Bilkent University
Department of Computer Engineering
CS342 Operating Systems

Virtual Memory

Last Update: April 20, 2023

Objectives and Outline

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

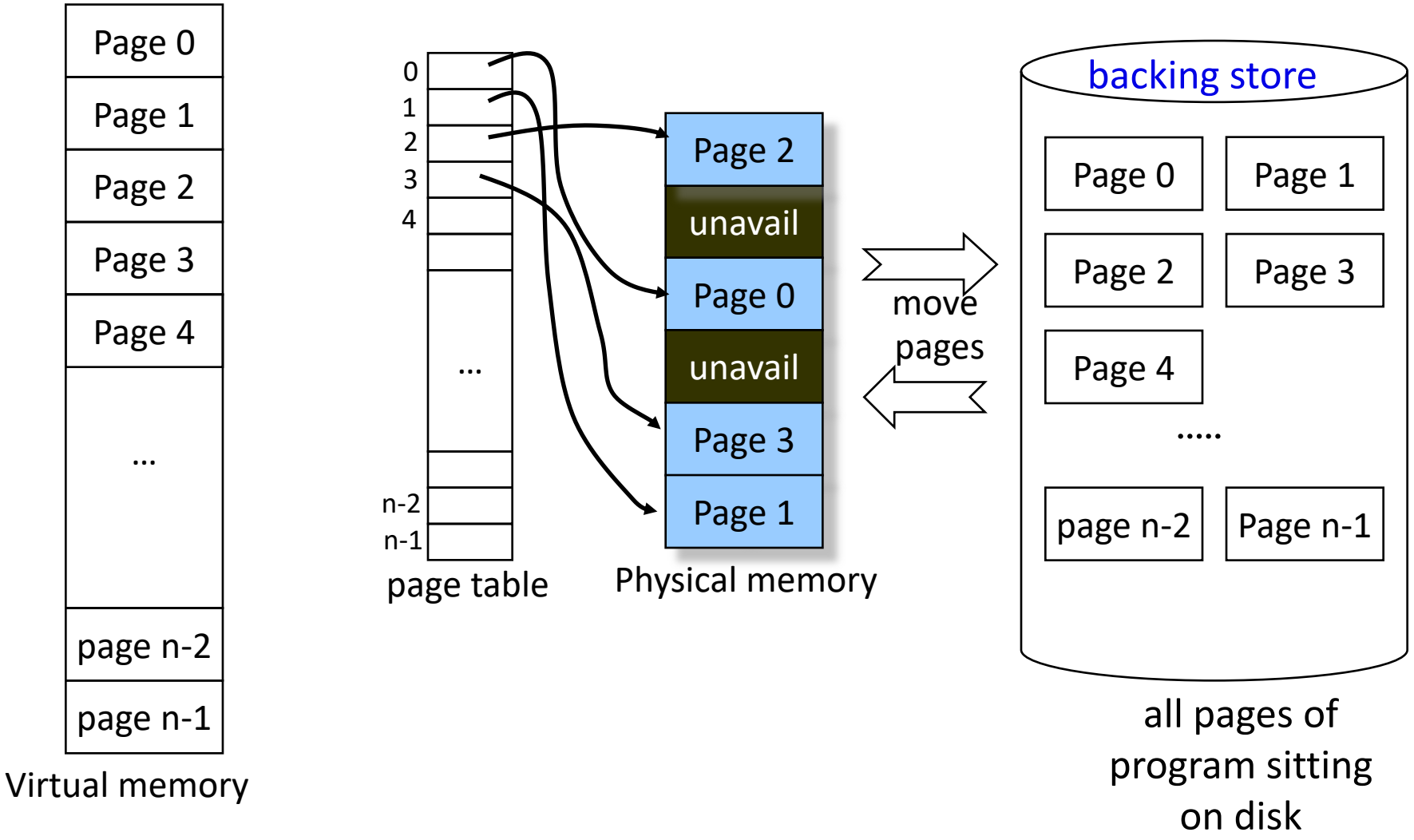
Outline

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

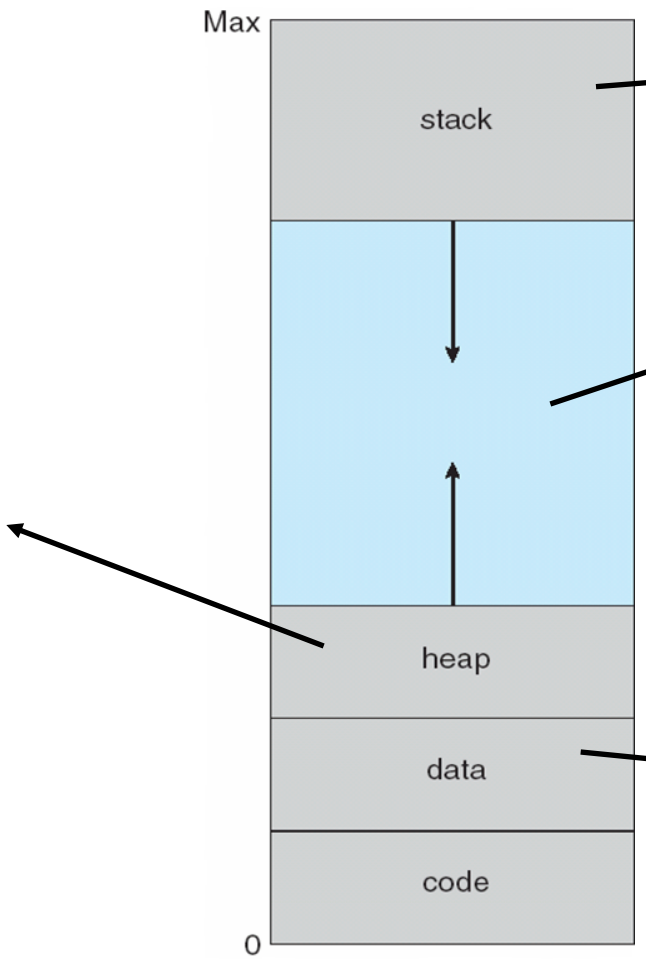
Background

- Virtual memory: program uses virtual memory which can be partially loaded into physical memory
- Benefits:
  - Only part of the program needs to be in memory for execution ==> more programs in memory
  - Logical address space can be much larger than physical address space ==> execute programs larger than RAM size
  - Easy sharing of address spaces by several processes
    - Library or a memory segment can be shared
  - Allows for more efficient process creation
    - Quick program start; quick child start.

Virtual Memory that is larger than Physical Memory



A typical virtual-address space layout of a process



function parameters;
local variables;
return addresses

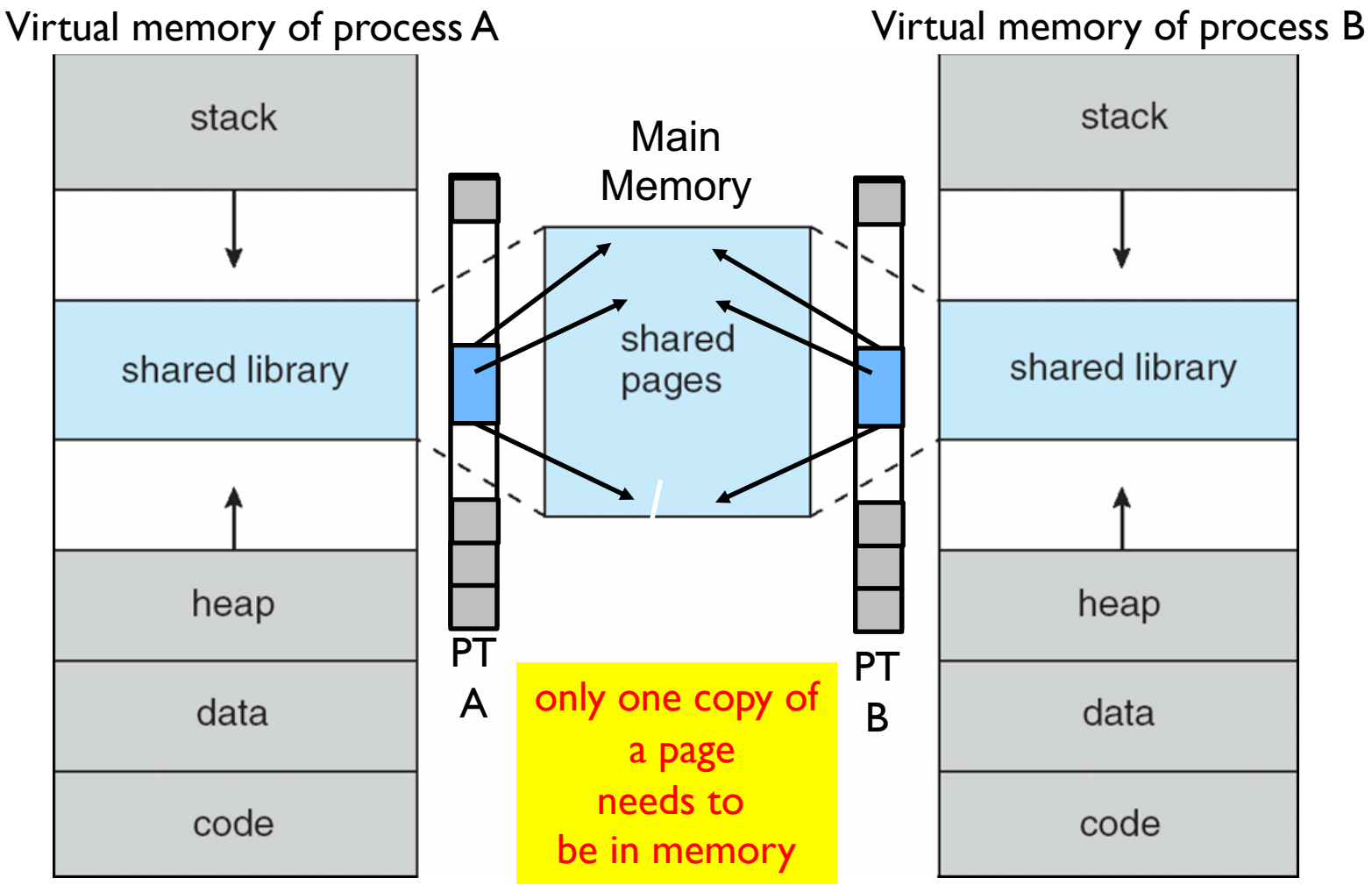
unused address space

can be used when needed

malloc() allocates space from here (dynamic memory allocation)

global data (variables)

Shared Library using Virtual Memory



Implementing Virtual Memory

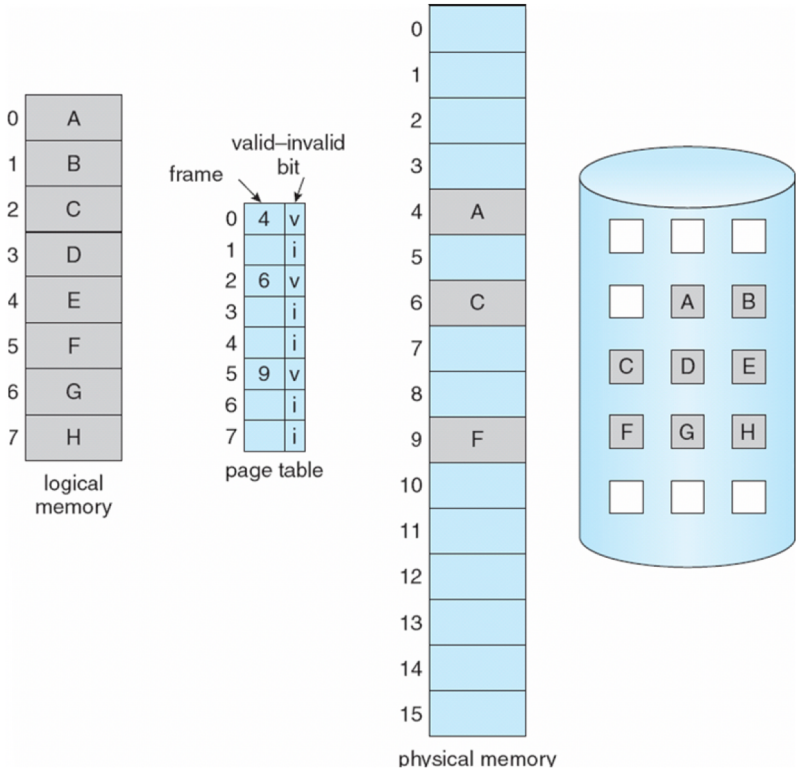
- Virtual memory can be implemented via:
  - Demand paging
    - Bring pages into memory when they are needed, i.e., allocate memory for pages when they are accessed (referenced)
  - Demand segmentation
    - Bring segments into memory when they are needed, i.e., allocate memory for segments when they are accessed.

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response – can start process quickly
  - More users – more programs in memory
- Page is needed when running program makes a reference to it
  - PTE bit valid: page is in used region of VM and is in Memory
  - PTE bit invalid:
    - page is not in used portion of address space ➔ abort
    - page is not in memory ➔ bring it to memory
- Pager never brings a page into memory unless page is needed

Valid-Invalid Bit

- With each page table entry (PTE) a valid–invalid bit (validation bit) is associated
- Initially validation bit is set to i (invalid) on all entries
- During address translation of a page
  - if validation bit is i, then page fault exception generated
    - Program suspended
    - Kernel runs



Page Fault Handling

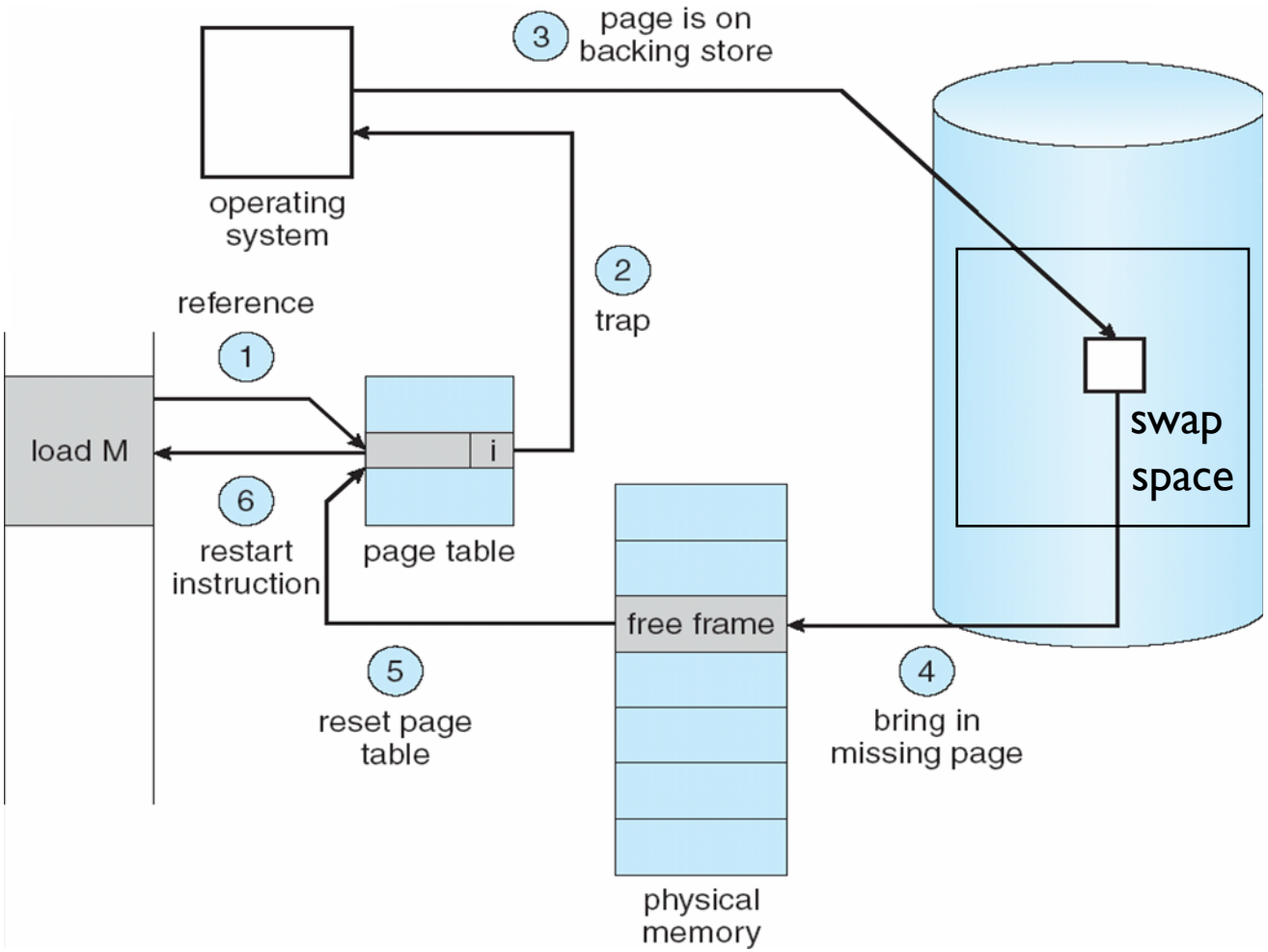
Kernel handles page fault exception as follows:

1. Kernel looks at another table/list (keeping used WM regions of the process) to decide
   - invalid reference (page is in *unused* portion of address space) → **abort**
   - in used portion, but just not in phy memory → **bring the page in.**
     - Find the location of the desired page on disk. Disk location info can be in the respective PTE.
2. Get a free frame. if there is no free frame, use a page replacement algorithm to select a victim frame/page and remove the page; if page was modified, need **disk I/O** to write back to disk.
3. Bring page from disk into frame - swapping in - (need **disk I/O**)
4. Reset the related entry in page table (install mapping into page table)
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault.

