Memory Management

Thierry Sans

Today's questions

- How to allocate free space?
  Dynamic Memory Allocation
- How to evict pages from memory? (a.k.a when to swap)
  Page Replacements Algorithms
- How much memory to give to each process?
  Working Set Model

Managing Free Memory

Memory allocation

Static Allocation a.k.a stack allocation (fixed in size)
data structures that do not need to grow or shrink
such as global and local variables e.g. `char name[16];`

- ➡ done at compile time
- ✓ restricted, but simple and efficient

Dynamic Allocation a.k.a heap allocation (change in size)
data structure that might increase/decrease in size according to different demands e.g `name = (char *) malloc(16);`

- ➡ done at run time
- ◉ general, but difficult to implement **(our focus today)**

Heap allocation more concretely

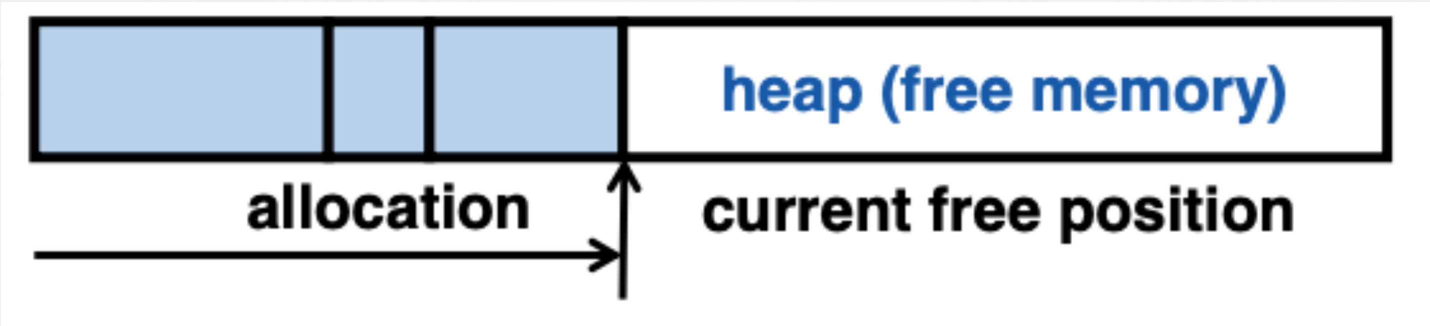
➡ Manage contiguous range of logical addresses

- `malloc(size)` returns a pointer to a block of memory of at least `size` bytes, or `NULL`
- `free(ptr)` releases the previously- allocated block pointed to by `ptr`

Why is heap allocation hard?

➡ Satisfy arbitrary set of allocation and frees.

✓ Easy without free : set a pointer to the beginning of some big chunk of memory (heap) and increment on each allocation



◉ Problem : free creates holes (fragmentation)
Lots of free space but cannot satisfy request!

What is fragmentation really?

➡ Inability to use memory that is free

Two factors required for fragmentation

1. Different lifetimes
   If all objects die at the same time, then no fragmentation
2. Different sizes
   if all requests the same size, then no fragmentation

Important decisions

Placement choice : where in free memory to put a requested block?

- Freedom : can select any memory in the heap
- Ideal : put block where it won't cause fragmentation later (impossible in general, requires future knowledge)

Split free blocks to satisfy smaller requests?

- Freedom : can choose any larger block to split
- Ideal : choose block to minimize fragmentation

Coalescing free blocks to yield larger blocks

Fragmentation is impossible to solve

Theoretical result

For any allocation algorithm, there exist streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation L

➡ Avoiding fragmentation is impossible

Heap Memory Allocator

What the memory allocator must do?

➡ Track which parts of memory in use, which parts are free ideally no wasted space, no time overhead

What the memory allocator cannot do?

- Control order of the number and size of requested blocks
- Know the number, size, & lifetime of future allocations

What makes a good memory allocator?

➡ The one that avoid compaction (time consuming)

➡ The one that minimize fragmentation

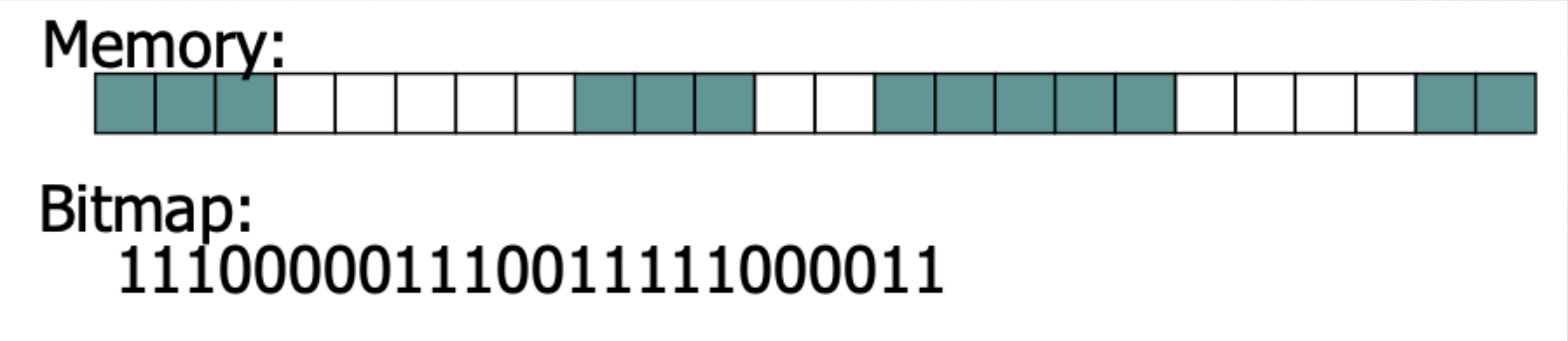
Tracking memory allocation with **bitmaps**

