenced for one complete revolution of the clock
- Pseudocode for algorithm:
  - func clock_replacement
  - begin
    - while(victim page not found) do
      - if(reference bit for current page is 0) then
        - » replace current page
      - else
        - » reset reference bit
      - advance clock pointer
    - end while
  - end clock_replacement
- Can be implemented with a circular list of all resident pages from the page table
- Is an approximation of the LRU algorithm

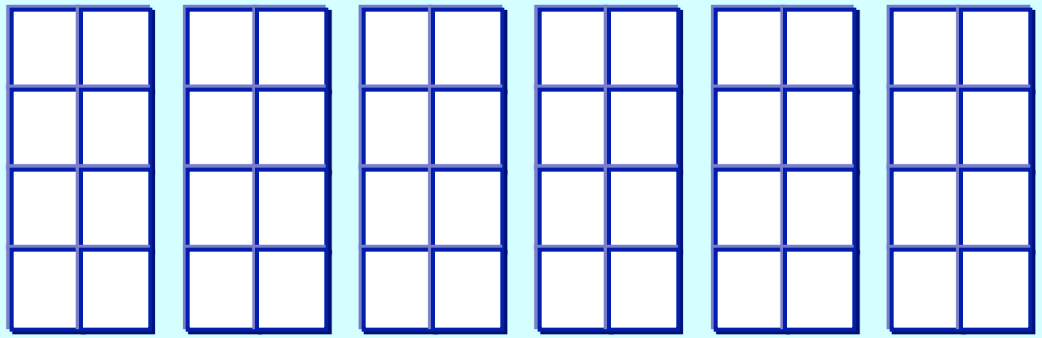
Clock Page Replacement

Example

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*						
	1	*b*	*b*	*b*	*b*	*b*						
	2	*c*	*c*	*c*	*c*	*c*						
	3	*d*	*d*	*d*	*d*	*d*						
Faults												

Page table entries for resident pages:

1	*a*
1	*b*
1	*c*
1	*d*



Clock Page Replacement: Example Text Description

- Same setup as earlier examples
- 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.
- In earlier examples, we learned that no page replacements were necessary until time 5, so this example starts at time 5.

Clock Page Replacement

Example: Solution

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*a*	*d*	*b*	*e*	*b*	*a*	*b*	*c*	*d*
Page Frames	0	*a*	*a*	*a*	*a*	*a*	***e***	*e*	*e*	*e*	*e*	***d***
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	***a***	*a*	*a*	*a*
	3	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*	***c***	*c*
Faults							•		•		•	•

Page table entries for resident pages:

1 *a*	1 *e*	1 *e*	1 *e*	1 *e*	1 *e*	1 *e*	1 *d*		
1 *b*	0 *b*	1 *b*	0 *b*	1 *b*	1 *b*	1 *b*	0 *b*		
1 *c*	0 *c*	0 *c*	1 *a*	1 *a*	1 *a*	1 *a*	0 *a*		
1 *d*	0 *d*	0 *d*	0 *d*	0 *d*	1 *c*	0 *c*			

Clock Page Replacement Example Solution: Text Description (1 of 2)

- Performance with 4 frames and 10 requests:
  - Time 0: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
    - clock pointer at frame 0
  - Time 1: request page c, in memory so no page fault
    - update reference bit to be 1
  - Time 2: request page a, in memory so no page fault
    - update reference bit to be 1
  - Time 3: request page b, in memory so no page fault
    - update reference bit to be 1
  - Time 4: request page d, in memory so no page fault
    - update reference bit to be 1
  - Time 5: request page e which isn't in memory, causes a page fault
    - have to run the clock algorithm
      - frame 0 (a) has reference bit of one, clear it and move pointer to frame 1
      - frame 1 (b) has reference bit of one, clear it and move pointer to frame 2
      - frame 2 (c) has reference bit of one, clear it and move pointer to frame 3
      - frame 3 (d) has reference bit of one, clear it and move pointer back to frame 0
      - since we cleared all the bits as we went frame 0 now has a reference bit of zero, this is the page to replace
      - put e in frame 0 and move pointer to frame 1
    - current frame setup: e in frame 0, b in frame 1, c in frame 2 and d in frame 3

Clock Page Replacement Example Solution: Text Description (2 of 2)