Page Fault Handling

- Restarting instruction:
  - If page fault occurs while trying to fetch an instruction, fetch the instruction again after bringing the page in.
  - If page fault occurs while we are executing an instruction: Restart the instruction after bringing the page in.
- For most instructions, restarting the instruction is no problem.
  - But for some, we need to be careful.

Steps in Handling a Page Fault



Performance of Demand Paging

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

EAT = (1 – p) x Memory-access-time + p x Page-fault-service-time

page fault service time = interrupt handling, loading page in, writing a page out (if needed), restarting the process and instruction

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault-service-time = 8 millisecond.
- 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 (p = 1/1000), then

  EAT = 8.2 microseconds.

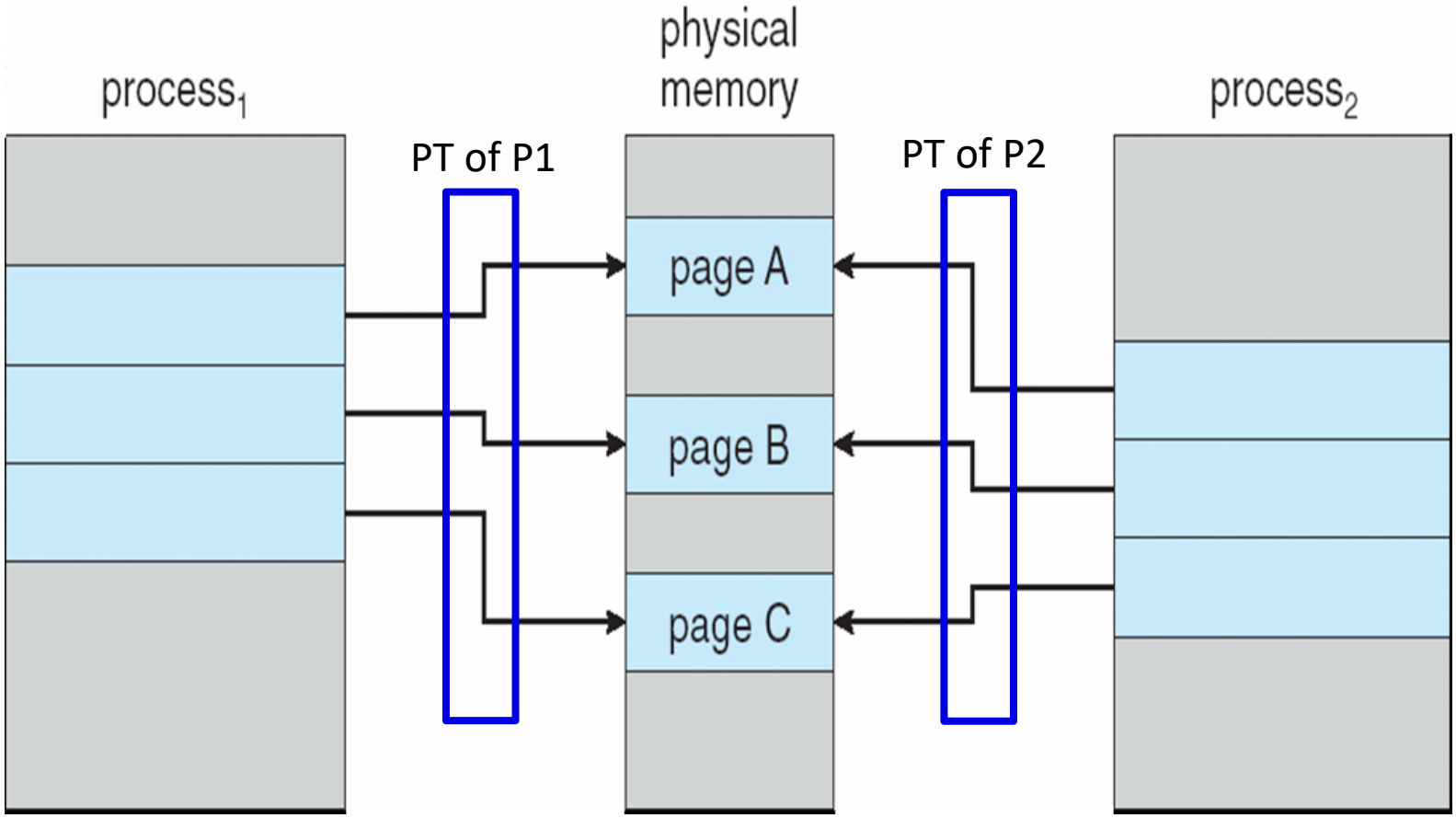
  This is a slowdown by a factor of 40!!

  (200 ns / 8.2 microsec ~= 1/40)

Other benefits of virtual memory

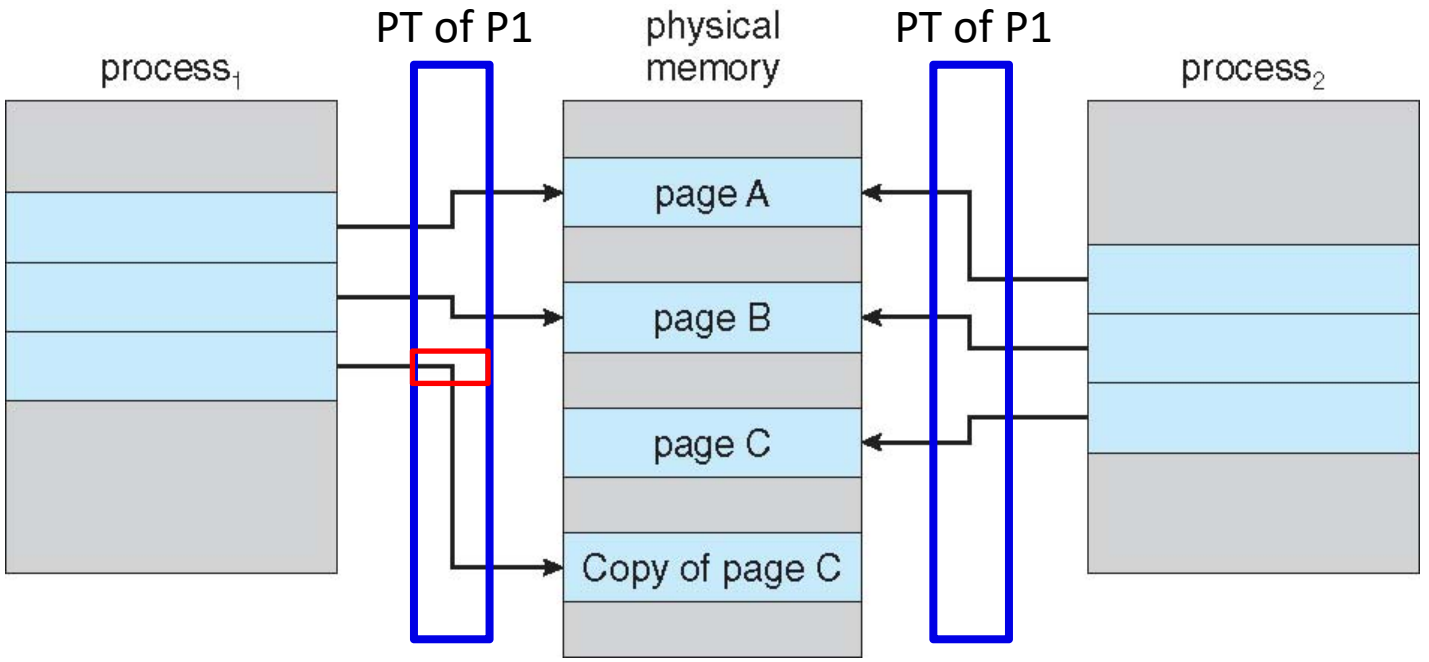
- Virtual memory has other benefits:
  - Copy-on-Write (fast process creation)
  - Memory-Mapped Files (later)
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory
- If either process modifies a shared page, only then is the page copied into an allocated frame.
- COW allows more efficient process creation as only modified pages are copied.

Before Process 1 Modifies Page C



When child is created, initially the page table of child points to the same frames that parent is using. No need to allocate frames and copy pages into them.

After Process 1 Modifies Page C



When one the processes tries to modify a page (write to memory), a frame is allocated, page is copied and modification is done in that page only. Page table entry updated.

Page Replacement

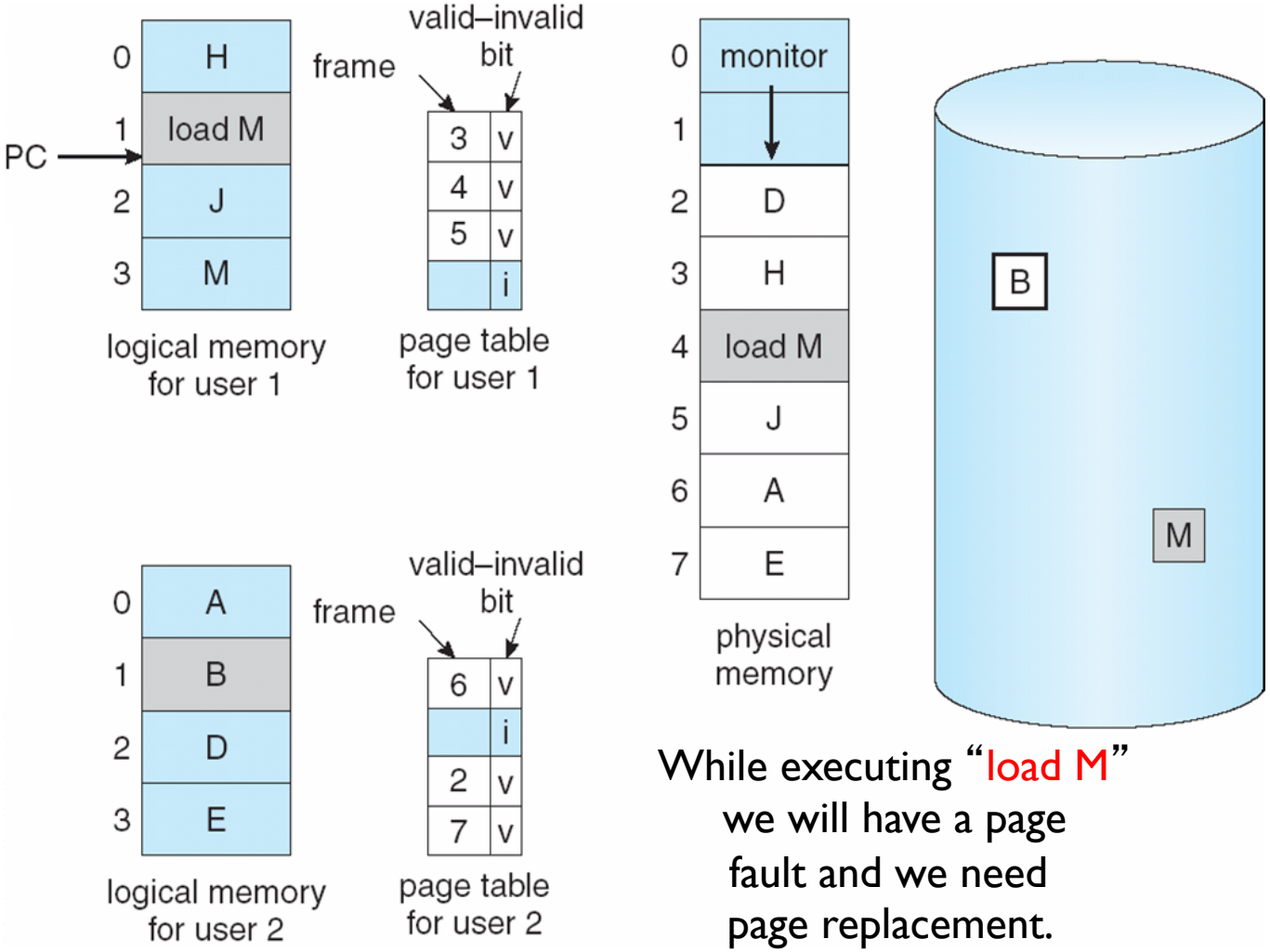
What happens if there is no free frame?

- Page replacement – find some page in memory, but not really in use, swap it out to disk (remove it).
  - Algorithm ? Which page should be removed?
  - performance – want an algorithm which will result in minimum number of page faults
- With page replacement, a page may be brought into memory several times (memory ←→ disk)
- We can prevent over-allocation of memory by modifying page-fault service routine to include page replacement.

Page Replacement

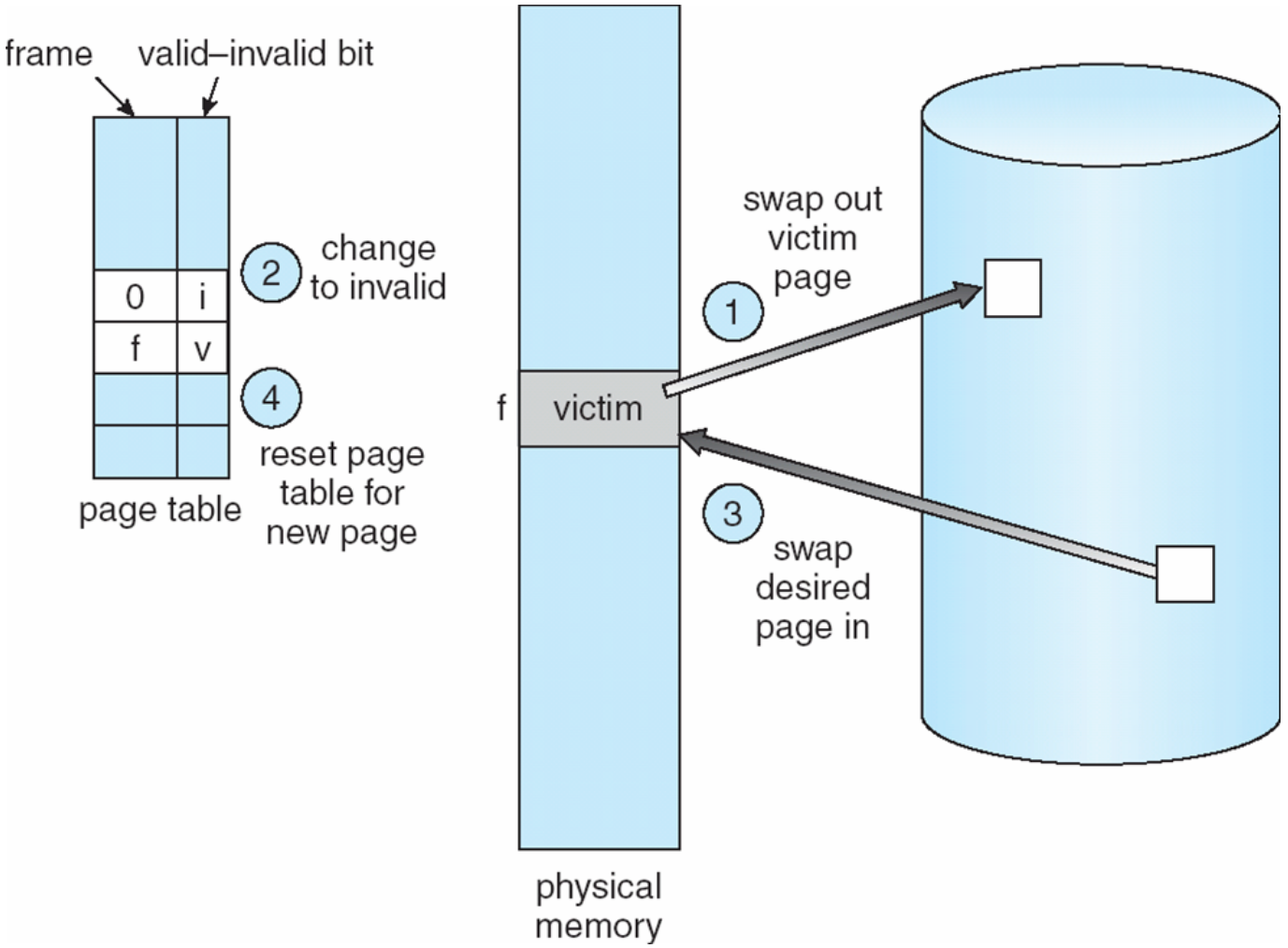
- Use modified (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk while removing/replacing a page.
- Page replacement completes separation between logical memory and physical memory.
- We have now true virtual memory.
  - large virtual memory can be provided on a smaller physical memory.

