Chapter 9: Virtual Memory

Allow the OS to hand out more memory than existing physical memory

Thanks to the author of the textbook [**SGG**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.
Turgay Korkmaz

Chapter 9: Virtual Memory

Background	*
Demand Paging	*****
Copy-on-Write	*
Page Replacement	*****
Memory-Mapped Files	***
Allocation of Frames	**
Thrashing	**
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
- To consider other issues affecting the performance

Background

- (**CH 8**) A process must be in physical memory
  - How to run a large program that does not fit into physical memory?
  - Observation: Not all code or data needed at the same time
    - Error handling codes
    - Big arrays with max size
    - Some options might not be needed at least at the same time
- **Virtual memory**
  - **Allows execution of processes that are not completely in the main memory**
    - *What are the benefits of executing a program which is partially in memory?*
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

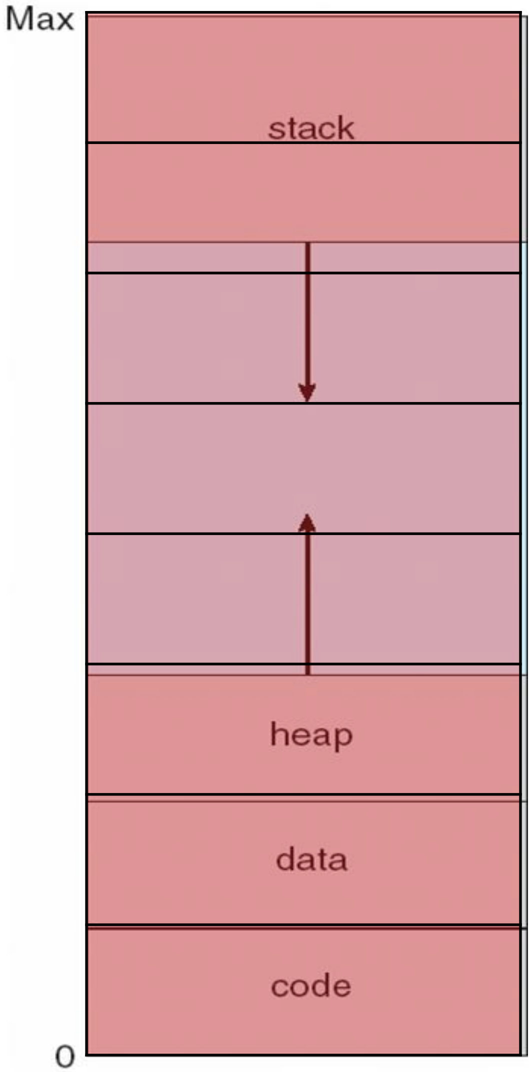
Benefits of Virtual Memory

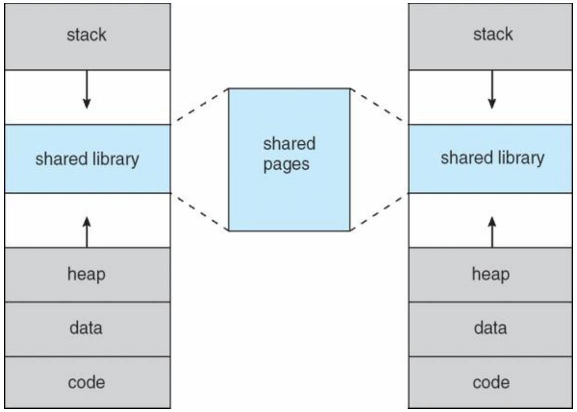
- User will have a very large logical address space
- User can execute programs larger than physical memory
- Especially helpful in multiprogrammed systems
  - Multiple processes can be executed concurrently because
  - Each process occupies small portion of memory
    - The only part of the program needs to be in physical memory is the one that is needed for execution at a given time
- Less I/O to load or swap user programs
- Physical Memory de/allocation
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - Keep recently used content in physical memory
  - Move less recently used stuff to disk
  - Movement to/from disk handled by the OS

Compare to swapping!

Virtual Memory

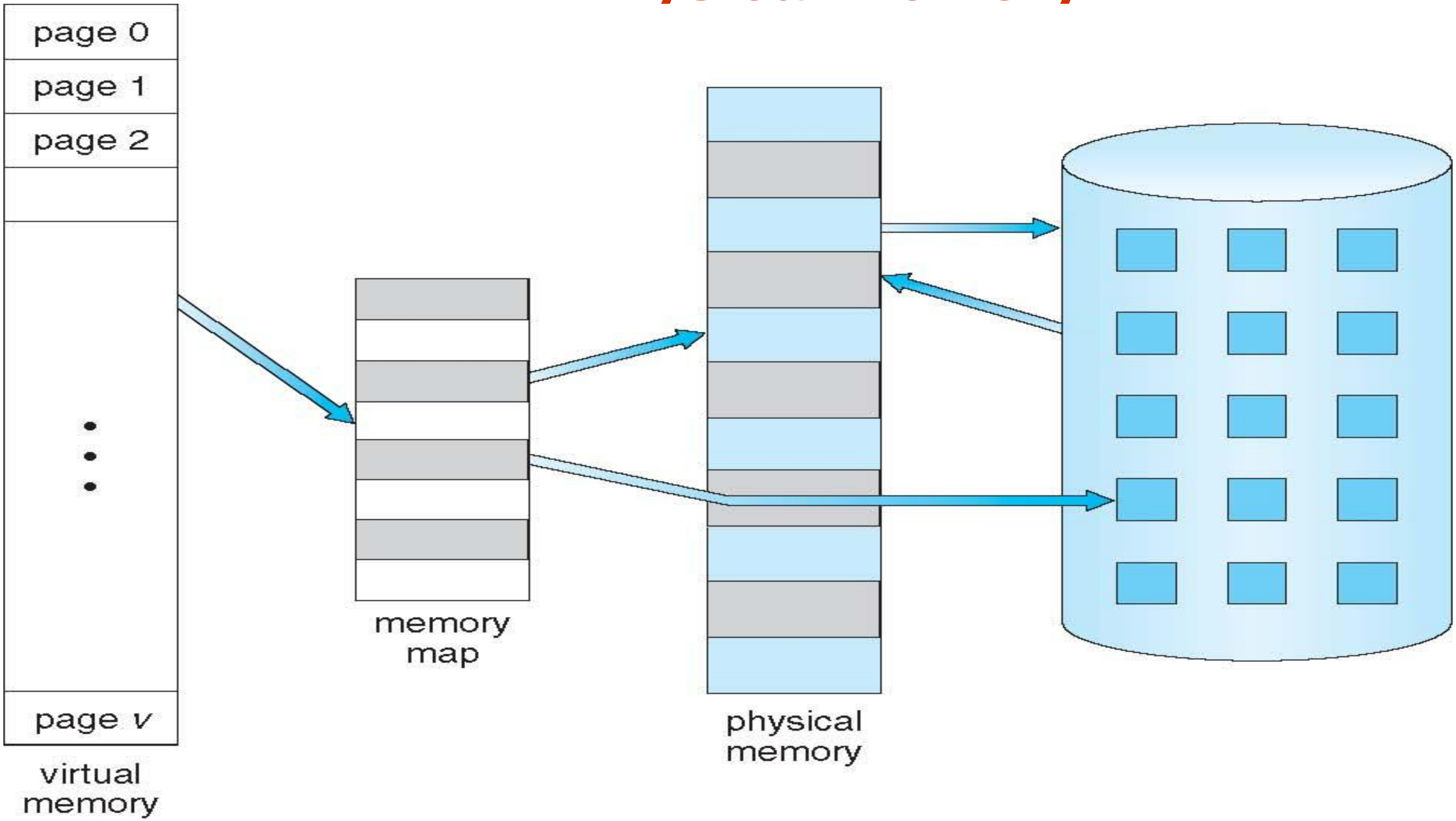
- Separation of user logical memory from physical memory
- Addresses local to the process
- Can be any size → limited by # of bits in address (32/64)
- Virtual memory >> physical memory
- Holes are part of virtual address space but require actual physical pages (frames) only when needed for growing heap stack or shared libs etc.





Natural extension of paging in CH 8

Virtual Memory That is Larger Than Physical Memory

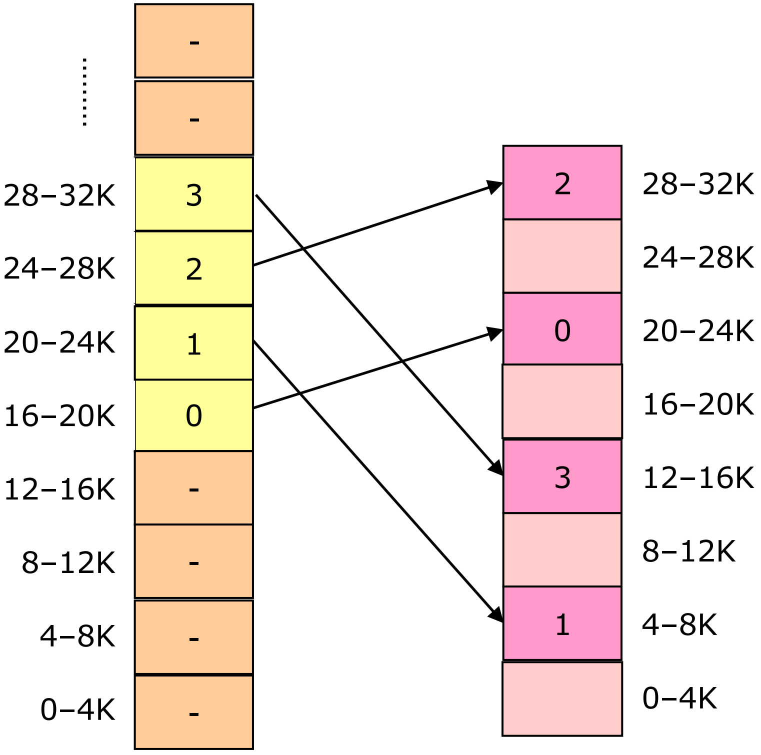


Natural extension of paging in CH 8

How to get physical address from the virtual one?!

Recall: Paging and Page Systems

- Virtual (logical) address
  - Divided into **pages**
- Physical memory
  - Divided into **frames**
- **Page vs. Frame**
  - **Same size** address blocks
  - Unit of mapping/allocation
- A page is mapped to a frame
  - All addresses in the same virtual page are in the same physical frame → **offset** in a page

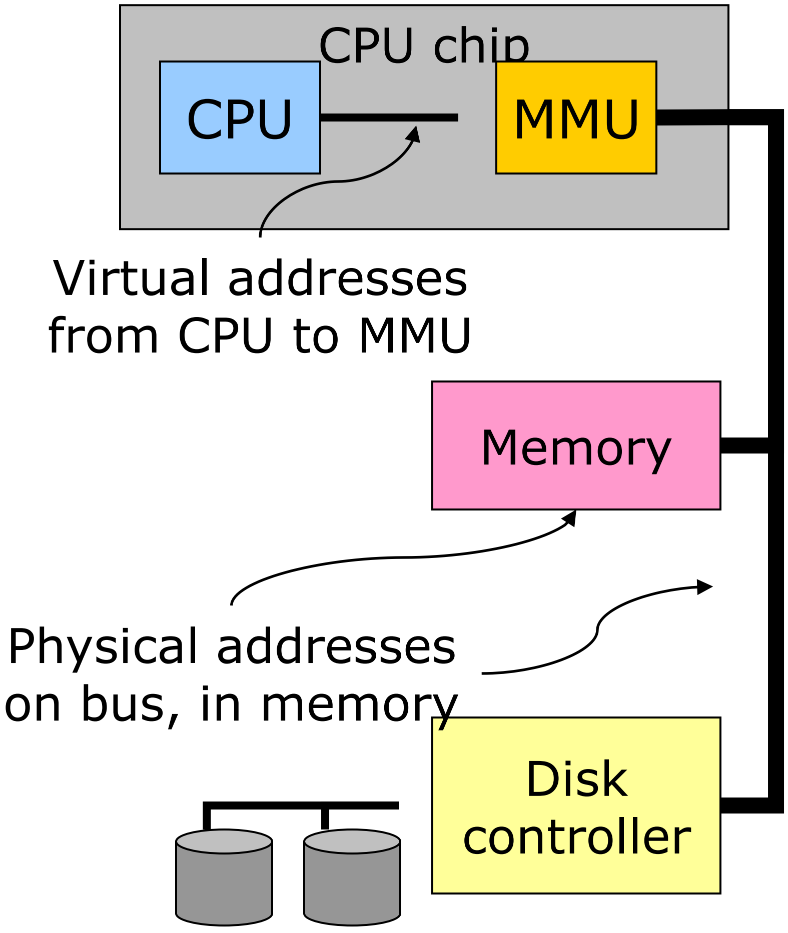


Virtual (Logical) memory

Physical memory

Virtual and Physical Addresses

same as in ch 8



- Virtual address space
  - Determined by instruction width
  - Same for all processes
- Physical memory indexed by physical addresses
  - Limited by bus size (# of bits)
  - Amount of available memory
- *Memory Management* ***U****nit* ***(MMU)***
  - Translation: virtual → physical addr.
  - Only physical addresses leave the CPU/MMU chip

How does MMU do the translation & what is needed?

