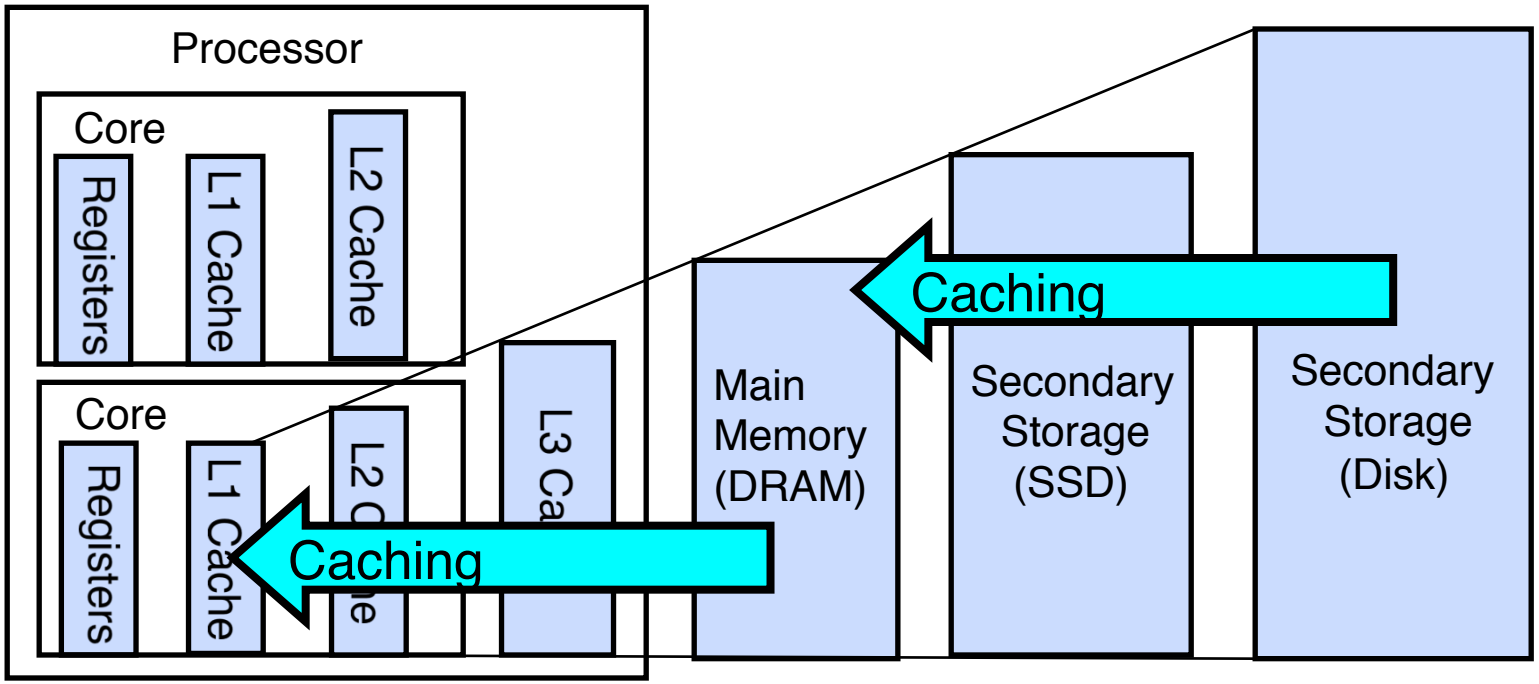
VIRTUAL MEMORY

READING: CHAPTER 9

MEMORY HIERARCHY



L1 cache exclusive to a single core

L2 slower access than L1

L3 shared among multiple cores

ON-DEMAND PAGING

Most processes terminate without having accessed their whole address space

- Code handling rare error conditions, . . .

Other processes go to multiple phases during which they access different parts of their address space

- Compilers

Wasteful to keep the entire address space of a process in memory the whole time

- Use 2nd storage: disk swap space

ON-DEMAND PAGING

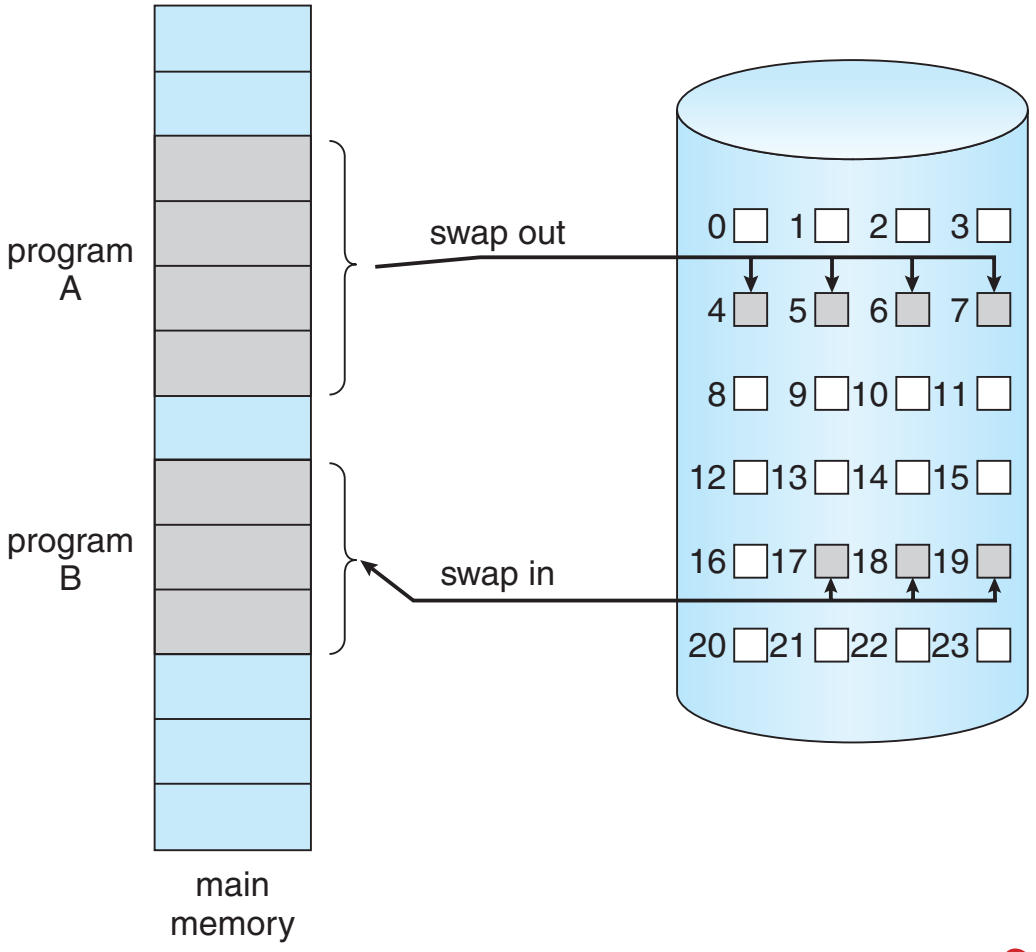
VM systems fetch individual pages on demand when they get accessed the first time

- **Page miss or page fault**

When memory is full, they expel from memory pages that are not currently in use

Advantages:

- **Not limited by the physical memory size**
- **More efficient use of physical memory**



KEY QUESTIONS

Memory access: ~n sec, disk access: ~m sec

Whether a page is in memory? (address translation)

Which pages to be put in memory? Which pages to be swapped out? (page replacement)

What happens during context switches? (write-out)

How to avoid thrashing:

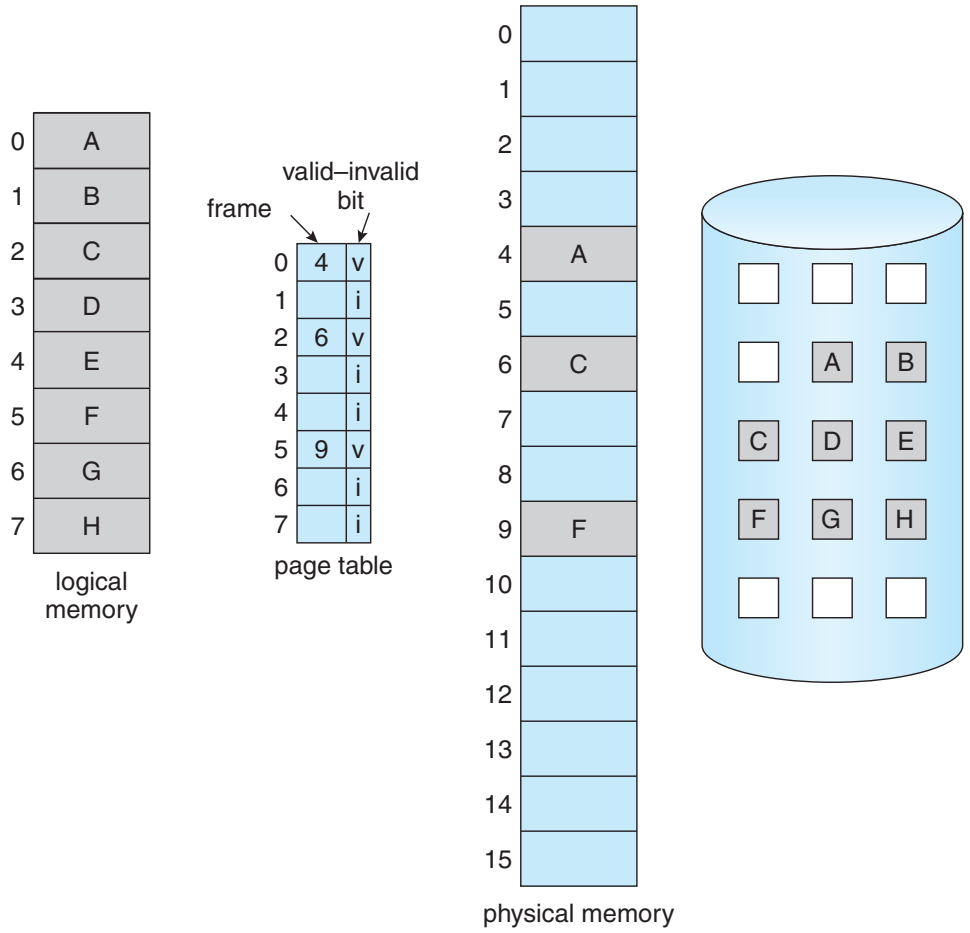
- A computer's virtual memory subsystem is in a constant state of paging, rapidly exchanging data in memory for data on disk, to the exclusion of most application-level processing

VALID BIT

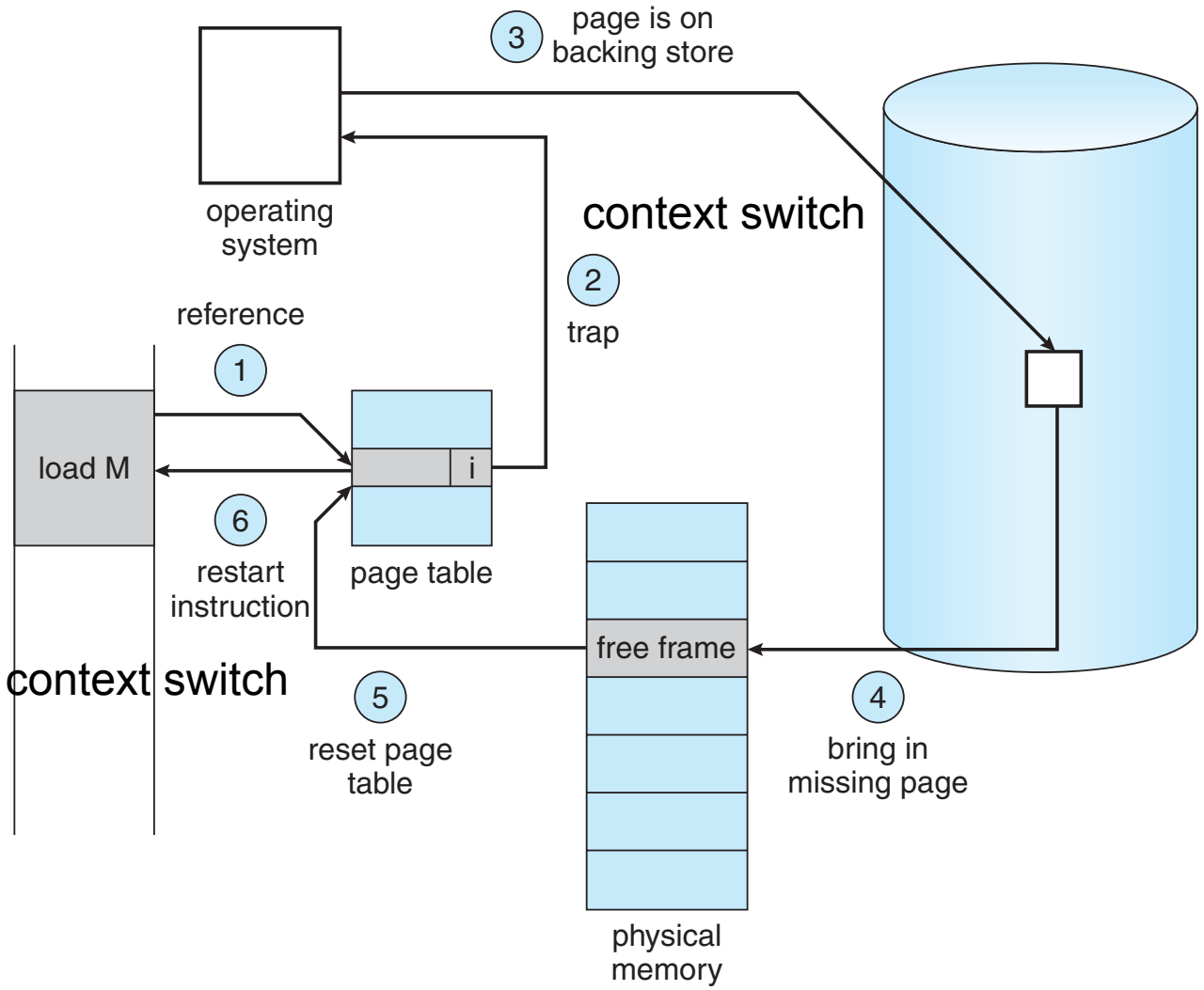
Recall special bits in page table entry

Valid bit indicates whether a page is valid and/or in memory

A page fault occurs if invalid



STEPS IN HANDLING A PAGE FAULT



WHAT HAPPENS DURING PAGE FAULTS

1. **Trap to the operating system.**
2. **Save the user registers and process state.**
3. **Determine that the interrupt was a page fault.**
4. **Check that the page reference was legal and determine the location of the page on the disk.**
5. **Issue a read from the disk to a free frame:**
   - Wait in a queue for this device until the read request is serviced
   - Wait for the devi
   - Begin the transfe
6. **While waiting, a** **optional).**
7. **Receive an interrupt from the disk I/O subsystem (I/O completed).**
8. **Save the registers and process state for the other process (if step 6 is executed).**
9. **Determine that the interrupt was from the disk.**
10. **Correct the page table and other tables to show that the desired page is now in memory.**
11. **Wait for the CPU to be allocated to this process again.**
12. **Restore the user registers, process state, and new page table, and then resume the interrupted instruction.**

Page fault is expensive!

EFFECTIVE ACCESS TIME (EAT)

EAT = Hit Rate x Hit Time + Miss Rate x Miss Time

Example:

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- Suppose p = Probability of miss, 1-p = Probably of hit
- Then, we can compute EAT as follows:

EAT = (1 – p) x 200ns + p x 8 ms

= (1 – p) x 200ns + p x 8,000,000ns

= 200ns + p x 7,999,800ns

If one access out of 1,000 causes a page fault, then EAT = 8.2 µs:

- This is a slowdown by a factor of 40!

What if want slowdown by less than 10%?

- EAT < 200ns x 1.1 ➔ p < 2.5 x 10-6
- This is about 1 page fault in 400,000 !

WHAT LEADS TO PAGE FAULT?

Capacity Misses:

- Not enough memory. Must somehow increase size.
- Can we do this?
  - Increase amount of DRAM (not quick fix!)
  - Reduce the needs for physical memory (copy-on-write)
  - Increase percentage of memory allocated to each one

Compulsory Misses:

- Pages that have never been paged into memory before
- How might we remove these misses?
  - Prefetching: loading them into memory before needed
  - Need to predict future somehow!.