Need For Page Replacement



While executing "load M" we will have a page fault and we need page replacement.

Page Replacement



Page Replacement Algorithms

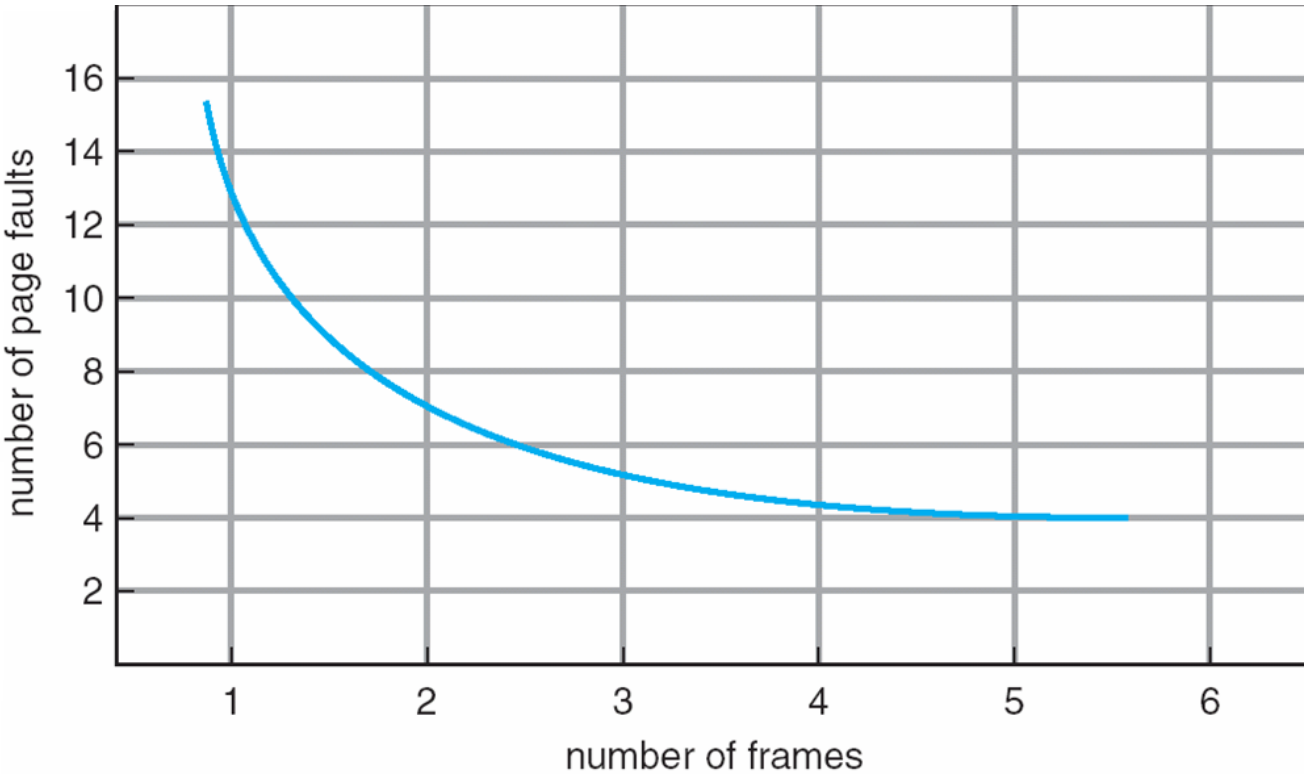
- Want lowest page-fault rate
- Evaluate an algorithm by running it on a particular *string of page (memory) references* (reference string) and computing the number of page faults on that string
- In all our examples, the page reference string is

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

Driving reference string

- Assume process makes the following memory references (in **decimal**) in a system with 100 bytes per page. Memory addresses referenced:
  - 0100 0432 0101 0612 0102 0103 0104 0101 0611 0102 0103 0104 0101 0610 0102 0103 0104 0609 0102 0105
- Example: Bytes (addresses) 0…99 will be in page 0
- Pages referenced are:
  - 1, 4, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 1, 6, 1, 1, 1, 6, 1, 1
- Corresponding page reference string
  - 1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

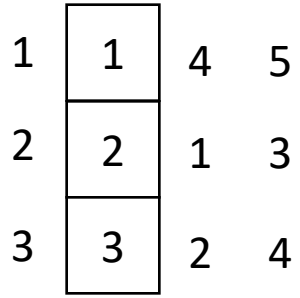
Graph of Page Faults Versus The Number of Frames



We expect less page fault rate when we use more frames for a process.

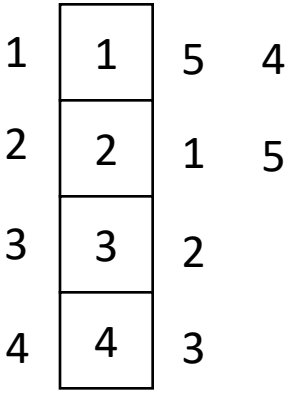
First-In-First-Out (FIFO) Algorithm

- 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)



9 page faults

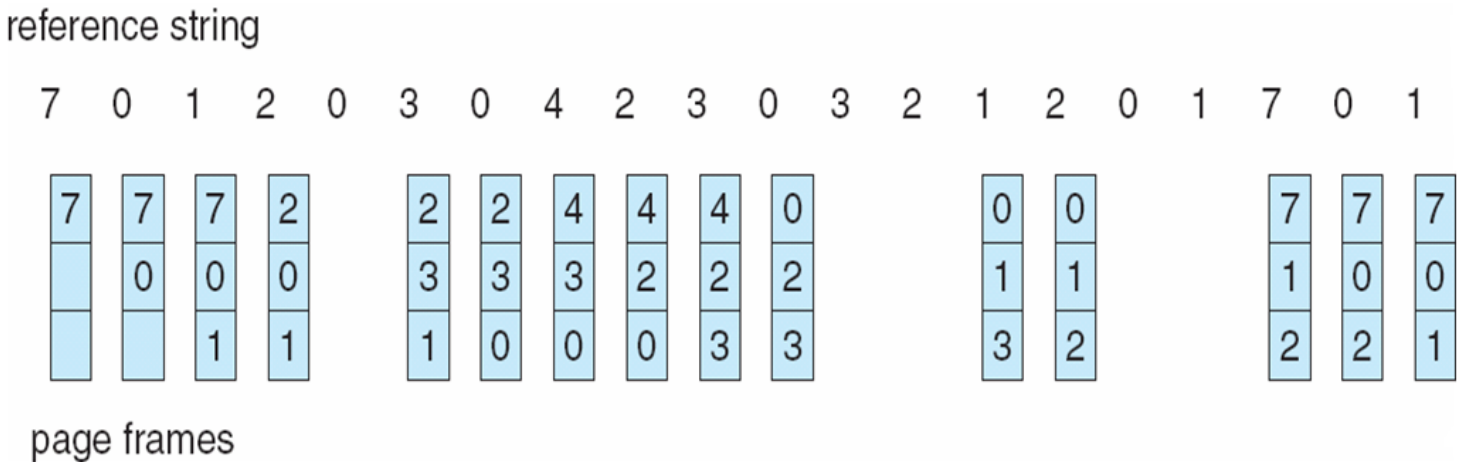
- 4 frames



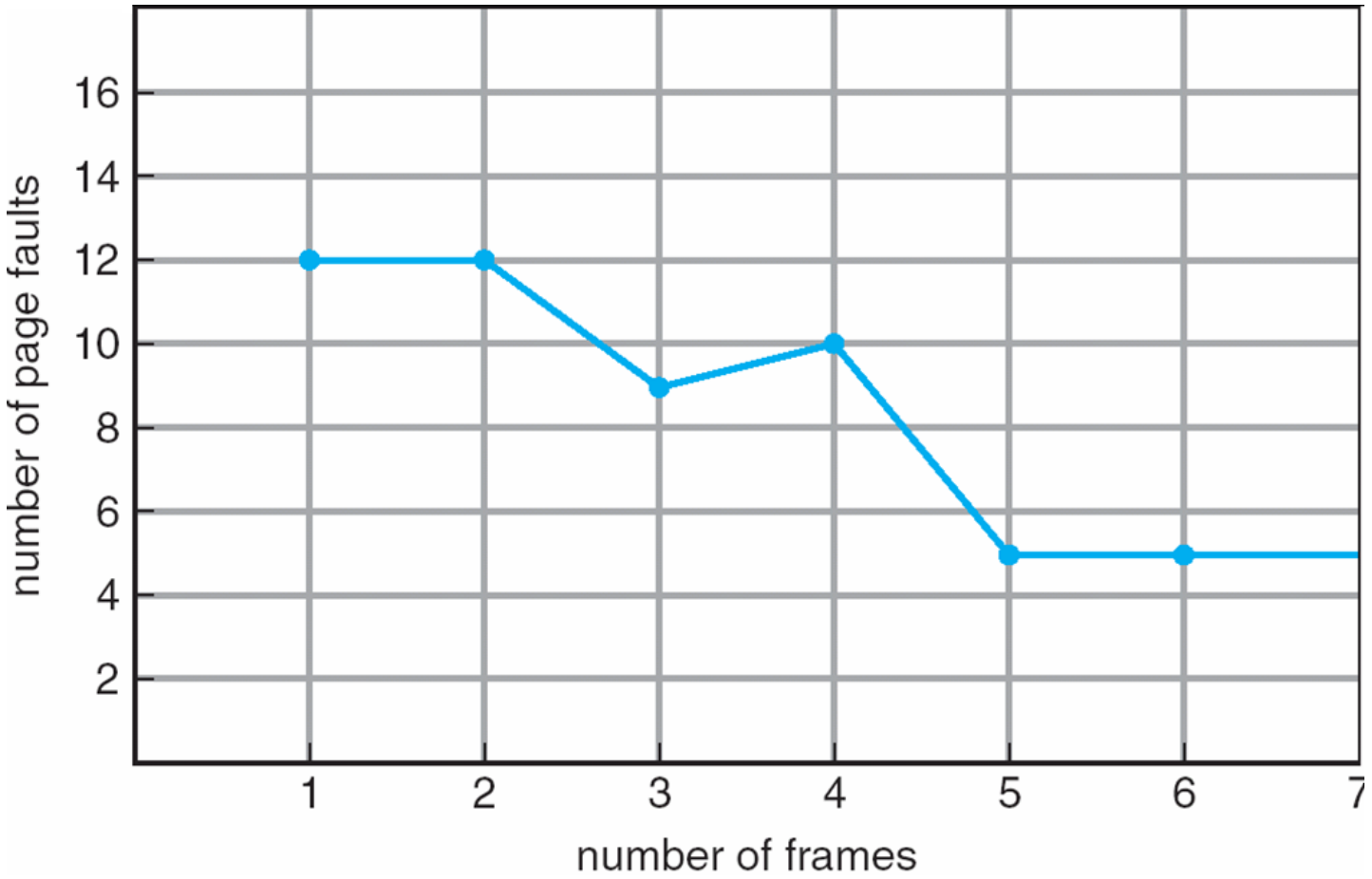
10 page faults

- Belady's Anomaly: more frames $\Rightarrow$ more page faults

FIFO Page Replacement



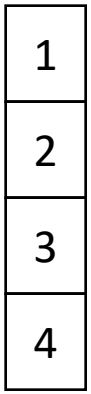
FIFO Illustrating Belady's Anomaly



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



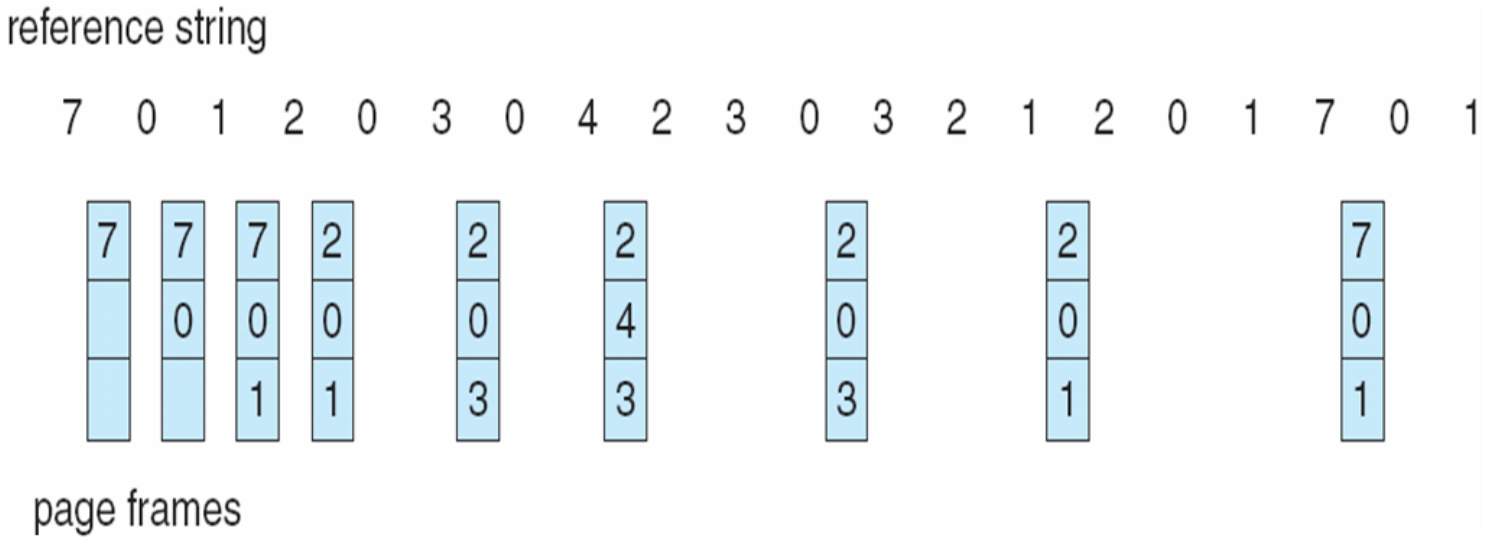
4

5

6 page faults

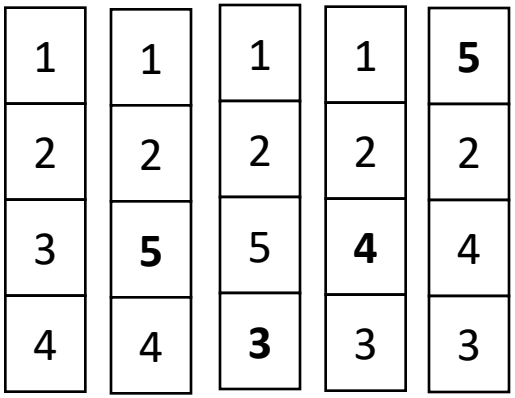
- How do you know this?
- Used for measuring how well your algorithm performs

