Chapter 9: Virtual Memory

Chapter 9: Virtual Memory

- 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

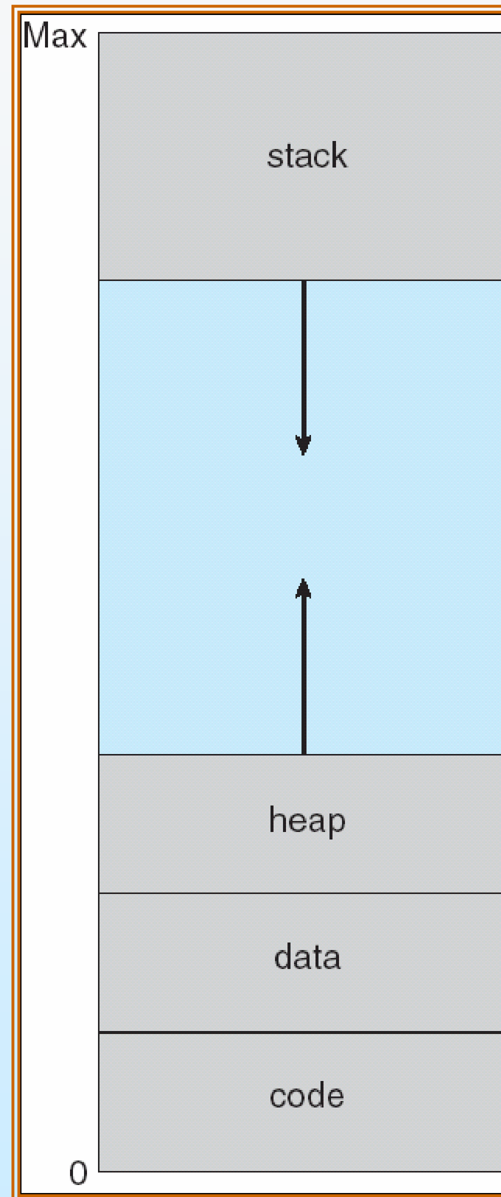
Background

- **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

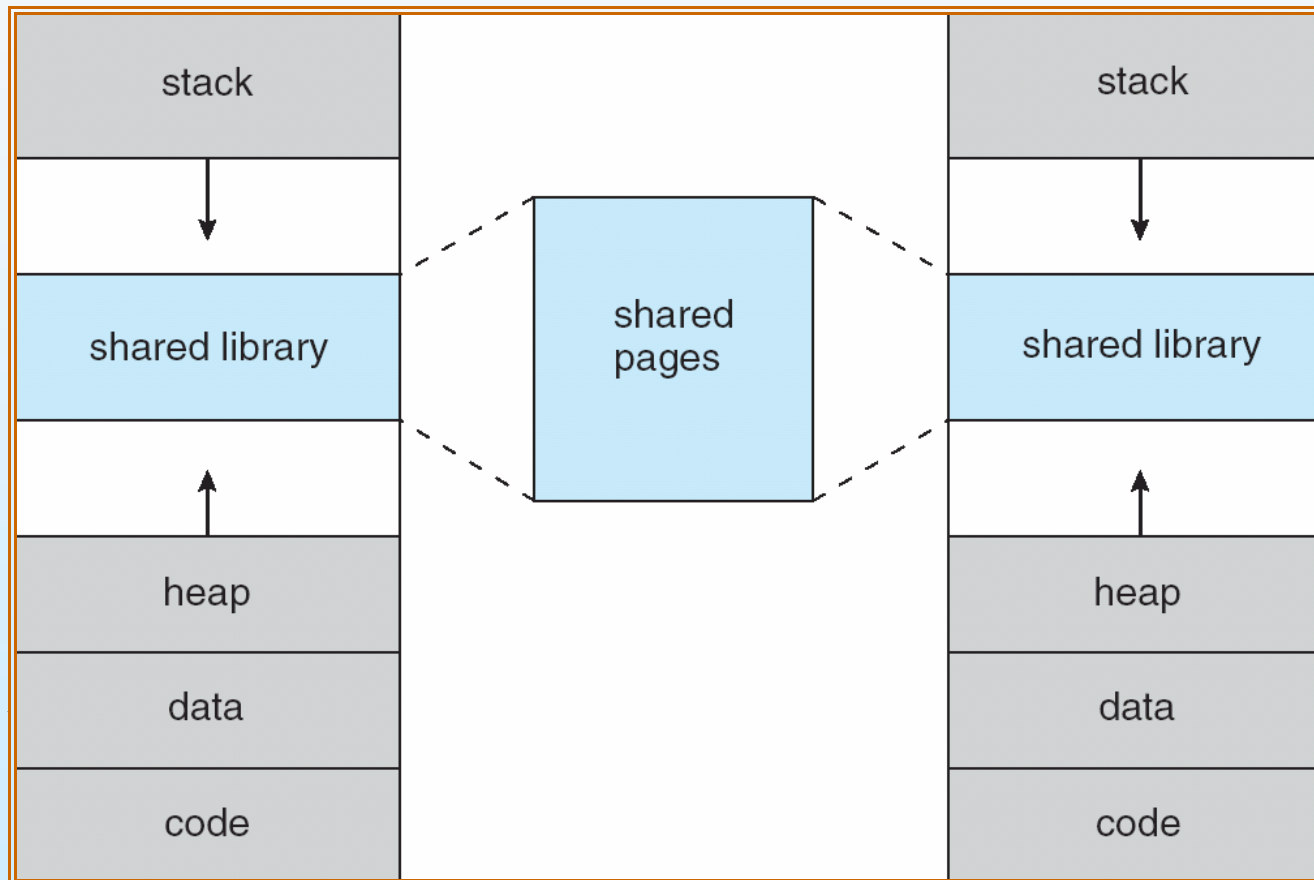
Virtual Memory That is Larger Than Physical Memory

