

Page Replacement Algorithms: Clock

Bringing It Together

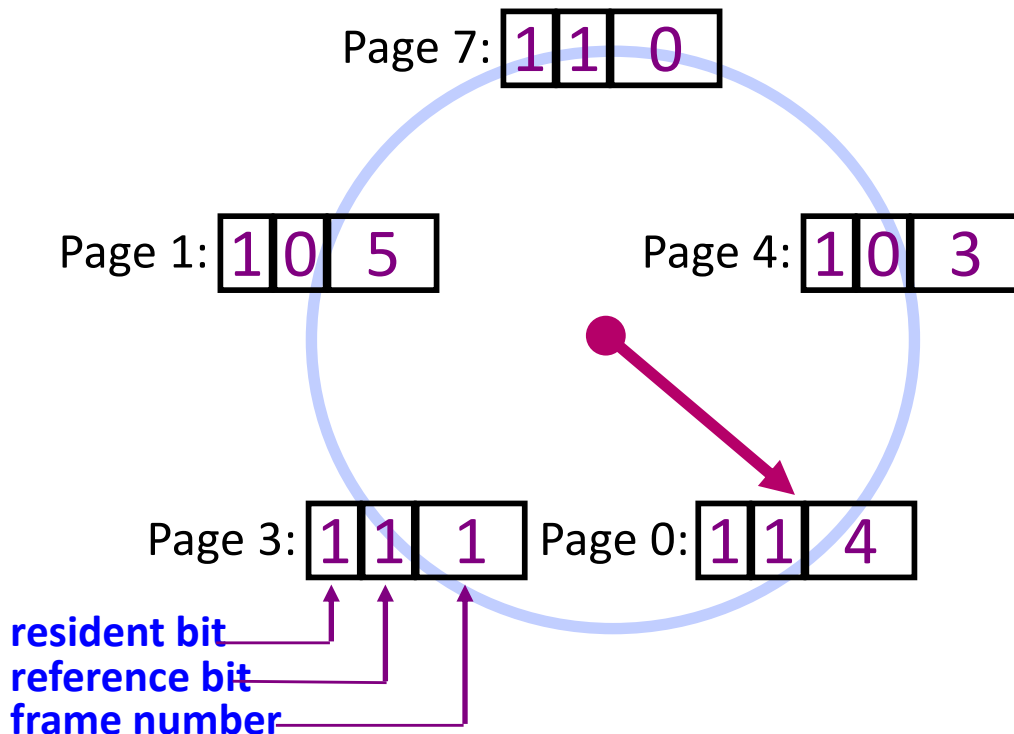
- Processes are the OS's main abstraction for protection
- A process defines an address space---but this is a virtual/logical address space
- Virtual address spaces are mapped onto physical address spaces in physical memory
- All virtual addresses range from 0->max for that process
- *Paging* describes how the OS provides the illusion of infinite memory
 - Page tables aid address translation and data finding
 - TLB improves performance in time
 - Multi-level page table improves performance in space

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		<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>						
1	<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>						
3	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>						
Faults											

Page table entries
for resident pages:

1	<i>a</i>
1	<i>b</i>
1	<i>c</i>
1	<i>d</i>



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		<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>e</i>	<i>e</i>	<i>e</i>	<i>e</i>	<i>e</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>a</i>	<i>a</i>	<i>a</i>	<i>a</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>d</i>	<i>c</i>	<i>c</i>
Faults						•		•		•	•

Page table entries
for resident pages:

1	<i>a</i>
1	<i>b</i>
1	<i>c</i>
1	<i>d</i>

1	<i>e</i>
0	<i>b</i>
0	<i>c</i>
0	<i>d</i>

1	<i>e</i>
1	<i>b</i>
0	<i>c</i>
0	<i>d</i>

1	<i>e</i>
0	<i>b</i>
1	<i>a</i>
0	<i>d</i>

1	<i>e</i>
1	<i>b</i>
1	<i>a</i>
0	<i>d</i>

1	<i>e</i>
1	<i>b</i>
1	<i>a</i>
1	<i>c</i>

1	<i>d</i>
0	<i>b</i>
0	<i>a</i>
0	<i>c</i>

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

Page Replacement Algorithms: Enhanced Clock

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)