- Time 6: request page b which is in memory, no page fault
  - frame 1's reference bit is set to one since it holds page b
- Time 7: request page a which isn't in memory, causes a page fault
  - have to run the clock algorithm
    - frame 1 (b) has reference bit of one, clear it and move pointer to frame 2
    - frame 2 (c) has reference bit of zero, this is the page to replace
    - put a in frame 2 and move pointer to frame 3
  - current frame setup: e in frame 0, b in frame 1, a in frame 2 and d in frame 3
- Time 8: request page b, in memory so no page fault
  - update reference bit to be 1
- Time 9: request page c which isn't in memory, causes a page fault
  - have to run clock algorithm
    - frame 3 (d) has reference bit of zero, this is the page to replace
    - put d in frame 3 and move pointer to frame 0
  - current frame setup: e in frame 0, b in frame 1, a in frame 2 and c in frame 3
- Time 10: request page d which isn't in memory, causes a page fault
  - have to run clock algorithm
    - frame 0 (e) has reference bit of one, clear it and move to frame 1
    - frame 1 (b) has reference bit of one, clear it and move to frame 2
    - frame 2 (a) has reference bit of one, clear it and move to frame 3
    - frame 3 (c) has reference bit of one, clear it and move to frame 0
    - since we cleared all the bits as we went frame 0 now has a reference bit of zero, this is the page to replace
    - put d in frame 0 and move pointer to frame 1
  - final memory setup: d in frame 0, b in frame 1, a in frame 2 and c in frame 3
- total number of page faults: 4

Second Chance

- Cheaper to replace a page that has not been written since it need not be written back to disk
- Check both the *reference bit* and *modify bit* to determine which page to replace
  - (reference, modify) pairs form classes:
    - (0,0): not used or modified, replace!
    - (0,1): not recently used but modified: OS needs to write, but may not be needed anymore
    - (1,0): recently used and unmodified: may be needed again soon, but doesn't need to be written
    - (1,1): recently used and modified
- On page fault, OS searches for page in the lowest nonempty class

Second Chance Implementation

The OS goes around at most three times searching for the (0,0) class:

1. If the OS finds (0,0) it replaces that page
2. If the OS finds (0,1) it
   - initiates an I/O to write that page,
   - locks the page in memory until the I/O completes,
   - clears the modified bit, and
   - continues the search in parallel with the I/O
3. For pages with the reference bit set, the reference bit is cleared
4. On second pass (no page (0,0) found on first), pages that were (0,1) or (1,0) may have changed

Second Chance Implementation (another option)

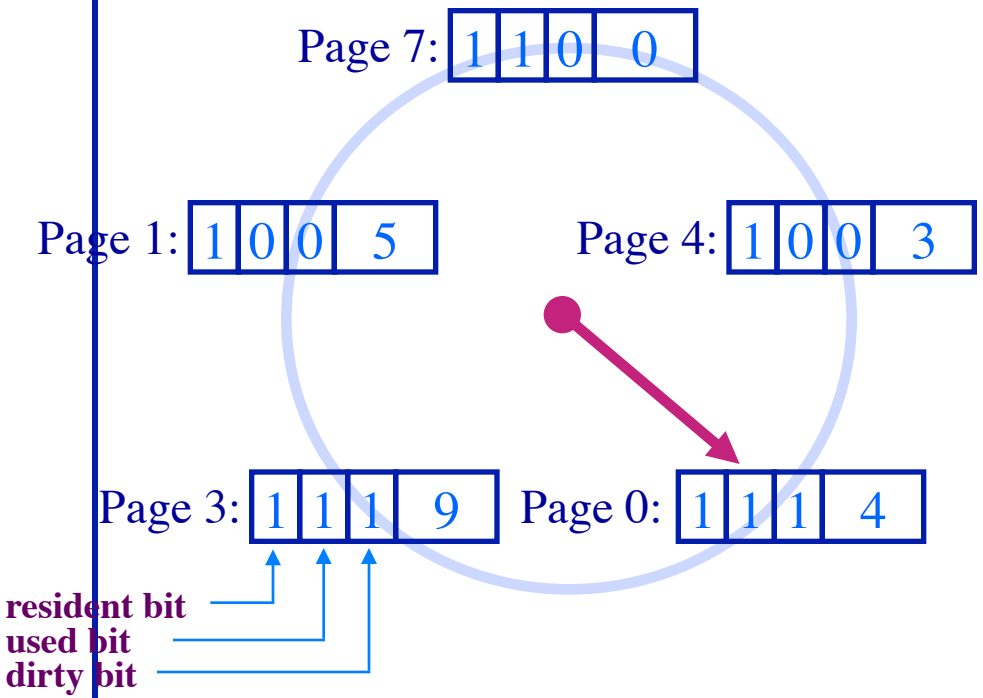
The OS goes around at most three times searching for the (0,0) class:

1. If the OS finds (0,0) it replaces that page
2. If the OS finds (0,1), *clear dirty bit and move on, but remember page is dirty. Write only if evicted.*
3. For pages with the reference bit set, the reference bit is cleared
4. On second pass (no page (0,0) found on first), pages that were (0,1) or (1,0) may have changed

Optimizing Approximate LRU Replacement

The Second Chance Algorithm

- There is a significant cost to replacing “dirty” pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the *dirty bit* and the *used bit* to drive replacement



Second Chance Algorithm

Before clock sweep		After clock sweep	
used	*dirty*	*used*	*dirty*
		replace page	
0	0	0	0
0	1	0	0
1	0	0	0
1	1	0	1