page 0
page 1
page 2
page v
virtual memory
memory map
physical memory

Virtual-address Space



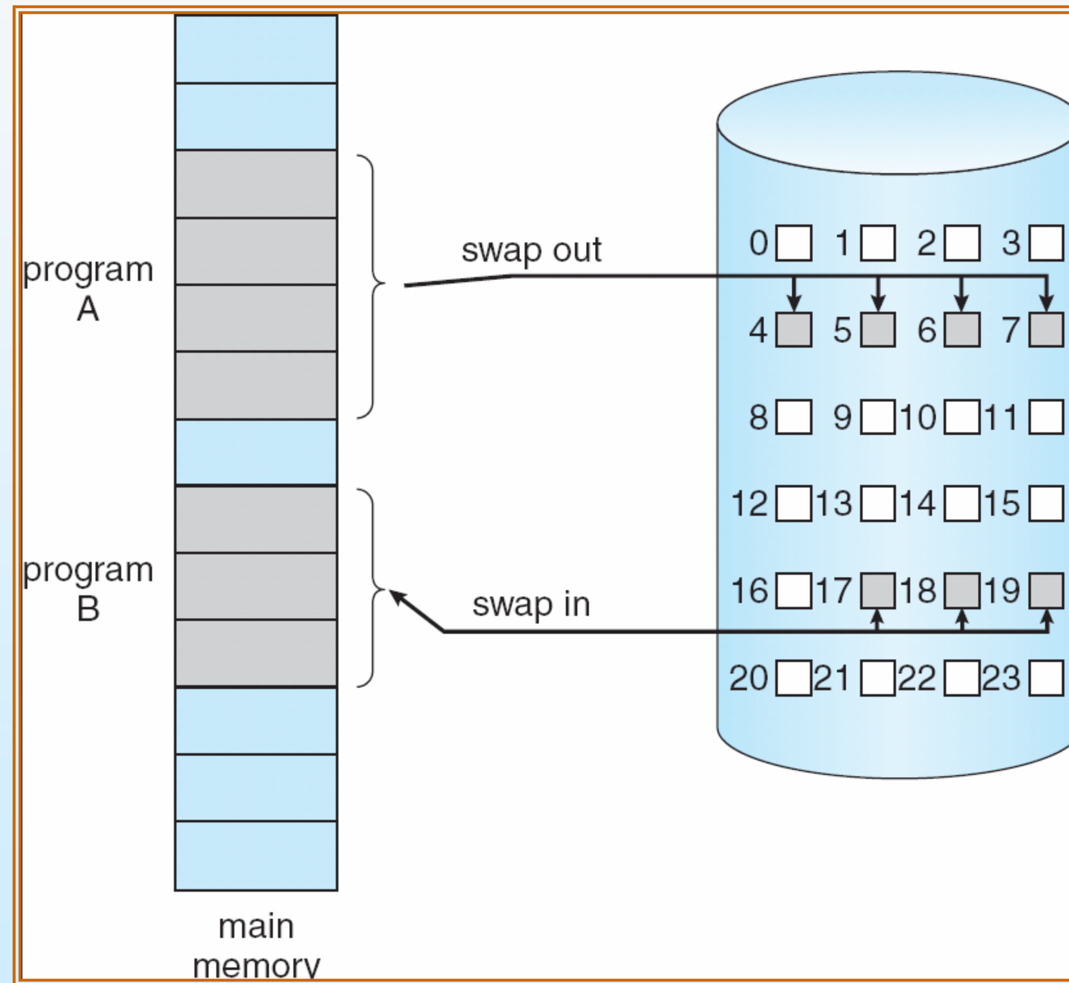
Shared Library Using Virtual Memory



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 ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ 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

logical memory

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

page table (frame, valid–invalid bit)

	frame	valid–invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

physical memory

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

Disk: A B C D E F G H

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 ⇒ abort
   - Just not in memory
2. Find a free frame
3. Schedule disk operation and swap page into frame
4. Modify page table
5. Set validation bit = **v**
6. **Restart the instruction** that caused the page fault

Steps in Handling a Page Fault

3 page is on backing store

operating system

2 trap

reference

1

load M

i

6

restart instruction

page table

free frame

5

reset page table

4

bring in missing page

physical memory

Pure demand paging

- In the extreme case, *we* can start executing a process with no pages in memory.
- When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory-resident page, the immediately faults for the page.
- After this page is brought into memory, the process continues to execute, faulting as necessary until every page that it needs is in memory.
- At that point, it can execute with no more faults. This scheme is **pure demand paging: Never bring a page into memory until it is required.**

 Silberschatz, Galvin and Gagne ©2005

Locality of reference

- Theoretically, some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), **possibly causing multiple page faults per instruction.**

- This situation would result in **unacceptable system performance.**

- Fortunately, analysis of running processes shows that this behavior is exceedingly unlikely.

- Programs tend to have **locality of reference**.

Performance of demand paging

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame

   Wait in a queue for this device until the read request is serviced.

   Wait for the device seek and/or latency time.