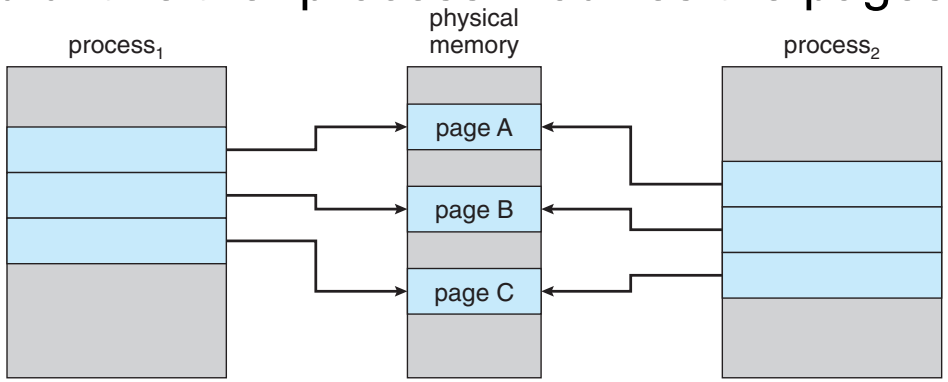
Policy Misses:

- Caused when pages were in memory, but kicked out prematurely because of the replacement policy
- How to fix? Better replacement policy

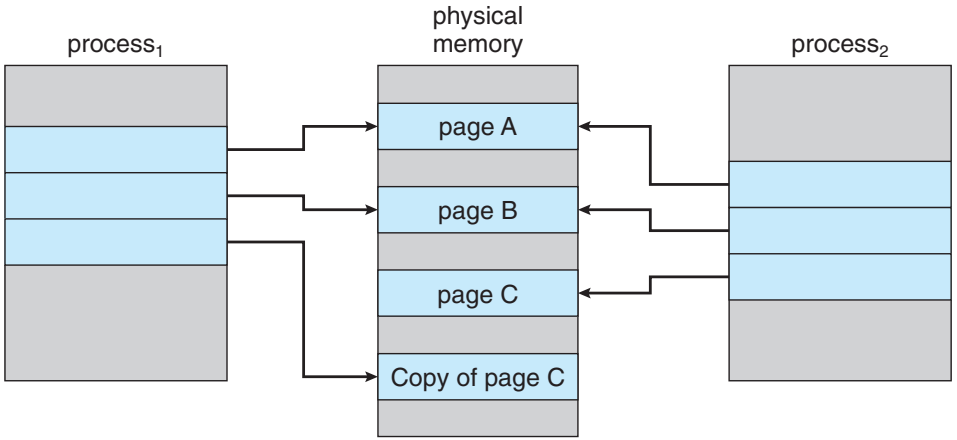
COPY-ON-WRITE

Recall fork() creates a copy of the parent's address space for the child process

- Shared until either process modifies the pages



shared

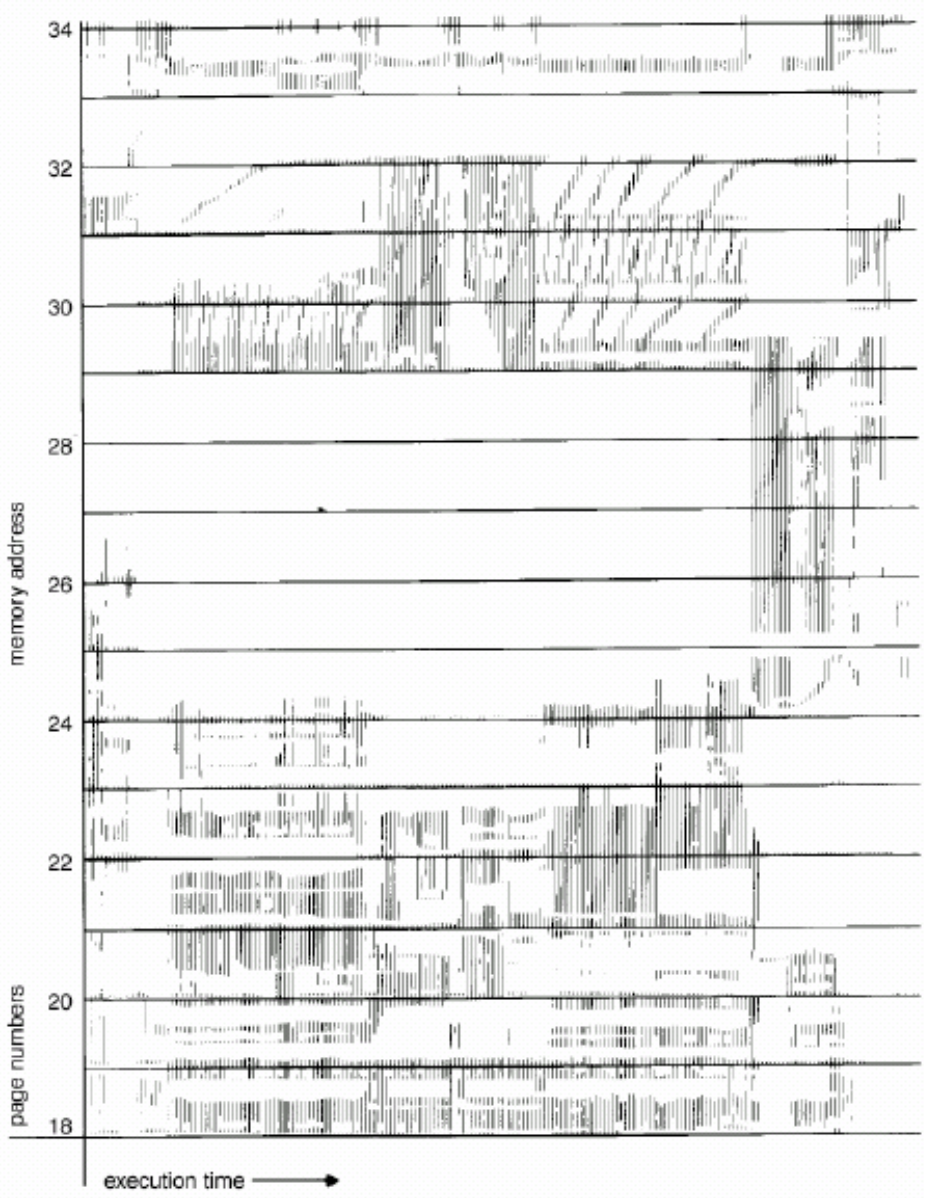


copy-on-write

LOCALITY IN MEMORY REFERENCE

Processes access at any time a small fraction of their addressing space (spatial locality) and they tend to reference again the pages they have recently referenced (temporal locality)

- How much physical memory needed (working set)
- What is likely to be accessed next



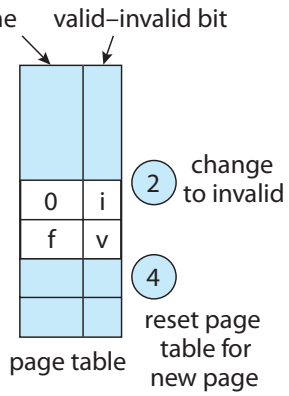
PAGE REPLACEMENT

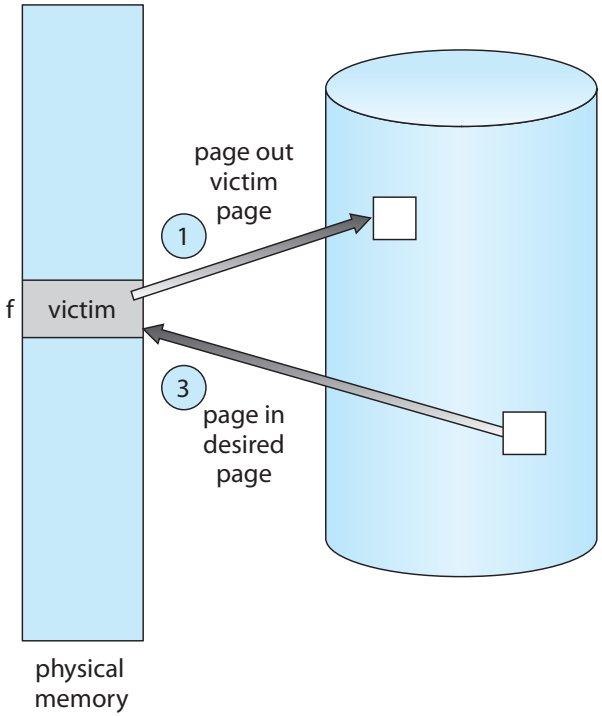
Selecting which page to expel from main memory (cache) when

- Memory (cache) is full
- Must bring in a new page

Note that two page transfers required if no free page in memory

- Can be alleviated by the use of "dirty bit" – not need to page out victim if dirty bit = 0
- (in page replacement, replace "clean" pages first)