The Second Chance Algorithm: Text Description

- There is a significant cost to replacing “dirty” pages
- Modify the clock algorithm to allow dirty pages to always survive one sweep of the clock hand
  - Use both the dirty bit and the used bit to drive replacement
- When a page is referenced the reference bit is set
  - If the page was modified, aka written to, the dirty bit is set
- During first “sweep” of the hand the used bit is cleared
- During second “sweep” of the hand the dirty bit is cleared
  - Assumption: OS remembers that the pages is really dirty
- Only replace pages that have both dirty and used bit equal to zero
  - So if a page is dirty have to wait full 2 sweeps for it to be replaced

Local vs. Global Page Replacement

- *Local* page replacement algorithms only consider the pages owned by the faulting process
  - Essentially a fixed number of pages per process
- *Global* page replacement algorithms consider all the pages

The Problem With Local Page Replacement

How much memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames	
0	*a*
1	*b*
2	*c*

Faults

Page Frames	
0	*a*
1	*b*
2	*c*
3	–

Faults

The Problem With Local Page Replacement: Text Description

- Two examples, one slide
- For both:
  - Time and requests on x axis
  - Page frames on the y axis
  - Keep track of faults as you go
  - The request string is a, b, c, d, a, b, c, d, a, b, c, d.
  - Each request occurs at another time tick, so requests are at times 1-12
- Example 1:
  - 3 page frames
  - At time 0, pages a, b, and c are loaded into frames 0, 1, and 2 respectively.
- Example 2:
  - 4 page frames
  - At time 0, pages a, b, and c are loaded into frames 0, 1, and 2 respectively. Frame 3 is empty.

The Problem With Local Page Replacement

How much memory do we allocate to a process? (Solution)

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*	*a*	*b*	*c*	*d*

Page Frames													
	0	*a*	*a*	*a*	*a*	***d***	*d*	*d*	***c***	*c*	*c*	***b***	*b*	*b*
	1	*b*	*b*	*b*	*b*	*b*	***a***	*a*	*a*	***d***	*d*	*d*	***c***	*c*
	2	*c*	*c*	*c*	*c*	*c*	*c*	***b***	*b*	*b*	***a***	*a*	*a*	***d***
Faults						•	•	•	•	•	•	•	•	•

Page Frames													
	0	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*	*a*
	1	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*	*b*
	2	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*	*c*
	3	–				***d***	*d*	*d*	*d*	*d*	*d*	*d*	*d*	*d*
Faults						•								

The Problem With Local Page Replacement: Solution Text Description (1 of 2)

- How much memory do we allocate to a process?
- Example with 3 page frames and 4 pages with FIFO replacement:
  - reference order: a b c d a b c d a b c d
  - Time 0: a in frame 0, b in frame 1 and c in frame 2
  - Time 1: request page a, in memory so no page fault
  - Time 2: request page b, in memory so no page fault
  - Time 3: request page c, in memory so no page fault
  - Time 4: request page d which isn't in memory, causes a page fault
    - put d where a used to be since a was the oldest page in memory
    - current frame setup: d in frame 0, b in frame 1 and c in frame 2
  - Time 5: request page a which isn't in memory, causes a page fault
    - put a where b used to be since b was the oldest page in memory
    - current frame setup: d in frame 0, a in frame 1 and c in frame 2
  - Time 6: request page b which isn't in memory, causes a page fault
    - put b where c used to be since c was the oldest page in memory
    - current frame setup: d in frame 0, a in frame 1 and b in frame 2
  - Time 7: request page c which isn't in memory, causes a page fault
    - put c where d used to be since d was the oldest page in memory
    - current frame setup: c in frame 0, a in frame 1 and b in frame 2
  - Time 8: request page d which isn't in memory, causes a page fault
    - put d where a used to be since a is the oldest page in memory
    - current frame setup: c in frame 0, d in frame 1 and b in frame 2

The Problem With Local Page Replacement: Solution Text Description (2 of 2)

  - Time 9: request page a which isn't in memory, causes a page fault
    - put a where b used to be since b was the oldest page in memory
    - current frame setup: c in frame, d in frame 1 and a in frame 2
  - Time 10: request page b which isn't in memory, causes a page fault
    - put b where c used to be since c was the oldest page
    - current frame setup: b in frame, d in frame 1 and a in frame 2
  - Time 11: request page c which isn't in memory, causes a page fault
    - put c where d used to be since d was the oldest page
    - current frame setup: b in frame, c in frame 1 and a in frame 2
  - Time 12: request page d which isn't in memory, causes a page fault
    - put d where a used to be since a was the oldest page
    - final frame setup: b in frame, c in frame 1 and d in frame 2
  - Total number of page faults: 9
    - page faulted on almost every request
- Example with 4 page frames and 4 frames and FIFO replacement
  - reference order: a b c d a b c d a b c d
  - Time 0: a in frame 0, b in frame 1, c in frame 2 and frame 3 is empty
  - Time 1 - 3: request pages a b and c which are already in memory
  - Time 4: request page d which isn't in memory, causes a page fault
    - put d in the empty frame
    - current frame setup: a in frame 0, b in frame 1, c in frame 2 and d in frame 3
  - now all pages are resident in memory so the rest of the requests don't cause any page faults
  - Total number of page faults: 1
- These examples show that allocating too few frames to a process can cause a massive jump in the number of page faults

So… how much memory should we allocate to each process?