*

Translate Virtual to Physical Address

same as in ch 8

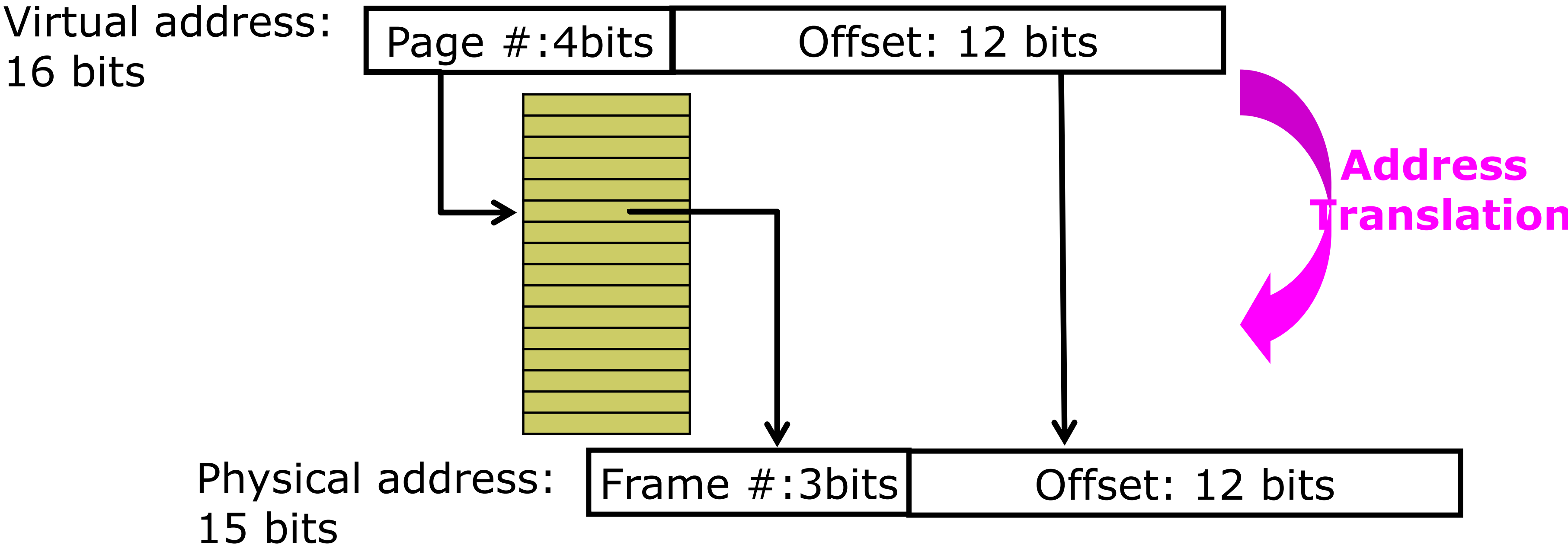
- Split virtual address (from CPU) into ***two*** pieces
  - Page number (*p*)
  - Page offset (*d*)
- **Page number**
  - Index into page table
  - Page table contains base address of page in physical memory
- Page **offset**
  - Added to base address to get actual physical memory address
- **Page size** = 2^d bytes: determined by offset size

An Example of Virtual/Physical Addresses

- Example:
  - 64 KB virtual memory (16-bit)
  - 32 KB physical memory (15-bit)
  - 4 KB page/frame size (12-bit) as offset (d)

How many pages?

How many frames?



Virtual address: 16 bits

Page #:4bits | Offset: 12 bits

Address Translation

Physical address: 15 bits

Frame #:3bits | Offset: 12 bits

How /when to load a page into memory

load everything at once (ch8)

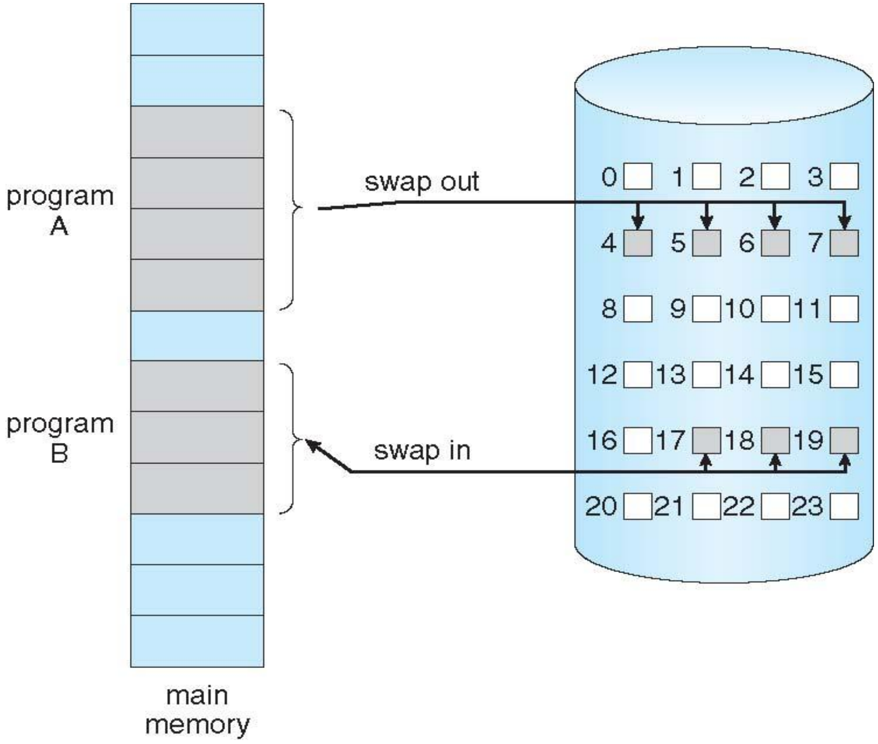
load as needed (ch9)

DEMAND PAGING

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 ⇒ reference to it
  - Valid in memory ⇒ use it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Demand Paging vs. Swapper**

  Page only vs. contiguous space
- **Lazy swapper** – bring only the pages that are needed



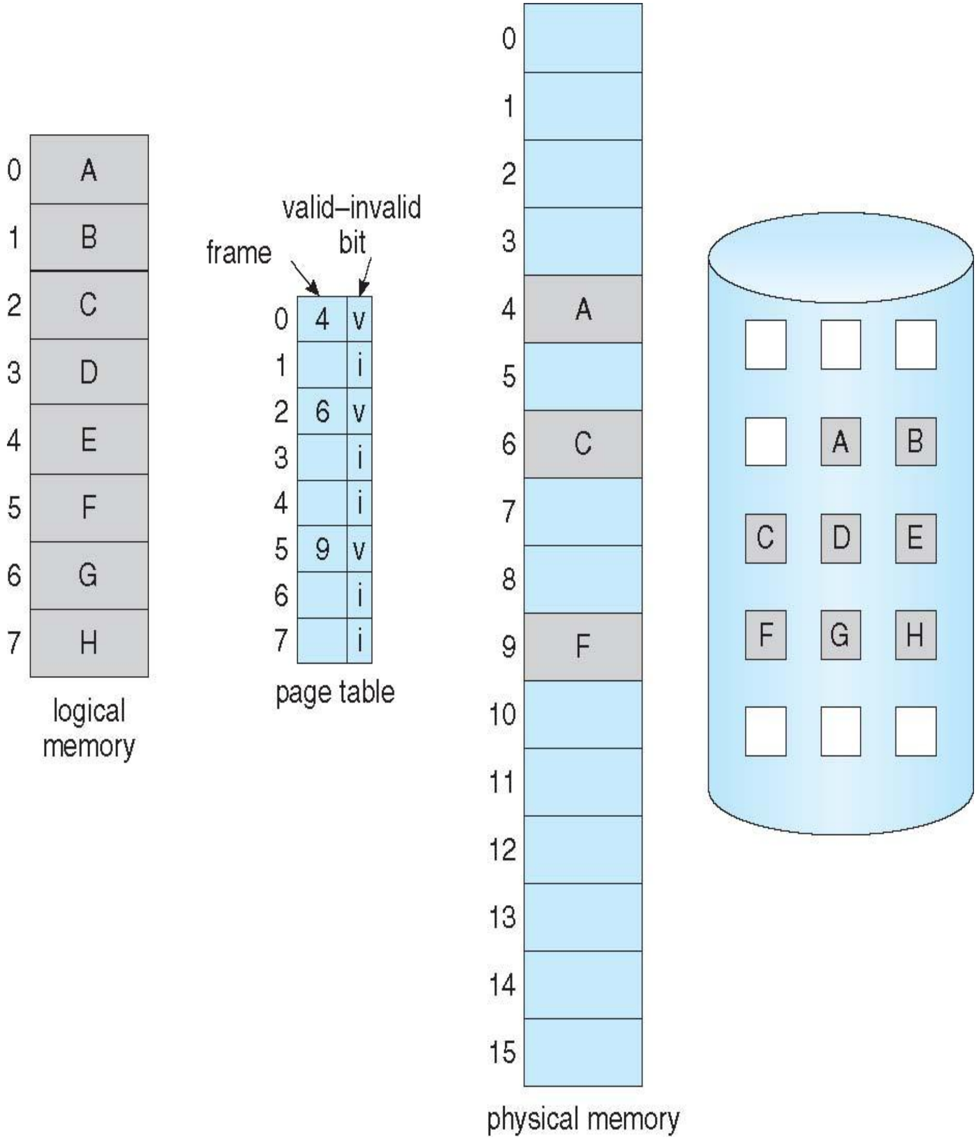
Valid-Invalid Bit

- 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
- During address translation,

  if valid–invalid bit in page table entry

  is **i** ⇒ page fault (trap)



Page Fault

1. Reference to a page,

   If Invalid reference ⇒ abort

2. If not in memory, page fault occurs *(trap to OS)*

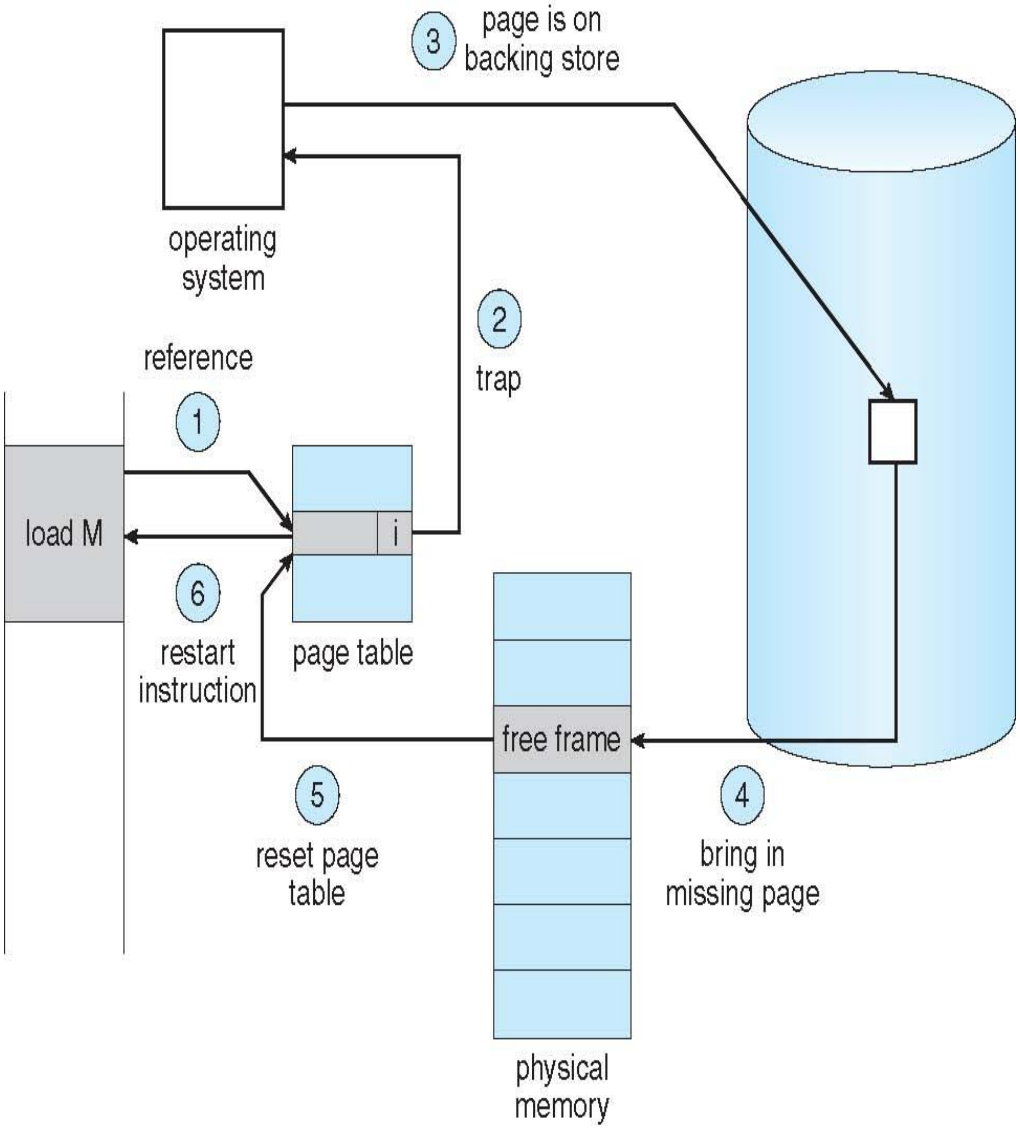
3. Operating system allocates an empty frame

4. Swap page into frame

5. Reset page tables,

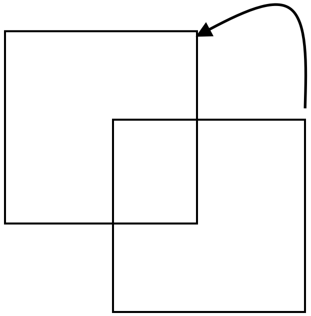
   set validation bit = ***v***

6. **Restart the instruction** that caused the page fault



Page Fault (Cont.)

