

Virtual Memory



Introduction

- Virtual memory != Virtual address

Virtual address is address issued by CPU's execution unit, later converted by MMU to physical address

Virtual memory is a memory management technique employed by OS (with hardware support, of course)

Unused parts of program

```
int a[4096][4096]
int f(int m[][4096]) {
    int i, j;
    for(i = 0; i < 1024; i++)
        m[0][i] = 200;
}
int main() {
    int i, j;
    for(i = 0; i < 1024; i++)
        a[1][i] = 200;
    if(random() == 10)
        f(a);
}
```

All parts of array `a` not accessed

Function `f()` may not be called

Some problems with schemes discussed so far

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

Error code, unusual routines, large data structures are rarely used

- So, entire program code, data not needed at same time
- So, consider ability to execute partially-loaded program

One Program no longer constrained by limits of physical memory

One Program and collection of programs could be larger than physical memory

What is virtual memory?

- 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

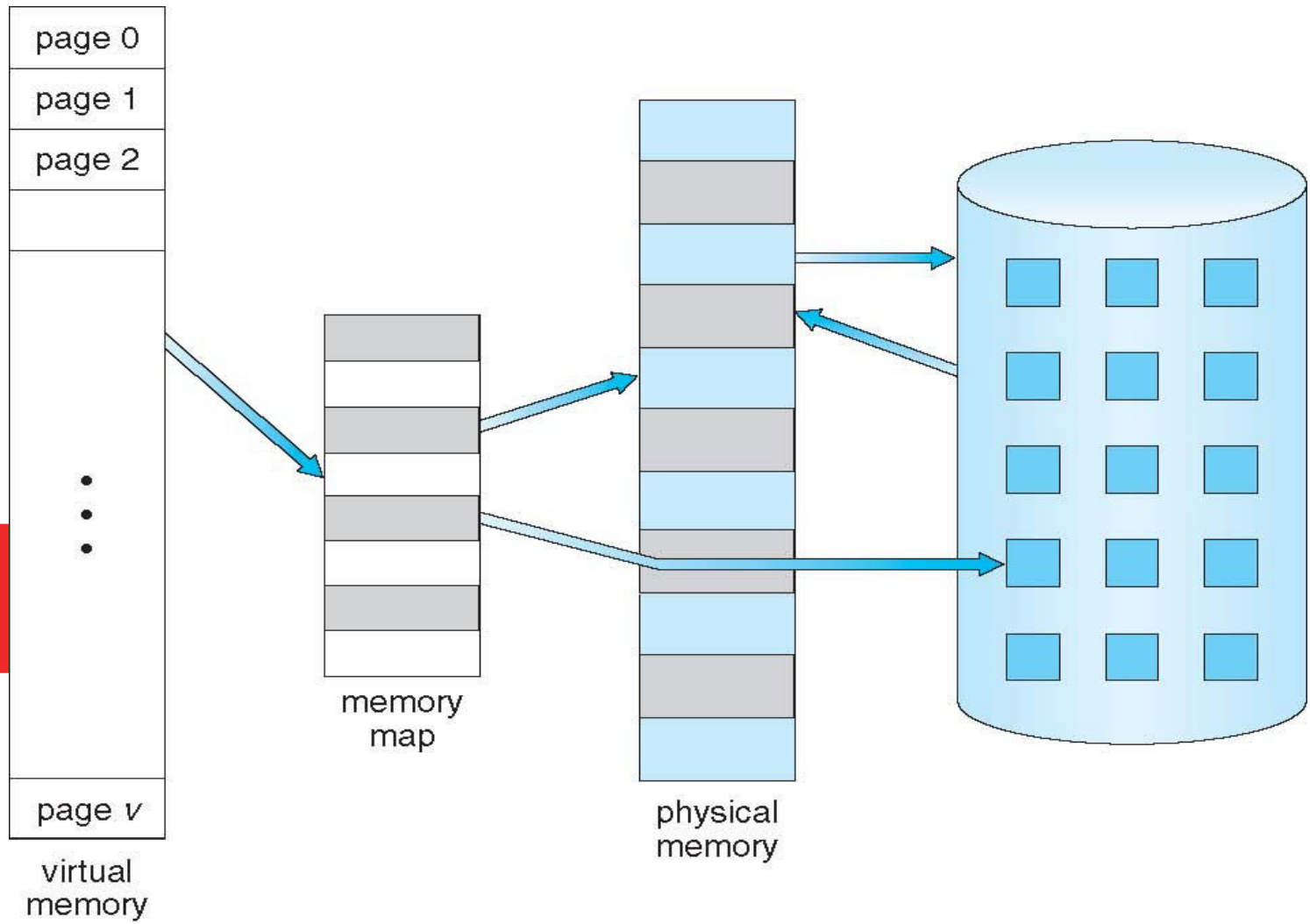
More programs running concurrently

Less I/O needed to load or swap processes

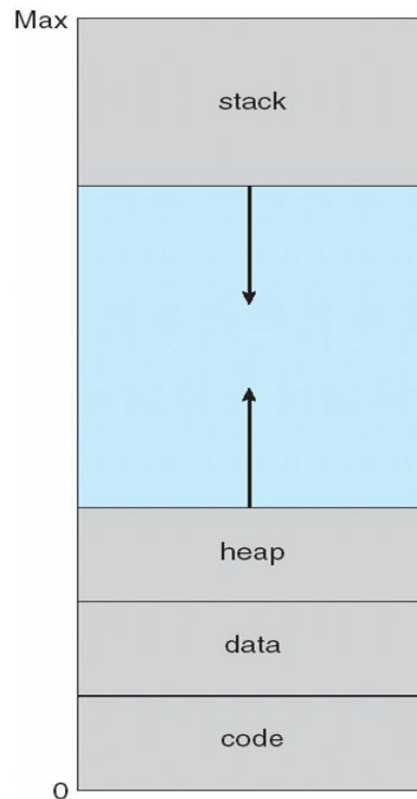
- Virtual memory can be implemented via:

Demand paging

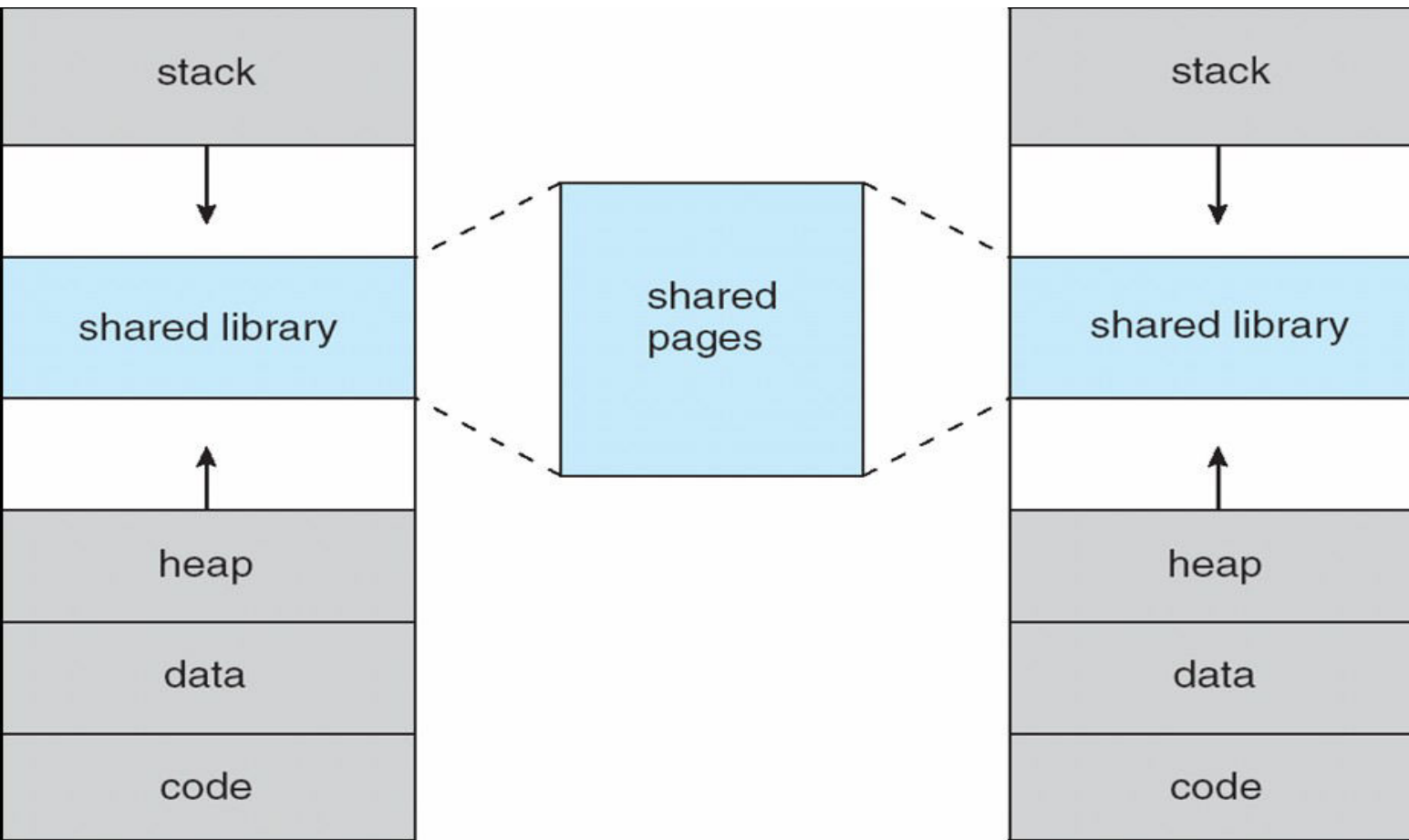
Demand segmentation



Virtual Address space



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



Shared pages Using Virtual Memory

System libraries shared via mapping into virtual address space

Shared memory by mapping same page-frames into page tables of involved processes

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



Demand Paging

Demand Paging

- Load a “page” to memory when it’s needed (on demand)

