

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
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed





Background

Code needs to be in memory to execute, but entire program rarely used

Error code, unusual routines, large data structures

Entire program code not needed at same time

Consider ability to execute partially-loaded program

Program no longer constrained by limits of physical memory

Each program takes less memory while running -> more programs run at the same time

- ▶ Increased CPU utilization and throughput with no increase in response time or turnaround time

Less I/O needed to load or swap programs into memory -> each user program runs faster





Background (Cont.)

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

- More programs running concurrently

- Less I/O needed to load or swap processes





Background (Cont.)

Virtual address space – logical view of how process is stored in memory

Usually start at address 0, contiguous addresses until end of space

Meanwhile, physical memory organized in page frames

MMU must map logical to physical

Virtual memory can be implemented via:

Demand paging

Demand segmentation





Virtual Memory That is Larger Than Physical Memory

```
linux1:~  
(linux1:~) ysw% cat /proc/self/maps  
00400000-0040b000 r-xp 00000000 09:03 100663537 /bin/cat  
0060a000-0060b000 r--p 0000a000 09:03 100663537 /bin/cat  
0060b000-0060c000 rw-p 0000b000 09:03 100663537 /bin/cat  
011b4000-011d5000 rw-p 00000000 00:00 0 [heap]  
7f015911f000-7f015f486000 r--p 00000000 09:03 627058 /usr/lib64/locale/locale-archive  
7f015f486000-7f015f61c000 r-xp 00000000 09:03 37223144 /lib64/libc-2.15.so  
7f015f61c000-7f015f81b000 r--p 00196000 09:03 37223144 /lib64/libc-2.15.so  
7f015f81b000-7f015f81f000 r--p 00195000 09:03 37223144 /lib64/libc-2.15.so  
7f015f81f000-7f015f821000 rw-p 00199000 09:03 37223144 /lib64/libc-2.15.so  
7f015f821000-7f015f825000 rw-p 00000000 00:00 0  
7f015f825000-7f015f846000 r-xp 00000000 09:03 37223145 /lib64/ld-2.15.so  
7f015fa29000-7f015fa2c000 rw-p 00000000 00:00 0  
7f015fa44000-7f015fa45000 rw-p 00000000 00:00 0  
7f015fa45000-7f015fa46000 r--p 00020000 09:03 37223145 /lib64/ld-2.15.so  
7f015fa46000-7f015fa47000 rw-p 00021000 09:03 37223145 /lib64/ld-2.15.so  
7f015fa47000-7f015fa48000 rw-p 00000000 00:00 0  
7ffffd9108000-7ffffd9129000 rw-p 00000000 00:00 0 [stack]  
7ffffd91ff000-7ffffd9200000 r-xp 00000000 00:00 0 [vdso]  
ffffffffffff600000-ffffffffffff601000 r-xp 00000000 00:00 0 [vsyscall]  
(linux1:~) ysw% █
```

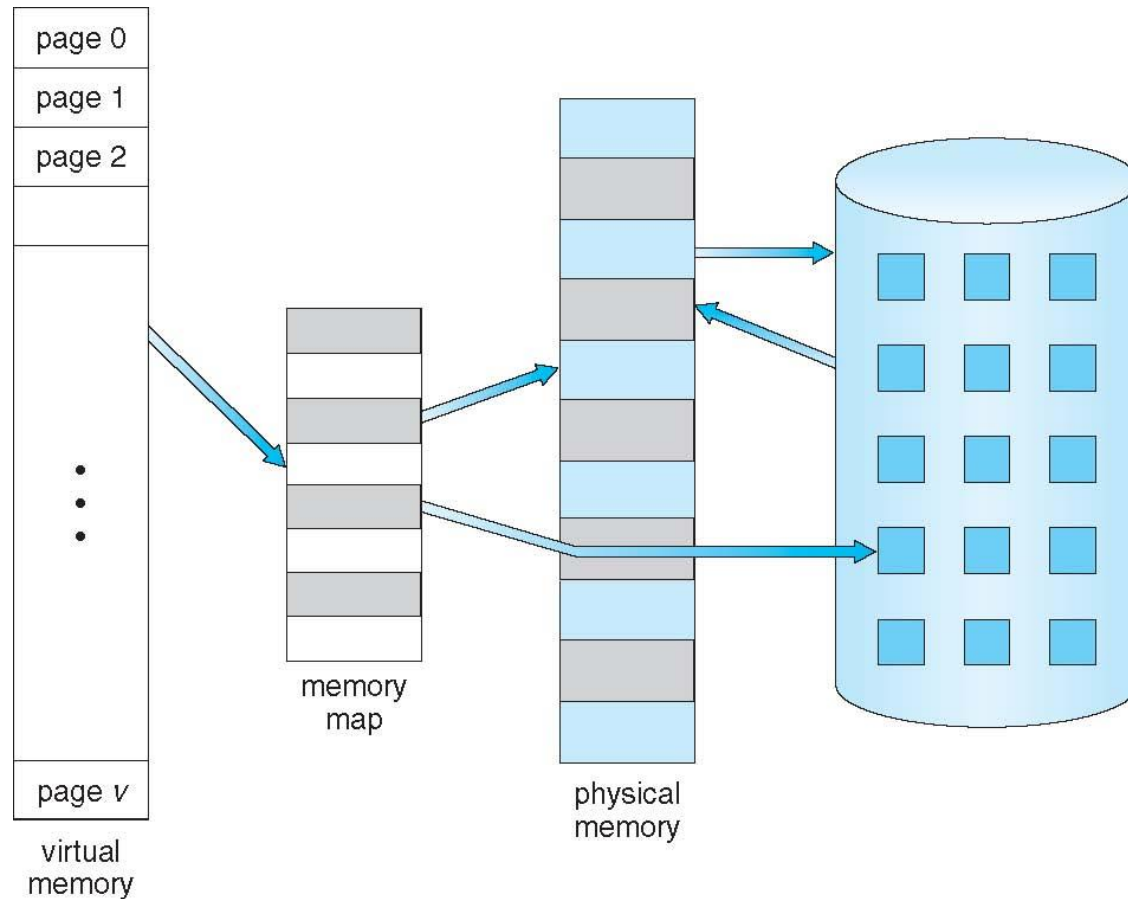
127 TB

16777216 TB





Virtual Memory That is Larger Than Physical Memory





Virtual-address Space

Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”

Maximizes address space use

Unused address space between the two is hole

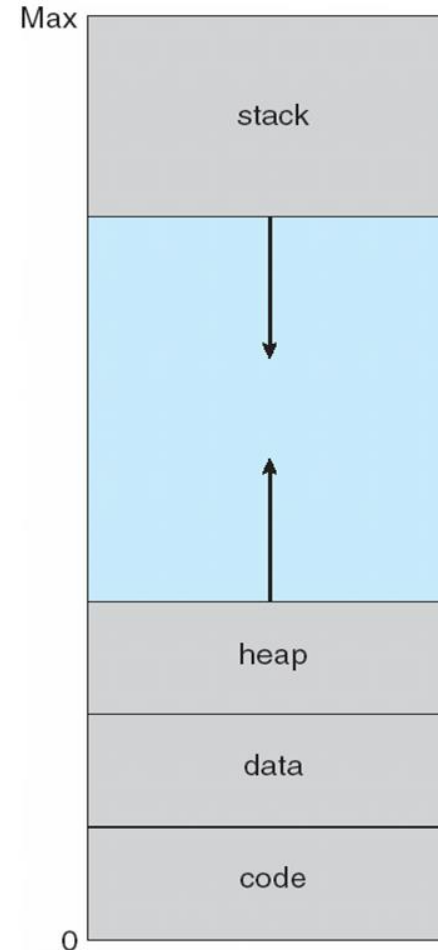
- ▶ No physical memory needed until heap or stack grows to a given new page

Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc

System libraries shared via mapping into virtual address space

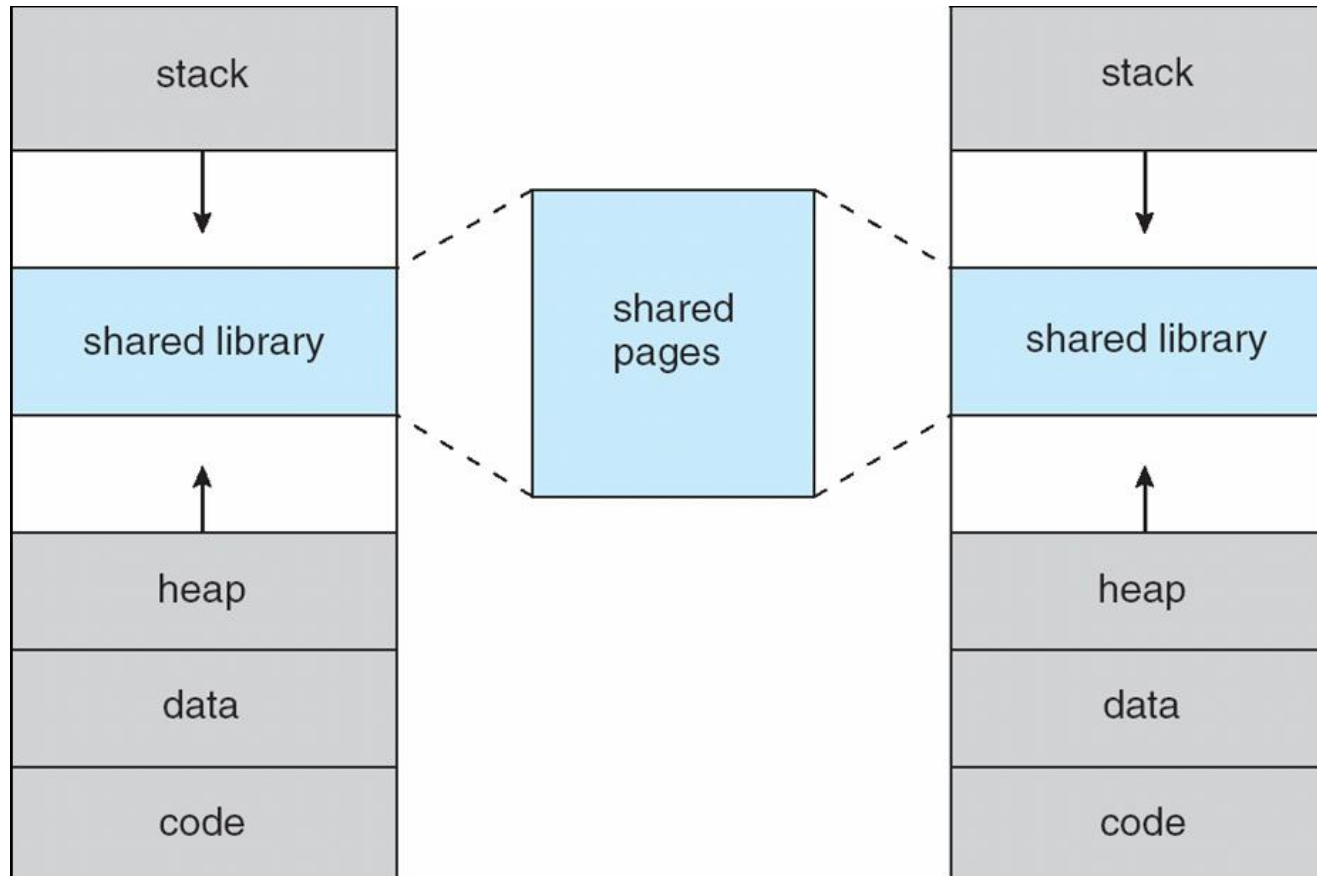
Shared memory by mapping pages read-write into virtual address space