- Restart instruction
  - During inst fetch, get the page and re-fetch
  - During operand fetch, get the page and re-fetch instruction
    - *(how many pages need depends on architecture, e.g., add a b c)*
  - But how about block move
    - Make sure both ends of the buffers are in the memory
    - Use temp buffer. If page fault occurs restore before re-starting



Performance of Demand Paging

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

  EAT = (1 – *p*) x memory_access + *p* x page_fault_time
- page_fault_time depends on several factors
  - Save user reg and proc state,
  - check page ref,
  - **read from the disk there might be a queue**, (CPU can be given to another proc),
  - get interrupt,
  - save other user reg and proc state,
  - correct the page table,
  - put this process into ready queue…..
  - **Due to queues, the page_fault_time is a random variable**

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!
- If we want just 10% performance degradation, then p should be

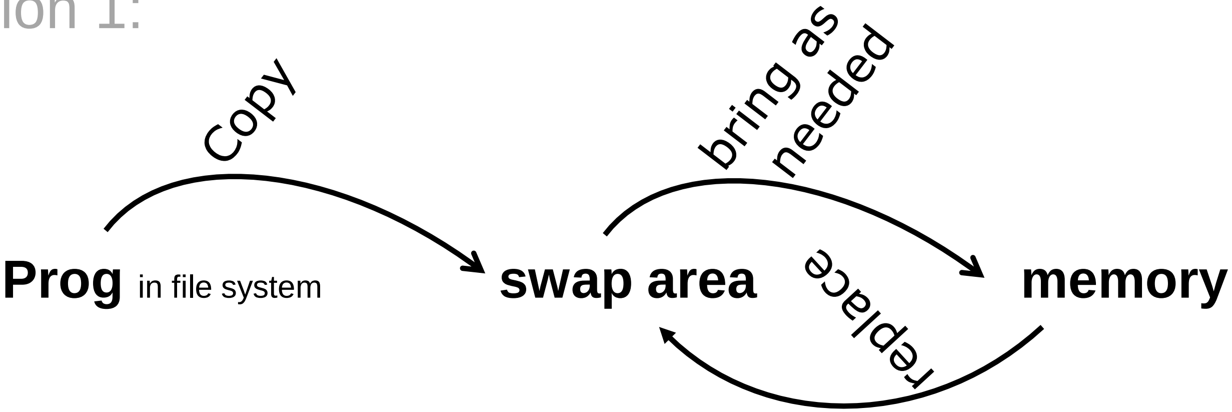
  220 > (1 – p) x 200 + p (8 milliseconds)

  p < 0.0000025 , i.e., 1 page fault out of 400,000 accesses

Disk I/O for Demand Paging

- Disk I/O to swap is generally faster than to the file system
  - Larger blocks, no indirect lookups etc.

Option 1:



Copy

bring as needed

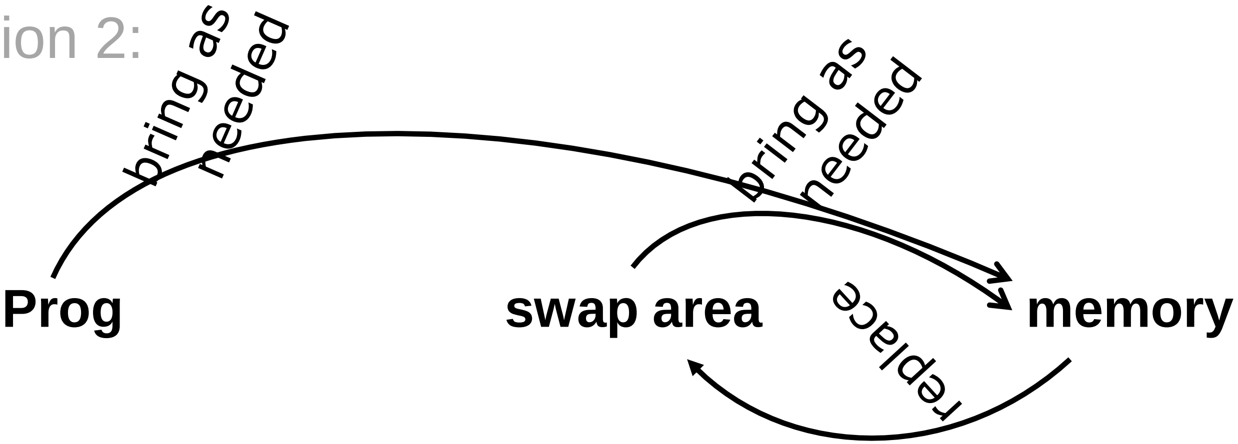
replace

Prog in file system

swap area

memory

Option 2:



bring as needed

bring as needed

replace

Prog

swap area

memory

Virtual memory has other benefits during process creation:

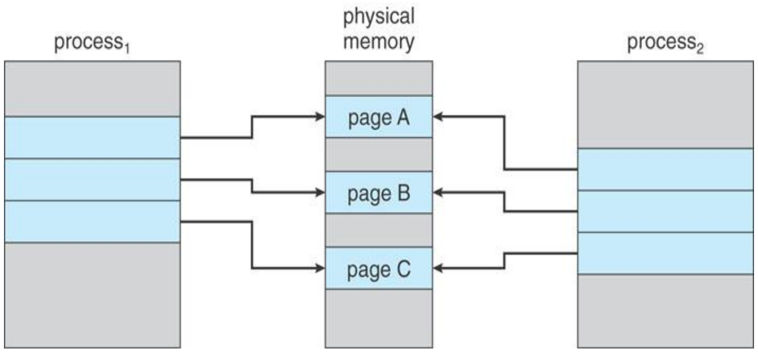
- Copy-on-Write

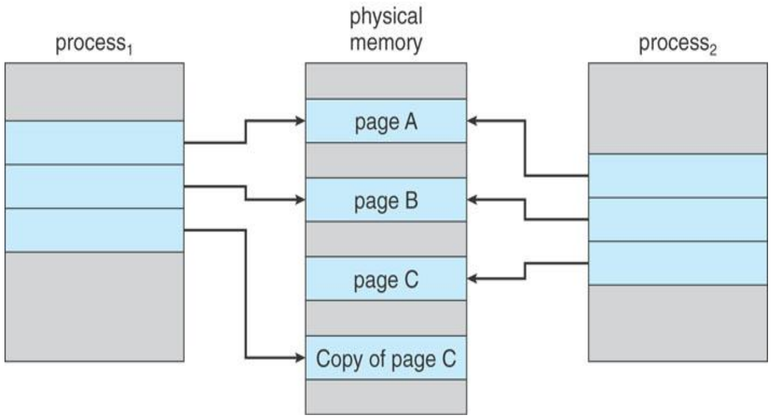
- Memory-Mapped Files (later)

PROCESS CREATION

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
- vfork() virtual memory fork is not like COW
  - Suspend parent, use its address space… be careful
  - Use it when child calls exec





What happens if there is no free frame?

Terminate user program or

Swap out some page

PAGE REPLACEMENT

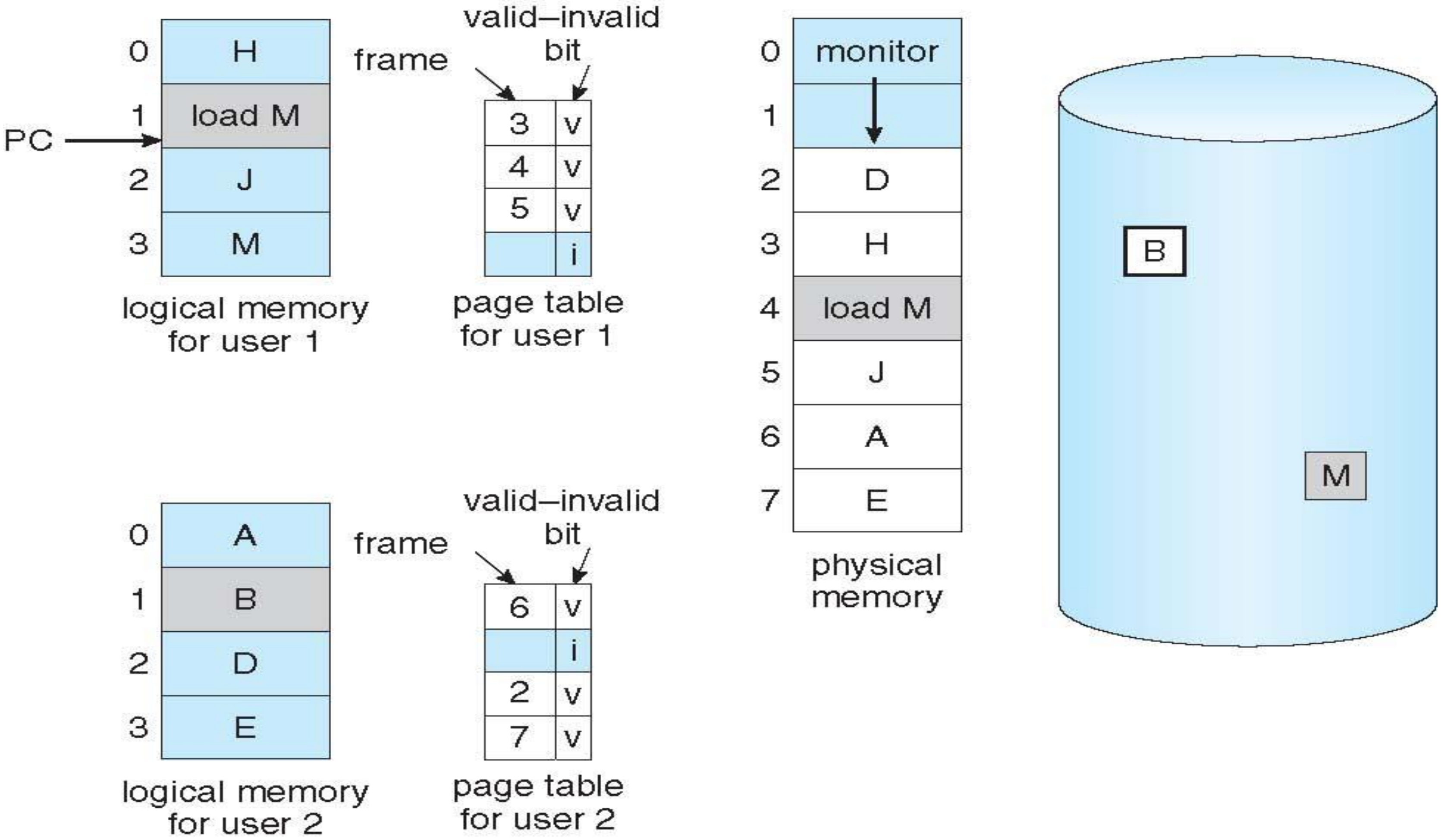
Page Replacement

- To prevent over-allocation of memory, modify page-fault service routine to include page replacement, which finds some page in memory and swaps it out
- Same page may be brought into memory several times
- We need algorithms to minimize the number of page faults

- Include other improvement, e.g., use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