Bitmap : 1 bit per allocation unit

- 0 means free
- 1 means allocated

➡ Allocating a N-unit chunk requires scanning bitmap for sequence of N zero's

◉ Slow



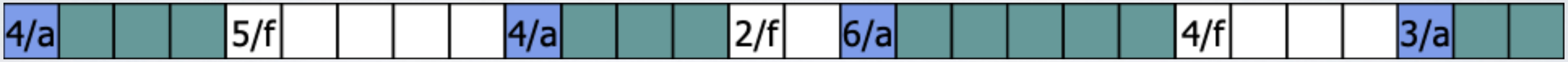
Tracking memory allocation with **lists**

Free lists

Maintain linked list of allocated and free segments

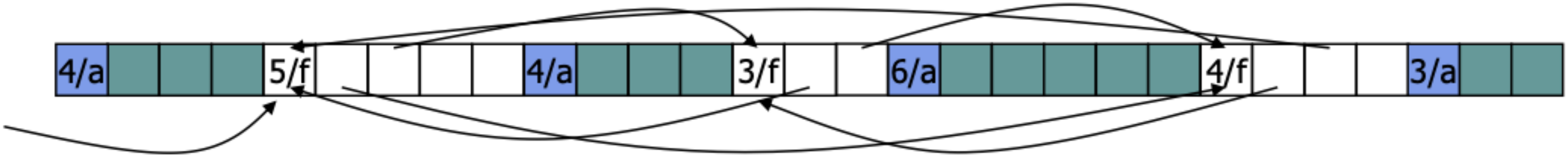
Implicit list

- Each block has header that records size and status (allocated or free)
- Searching for free block is linear in total number of blocks



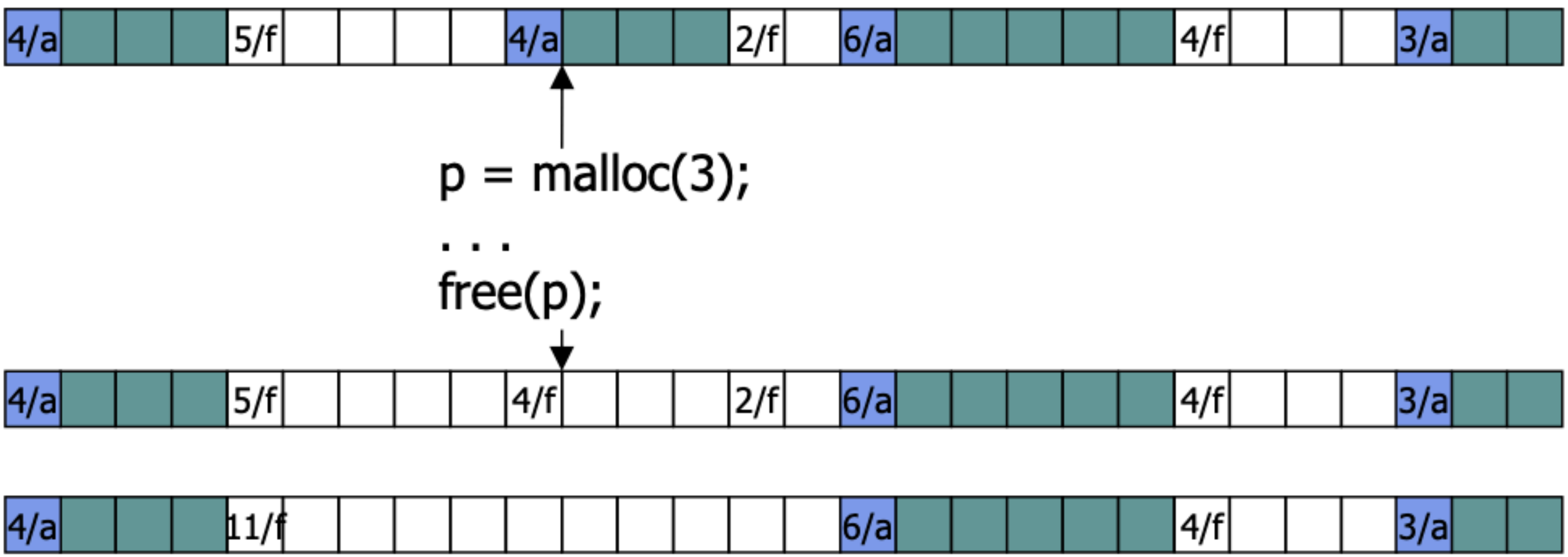
Explicit list

Store pointers in free blocks to create doubly-linked list



Freeing Blocks

- ➡ Adjacent free blocks can be coalesced (merged)



