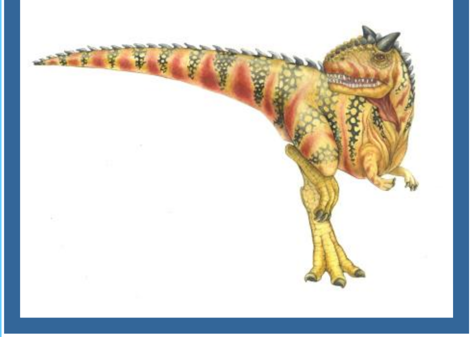
Chapter 9: Virtual-Memory Management



Chapter 9: Virtual-Memory Management

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Background

- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

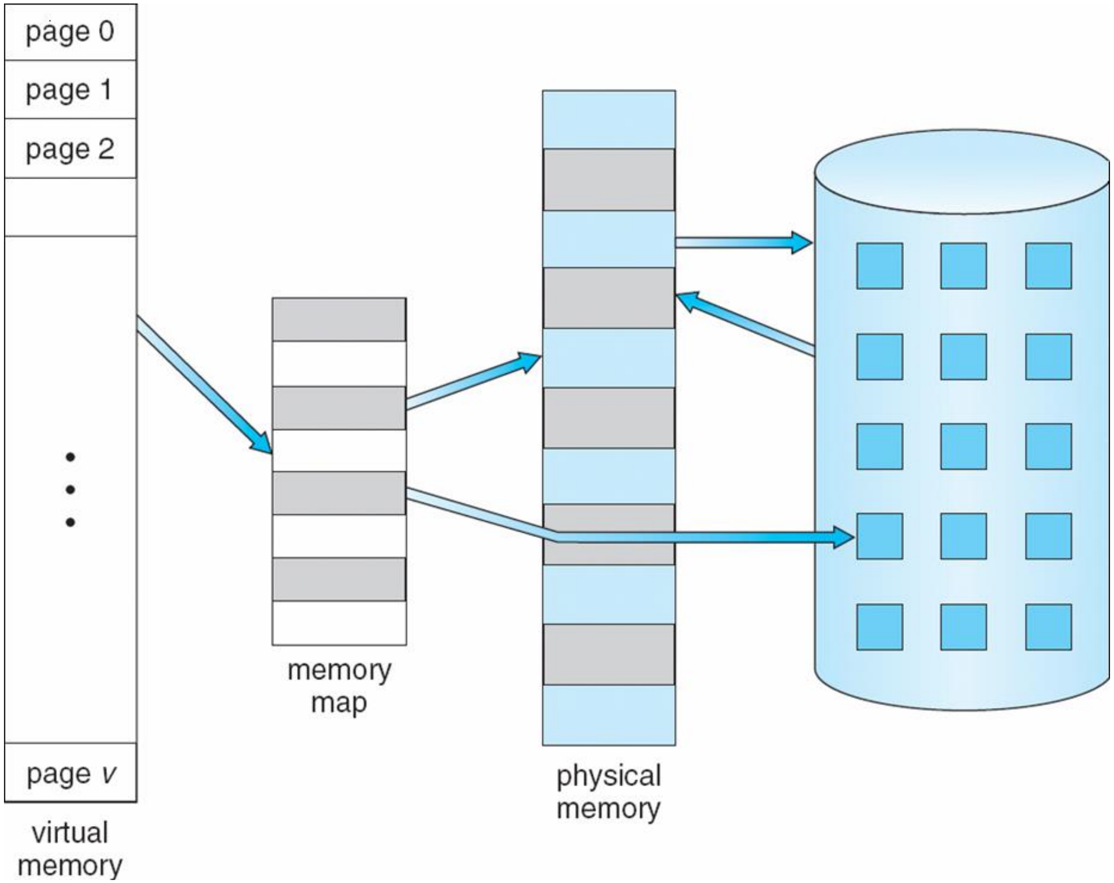
linux1:~

```
(linux1:~) ysw% cat /proc/self/maps
00400000-0040b000 r-xp 00000000 09:03 100663537                          /bin/cat
0060a000-0060b000 r--p 0000a000 09:03 100663537                          /bin/cat
0060b000-0060c000 rw-p 0000b000 09:03 100663537                          /bin/cat
011b4000-011d5000 rw-p 00000000 00:00 0                                  [heap]
7f015911f000-7f015f486000 r--p 00000000 09:03 627058                     /usr/lib64/locale/locale-archive
7f015f486000-7f015f61c000 r-xp 00000000 09:03 37223144                   /lib64/libc-2.15.so
7f015f61c000-7f015f81b000 ---p 00196000 09:03 37223144                   /lib64/libc-2.15.so
7f015f81b000-7f015f81f000 r--p 00195000 09:03 37223144                   /lib64/libc-2.15.so
7f015f81f000-7f015f821000 rw-p 00199000 09:03 37223144                   /lib64/libc-2.15.so
7f015f821000-7f015f825000 rw-p 00000000 00:00 0
7f015f825000-7f015f846000 r-xp 00000000 09:03 37223145                   /lib64/ld-2.15.so
7f015fa29000-7f015fa2c000 rw-p 00000000 00:00 0
7f015fa44000-7f015fa45000 rw-p 00000000 00:00 0
7f015fa45000-7f015fa46000 r--p 00020000 09:03 37223145                   /lib64/ld-2.15.so
7f015fa46000-7f015fa47000 rw-p 00021000 09:03 37223145                   /lib64/ld-2.15.so
7f015fa47000-7f015fa48000 rw-p 00000000 00:00 0
7fffd9108000-7fffd9129000 rw-p 00000000 00:00 0                          [stack]
7fffd91ff000-7fffd9200000 r-xp 00000000 00:00 0                          [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0                  [vsyscall]
(linux1:~) ysw%
```

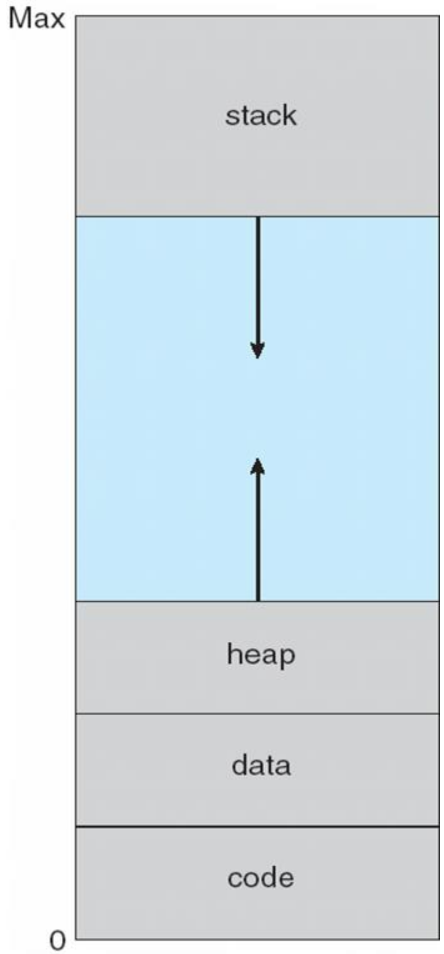
127 TB

16777216 TB

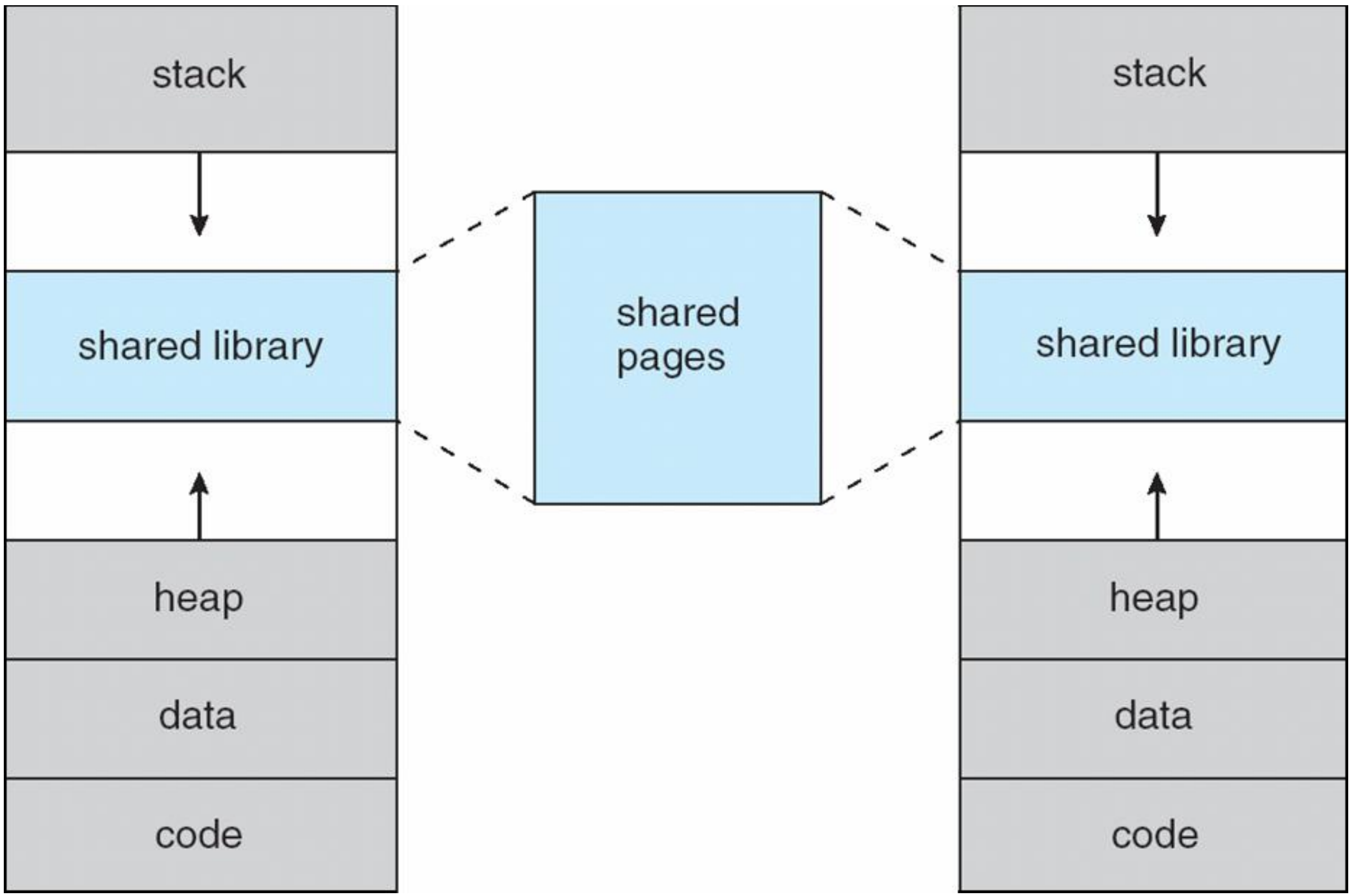
Virtual Memory That is Larger Than Physical Memory