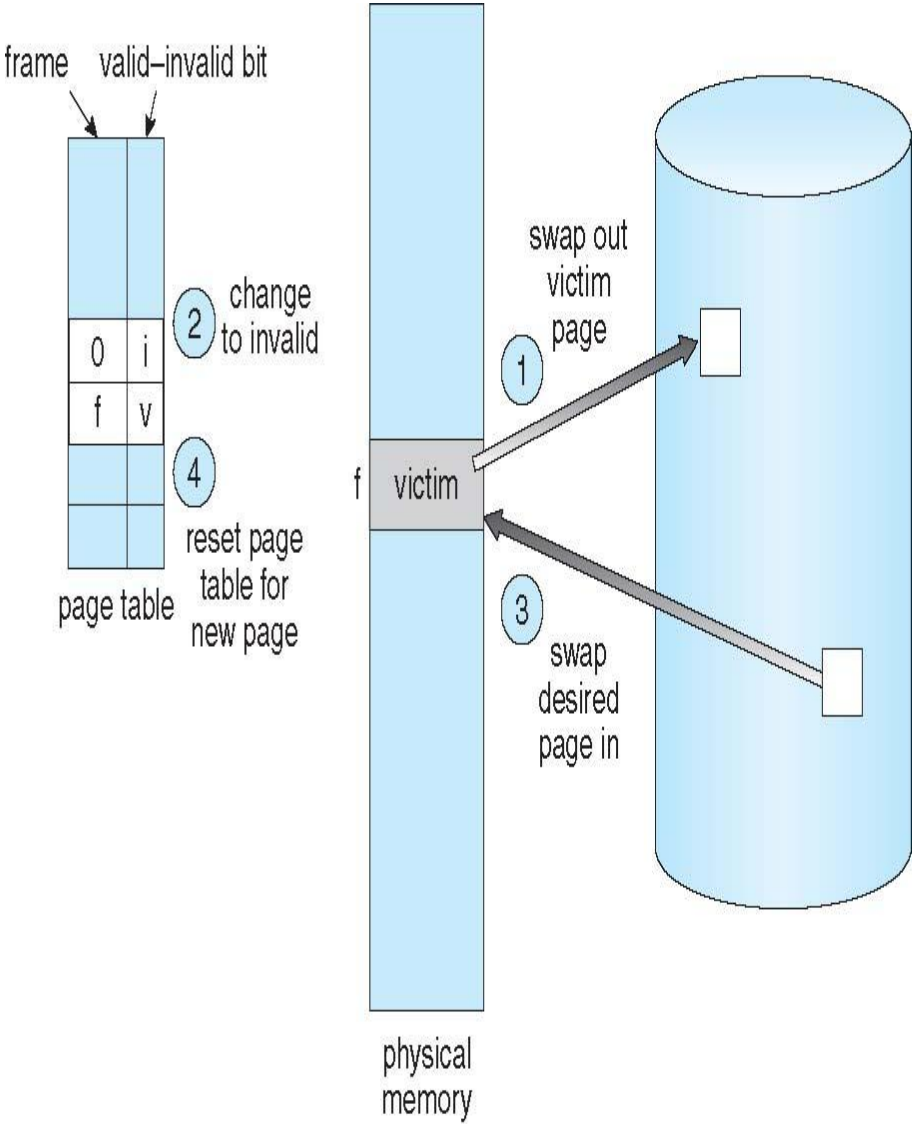
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement



Basic Page Replacement

- Find the location of the desired page on disk
- If there is a free frame, use it
- If there is no free frame, use a page replacement algorithm
    1. Select a **victim** frame, swap it out *(use dirty bit to swap out only modified frames)*
    2. Bring the desired page into the (newly) free frame;
    3. update the page and frame tables
- Restart the process



Page Replacement Algorithms

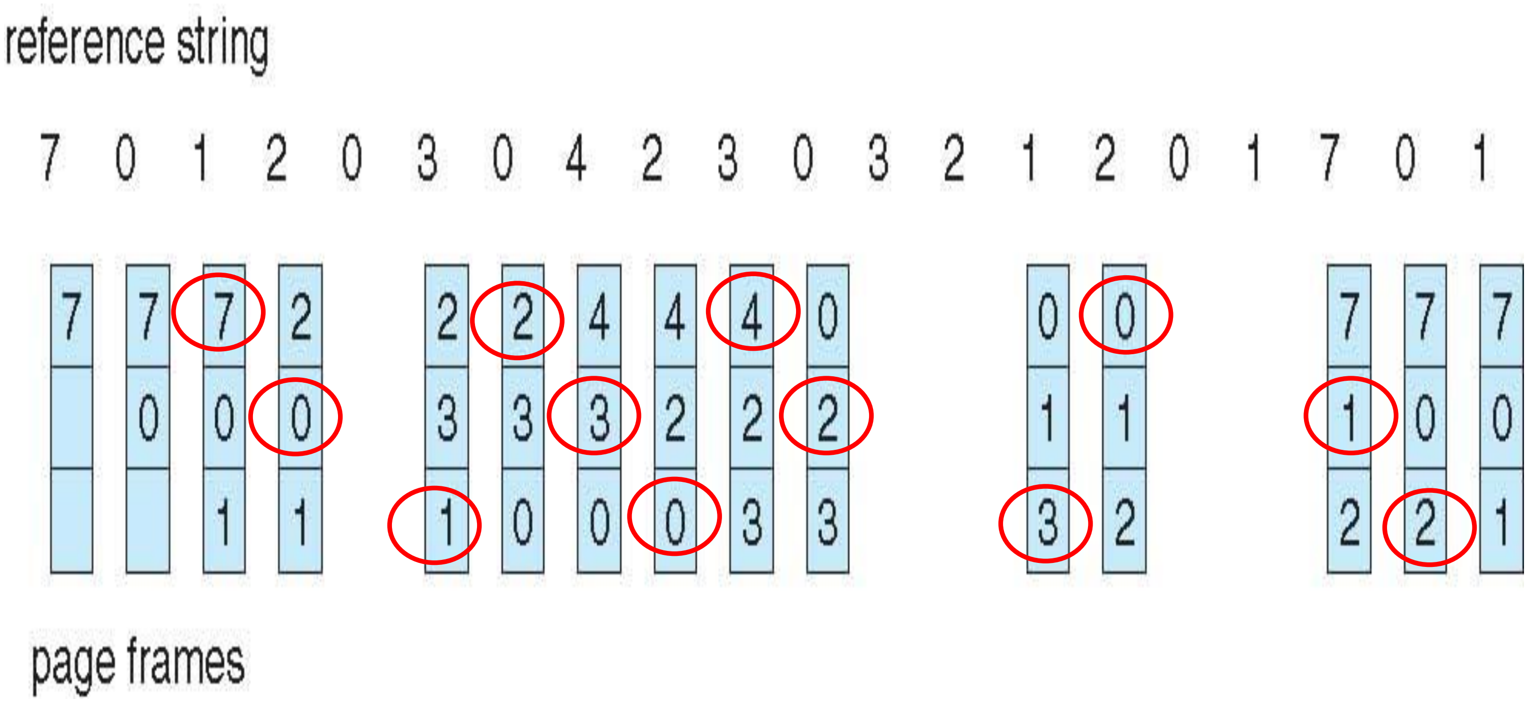
- How to select the victim frame?
  - You can select any frame, the page replacement will work;
  - but the performance???
- So we want an algorithms that gives the lowest page-fault rate
- Evaluate an algorithm by running it on a particular string of memory references (reference string) and compute the number of page faults on that string

  In all our examples, we will have 3 frames and the following reference string

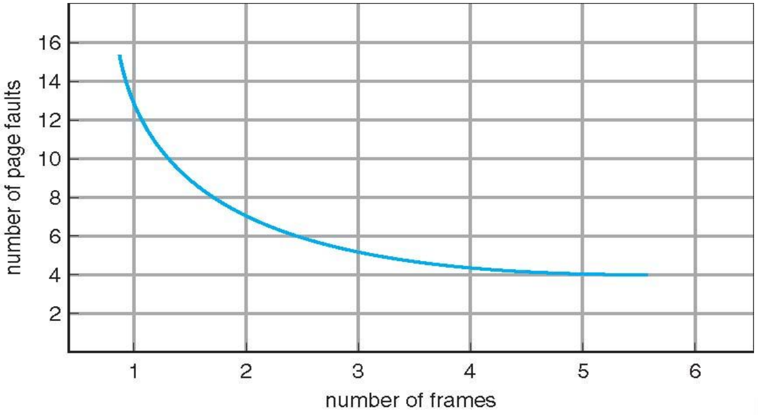
7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

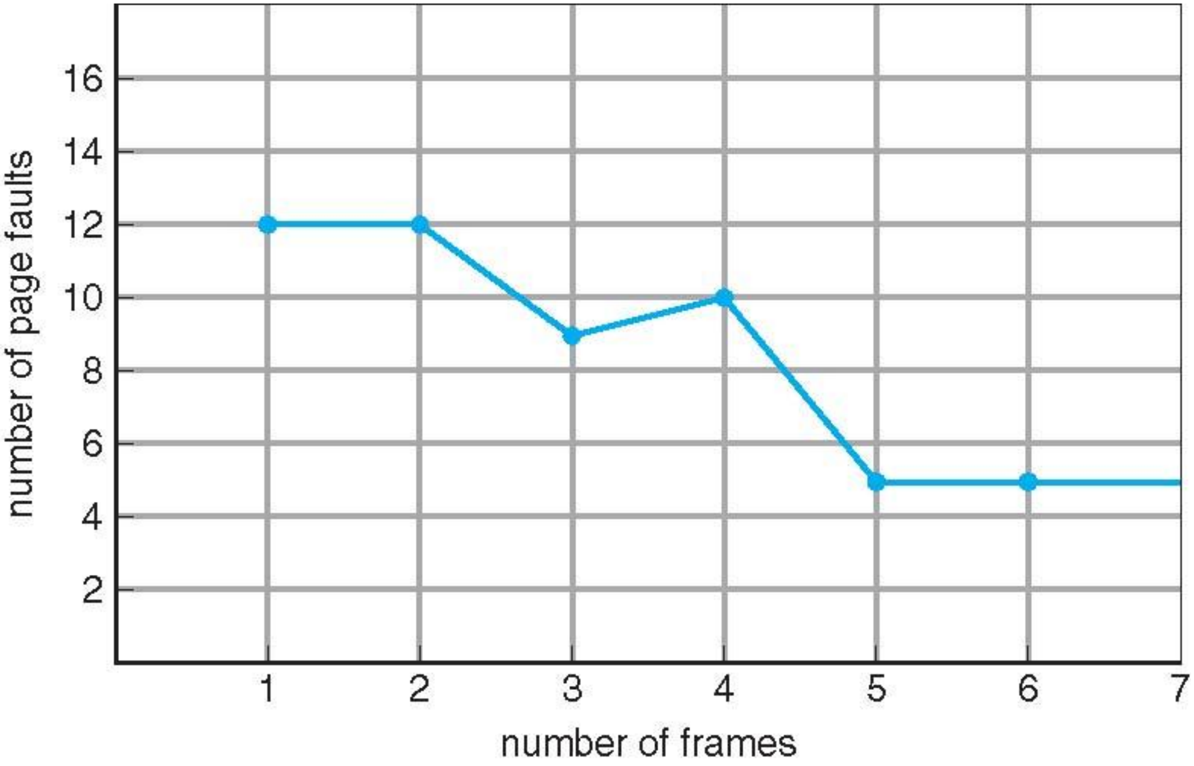
First-In-First-Out (FIFO) Algorithm

- Maintain an FIFO buffer
  - + The code used before may not be needed
  - - An array used early, might be used again and again
- Easy to implement
- Belady’s Anomaly: more frames ⇒ more page faults