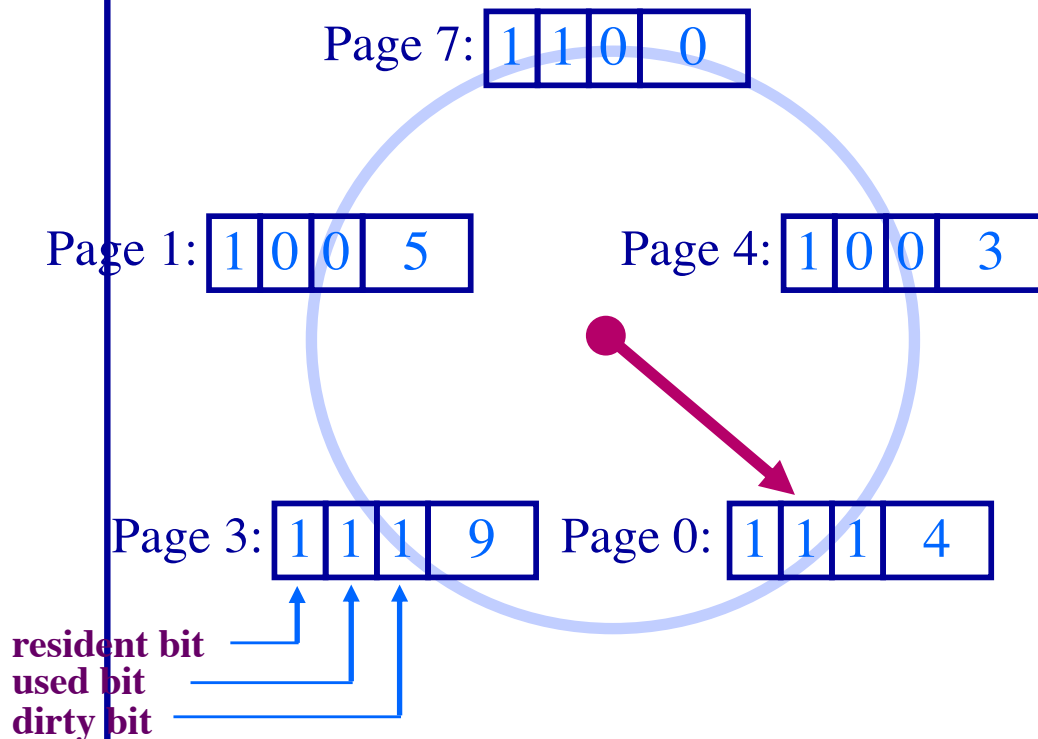
The OS goes around at most three times searching for the (0,0) class:

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

<i>used</i>	<i>dirty</i>
0	0
0	1
1	0
1	1

After clock sweep

<i>used</i>	<i>dirty</i>
<i>replace page</i>	
0	0
0	0
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

Page Replacement Algorithms: Local vs. Global

How much physical memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</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>	

Faults

Page Frames	0	<i>a</i>	
	1	<i>b</i>	
	2	<i>c</i>	
	3	—	

Faults

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.

How much physical memory do we allocate to a process?

Time	0	1	2	3	4	5	6	7	8	9	10	11	12
Requests		<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</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>d</i>	<i>d</i>	<i>d</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>b</i>	<i>b</i>	<i>b</i>
	1	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>c</i>	<i>c</i>
	2	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>c</i>	<i>b</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>a</i>	<i>d</i>
	Faults					•	•	•	•	•	•	•	•	•

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>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>	<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>	<i>c</i>	<i>c</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>d</i>
	Faults					•								

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

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 physical 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?

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 Working Set

Could We Better Leverage 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 Eviction 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:

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 Process	Page a	• $t=1$										
	Page b	-										
	Page c	-										
	Page d	• $t=2$										
	Page e	• $t=3$										
Faults												



Working Set Page Eviction:

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 Eviction 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 Process	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 Eviction: Solution

Text Description (1 of 2)

- Example: working set computation with $T = 4$ 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
 - at this time page a becomes 5 ticks old so is kicked out of memory
 - pages in memory: b c and d

Working Set Page Eviction: 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 so 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 ?