Virtual-address Space



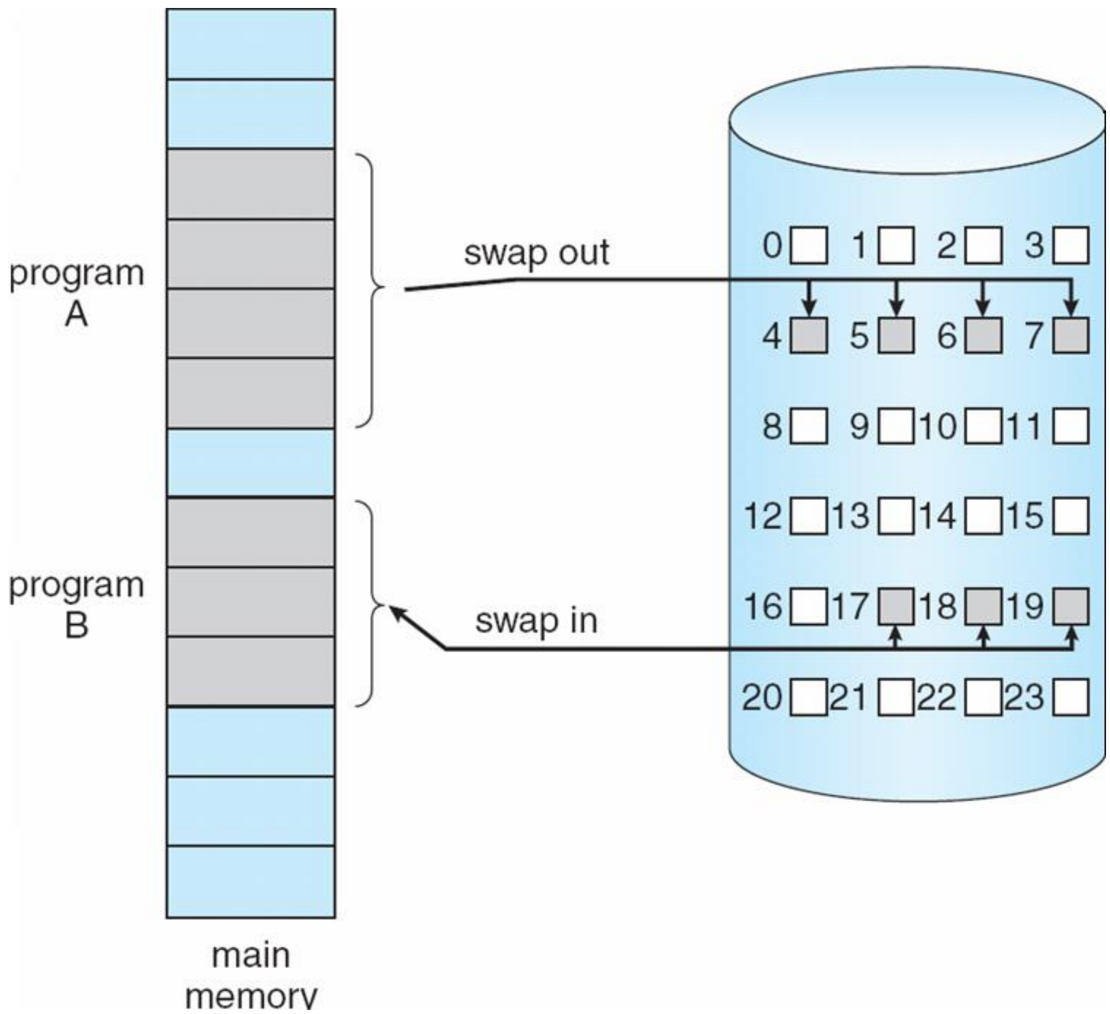
Shared Library Using Virtual Memory



Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space



Valid-Invalid Bit

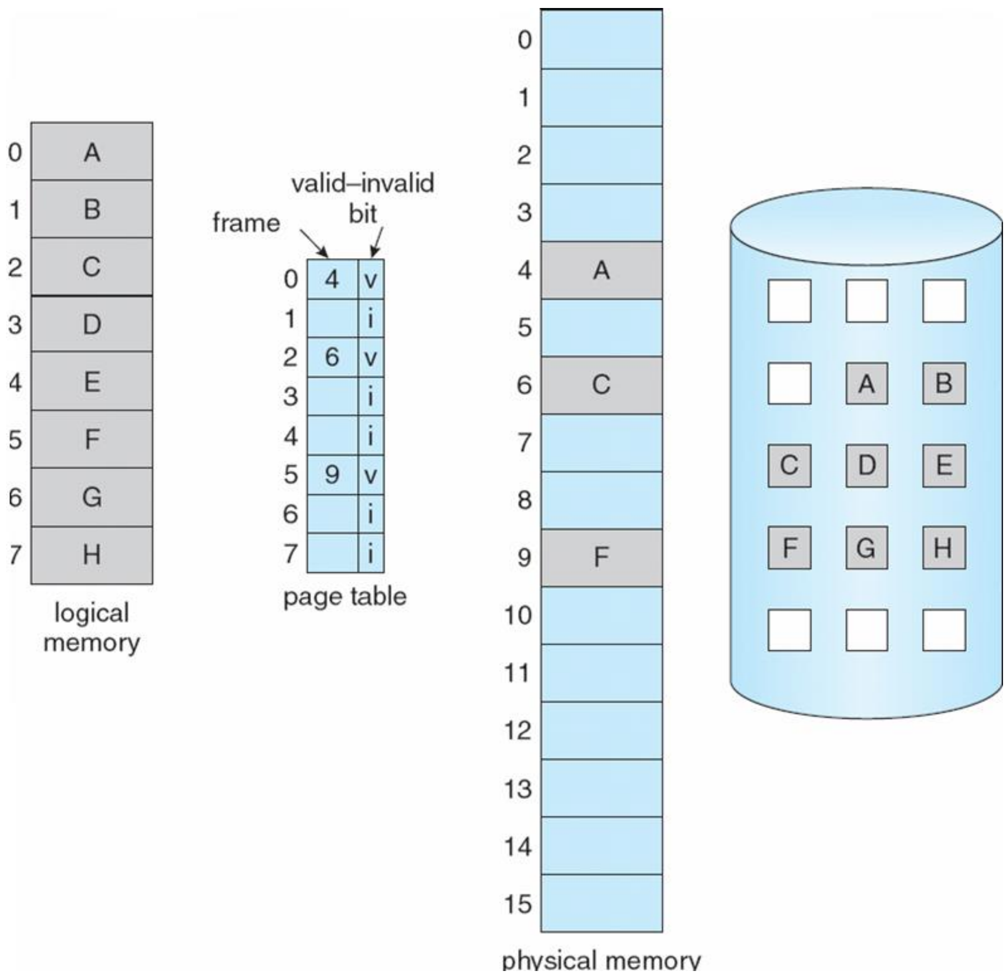
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

Frame #	valid-invalid bit
	v
	v
	v
	v
	i
….	
	i
	i

page table

- During address translation, if valid–invalid bit in page table entry
  is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory



Page Fault

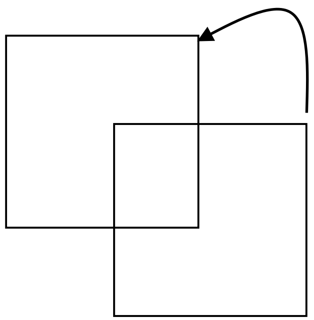
- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

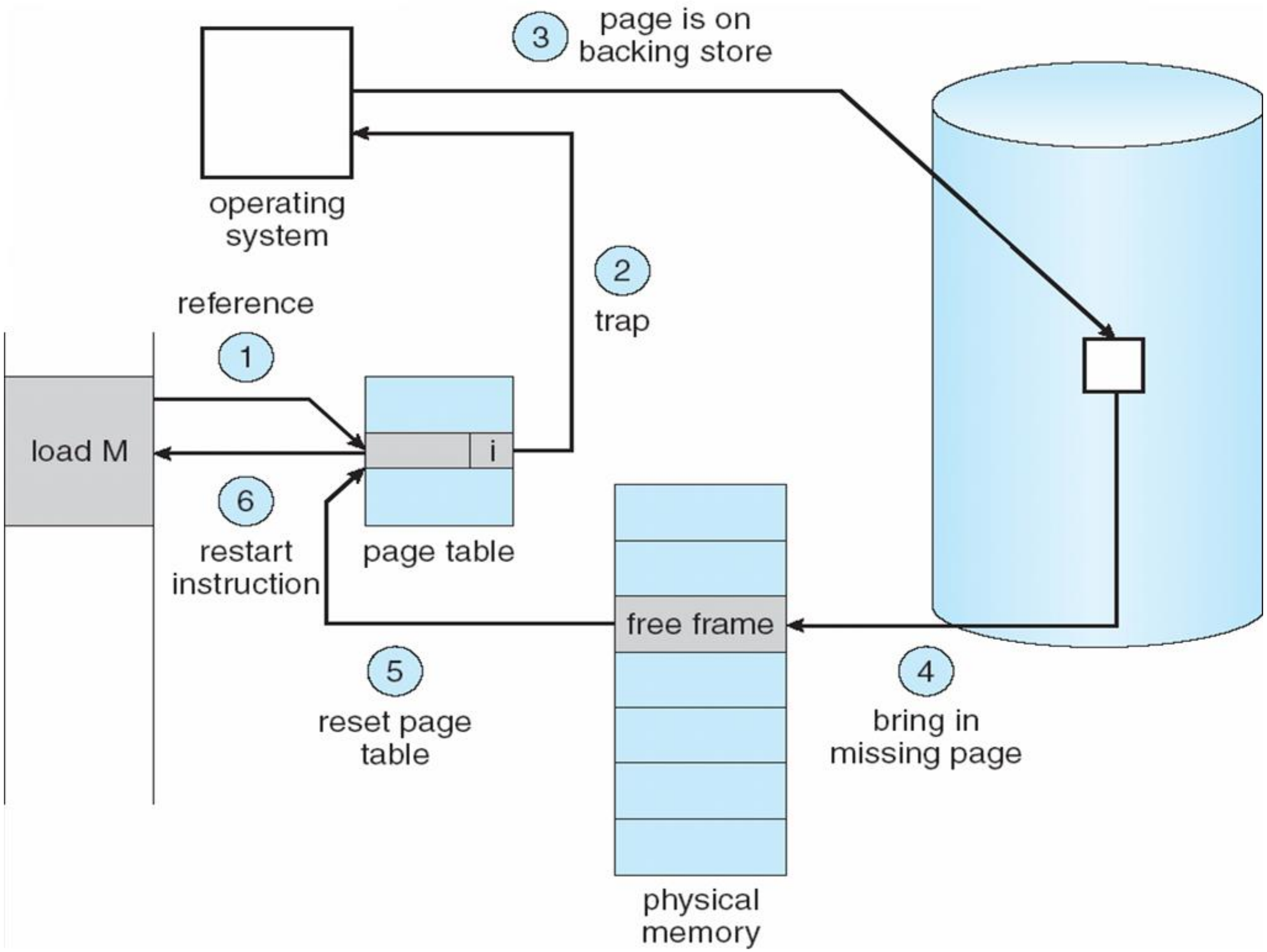
Page Fault (Cont.)

- Restart instruction
  - block move



  - auto increment/decrement location

Steps in Handling a Page Fault



Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$
\begin{aligned}
\text{EAT} = (1 - p) &\times \text{memory access} \\
&+ p\,(\text{page fault overhead} \\
&\qquad + \text{swap page out} \\
&\qquad + \text{swap page in} \\
&\qquad + \text{restart overhead} \\
&\quad)
\end{aligned}
$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!

Process Creation

- Virtual memory allows other benefits during process creation:

  - Copy-on-Write

  - Memory-Mapped Files (later)

Copy-on-Write

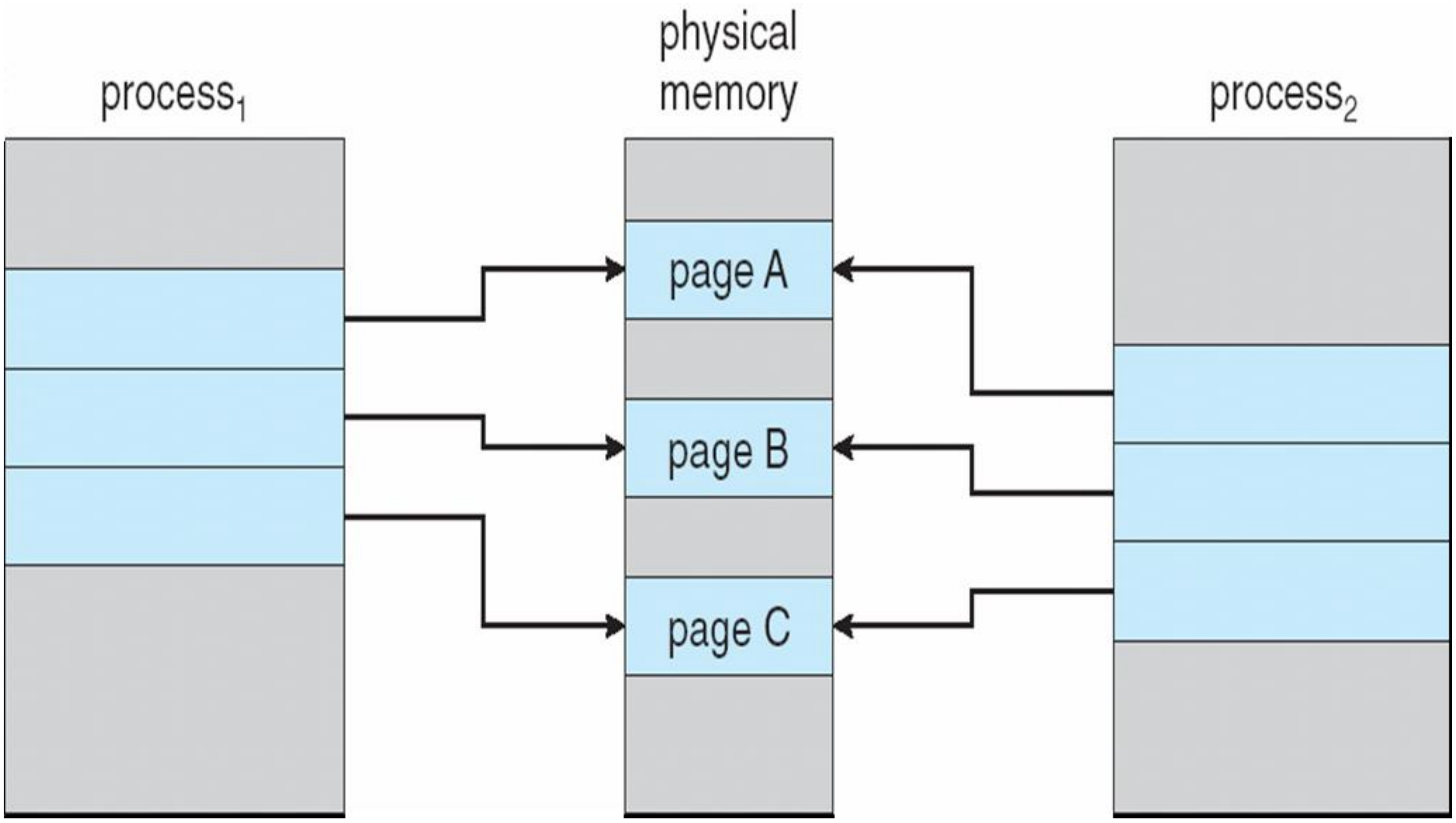
- Copy-on-Write (COW) allows both parent and child processes to initially *share* the same pages in memory