Less I/O needed, no unnecessary I/O

Less memory needed

Faster response

More users



-

Demand Paging

- Options:

Load entire process at load time : achieves little

Load some pages at load time: good

Load no pages at load time: pure demand paging



New meaning for valid/invalid bits in page table

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

- With each page table entry a valid-invalid bit is associated
 - v: in-memory – memory resident
 - i : not-in-memory or illegal
- During address translation, if valid-invalid bit in page table entry is I : **raises trap called page fault**

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

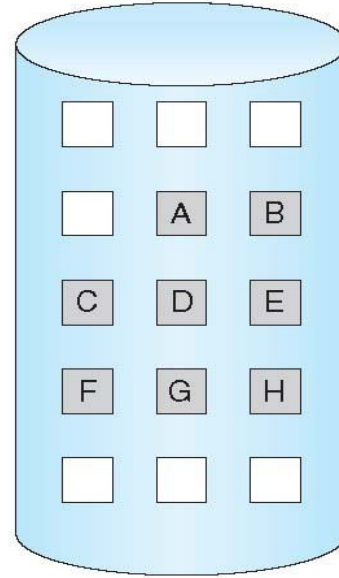
logical memory

	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

Table

With

Some pages

Not

In memory



Page fault

Page fault

- Page fault is a hardware interrupt
- It occurs when the page table entry corresponding to current memory access is “i”
- All actions that a kernel takes on a hardware interrupt are taken!

Change of stack to kernel stack

Saving the context of process

Switching to kernel code

On a Page fault

1) Operating system looks at another data structure (table), most likely in PCB itself, to decide:

If it's Invalid reference -> abort the process (segfault)

Just not in memory -> Need to get the page in memory

2) Get empty frame (this may be complicated!)

3) Swap page into frame via scheduled disk/IO operation

4) Reset tables to indicate page now in memory.

5) Set validation bit = v

6) Restart the instruction that caused the page fault

Additional problems

- 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
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

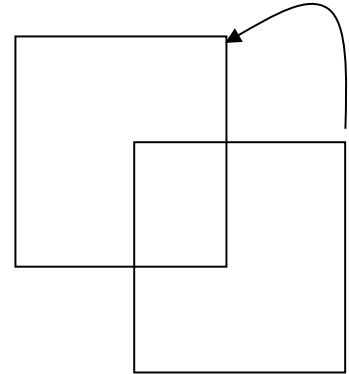
- A critical Problem
- Consider an instruction that could access several different locations

```
movarray 0x100, 0x200, 20
```

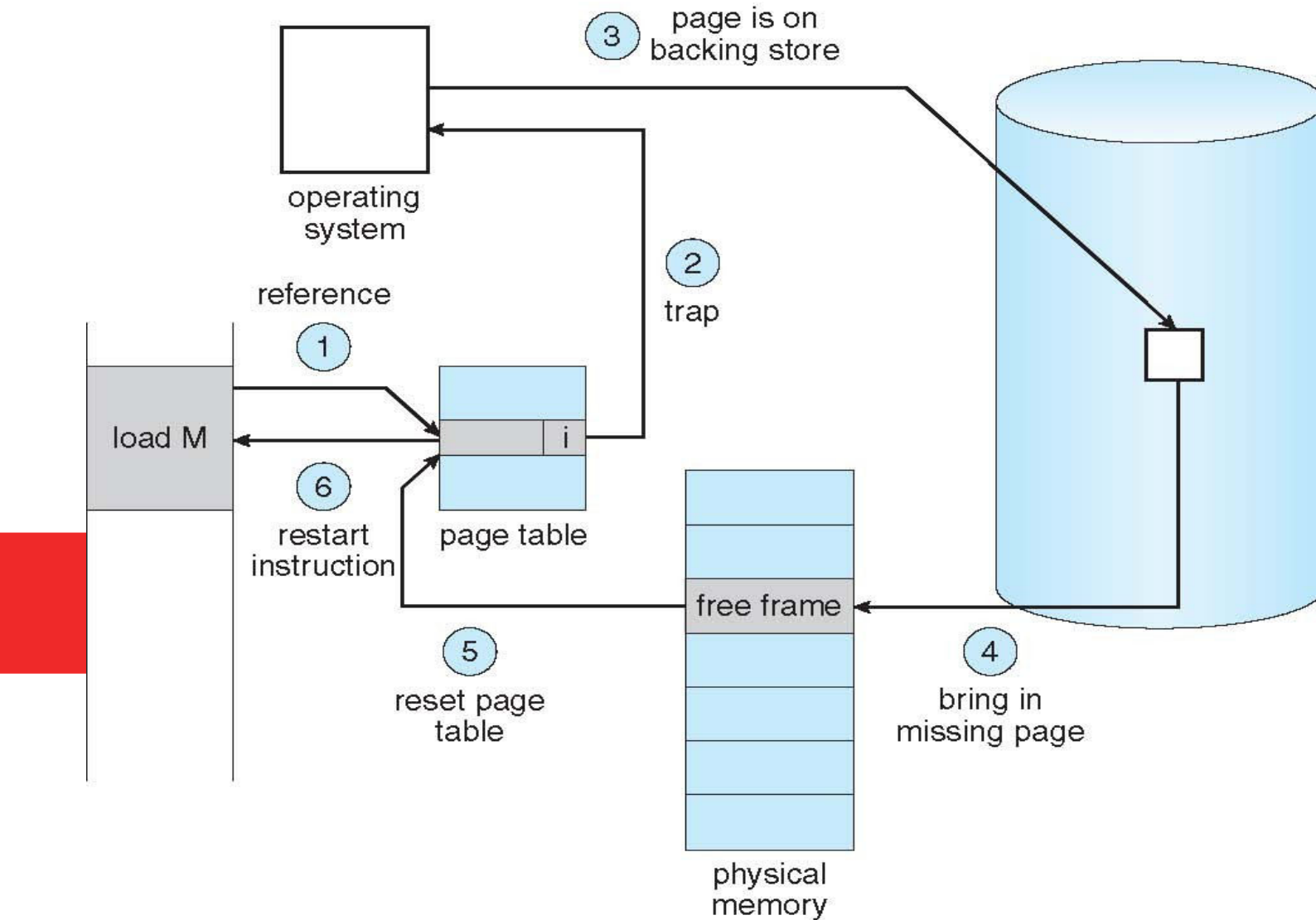
copy 20 bytes from address 0x100 to address 0x200

```
movarray 0x100, 0x110, 20
```

what to do in this case?



Handling A Page Fault



Page fault handling

1) Trap to the operating system

2) Default trap handling():

Save the process registers and process state

Determine that the interrupt was a page fault. Run page fault handler.

3) Page fault handler(): Check that the page reference was legal and determine the location of the page on the disk. If illegal, terminate process.

4) Find a free frame. Issue a read from the disk to a free frame:

Process waits in a queue for disk read. Meanwhile many processes may get scheduled.

Disk DMA hardware transfers data to the free frame and raises interrupt in end

Page fault handling

6) (as said on last slide) While waiting, allocate the CPU to some other process

7) (as said on last slide) Receive an interrupt from the disk I/O subsystem (I/O completed)

8) Default interrupt handling():

- Save the registers and process state for the other user

- Determine that the interrupt was from the disk

9) Disk interrupt handler():

- Figure out that the interrupt was for our waiting process

- Make the process runnable

10) Wait for the CPU to be allocated to this process again

- Kernel restores the page table of the process, marks entry as "v"

- Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of demand paging

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

if $p = 0$ no page faults

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

Effective (memory) Access Time (EAT)

EAT = $(1 - p) * \text{memory access time} +$

$p * (\text{page fault overhead // Kernel code execution time}$

$+ \text{swap page out // time to write an occupied frame to disk}$

$+ \text{swap page in // time to read data from disk into free frame}$

$+ \text{restart overhead) // time to reset process context, restart it}$

Performance of demand paging

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 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

An optimization: Copy on write

The problem with `fork()` and `exec()`. Consider the case of a shell

```
scanf("%s", cmd);  
if(strcmp(cmd, "exit") == 0)  
    return 0;  
pid = fork(); // A->B  
if(pid == 0) {  
    ret = execl(cmd, cmd, NULL);  
    if(ret == -1) {  
        perror("execution failed");  
        exit(errno);  
    }  
} else {  
    wait(0);  
}
```

- During `fork()`
 - Pages of parent were duplicated
 - Equal amount of page frames were allocated
 - Page table for child differed from parent (as it has another set of frames)
- In `exec()`
 - The page frames of child were taken away and new frames were allocated
 - Child's page table was rebuilt!
- Waste of time during `fork()` if the `exec()` was to be called immediately