Optimal Page Replacement



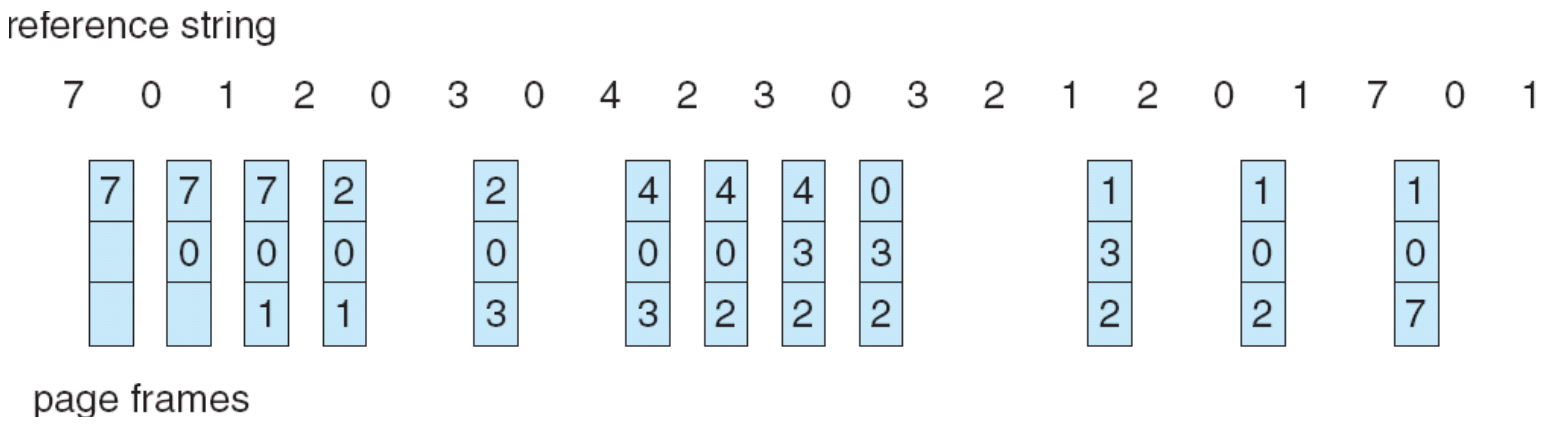
Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



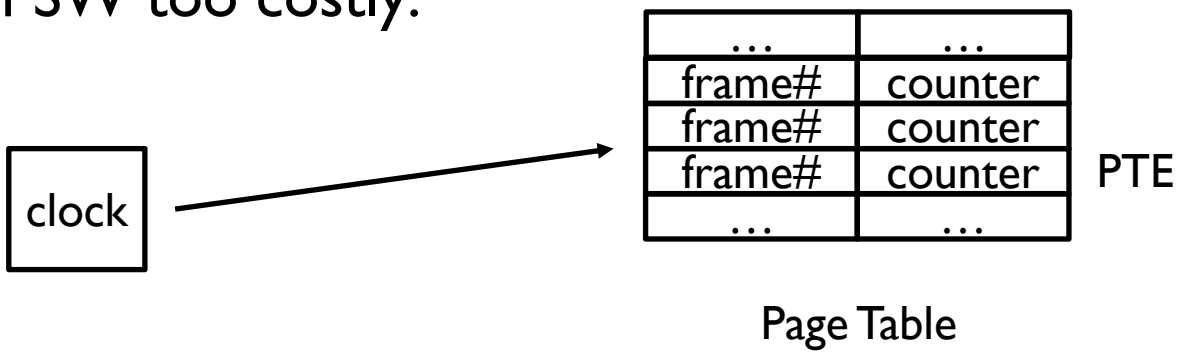
8 page faults

LRU Page Replacement



LRU Algorithm Implementation

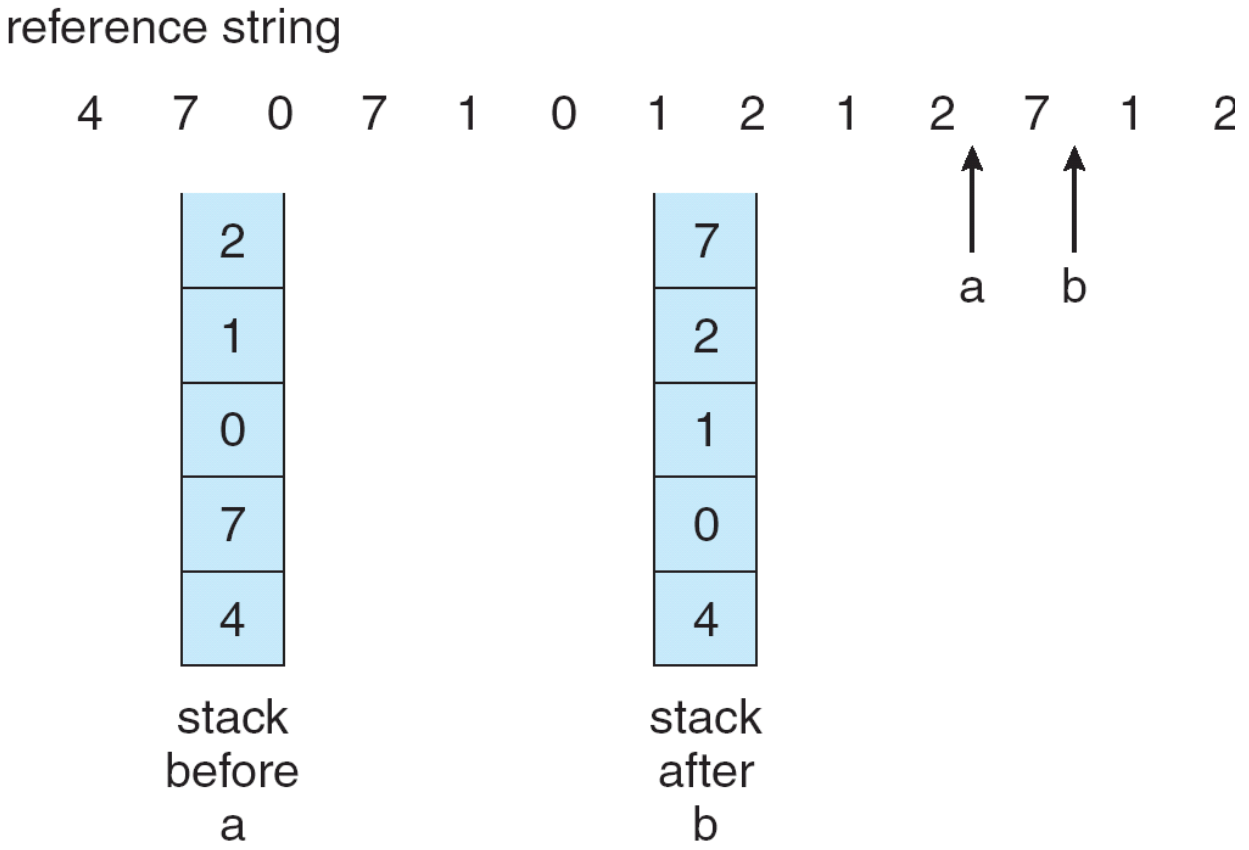
- **Counter** implementation
  - Every *page entry* has *a counter field*; with every page reference, copy the *clock* value into the counter field.
  - When replacement needed, select the one with smallest counter value.
- Need hardware support (rare).
- Copying in SW too costly.



LRU Algorithm Implementation

- **Stack** implementation – keep a *stack of page numbers* in a doubly link form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed (with every memory reference; costly)
  - No search for replacement
- Costly since every memory reference requires updating the stack.

Use of a Stack to Record The Most Recent Page References



LRU Approximation Algorithms

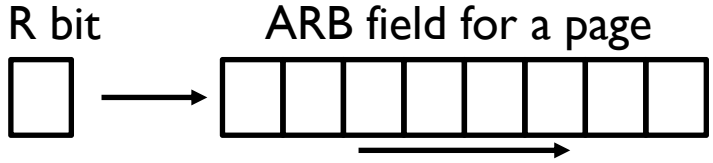
- Additional Reference bits
- Second Chance 1
- Second Chance 2 (clock)
- Enhanced Second Chance

Hardware Support
Reference Bit

- Utilize the reference bit (R bit). Most HW has it.
  - With each page, an R bit stores in its PTE.
  - When page is referenced, bit set to 1 (by HW)
- Reference bits are cleared *periodically* by OS.
  - With every timer interrupt, for example.
  - For example: every 10 ms.
- We can utilize the R bit to implement a practical algorithm that approximates LRU.

Additional Reference Bits

- Besides the reference bit (R bit) for each page, we can keep an Additional Reference Bits (lets call it as ARB) field associated with each page. For example, an 8-bit field that can store 8 reference bits.
- At each timer interrupt (or periodically) R bit shifted into ARB. All ARB bits are also shifted right by one bit.
  - The value in the ARB field indicates when the page is accessed approximately.

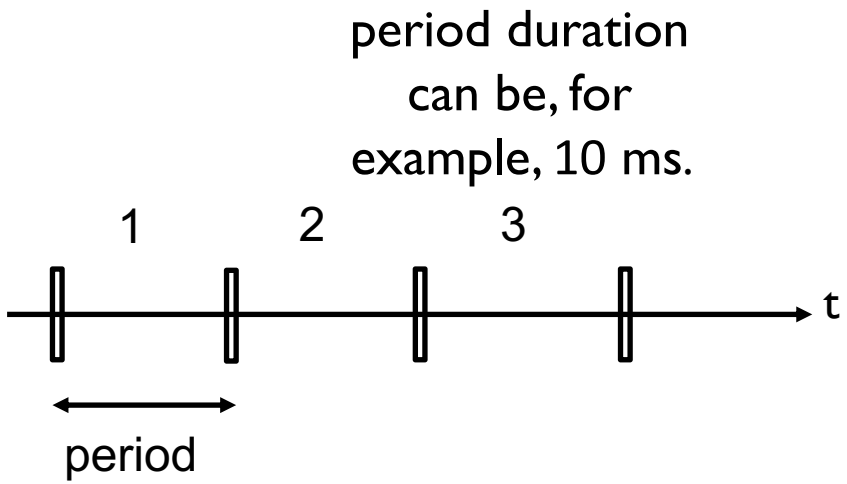


- Select the page with *smallest* ARB field value as victim.

Additional Reference Bits
Example for a page

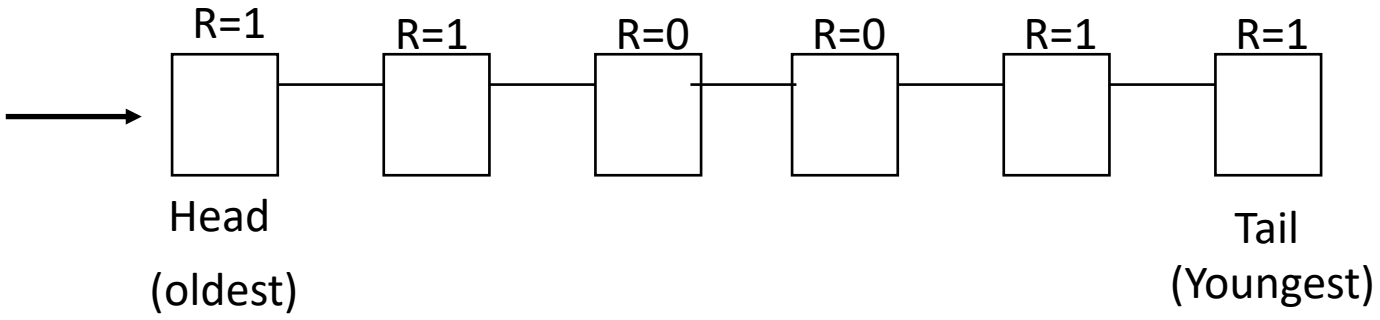
- At tick 1: R: 0, ARB: 0000000
- During period 1: R is set (R:1)
- At tick 2: R:0, ARB: 1000000
- During period 2: R is not set
- At tick 3: R:0, ARB: 0100000
- During period 3: R is set (R:1)
- At tick 4: R:0, ARB: 1010000

- ….



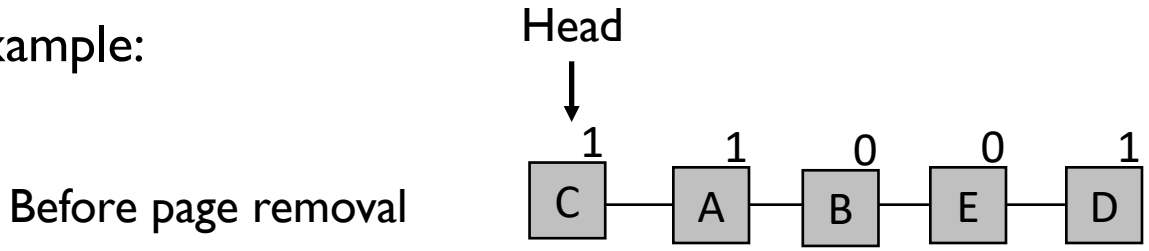
Second-Chance Algorithm 1

- FIFO that is using R bit. Select the the page close to head of the list and that has R bit 0 as victim page.
- While scanning the list of pages starting from head, if we see a page with R=1, we move it to the tail and make R=0 (a second is given to the page)
- May require to change all 1's to 0's and then come back to the head of the list.
- Add a newly loaded page to the tail with R = 1.

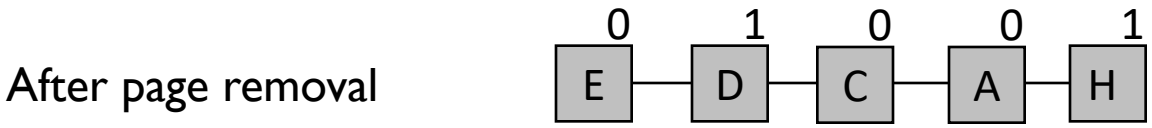


Second-Chance Algorithm 1

- Example:



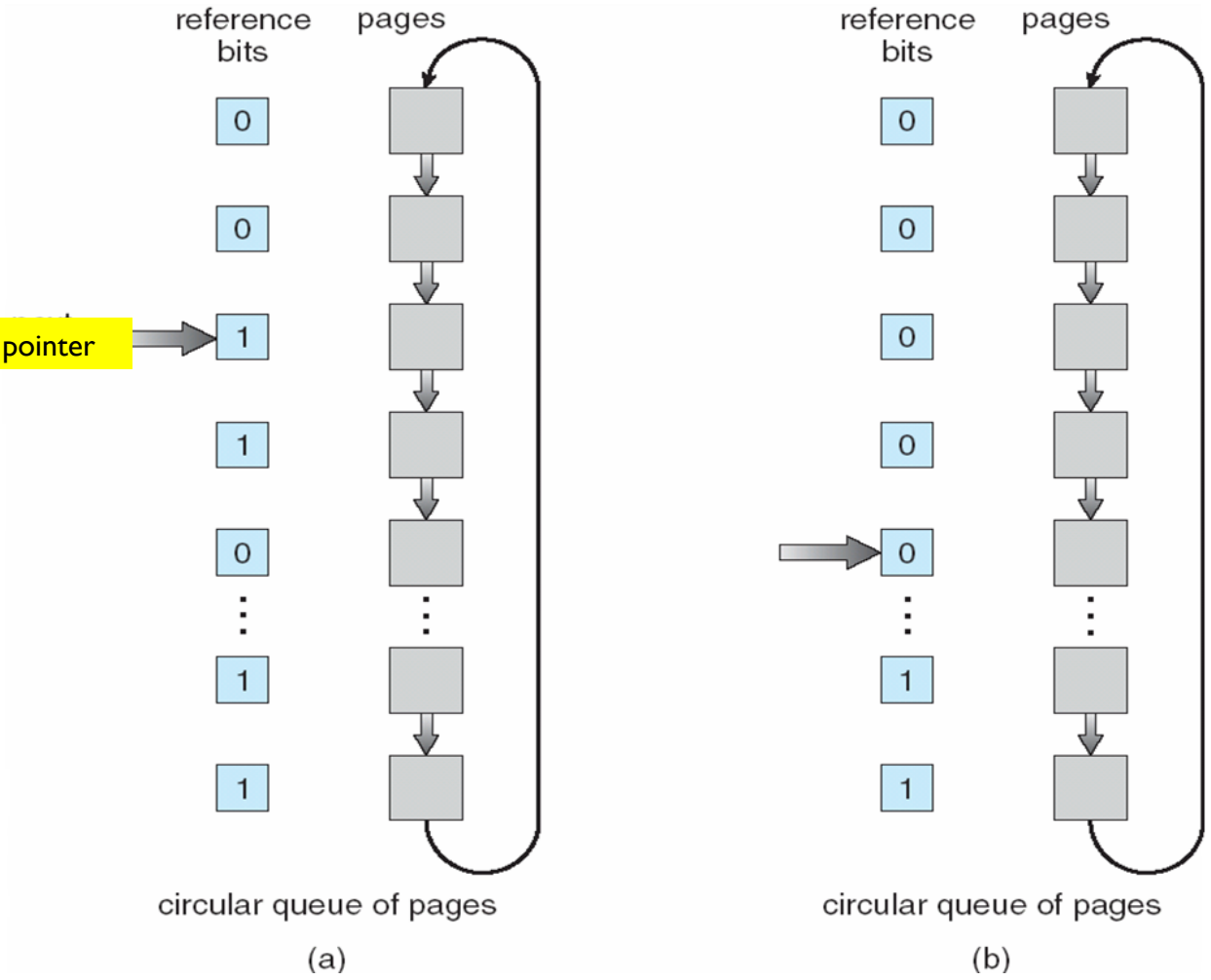
Access page H. Page fault. Remove a page. B is removed.