Placement Algorithms

- **First-fit**
  choose first block that is large enough; search can start at beginning, or where previous search ended (a.k.a next-fit)
- **Best-fit**
  choose the block that is closest in size to the request
- Worst-fit
  choose the largest block
- Quick-fit
  keep multiple free lists for common block sizes
- **Buddy systems**
  round up allocations to power of 2 to make management faster

Best Fit

- ➡ Minimize fragmentation by allocating space from block that leaves smallest fragment

Data structure
heap is a list of free blocks, each has a header holding block size and a pointer to the next block

Code
search freelist for block closest in size to the request

First Fit

- ➡ Pick the first block that fits

Data structure
free list, sorted LIFO, FIFO, or by address

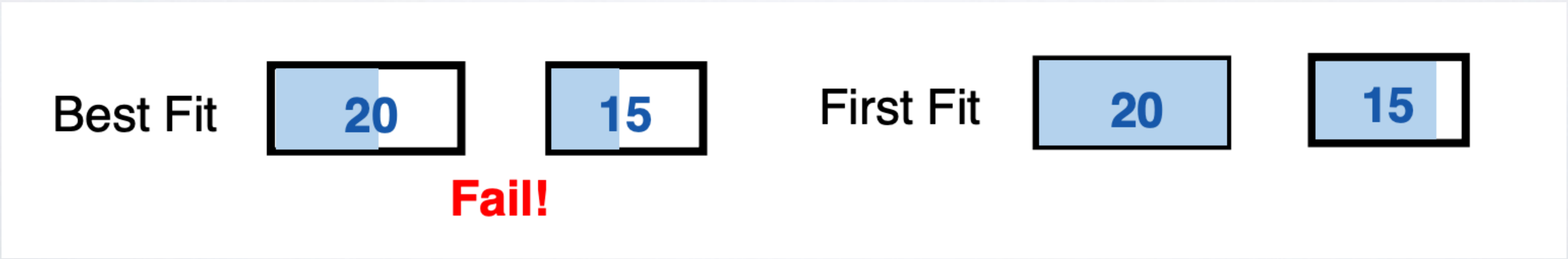
Code
scan list, take the first one

Best Fit vs First Fit

Suppose memory has two free blocks (size 20 and 15)