  If either process modifies a shared page, only then is the page copied

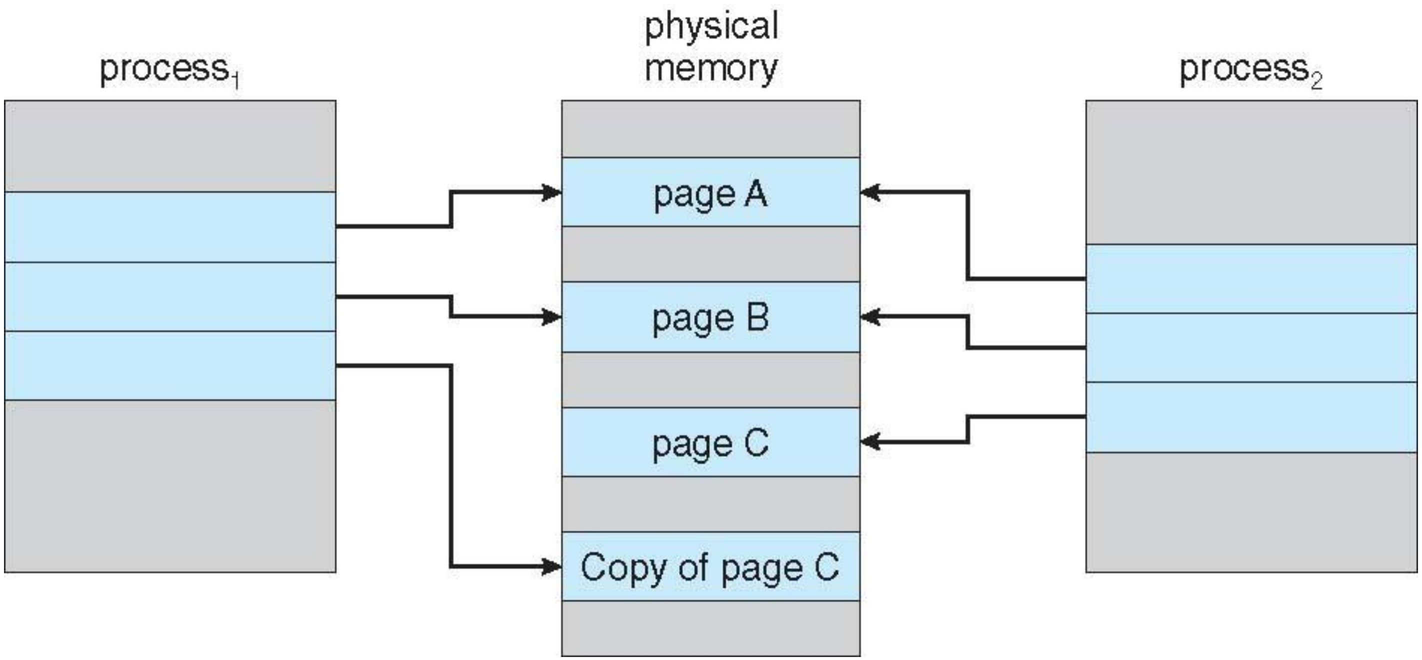
- COW allows more efficient process creation as only modified pages are copied

- Free pages are allocated from a **pool** of zeroed-out pages

Before Process 1 Modifies Page C



After Process 1 Modifies Page C



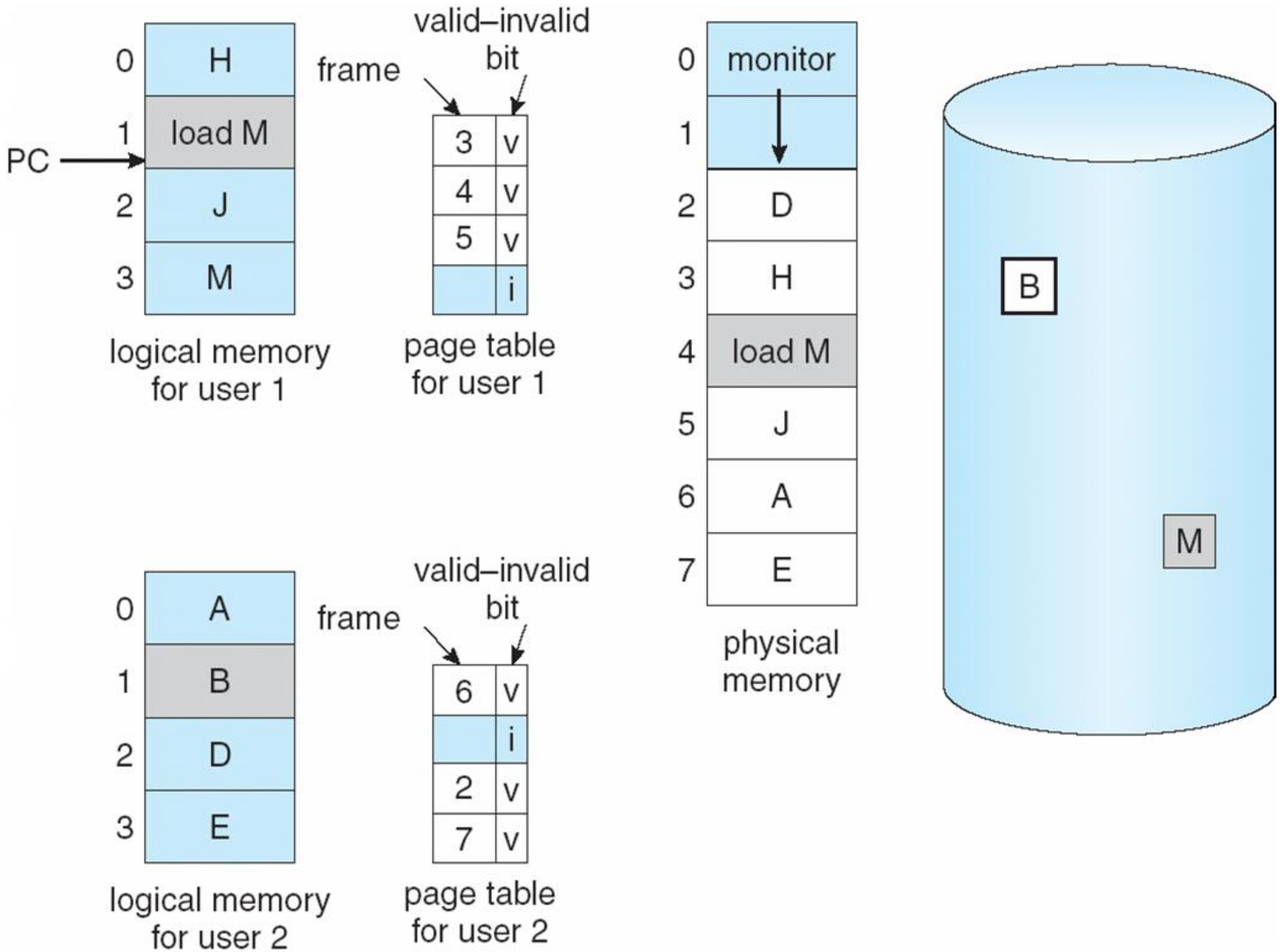
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Basic Page Replacement

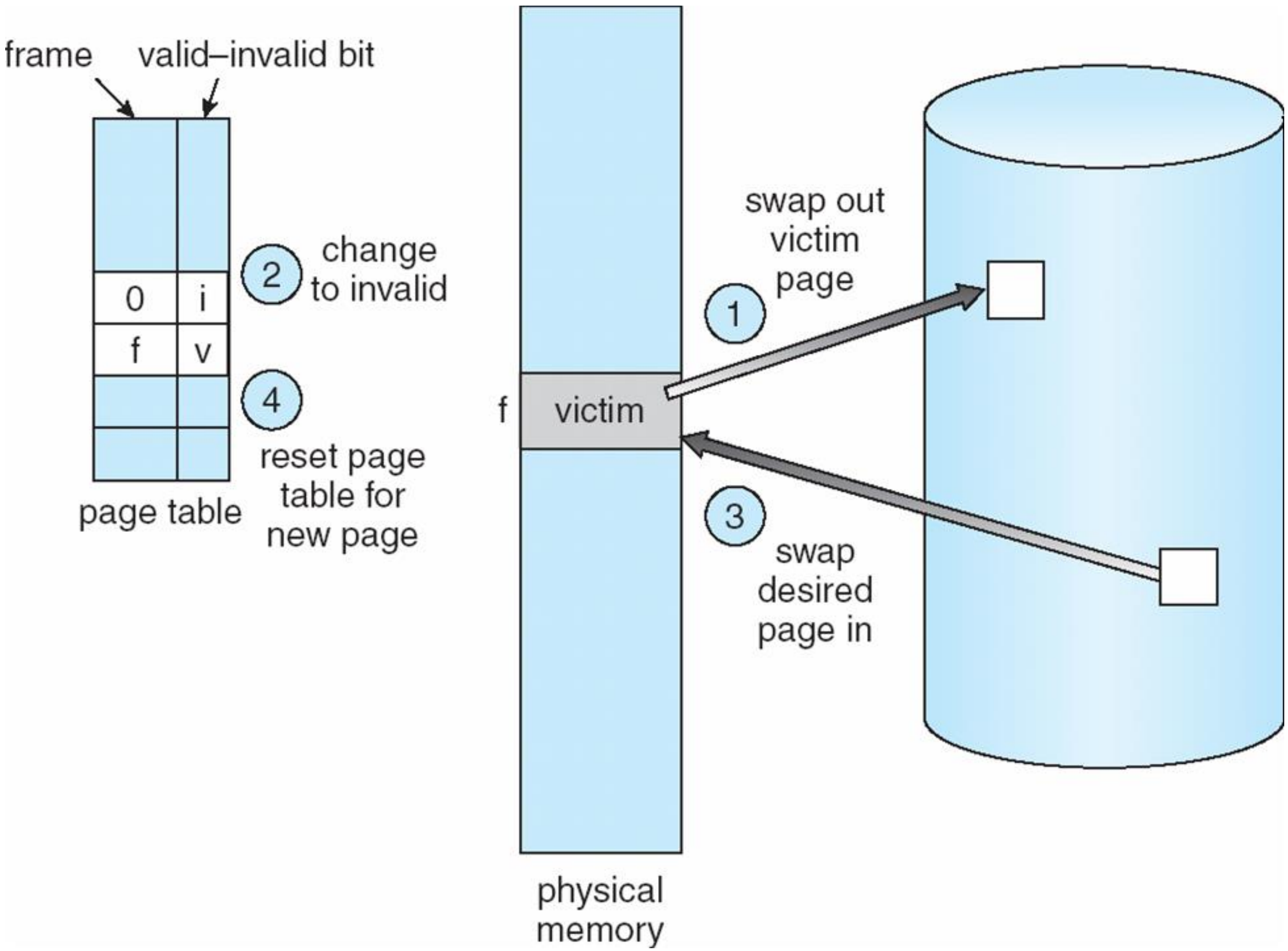
1. Find the location of the desired page on disk

2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim** frame

3. Bring the desired page into the (newly) free frame; update the page and frame tables

4. Restart the process

Page Replacement

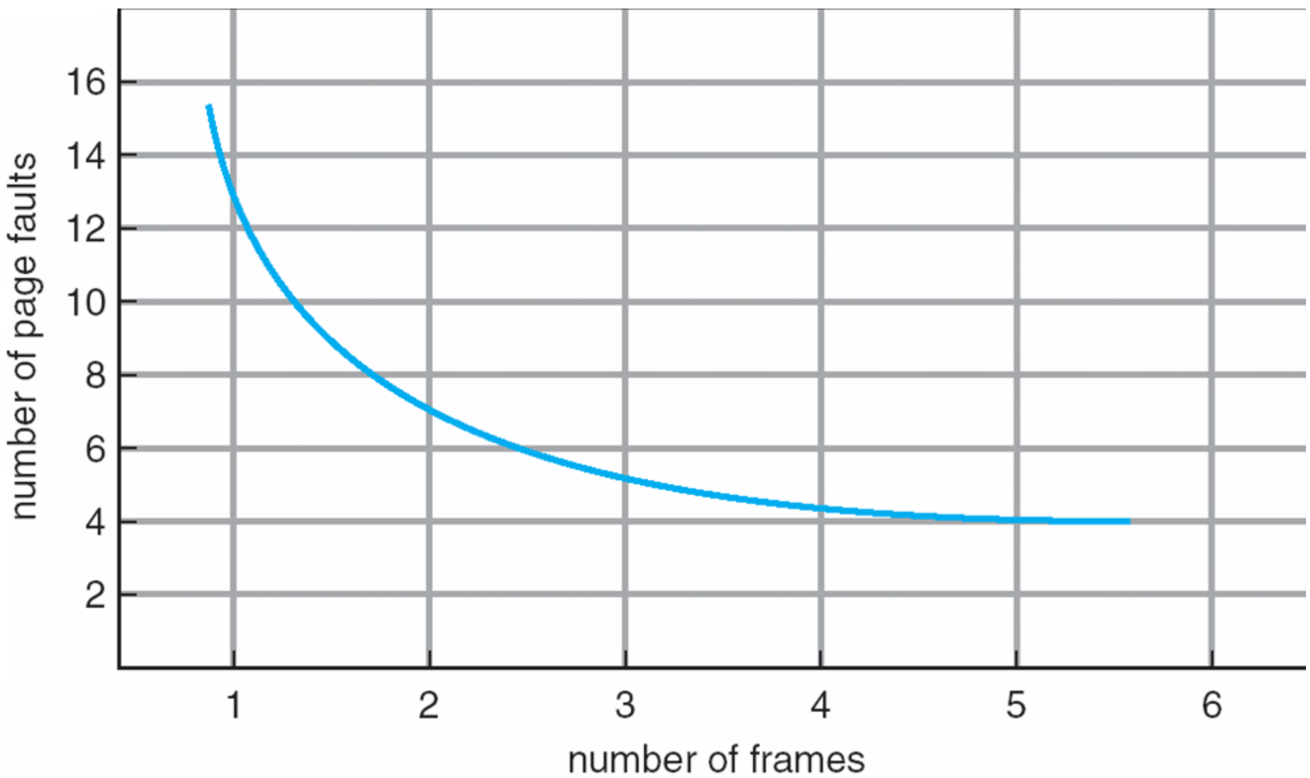


Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference string is

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Graph of Page Faults Versus The Number of Frames



- One desirable property: When you add memory, # of page faults goes down
  - Does this always happen?
  - Seems like it should, right?