   Begin the transfer of the page to a free frame,
6. While waiting, allocate the CPU to some other user (CI)U scheduling, optional).
7. Receive an interrupt from the disk [/0 subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state and new page table and then resume the interrupted instruction.

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)

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

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!!
- If we want performance degradation to be less than 10 percent, we need

  $220 > 200 + 7{,}999{,}800 \times P,$

  $20 > 7{,}999{,}800 \times P,$

  $\boldsymbol{P < 0.0000025.}$

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
  - These free pages are typically allocated when the stack or heap of a process must expand or when there are copy-on-write to be managed.
  - Operating systems typically allocate these pages using a technique known **as zero-fill-on-demand**.
  - **Zero-fill-on-demand pages** have been zeroed-out before being allocated, thus erasing the previous contents.

 Silberschatz, Galvin and Gagne ©2005

Before Process 1 Modifies Page C

physical memory

process$_1$	physical memory	process$_2$
	page A	
	page B	
	page C	

After Process 1 Modifies Page C

process$_1$

physical memory

process$_2$

page A

page B

page C

Copy of page C

Page Replacement

- For the page-fault rate, we assumed that each page faults at most once, when it is first referenced.
  - This representation is not strictly accurate, however.
- If a process of ten pages actually uses only half of them, then demand paging saves the I/0 necessary to load the five pages that are never used.
- We could also increase our degree of multiprogramming by running twice as many processes.
- Thus, if we had **forty frames**, we could run eight processes, rather than the four that could run if each required ten frames (five of which were never used).
  - If we increase our degree of multiprogramming, we are over-allocating memory.
  - If we run six processes, each of which is ten pages in size but actually uses only five pages, we have **higher CPU utilization and throughput**, with ten frames to spare.
- It is possible, however, that each of these processes, for a particular data set, may **suddenly try to use all ten of its pages**, resulting in a need for sixty frames when only forty are available.

Silberschatz, Galvin and Gagne ©2005

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

Silberschatz, Galvin and Gagne ©2005

Page Replacement

- **Prevent over-allocation** of memory by modifying page-fault service routine to include page replacement
- Notice that, if no frames are free, ***two* page transfers** (one out and one in) are required. This situation effectively **doubles the page-fault service time** and increases the effective access time accordingly.
  - 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

Logical memory for user 1:

0	H
1	load M
2	J
3	M

PC → 1

logical memory for user 1

Page table for user 1:

frame	valid–invalid bit
3	v
4	v
5	v
	i

page table for user 1

Logical memory for user 2:

0	A
1	B
2	D
3	E

logical memory for user 2

Page table for user 2:

frame	valid–invalid bit
6	v
	i
2	v
7	v

page table for user 2

Physical memory:

0	monitor
1	
2	D
3	H
4	load M
5	J
6	A
7	E

physical memory

B

M

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

frame
valid–invalid bit

0	i
f	v

page table

2 change to invalid

4 reset page table for new page

f victim

physical memory

1 swap out victim page

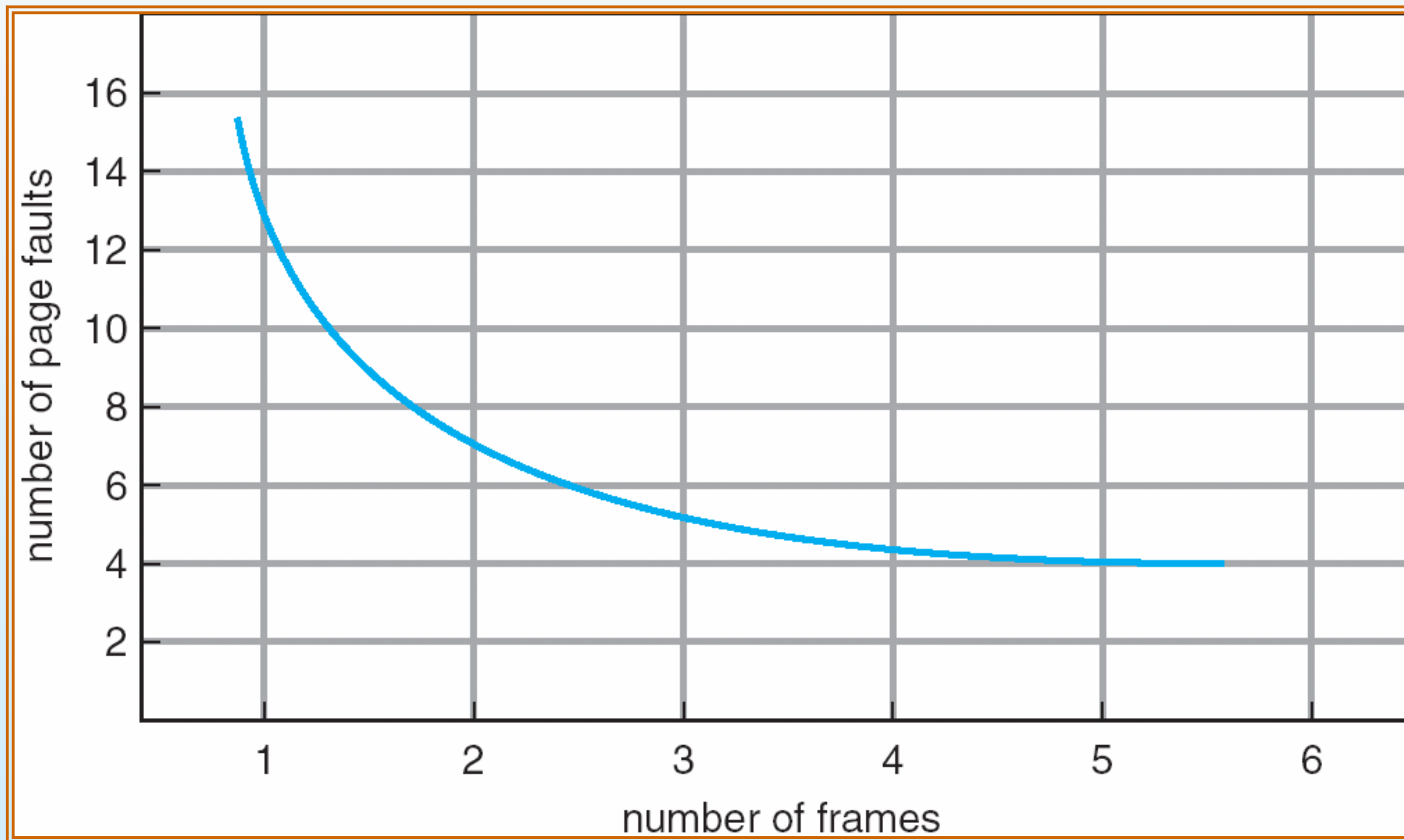
3 swap desired page in

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



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

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

FIFO Page Replacement

reference string

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

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

page frames

15 PAGE FAULTS

FIFO Illustrating Belady's Anomaly

number of page faults

16
14
12
10
8
6
4
2

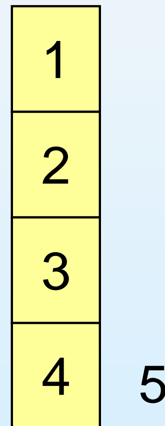
1 2 3 4 5 6 7

number of frames

Optimal Algorithm (does not exist ☺)