B is removed

Second-Chance Algorithm 2 (Clock Algorithm)

Second chance can be implemented using a circular list of pages;
Then it is also called Clock algorithm



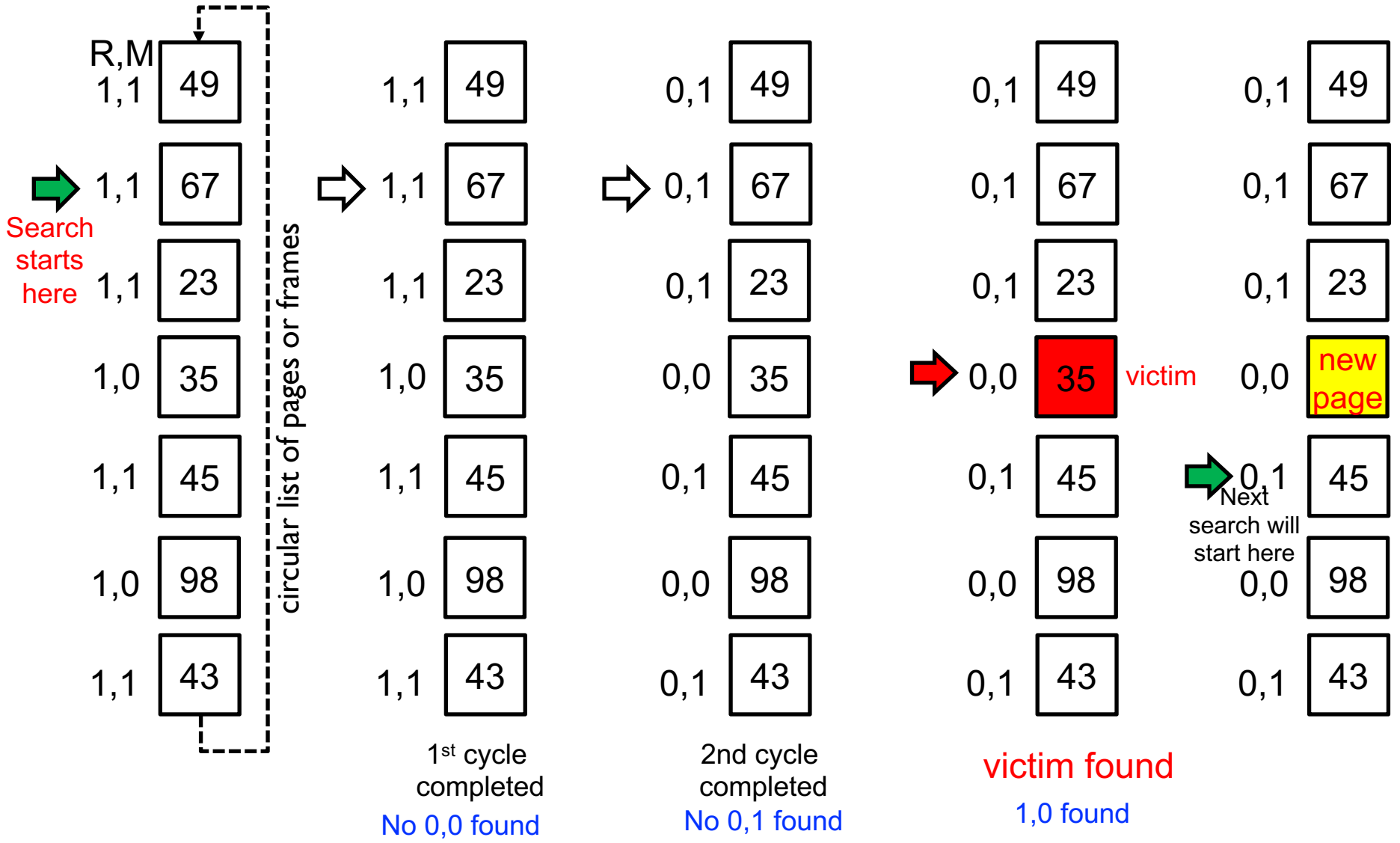
Enhanced Second-Chance (Clock) Algorithm

- Consider reference bits and the modified bits (dirty bit) of pages
  - Reference (R) bit: page is referenced in the last interval
  - Modified (M) bit: page is modified after being loaded into memory.
- Four possible cases (R,M):
  - 0,0: neither recently used nor modified
  - 0,1: not recently used but modified
  - 1,0: recently used but clean
  - 1,1: recently used and modified
- We replace the first page encountered in the lowest non-empty class.

Enhanced Clock Algorithm

- Step 1) Begin at pointer. Scan the circular list. In this scan, do not reset R bits. The first page that has R=0, M=0 is the victim. If cycle completed and victim not found, go to step 2.
- Step 2) Scan again the circular list. This time reset R bit of a bypassed page that has R=1. The first page with R=0, M=1 is victim. If cycle completed and victim not found, go to step 3.
- Step 3) Continue scanning and select the first page that has R=0,M=0 as victim. If cycle completed and victim not found, go to step 4.
- Step 4) Continue scanning and select the first page with R=0,M=1 as victim.

Example
Handling page fault



- For a newly loaded page, R bit can be set to 0 or 1, depending on the system.
  - If needed, we will say what will be assumed for the the R bit of a newly loaded page.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - counter incremented with every memory reference,
  - or at each timer interrupt if R=1.
- 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: 6 pages to handle a "move from to" instruction:
  - instruction may be 6 bytes, might span 2 pages
  - 2 pages to handle *from;* 2 pages to handle *to*
- Various allocation approaches
  - fixed allocation (this is a kind of *local* allocation)
    - *Equal* allocation
    - *Proportional* allocation (proportional to the size)
  - priority allocation (this is a kind of global allocation)
  - global allocation
  - local 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.

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

$S = \sum s_i$

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

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

Example:

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \left(\frac{10}{137}\right) x64 \cong 5$

$a_2 = \left(\frac{127}{137}\right) x64 \cong 59$

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

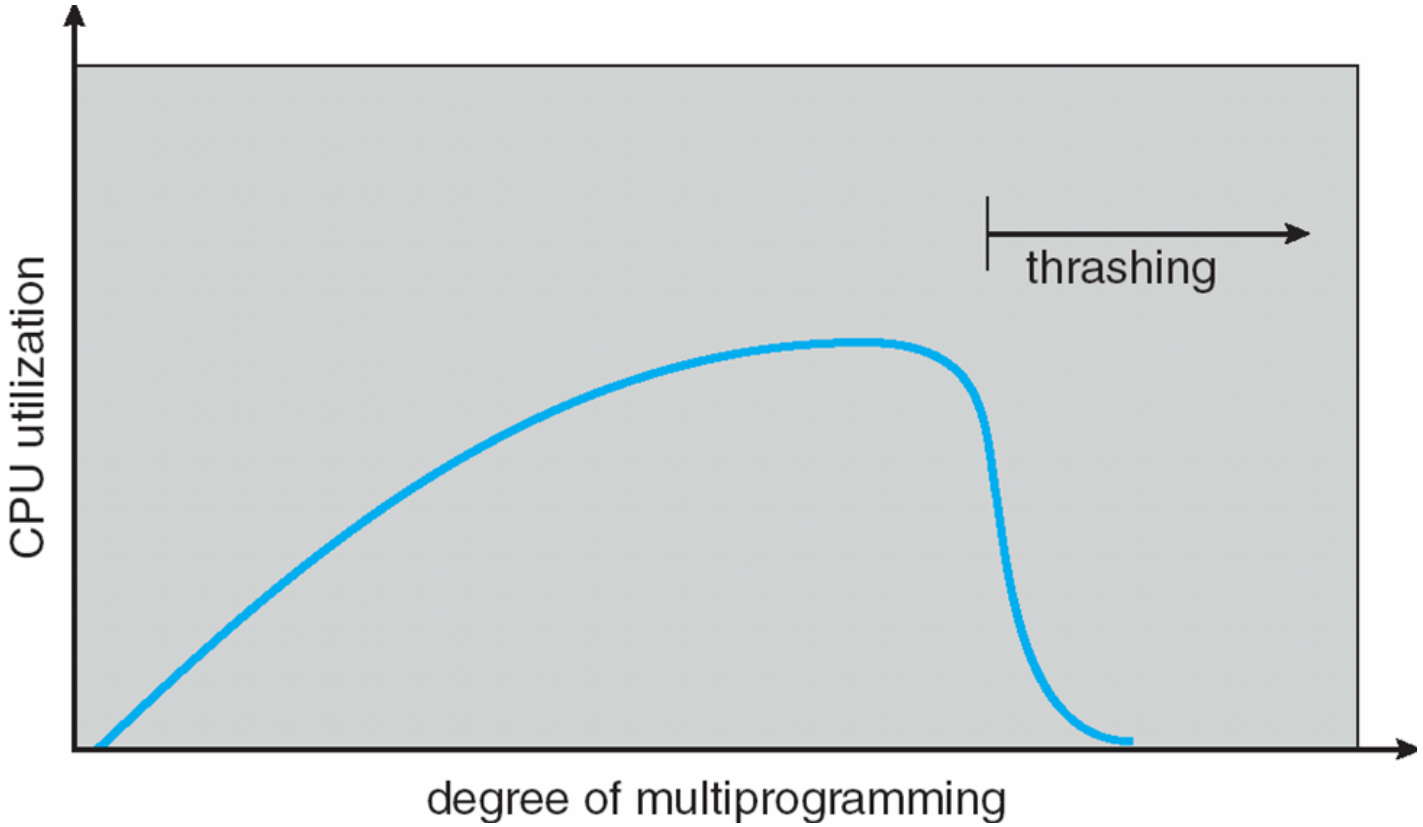
Global versus Local Allocation

- When a page fault occurs for a process and we need page replacement, there are two general approaches:
  - **Global replacement** – can select a victim frame from the set of all frames;
    - one process can take a frame from another
  - **Local replacement** – can select a victim frame only from the frames allocated to the process.
    - A process uses always its allocated frames

Thrashing

- If a process does not have "enough" pages, the **page-fault rate** is very high. This leads to:
  - low CPU utilization
  - Scheduler may think that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)

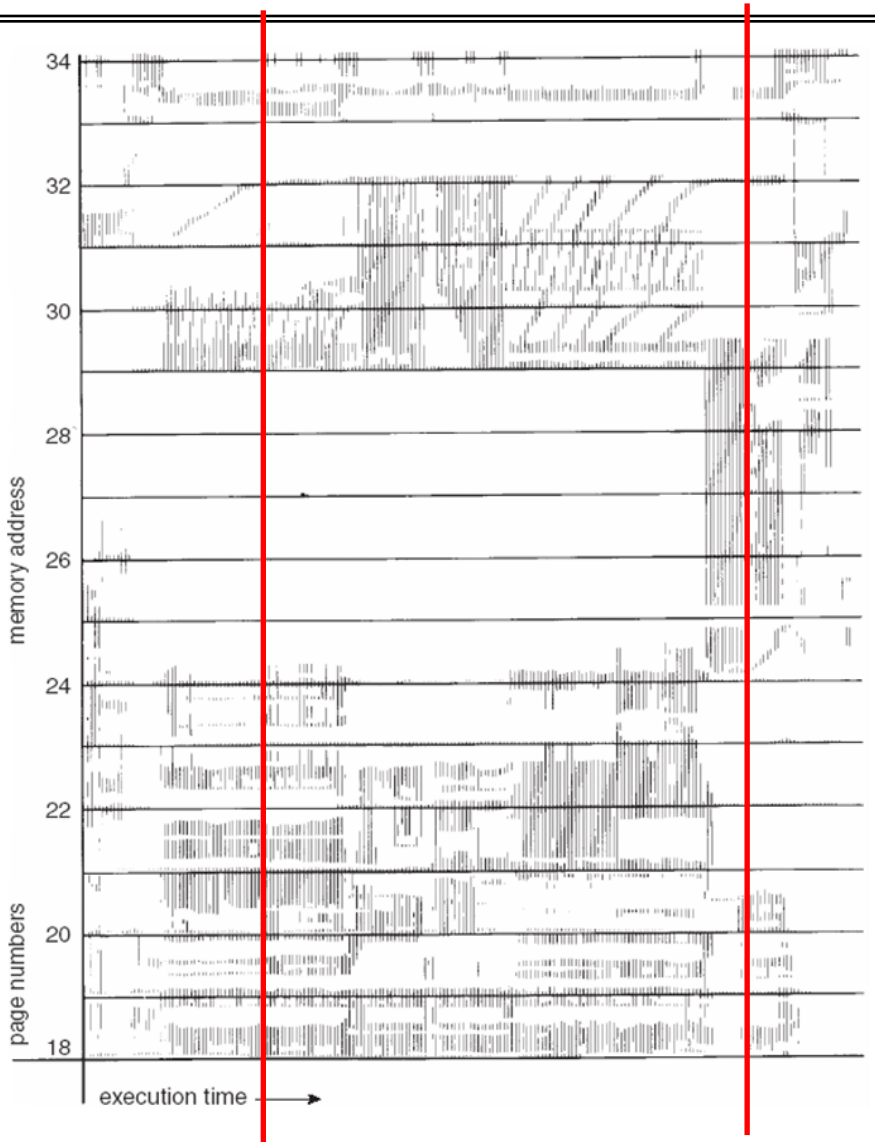


Demand Paging and Thrashing

- Why does demand paging work?
  Locality model (locality of reference)
  - Process migrates from one locality to another
  - Locality: set of pages actively used
  - Localities may overlap

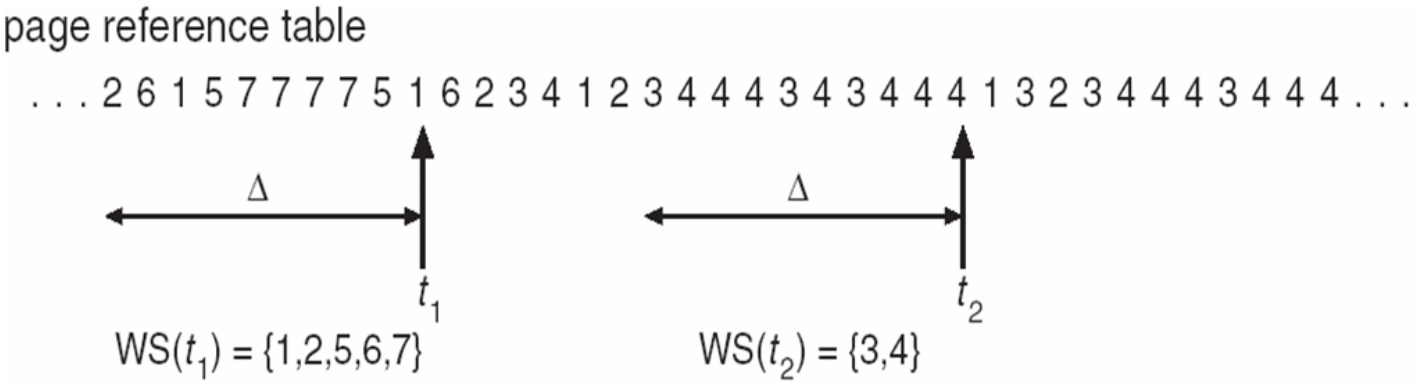
- Why does thrashing occur?
  Σ size of locality > total physical memory size

Locality In A Memory-Reference Pattern



Working-Set Model

- A method for deciding
  a) how many frames to allocate to a process, and also
  b) for selecting which page to replace.
- Maintain a Working Set (WS) for each process.
  - Look to the past Δ page references
  - Δ ≡ working-set window ≡ a fixed number of page references.
  - Working set: the pages process is using around this time