An optimization: Copy on write

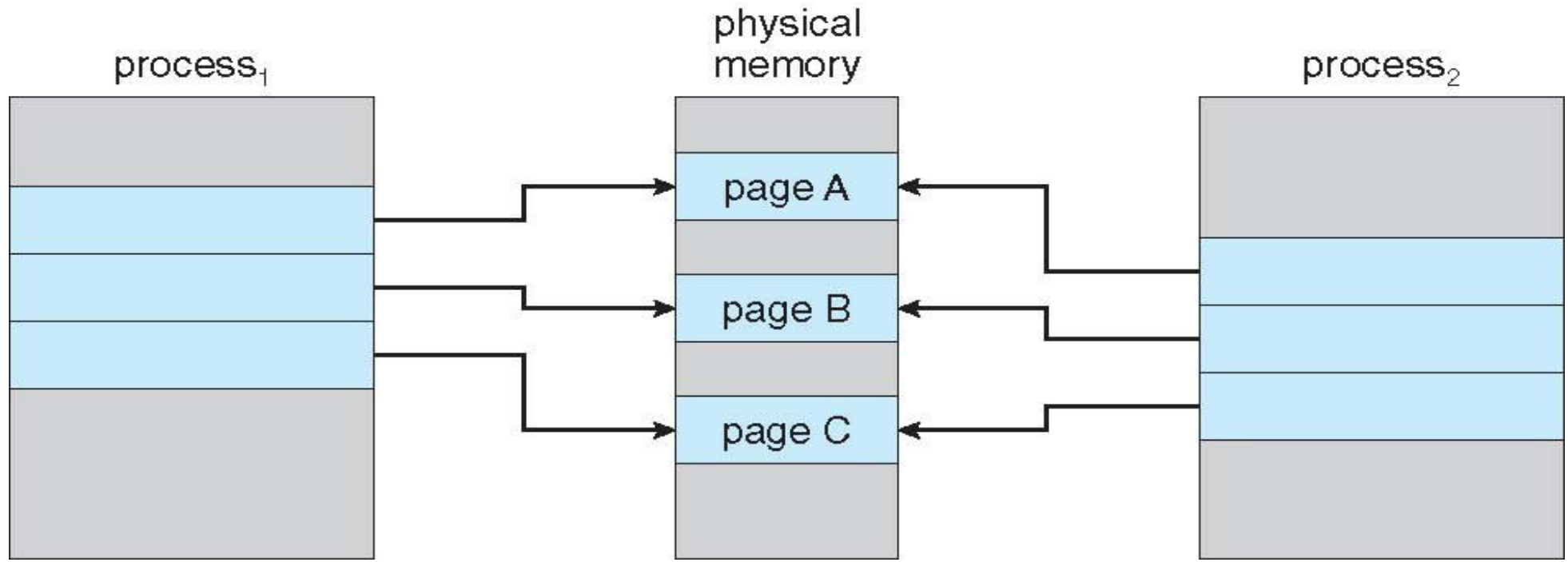
- Copy-on-Write (COW) allows both parent and child processes to initially share the same pages in memory

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

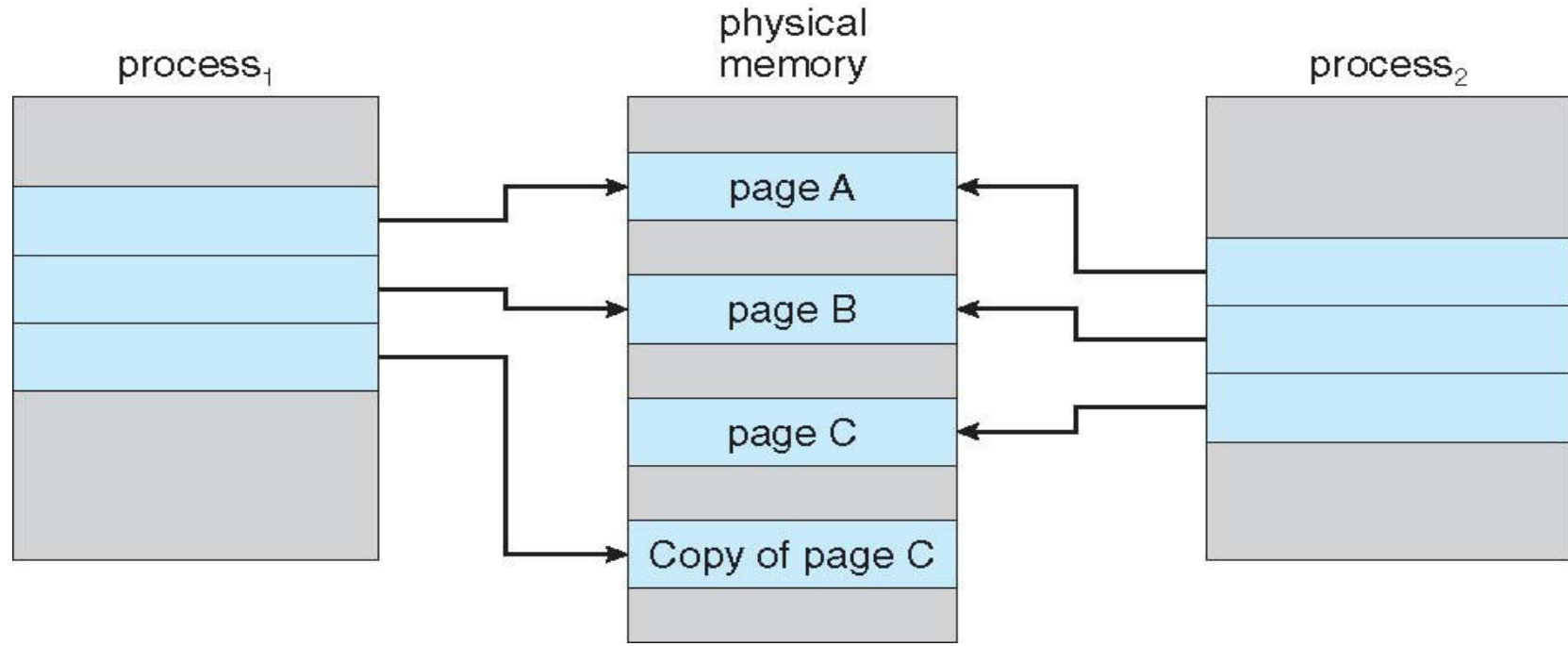
- COW allows more efficient process creation as only modified pages are copied
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent

Designed to have child call `exec()`

Very efficient



Before Process 1 Modifies Page C



After Process 1 Modifies Page C

Challenges and improvements in implementation

- Choice of backing store

For stack, heap pages: on swap partition

For code, shared library? : swap partition or the actual executable file on the file-system?

If the choice is file-system for code, then the page-fault handler needs to call functions related to file-system

- Is the page table itself pagable?

If no, good

If Yes, then there can be page faults in accessing the page tables themselves! More complicated!

- Is the kernel code pagable?

If no, good

If yes, life is very complicated ! Page fault in running kernel code, interrupt handlers, system calls, etc.



Page replacement

Review

- Concept of virtual memory, demand paging.
- Page fault
- Performance degradation due to page fault: Need to reduce #page faults to a minimum
- Page fault handling process, broad steps: (1) Trap (2) Locate on disk (3) find free frame (4) schedule disk I/O (5) update page table (6) resume
- More on (3) today

List of free frames

- Kernel needs to maintain a list of free frames
- At the time of loading the kernel, the list is created
- Frames are used for allocating memory to a process

But may also be used for managing kernel's own data structures also

- More processes --> more demand for frames

What if no free frame found on page fault?

- Page frames in use depends on “Degree of multiprogramming”

More multiprogramming -> overallocation of frames

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

How much to allocate to each process? How many processes to allow?

- Page replacement – find some page(frame) in memory, but not really in use, page it out

Questions : terminate process? Page out process? replace the page?

For performance, need an algorithm which will result in minimum number of page faults