- 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



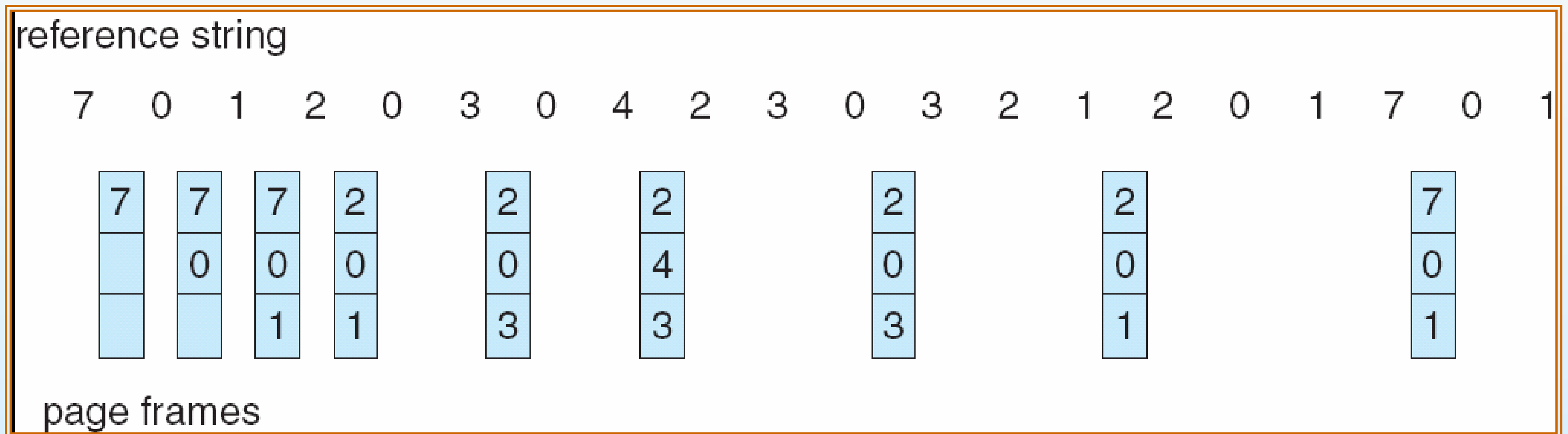
4

5

6 page faults

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

Optimal Page Replacement



9 PAGE FAULTS (compared to 15 of FIFO)

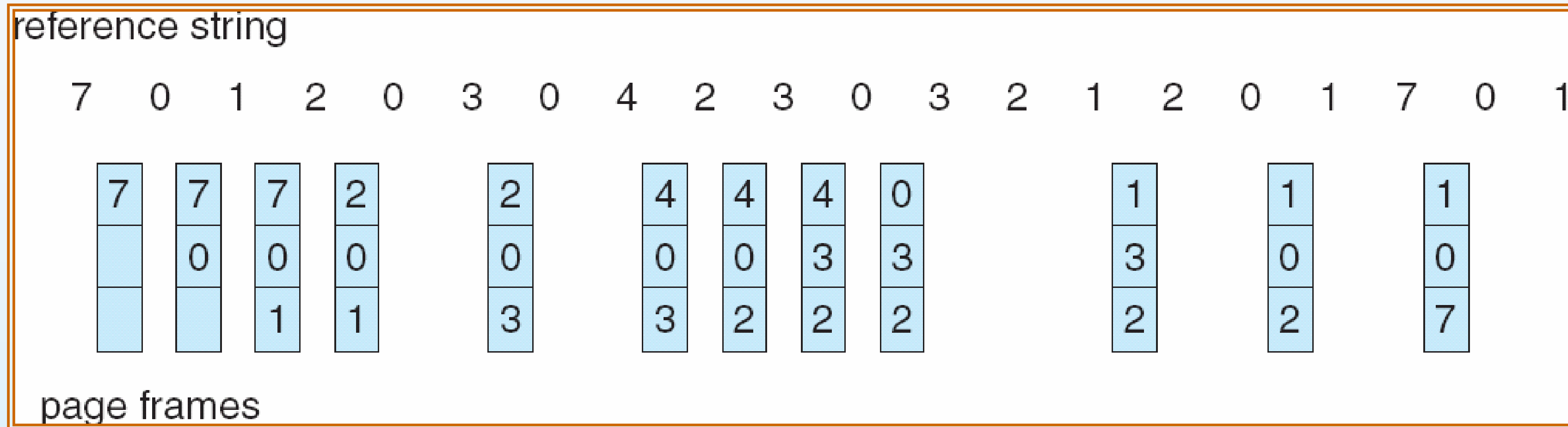
Least Recently Used (LRU) Algorithm

- 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



12 PAGE FAULTS (compared to 9 of OPT and 15 of FIFO)

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

Silberschatz, Galvin and Gagne ©2005

Use Of A Stack to Record The Most Recent Page References

reference string

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

2
1
0
7
4

stack before a

7
2
1
0
4

stack after b

a b

LRU Approximation Algorithms

- 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

Additional-Reference-Bits algorithm

- We can keep an 8-bit byte for each page in a table in memory.
- At regular intervals (say, every 100 milliseconds), a timer interrupt transfers control to the operating system.
  - The operating system shifts the reference bit for each page into the high-order bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
- These 8-bit shift registers contain the history of page use for the last eight time periods.
  - **If the shift register contains 00000000, for example, then the page has not been used for eight time periods; a page that is used at least once in each period has a shift register value of 11111111.**
- A page with a history register value of **11000100** has been used more recently than one with a value of **01110111**. If we interpret these 8-bit bytes as unsigned integers, the page with the lowest number is the LRU page, and it can be replaced.

Second-chance Page-Replacement Algorithm

- 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

reference bits | pages

next victim

(a) circular queue of pages

reference bits | pages

(b) circular queue of pages

Enhanced Second-Chance Algorithm

- We can enhance the second-chance algorithm by considering the reference bit and the modify bit as an **ordered pair**.
- With these two bits, we have the following four possible classes:

1. (0,0) neither recently used nor modified-best page to replace
2. (0, 1) not recently used but modified – not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean-probably will be used again soon
4. (1, 1) recently used and modified-probably will be used again soon, and the page will need to be written out to disk before it can be replaced.

- *We* replace the first page encountered in the lowest nonempty class.

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used Algorithm**: replaces page with smallest count
  - **A problem arises, however, when a page is used heavily during the initial phase of a process but then is never used again.**
  - **Since it was used heavily, it has a large count and remains in memory even though it is no longer needed.**
- **Most Frequently Used Algorithm**: based on the argument that the page with the smallest count was probably **just brought in** and has yet to be used

Applications and Page Replacement

- In certain cases, applications accessing data through the operating system's virtual memory perform worse than if the operating system provided no buffering at all.
  - A typical example is a **database**, which provides its own memory management and I/O buffering. Applications like this understand their memory use and disk use **better than does an operating system** that is implementing algorithms for general-purpose use. If the operating system is buffering I/O, and the application is doing so as well, then twice the memory is being used for a set of I/O.
- In another example, **data warehouses** frequently perform massive sequential disk reads, followed by computations and writes. The LRU algorithm would be removing old pages and preserving new ones, while the application would more likely be reading older pages than newer ones (as it starts its sequential reads again). Here, MFU would actually be more efficient than LRU.

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 = size of process p_i
  - $S = \sum s_i$
  - m = total number of frames
  - a_i = 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