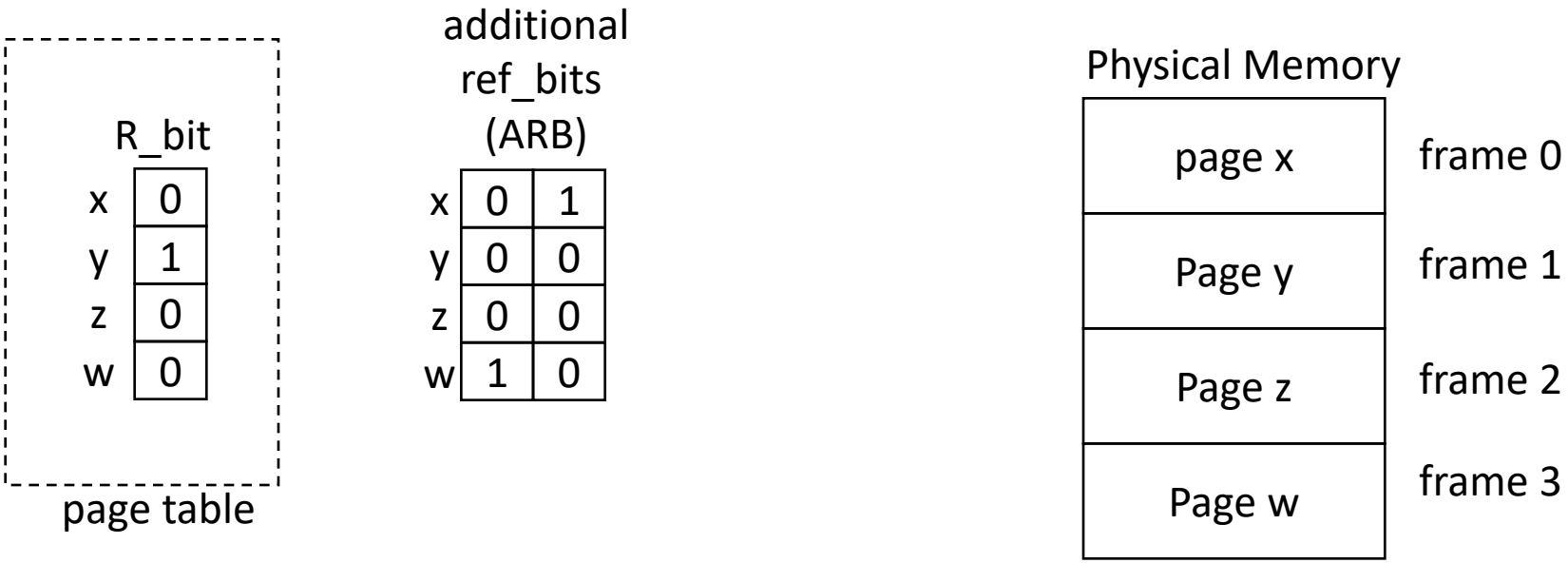
Working-Set Model

- WSS_i (working set size of Process P_i) = total number of distinct pages referenced in the most recent Δ
  - WSS varies in time
  - Value of Δ is important
    - 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
- $D = \Sigma\ WSS_i \equiv$ total demand for frames
- if $D > m \Rightarrow$ Thrashing (m: #frames in memory)
- A possible policy: if $D >$ m, then suspend one of the processes.

Keeping Track of Working-Set a method

- Approximate with interval timer + R bit and ARB field
- Example:
  - Δ = 10,000 (time units)
  - Assume timer interrupts after every 5000 time units.
  - Keep 2 reference bits for each page
  - Whenever timer interrupts, shift R into ARB and clear R.
  - If R, ARB has at least one 1 $\Rightarrow$ page in working set.
- We can increase granularity by increasing the size of ARB and decreasing the timer interrupt interval

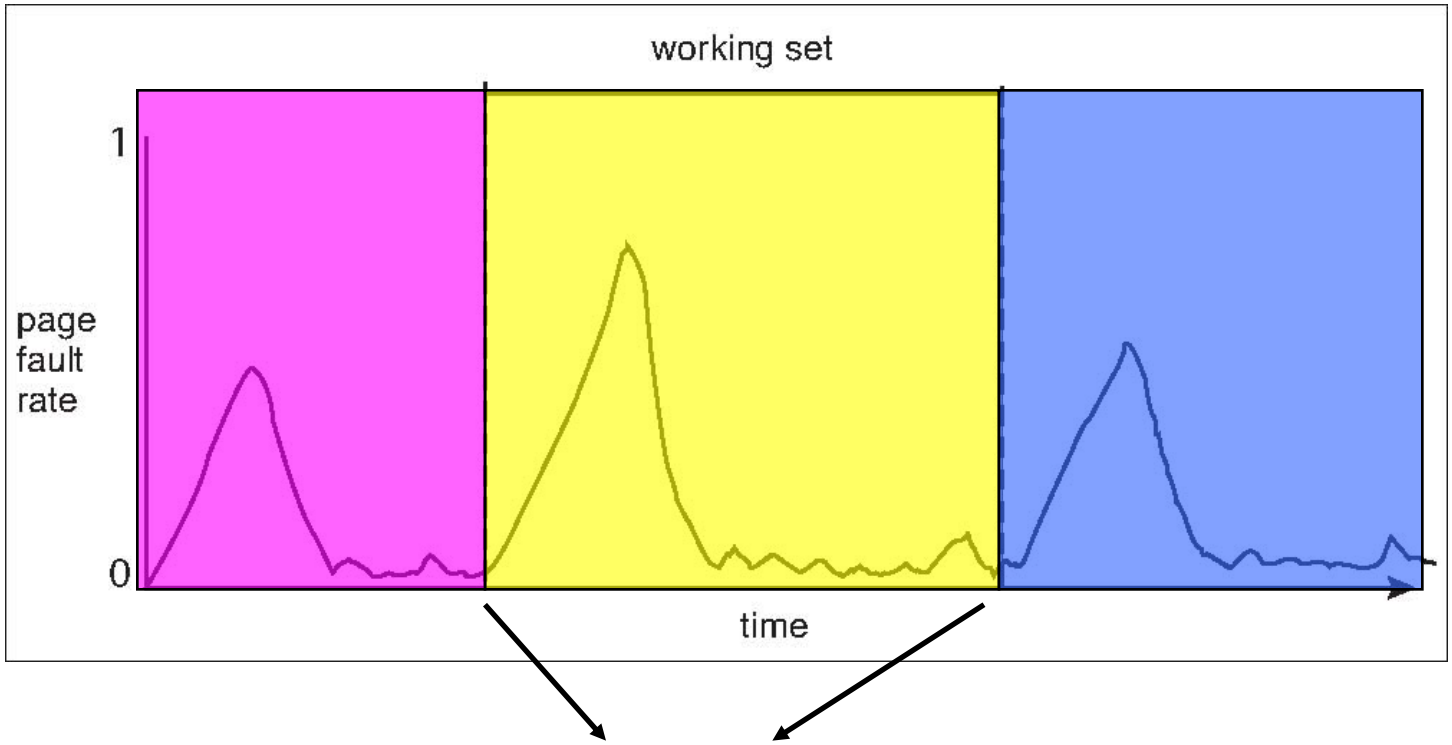
Keeping Track of Working-Set
a method



pages x, y, w in working set at the moment.

ARB is 2 bits here, but could be more (like 8 bits)

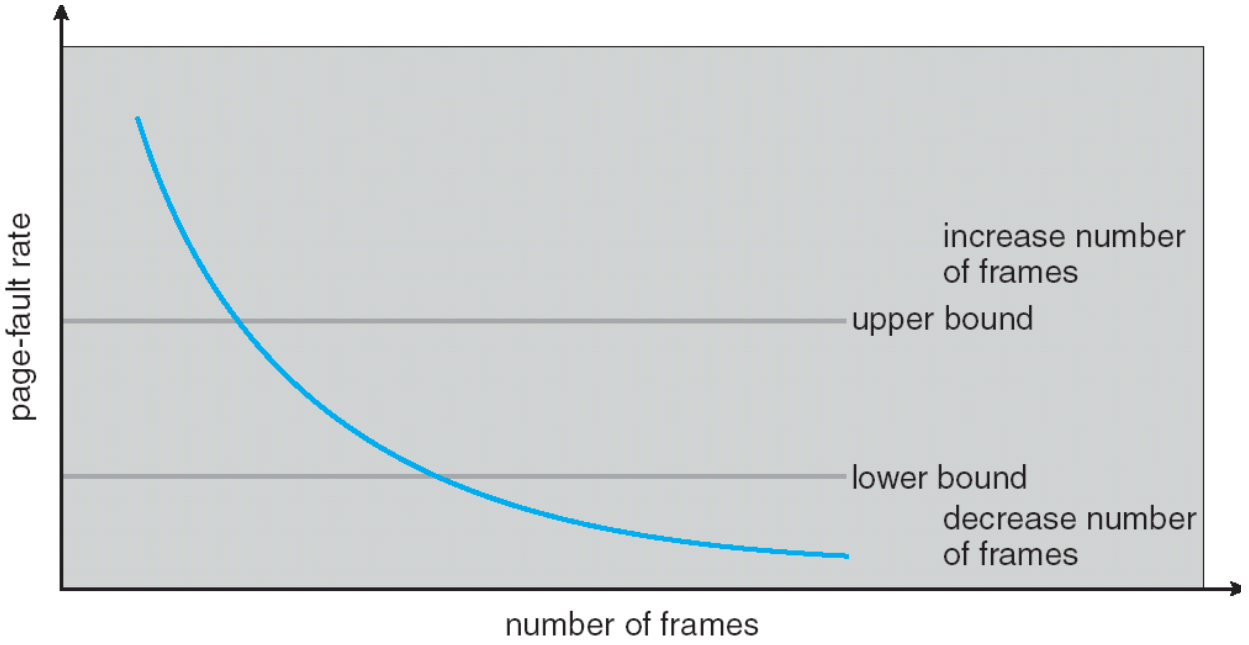
Working Sets and Page Fault Rates



transition from one working set to another

Page-Fault Frequency (PFF) Scheme

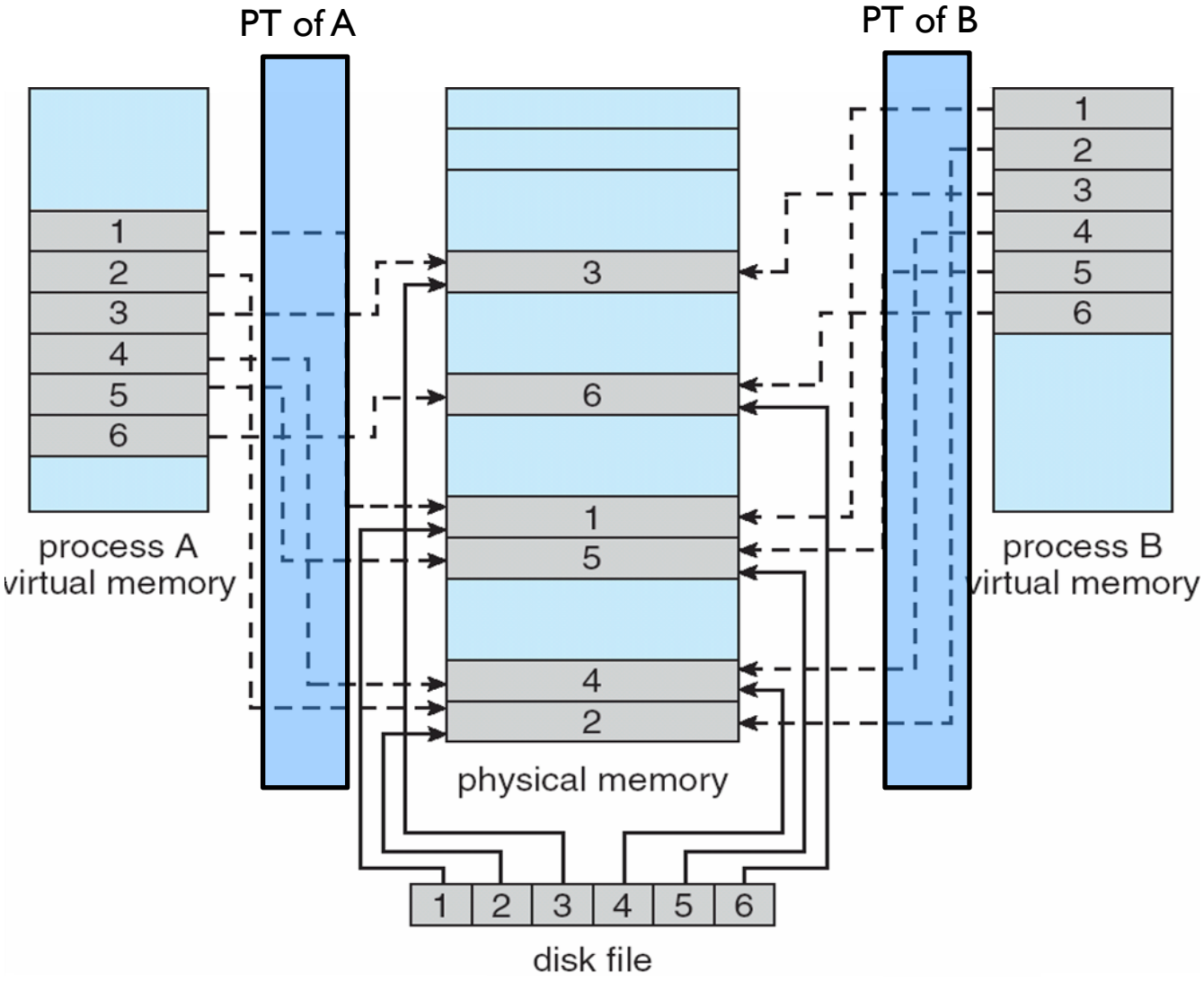
- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



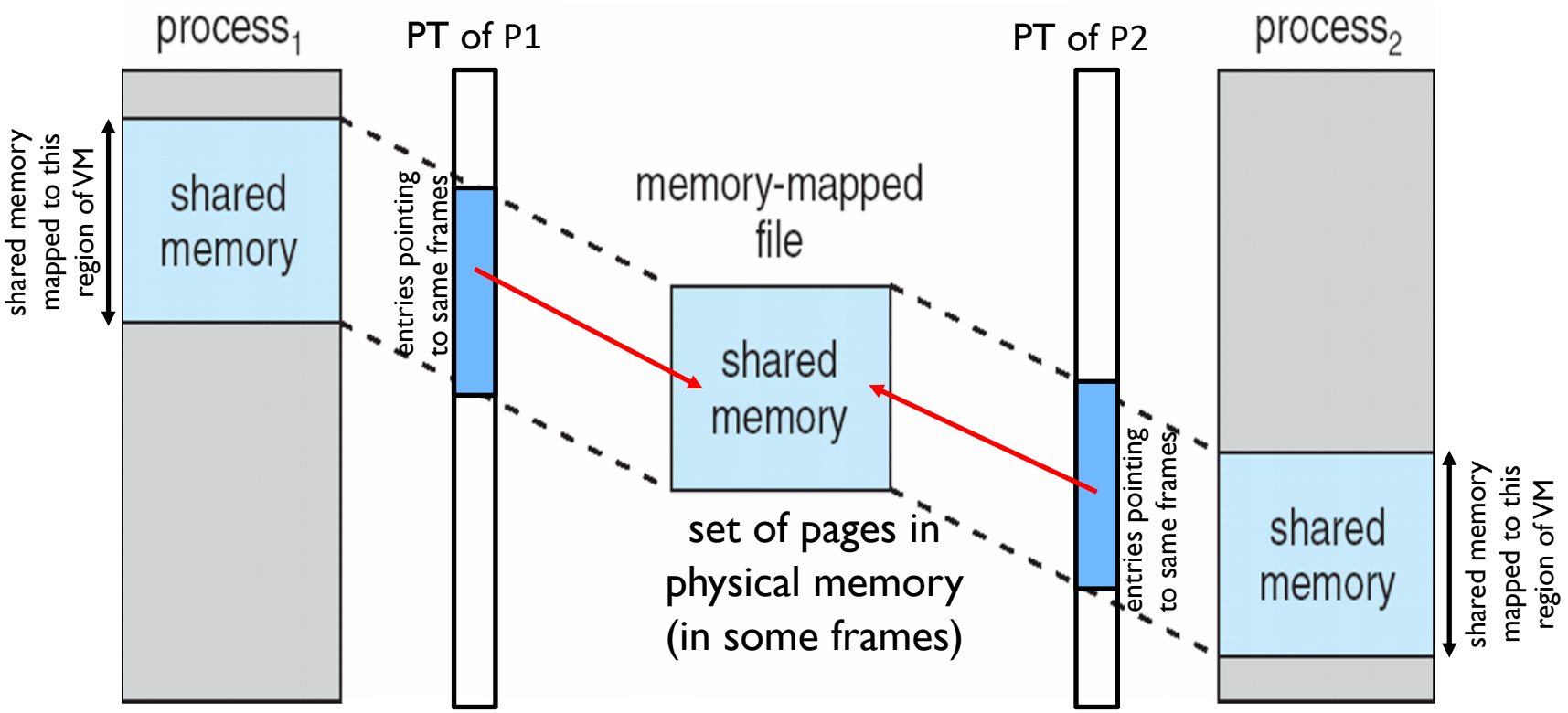
Memory-Mapped Files

- A file can be mapped to a virtual memory region of a process
  - One or more disk blocks are mapped to a page in memory.
- Memory-mapped file I/O allows file I/O to be treated as routine memory access.
- 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.
- 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



Shared memory can be implemented with memory mapping.