- Bad choices may result in same page being brought into memory several times

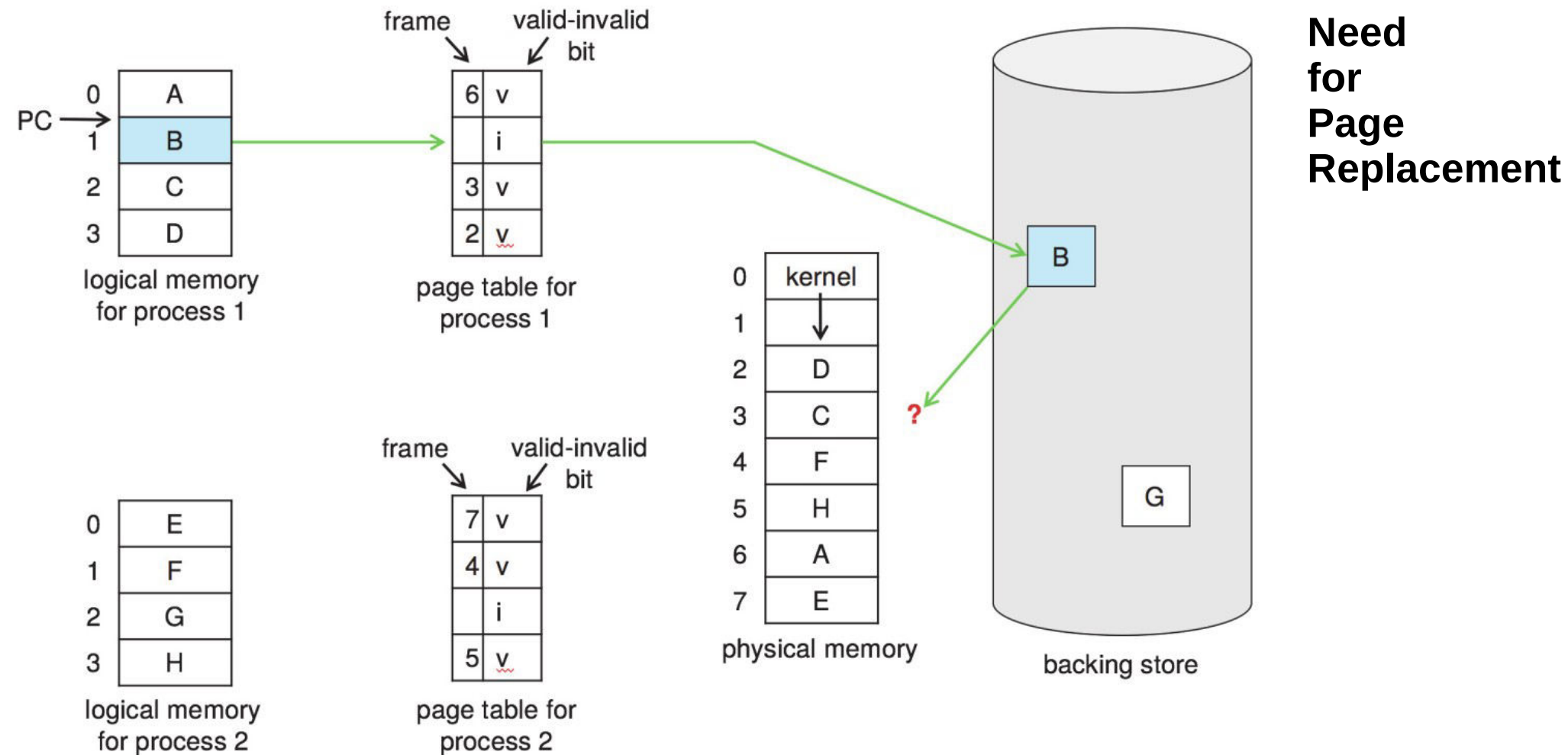


Figure 10.9 Need for page replacement.

Page replacement

- Strategies for performance

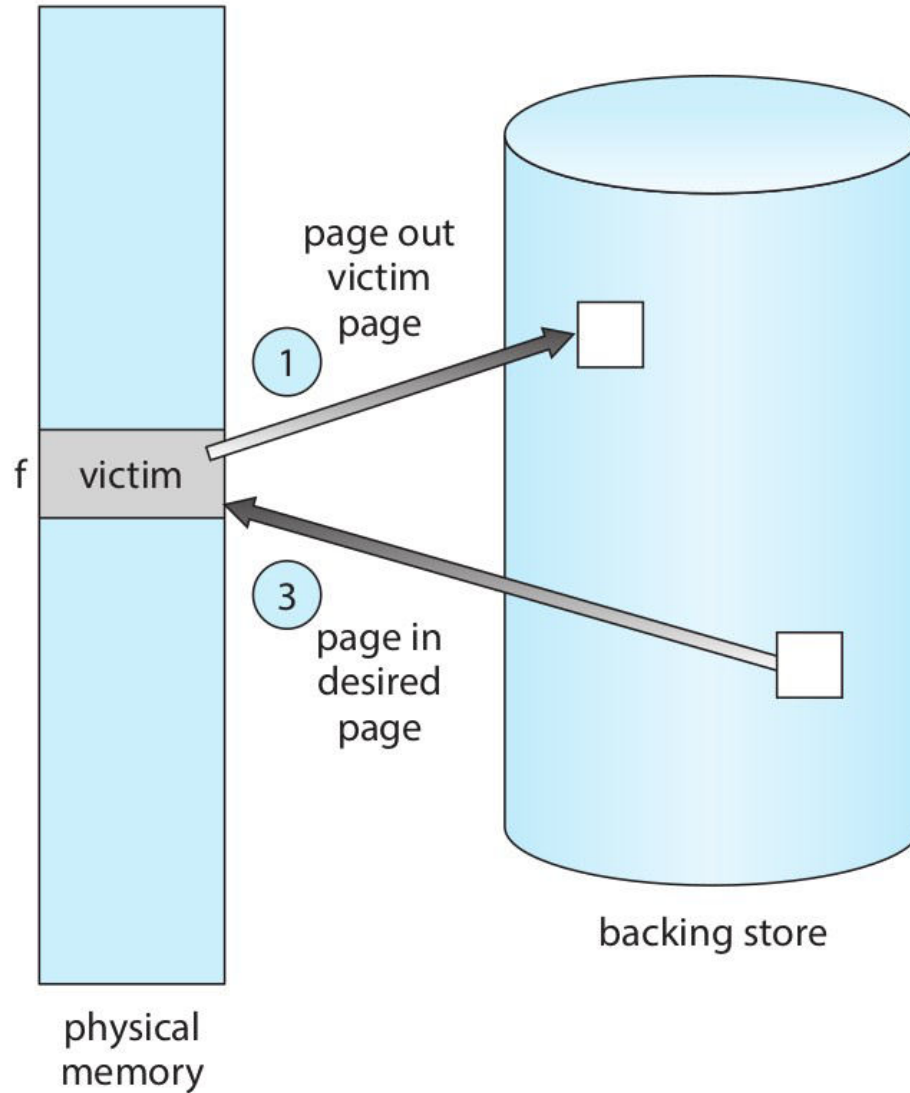
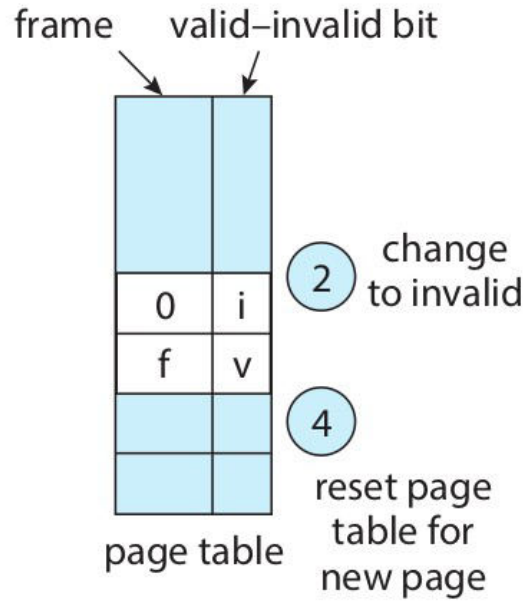
Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

Use modify (dirty) bit in page table. To reduce overhead of page transfers – only modified pages are written to disk. If page is not modified, just reuse it (a copy is already there in backing store)

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 free frame; update the page table of process and global frame table/list
- 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

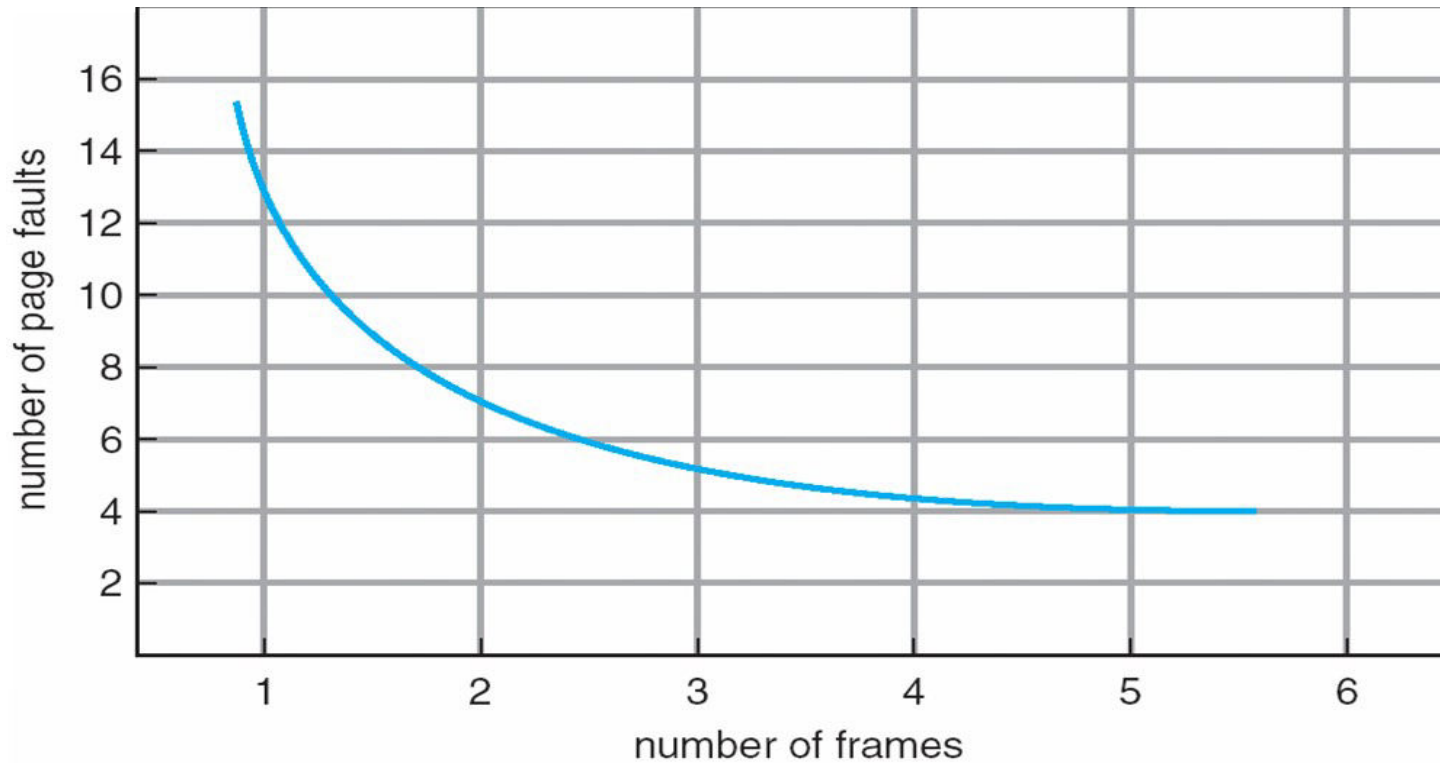


Two problems to solve

- 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

Evaluating algorithm: Reference string

- 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
- In all our examples, the reference string is
7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1