- Workload 1 : `alloc(10), alloc(20)`

- Workload 2 : `alloc(8), alloc(12), alloc(12)`



Comparing First Fit and Best Fit

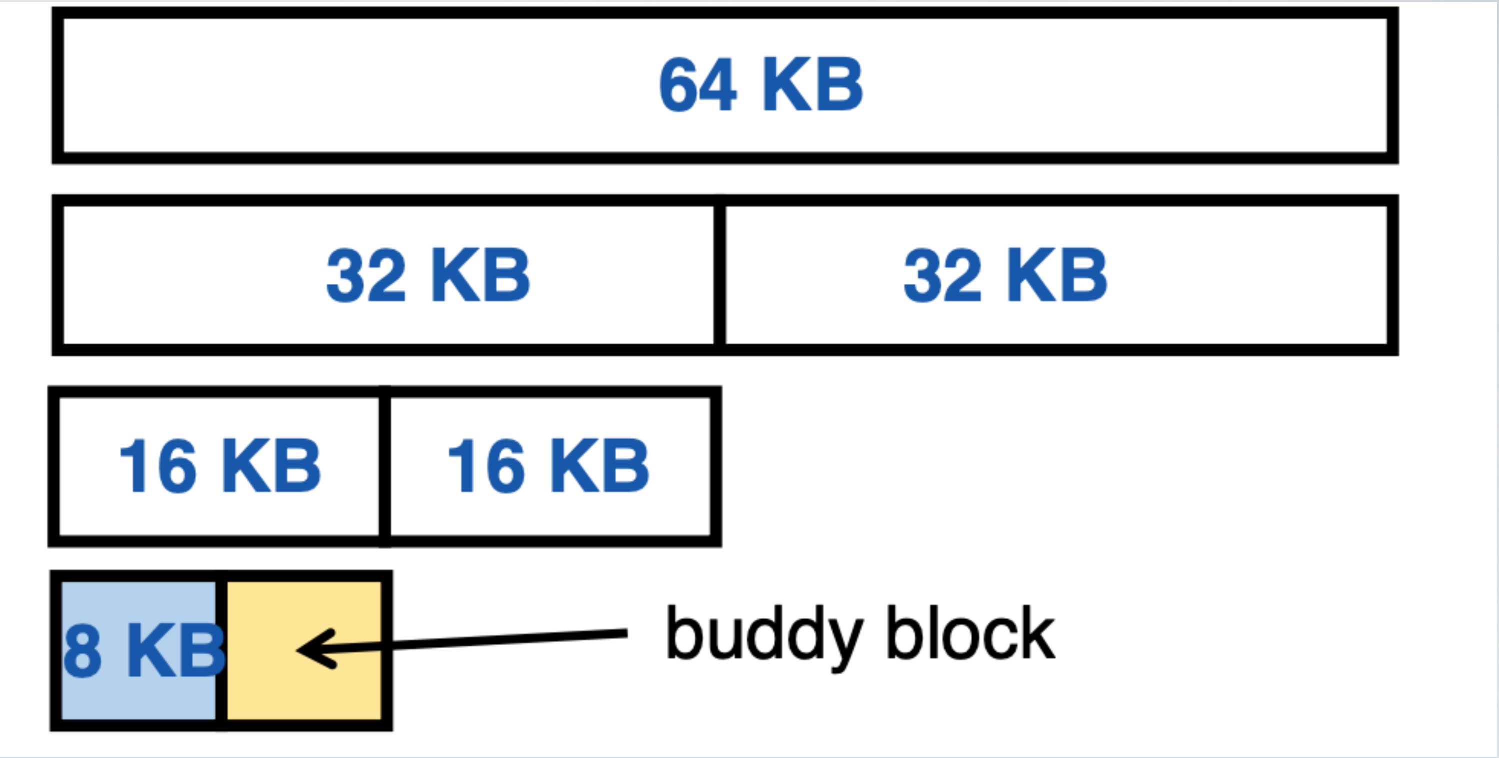
First Fit

- ✓ Simplest, and often fastest and most efficient
- ◉ May leave many small fragments near start of memory that must be searched repeatedly

Best Fit

- ✓ In practice, similar storage utilization to first-fit
- ◉ Left-over fragments tend to be small (unusable)

Buddy Allocation



➡ Allocate blocks in 2^k

Data structure

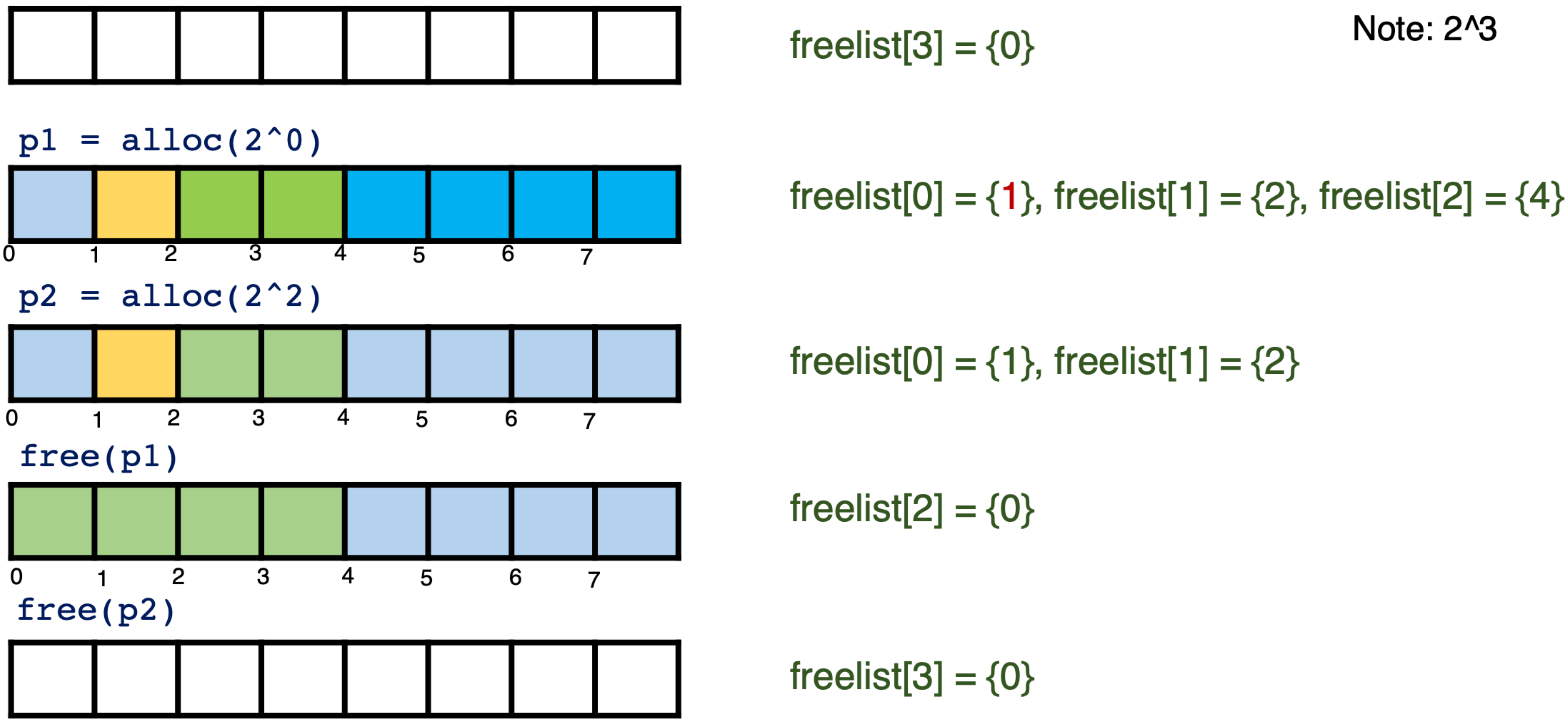
Maintain n free lists of blocks of size $2^0, 2^1, \ldots, 2^n$