Allocating Kernel Memory

- Treated differently from user memory.
- Often allocated from a free-memory pool
- Kernel requests memory for its structures (objects) of varying sizes.
  - Object types: process descriptors, semaphores, file objects,…
- Allocation for objects of the same type (size) needed many times.
- A lot of kernel structures have sizes much less than the page size.

Allocating Kernel Memory

- Some kernel memory needs to be contiguous
- This is *dynamic memory allocation* problem.
- Hole-list approach (first-fit like strategies) is not efficient.
- Need for alternate strategies.

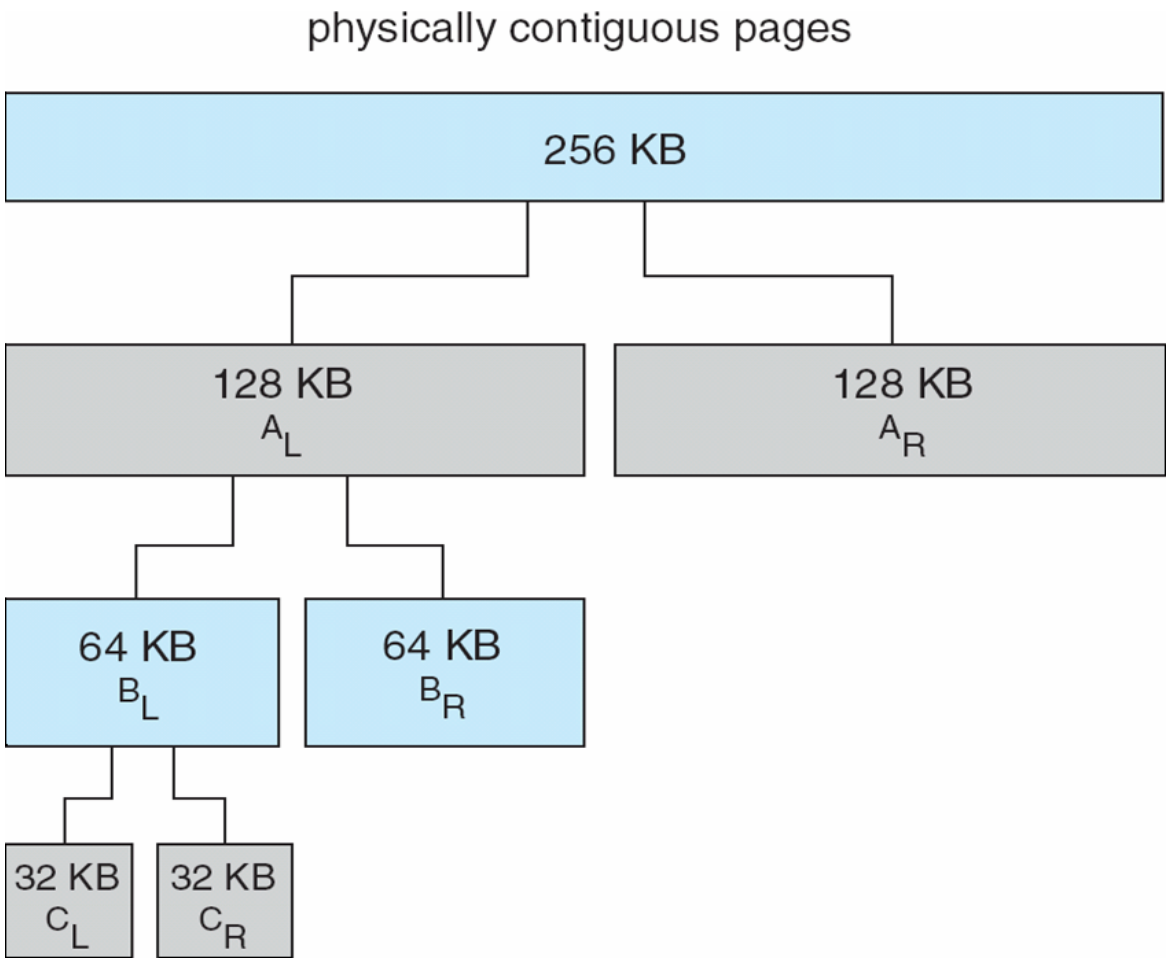
Allocating Kernel Memory

- We will see two methods
  - Buddy System Allocator
  - Slab Allocator

Buddy System Allocator

- 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 free chunk split into two **buddies** of next-lower power of 2
    - Continue until appropriate sized free chunk obtained

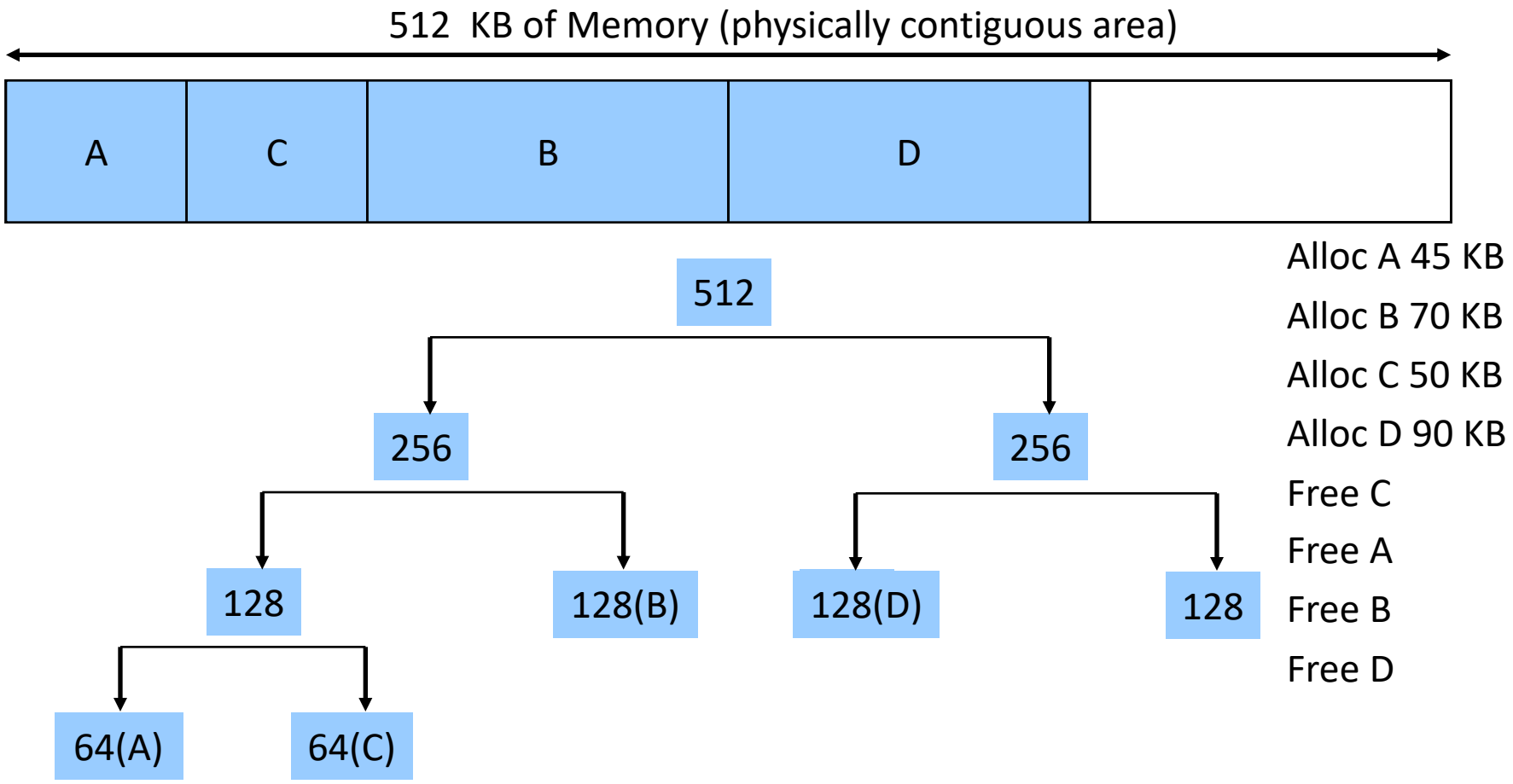
Buddy System Allocator



Example

- A needs memory 45 KB in size
- B needs memory 70 KB in size
- C needs memory 50 KB in size
- D needs memory 90 KB in size
-
- C removed
- A removed
- B removed
- D removed

Example

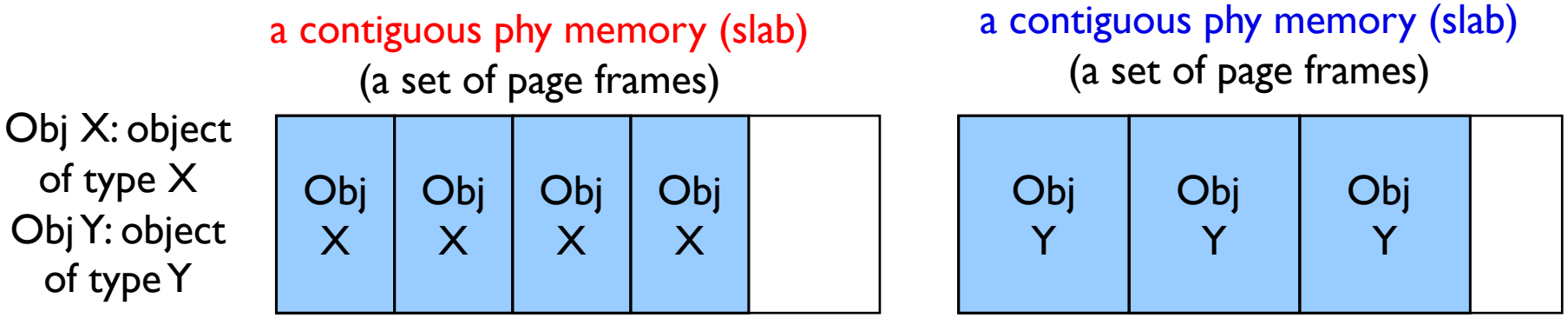

512 KB of Memory (physically contiguous area)
A
C
B
D
512
256
256
128
128(B)
128(D)
128
64(A)
64(C)
Alloc A 45 KB
Alloc B 70 KB
Alloc C 50 KB
Alloc D 90 KB
Free C
Free A
Free B
Free D

Linux and Buddy System

- Linux kernel uses buddy system algorithm to manage the physical memory.
- Allocates chunks of contiguous physical memory using buddy algorithm.
  - A chunk: one or more contiguous page frames (size of chunk = power of 2).
- Hence allocation unit is 1 page at least.
- Those chunks can be used for various purposes.
  - For caches
  - For user programs
  - For I/O transfers, etc.

Slab Allocator

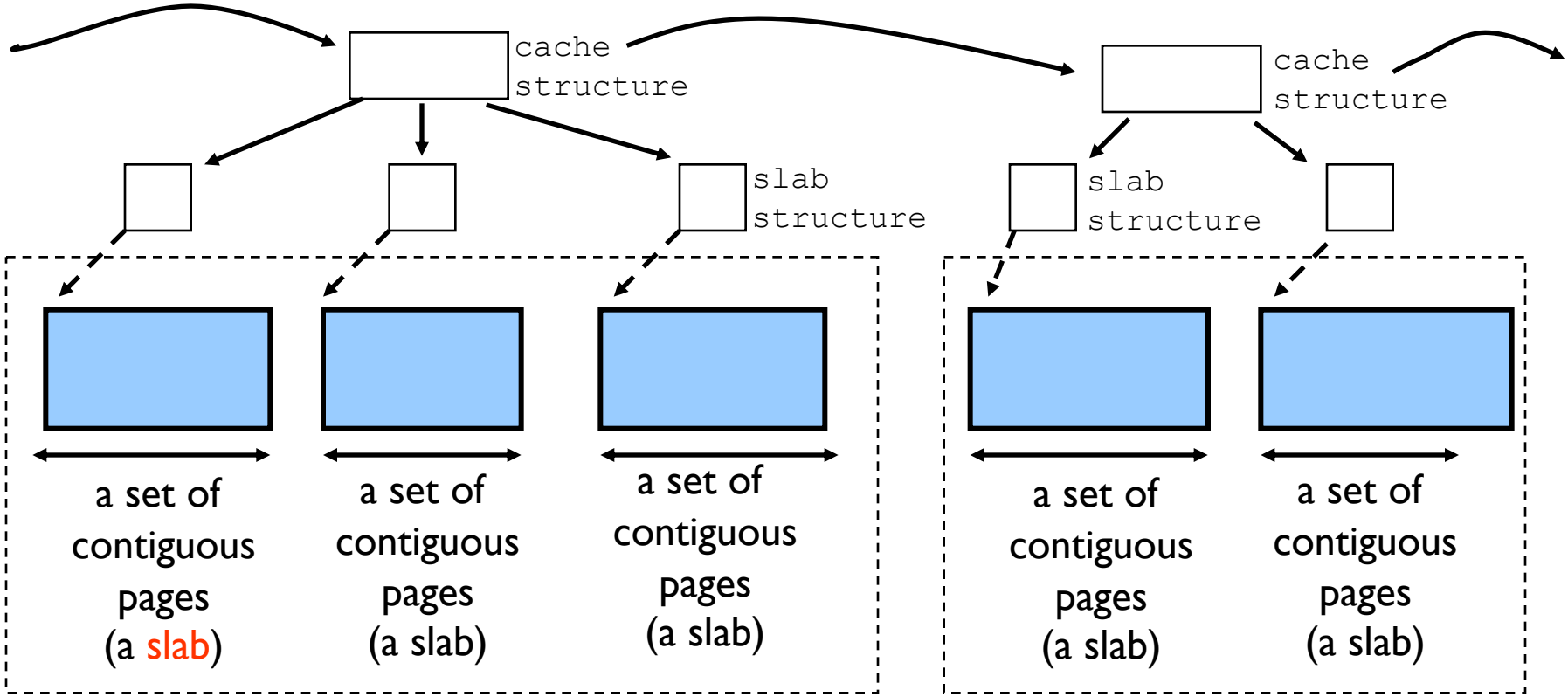
- Benefits: no fragmentation; fast allocation. for kernel objects (smaller than pagesize).
- Within kernel, memory needed (dynamically) for a finite set of object types, such as process descriptors, file descriptors, semaphores, inodes, other common structures.
- Store objects of the same type in a set of contiguous page frames (called slab).



Slab Allocator

- Slab is one or more physically contiguous pages. Actual location of objects.
- A cache consists of one or more slabs (i.e., a cache structure pointing to slabs).
- We have single cache for each unique kernel data structure
  - Ex: task structure, semaphores, locks, file descriptors, inodes, memory-man-structures, 64-bit data types, ….
- When cache created, the respective slabs are filled with slots (objects) marked as free. (slot-size == object-size).
- When objects stored, slots marked as used
- If slab is full, next object allocated from an empty slab. If no empty slab, new slab allocated.