**An
Expectation**

**More page
Frames
Means less
faults**

FIFO Algorithm

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																
	0	0	0																
		1	1																


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

0	0																		
1	1																		
3	2																		

7	7	7																	
1	0	0																	
2	2	1																	

page frames

FIFO Algorithm

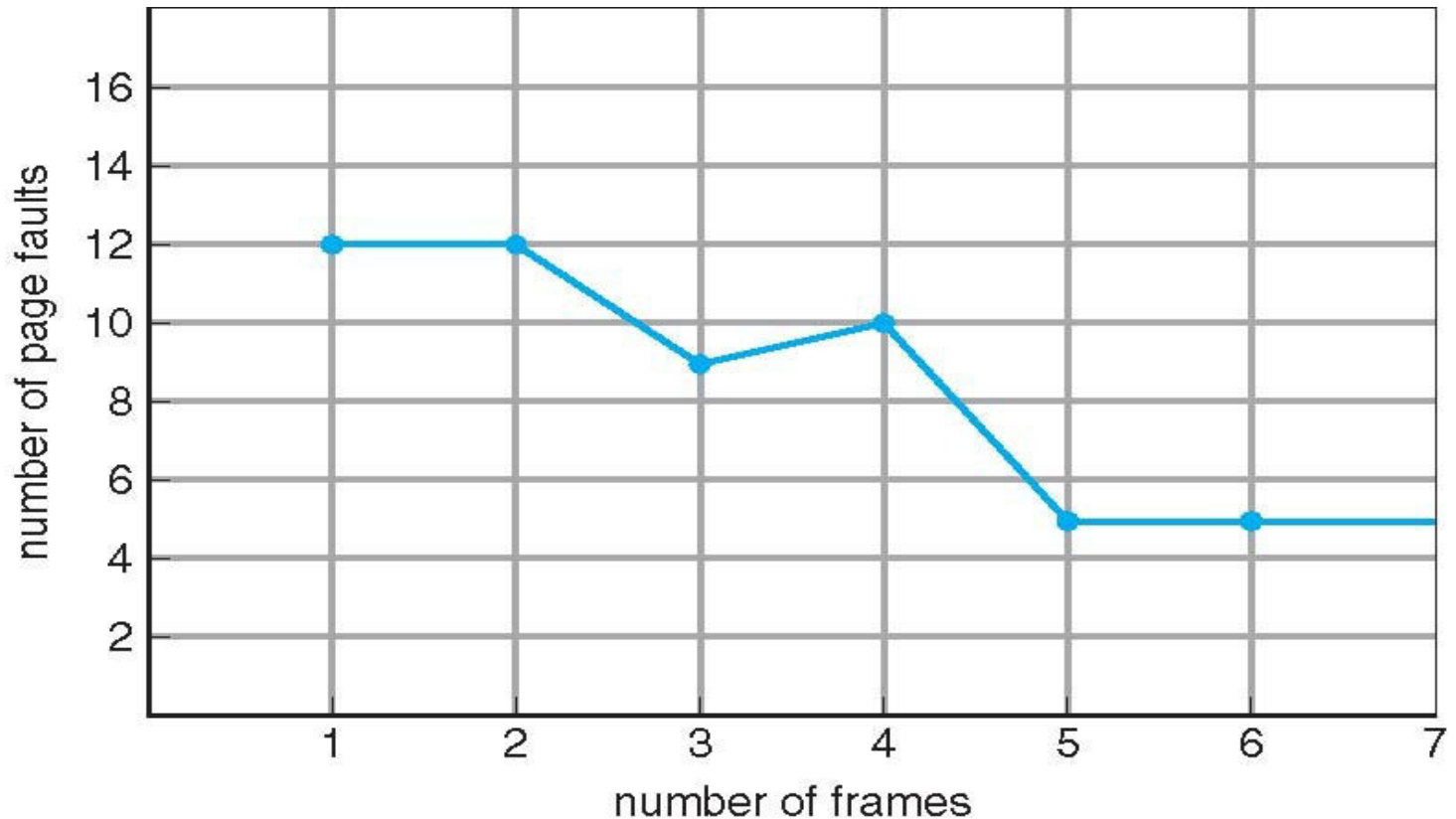


1	7	2	4	0	7
2	0	3	2	1	0
3	1	0	3	2	1

15 page faults

- 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)
- **Belady's Anomaly**
- Adding more frames can cause more page faults!
- Can vary by reference string:
consider 1,2,3,4,1,2,5,1,2,3,4,5

FIFO Algorithm: Balady's anomaly



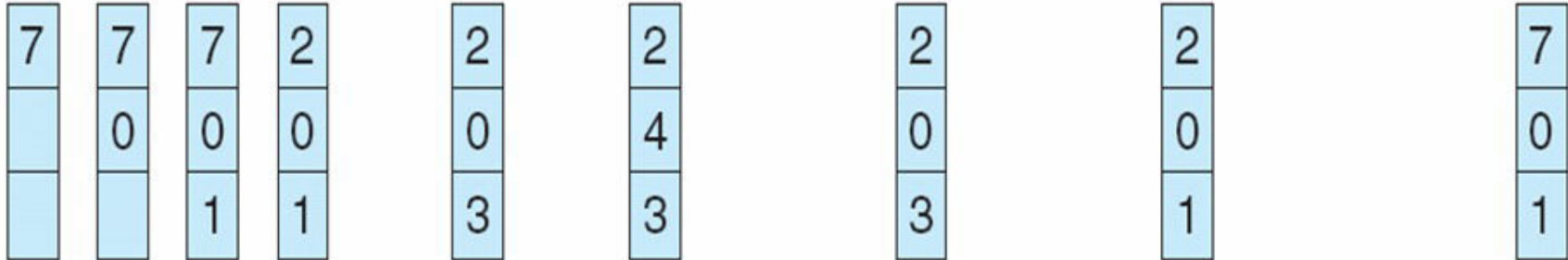
Optimal Algorithm

- Replace page that will not be used for longest period of time
 - 9 is optimal #replacements for the example on the next slide
- How do you know this?
 - Can't read the future
- 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



page frames

Least Recently Used: an approximation of optimal

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

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

12 faults – better than FIFO but worse than OPT

Generally good algorithm and frequently used

But how to implement?

LRU: Counter implementation

- **Counter implementation**

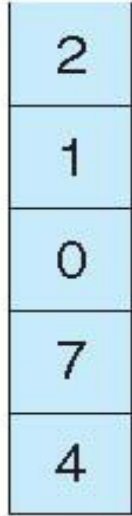
- Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
- When a page needs to be changed, look at the counters to find smallest value
- Search through table needed

LRU: Stack implementation

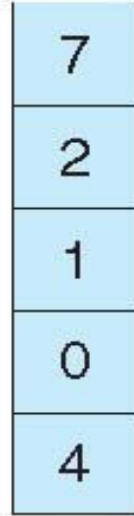
- Keep a stack of page numbers in a double link form:
- Page referenced: move it to the top
 - requires 6 pointers to be changed and
 - each update more expensive
 - But no need of a search for replacement

reference string

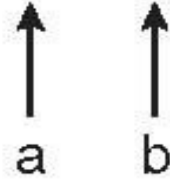
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



**Use Of A
Stack to
Record The**

**Most
Recent
Page
References**

Stack algorithms

- An algorithm for which it can be shown that the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with $n + 1$ frames
- Do not suffer from Balady's anomaly
- For example: Optimal, LRU