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?

How do we choose the value of T?

What if T is too big?

What if T is too small?

1 page fault = 10ms

10ms = 2M instructions

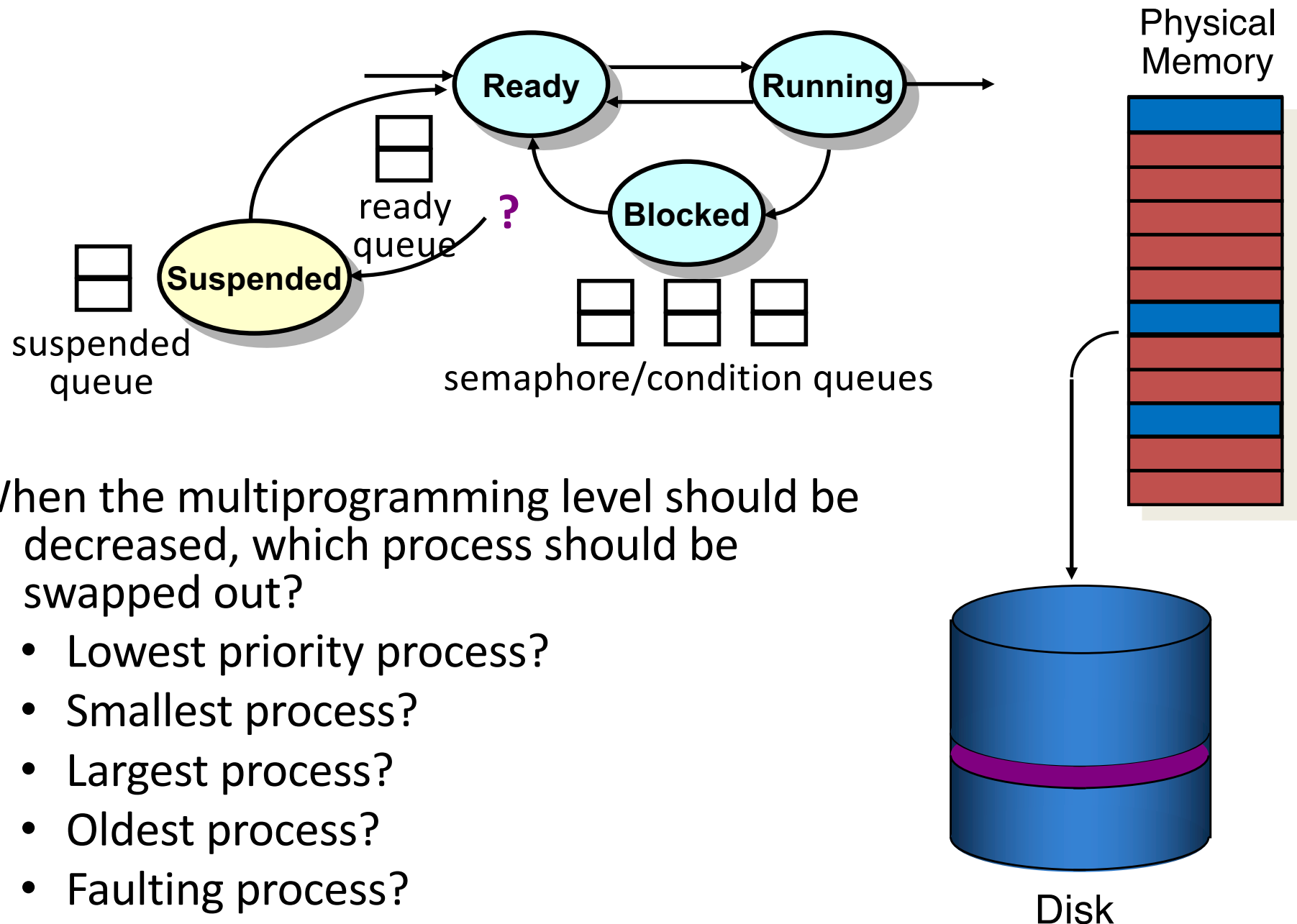
T needs to be a lot bigger than 2 million instructions

Load Control

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



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

Page Sizes

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

Question

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

A. Yes

B. No