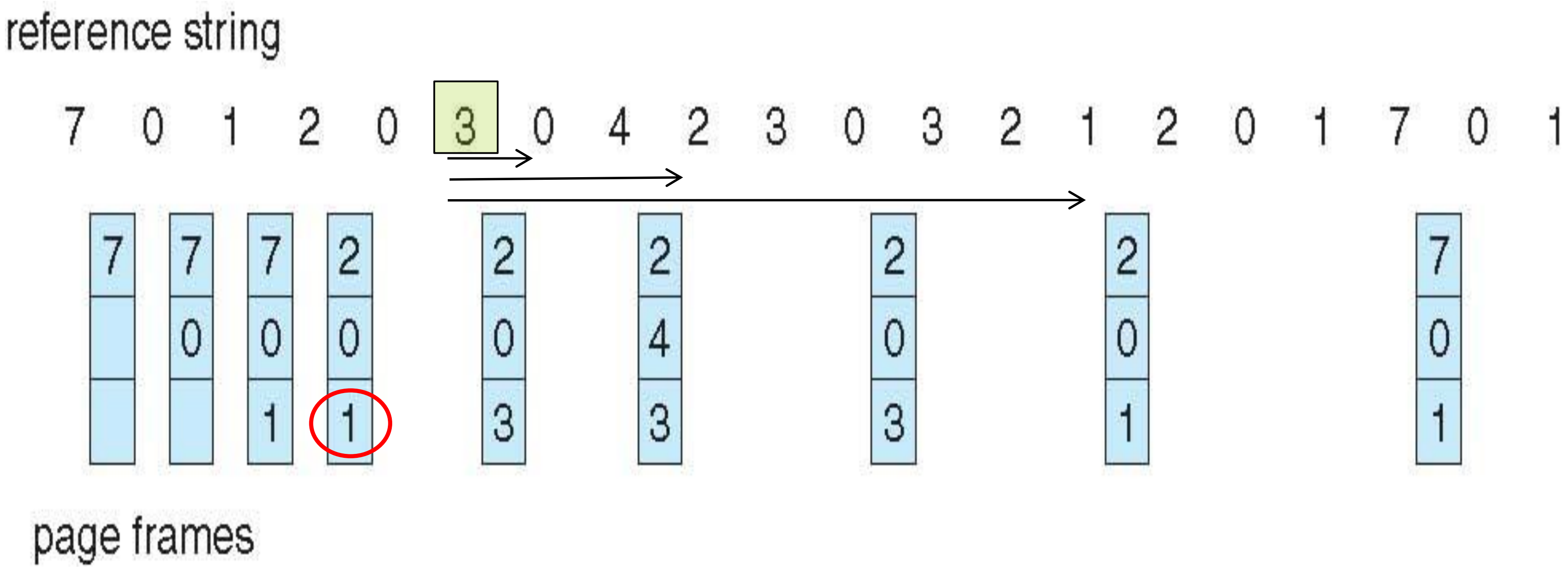
FIFO Illustrating Belady's Anomaly





Optimal Algorithm

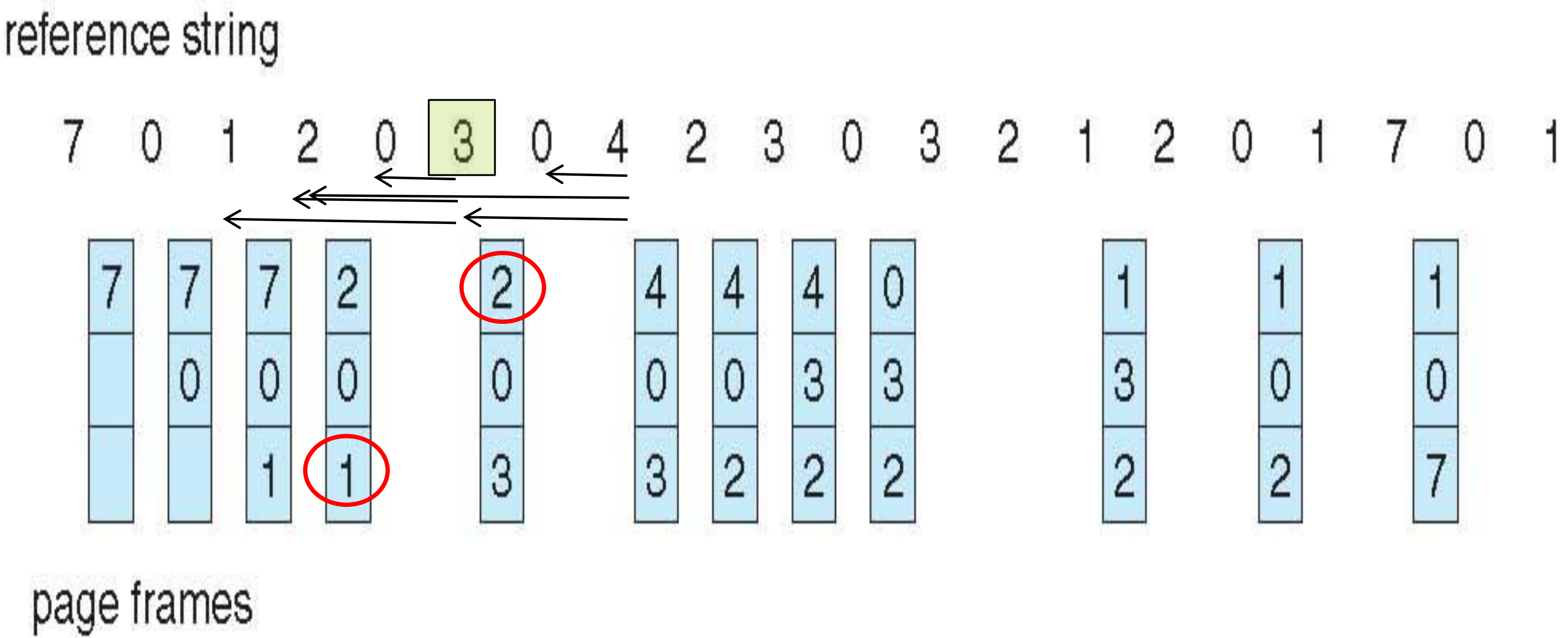
- Replace page that will not be used for longest period of time



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

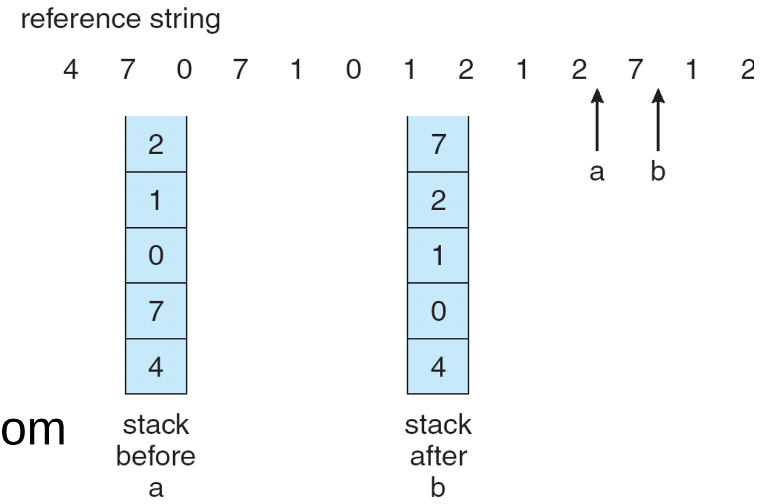
Least Recently Used (LRU) Algorithm

- Use recent past as an approximation of the future
- Select the page that is not used for a long time…
  - OPT if you look at from backward
  - NO Belady’s Anomaly: so more frames ⇒ less page faults
- Hard to implement (why?)



LRU Algorithm (Cont.)

- Counter (logical clock) implementation
  - Increase the counter every time a page is referenced
  - Save it into time-of-use field associated with this page’s entry in the page table
  - When a page needs to be replaced, find the one that has the smallest time-of-use value
  - Problems: Counter overflow and linear search
- 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
    - Least recently used one is at the bottom



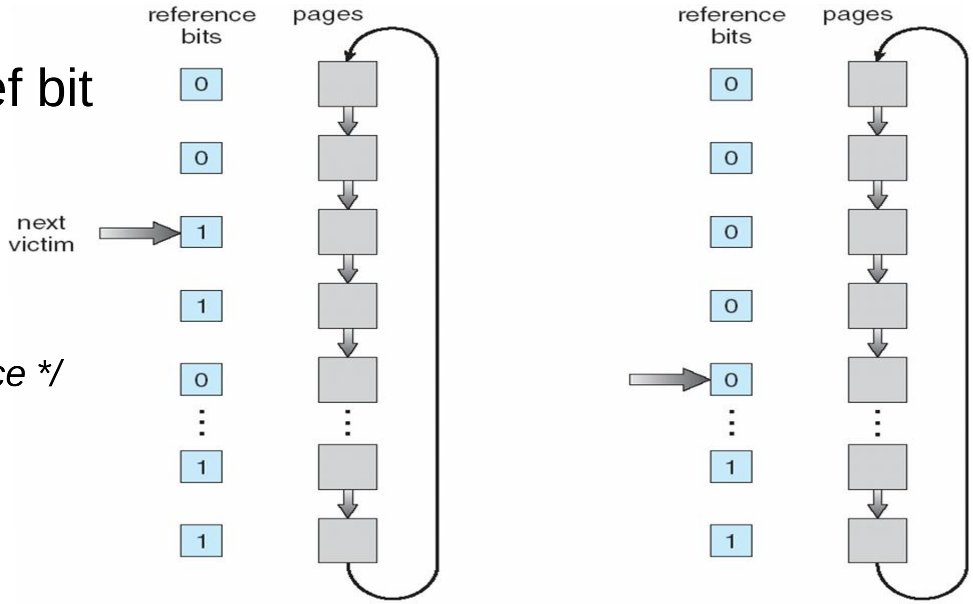
Hardware assistance needed to do updates for every memory reference

LRU Approximation Algorithms

- Reference bit
  - With each page associate a reference bit, initially = 0
  - When page is referenced, set this bit to 1 by hardware
  - Replace the one which is 0 (if one exists)
    - We do not know the order, however
    - Additional bits can help to gain more ordering information
    - In the extreme case, use just reference bit, no additional bit

What if all bits are 1 …. All pages will get second chance…. Degenerates FIFO

- Second chance Alg
  - FIFO with an inspection of ref bit
  - If ref bit is 0,
    - replace that page
    - set its ref bit to 1
  - If ref bit is 1, */* give a second chance */*
    - set ref bit to 0
    - leave page in memory
    - go to next one
  - Enhance it modify bit, avoid replacing modified pages



circular queue of pages (a)

circular queue of pages (b)

Counting Algorithms: LFU and MFU

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count
  - + Active pages are likely to be used again
  - - Code within a big loop may not be used again..
  - Shift counters to form an exponential decaying
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Expensive, don't perform well in general, but might be useful for some applications
  - (database application may read a lot of data first then search, but LRU will remove the old ones
  - LFU/MFU might work depending on the application)

Other improvements

- Page Buffering
  - Have free frame pools
  - First get the page from disk to free frame, then
  - As before select victim and write it out
  - Whenever paging device is idle write them out
  - Mark a frame as free but remember for which page it was used (like recycle bin) so if needed that frame can be used again without going to disk
- Applications and Page Replacements
  - For some applications general purpose solutions may not work well
  - For example database application may make a better use of resources as it understands the nature of data better….

Summary: Page Replacement Algorithms

Algorithm	**Comment**
FIFO (First-In, First Out)	Might throw out useful pages
Second chance	Big improvement over FIFO
LRU (Least Recently Used)	Excellent, but hard to implement exactly
OPT (Optimal)	Not implementable, but useful as a benchmark

How paging may impact the performance of a Program

- 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

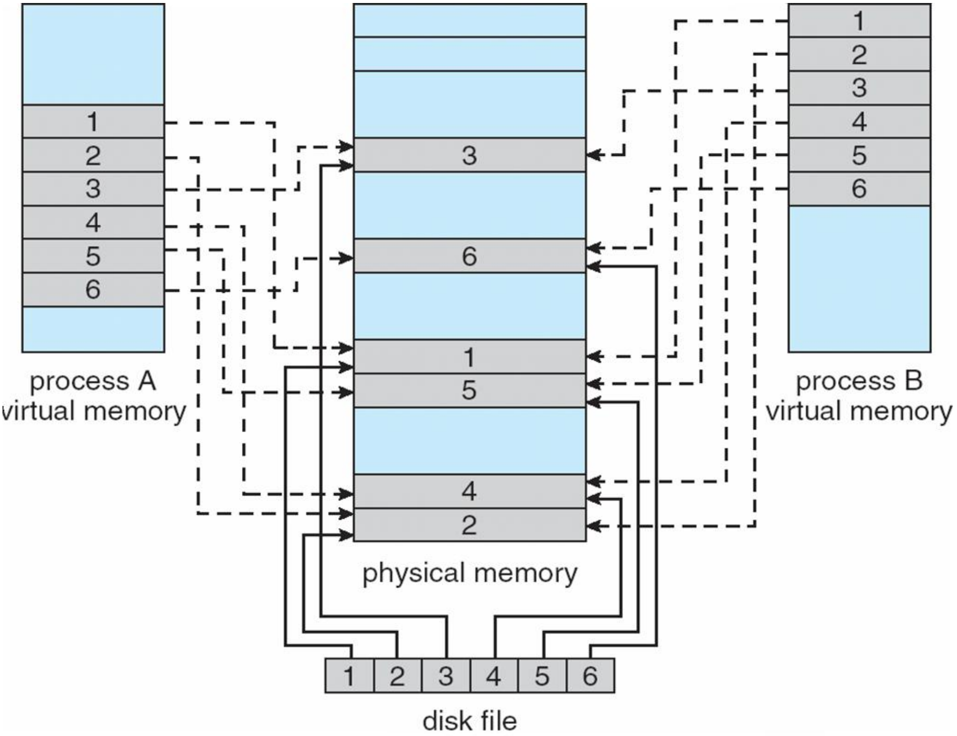
- Increase locality, separate code and data, avoid page boundaries for routines arrays,
  - Stack has good locality but hash has bad locality
  - Pointers, Objects may diminish locality