Slabs and Caches



set of slabs containing same type of objects (a cache) (can store objects of type/size X)

a set of slabs (another cache) (can store objects of type/size Y)

Cache structure in Linux

- Cache structure is a structure that keeps information about the cache and includes pointers to the slabs.

```
struct kmem_cache_s {
        struct list_head slabs_full; /* points to the full slabs */
        struct list_head slabs_partial;/* points to the partial slabs */
        struct list_head slabs_free; /* points to the free slabs */
        unsigned int objsize; /* size of objects stored in this cache */
        unsigned int flags;
        unsigned int num;
        spinlock_t spinlock;
        …
        …
}
```

Slab structure in Linux

- A slab stucture is a data structure that points to a contiguous set of page frames (a slab) that can store some number of objects of same size.
- A slab can be considered as a set of slots (slot size = object size). Each slot in a slab can hold one object.
- Which slots are free are maintained in the slab structure

```
typedef struct slab_s {
        struct list_head list;
        unsigned long colouroff;
        void *s_mem;        /* start address of first object */
        unsigned int inuse;     /* number of active objects */
        kmem_bufctl_t free;  /* info about free objects */
} slab_t;
```

Slab Allocator in Linux

- cat **/proc/slabinfo** will give info about the current slabs and objects

cache names: one cache for each different object type

```
# name            <active_objs> <num_objs> <objsize> <objperslab> <pagesperslab> : tunables <limit> <batchcount> <
sharedfactor> : slabdata <active_slabs> <num_slabs> <sharedavail>
ip_fib_alias           15    113     32  113    1 : tunables  120   60    8 : slabdata      1      1      0
ip_fib_hash            15    113     32  113    1 : tunables  120   60    8 : slabdata      1      1      0
dm_tio                  0      0     16  203    1 : tunables  120   60    8 : slabdata      0      0      0
dm_io                   0      0     20  169    1 : tunables  120   60    8 : slabdata      0      0      0
uhci_urb_priv           4    127     28  127    1 : tunables  120   60    8 : slabdata      1      1      0
jbd_4k                  0      0   4096    1    1 : tunables   24   12    8 : slabdata      0      0      0
ext3_inode_cache   128604 128696    504    8    1 : tunables   54   27    8 : slabdata  16087  16087      0
ext3_xattr          24084  29562     48   78    1 : tunables  120   60    8 : slabdata    379    379      0
journal_handle         16    169     20  169    1 : tunables  120   60    8 : slabdata      1      1      0
journal_head           75    144     52   72    1 : tunables  120   60    8 : slabdata      2      2      0
revoke_table            2    254     12  254    1 : tunables  120   60    8 : slabdata      1      1      0
revoke_record           0      0     16  203    1 : tunables  120   60    8 : slabdata      0      0      0
scsi_cmd_cache         35     60    320   12    1 : tunables   54   27    8 : slabdata      5      5      0
….
files_cache           104    170    384   10    1 : tunables   54   27    8 : slabdata     17     17      0
signal_cache          134    144    448    9    1 : tunables   54   27    8 : slabdata     16     16      0
sighand_cache         126    126   1344    3    1 : tunables   24   12    8 : slabdata     42     42      0
task_struct           179    195   1392    5    2 : tunables   24   12    8 : slabdata     39     39      0
anon_vma             2428   2540     12  254    1 : tunables  120   60    8 : slabdata     10     10      0
pgd                    89     89   4096    1    1 : tunables   24   12    8 : slabdata     89     89      0
pid                   170    303     36  101    1 : tunables  120   60    8 : slabdata      3      3      0
```

active objects

size

Prepaging

- Prepaging
  - Prepage all or some of the pages a process will need, before they are referenced
  - If prepaged pages are not used, I/O and memory wasted
- Assume *n* pages are prepaged and ratio of pages used is α
  - C = cost of prepaging n x (1- α) unused pages
  - G = gain from n x α used pages (we prevented page faults for them).
  - Compare C and G to see if prepaging is helping.

Other Issues – Page Size

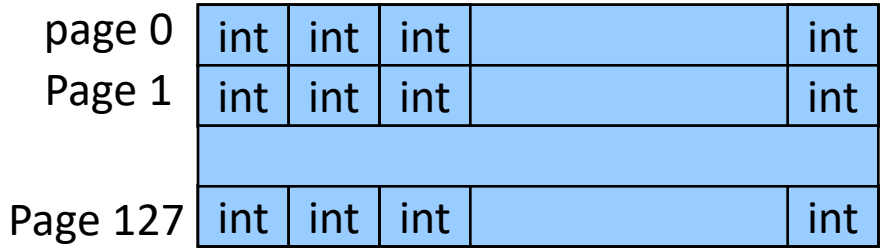
- Page size selection must take into consideration:
  - Fragmentation
    - Small page size reduces fragmentation
  - Table size
    - Large page size reduces page table size
  - I/O overhead
    - Large page size reduce I/O overhead (seek time, rotation time)
  - Locality
    - Locality is improved with smaller page size.

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 number of visits to the pagetable
- To increase TLB reach:
  - Increase number of entries (not easy)
  - Increase the page size (fragmentation may increase)
  - Provide multiple page sizes.

Other Issues – Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
    assuming pagesize=512 bytes



- 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

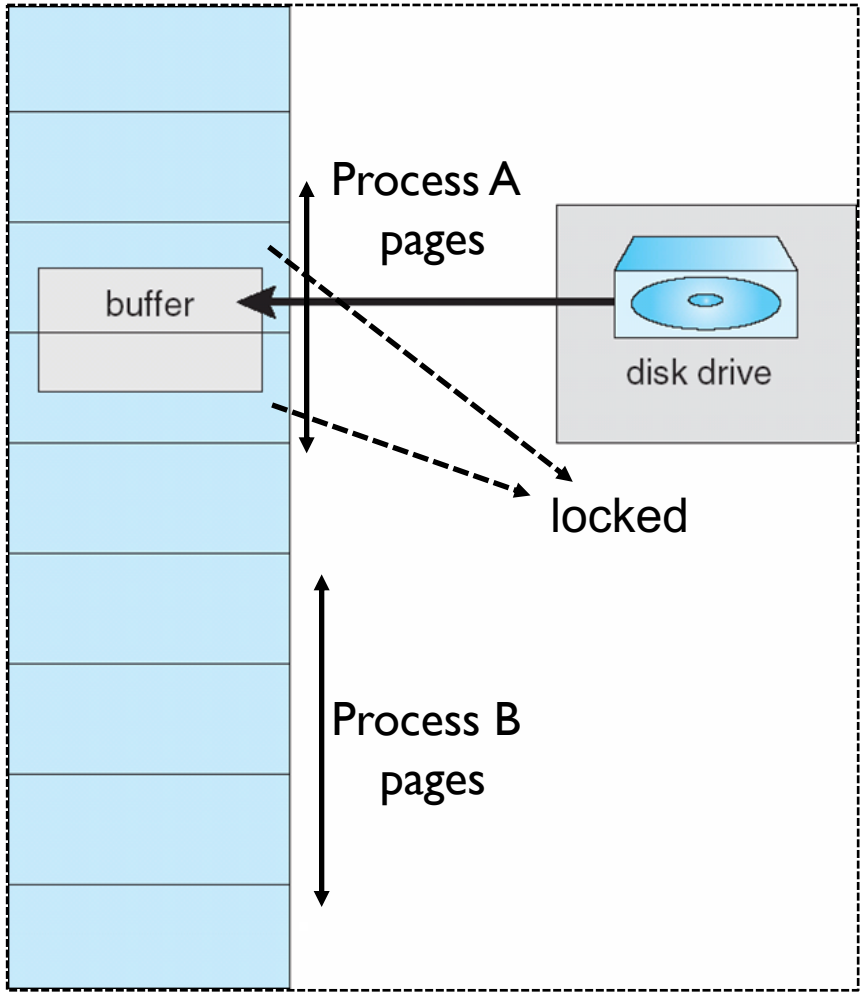
- Program 2

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

128 page faults

Other Issues – Locking pages

- Sometimes, pages must be locked into memory (pinned).
- A locked page can not be replaced.
- Consider I/O: pages that are used for copying a file from a device must be locked.

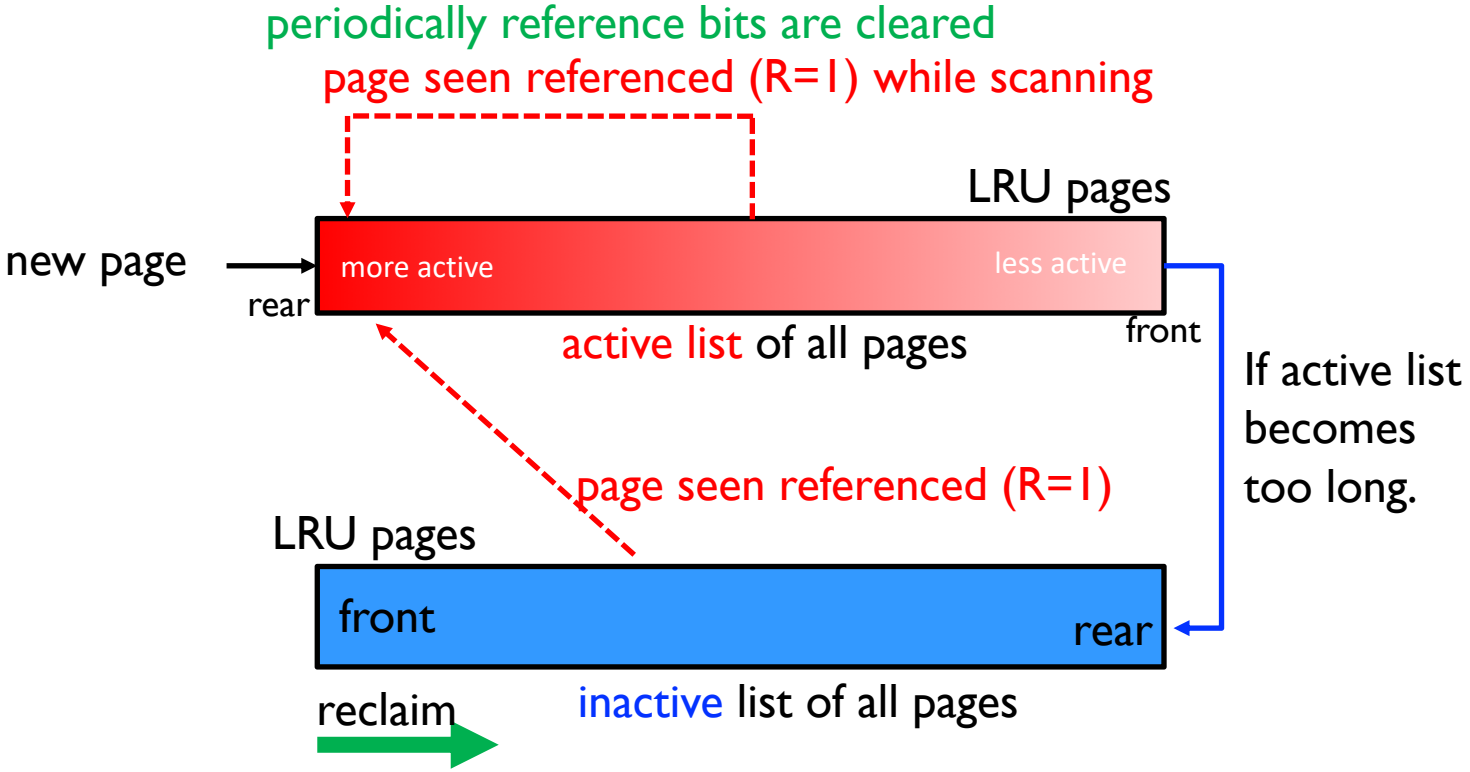


If B requires a new frame, A's locked pages can not be removed.

Linux page replacement

- Uses global allocation
- A variant of LRU similar to clock algorithm
- Keeps a free frame list ready to allocate from.
- Populates free frame list when necessary (via page reclaiming)
  - When free memory drops below a threshold.
- 2 lists of pages
  - Active list (pages are in active use)
  - Inactive list (pages removed – reclaimed – from here)
- R bits utilized. Periodically cleared.
- Pages scanned from time to time. if R = 1 for a pages, that page is seen used.

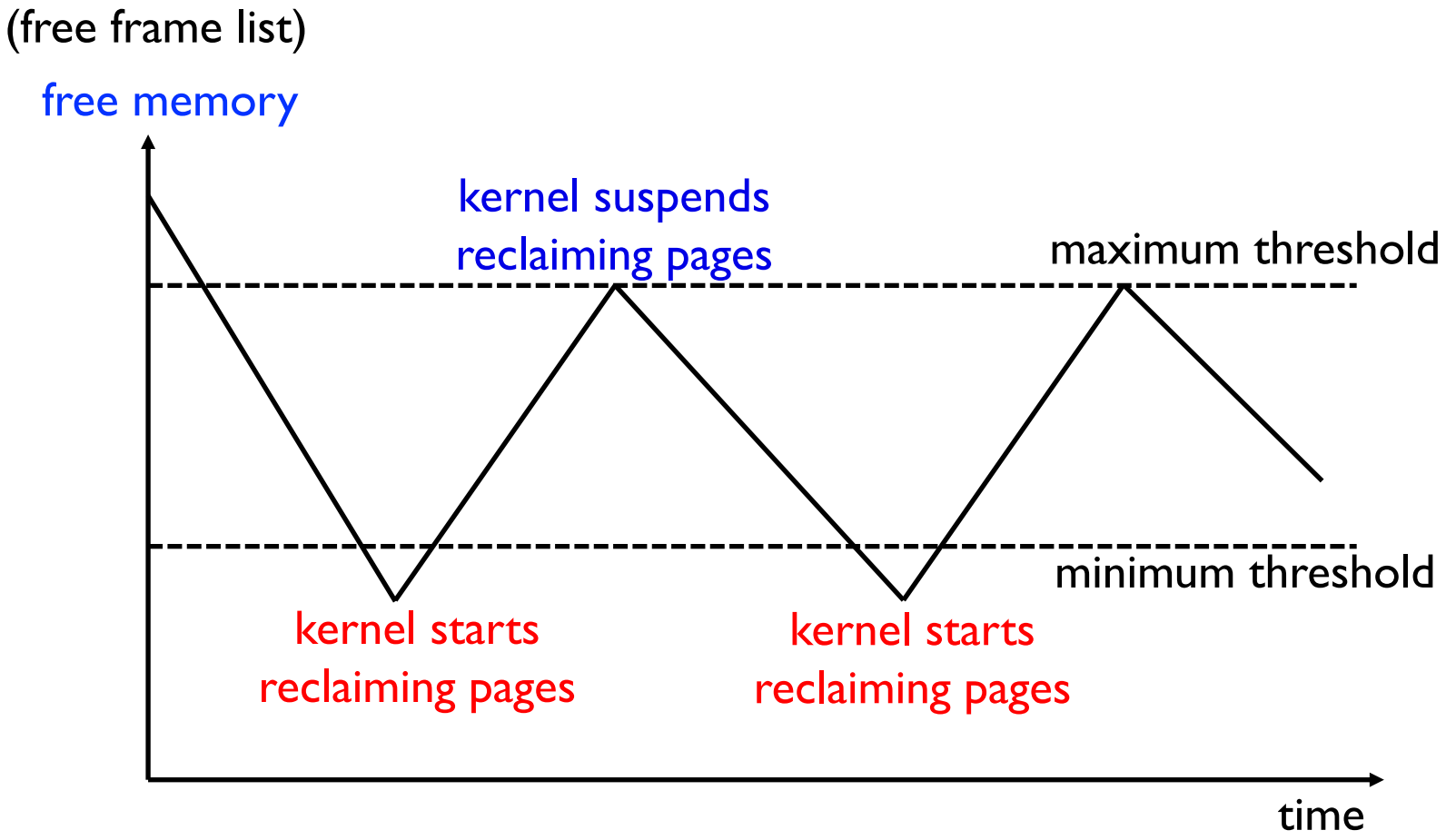
Linux page replacement



Pages reclaimed (removed) when needed. That means the respective frames become empty. Those frames are put into Free Frame List

Nearly all pages (frames) in the system are in one of two queues.

Linux page replacement



reclaiming (making some frames empty) done globally.

References

- Operating System Concepts, Silberschatz et al. Wiley.
- Modern Operating Systems, Andrew S. Tanenbaum et al.
- Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau et al.

Additional Slides (optional)

Solaris

- Maintains a list of free pages to assign faulting processes
- *Lotsfree* – threshold parameter (amount of free memory) to begin paging
- *Desfree* – threshold parameter to increasing paging
- *Minfree* – threshold parameter to being swapping
- Paging is performed by *pageout* process
- Pageout scans pages using modified clock algorithm
- *Scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- Pageout is called more frequently depending upon the amount of free memory available

Solaris Page Scanner