Pages can be shared during `fork()`, speeding process creation





Shared Library Using Virtual Memory





Demand Paging

Could bring entire process into memory at load time

Or bring a page into memory only when it is needed

- Less I/O needed, no unnecessary I/O

- Less memory needed

- Faster response

- More users

Similar to paging system with swapping (diagram on right)

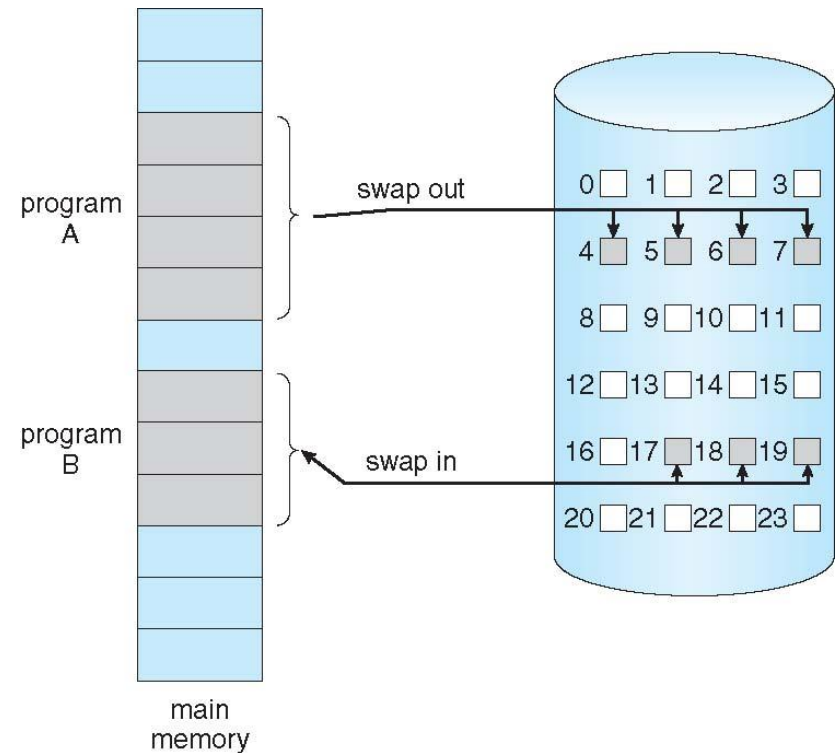
Page is needed \Rightarrow reference to it

- invalid reference \Rightarrow abort

- not-in-memory \Rightarrow bring to memory

Lazy swapper – never swaps a page into memory unless page will be needed

Swapper that deals with pages is a **pager**





Basic Concepts

With swapping, pager guesses which pages will be used before swapping out again

Instead, pager brings in only those pages into memory

How to determine that set of pages?

Need new MMU functionality to implement demand paging

If pages needed are already **memory resident**

No difference from non demand-paging

If page needed and not memory resident

Need to detect and load the page into memory from storage

- ▶ Without changing program behavior
- ▶ Without programmer needing to change code





Valid-Invalid Bit

With each page table entry a valid–invalid bit is associated
(**v** \Rightarrow in-memory – **memory resident**, **i** \Rightarrow 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
	i
...	
	i
	i

page table

During MMU address translation, if valid–invalid bit in page table entry is **i** \Rightarrow page fault





Page Table When Some Pages Are Not in Main Memory

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

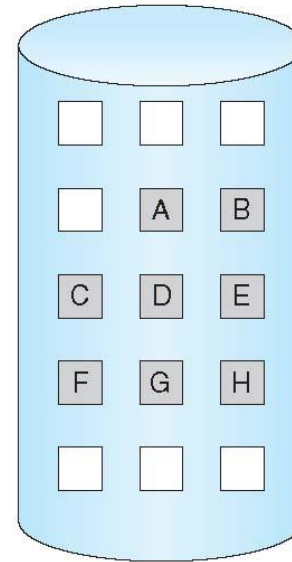
logical
memory

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

page table

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

physical memory





Page Fault

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

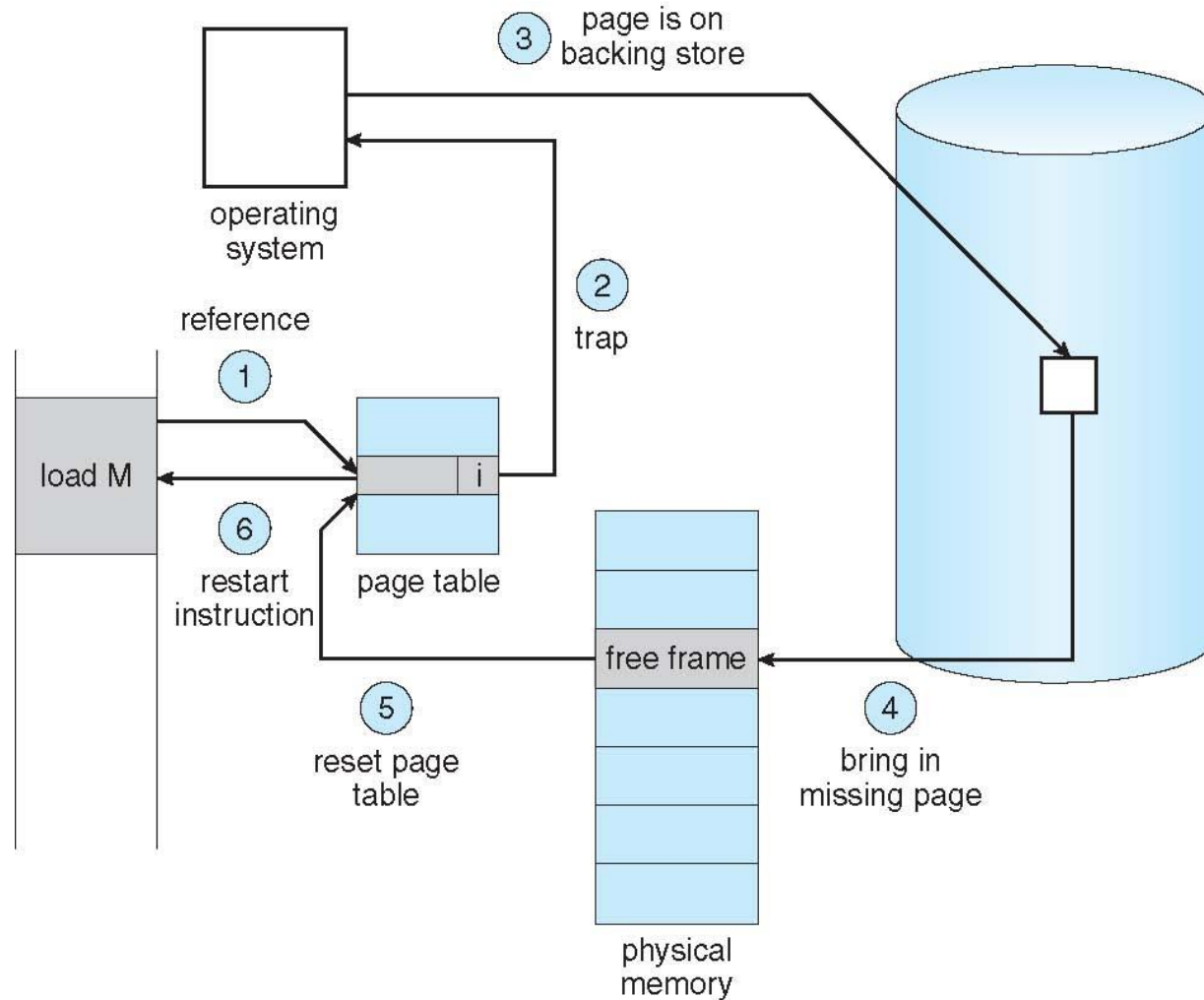
page fault

1. Operating system looks at another table to decide:
Invalid reference \Rightarrow abort
Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory
Set validation bit = **v**
5. Restart the instruction that caused the page fault





Steps in Handling a Page Fault





Aspects of Demand Paging

Extreme case – start process with *no* pages in memory

OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault

And for every other process pages on first access

Pure demand paging

Actually, a given instruction could access multiple pages -> multiple page faults

Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory

Pain decreased because of **locality of reference**

Hardware support needed for demand paging

Page table with valid / invalid bit

Secondary memory (swap device with **swap space**)

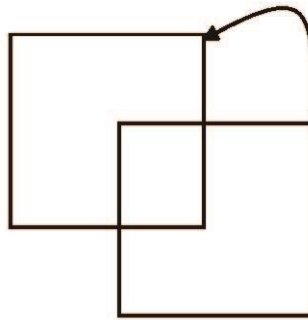
Instruction restart





Instruction Restart

Consider an instruction that could access several different locations
block move



auto increment/decrement location

Restart the whole operation?

- ▶ What if source and destination overlap?





Performance of Demand Paging

Stages in Demand Paging (worst case)

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:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show 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 (Cont.)

Three major activities

Service the interrupt – careful coding means just several hundred instructions needed

Read the page – lots of time

Restart the process – again just a small amount of time

Page Fault Rate $0 \leq p \leq 1$

if $p = 0$ no page faults

if $p = 1$, every reference is a fault

Effective Access Time (EAT)

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





Demand Paging Example

Memory access time = 200 nanoseconds

Average page-fault service time = 8 milliseconds

$EAT = (1 - p) \times 200 + p (8 \text{ milliseconds})$

$= (1 - p) \times 200 + p \times 8,000,000$

$= 200 + p \times 7,999,800$

If one access out of 1,000 causes a page fault, then

$EAT = 8.2 \text{ microseconds.}$

This is a slowdown by a factor of 40!!

If want performance degradation < 10 percent

$220 > 200 + 7,999,800 \times p$

$20 > 7,999,800 \times p$

$p < .0000025$

< one page fault in every 400,000 memory accesses





Demand Paging Optimizations

Swap space I/O faster than file system I/O even if on the same device

Swap allocated in larger chunks, less management needed than file system

Copy entire process image to swap space at process load time

Then page in and out of swap space

Used in older BSD Unix

Demand page in from program binary on disk, but discard rather than paging out when freeing frame

Used in Solaris and current BSD

Still need to write to swap space

- ▶ Pages not associated with a file (like stack and heap) – **anonymous memory**
- ▶ Pages modified in memory but not yet written back to the file system

Mobile systems

Typically don't support swapping

Instead, demand page from file system and reclaim read-only pages (such as code)





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

In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages

Pool should always have free frames for fast demand page execution

- ▶ Don't want to have to free a frame as well as other processing on page fault

Why zero-out a page before allocating it?

`vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent

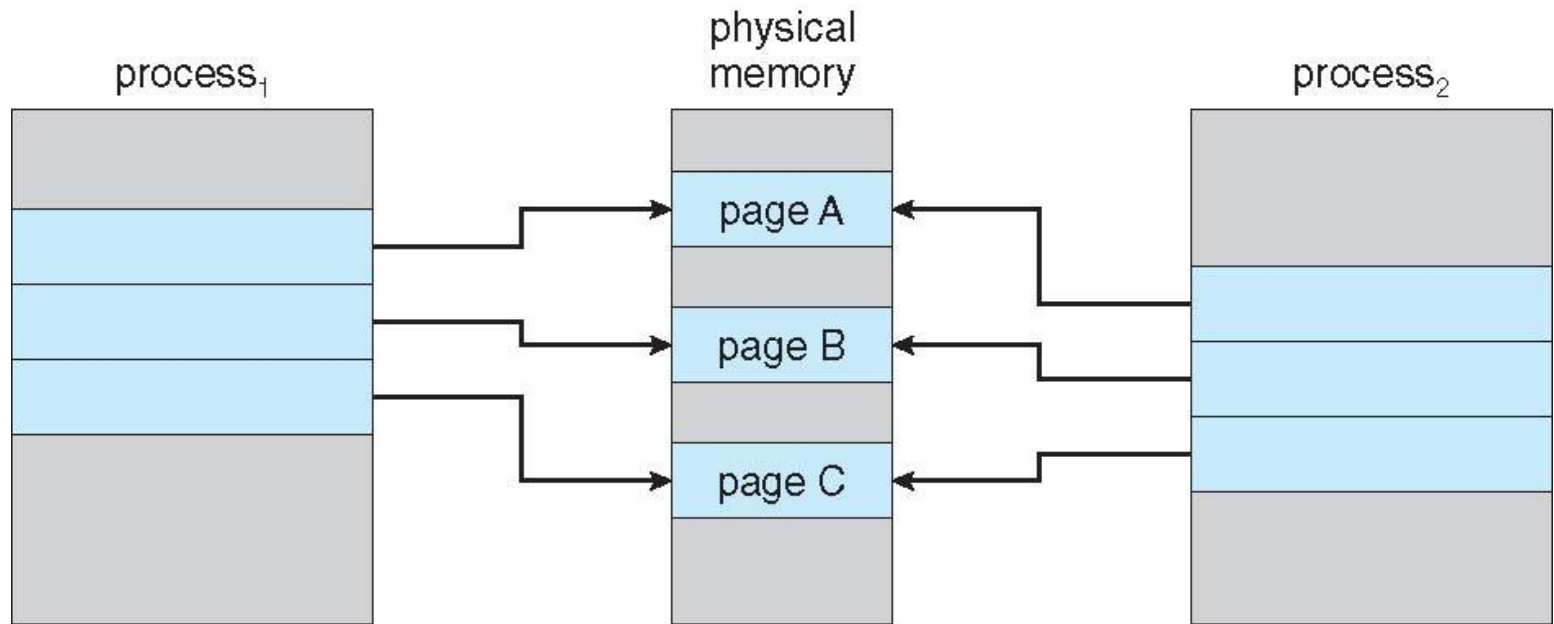
Designed to have child call `exec()`

Very efficient



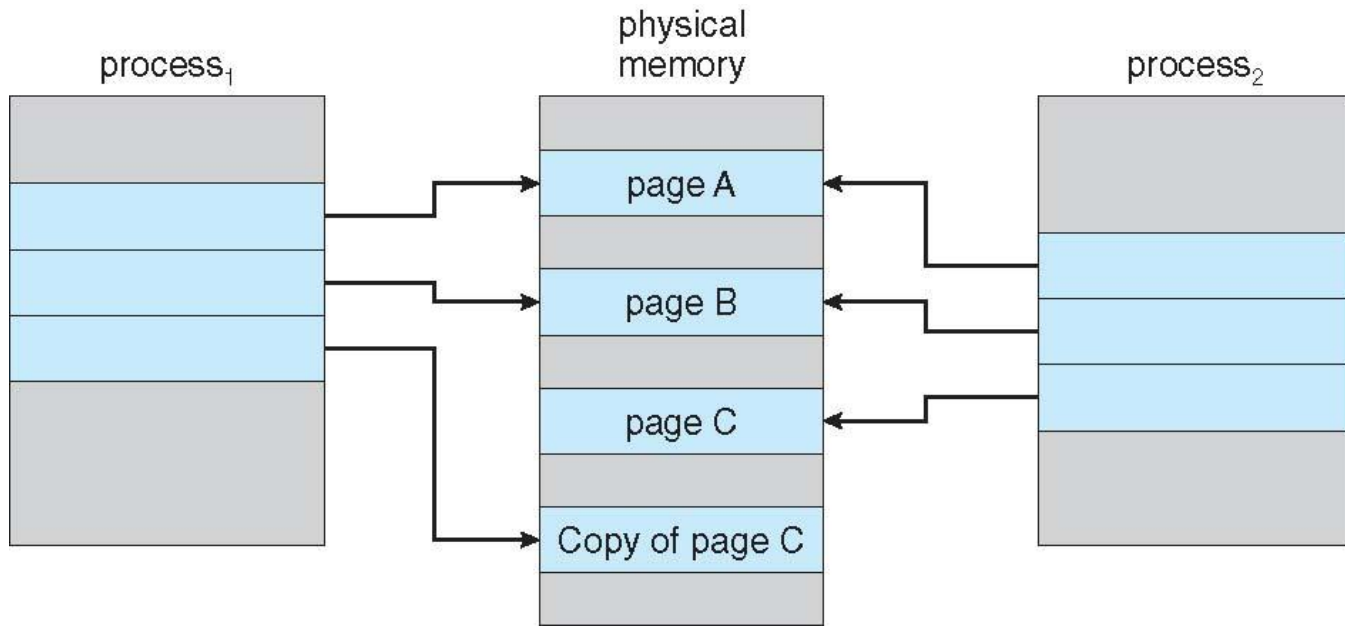


Before Process 1 Modifies Page C





After Process 1 Modifies Page C





What Happens if There is no Free Frame?

Used up by process pages

Also in demand from the kernel, I/O buffers, etc

How much to allocate to each?

Page replacement – find some page in memory, but not really in use, page it out

Algorithm – terminate? swap out? replace the page?

Performance – want an algorithm which will result in minimum number of page faults

Same page may be brought into memory several times





Page Replacement

Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement

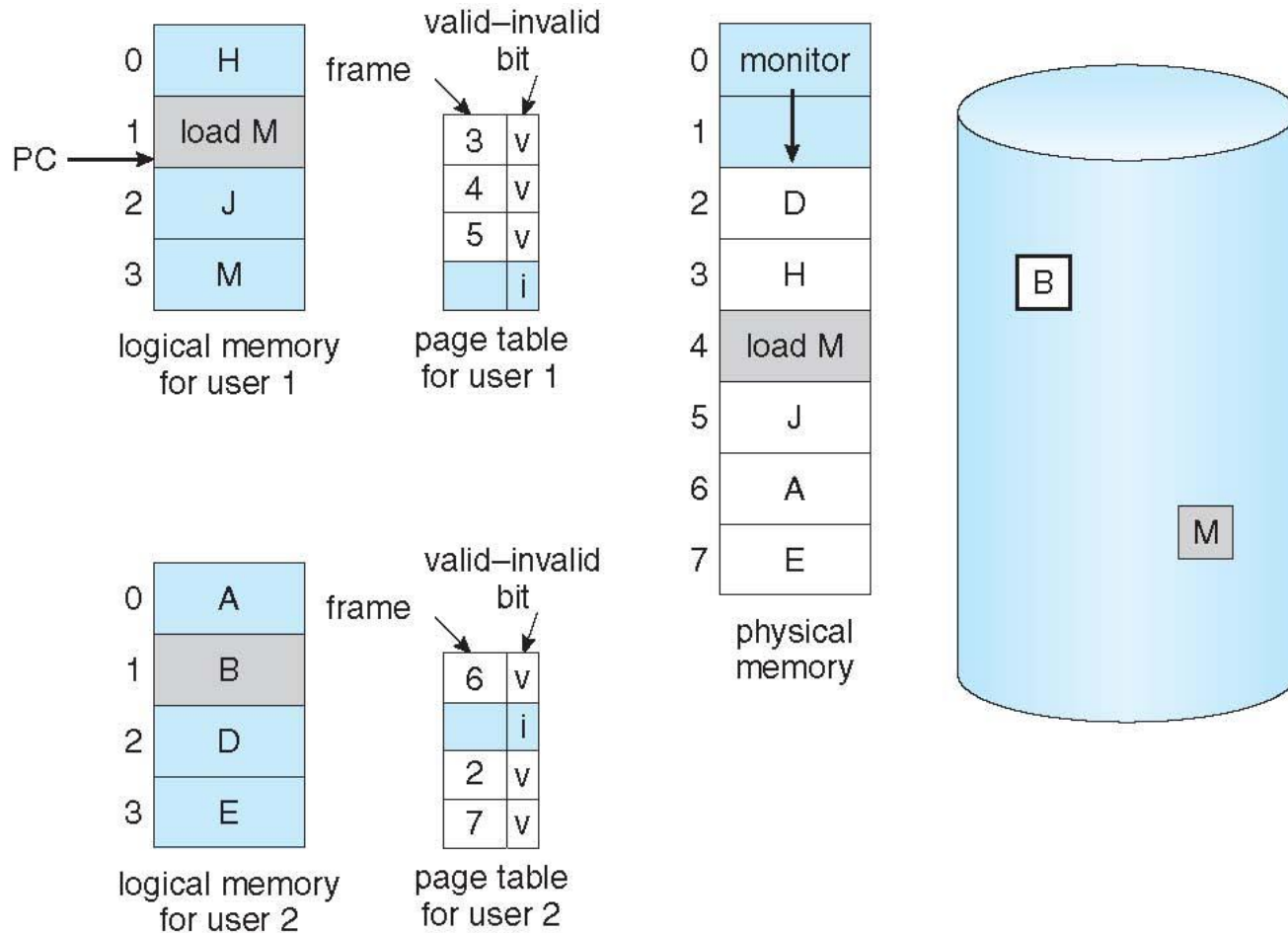
Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