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

1	1	4	5
2	2	1	3
3	3	2	4

9 page faults

- 4 frames

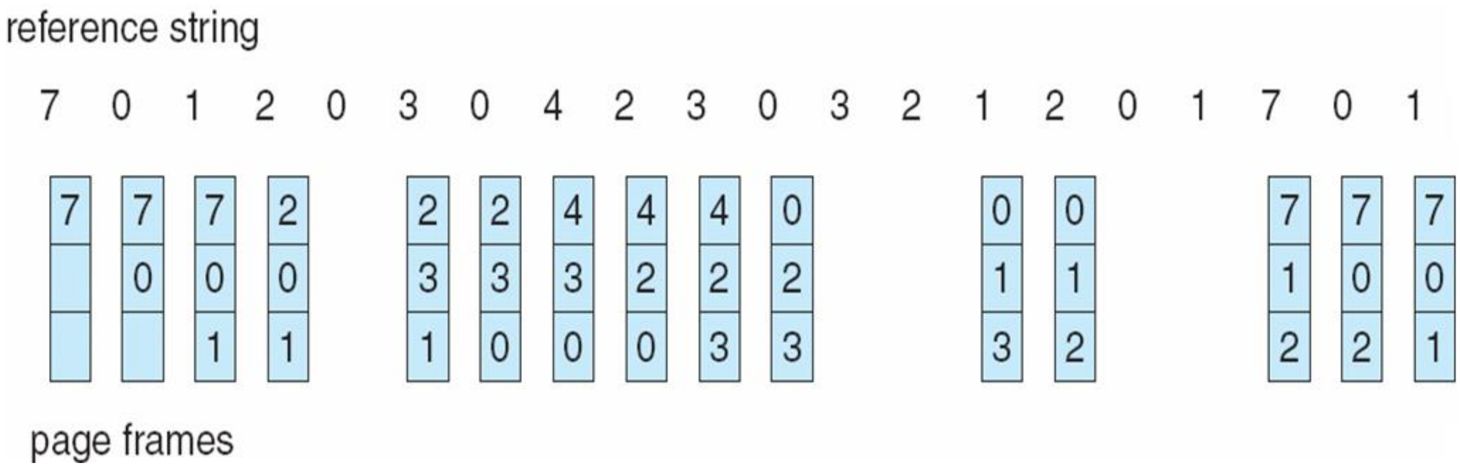
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	

10 page faults

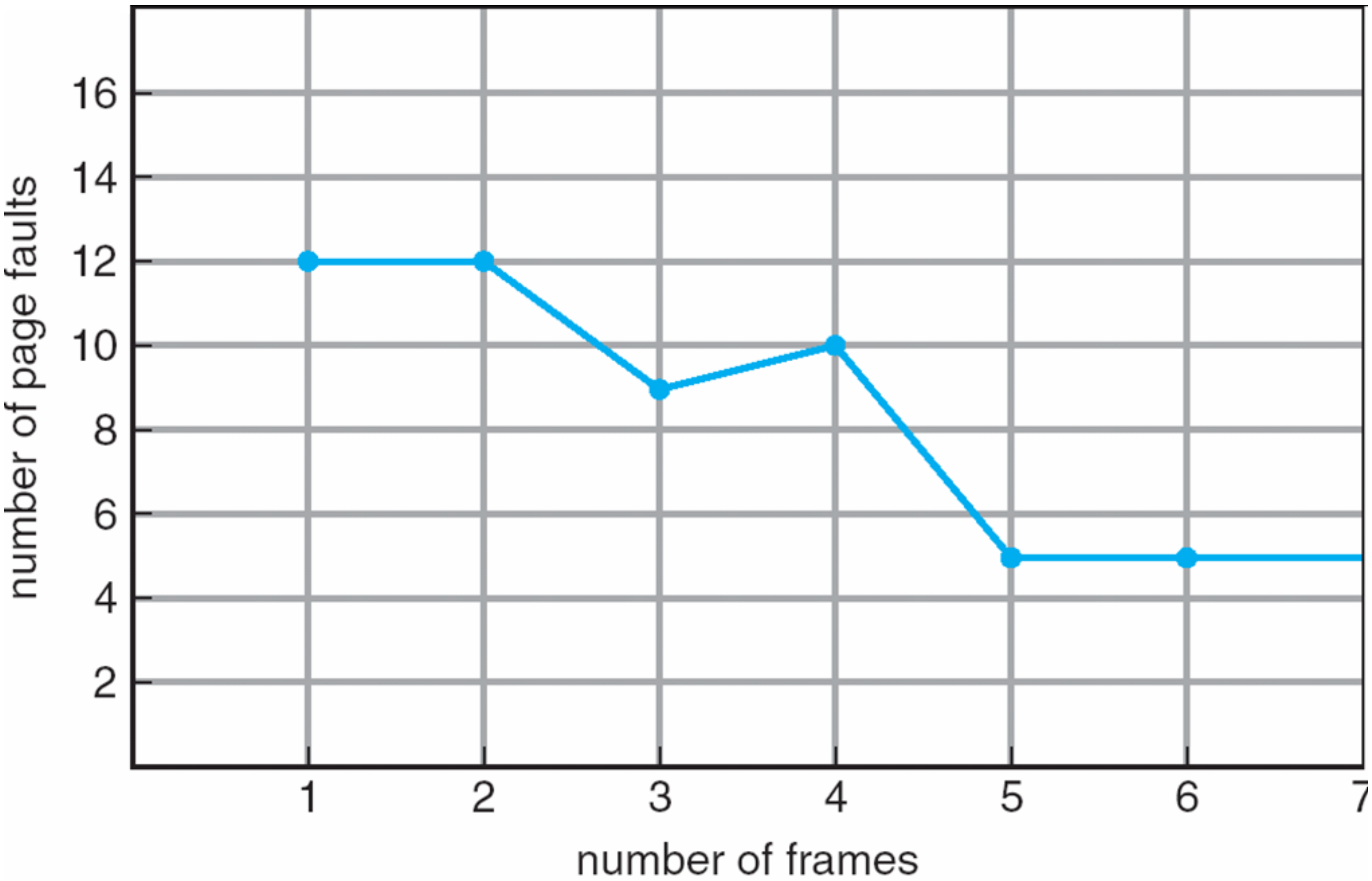
With FIFO, contents can be completely different after adding memory

- Belady's Anomaly: more frames ⇒ more page faults

FIFO Page Replacement



FIFO Illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that will not be used for longest period of time
- 4 frames example

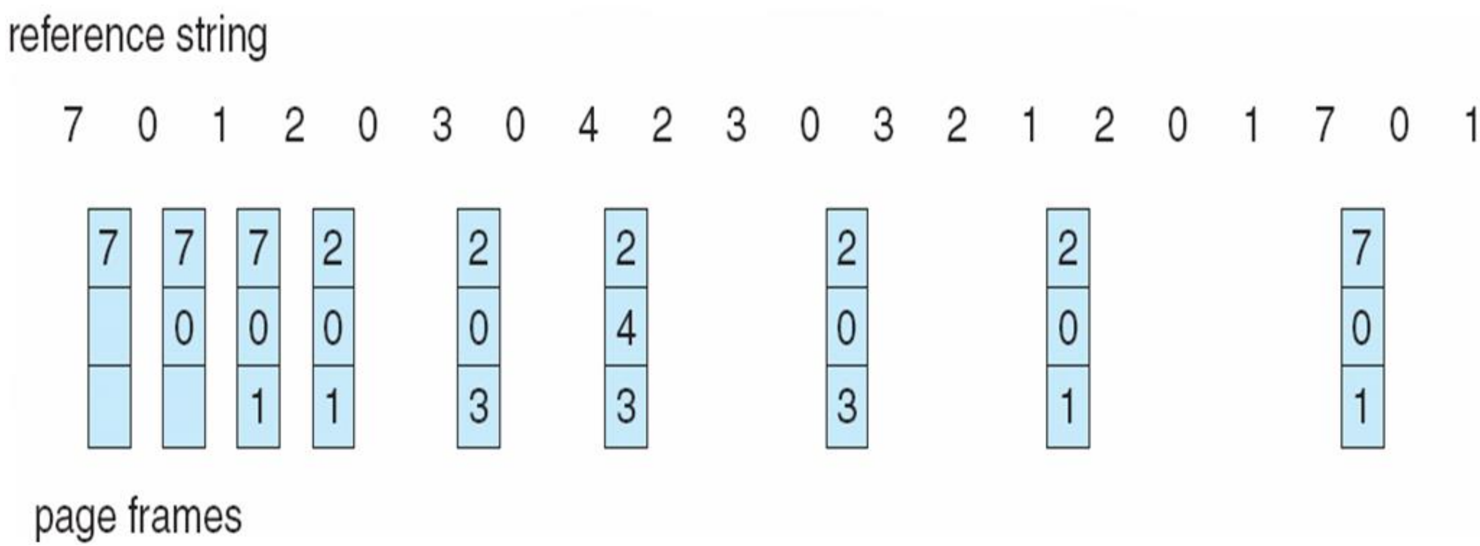
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	4
2	
3	
4	5

6 page faults

- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



With OPTIMAL, contents of memory with X pages are a subset of contents with X+1 pages after adding memory

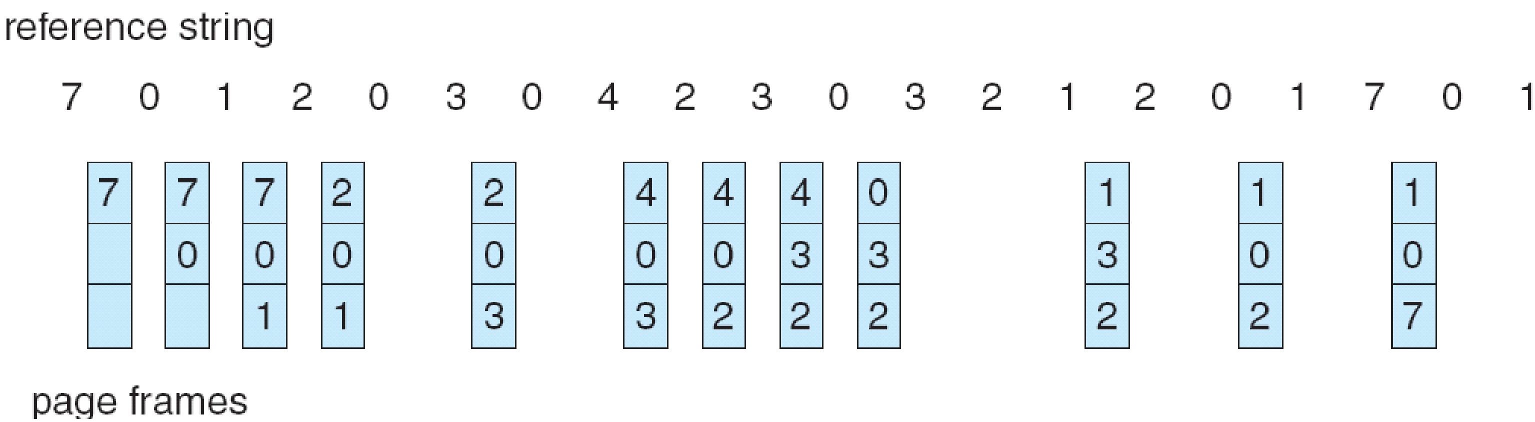
Least Recently Used (LRU) Algorithm

- Assumption: A page that has not been referenced for the longest time would wait for the longest time to be accessed again
  - Use of known history to predict unknown future
  - Does the intuition really work?
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

LRU Page Replacement

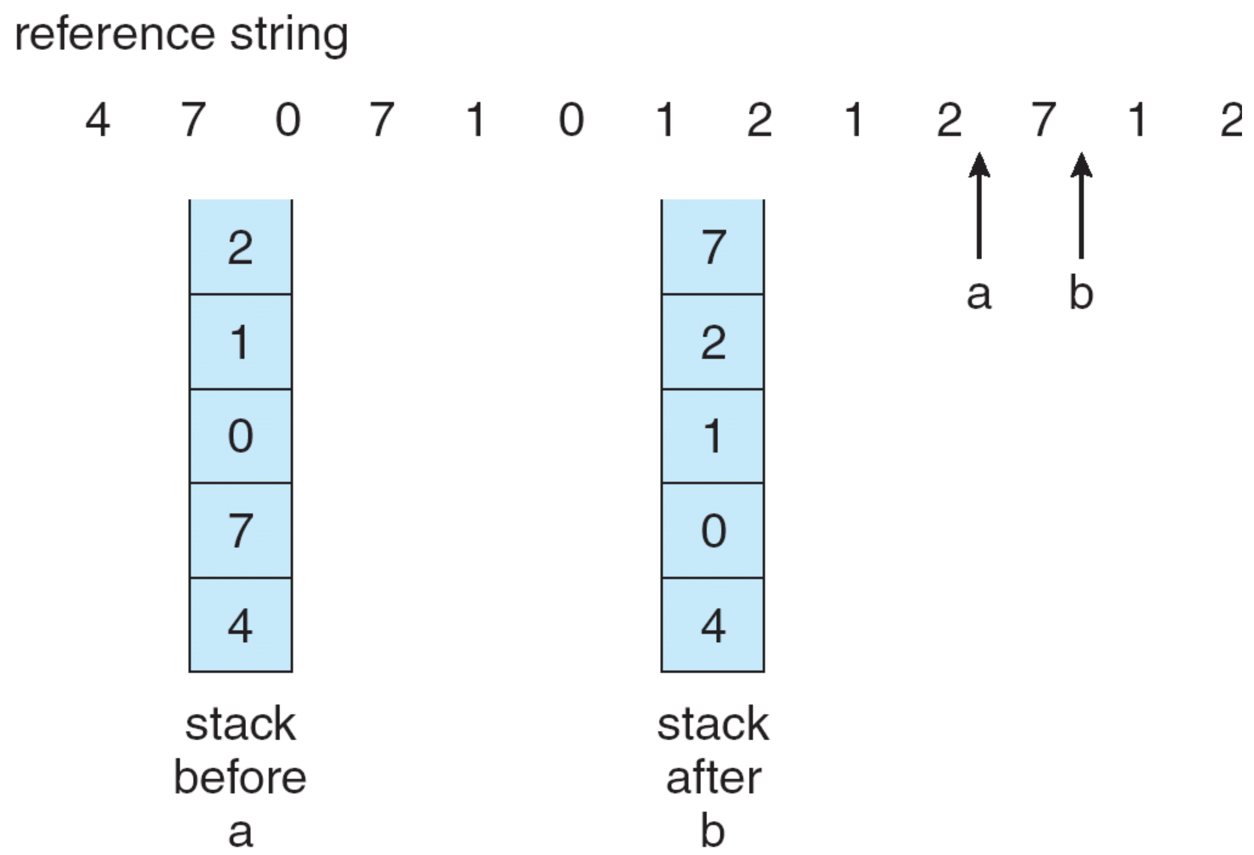


With LRU, contents of memory with X pages are a subset of contents with X+1 pages after adding memory

LRU Algorithm (Cont.)