A. 4 frames
B. 8 frames
C. 10% of the frames
D. 50% of the frames
E. It depends.

Do we really want to decide this ahead of time? Do we really want a fixed number per process? Does that make sense?

Is there another way?

The Principle of Locality

Recall: programs exhibit *temporal* and *spatial* locality

– 90% of execution is sequential

– Most iterative constructs consist of a relatively small number of instructions

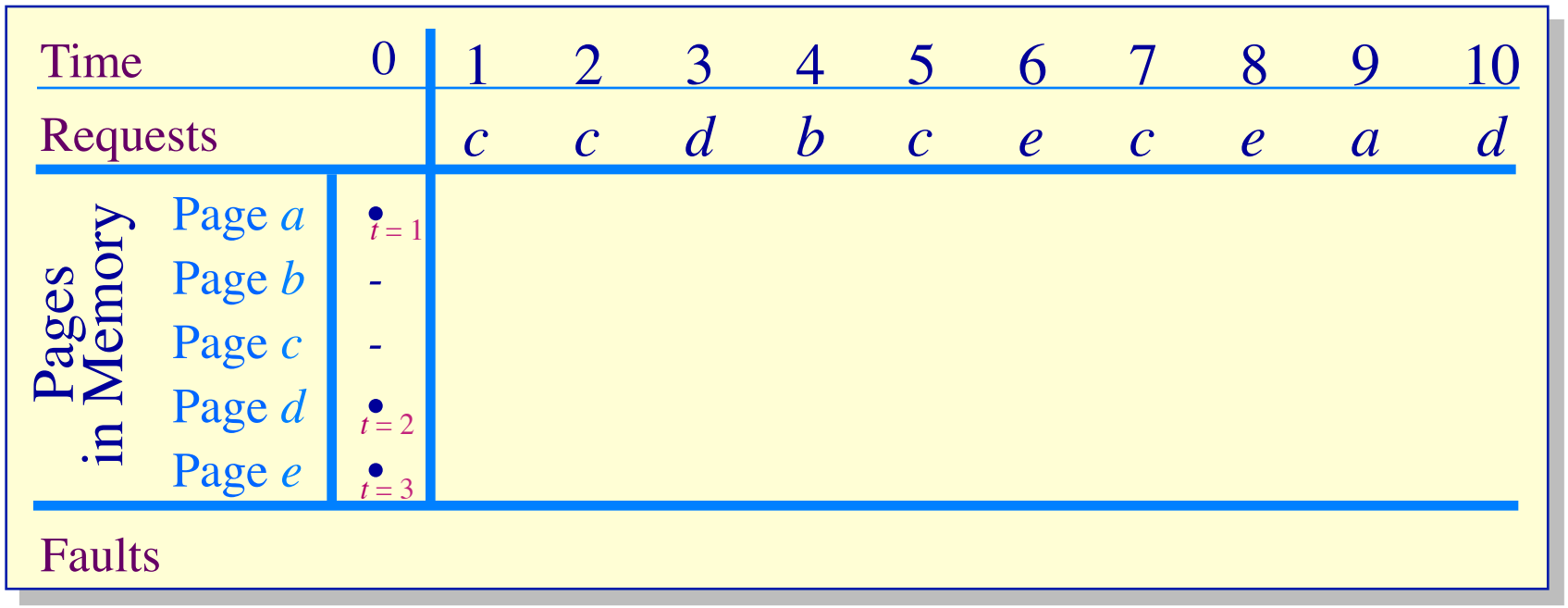
– When processing large data structures, the dominant cost is sequential processing on individual structure elements

Explicitly Using Locality: The Working Set Model

- The *working set* is:
  - informally, the pages the process is using right now
  - formally, the set of all pages that a process referenced in the last T seconds
- Assume that recently references pages are likely to be referenced again soon
- Only keep those pages in memory (the working set!)
  - Pages may be removed even when no page fault occurs
  - Number of frames allocated to a process will vary over time
- Also allows *pre-paging.*

Working Set Page Replacement
Implementation

- Keep track of the last T references
  - The pages referenced during the last T memory accesses are the working set
  - T is called the *window size*
- Example: Working set computation, $T = 4$ references:



Working Set Page Replacement: Text Description

- Implementation of working set
- keep track of the last T references
  - the pages referenced during the last T memory accesses are the working set
  - T is called the window size
- Example: working set computation with T = 4 references
  - Time and requests on x axis
  - Pages (not frames) on the y axis
  - Are keeping track of whether or not each page is in memory
    - instead of keeping track of what each frame is holding
    - also keep track of the “age” of each page in memory
      - when a page becomes 5 ticks old it is kicked out of memory
  - Keep track of faults as you go
  - Time 0: pages a, d and e in memory
    - a is 1 tick old, d is 2 ticks old and e is 3 ticks old
  - The request string is c, c, d, b, c, e, c, e, a, d.
  - Each request occurs at another time tick, so requests are at times 1-10.