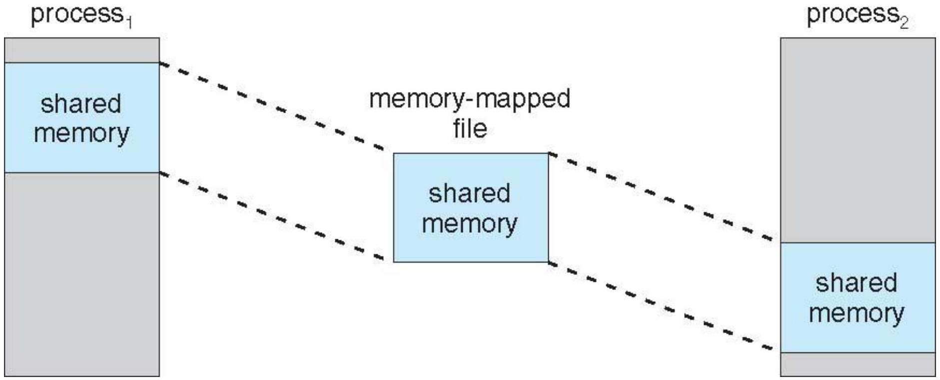
Treat file I/O as routine memory access

MEMORY-MAPPED FILES

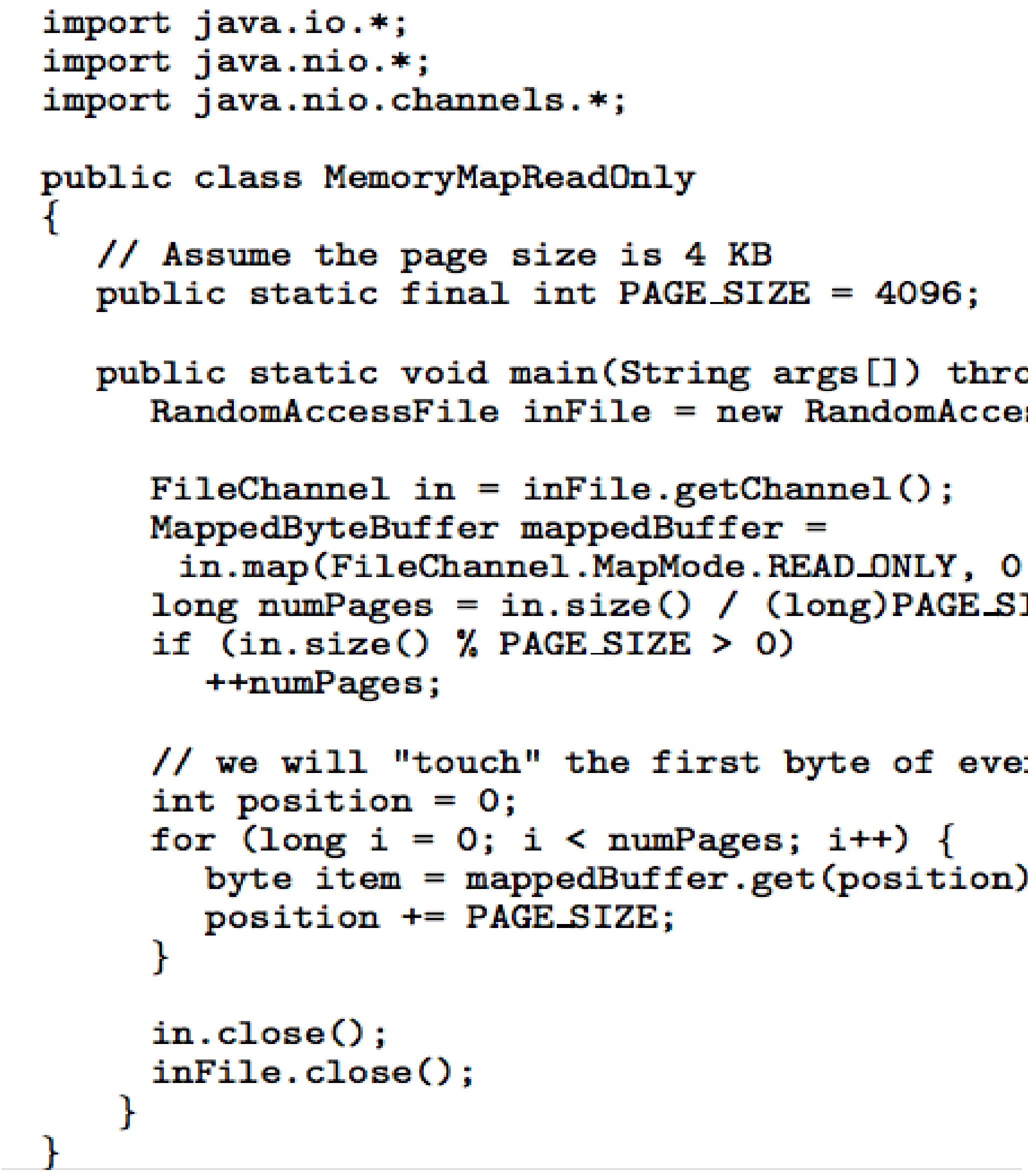
Memory-Mapped Files

- **Map** a disk block to a page in memory, then file I/O can be treated as routine memory access and avoid avoiding system calls like **`read()`** **`write()`**
  - Data written into memory is not immediate written to disk!
- 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.
- Also allows several processes to map the same file allowing the pages in memory to be shared.





Memory-Mapped Files in Java



```
import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly
{
   // Assume the page size is 4 KB
   public static final int PAGE_SIZE = 4096;

   public static void main(String args[]) throws IOException {
      RandomAccessFile inFile = new RandomAccessFile(args[0],"r");

      FileChannel in = inFile.getChannel();
      MappedByteBuffer mappedBuffer =
        in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
      long numPages = in.size() / (long)PAGE_SIZE;
      if (in.size() % PAGE_SIZE > 0)
         ++numPages;

      // we will "touch" the first byte of every page
      int position = 0;
      for (long i = 0; i < numPages; i++) {
         byte item = mappedBuffer.get(position);
         position += PAGE_SIZE;
      }

      in.close();
      inFile.close();
    }
}
```

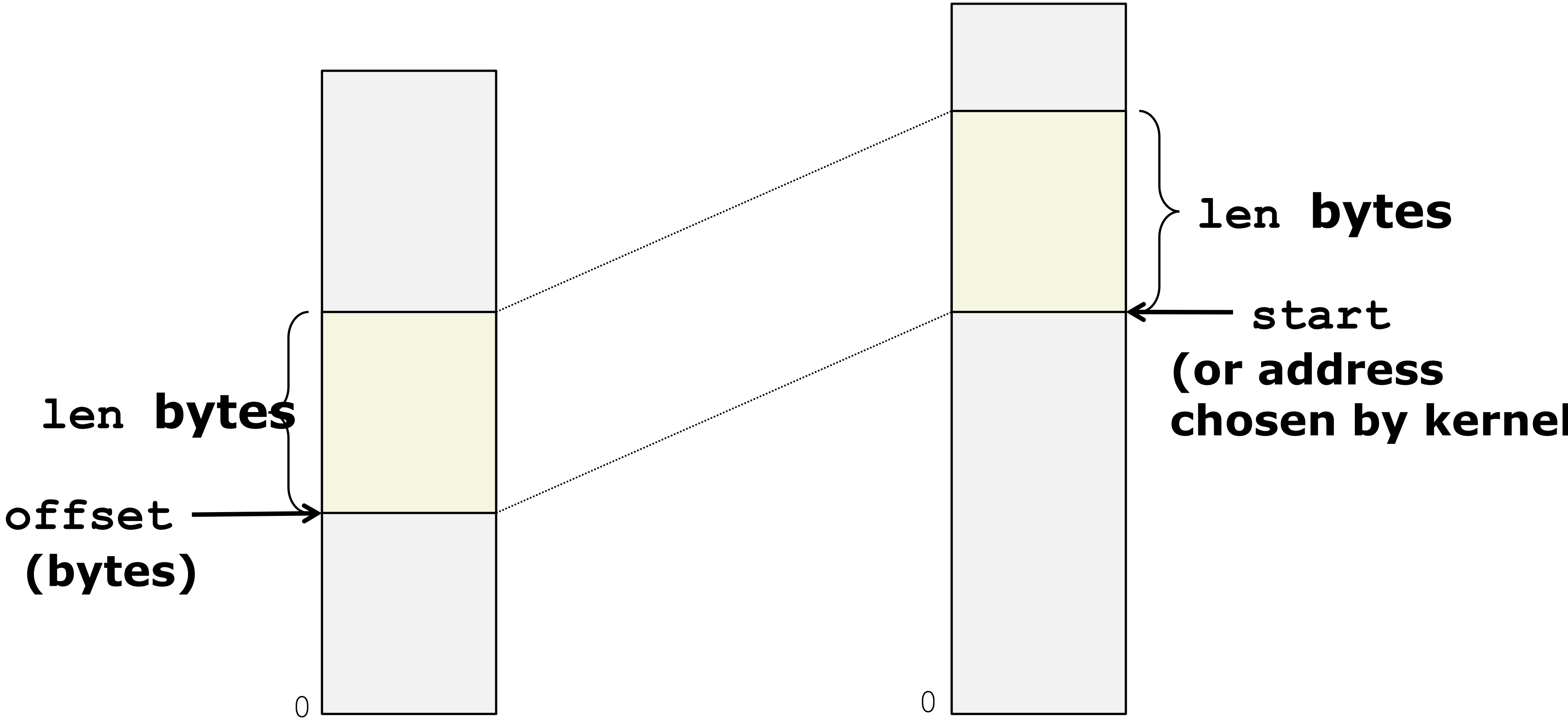
User-Level Memory Mapping in C

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map **`len`** bytes starting at offset **`offset`** of the file specified by file description **`fd`**, preferably at address **`start`**
  - **`start:`** may be 0 for “pick an address”
  - **`prot`**: PROT_READ, PROT_WRITE, ...
  - **`flags`**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...
- Return a pointer to start of mapped area (may not be **`start`**)
  - Anonymous: No backup on files
  - File-backed mapping: Backed up by a file.

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



***Disk file specified by file descriptor* `fd`**

Process virtual memory

Memory-Mapped I/O

- I/O is mapped to memory actually some ranges of addresses are allocated for different devices
- CPU can communicate these devices through memory accesses
- Programmed I/O vs. Interrupt driven I/O
  - One at a time vs. all at once then followed by interrupt

Two major allocation schemes

- fixed allocation
- priority allocation

ALLOCATION OF FRAMES

Minimum Number of Frames

- Each process needs *minimum* number of pages

Examples

- `add a b c` might require 3 pages
- 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*

- Level of indirection…
- Min depends on architecture
- Maximum depends on available memory
- How about the optimal to maximize CPU utilization?

Allocation Algorithms

- Fixed allocation
  - Equal allocation: – Allocate same amount to each process
    - For example, if there are 100 frames and 5 processes, each gets 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
  - High priority processes can take all frames from low priority ones (cause thrashing)
  - A process cannot control its page fault rate
- **Local replacement** – each process selects from only its own set of allocated frames
  - How determine the size of the set ???

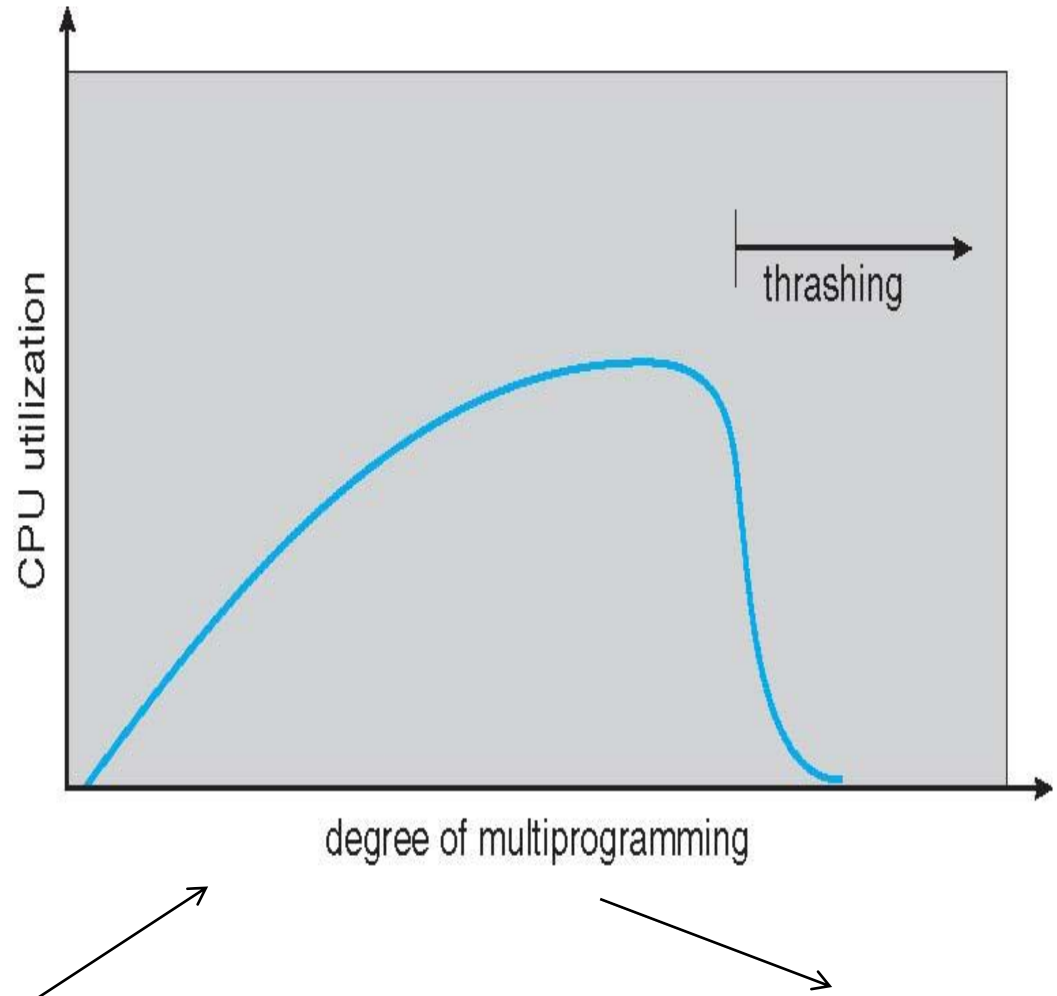
Cover the rest as much as the time permits…

A process is busy swapping pages in and out

THRASHING

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
  - But then trashing happens



increase the degree of multiprogramming

Decrease the degree of multiprogramming

Locality and Thrashing

- To prevent thrashing we should give **enough** frames to each process
- But how much is "**enough**"