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





Need For Page Replacement





Basic Page Replacement

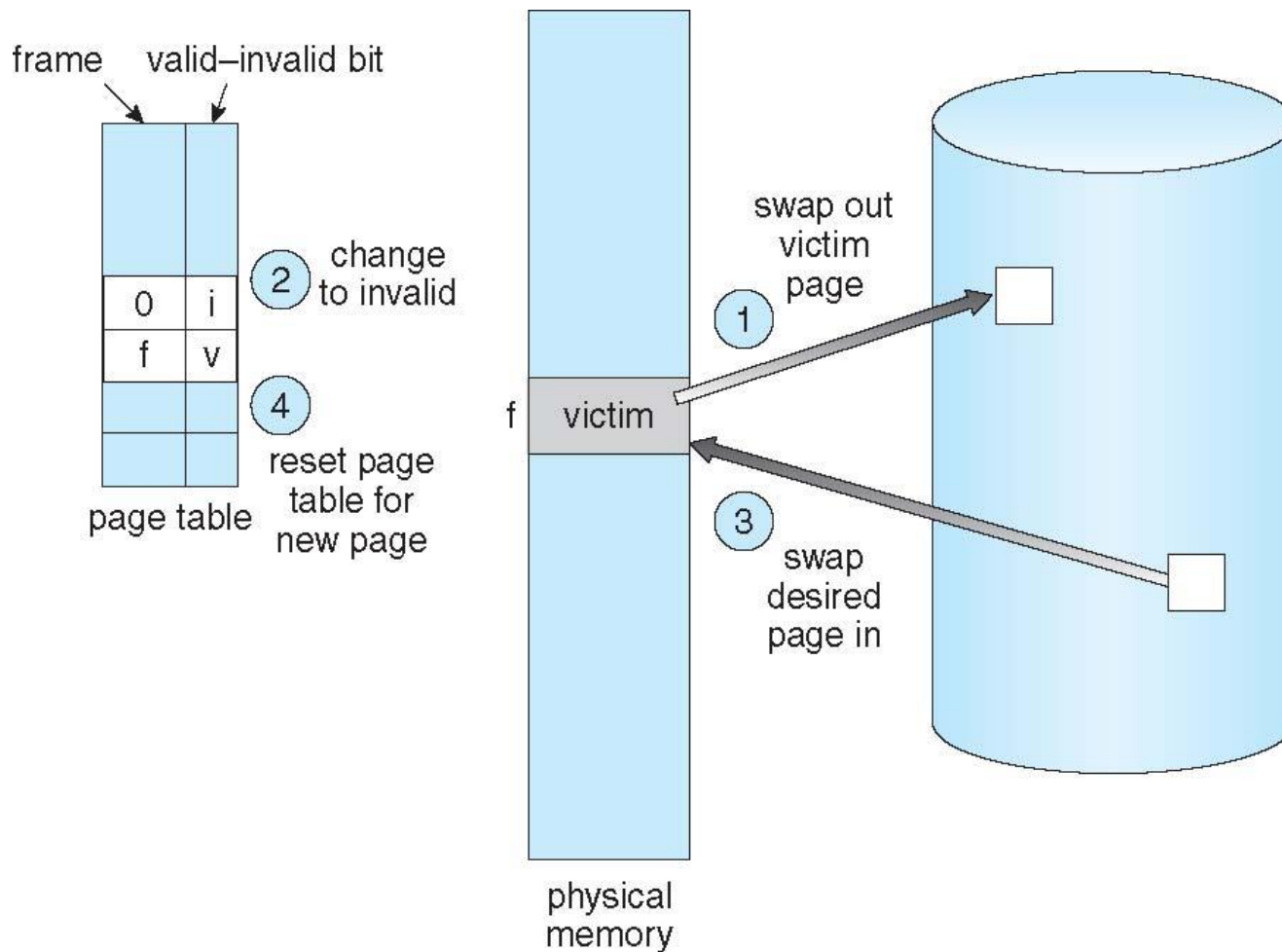
1. Find the location of the desired page on disk
2. Find a free frame:
 - If there is a free frame, use it
 - If there is no free frame, use a page replacement algorithm to select a **victim frame**
 - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT





Page Replacement





Page and Frame Replacement Algorithms

Frame-allocation algorithm determines

- How many frames to give each process

- Which frames to replace

Page-replacement algorithm

- Want lowest page-fault rate on both first access and re-access

Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string

- String is just page numbers, not full addresses

- Repeated access to the same page does not cause a page fault

- Results depend on number of frames available

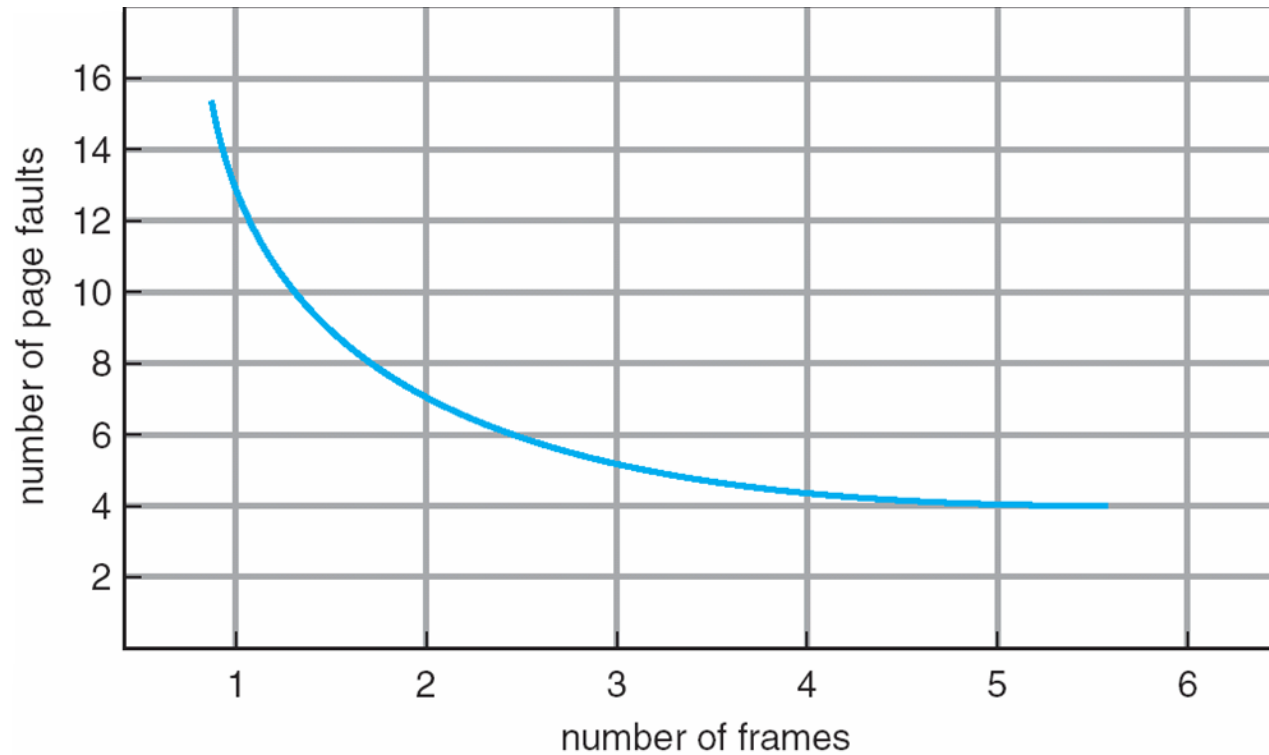
In all our examples, the **reference string** of referenced page numbers is

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





Graph of Page Faults Versus The Number of Frames





First-In-First-Out (FIFO) Algorithm

Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

3 frames (3 pages can be in memory at a time per process)

reference string

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

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

page frames

15 page faults

Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5

Adding more frames can cause more page faults!

► **Belady's Anomaly**

How to track ages of pages?

Just use a FIFO queue





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)

4 frames

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

9 page faults

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

10 page faults

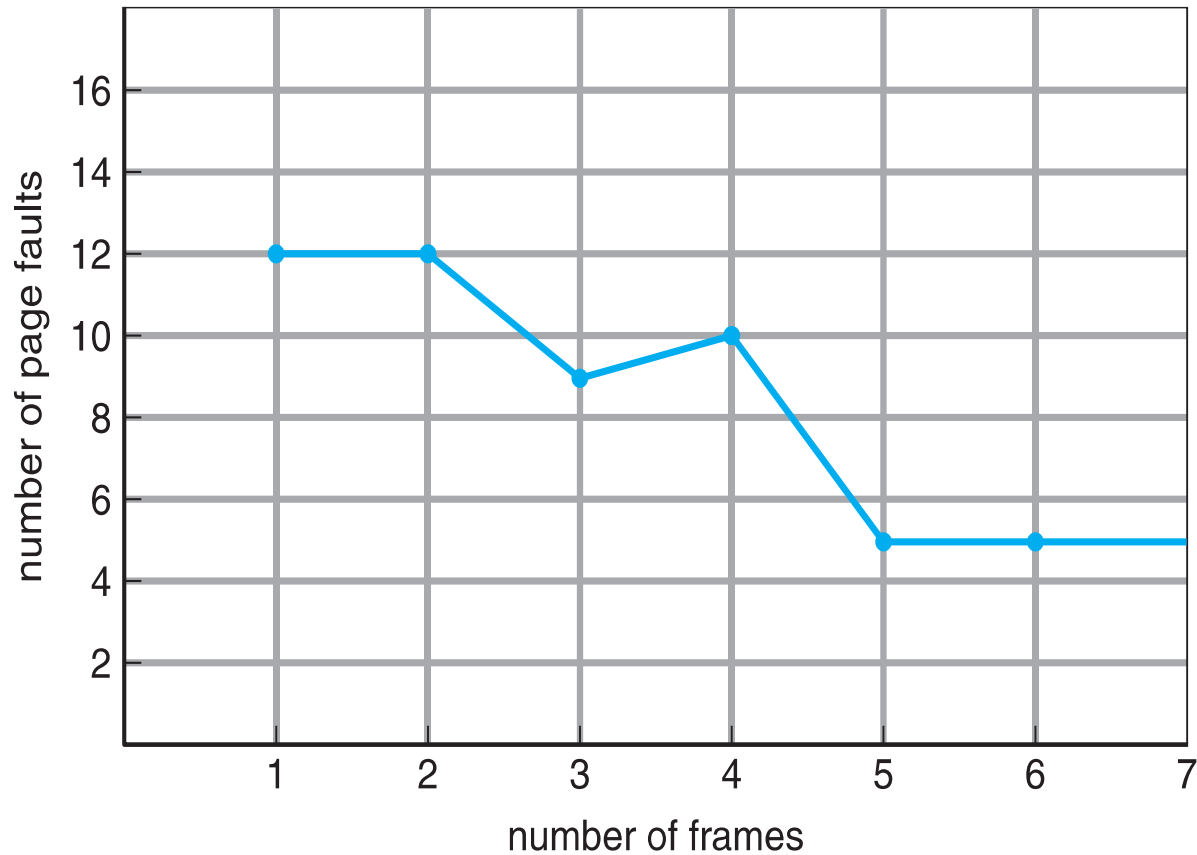
With FIFO,
contents can be
completely
different after
adding memory

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





FIFO Illustrating Belady's Anomaly





Optimal Algorithm

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

4 frames example

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

1
2
3
4

4

6 page faults

5

How do you know this?