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

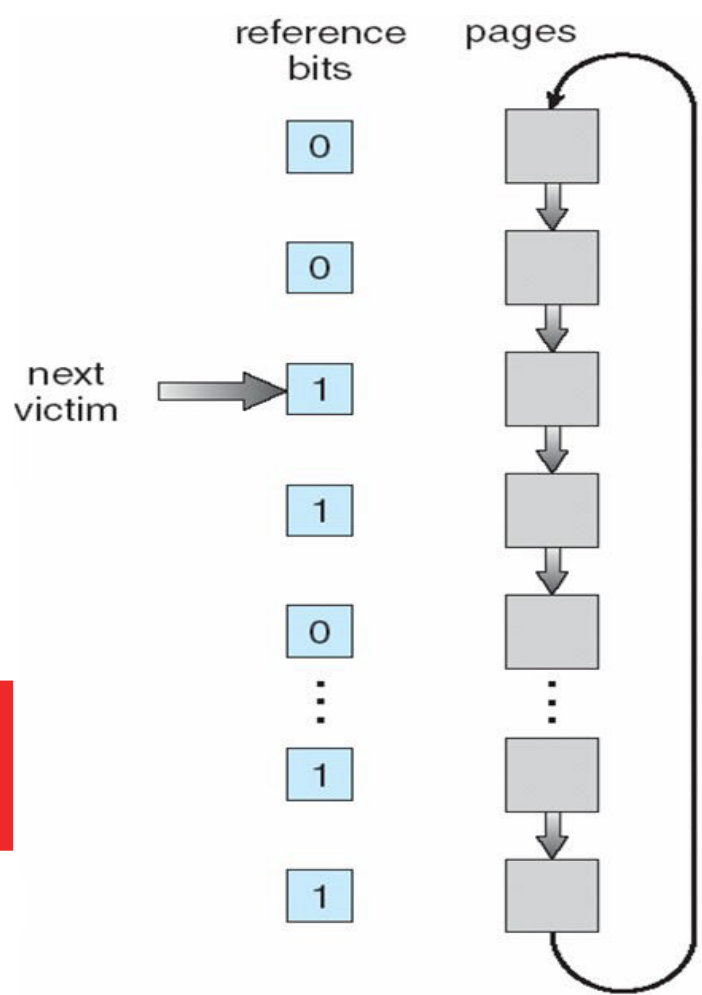
LRU: Approximation algorithms

- **Second-chance algorithm**

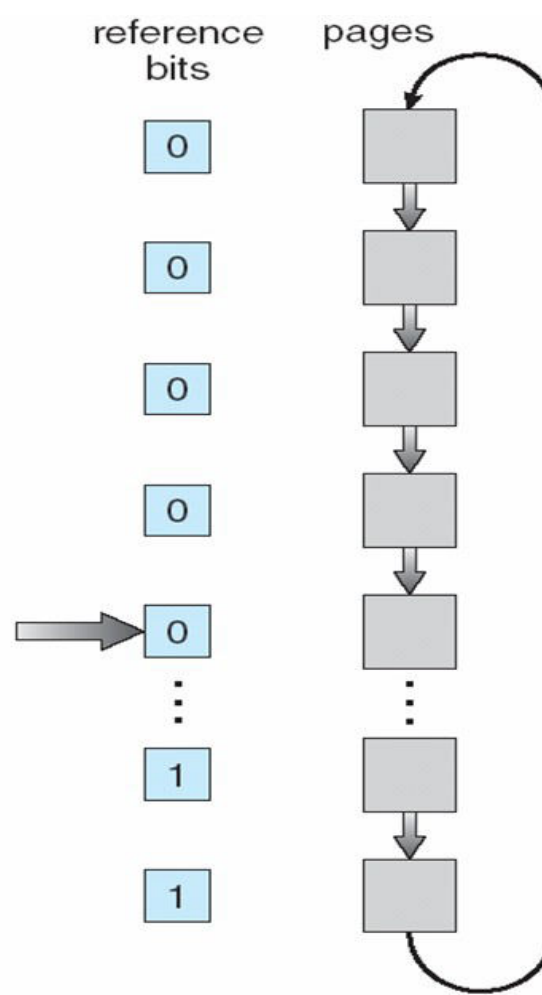
- FIFO + hardware-provided reference bit. If bit is 0 select, if bit is 1, then set it to 0 and move to next one.

- **An implementation of second-chance: 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



(a)



(b)

Second-Chance (clock) Page-Replacement Algorithm

Counting Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **LFU Algorithm**: replaces page with smallest count
- **MFU Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

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 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

Major and Minor page faults

- Most modern OS refer to these two types
- Major fault
 - Fault + page not in memory
- Minor fault
 - Fault, but page is in memory
 - For example shared memory pages; second instance of fork(), page already on free-frame list,
- On Linux run
 - `$ ps -eo min_flt,maj_flt,cmd`

Special rules for special applications

- All of earlier algorithms have OS guessing about future page access
- But some applications have better knowledge – e.g. 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 of frames

- 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

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


$$s_1 = 10$$

$$s_2 = 127$$

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

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

Priority Allocation of frames

- 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



Virtual Memory – Remaining topics

Agenda

- Problem of Thrashing and possible solutions
 - Mmap(), Memory mapped files
 - Kernel Memory Management
 - Other Considerations
- 

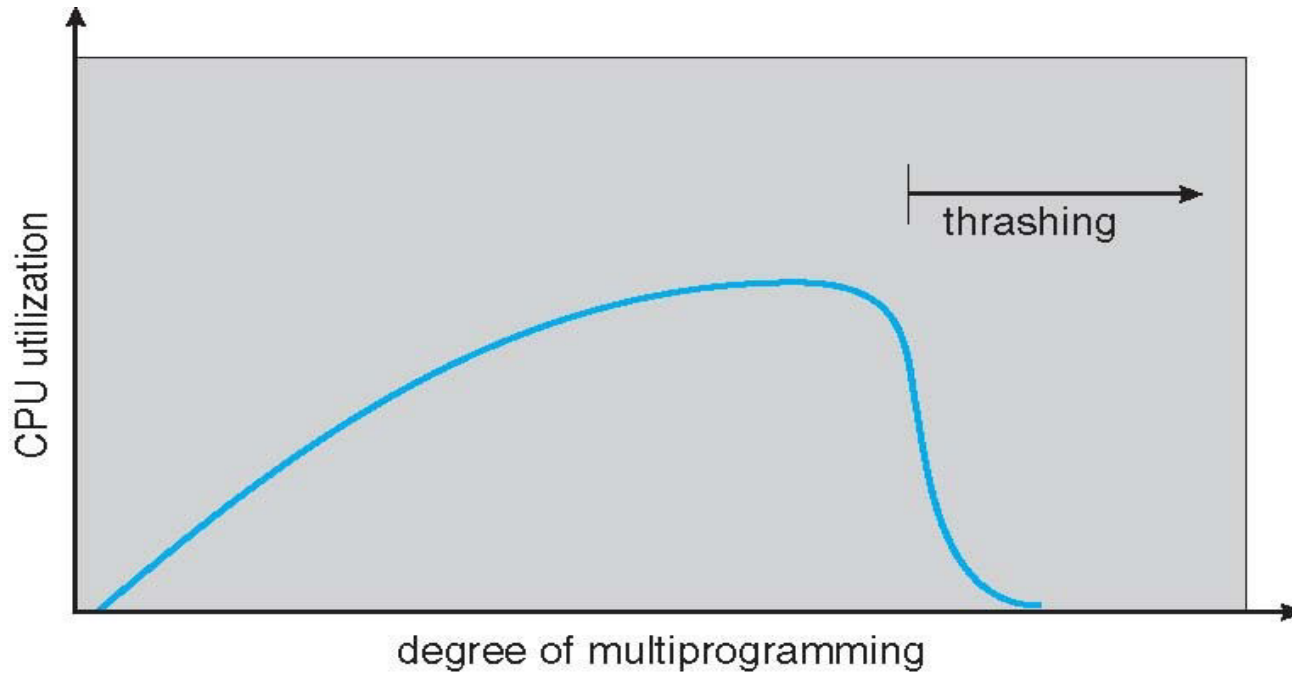


Thrashing

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 : a process is busy swapping pages in and out

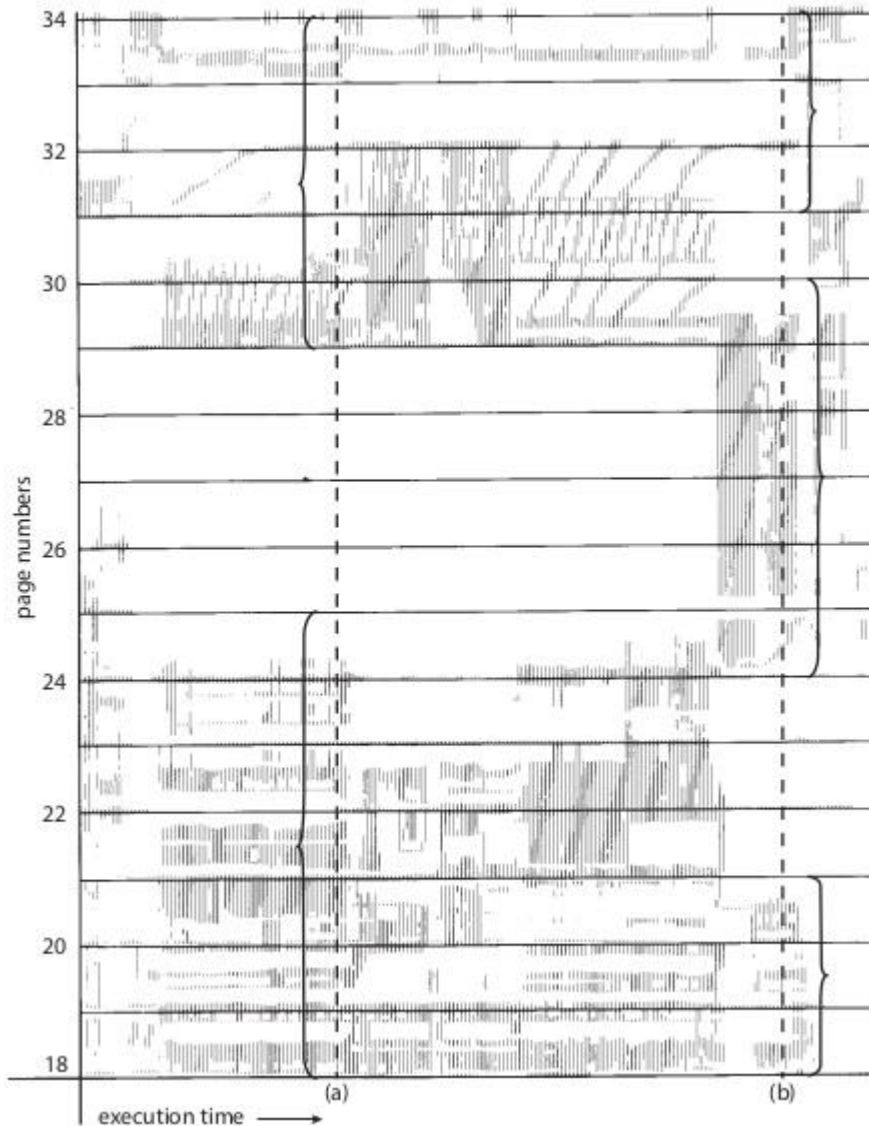
Thrashing



Demand paging and thrashing

- Why does demand paging work?
 - Locality model
 - Process migrates from one locality to another
 - Localities may overlap
- Why does thrashing occur?
 - size of locality $>$ total memory size
 - Limit effects by using local or priority page replacement