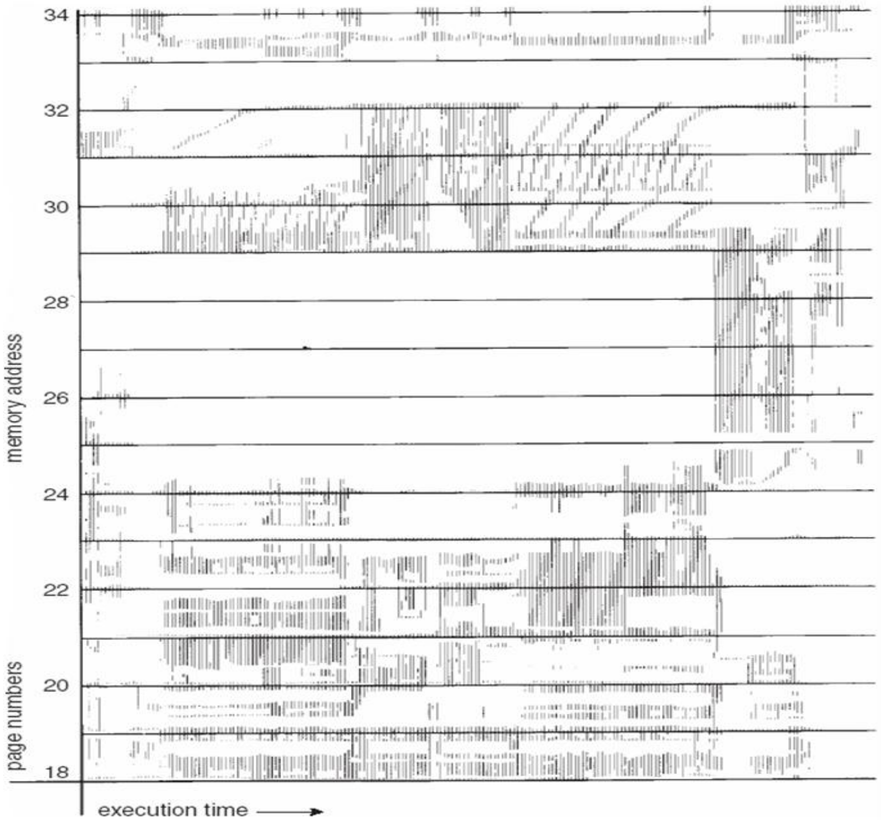
Locality model

- Process migrates from one locality to another *(that is actually why demand paging or cashing works)*
- Localities may overlap

When   Σ size of locality >

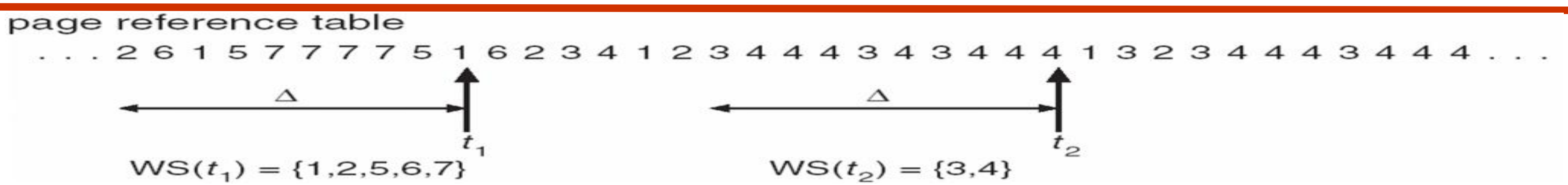
total memory size,

thrashing occurs…

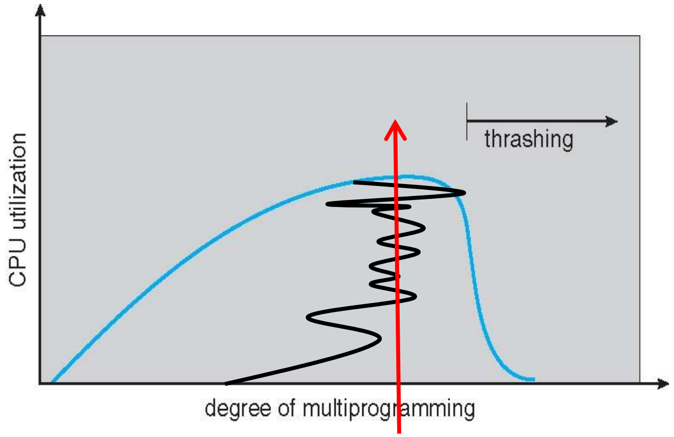


Increase locality in
your programs!

Working-Set Model



- $\Delta \equiv$ 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



- $D = \Sigma\, WSS_i \equiv$ total demand frames
- if $D >$ (available frames) $m \Rightarrow$ Thrashing
- Policy if $D > m$, then

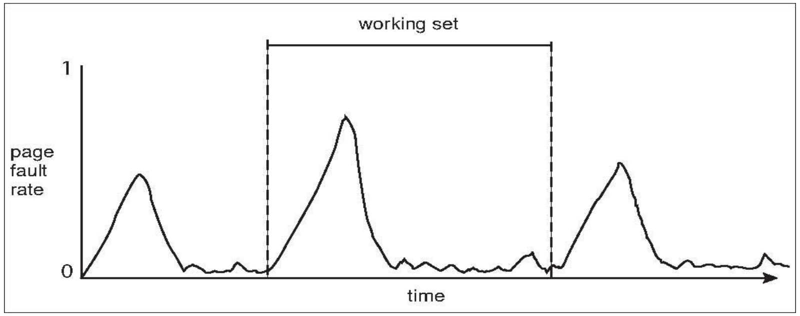
  suspend one of the processes (reduce degree of multiprogramming)

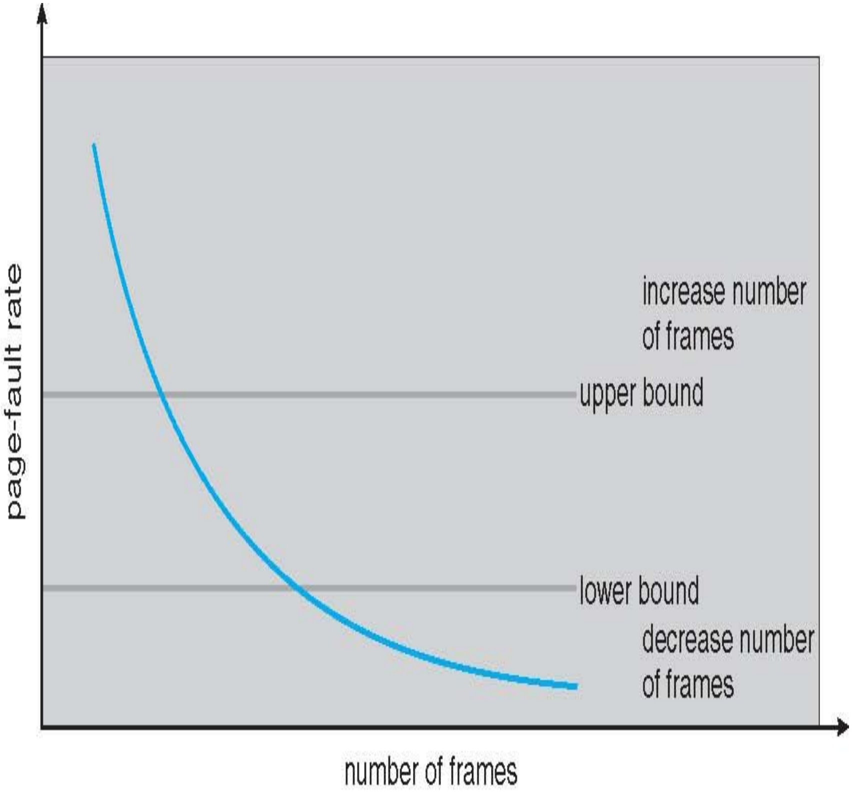
Keeping Track of the Working Set

- 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 set 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 (PFF) Scheme

- Working set is a clumsy way to control thrashing
- PFF takes more direct approach
  - High PFF → more thrashing
  - Establish “acceptable” page-fault rate
    - If actual rate is too low, process loses frame
    - If actual rate is too high, process gains frame
  - Suspend a process if PFF is above upper bound and there is no free frames!





Typically, the user will get one big block of memory and setup its page table.

Allocate 1 page even when 1 byte is needed…

Then this memory will be managed by user space memory manager.

How to manage the memory inside user space?

USER MEMORY ALLOCATION

Memory allocation (using mmap/brk)

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

```
08048000-08049000 r-xp   test
08049000-0804a000 r—p  test
0804a000-0804b000 rw-p  test
b7e7b000-b7e7c000 rw-p  0
b7e7c000-b7fd8000 r-xp    libc-2.9.so
b7fd8000-b7fd9000 ---p     libc-2.9.so
b7fd9000-b7fdb000 r--p     libc-2.9.so
b7fdb000-b7fdc000 rw-p    libc-2.9.so
b7fdc000-b7fe1000 rw-p    0
b7fe1000-b7fe2000 r-xp     0       [vdso]
b7fe2000-b7ffe000 r-xp      ld-2.9.so
b7ffe000-b7fff000 r—p       ld-2.9.so
b7fff000-b8000000 rw-p     ld-2.9.so
bffeb000-c0000000 rw-p    [stack]
```

Currently, no heap space at all because we didn’t use any heap

Memory allocation

```c
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```

```
08048000-08049000 r-xp   test
08049000-0804a000 r—p  test
0804a000-0804b000 rw-p  test
0804b000-0806c000 rw-p  [heap]
b7e7b000-b7e7c000 rw-p  0
b7e7c000-b7fd8000 r-xp    libc-2.9.so
b7fd8000-b7fd9000 ---p     libc-2.9.so
b7fd9000-b7fdb000 r--p     libc-2.9.so
b7fdb000-b7fdc000 rw-p    libc-2.9.so
b7fdc000-b7fe1000 rw-p    0
b7fe1000-b7fe2000 r-xp     0       [vdso]
b7fe2000-b7ffe000 r-xp      ld-2.9.so
b7ffe000-b7fff000 r—p       ld-2.9.so
b7fff000-b8000000 rw-p     ld-2.9.so
bffeb000-c0000000 rw-p    [stack]
```

Now, the heap is allocated from the kernel, which means the virtual address from 0x0804b000 to 0x0806c000 (total 33K) are usable. ptr is actually 0x804b008.

Memory Mapping (mmap or brk)

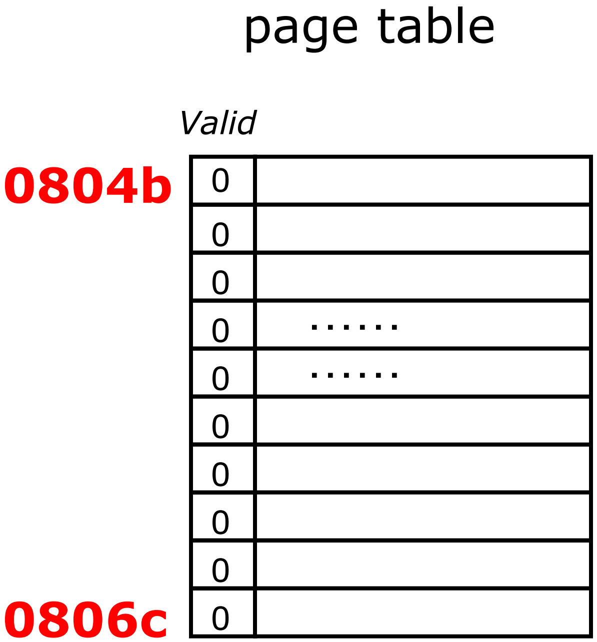
```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```



page table

	Valid	
0804b	0	
	0	
	0	
	0	……
	0	……
	0	
	0	
	0	
	0	
0806c	0	

0804b000-0806c000 rw-p [heap]

Memory Mapping (mmap or brk)

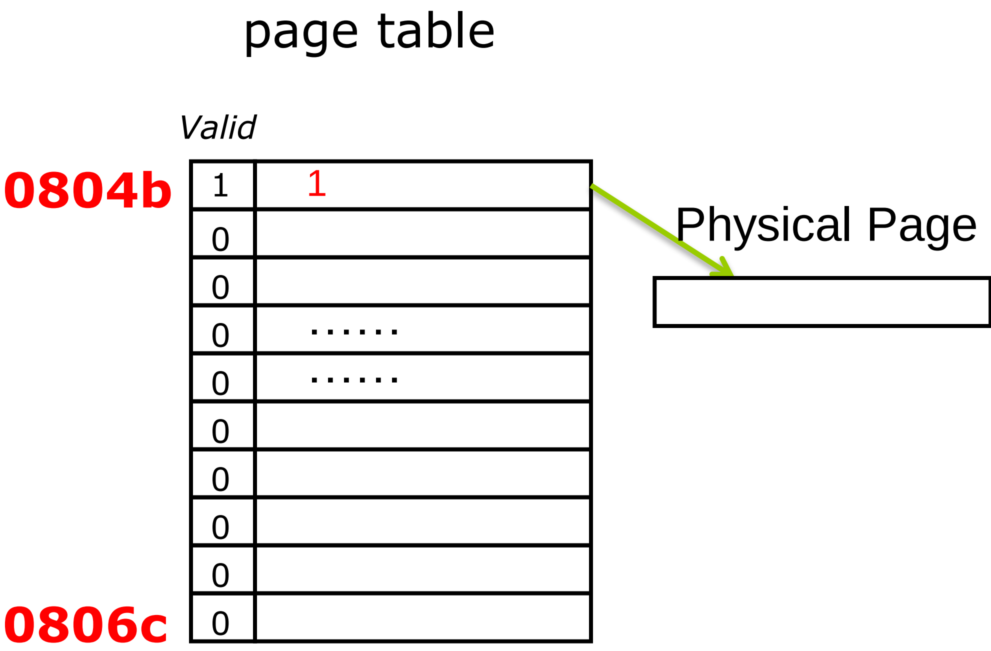
```
#include <stdio.h>
#include <stdlib.h>

int main() {

  int * ptr = malloc(4);

  *ptr = 1;

  free(ptr);
}
```