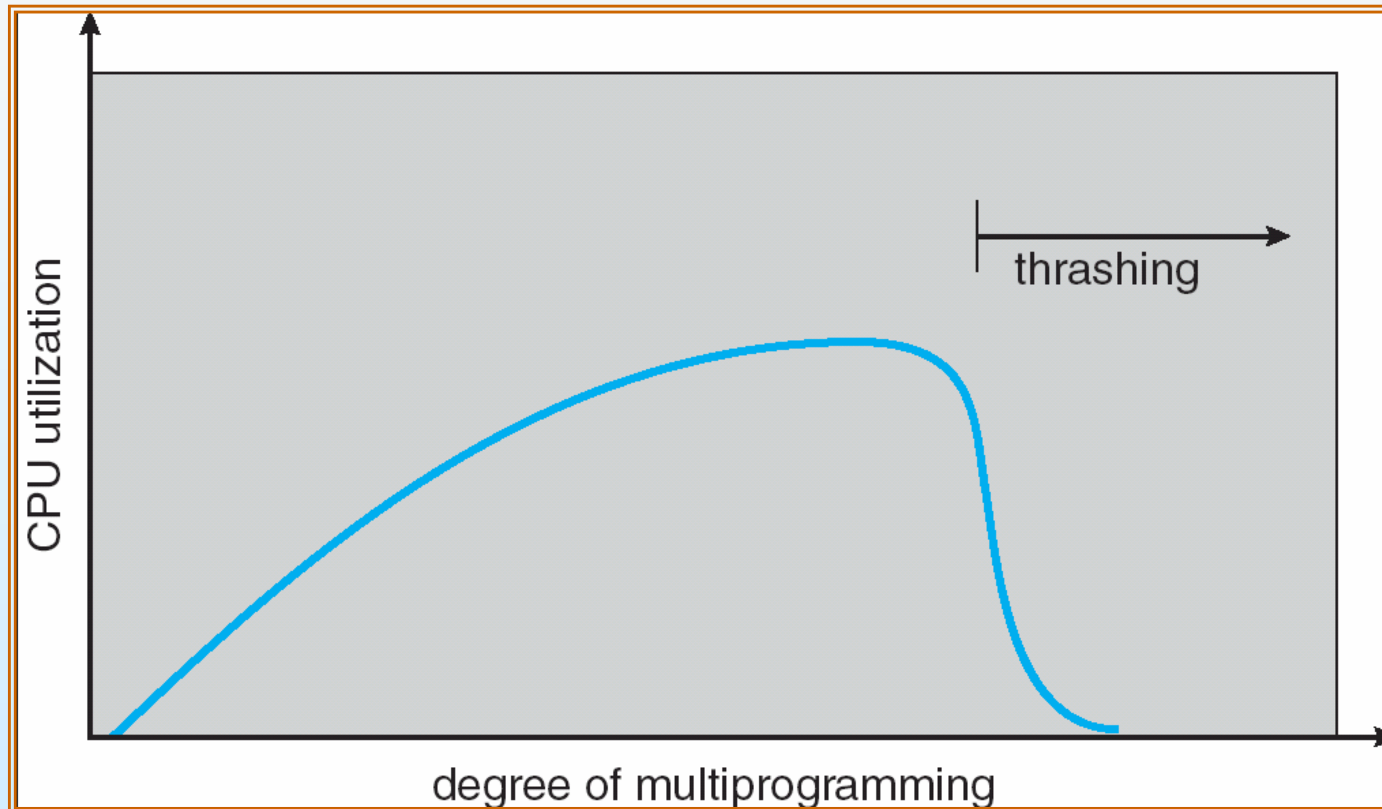
Global Vs. Local

- One problem with a global replacement algorithm is that a process cannot control its own page-fault rate.
  - The set of pages in memory for a process depends not only on the paging behavior of that process but also on the paging behavior of other processes.
  - Therefore, the same process may perform quite differently (for example, taking 0.5 seconds for one execution and 10.3 seconds for the next execution) because of **totally external circumstances.**
- Under local replacement, the set of pages in memory for a process is affected by the **paging behavior of only that process**.
  - Local replacement might hinder a process, by not making available to it less used pages of memory.
- Global replacement generally results in greater system throughput and is therefore the more common method.

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
- **Thrashing** ≡ a process is busy swapping pages in and out

Thrashing (Cont.)



Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another (A locality is a set of pages that are actively used together)
  - **Localities may overlap**

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

Silberschatz, Galvin and Gagne ©2005

Locality In A Memory-Reference Pattern

memory address

page numbers

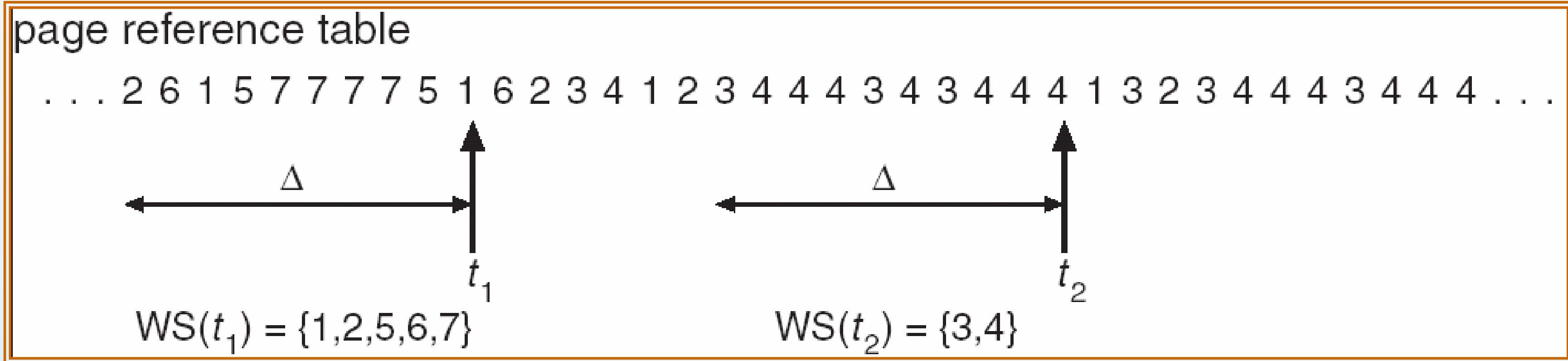
34
32
30
28
26
24
22
20
18

execution time →

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
  - 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, *m:* total number of available frames
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend one of the processes