Working Set Page Replacement
Implementation

- Keep track of the last T references
  - The pages referenced during the last T memory accesses are the working set
  - T is called the *window size*
- Example: Working set computation, T = 4 references:
  - What if T is too small? too large?

Time		0	1	2	3	4	5	6	7	8	9	10
Requests			*c*	*c*	*d*	*b*	*c*	*e*	*c*	*e*	*a*	*d*
Pages in Memory	Page *a*	• $t=1$	•	•	•	-	-	-	-	-	F	•
	Page *b*	-	-	-	-	F	•	•	•	-	-	-
	Page *c*	-	F	•	•	•	•	•	•	•	•	•
	Page *d*	• $t=2$	•	•	•	•	•	•	-	-	-	F
	Page *e*	• $t=3$	•	-	-	-	-	F	•	•	•	•
Faults			•			•		•			•	•

Working Set Page Replacement: Solution Text Description (1 of 2)

- Example: working set computation with $\tau = 4$ references
  - are keeping track of whether or not each page is in memory
    - also keep track of the "age" of each page in memory
      - when a page becomes 5 ticks old it is kicked out of memory
  - Time 0: pages a, d and e in memory
    - a is 1 tick old, d is 2 ticks old and e is 3 ticks old
  - Time 1: page c is requested which isn't in memory, causes a page fault
    - pages in memory: a, c, d and e
  - Time 2: page c is requested, is in memory so no page fault
  - Time 3: page d is requested, is in memory so no page fault
    - at this time page e becomes 5 ticks old so is kicked out of memory
    - pages in memory: a, c and d
  - Time 4: page b is requested which isn't in memory, causes a page fault

Working Set Page Replacement: Solution
Text Description (2 of 2)

- Time 5: page c is requested, is in memory so no page fault
- Time 6: page e is requested which isn't in memory, causes a page fault
  - pages in memory: b c d and e
- Time 7: page c is requested, is in memory so no page fault
  - at this time page d becomes 5 ticks old so is kicked out of memory
  - pages in memory: b c and e
- Time 8: page e is requested, is in memory os no page fault
  - at this time page b becomes 5 ticks old so is kicked out of memory
  - pages in memory: c and e
- Time 9: page a is requested which isn't in memory, causes a page fault
  - pages in memory: a c and e
- Time 10: page d is requested which isn't in memory, causes a page fault
  - pages in memory: a c d and e
- Total number of page faults: 5

How do we choose the value of T?

1 page fault = 10ms

10ms = 2M instructions

T needs to be a lot bigger than 2 million instructions

What if T is too big?

What if T is too small?