Used for measuring how well your algorithm performs





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

page frames

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





Least Recently Used (LRU) Algorithm

Use past knowledge rather than future

Replace page that has not been used in the most amount of time

Associate time of last use with each page

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

page frames

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

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





Least Recently Used (LRU) Algorithm

Assumption: A page that has not been referenced for the longest time would wait for the longest time to be accessed again

Use of known history to predict unknown future

Does the intuition really work?

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

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

Counter implementation

Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter

When a page needs to be changed, look at the counters to determine which are to change





LRU Algorithm (Cont.)

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 find smallest value

- ▶ Search through table needed

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

But each update more expensive

No search for replacement

LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly





Use Of A Stack to Record 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

LRU needs special hardware and still slow

Reference bit

With each page associate a bit, initially = 0

When page is referenced bit set to 1

Replace any with reference bit = 0 (if one exists)

- ▶ We do not know the order, however

Second-chance algorithm

Generally FIFO, plus hardware-provided reference bit

Clock replacement

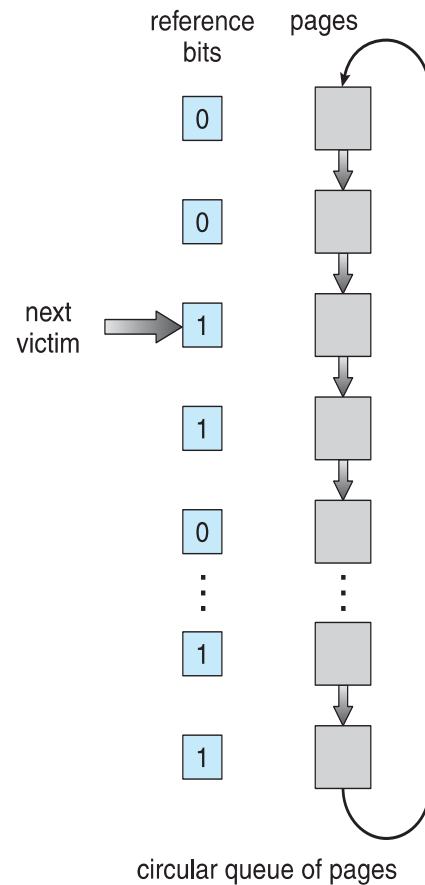
If page to be replaced has

- ▶ Reference bit = 0 -> replace it
- ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory
 - replace next page, subject to same rules

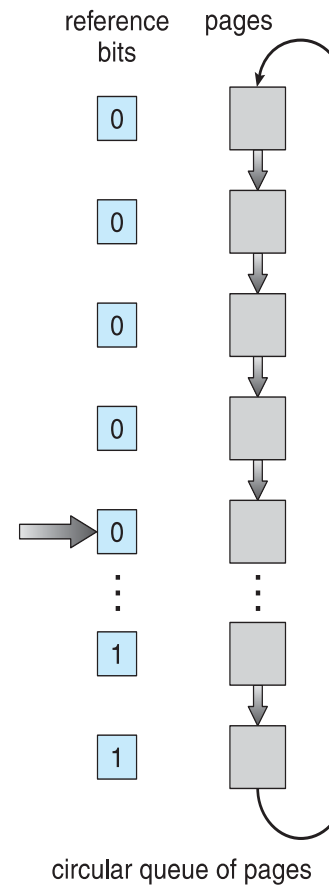




Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)





Enhanced Second-Chance Algorithm

Improve algorithm by using reference bit and modify bit (if available) in concert

Take ordered pair (reference, modify)

1. (0, 0) neither recently used nor modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write 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 need to write out before replacement

When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class

Might need to search circular queue several times





Counting Algorithms

Keep a counter of the number of references that have been made to each page

Not common

Least Frequently Used (LFU) Algorithm: replaces page with smallest count

Most Frequently Used (MFU) Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used





Page-Buffering Algorithms

Keep a pool of free frames, always

Then frame available when needed, not found at fault time

Read page into free frame and select victim to evict and add to free pool

When convenient, evict victim

Possibly, keep a list of modified pages

When backing store otherwise idle, write pages there and set to non-dirty

Possibly, keep free frame contents intact and note what is in them

If referenced again before reused, no need to load contents again from disk

Generally useful to reduce penalty if wrong victim frame selected





Applications and Page Replacement

All of these algorithms have OS guessing about future page access

Some applications have better knowledge – i.e. databases

Memory intensive applications can cause double buffering

- OS keeps copy of page in memory as I/O buffer

- Application keeps page in memory for its own work

Operating system can given direct access to the disk, getting out of the way of the applications

- Raw disk** mode

- Bypasses buffering, locking, etc





Allocation of Frames

Each process needs ***minimum*** number of frames

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*

Maximum of course is total frames in the system

Two major allocation schemes

fixed allocation

priority allocation

Many variations





Fixed Allocation

Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

Keep some as free frame buffer pool

Proportional allocation – Allocate according to the size of process

Dynamic as degree of multiprogramming, process sizes change

– s_i = size of process p_i

– $S = \sum s_i$

– m = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \cdot 62 \gg 4$$

$$a_2 = \frac{127}{137} \cdot 62 \gg 57$$





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

But then process execution time can vary greatly

But greater throughput so more common

Local replacement – each process selects from only its own set of allocated frames

More consistent per-process performance

But possibly underutilized memory





Non-Uniform Memory Access

So far all memory accessed equally

Many systems are **NUMA** – speed of access to memory varies

Consider system boards containing CPUs and memory,
interconnected over a system bus

Optimal performance comes from allocating memory “close to”
the CPU on which the thread is scheduled

And modifying the scheduler to schedule the thread on the
same system board when possible

Solved by Solaris by creating **lggroups**

- ▶ Structure to track CPU / Memory low latency groups
- ▶ Used my schedule and pager
- ▶ When possible schedule all threads of a process and
allocate all memory for that process within the lggroup





Thrashing

If a process does not have “enough” pages, the page-fault rate is very high

Page fault to get page

Replace existing frame

But quickly need replaced frame back

This leads to:

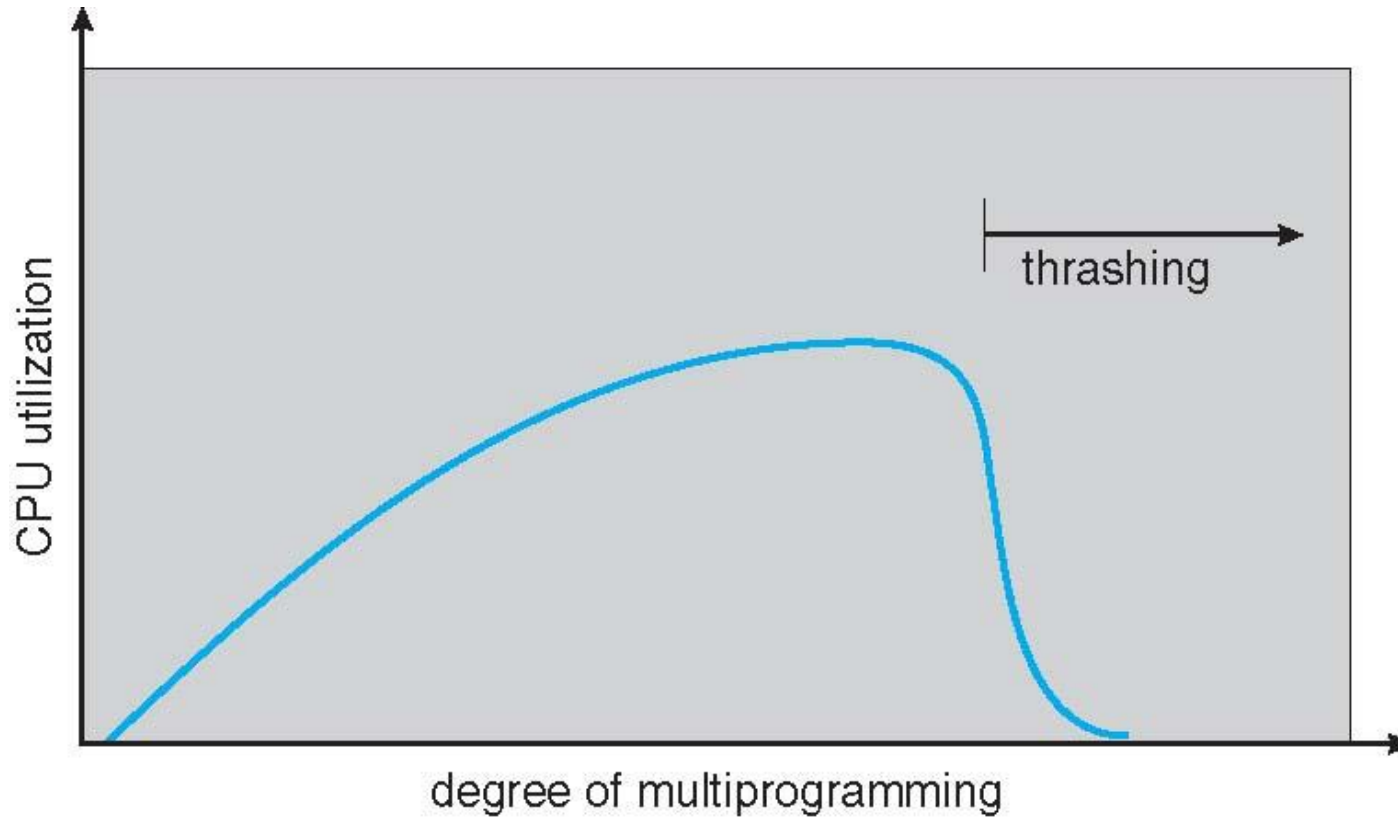
- ▶ Low CPU utilization
- ▶ Operating system thinking that it needs to increase the degree of multiprogramming
- ▶ Another process added to the system

Thrashing \equiv a process is busy swapping pages in and out





Thrashing (Cont.)