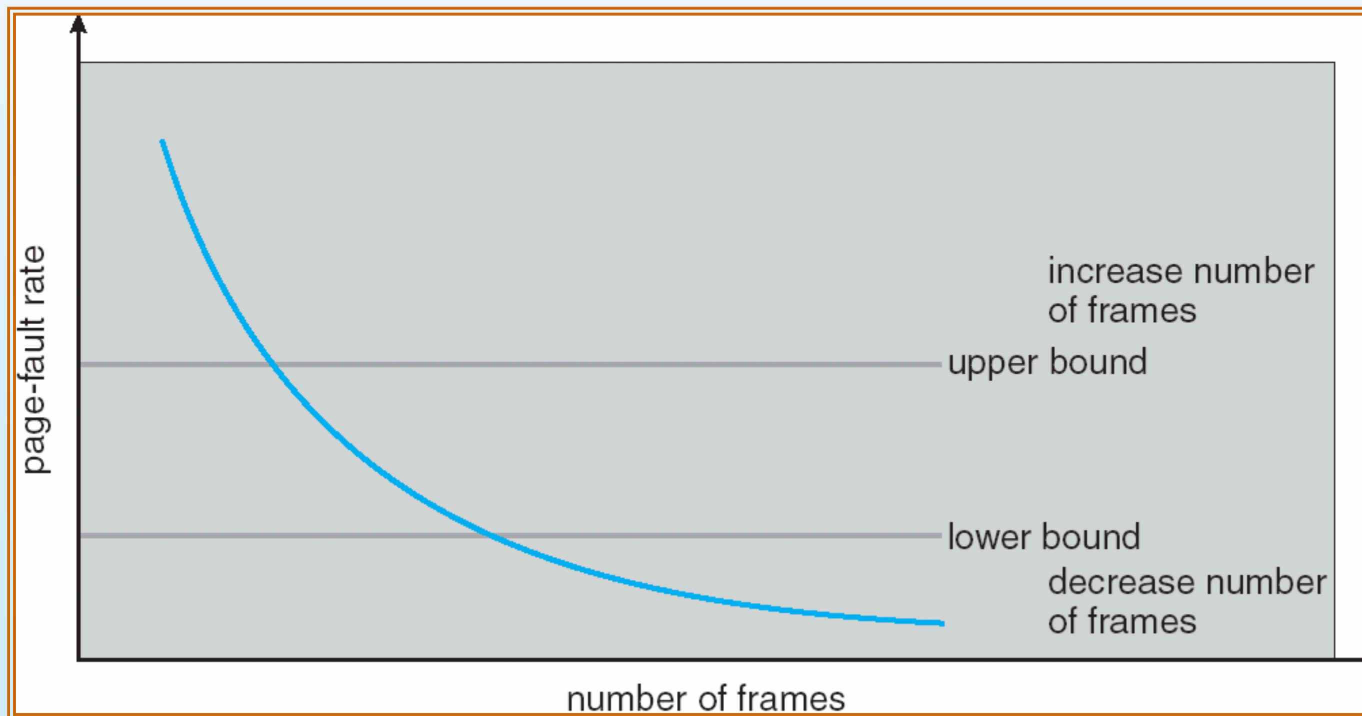
Working-set model



This working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus, it **optimizes CPU utilization.**

Page-Fault Frequency 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

- 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

process$_1$

shared memory

memory-mapped file

shared memory

process$_2$

shared memory

Allocating Kernel Memory

- Treated differently from user memory
  - Kernel memory, however, is often allocated from a **free-memory pool** different from the list used to satisfy ordinary user-mode processes.
- Two reasons for doing this:
  - Kernel requests memory for structures of varying sizes some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation.
  - Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory-without the benefit of a virtual memory interface-and consequently may require memory residing in physically contiguous pages.

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

Silberschatz, Galvin and Gagne ©2005

Buddy System Allocator: 21KB request by kernel

physically contiguous pages

256 KB

128 KB A_L

128 KB A_R

64 KB B_L

64 KB B_R

32 KB C_L

32 KB C_R

Slab Allocator

- Alternate strategy
- **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

Silberschatz, Galvin and Gagne ©2005

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 the cost of ***s* * *α* save pages faults** > or < than the cost of prepaging *s* * *(1- α)* unnecessary pages*?*
    - *If α* near zero ⇒ **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

- Assume that pages are 128 words in size
- 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

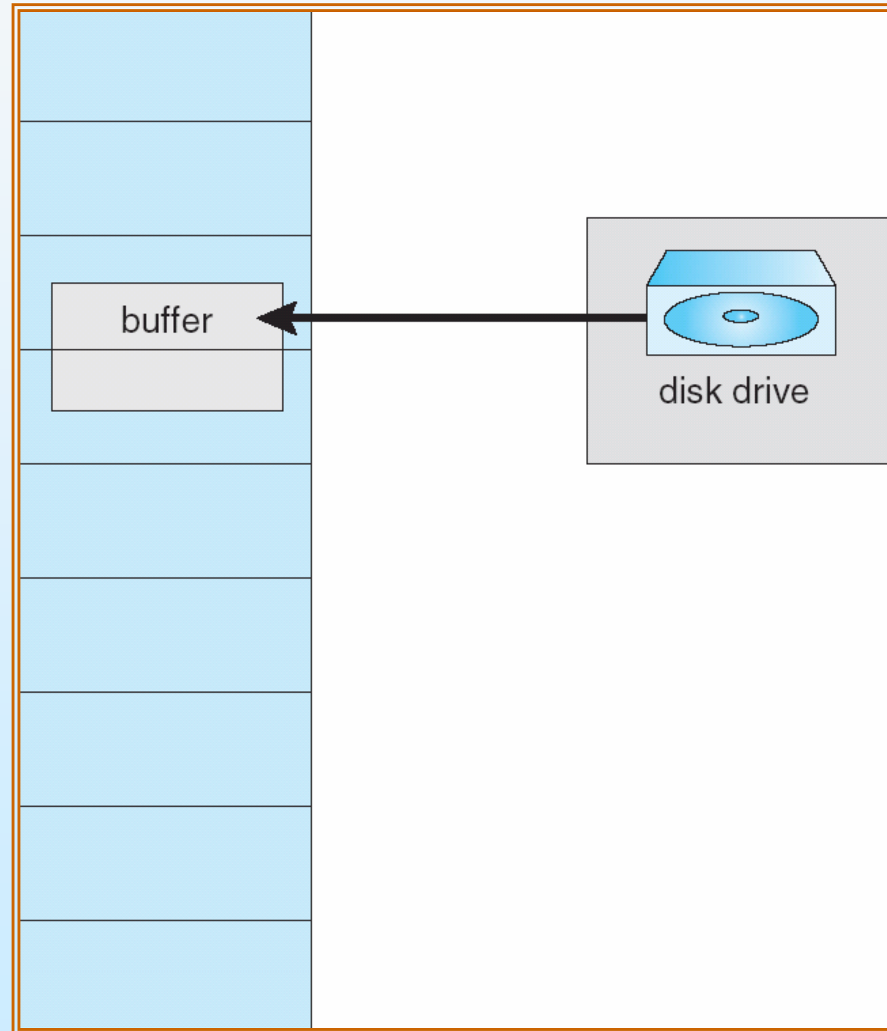
```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
            data[i,j] = 0;
```