Thrashing

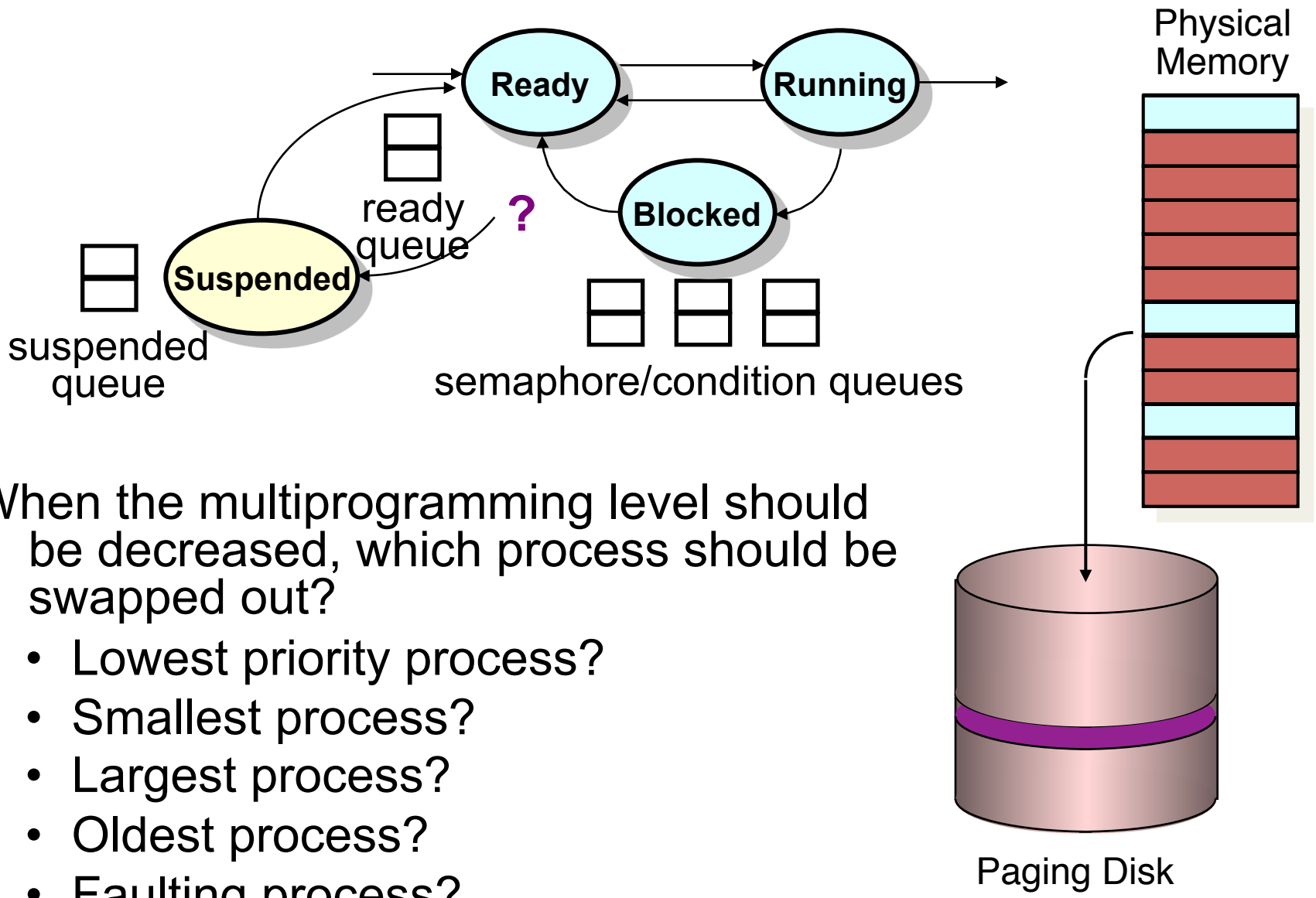
- *Thrashing* occurs when the memory is over-committed and pages are tossed out while they are still in use
- Many memory references cause pages to be faulted in
  - Very serious and very noticeable loss of performance

How do we limit thrashing in a multiprogrammed system?

Load Control

- *Load control* refers to the number of processes that can reside in memory at one time
- Working set model provides implicit load control by only allowing a process to execute if its working set fits in memory
- BUT process frame allocations are variable
- What happens when the total number of pages needed is greater than the number of frames available?
  - Processes are swapped out to disk

Load Control



When the multiprogramming level should be decreased, which process should be swapped out?

- Lowest priority process?
- Smallest process?
- Largest process?
- Oldest process?
- Faulting process?

Load Control: Text Description

- When a process is totally swapped out of memory it is put onto swap (aka the paging disk)
- Adds another stage to the process life cycle
  - This new stage is called suspended
    - We also saw this stage in relocation, when we also swapped out entire processes
  - Can go from any of the other states to suspended
    - Usually blocked or ready
  - Process can go from suspended to ready

Another Decision: Page Sizes

Page sizes are growing slowly but steadily. Why?

- Benefits for small pages: more effective memory use, higher degree of multiprogramming possible
- Benefits for large pages: smaller page tables, reduced I/O time, fewer page faults
- Growing because:
  - memory is cheap---page tables could get huge with small pages and internal fragmentation is less of a concern
  - CPU speed is increasing faster than disk speed, so page faults cause a larger slow down

iClicker Question

Can an application modify its own translation tables (however they are implemented)?

A. Yes

B. No

Summary: Page Replacement Algorithms

- Unix and Linux use a variant of the second chance algorithm
- Windows NT uses FIFO replacement
- All algorithms do poorly on typical processes if processes have insufficient physical memory
- All algorithms approach optimal as the physical memory allocated to a process approaches virtual memory size
- The more processes running concurrently, the less physical memory each process can have

Summary: Paging

We've considered:

- Placement Strategies
  - None needed, can place pages anywhere
- Replacement Strategies
  - What to do when more jobs exist than can fit in memory
- Load Control Strategies
  - Determine how many jobs can be in memory at one time