Locality In A Memory- Reference Pattern



Working set model

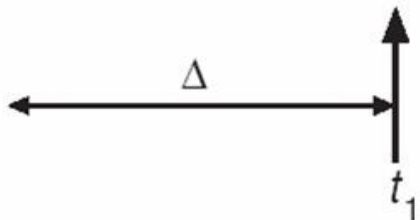
- $\Delta \equiv$ working-set window \equiv a fixed number of page references
 - Example: 10,000 instructions
- Working Set Size, WSS_i (working set of Process P_i) =
 - total number of pages referenced in the most recent Δ (varies in time)
 - if Δ too small will not encompass entire locality
 - if Δ too large will encompass several localities
 - if $\Delta = \infty \Rightarrow$ will encompass entire program

Working set model

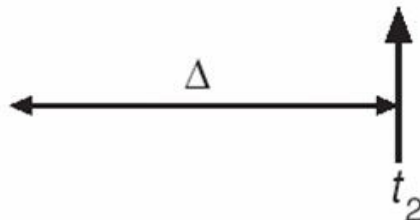
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
- if $D > m$ (total available frames) \Rightarrow Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

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 (to memory) and sets the values of all reference bits to 0

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

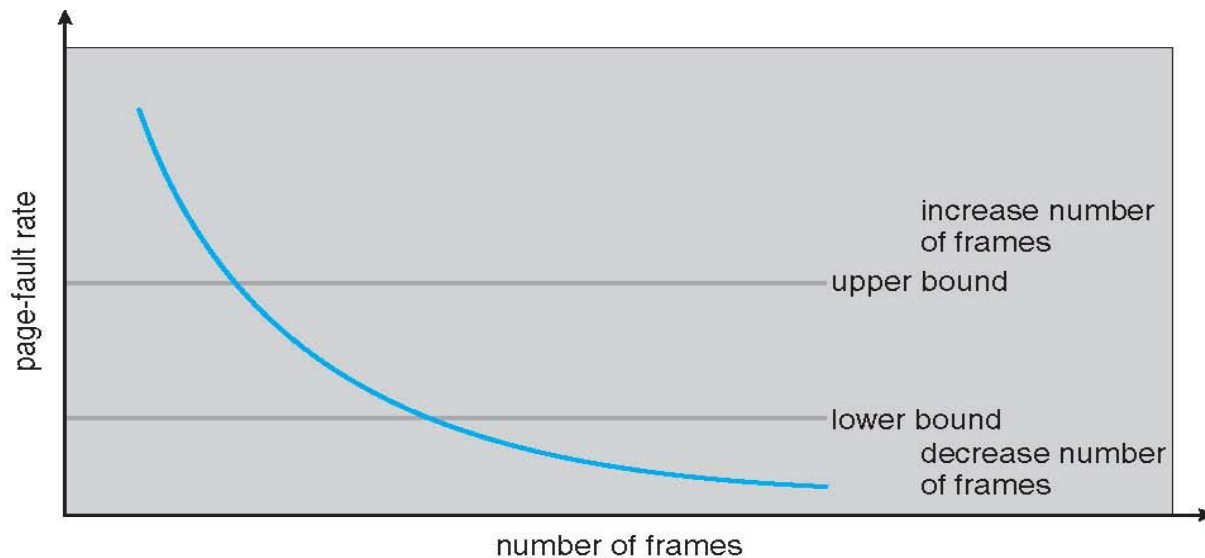
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Page fault frequency

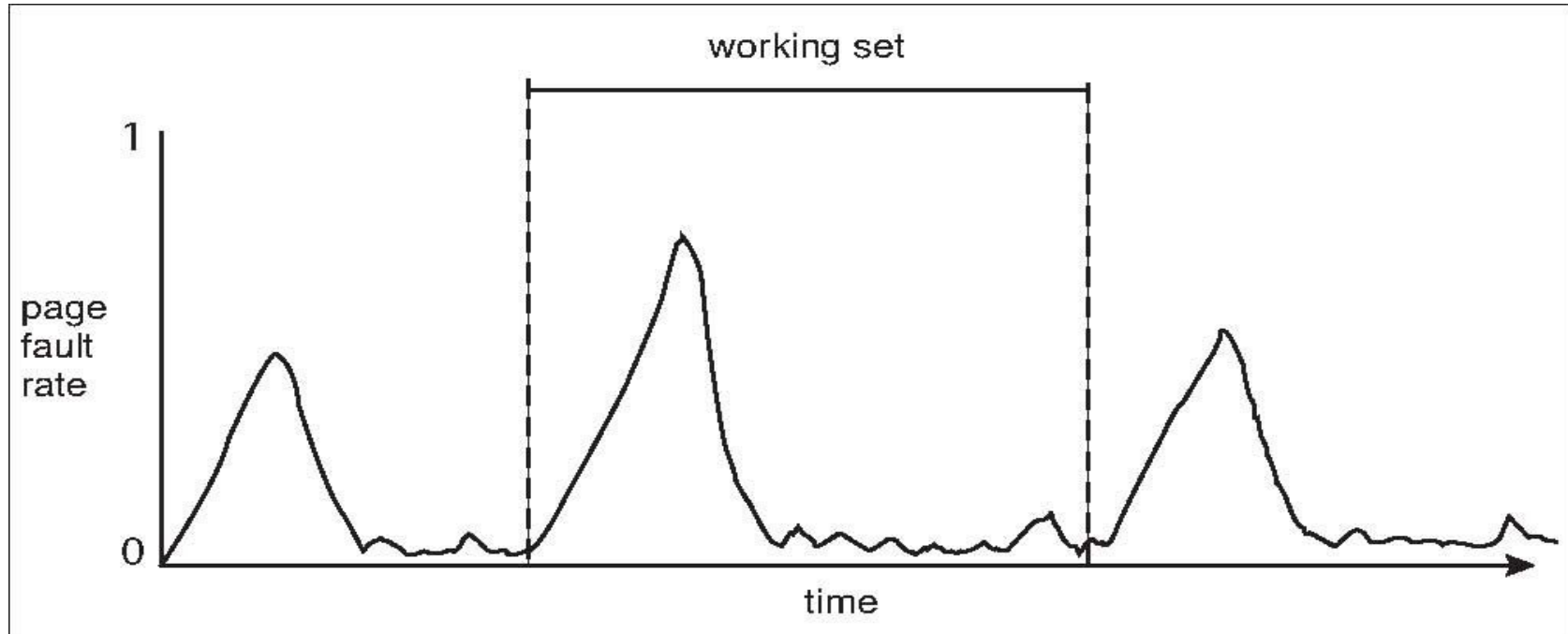
- More direct approach than WSS
- Establish “acceptable” page-fault frequency rate and use local replacement policy