Code

- recursively divide larger blocks until reach suitable block
- insert buddy blocks into free lists
- upon free, recursively coalesce block with buddy if buddy free

➡ the addresses of the buddy pair only differ by one bit

Example


Note: 2^3
freelist[3] = {0}
p1 = alloc(2^0)
freelist[0] = {1}, freelist[1] = {2}, freelist[2] = {4}
0 1 2 3 4 5 6 7
p2 = alloc(2^2)
freelist[0] = {1}, freelist[1] = {2}
0 1 2 3 4 5 6 7
free(p1)
freelist[2] = {0}
0 1 2 3 4 5 6 7
free(p2)
freelist[3] = {0}

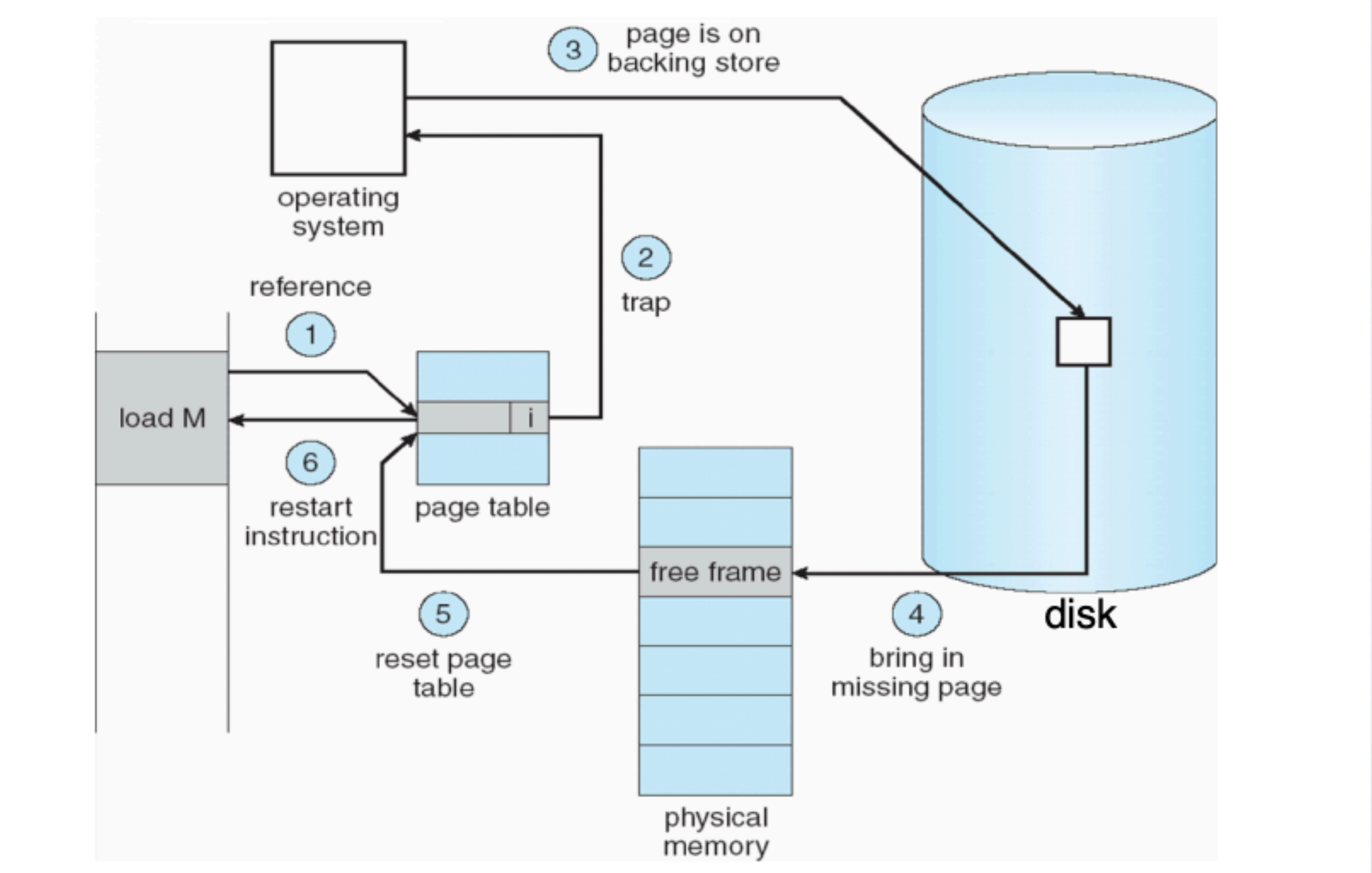
Advantages

- ✓ Fast search (allocate) and merge (free)
- ✓ Avoid iterating through free list
- ✓ Avoid external fragmentation for req of 2^n
- ✓ Keep physical pages contiguous
- ➡ Used by Linux, FreeBSD

Page Replacements Algorithms

(recap) Swapping

➡ Use disk to simulate larger virtual than physical memory



Page Fault and Page Replacement

What happen when there is a page fault?