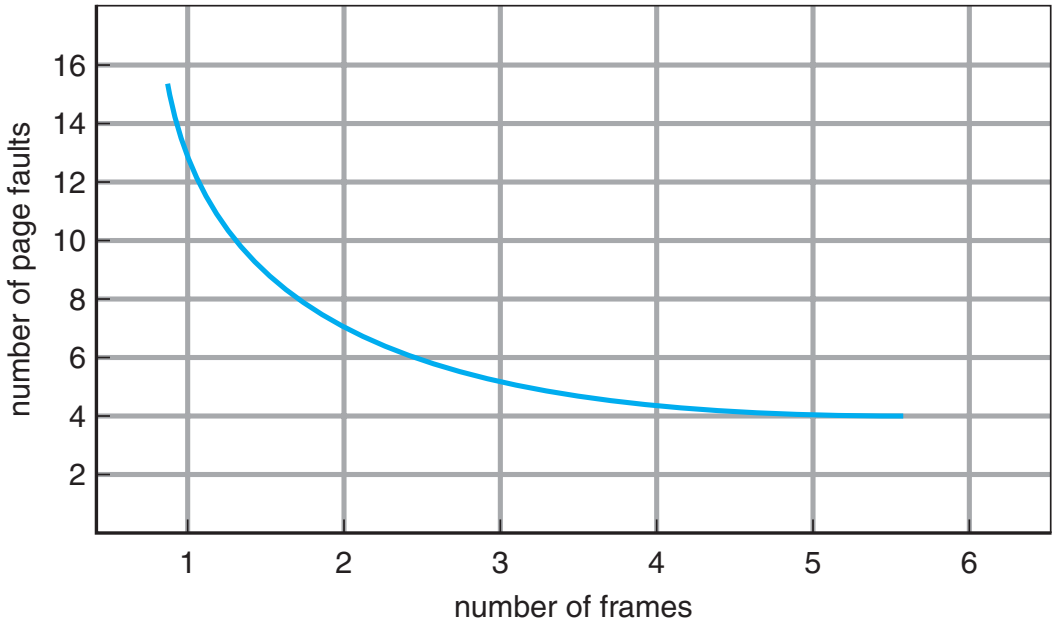




PAGE REPLACEMENT POLICY

Objectives:

- Select the right page to expel (victim)
- Have a reasonable run-time overhead
- The more physical memory, the less the page fault



PAGE REPLACEMENT POLICY

Four classes of page replacement policies

- Local policies vs globe policies:
  - Local: expel own pages
  - Global: maintain a global pool
- Fixed sized vs variable sized: each process a fixed vs variable number of frames

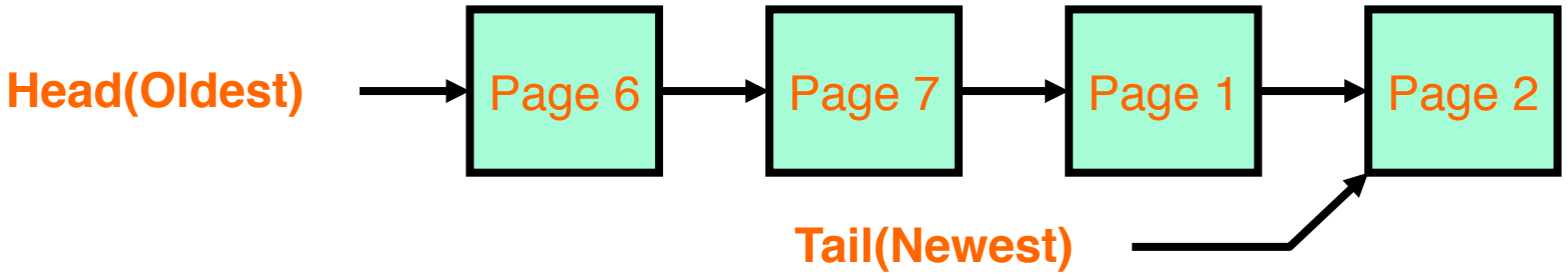
FIFO (FIRST IN, FIRST OUT)

Replace page that has been in for the longest time.

Be "fair" to pages and give them equal time.

How to implement FIFO? It's a queue (can use a linked list)

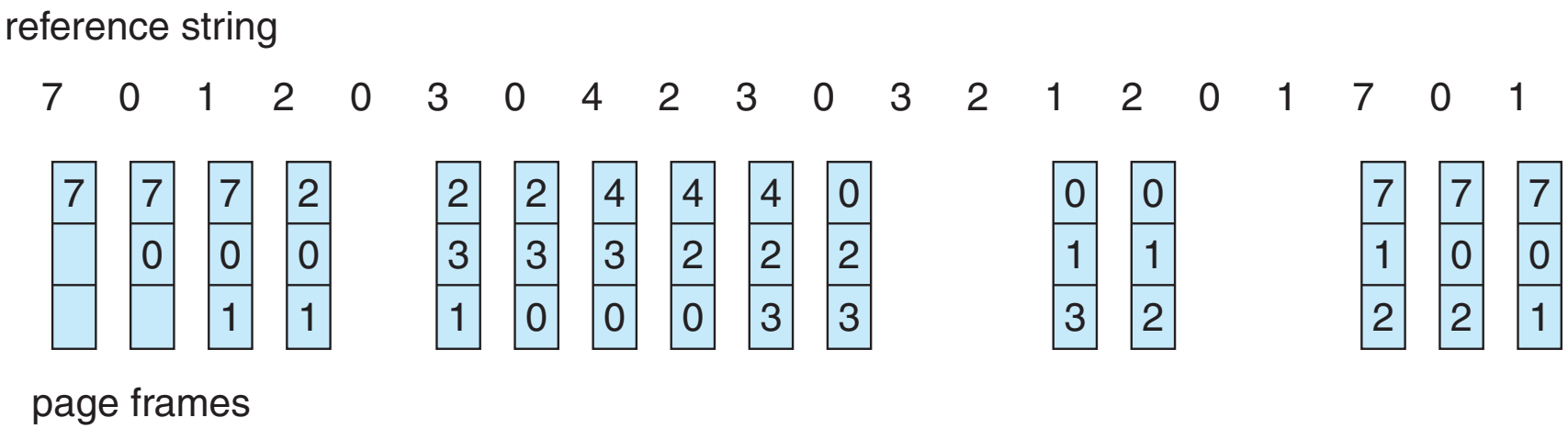
- Oldest page is at head
- When a page is brought in, add it to tail.
- Eject head if list longer than capacity



EXAMPLE

Reference string: the string of reference 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

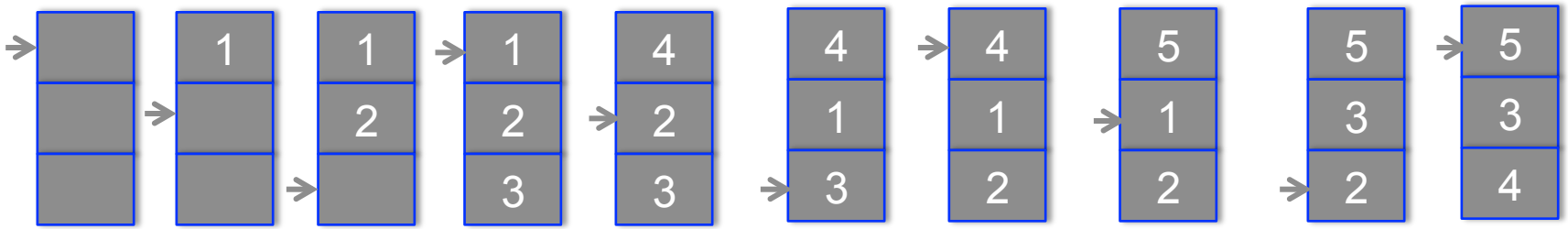
3 pages



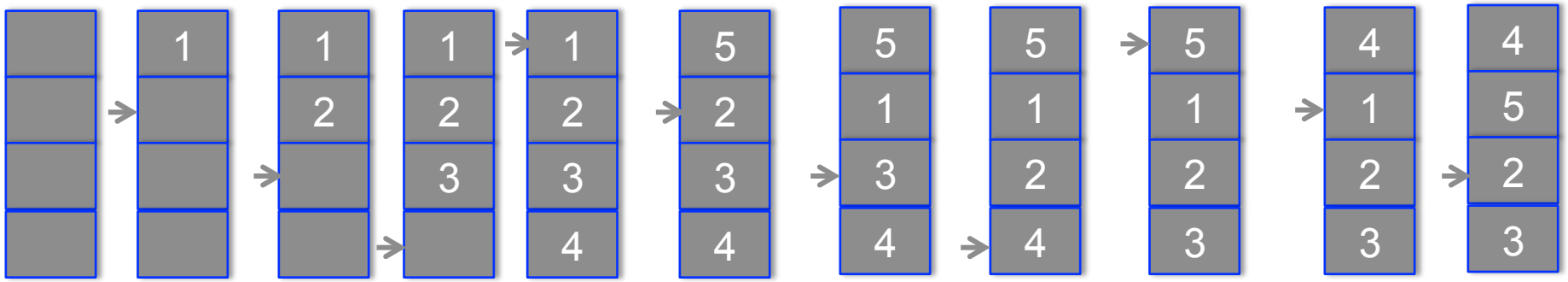
How many page faults?