- Stack implementation – keep a stack of page numbers in a double link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement

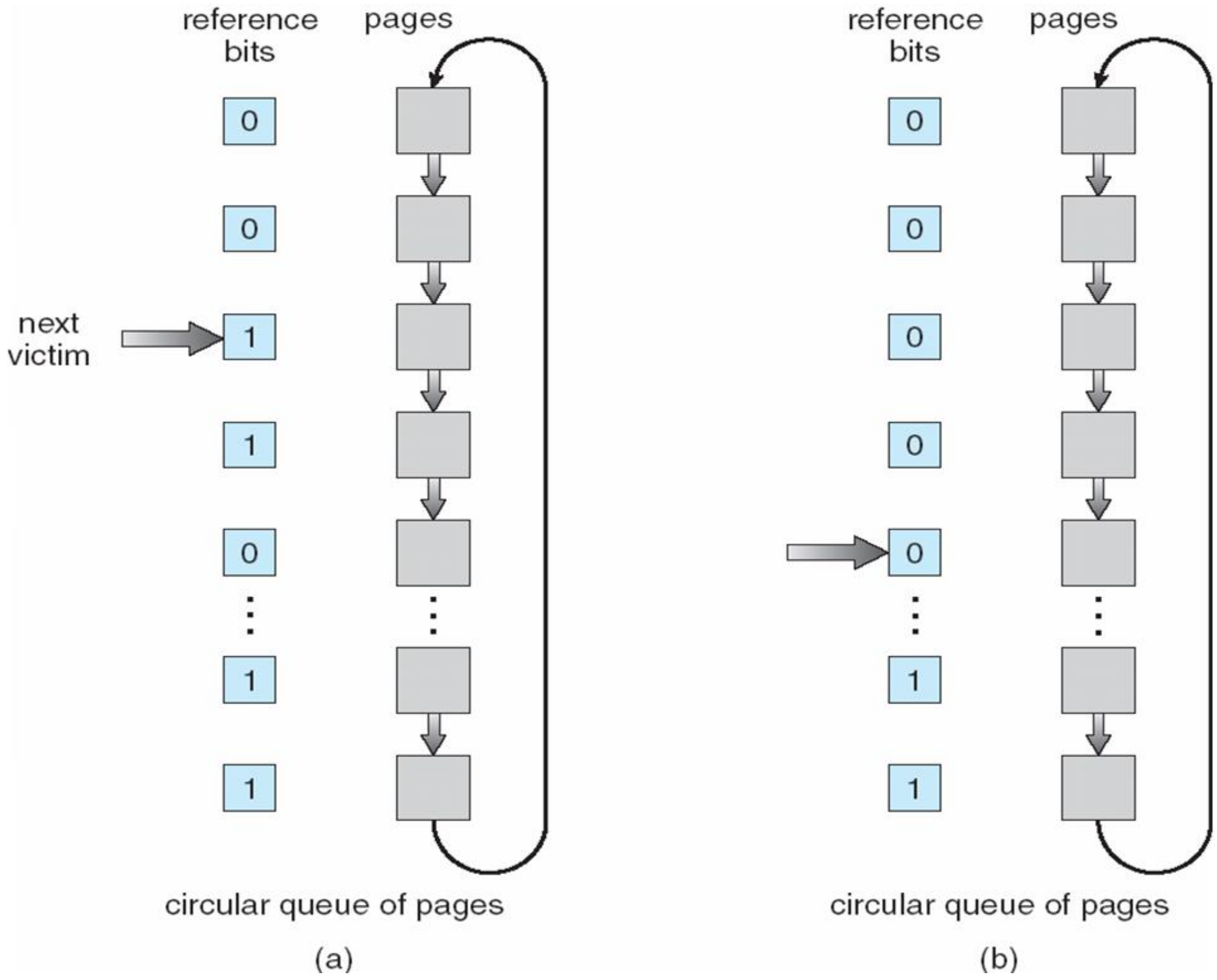
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Both timestamp and stack implementations are too expensive to implement in practice
- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
- Second chance
  - Need reference bit
  - Clock replacement
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



Second-Chance (clock) Page-Replacement Algorithm

- What if all reference bits are set?
  - The original victim will be selected after looping around
  - The clock algorithm degenerates into FIFO

- The clock algorithm replaces an old page, not the oldest page
- What if hand moves slowly?
  - Not many page faults
  - Or, victim page can be found quickly
- What if hand moves quickly?
  - Lots of page faults
  - Or, lots of reference bits set
- One way to view clock algorithm
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks reference bit:
    - 1⇒clear reference bit 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 approximation 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

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Allocation of Frames

- Each process needs *minimum* number of pages
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- Two major allocation schemes
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – For example, if there are 100 frames and 5 processes, give each process 20 frames.
- Proportional allocation – Allocate according to the size of process

$$-\ s_i = \text{size of process } p_i$$

$$-\ S = \sum s_i$$

$$-\ m = \text{total number of frames}$$

$$-\ a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process P_i generates a page fault,
  - select for replacement one of its frames
  - select for replacement a frame from a process with lower priority number

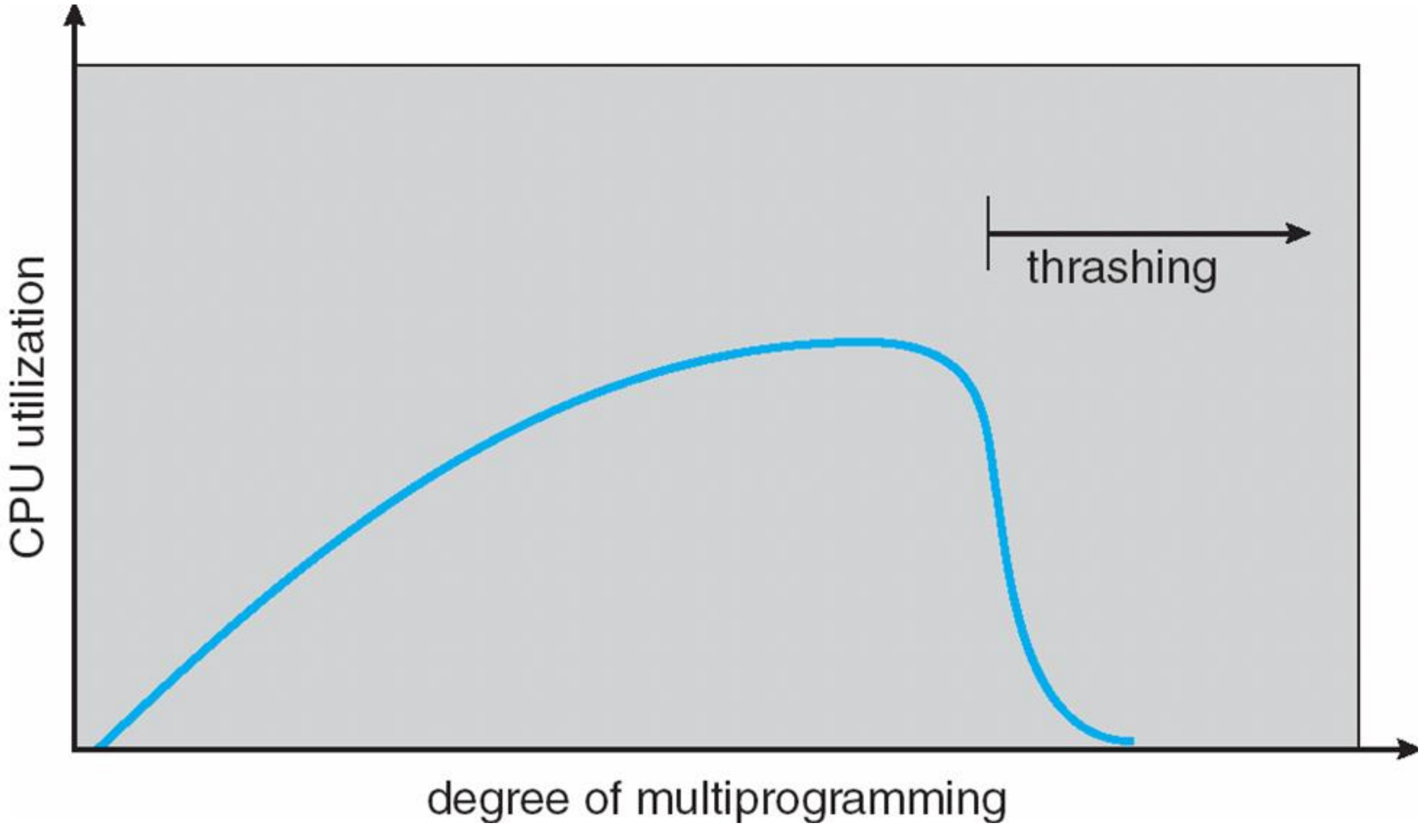
Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
- **Local replacement** – each process selects from only its own set of allocated frames

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
  - Even worse thrashing
- **Thrashing** ≡ a system experiences a high degree of paging activity
  - Each process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

- Pages should only be brought into memory if the executing process demands them
  - As opposed to anticipatory paging
  - Many page faults will occur until most of a process's working set of pages is located in physical memory
- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap

Locality In A Memory-Reference Pattern

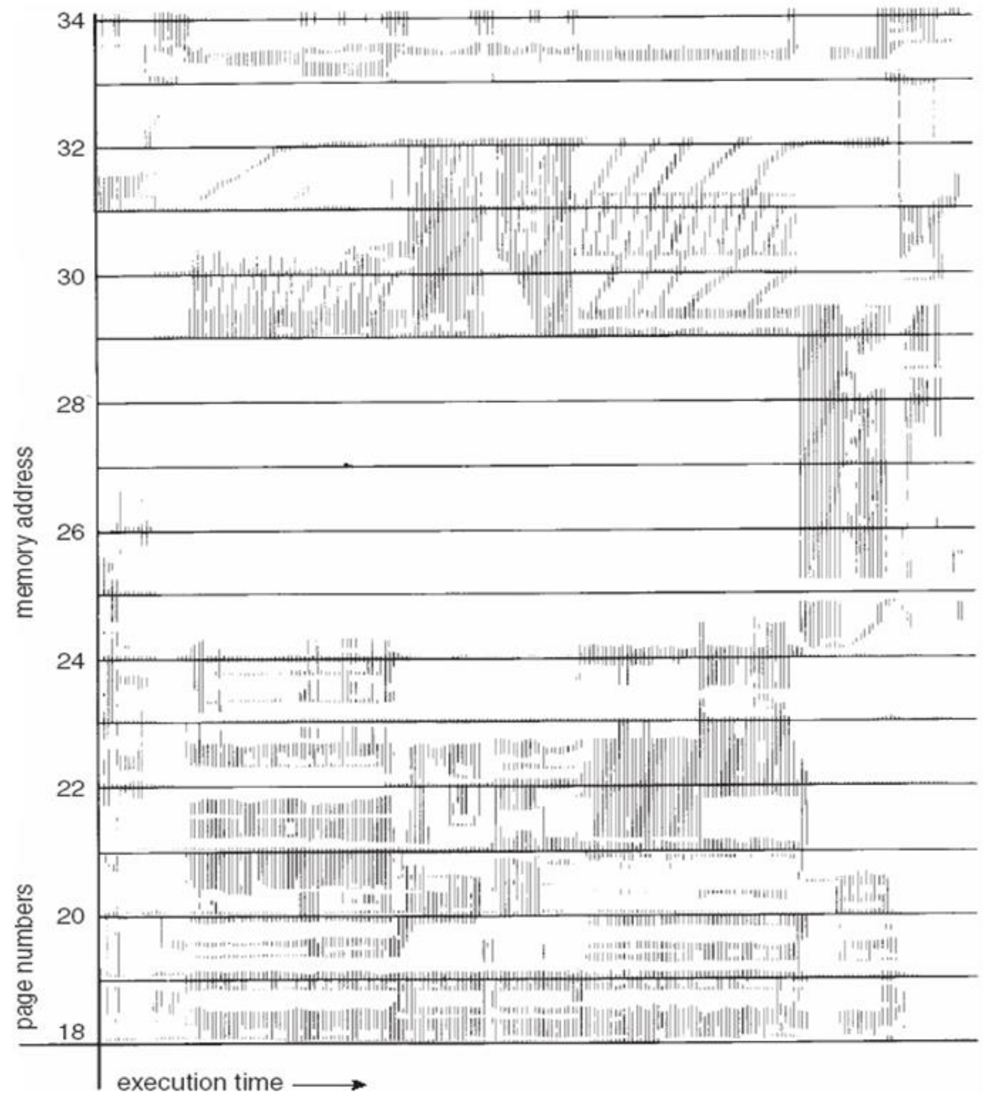
why does paging work?

locality model

- a locality is a set of pages that are actively used together
- program is composed of several different localities, which may overlap
- process migrates from one locality to another.
- if we allocate enough frames to accommodate the locality, faults will only occur when transitioning

why does thrashing occur?