➡ The OS loads the faulted page frame from disk into physical memory

What when there is no physical memory available?
(or the process has reach its limit of maximum page frame allowed)

➡ The OS must evict an existing frame (swap) to replace it with the new one

How to determine which page frame should be evicted?

➡ The page replacement algorithm (a.k.a page eviction policy) determines which page frame to evict to minimize the fault rate (affecting paging performances)

Page Replacement Algorithms

The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove

- **FIFO - First In, First Out**
  evict the oldest page in the system
- **LRU - Last Recently Used**
  evict the page that has not been used for the longest time in the past
- **Second Chance**
  an approximation of LRU (more implementable)

➡ Replacement algorithms are evaluated on a reference string by counting the number of page faults

FIFO - First In, First Out (with 3 physical pages)

➡ Evict the oldest page in the system

Access	Hit/Miss	Evict	P0	P1	P2
1	Miss		1		
2	Miss		1	2	
3	Miss		1	2	3
4	Miss	1	4	2	3
1	Miss	2	4	1	3
2	Miss	3	4	1	2
5	Miss	4	5	1	2
1	Hit		5	1	2
2	Hit		5	1	2
3	Miss	1	5	3	2
4	Miss	2	5	3	4
5	Hit		5	3	4

- Total 9 misses

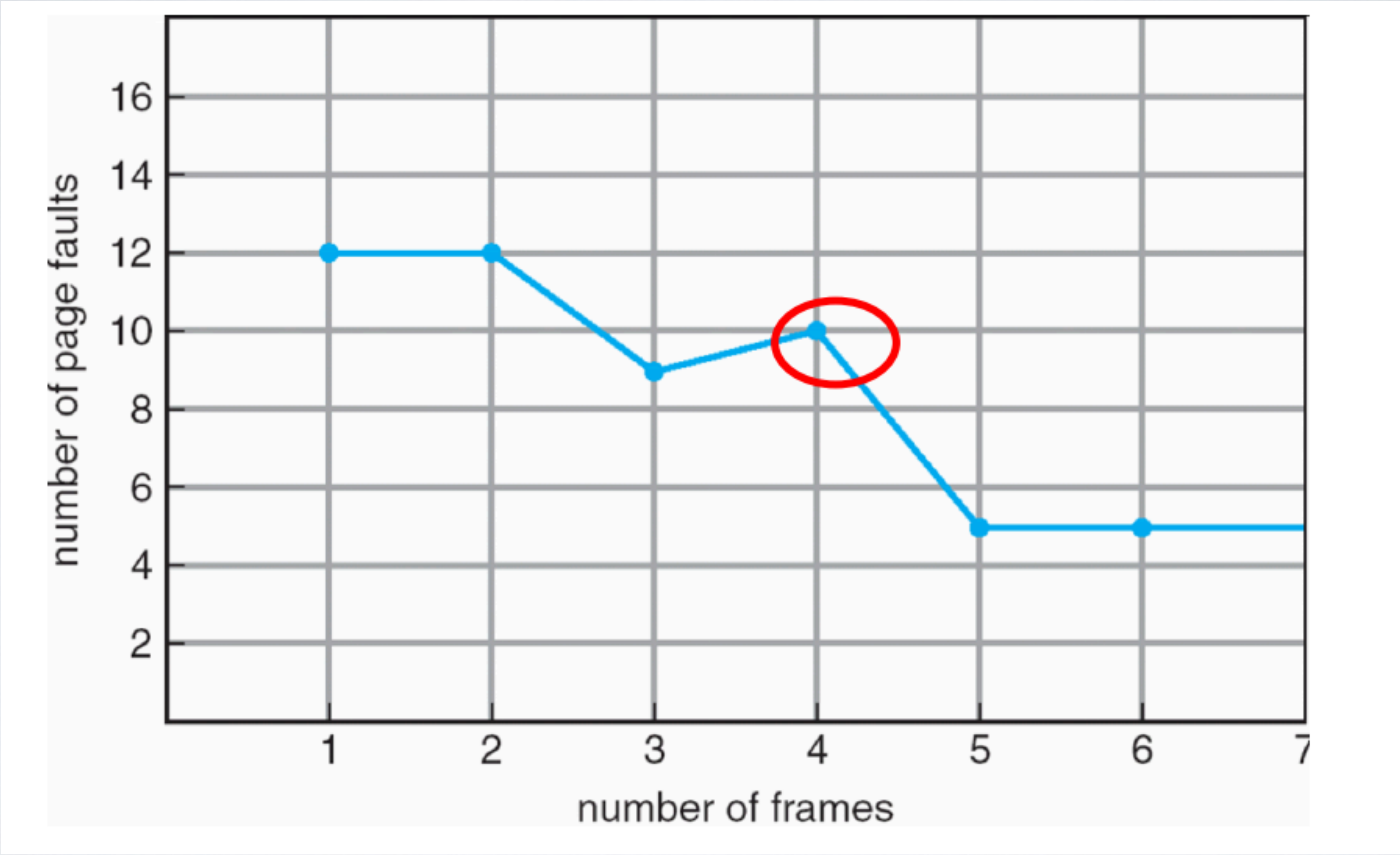
Does having **more physical memory** automatically means **fewer page faults**?