Locality In A Memory-Reference Pattern

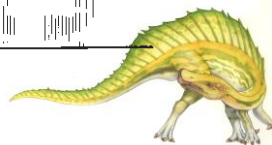
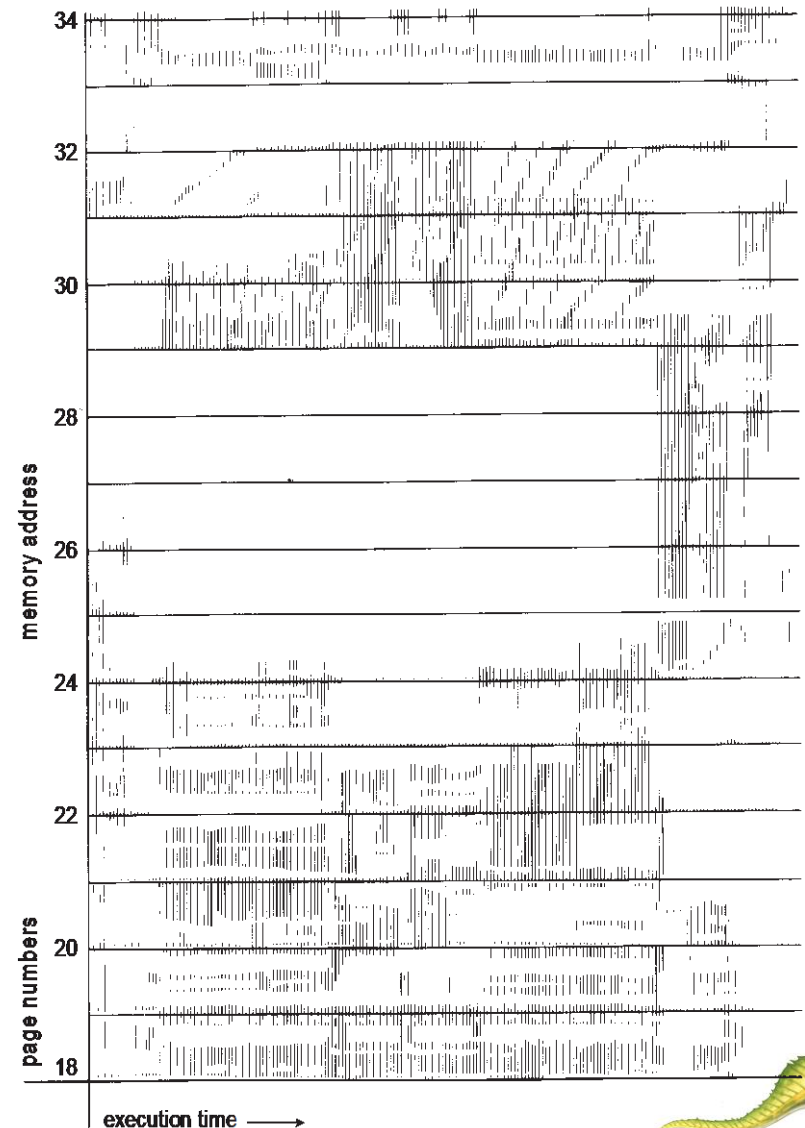
why does paging work?

locality model

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

why does thrashing occur?

Σ locality sizes > total memory size





Working-Set Model

$\Delta \equiv$ working-set window \equiv a fixed number of page references

Example: 10,000 instructions

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

$D = \sum WSS_i \equiv$ total demand frames

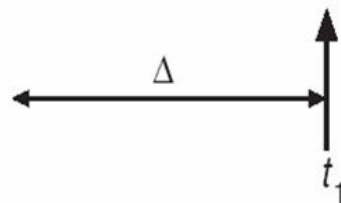
Approximation of locality

if $D > m \Rightarrow$ Thrashing

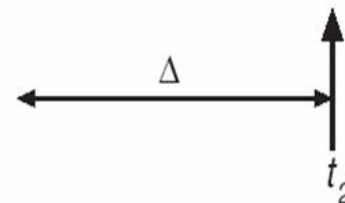
Policy if $D > m$, then suspend or swap out one of the processes

page reference table

... 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$WS(t_1) = \{1, 2, 5, 6, 7\}$



$WS(t_2) = \{3, 4\}$





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 sets the values of all reference bits to 0

If one of the bits in memory = 1 \Rightarrow page in working set

Why is this not completely accurate?

Improvement = 10 bits and interrupt every 1000 time units





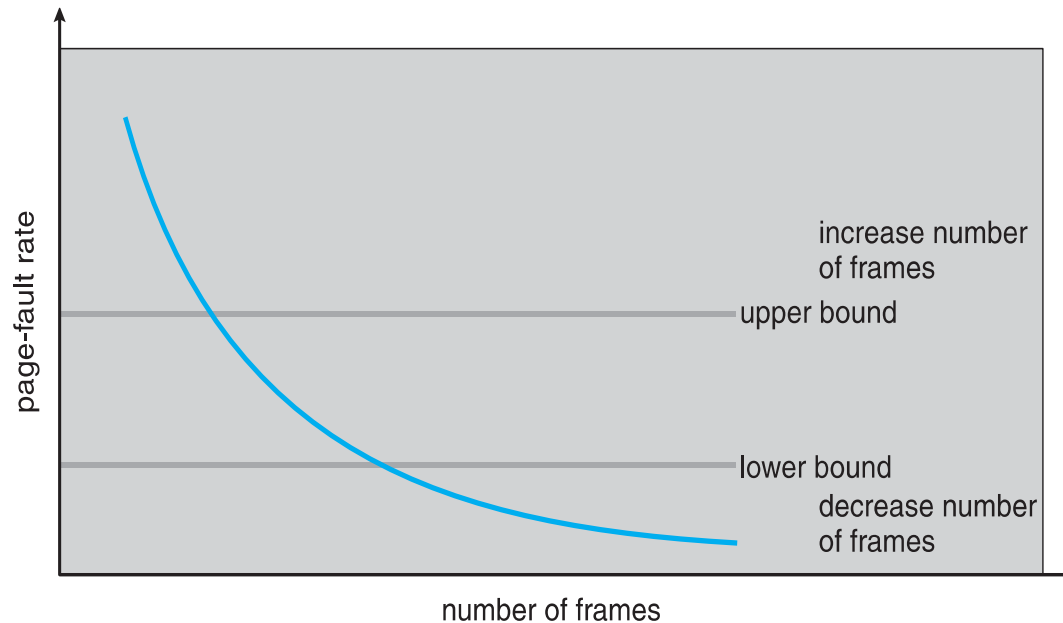
Page-Fault Frequency

More direct approach than WSS

Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy

If actual rate too low, process loses frame

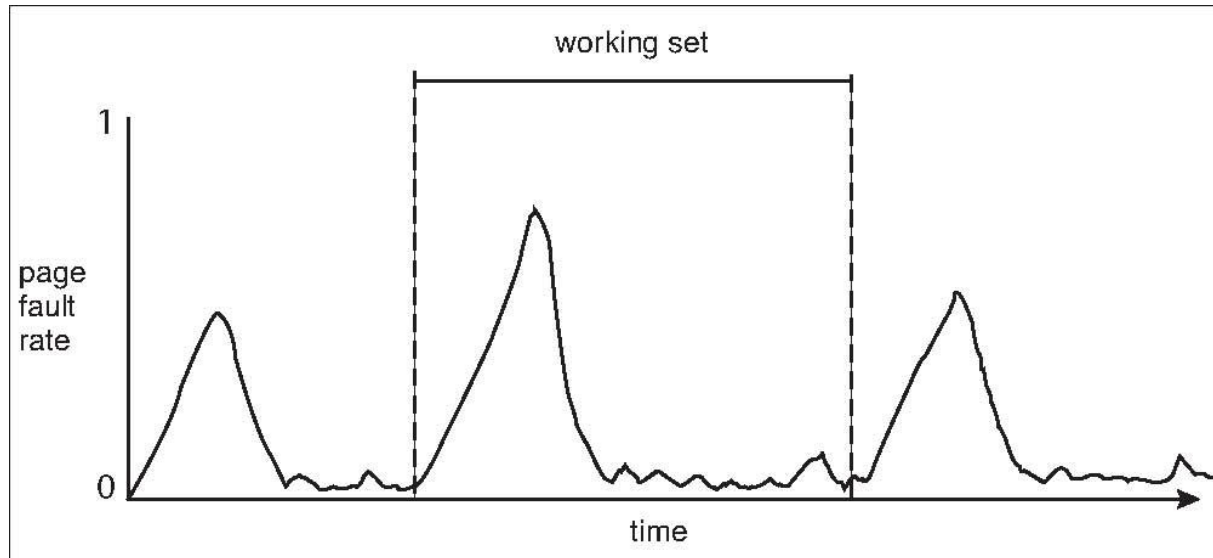
If actual rate too high, process gains frame

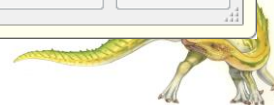
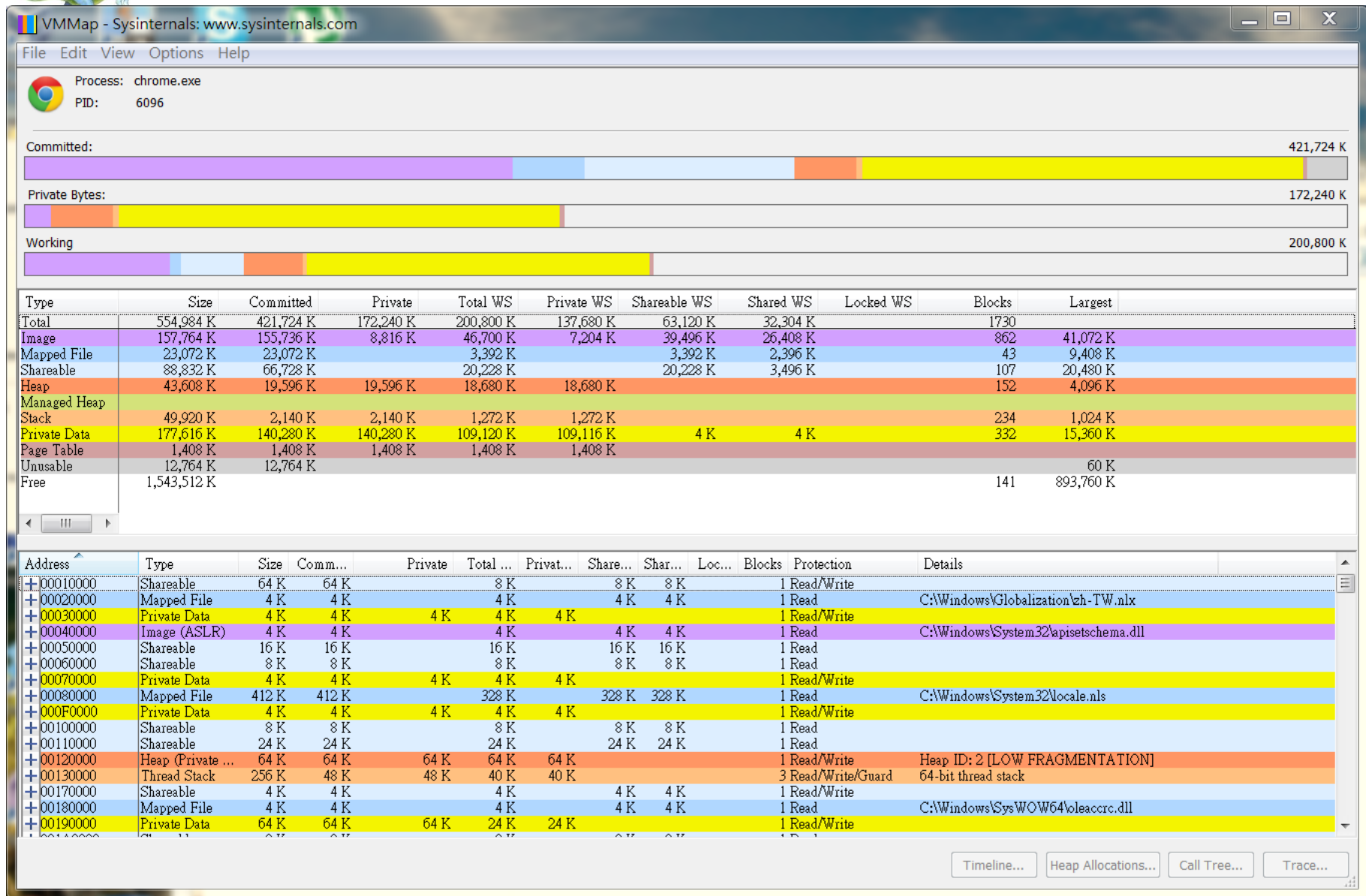




Working Sets and Page Fault Rates

- n Direct relationship between working set of a process and its page-fault rate
- n Working set changes over time
- n Peaks and valleys over time







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 and speeds file access by driving file I/O through memory rather than `read()` and `write()` system calls

Also allows several processes to map the same file allowing the pages in memory to be shared

But when does written data make it to disk?