Σ locality sizes > total memory size



Demand Paging and Thrashing

- Why does thrashing occur?
  size of locality > total memory size
  - Could be a process is too large for memory
    - There is nothing the OS can do
  - Could also be the sum of several processes is too large
    - Figure out how much memory each process needs and schedule them accordingly
- Thrashing occurs because the system doesn't know when it has taken on more work than it can handle
- LRU-type mechanism, such as clock, order pages in terms of last access, but don't give absolute numbers indicating pages that mustn't be thrown out

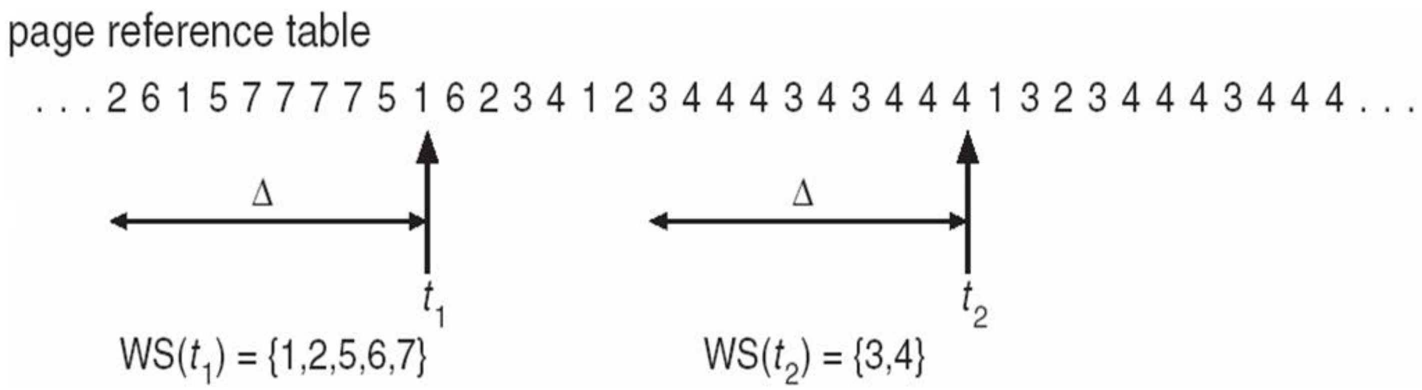
Working-Set Model

- A conceptual model by Peter Denning to prevent thrashing
- Informal definition of working set
  - The collection of pages that a process is working with, and which must thus be resident if the process is to avoid thrashing
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSS_i (working set of Process P_i) =
  total number 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 $\Delta = \infty \Rightarrow$ will encompass entire program

Working-Set Model

- Use the recent needs of a process to predict its future needs
  - At any given time, all pages referenced by a process in the last working-set window are considered to comprise its working set
- A process will never be executed unless its working set is resident in memory
- Pages outside the working set may be swapped out at any time
- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend one of the processes (swap it out if needed)

Working-set model

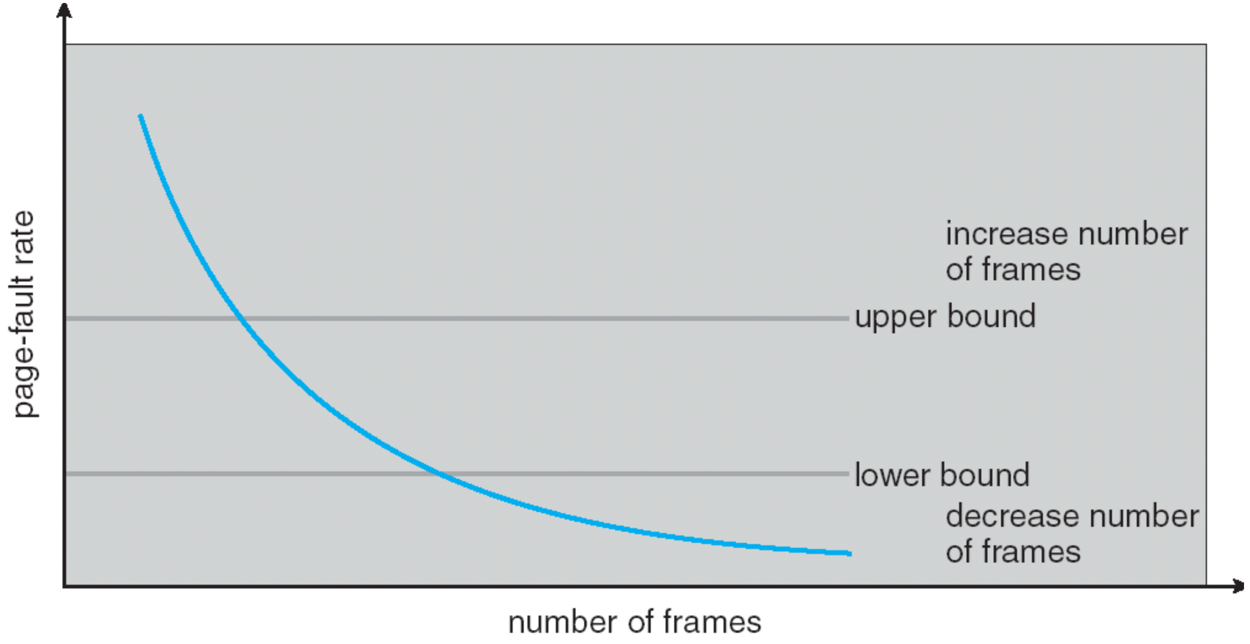


Keeping Track of the Working Set

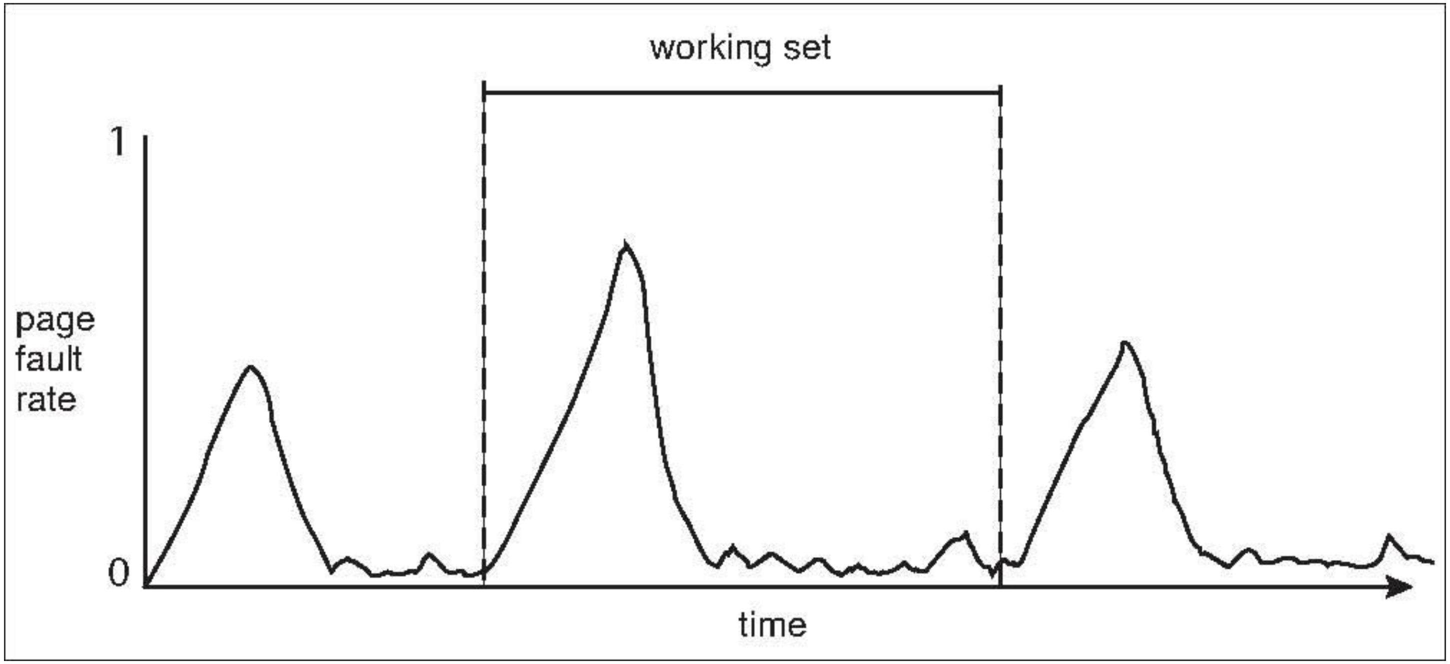
- How to determine when a page was last accessed?
- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

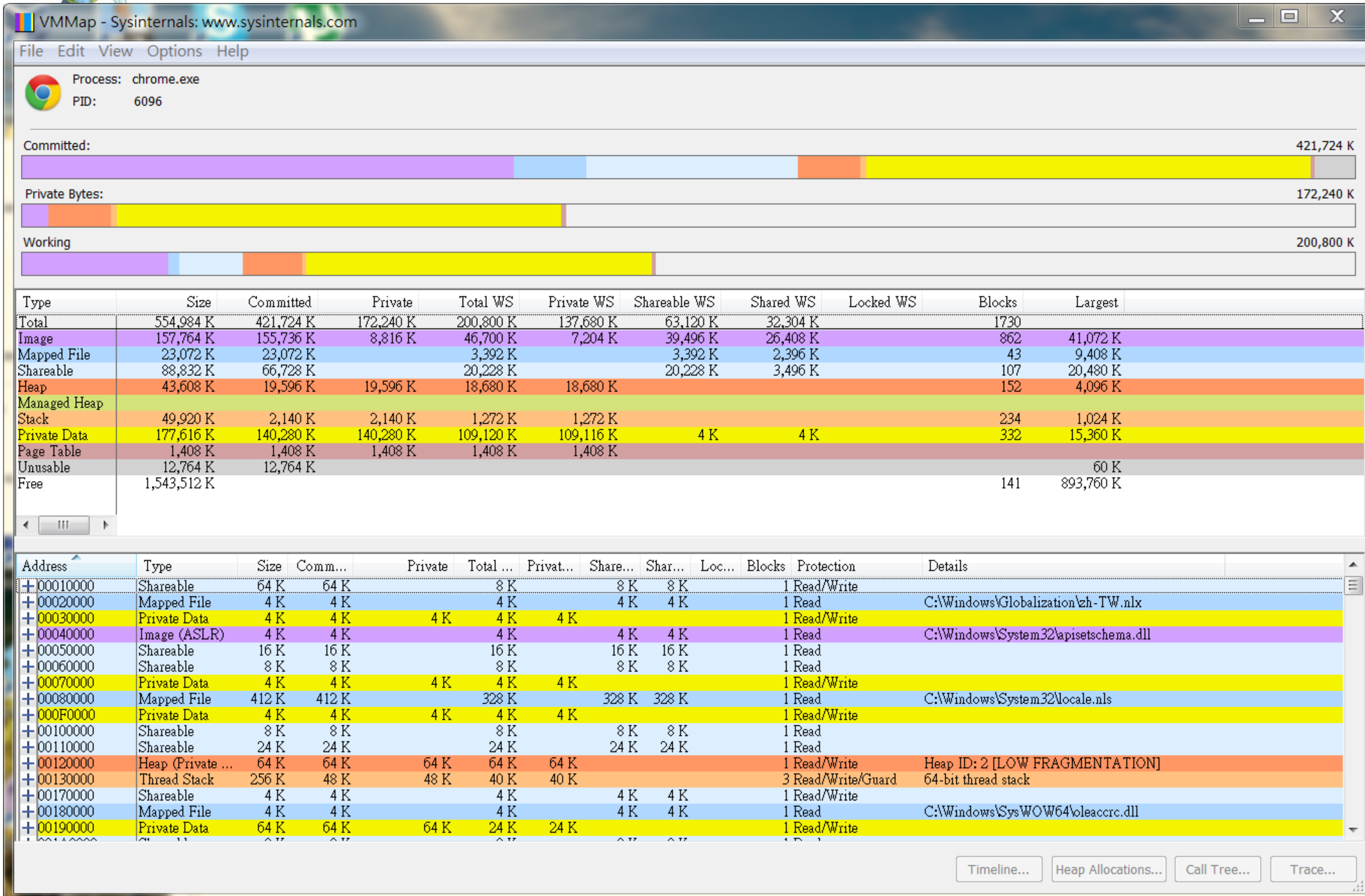
Page-Fault Frequency Scheme

- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
    - Suspend processes to decrease degree of multiprocessing
  - If actual rate too high, process gains frame
    - Restart suspended processes to increase degree of multiprocessing