If actual rate too low, process loses frame

If actual rate too high, process gains frame



Working Sets and Page Fault Rates





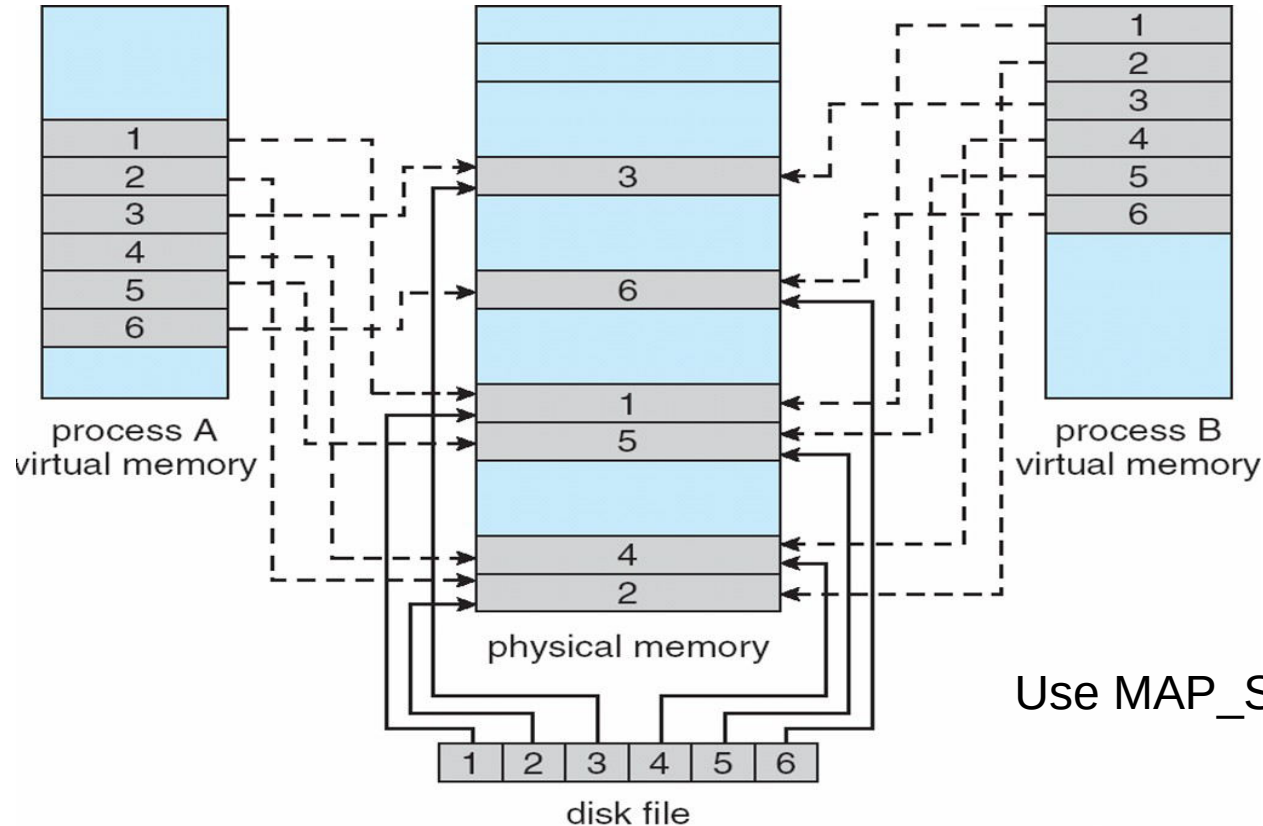
Memory Mapped Files

Memory-Mapped Files

- First, let's see a demo of using `mmap()`



Memory-Mapped Files



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 Files

- 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)



Allocating Kernel Memory

Allocating kernel memory

- Treated differently from user memory
- Often allocated from a free-memory pool

Kernel requests memory for structures of varying sizes

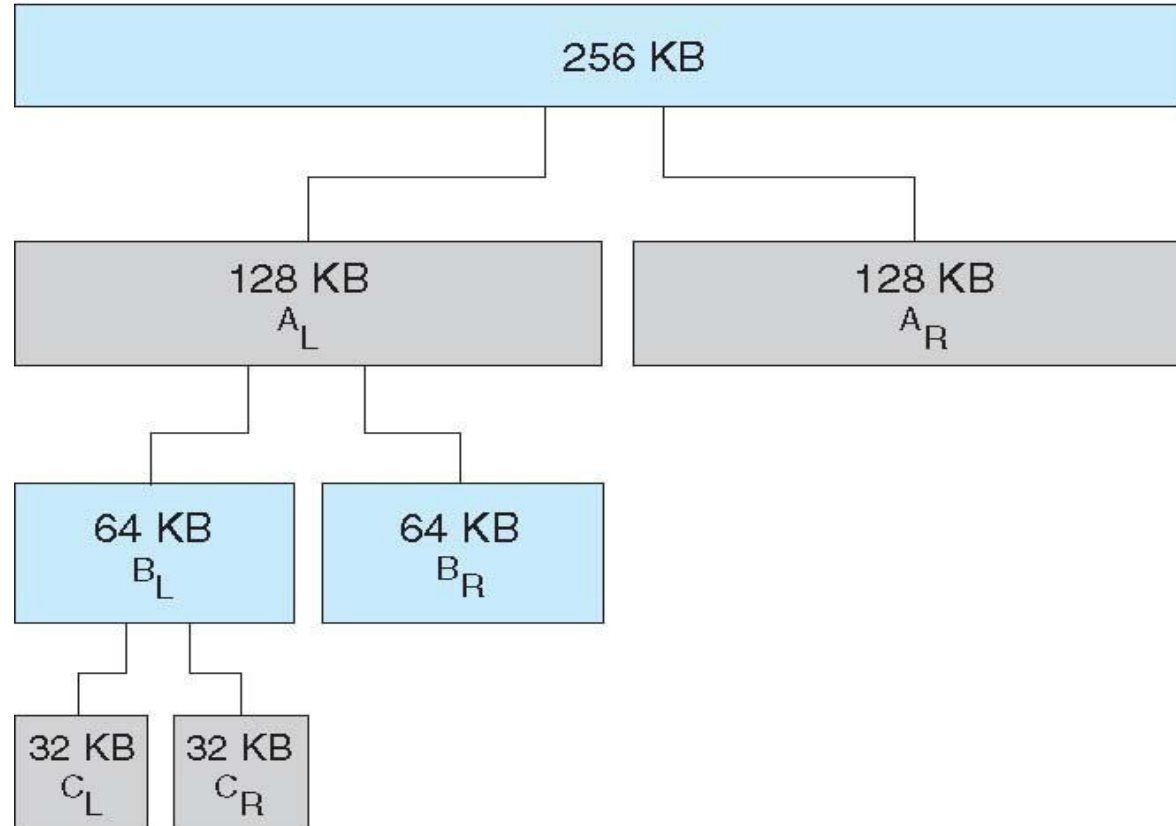
Some kernel memory needs to be contiguous

I.e. for device I/O

- 
-

Buddy Allocator

physically contiguous pages



Buddy Allocator

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using power-of-2 allocator

Satisfies requests in units sized as power of 2

Request rounded up to next highest power of 2

When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2

Continue until appropriate sized chunk available

Buddy Allocator

- Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB

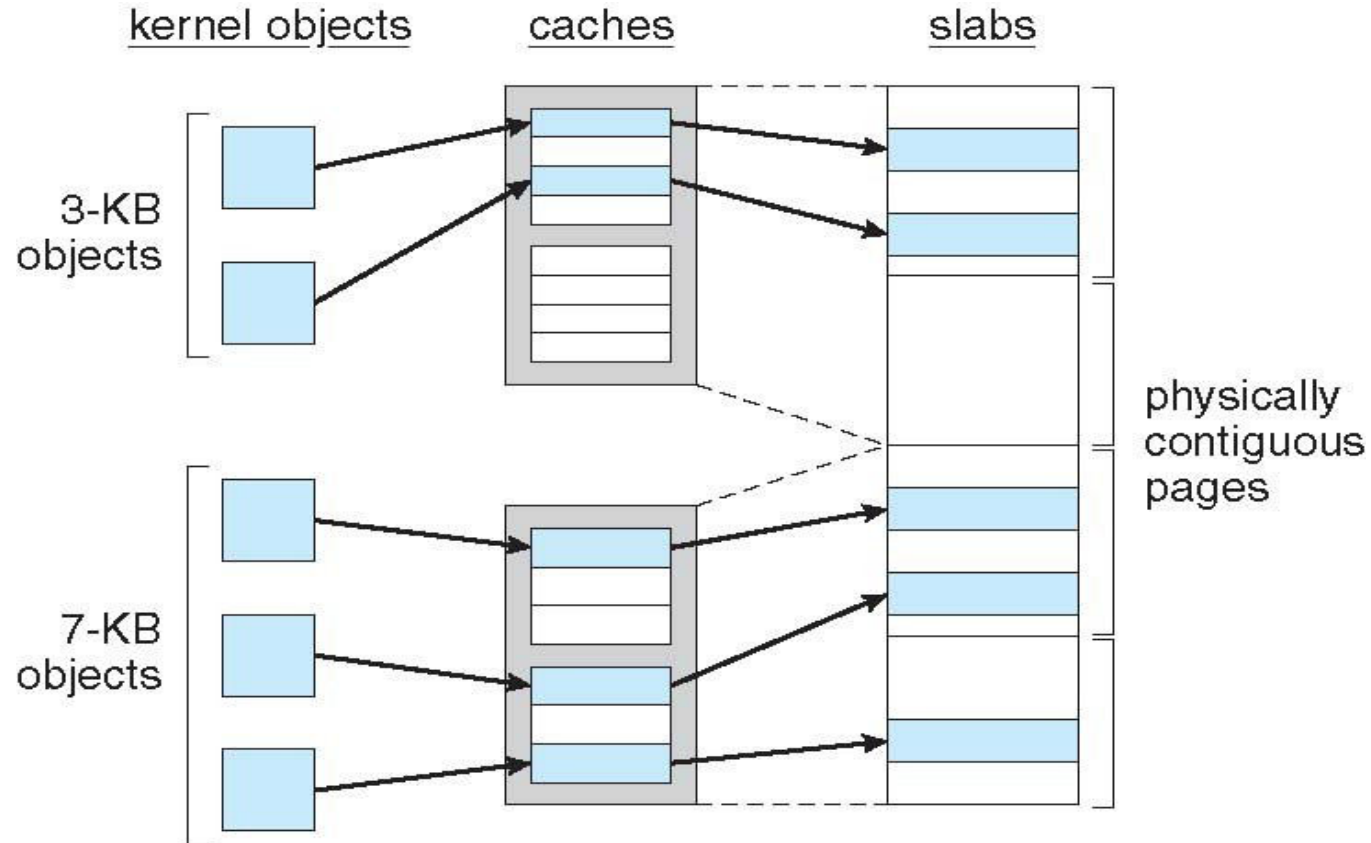
Split into AL and Ar of 128KB each

One further divided into BL and BR of 64KB

One further into CL and CR of 32KB each – one used to satisfy request

- Advantage – quickly coalesce unused chunks into larger chunk
- Disadvantage - fragmentation

Slab Allocator



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



Other considerations

Other Considerations -- 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 \longrightarrow prepaging loses

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

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

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

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