- Periodically and / or at file `close()` time

- For example, when the pager scans for dirty pages





Memory-Mapped File Technique for all I/O

Some OSes use memory mapped files for standard I/O

Process can explicitly request memory mapping a file via `mmap()` system call

Now file mapped into process address space

For standard I/O (`open()`, `read()`, `write()`, `close()`), `mmap` anyway

But map file into kernel address space

Process still does `read()` and `write()`

- ▶ Copies data to and from kernel space and user space

Uses efficient memory management subsystem

- ▶ Avoids needing separate subsystem

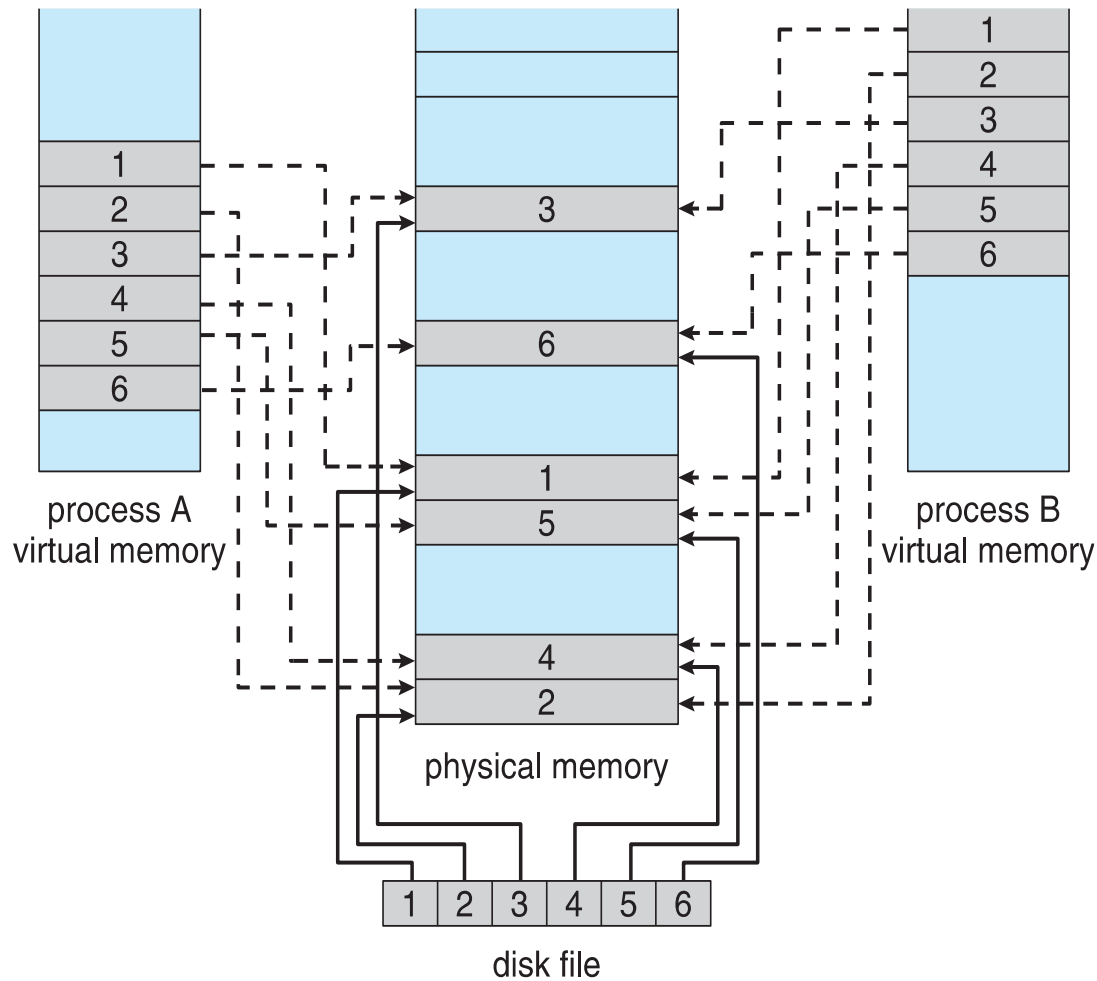
COW can be used for read/write non-shared pages

Memory mapped files can be used for shared memory (although again via separate system calls)



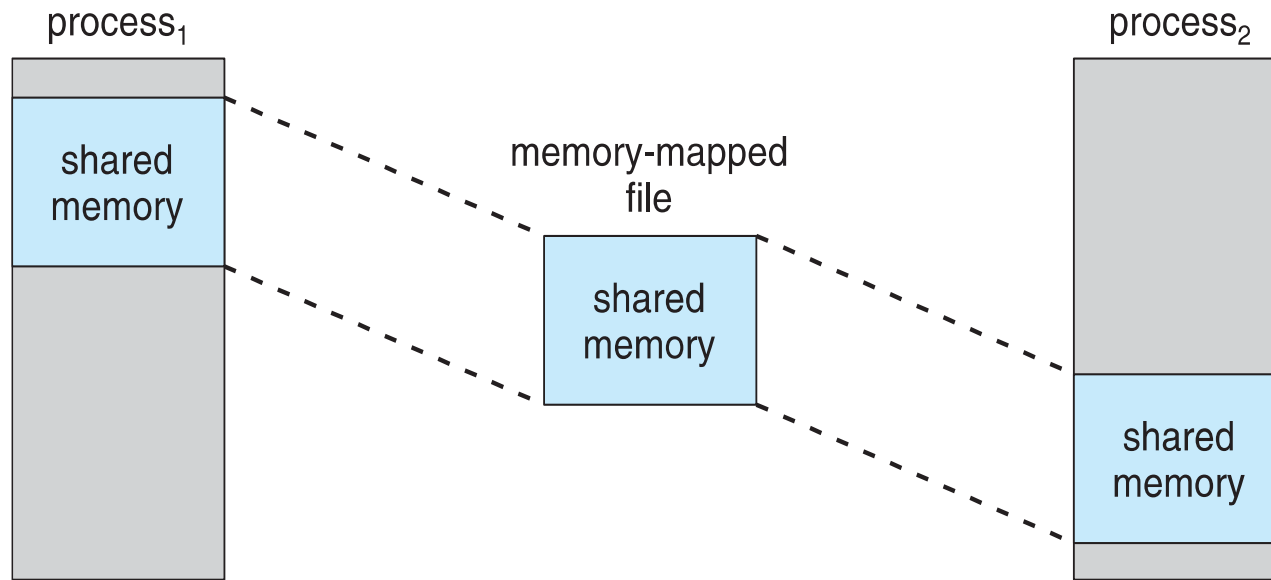


Memory Mapped Files





Shared Memory via Memory-Mapped I/O





Shared Memory in Windows API

First create a **file mapping** for file to be mapped

Then establish a view of the mapped file in process's virtual address space

Consider producer / consumer

Producer create shared-memory object using memory mapping features

Open file via `CreateFile()`, returning a `HANDLE`

Create mapping via `CreateFileMapping()` creating a **named shared-memory object**

Create view via `MapViewOfFile()`

Sample code in Textbook





Allocating Kernel Memory

Treated differently from user memory

Often allocated from a free-memory pool

Kernel requests memory for structures of varying sizes

Some kernel memory needs to be contiguous

- ▶ I.e. for device I/O





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

For example, assume 256KB chunk available, kernel requests 21KB

Split into A_L and A_R of 128KB each

- ▶ One further divided into B_L and B_R of 64KB
 - One further into C_L and C_R of 32KB each – one used to satisfy request

Advantage – quickly **coalesce** unused chunks into larger chunk

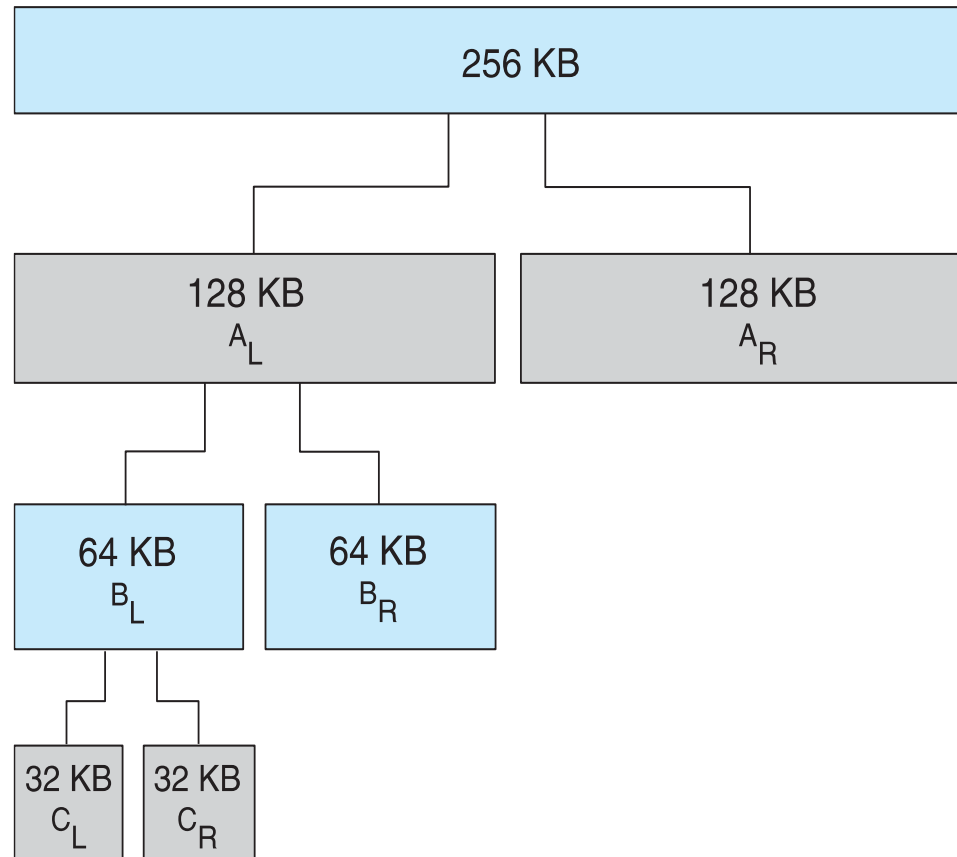
Disadvantage - fragmentation





Buddy System Allocator

physically contiguous pages





Buddy System Allocator

Buddy system addresses external fragmentation

How about internal fragmentation?