Working Sets and Page Fault Rates





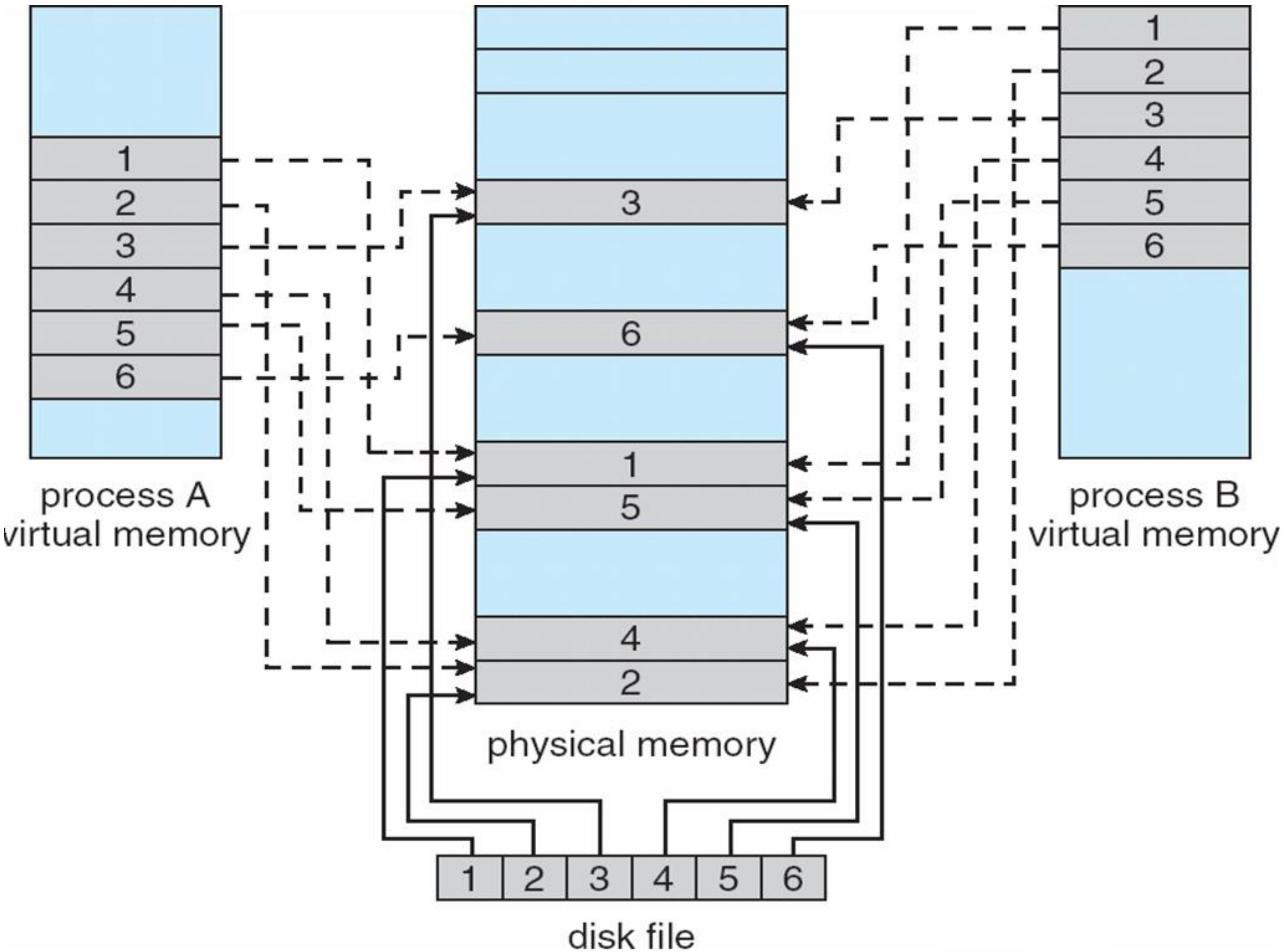
Type	Size	Committed	Private	Total WS	Private WS	Shareable WS	Shared WS	Locked WS	Blocks	Largest
Total	554,984 K	421,724 K	172,240 K	200,800 K	137,680 K	63,120 K	32,304 K		1730	
Image	157,764 K	155,736 K	8,816 K	46,700 K	7,204 K	39,496 K	26,408 K		862	41,072 K
Mapped File	23,072 K	23,072 K		3,392 K		3,392 K	2,396 K		43	9,408 K
Shareable	88,832 K	66,728 K		20,228 K		20,228 K	3,496 K		107	20,480 K
Heap	43,608 K	19,596 K	19,596 K	18,680 K	18,680 K				152	4,096 K
Managed Heap										
Stack	49,920 K	2,140 K	2,140 K	1,272 K	1,272 K				234	1,024 K
Private Data	177,616 K	140,280 K	140,280 K	109,120 K	109,116 K	4 K	4 K		332	15,360 K
Page Table	1,408 K	1,408 K	1,408 K	1,408 K	1,408 K					
Unusable	12,764 K	12,764 K								60 K
Free	1,543,512 K								141	893,760 K

Address	Type	Size	Comm...	Private	Total ...	Privat...	Share...	Shar...	Loc...	Blocks	Protection	Details
00010000	Shareable	64 K	64 K		8 K		8 K	8 K		1	Read/Write	
00020000	Mapped File	4 K	4 K		4 K		4 K	4 K		1	Read	C:\Windows\Globalization\zh-TW.nlx
00030000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00040000	Image (ASLR)	4 K	4 K		4 K		4 K	4 K		1	Read	C:\Windows\System32\apisetschema.dll
00050000	Shareable	16 K	16 K		16 K		16 K	16 K		1	Read	
00060000	Shareable	8 K	8 K		8 K		8 K	8 K		1	Read	
00070000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00080000	Mapped File	412 K	412 K		328 K		328 K	328 K		1	Read	C:\Windows\System32\locale.nls
000F0000	Private Data	4 K	4 K	4 K	4 K	4 K				1	Read/Write	
00100000	Shareable	8 K	8 K		8 K		8 K	8 K		1	Read	
00110000	Shareable	24 K	24 K		24 K		24 K	24 K		1	Read	
00120000	Heap (Private ...	64 K	64 K	64 K	64 K	64 K				1	Read/Write	Heap ID: 2 [LOW FRAGMENTATION]
00130000	Thread Stack	256 K	48 K	48 K	40 K	40 K				3	Read/Write/Guard	64-bit thread stack
00170000	Shareable	4 K	4 K		4 K		4 K	4 K		1	Read/Write	
00180000	Mapped File	4 K	4 K		4 K		4 K	4 K		1	Read	C:\Windows\SysWOW64\oleaccrc.dll
00190000	Private Data	64 K	64 K	64 K	24 K	24 K				1	Read/Write	

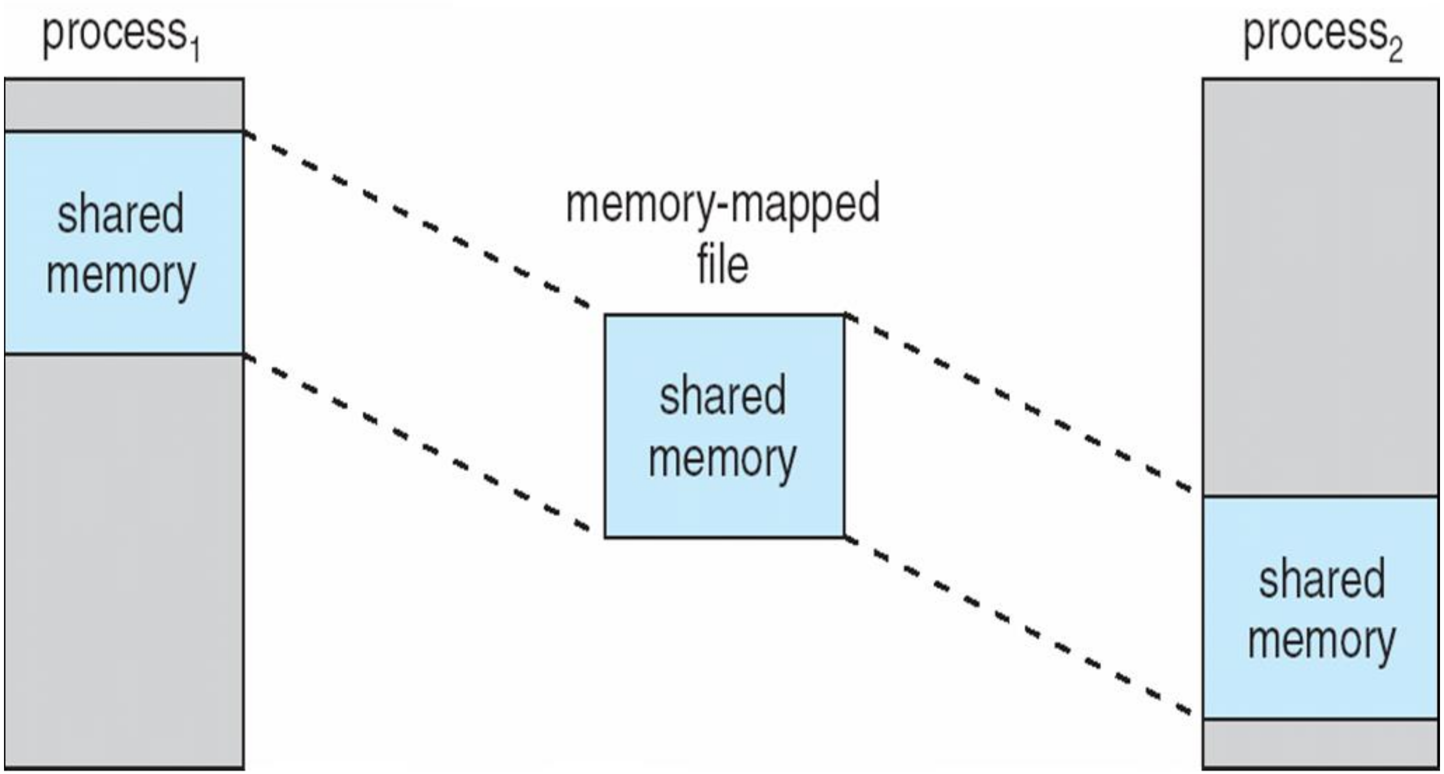
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file are treated as ordinary memory accesses.
- Simplifies file access by treating file I/O through memory rather than **`read()`** **`write()`** system calls
- Also allows several processes to map the same file allowing the pages in memory to be shared

Memory Mapped Files



Memory-Mapped Shared Memory in Windows



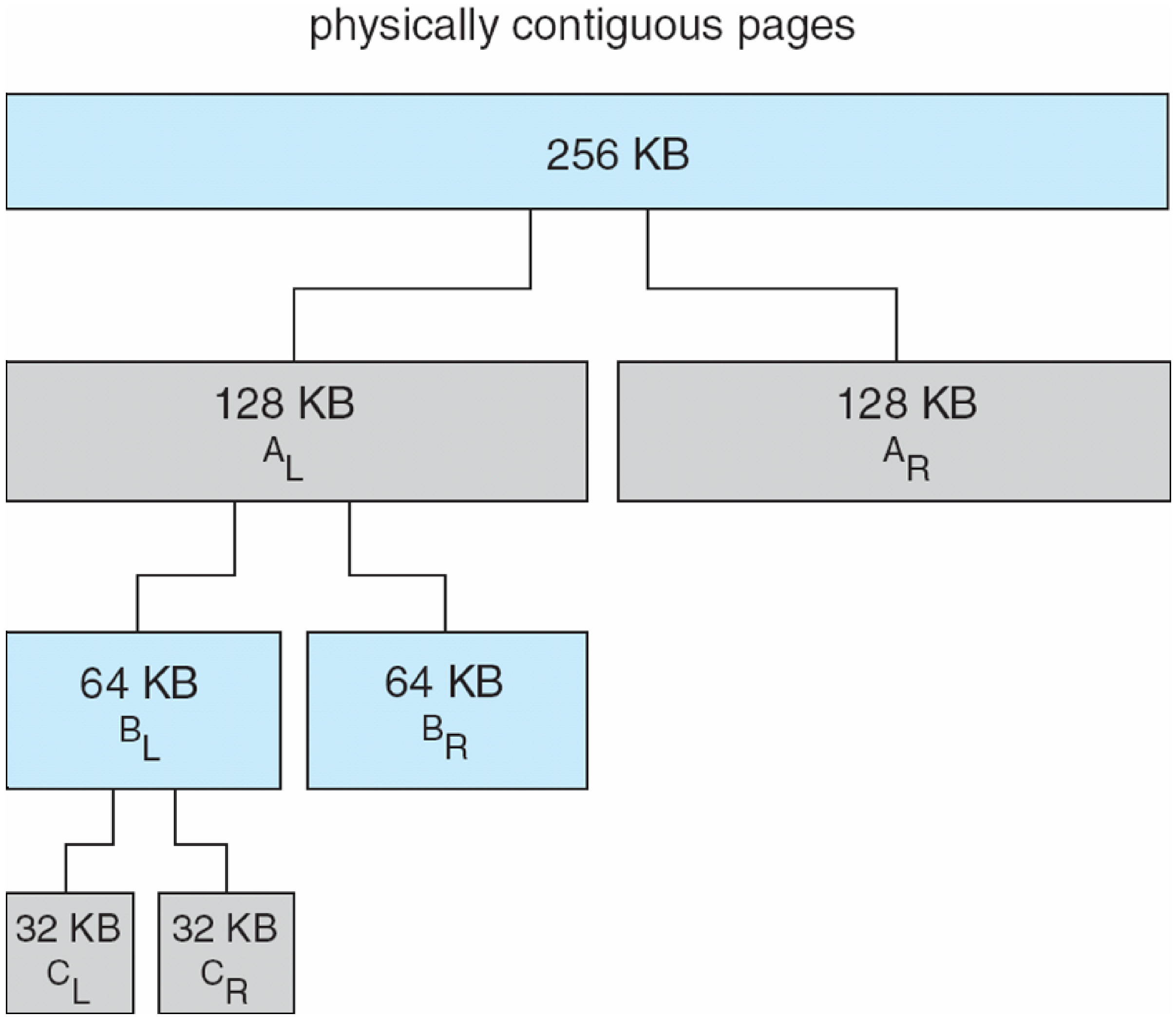
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - Memory buffer will be accessed by a DMA device on a physically addressed bus (like PCI)
    - Base kernel is placed on a contiguous block that can fit into one page
      - Reduce chance of TLB miss
    - Contiguous page frame allocation leaves kernel page tables unchanged, preserving TLB and reducing effective access time
  - kmalloc