EXAMPLE 2

Reference strings: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



9 page faults

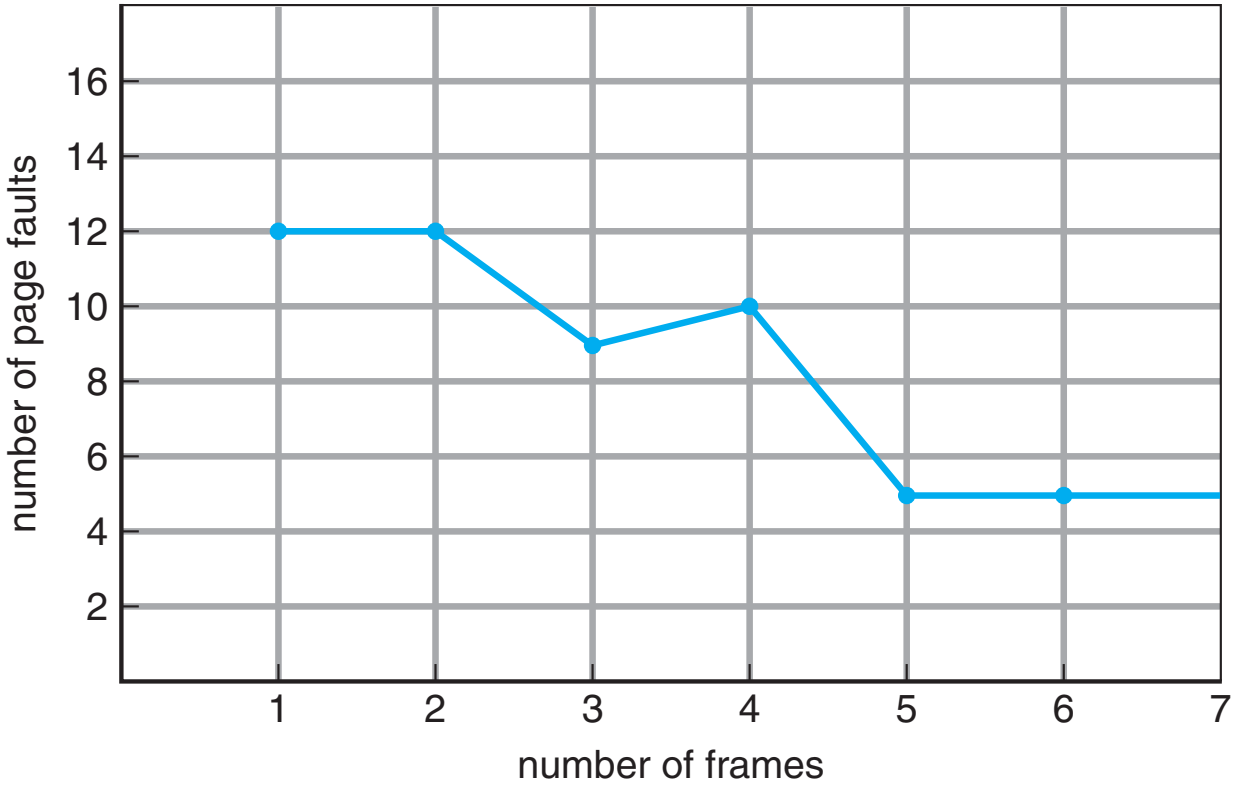


10 page faults!

BELADY'S ANOMALY

FIFO suffers from the anomaly

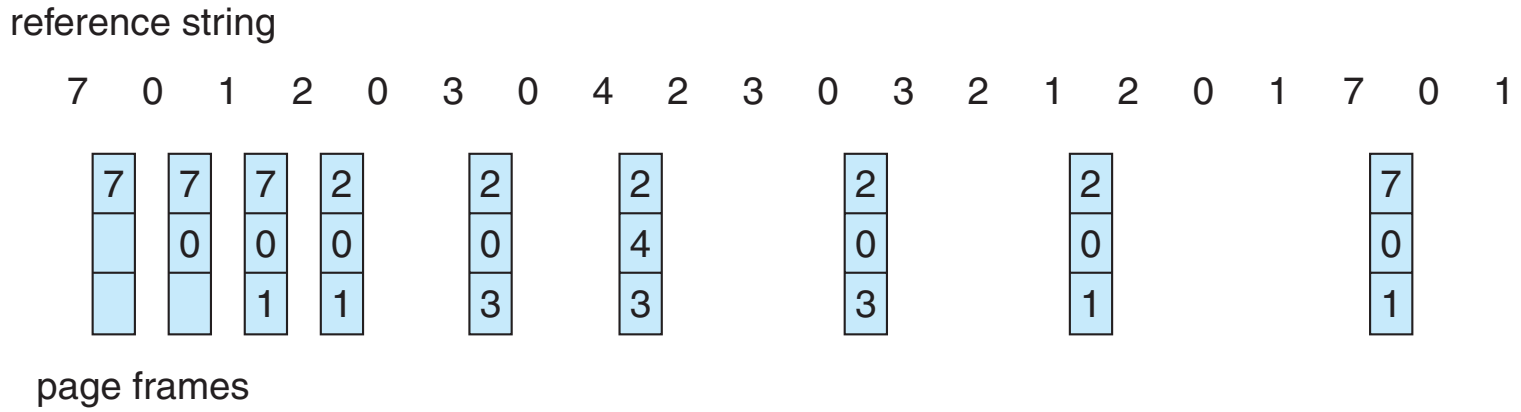
Didn't account of page usage: need to give more chances to pages that were likely to be used soon



OPTIMAL PAGE REPLACEMENT

Replace the page that will not be used for the longest period of time

Example:



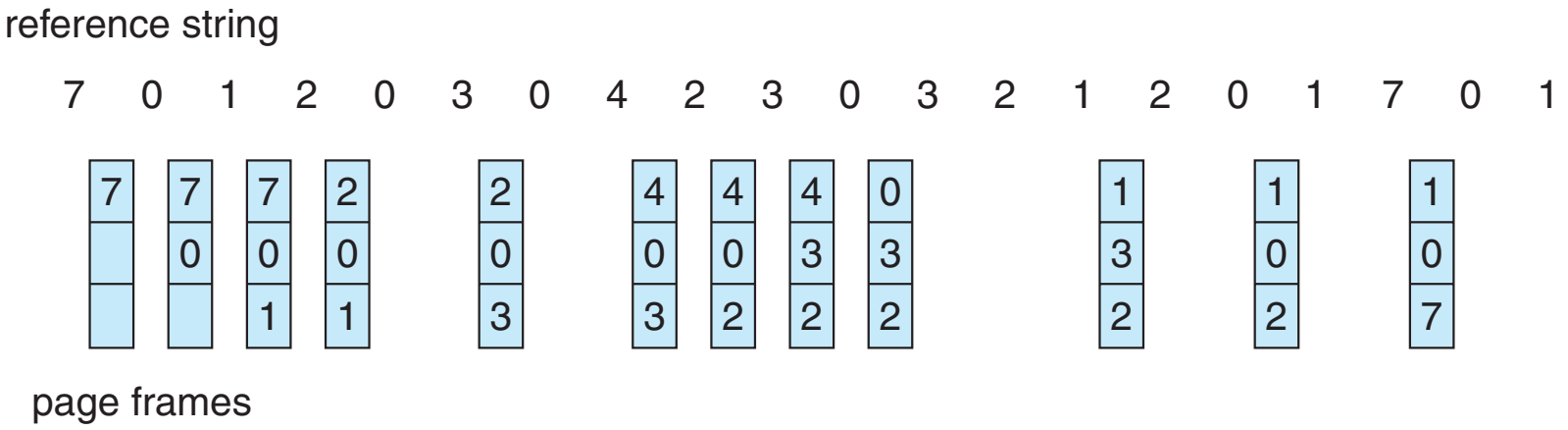
Unfortunately, we cannot look into the future

LEAST RECENTLY USED (LRU)

LRU (Least Recently Used):

- Replace page that hasn't been used for the longest time
- Programs have locality, so if something not used for a while, unlikely to be used in the near future.
- Seems like LRU should be a good approximation to OPT.

Example



IS LRU A GOOD APPROXIMATION?

Consider the following: A B C D A B C D A B C D

LRU Performs as follows (same as FIFO here):

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

- Every reference is a page fault!