Summary: Paging

The Good

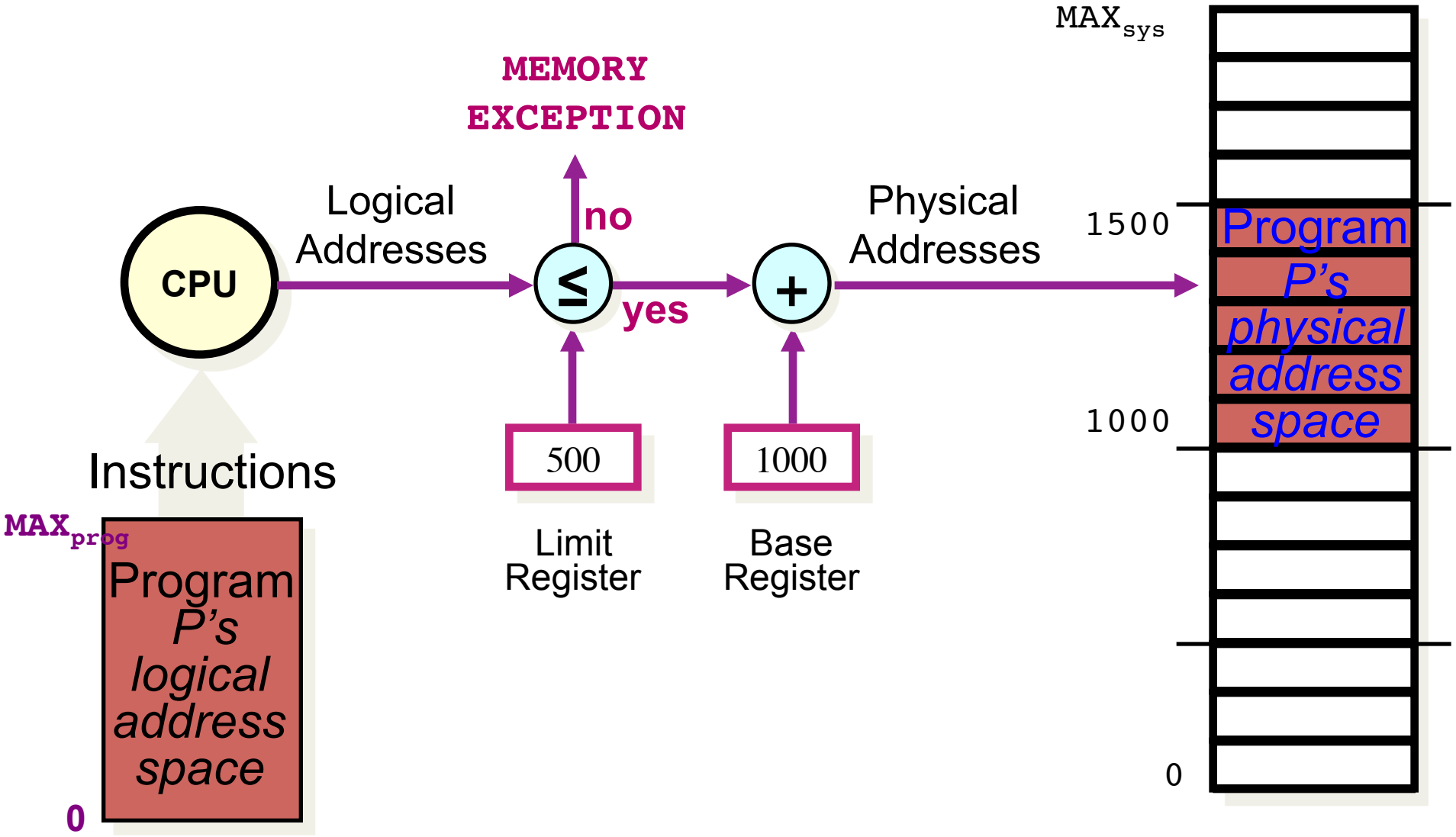
- Eliminates the problem of external fragmentation
- Allows sharing of memory pages amongst processes
- Enables processes to run when they are only partially loaded into main memory

The Cost

- Translating from a virtual address to a physical address is time consuming
- Requires hardware support (TLB) to be decently efficient
- Requires more complex OS to maintain the page table

The expense of memory accesses and the flexibility of paging make paging cost effective.