0804b000-0806c000 rw-p [heap]

Treated differently from user memory (allocate 1 page even when 1 byte is needed)

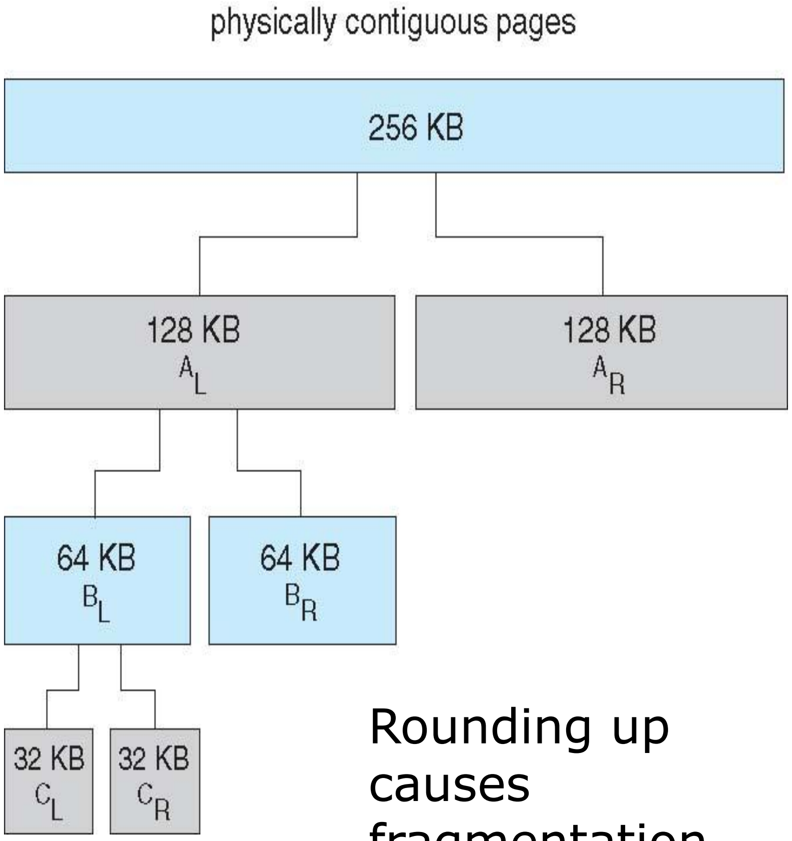
Often allocated from a different free-memory pool

- Kernel requests memory for structures of varying sizes
- Some kernel memory needs to be contiguous

ALLOCATING KERNEL MEMORY

Buddy System

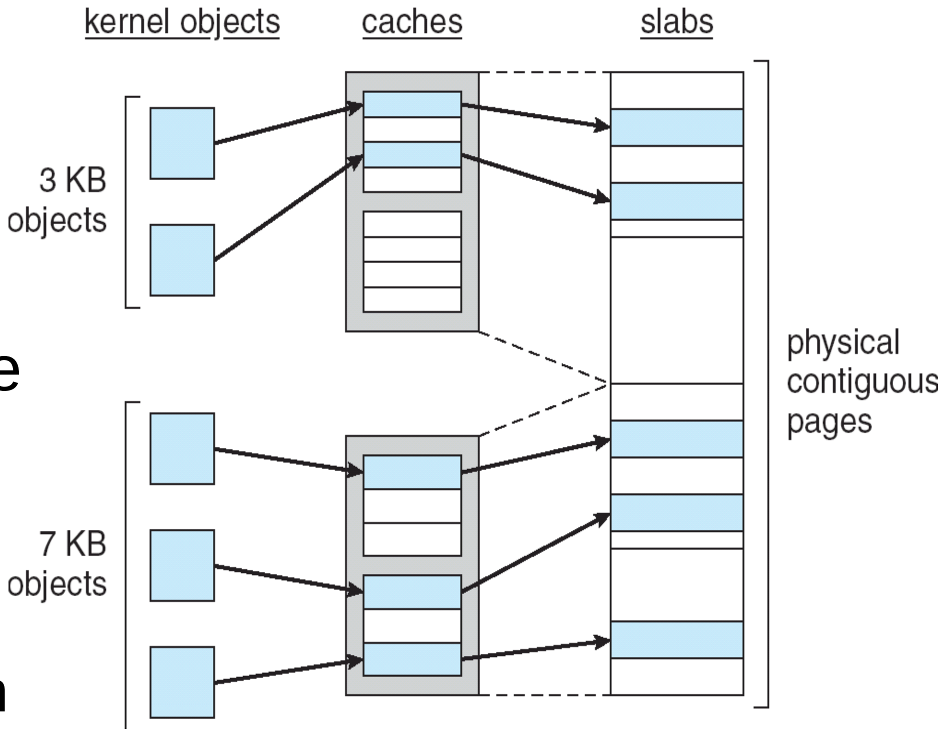
- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk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



Rounding up causes fragmentation, e.g., 33K needs 64K …
50% might be wasted

Slab Allocator

- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure *(process descriptions, file objects, semaphores)*
  - 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**, next object is allocated from empty slab
  - If no empty slabs, new slab allocated



Benefits include

- no fragmentation,
- memory request is satisfied quickly

Main concerns were Replacement and Allocation

But we have several other issues too

OTHER ISSUES

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 (small size page is better)
  - Table size (large size page is better)
  - I/O overhead
    - Seek
    - Latency
    - Transfer
  - Locality
- New Oses tends to use larger an larger sizes….

Other Issues – TLB Reach

Increasing hit rate is good but associative memory is expensive and power hungry

- 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
  - Increases TLB reach but 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

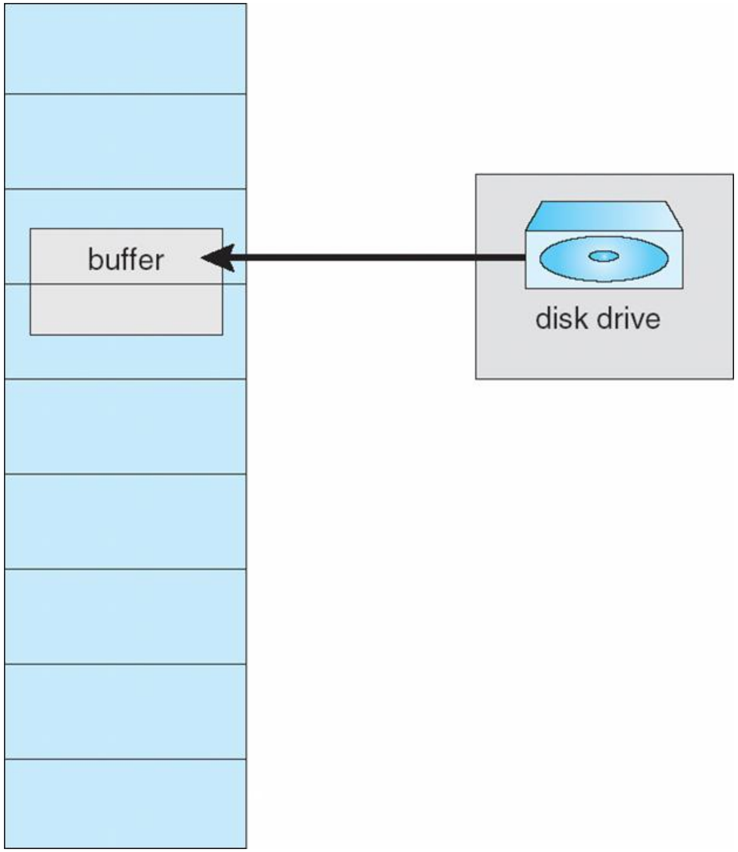
```
for (i = 0; i < 128; i++)
   for (j = 0; j < 128; j++)
            data[i,j] = 0;
```

128 page faults

- Increase locality, separate code and data, avoid page boundaries for routines arrays,
  - Stack has good locality but hash has bad locality
  - Pointers, Objects may diminish locality

Other Issues – I/O interlock

- Users I/O might be done through kernel *(mem-to-mem copy overhead)*
- **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
- Lock bit might be dangerous
  - What if it locked due to a bug in OS
  - Some uses it as a hint but ignore it
  - Some periodically clears it



Windows XP

Solaris

OPERATING SYSTEM EXAMPLES

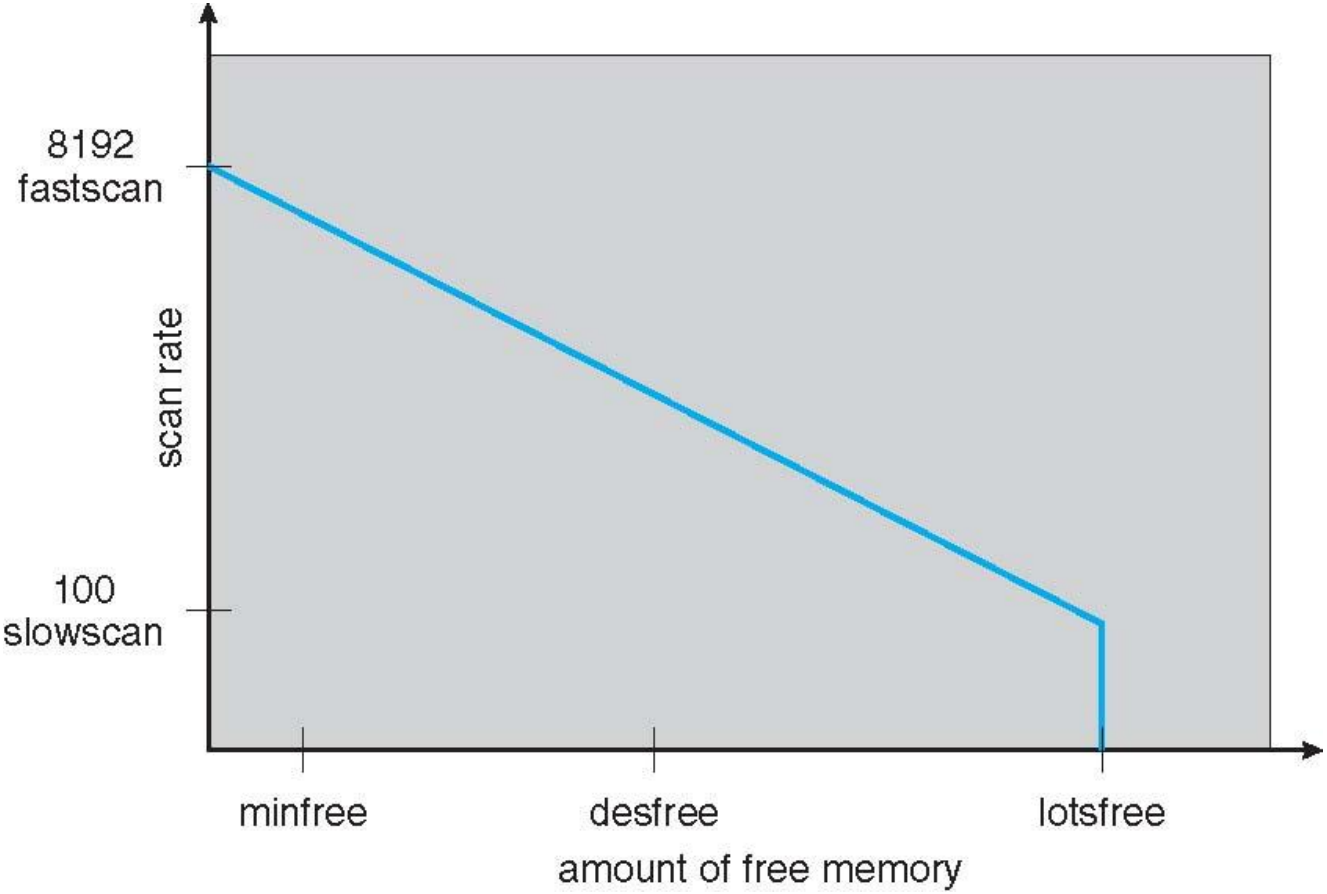
Windows XP

- 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.
- 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.

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 2 Page Scanner



End of Chapter 9