OPT Does much better:

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

PROPERTIES OF LRU

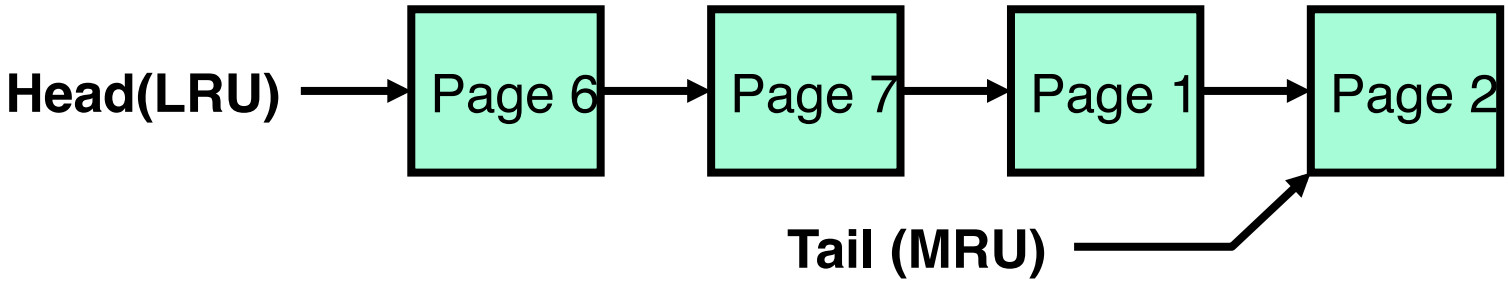
LRU can be as bad as FIFO for some reference strings

However, LRU does not suffer from Belady's anomaly

IMPLEMENTATION OF LRU

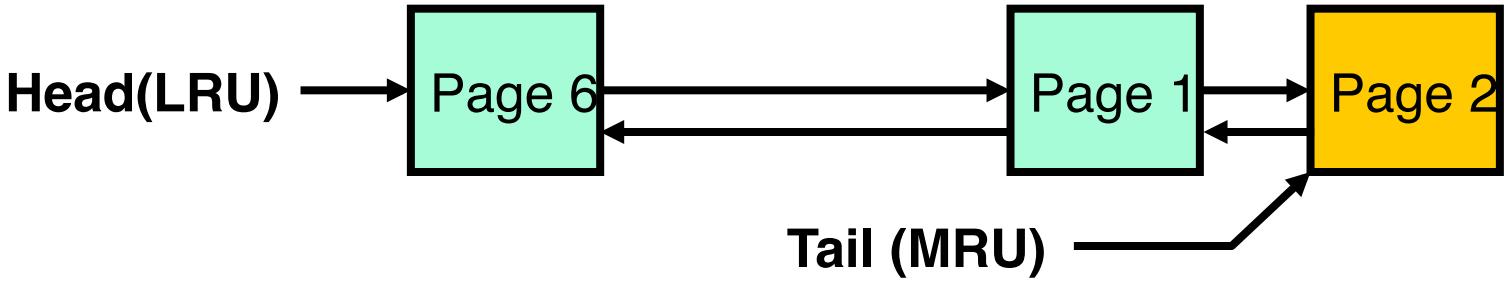
LRU page is at head

- When a page is used for the first time, add it to tail.
- Eject head if list longer than capacity



Different if we access a page that is already loaded:

- When a page is used again, **remove from list**, add it to tail.
- Eject head if list longer than capacity



IMPLEMENTATION OF LRU

Problems with this scheme for paging?

- Updates are happening on page use, not just swapping
- List structure requires extra pointers c.f. FIFO, more updates

In practice, approximate LRU with simpler implementation

- **Use Reference bits**
  - Second chance
  - Clock algorithm

IMPLEMENTING LRU & SECOND CHANCE

Perfect:

- Timestamp page on each reference
- Keep list of pages ordered by time of reference
- Too expensive to implement in reality for many reasons

Second Chance Algorithm:

- Approximate LRU
  - Replace an old page, not the oldest page
- FIFO with “use” bit

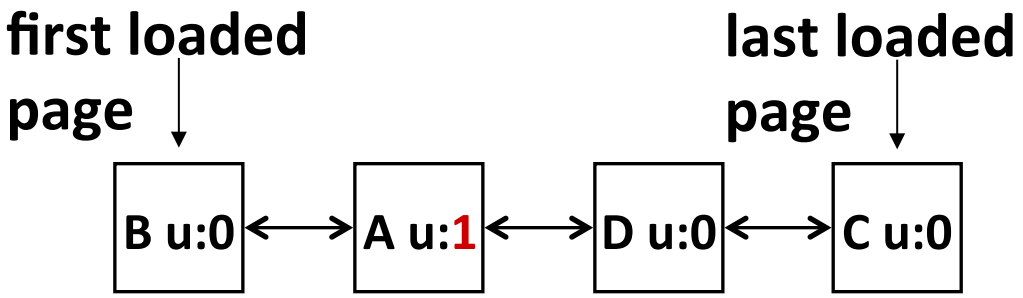
Details

- A “use” bit per physical page
  - set when page accessed
- On page fault check page at head of queue
  - If use bit=1 → clear bit, and move page to tail (give the page second chance!)
  - If use bit=0 → replace page
- Moving pages to tail still complex

SECOND CHANCE ILLUSTRATION

Max page table size 4

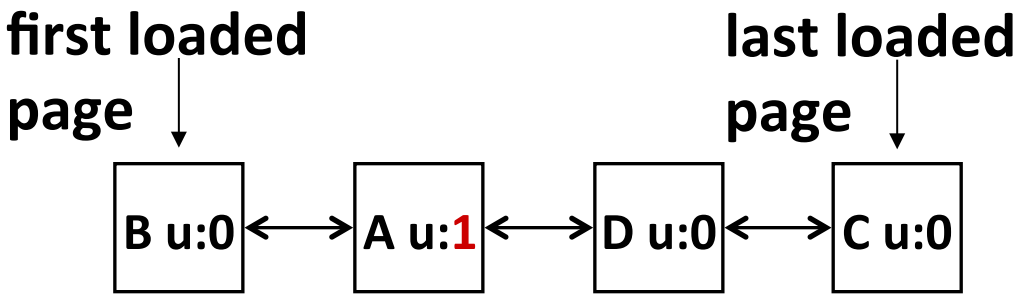
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives



SECOND CHANCE ILLUSTRATION

Max page table size 4

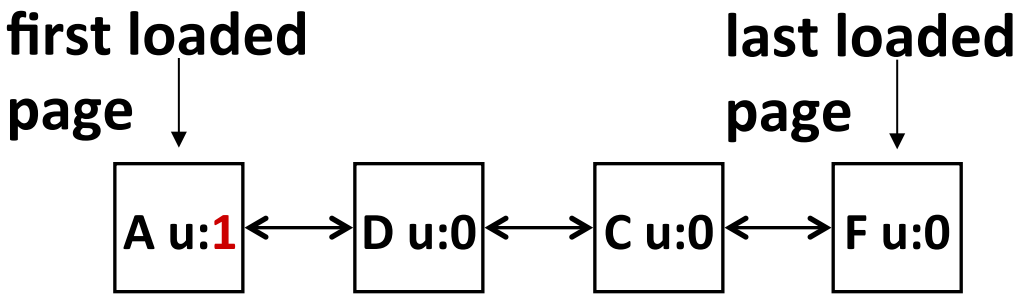
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives



SECOND CHANCE ILLUSTRATION

Max page table size 4

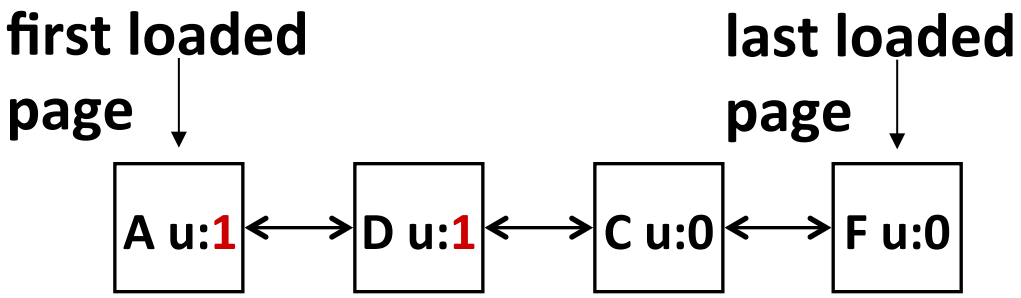
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives



SECOND CHANCE ILLUSTRATION

Max page table size 4

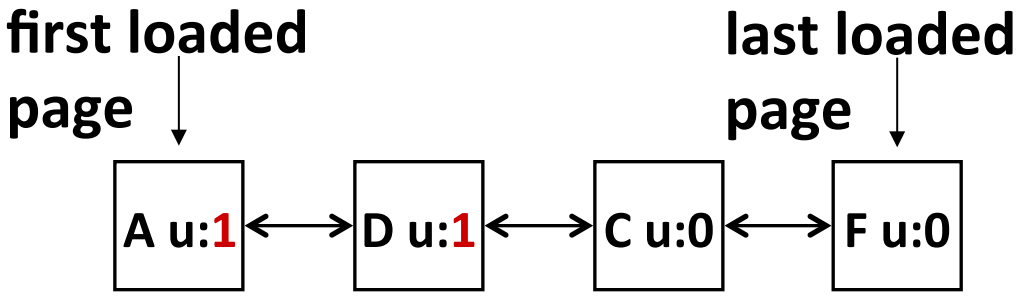
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D