i.e. when allocating a 65 bytes structure, buddy system returns a 128 bytes block

- ▶ $128 - 65 = 63$ bytes are wasted

<http://utcc.utoronto.ca/~cks/space/blog/linux/KernelMemoryZones>

```
linux1:~  
(linux1:~) ysw% ls /proc/buddyinfo  
/proc/buddyinfo  
(linux1:~) ysw% cat /proc/buddyinfo  
Node 0, zone    DMA      0      0      1      0      2      1      1      0      1      1      3  
Node 0, zone    DMA32    1100    1384    709    358    161    89     56    55    46    27    72  
Node 0, zone    Normal    401      0      0      0      0      0      0      0      0      0      0  
Node 1, zone    Normal    1361    1936    755    487    493    413    381    247    142    166    758  
(linux1:~) ysw%   
      ↑      ↑      ↑      ↑  
    1k    2k    4k    8k
```





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

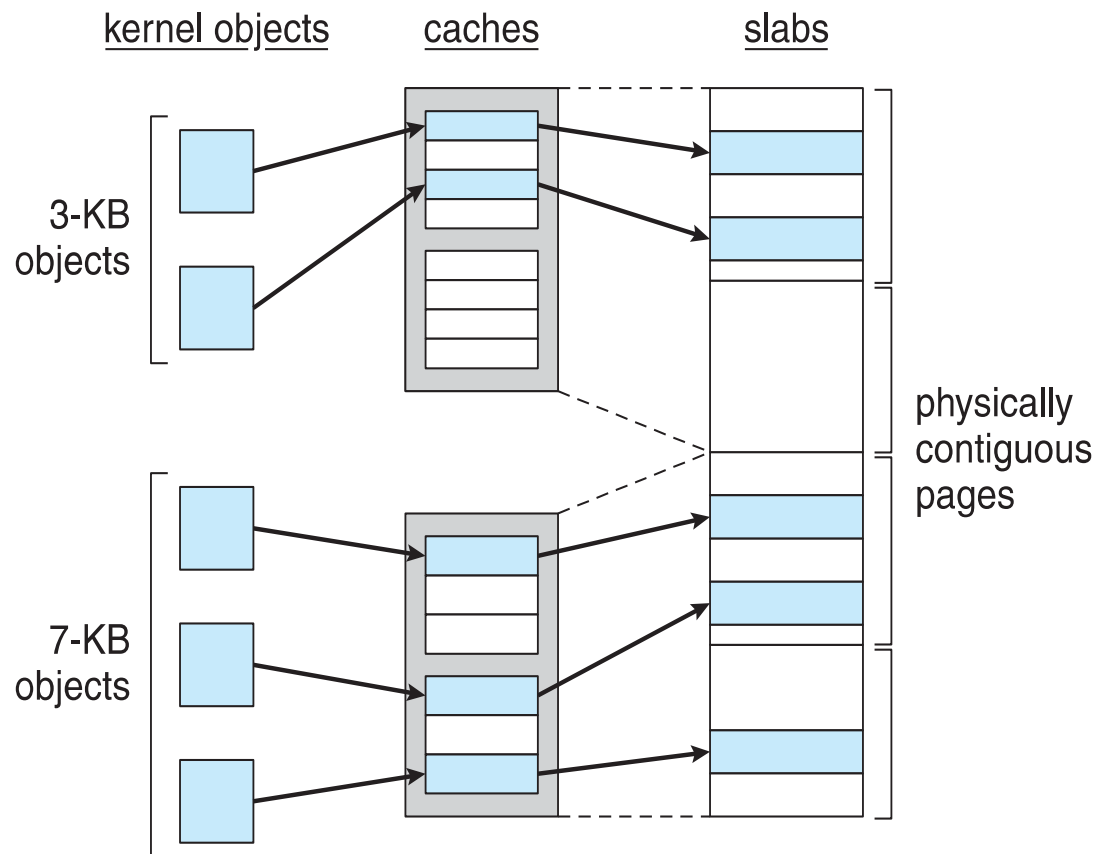
If no empty slabs, new slab allocated

Benefits include no fragmentation, fast memory request satisfaction





Slab Allocation





Slab Allocator in Linux

For example process descriptor is of type `struct task_struct`

Approx 1.7KB of memory

New task -> allocate new struct from cache

Will use existing free `struct task_struct`

Slab can be in three possible states

1. Full – all used
2. Empty – all free
3. Partial – mix of free and used

Upon request, slab allocator

1. Uses free struct in partial slab
2. If none, takes one from empty slab
3. If no empty slab, create new empty





Slab Allocator in Linux (Cont.)

Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes

Linux 2.2 had SLAB, now has both SLOB and SLUB allocators

SLOB for systems with limited memory

- ▶ Simple List of Blocks – maintains 3 list objects for small, medium, large objects

SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure





Slab Allocator

linux1:~

```
(linux1:~) ysw% slabtop
```

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

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



Other Considerations -- 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

Sometimes OS designers have a choice

Especially if running on custom-built CPU

Page size selection must take into consideration:

Fragmentation

Page table size

Resolution

I/O overhead

Number of page faults

Locality

TLB size and effectiveness

Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)

On average, growing over time





Other Issues – TLB Reach

TLB Reach - The amount of memory accessible from the TLB

$$\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$$

Ideally, the working set of each process is stored in the TLB

Otherwise there is a high degree of page faults

Increase the Page Size

This may lead to an increase in fragmentation as not all applications require a large page size

Provide Multiple Page Sizes

This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





Other Issues – Program Structure

Program structure

```
int[128,128] data;
```

Each row is stored in one page

Program 1

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

128 x 128 = 16,384 page faults

Program 2

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

128 page faults



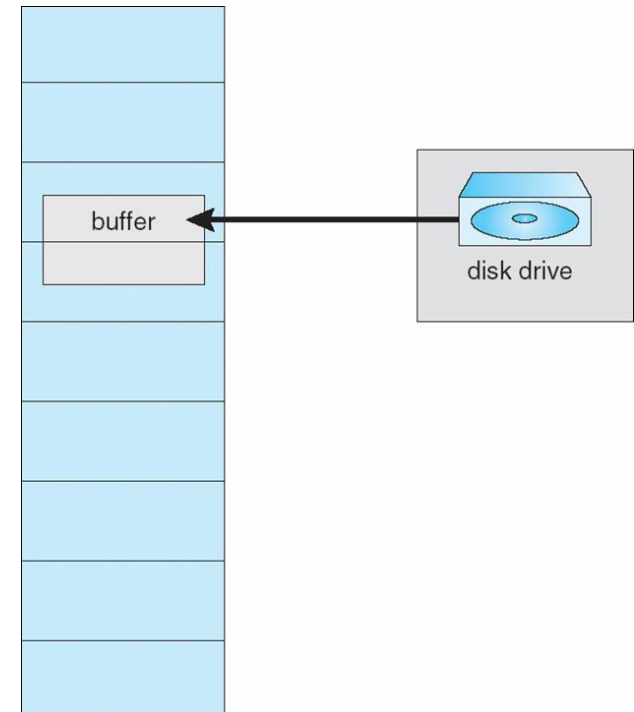


Other Issues – I/O interlock

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

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

Pinning of pages to lock into memory





Operating System Examples

Windows

Solaris





Windows

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**

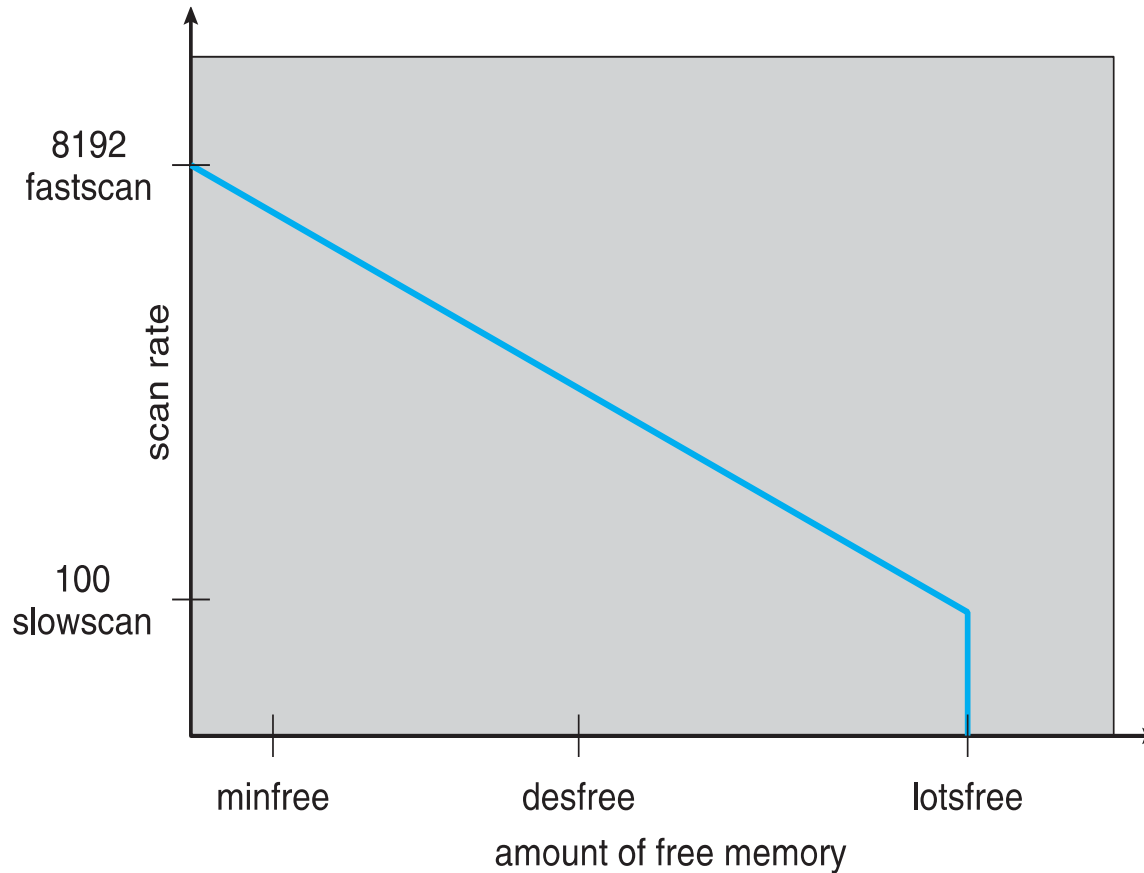
Pageout is called more frequently depending upon the amount of free memory available

Priority paging gives priority to process code pages





Solaris 2 Page Scanner



End of Chapter 9