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy System Allocator



Buddy System Allocator

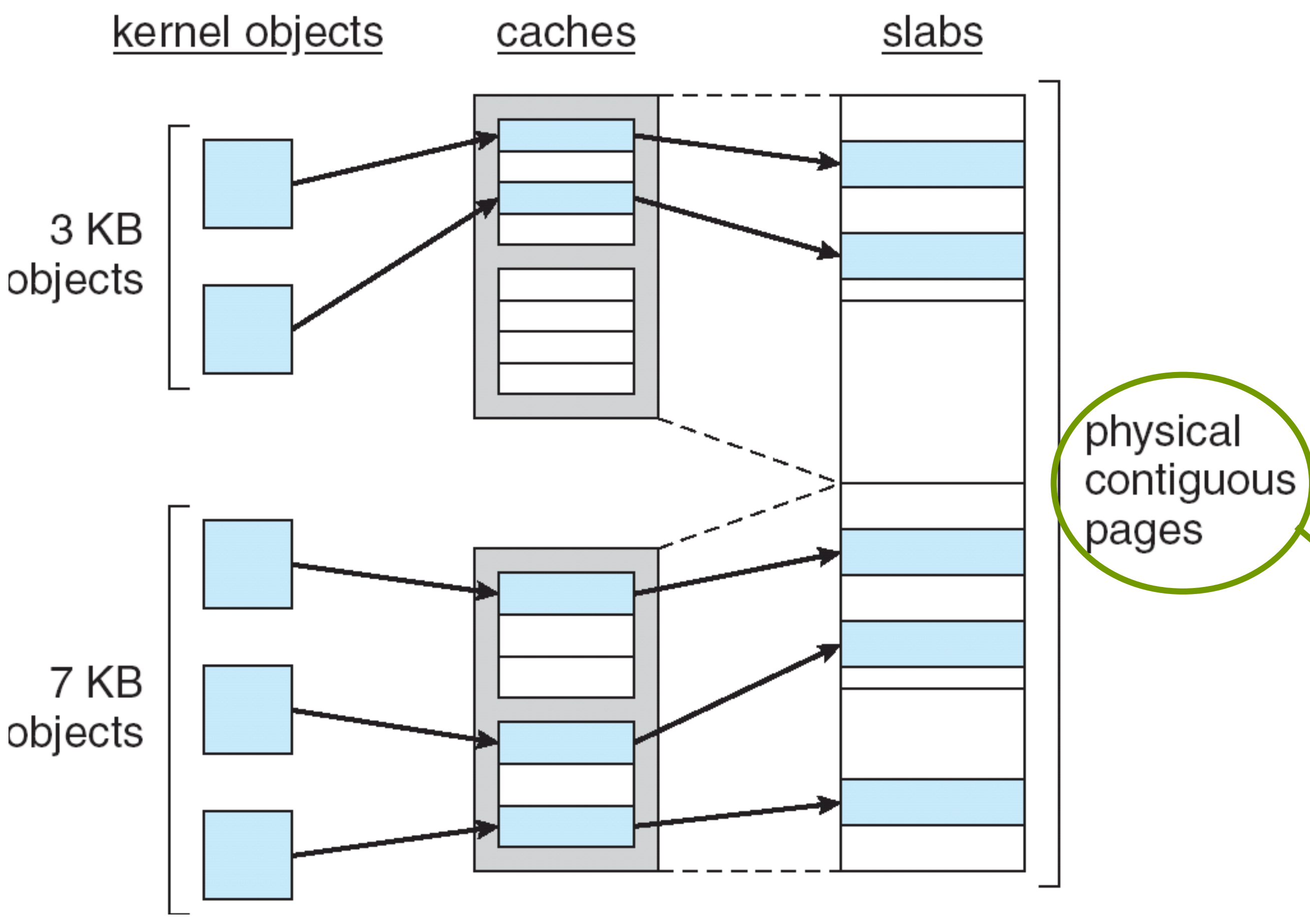
- Buddy system addresses external fragmentation
- How about internal fragmentation?
  - i.e. when allocating a 65 bytes structure, buddy system returns a 128 bytes block
    - 128-65 = 63 bytes are wasted
- http://utcc.utoronto.ca/~cks/space/blog/linux/KernelMemoryZones

```
linux1:~
(linux1:~) ysw% ls /proc/buddyinfo
/proc/buddyinfo
(linux1:~) ysw% cat /proc/buddyinfo
Node 0, zone      DMA      0      0      1      0      2      1      1      0      1      1      3
Node 0, zone    DMA32   1100   1384    709    358    161     89     56     55     46     27     72
Node 0, zone   Normal    401      0      0      0      0      0      0      0      0      0      0
Node 1, zone   Normal   1361   1936    755    487    493    413    381    247    142    166    758
(linux1:~) ysw%
```

Slab Allocator

- Introduced in Linux 2.2 kernel to deal with internal fragmentation
- Kernel functions often request small objects of the same type repeatedly
  - Process descriptors, file descriptors, etc.
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **free**
- When structures stored, objects marked as **used**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Each slab is allocated by buddy system

Slab Allocator

linux1:~

```
(linux1:~) ysw% slabtop
 Active / Total Objects (% used)    : 796954 / 892955 (89.2%)
 Active / Total Slabs (% used)      : 20248 / 20248 (100.0%)
 Active / Total Caches (% used)     : 73 / 108 (67.6%)
 Active / Total Size (% used)       : 217238.22K / 230534.95K (94.2%)
 Minimum / Average / Maximum Object : 0.01K / 0.26K / 8.00K

  OBJS ACTIVE  USE OBJ SIZE  SLABS OBJ/SLAB CACHE SIZE NAME
138474 138009  99%    0.19K   3297       42     26376K dentry
122421 122413  99%    0.10K   3139       39     12556K buffer_head
102200 101492  99%    0.07K   1825       56      7300K selinux_inode_security
 62310  61882  99%    1.02K   2010       31     64320K nfs_inode_cache
 55904  55871  99%    0.25K   1747       32     13976K kmalloc-256
 50496  24688  48%    0.06K    789       64      3156K kmalloc-64
 35840  22299  62%    0.02K    140      256       560K kmalloc-16
 35374  25868  73%    0.17K    769       46      6152K vm_area_struct
 35328  35089  99%    0.01K     69      512       276K kmalloc-8
 34300  34292  99%    0.55K   1225       28     19600K radix_tree_node
 31290  27748  88%    0.19K    745       42      5960K kmalloc-192
 28050  19053  67%    0.05K    330       85      1320K shared_policy_node
 22784   9567  41%    0.03K    178      128       712K kmalloc-32
 17416  12381  71%    0.07K    311       56      1244K anon_vma
 16779  16665  99%    0.08K    329       51      1316K sysfs_dir_cache
 11184  10037  89%    0.65K    466       24      7456K proc_inode_cache
```

Noncontiguous Memory Area Allocation

- In the Linux kernel, we try to avoid allocating noncontiguous memory areas
- But there are occasions when we want to create a large buffer in the kernel that can’t fit into a contiguous kernel memory area
  - We can use paging for the allocation
  - Linux uses most of the reserved addresses above PAGE_OFFSET to map non-contiguous memory area
  - vmalloc

Other Issues -- Prepaging

- Prepaging
  - To reduce the large number of page faults that occurs at process startup
  - Prepage all or some of the pages a process will need, before they are referenced
  - But if prepaged pages are unused, I/O and memory was wasted
  - Assume s pages are prepaged and α of the pages is used
    - Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging
      $s * (1- \alpha)$ unnecessary pages?
    - α near zero $\Rightarrow$ prepaging loses

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead
  - locality

Other Issues – TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Other Issues – Program Structure

- Program structure
  - `Int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

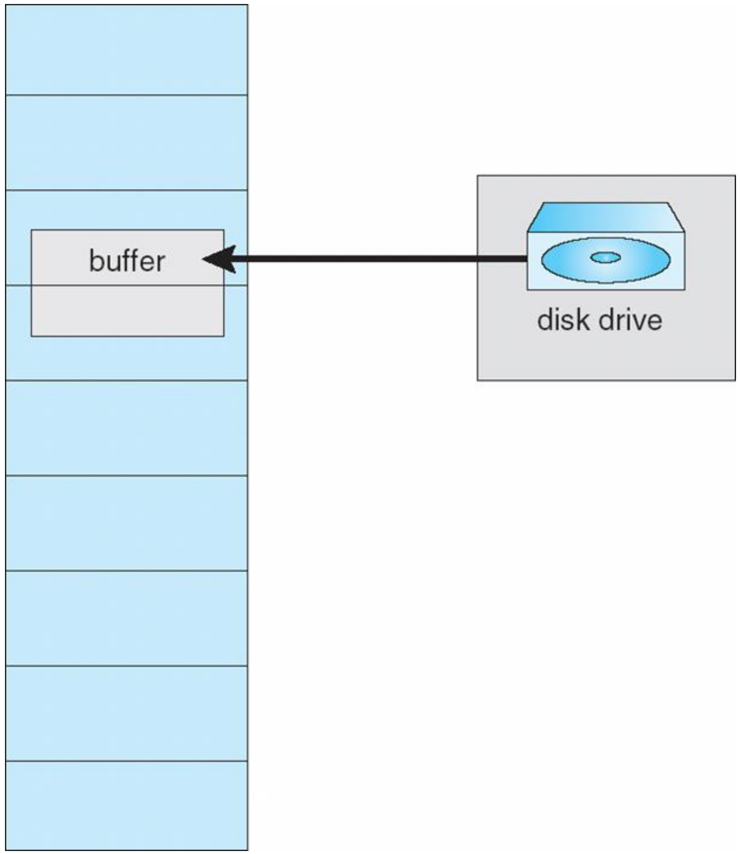
128 page faults

Other Issues – I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory

- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm

Reason Why Frames Used For I/O Must Be In Memory



Operating System Examples

- Windows XP
- Solaris

Windows XP

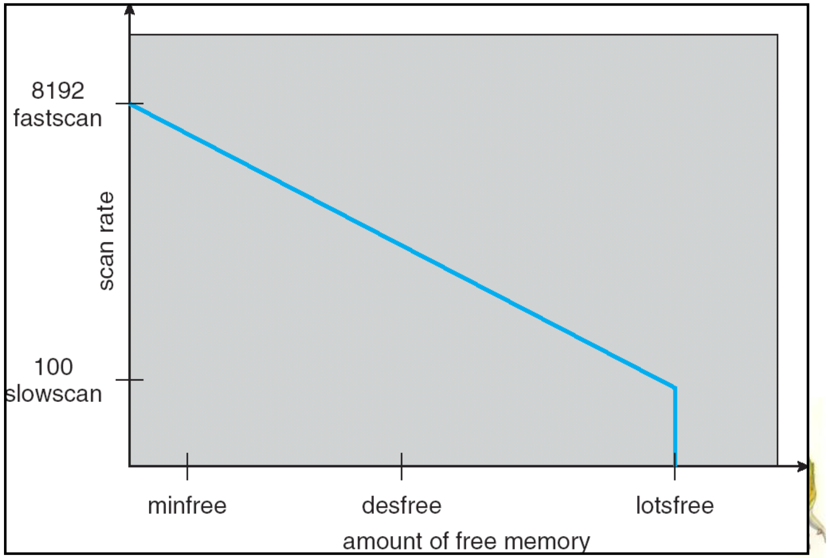
- uses demand paging with *clustering*
  - clustering brings in pages surrounding the faulting page
- processes are assigned *working set minimum* and *working set maximum*
  - working set minimum is the minimum number of pages the process is guaranteed to have in memory (for most apps, 50…345)
  - a process may be assigned as many pages up to its working set maximum
  - when the amount of free memory in the system falls below a threshold, *automatic working set trimming* is performed to restore the amount of free memory
  - working set trimming removes pages from processes that have pages in excess of their working set minimum
- page replacement algorithm varies depending on processor
  - single processor Intel CPU: variation of the clock (timestamp) algorithm
  - multiprocessor or Alpha CPU: variation of FIFO

Solaris

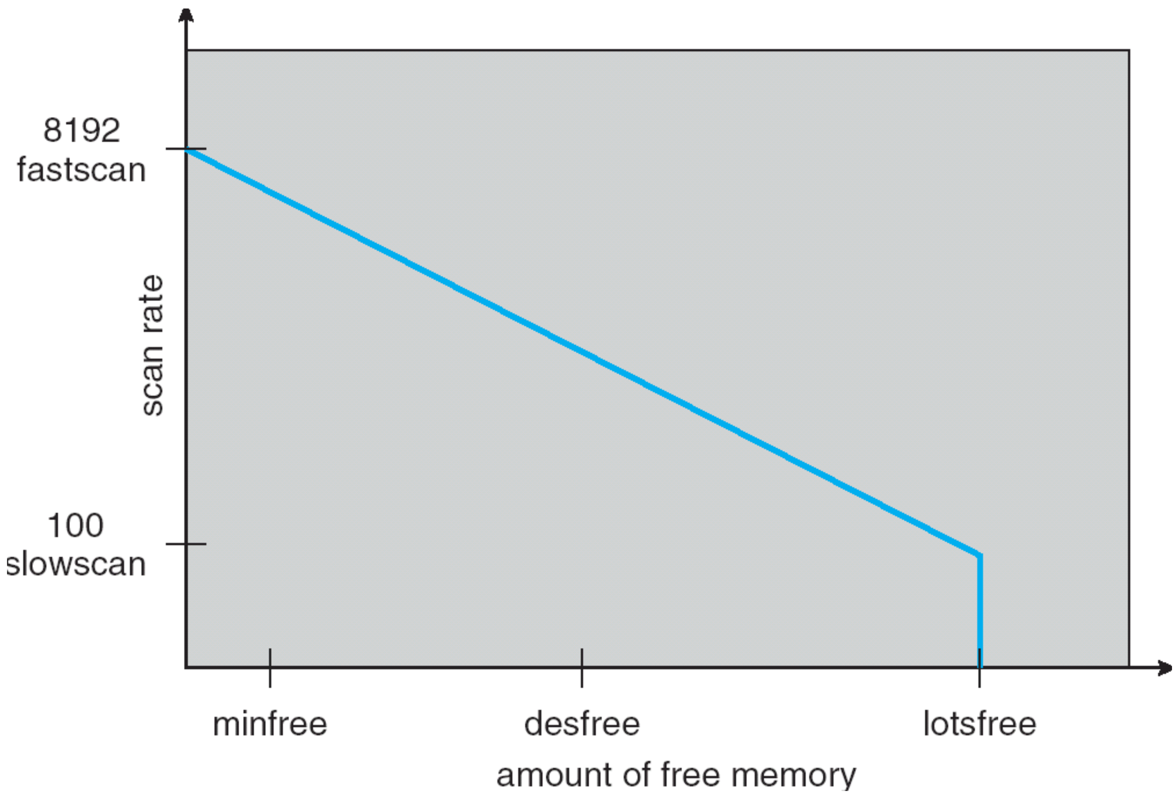
- system maintains a list of free pages
  - paging occurs when the number of free pages drops below a threshold (e.g., 1/64 size of physical memory)

- paging is performed by a *pageout* process
  - pageout scans pages using a modified clock algorithm
  - pages not referenced since last scan are returned to free list
  - the smaller the free list, the more frequent scanning occurs

lotsfree – threshold parameter to begin paging
desfree – threshold parameter to increase paging
minfree – threshold parameter to begin swapping



Solaris 2 Page Scanner



End of Chapter 9