SECOND CHANCE ILLUSTRATION

Max page table size 4

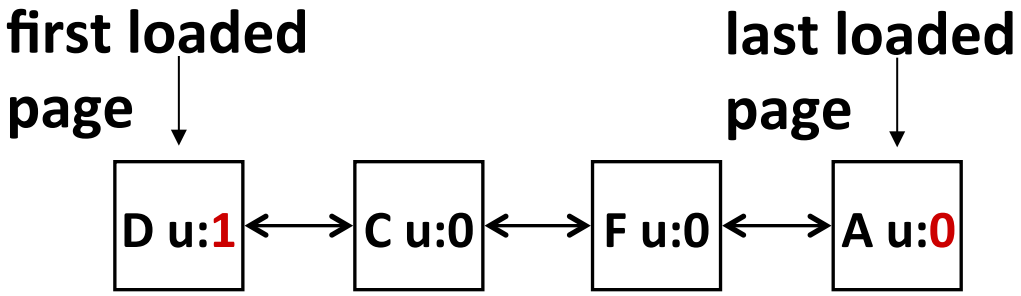
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D
- Page E arrives



SECOND CHANCE ILLUSTRATION

Max page table size 4

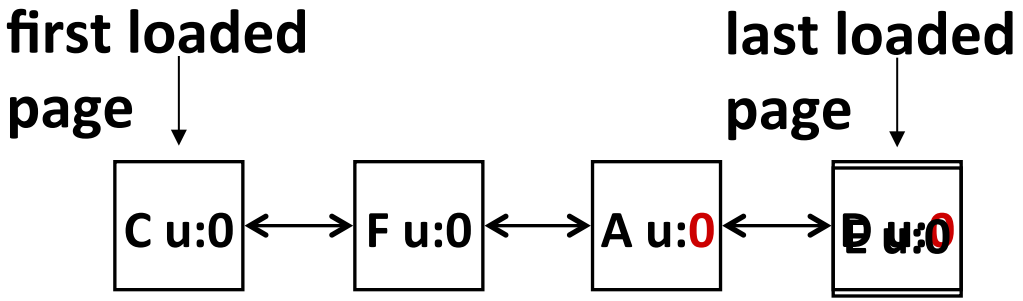
- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D
- Page E arrives



SECOND CHANCE ILLUSTRATION

Max page table size 4

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D
- Page E arrives



CLOCK ALGORITHM

Clock Algorithm: more efficient implementation of second chance algorithm

- Arrange physical pages in circle with single clock hand

Details:

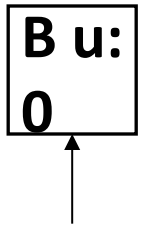
- On page fault:
  - Check use bit: 1 → used recently; clear and leave it alone
    0 → selected candidate for replacement
  - Advance clock hand (not real time)
- Will always find a page or loop forever?

CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives

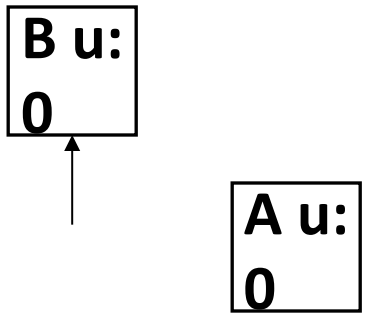


CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A

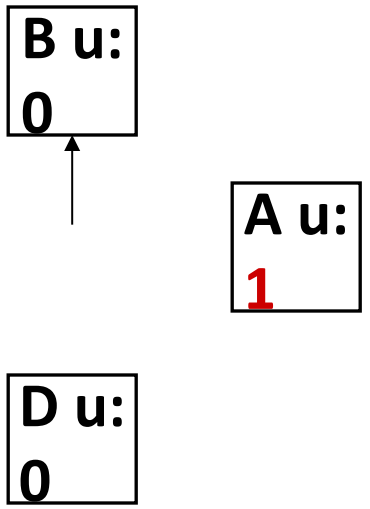


CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives

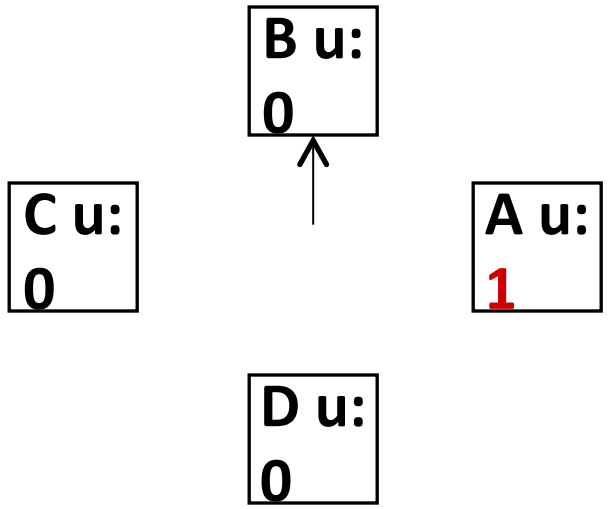


CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives

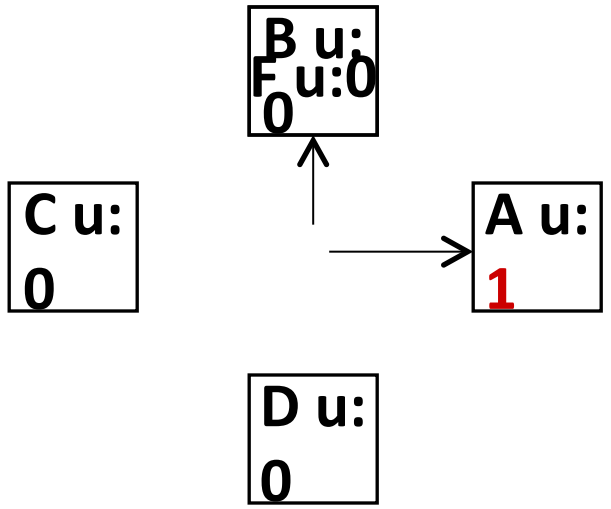


CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D

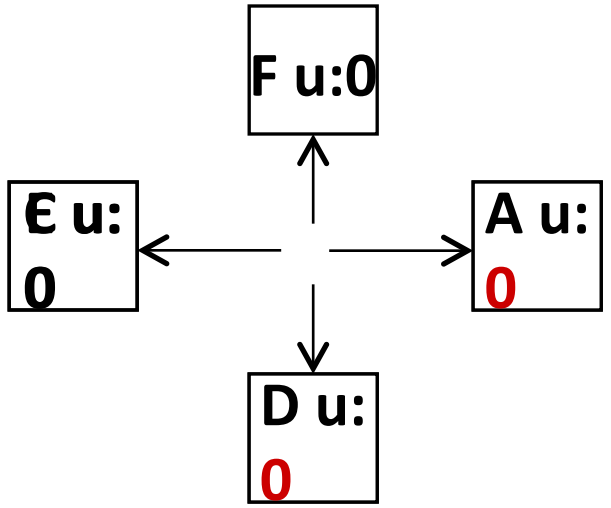


CLOCK REPLACEMENT ILLUSTRATION

Max page table size 4

Invariant: point at oldest page

- Page B arrives
- Page A arrives
- Access page A
- Page D arrives
- Page C arrives
- Page F arrives
- Access page D
- Page E arrives



CLOCK ALGORITHM: DISCUSSION

What if hand moving slowly?

- Good sign or bad sign?
  - Not many page faults and/or find page quickly

What if hand is moving quickly?