FIFO - First In, First Out (with 4 physical pages)

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	1	5	2	3	4
1	Miss	2	5	1	3	4
2	Miss	3	5	1	2	4
3	Miss	4	5	1	2	3
4	Miss	5	4	1	2	3
5	Miss	1	4	5	2	3

- Total 10 misses with 4 physical pages (only 9 with 3 physical pages)

Belady's Anomaly



- More physical memory doesn't always mean fewer faults

Belady's Algorithm

- ➡ What is optimal if you knew the future?

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	4	1	2	3	5
1	Hit		1	2	3	5
2	Hit		1	2	3	5
3	Hit		1	2	3	5
4	Miss	1	4	2	3	5
5	Hit		4	2	3	5

- Total 6 misses

Belady's Algorithm

Belady's Algorithm is known (proven) to be the optimal page replacement algorithm

- Problem : it is hard (impossible) to predict the future

➡ Belady's algorithm is useful to compare page replacement algorithms with the optimal to gauge room for improvement

LRU - Last Recently Used

➡ Evict the page that has not been used for the longest time in the past

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1	2	3	4
2	Hit		1	2	3	4
5	Miss	3	1	2	5	4
1	Hit		1	2	5	4
2	Hit		1	2	5	4
3	Miss	4	1	2	5	3
4	Miss	5	1	2	4	3
5	Miss	1	5	2	4	3

◉ Total 8 misses

How to implement LRU

Idea 1 : stamp the pages with timer value

- On access, stamp the PTE with the timer value
- On miss, scan page table to find oldest counter value
- Problem : would double memory traffic!

Idea 2 : keep doubly-linked list of pages

- On access, move the page to the tail
- On miss, remove the head page
- Problem : again, very expensive!

So, we need to approximate LRU instead

➡ Second Chance page replacement algorithm

Second Chance

Access	Hit/Miss	Evict	P0	P1	P2	P3
1	Miss		1			
2	Miss		1	2		
3	Miss		1	2	3	
4	Miss		1	2	3	4
1	Hit		1*	2	3	4
2	Hit		1*	2*	3	4
5	Miss	3	1	2	5	4
1	Hit		1*	2	5	4
2	Hit		1*	2*	5	4
3	Miss	4	1*	2*	5	3
4	Miss	5	1	2	4	3
5	Miss	3	1	2	4	5

- Total 8 misses

Second Chance implementation
Version 1 : FIFO-like algorithm

➡ use the accessed bit supported by most hardware

Data structure
linked list of pages with two pointers head and tail

Code

- on hit, set the corresponding page's accessed bit to 1
- on miss
  1. while head's accessed bit is 1, set head's accessed bit to 0 and move it to tail
  2. else head's accessed bit is 0, swap the head an move the new page to tail

◉ Good performances but requires moving pages on every miss

Second Chance implementation Version 2 : Clock algorithm

➡ use the accessed bit supported by most hardware

Data structure
circular linked list of pages (clock) with one pointer (hand)

Code

- on hit, set the corresponding page's accessed bit to 1
- on miss
  1. while hand's accessed bit is 1, set hand's accessed bit to 0 and move to next page
  2. else if hand's accessed bit is 0, swap the hand's page with the new page and an move next page

◉ Better performances than fifo-like second chance (no rotation on miss)

Other Replacement Algorithms

Random eviction

- Dirt simple to implement
- Not overly horrible (avoids Belady's anomaly)

LFU (least frequently used) eviction

- Instead of just A bit, count # times each page accessed
- Least frequently accessed must not be very useful (or maybe was just brought in and is about to be used)
- Decay usage counts over time (for pages that fall out of usage)

MFU (most frequently used) algorithm

- Because page with the smallest count was probably just brought in and has yet to be used

➡ Neither LFU nor MFU used very commonly

Working Set Model

Fixed vs. Variable Space

How to determine how much memory to give to each process?

Fixed space algorithms

- Each process is given a limit of pages it can use
- When it reaches the limit, it replaces from its own pages

➡ Local replacement : some processes may do well while others suffer

Variable space algorithms

- Process' set of pages grows and shrinks dynamically

➡ Global replacement : one process can ruin it for the rest

Working Set Model

A working set of a process is used to model the dynamic locality of its memory usage

WS(t,w) = {pages P | P was referenced in the time interval (t, t-w)}
t – time, w – working set window (measured in page refs)

- ➡ A page is in the working set (WS) only if it was referenced in the last w references

Working Set Size

The working set size is the # of unique pages in the working set
i.e the number of pages referenced in the interval (t, t-w)