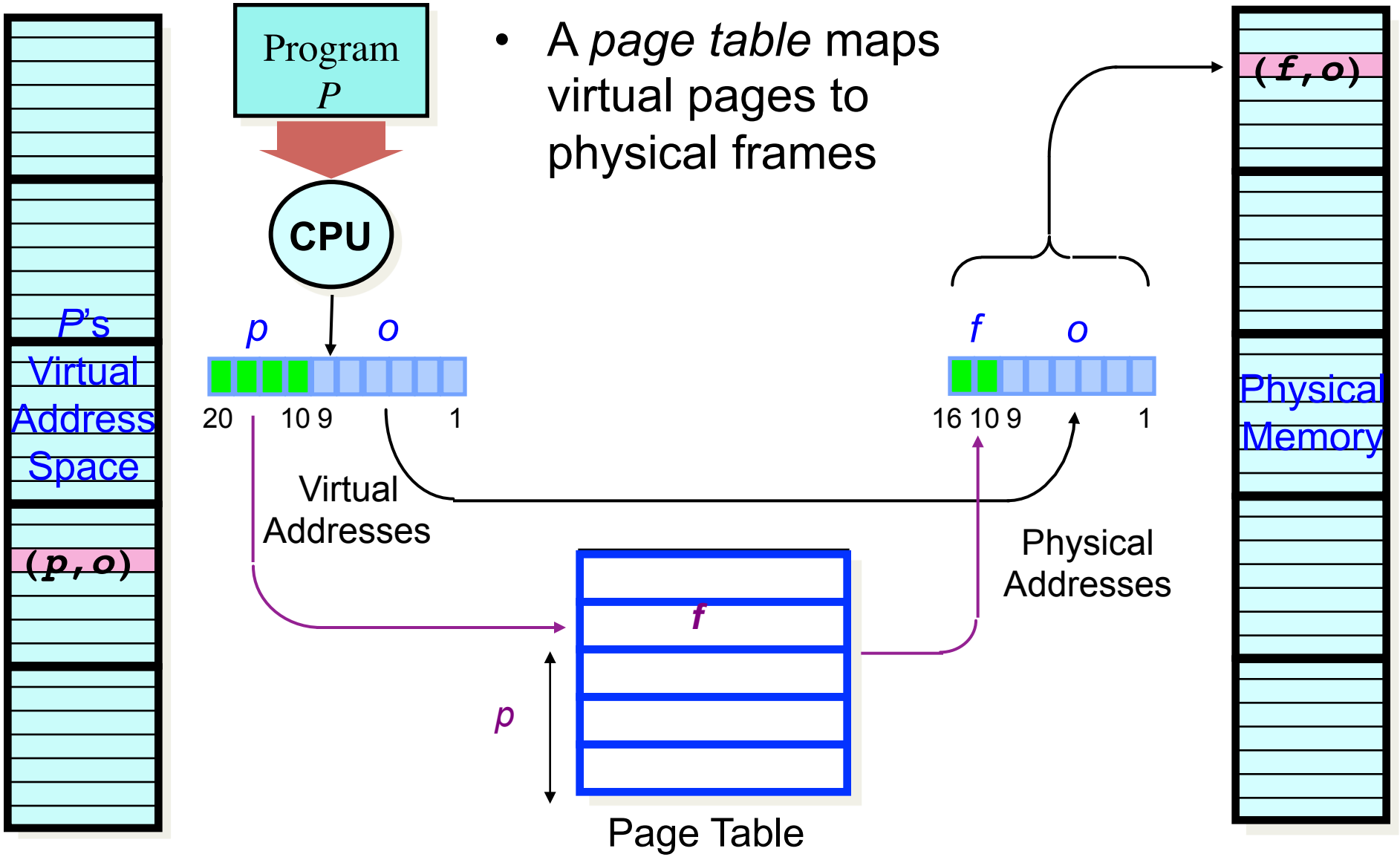
Dynamic Relocation: Address Translation



Dynamic Relocation: Text Description

- How we get from a program's logical address space to physical memory.
- The bound register is used to make sure that the program isn't trying to access memory outside of its space. For example, if the programs logical address space ranges from 0 to 500 then the bound register would hold a value of 500.
- The base register holds the beginning of that program's addresses in physical memory. So if the programs memory started at address 1000 then the base register would hold 1000.
- Steps to Address Translation:
  - The program gives a logical address, or instruction, to the CPU.
  - The MMU now does 2 things at once:
    - It checks the address against the bound register, if the address is greater than the bound register a memory exception is thrown. This exception is a hardware interrupt that will be handled by the OS. This exception indicates the program was trying to access something that doesn't belong to it.
    - It adds the base register to the logical address to get the physical address. For example logical address 8 would become 1008 if the base register was 1000.
- NOTE: Even when the address is successfully translated, you are not protected from memory errors: you could still accidentally try to access memory in your address space that hasn't be initialized yet.

Virtual Address Translation



- A *page table* maps virtual pages to physical frames

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

Memory Management: Putting it all Together

- Dynamic Relocation with Base and Bounds:
  - Simple, but inflexible
  - Degree of multiprogramming limited, memory limited to physical memory size, no sharing of memory, memory allocation/deallocation difficult
  - Use compaction to solve external fragmentation
- Paging:
  - Process generates virtual addresses from 0 to Max
  - OS divides processes into pages
    - manages a page table for every process
    - manages the pages in memory
  - Simplifies memory allocation since any page can be allocated to any frame
  - Page tables can be very large
  - Page Replacement Algorithms
    - FIFO, Optimal, LRU, Clock, Enhanced Clock, Working Set
  - Design Considerations (page size, global vs. local, …)

Announcements

- Homework 6 posted due Friday 8:45a
- Stack Check appointments today and tomorrow
  - You should have received email at your CS account telling you who to meet
    - If you didn't, check that you didn't give me your EID
  - If you haven't signed up, see me
- Project 2 due Friday after Spring Break, 3/27
  - You MUST get it working