- Lots of page faults and/or lots of reference bits set

N^{TH} CHANCE VERSION OF CLOCK ALGORITHM

Nth chance algorithm: Give page N chances

- OS keeps counter per page: # sweeps
- On page fault, OS checks use bit:
  - 1 → clear use and also clear counter (used in last sweep)
  - 0 → increment counter; if count=N, replace page
- Means that clock hand has to sweep by N times without page being used before page is replaced

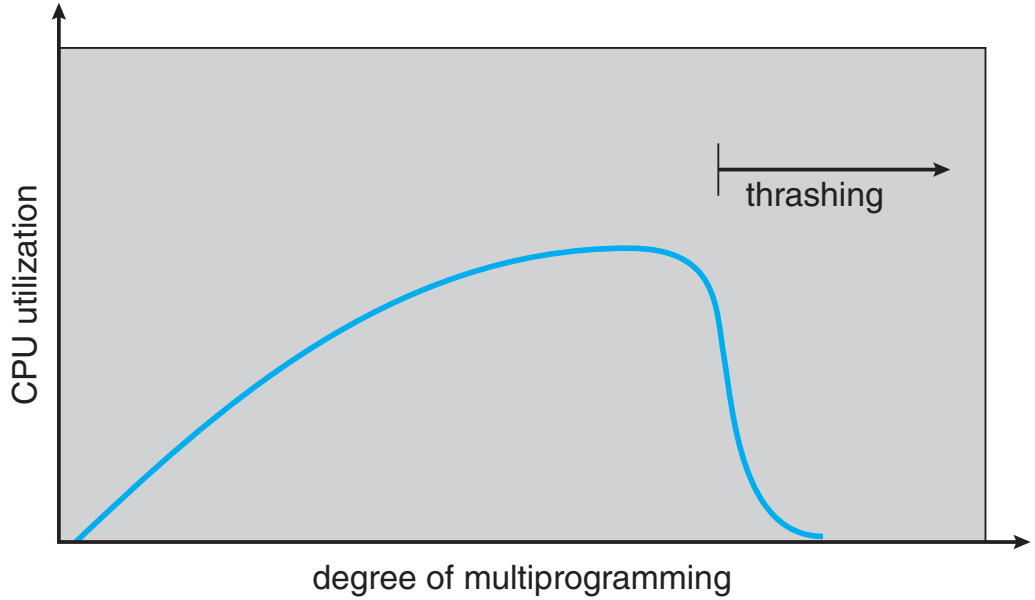
How do we pick N?

- Why pick large N? Better approx to LRU
  - If N ~ 1K, really good approximation
- Why pick small N? More efficient
  - Otherwise might have to look a long way to find free page

What about dirty pages?

- Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing?
- Common approach:
  - Clean pages, use N=1
  - Dirty pages, use N=2 (and write back to disk when N=1)

THRASHING



If a process does not have "enough" pages, the page-fault rate is very high. This leads to:

- low CPU utilization
- operating system spends most of its time swapping to disk

Thrashing: a process is busy swapping pages in and out

Questions:

- How do we detect Thrashing?
- What is best response to Thrashing?

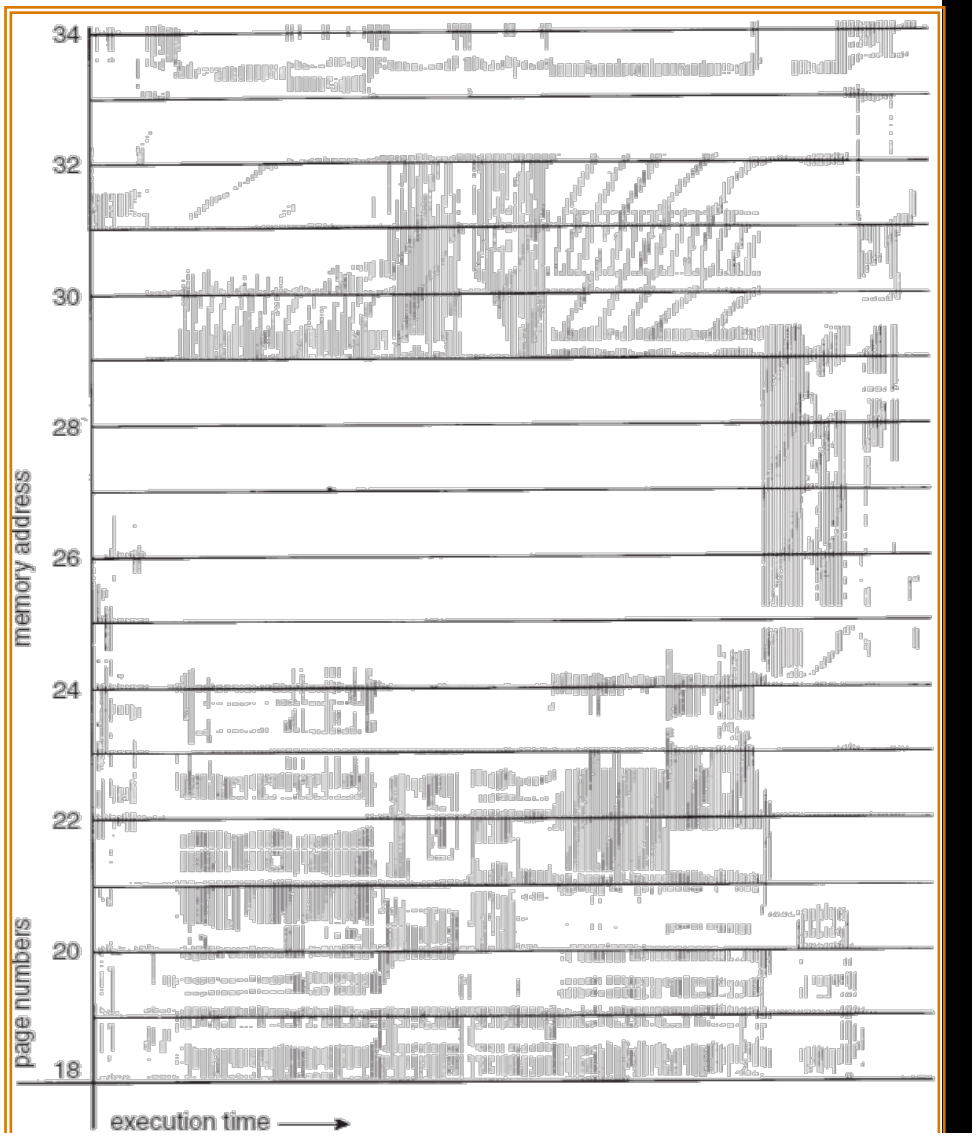
LOCALITY IN A MEMORY-REFERENCE PATTERN

Program Memory Access Patterns have temporal and spatial locality

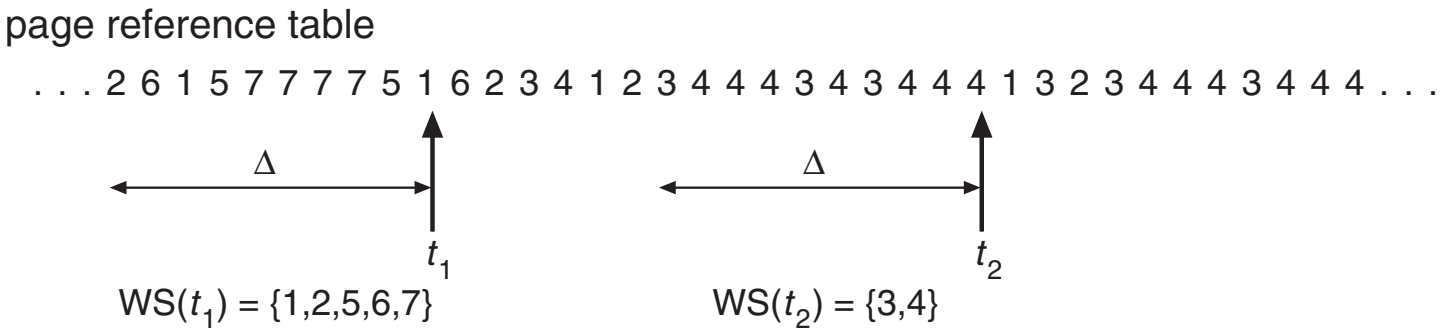
- Group of Pages accessed along a given time slice called the “Working Set”
- Working Set defines minimum number of pages needed for process to behave well

Not enough memory for Working Set ➔ Thrashing

- Better to swap out process?



WORKING-SET MODEL



Δ = working-set window = fixed number of page references

- Example: 10,000 accesses

WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)

- if Δ too small will not encompass entire locality
- if Δ too large will encompass several localities
- if Δ= ∞, will encompass entire program

D = Σ|WS_i| = total demand frames

if D > physical memory ➔Thrashing

- Policy: if D > physical memory, then suspend/swap out processes
- This can improve overall system behavior by a lot!

WHAT ABOUT COMPULSORY MISSES?

Recall that compulsory misses are misses that occur the first time that a page is seen

- Pages that are touched for the first time
- Pages that are touched after process is swapped out/swapped back in

Clustering:

- On a page-fault, bring in multiple pages “around” the faulting page
- Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Tradeoff: Prefetching may evict other in-use pages for never-used prefetched pages

Working Set Tracking:

- Use algorithm to try to track working set of application
- When swapping process back in, swap in working set

POP QUIZ 4: ADDRESS TRANSLATION

Q1: True _ False _ Paging does not suffer from external fragmentation

Q2: True _ False _ The segment offset can be larger than the segment size

Q3: True _ False _ Paging: to compute the physical address, add physical page # and offset

Q4: True _ False _ Uni-programming doesn't provide address protection

Q5: True _ False _ Virtual address space is always larger than physical address space

Q6: True _ False _ Inverted page tables keeps fewer entries than multi-level page tables