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



Silberschatz, Galvin and Gagne ©2005

Operating System Examples

- Windows XP
- Solaris

Windows XP

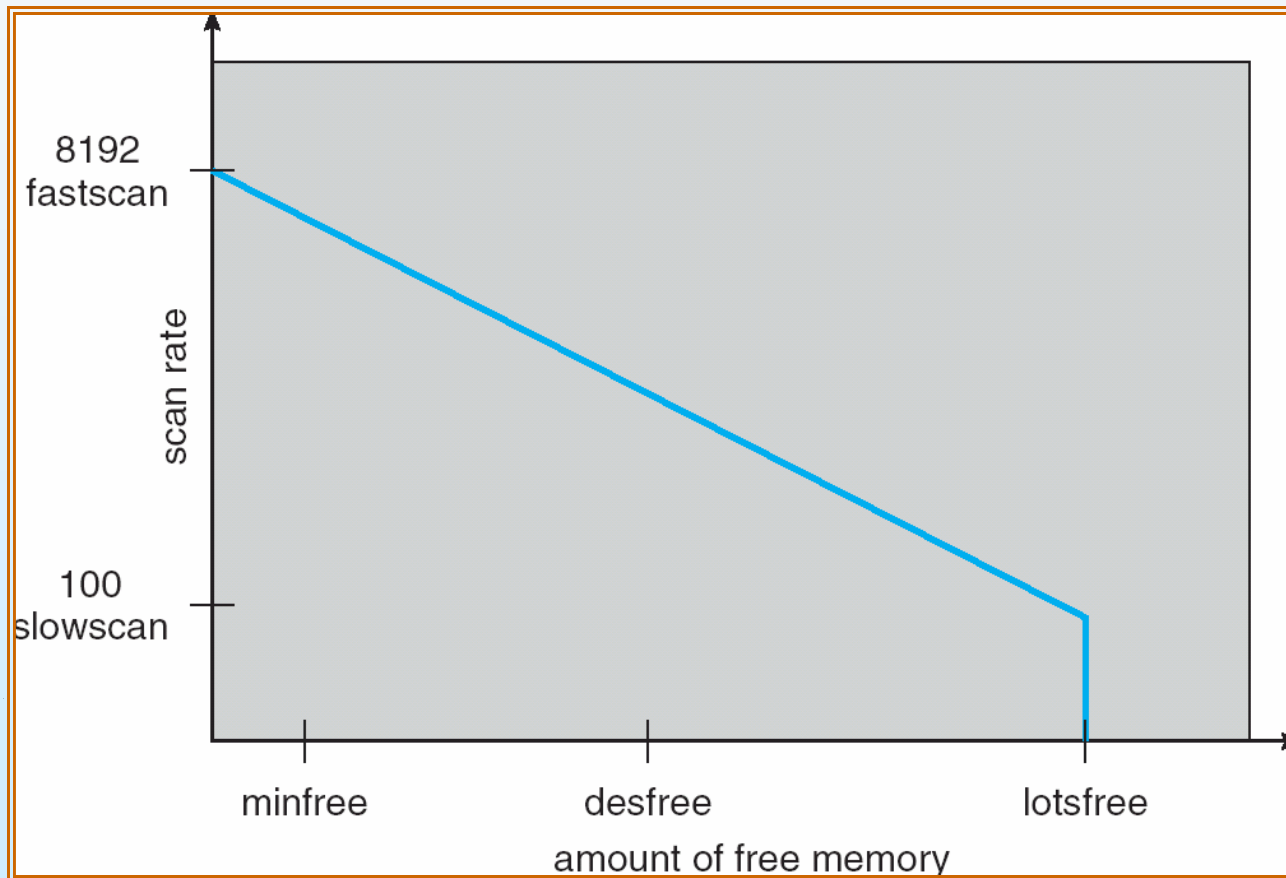
- Uses demand paging with **clustering**.
  - **Clustering brings in pages surrounding the faulting page.**
- Processes are assigned **working set minimum** and **working set maximum**
  - Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be 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

Silberschatz, Galvin and Gagne ©2005

Solaris 2 Page Scanner



Readings

- Chapter 9, except Sections 9.4.7; 9.7.3.

End of Chapter 9