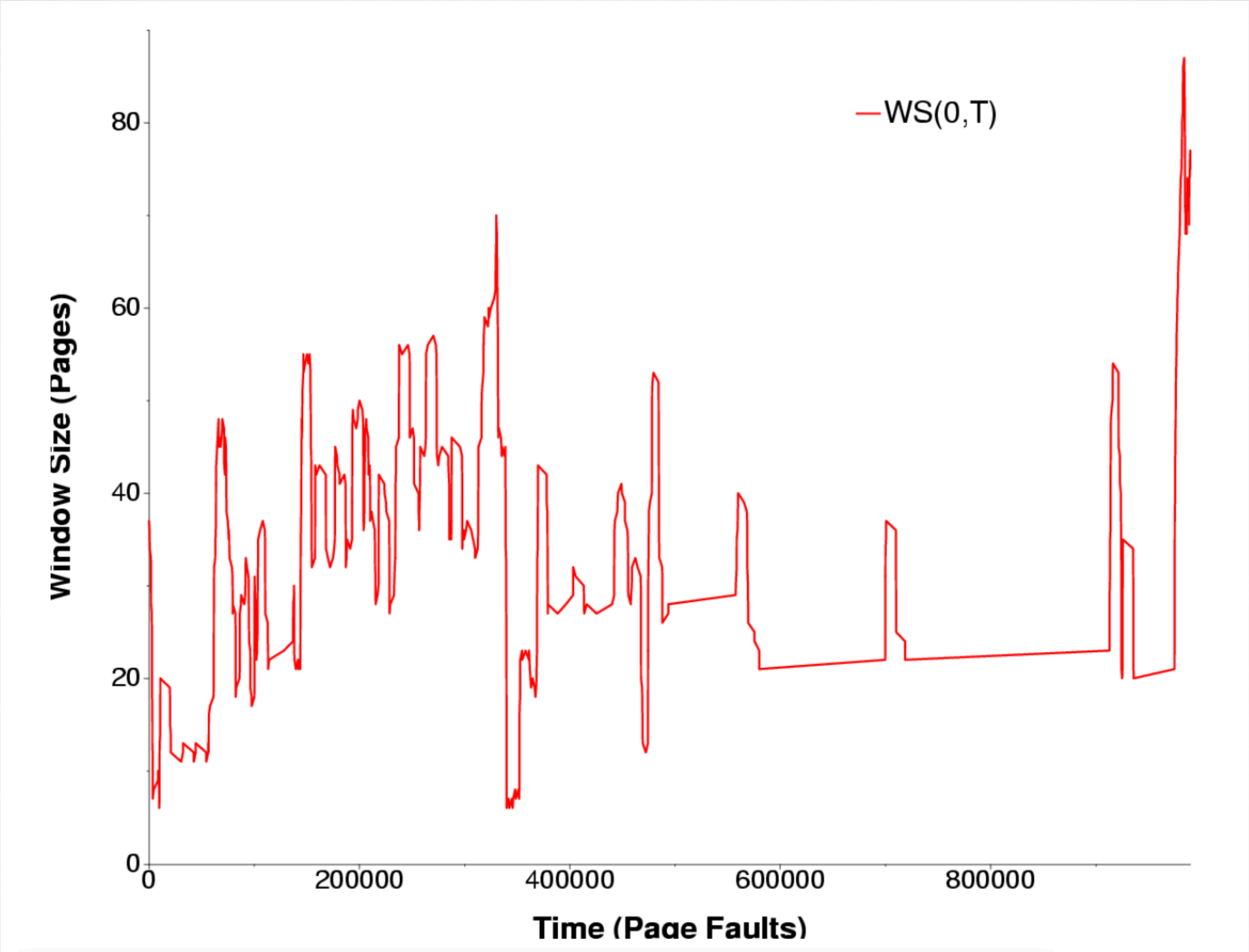
The working set size changes with program locality

- During periods of poor locality, you reference more pages
- Within that period of time, the working set size is larger

Intuitively, want the working set to be the set of pages a process needs in memory to prevent heavy faulting

- Each process has a parameter w that determines a working set with few faults
- Don't run a process unless working set is in memory

Example : gcc working set



Working Set Problems

- Hard to determine w
- Hard to know when the working set changes

➡ However, still used as an abstraction
when people ask, "How much memory does Firefox need?",
they are in effect asking for the size of Firefox's working set

Page Fault Frequency (PFF)

➡ Page Fault Frequency (PFF) is a variable space algorithm that uses a more ad-hoc approach

Monitor the fault rate for each process

- If the fault rate is above a high threshold, give it more memory
- If the fault rate is below a low threshold, take away memory

◉ Hard to use PFF to distinguish between changes in locality and changes in size of working set

Thrashing

Overcommitted system
when OS spent most of the time in paging data back and forth from disk (and so spending little time doing useful work)

- The problem comes from either
  - a bad page replacement algorithm
    (that does not help minimizing page fault)
  - or not enough physical memory for all processes

Windows XP Paging Policy

➡ Local page replacement

- Per-process FIFO
- Processes start with a default of 50 pages
- XP monitors page fault rate and adjusts working-set size accordingly
- On page fault, cluster of pages around the missing page are brought into memory

Linux Paging

- ➡ Global replacement (like most Unix)
  - Modified second-chance clock algorithm
  - Pages age with each pass of the clock hand
  - Pages that are not used for a long time will eventually have a value of zero

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*
- Sina Meraji - teaching CS 369 at *University of Toronto*