Chapter 9: Virtual Memory

De la mémoire vers la mémoire virtuelle

- So far: various memory management strategies
  - Keep many processes in memory → multiprogramming
  - Require that the entire process is in memory

- Virtual memory allows the execution of a process not completely in memory
  - Programs >> main memory size
  - Abstracts main memory into extremely large uniform array
  - Allows processes to share files, to implement shared memory

---

Chapter 9: Virtual Memory

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

---

1. Background

- Memory management (previous chapter)
  - Instruction being executed **must** be in physical memory
  - Place the entire logical memory in physical memory
  - Dynamic loading may help
    - Special precaution / extra work
  - Seems necessary & reasonable
    - Unfortunate
    - Limits the size of a program

- Entire program is not needed in many cases:
  - Code for unusual error condition
  - Array / lists allocate more memory than needed
  - Some options / features rarely used

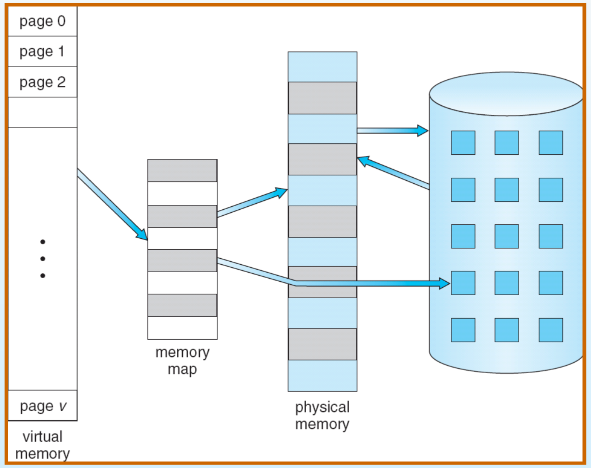
Background (cont)

- Benefits
  - No constraint by the limit of the physical memory
  - More programs could be run at the same time
    - Increase in CPU utilization, throughput
    - Same response time or turnaround
  - Less I/O to load/swap each user program / run faster

Background (cont)

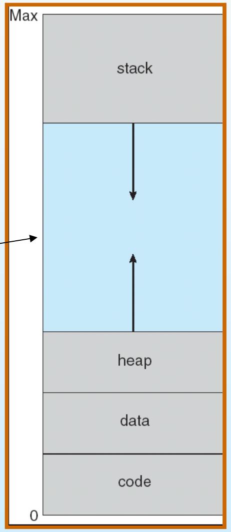
- **Virtual memory** – separation of user logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Programming task much more easier
  - Allows address spaces to be shared 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

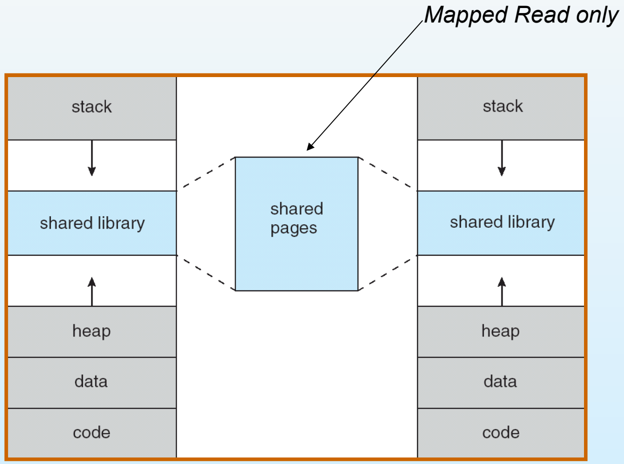


Virtual Address Space

- Refers to the logical view of how a process is stored in memory
- In fact physical memory may be organized in page frames
- Pages frames may be assigned to a process in a non contiguous way
- The MMU maps logical pages to physical pages
- Hole (***sparse address space***) is part of the virtual address space
  - Require physical addresses only if the heap/stack grows
- Allows also sharing of files, memory, process creation, libraries



Shared Library Using Virtual Memory

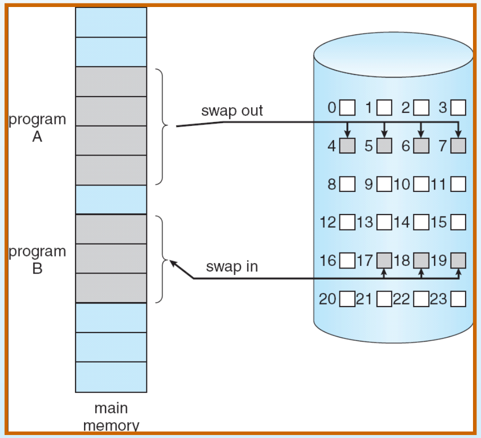


---

2. 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

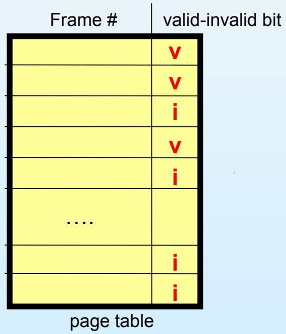


Basic concept

- Pager "*guesses*" which pages will be used before the process is swapped out again
- Need support to distinguish between the pages that are
  - In memory
  - On the disk

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
	i
	v
	i
….	
	i
	i

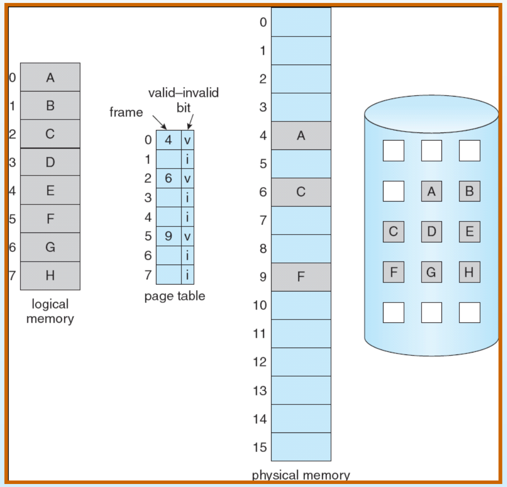
page table

- 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

- Page marked invalid has no effect if the process never attempts to access that page
- If we *guess right*, the process will run exactly as though we have brought in all pages
- While pages are **memory resident**, execution proceeds normally



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 process
   - Just not in memory → page it in
2. Get empty frame
3. Swap page into frame
4. Update page table: set validation bit = **v**
5. Restart the instruction that caused the page fault

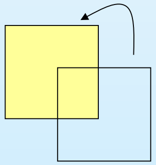
Extreme case: start executing a process with *no* page in memory
→ pure demand paging

Page Fault (Cont.)

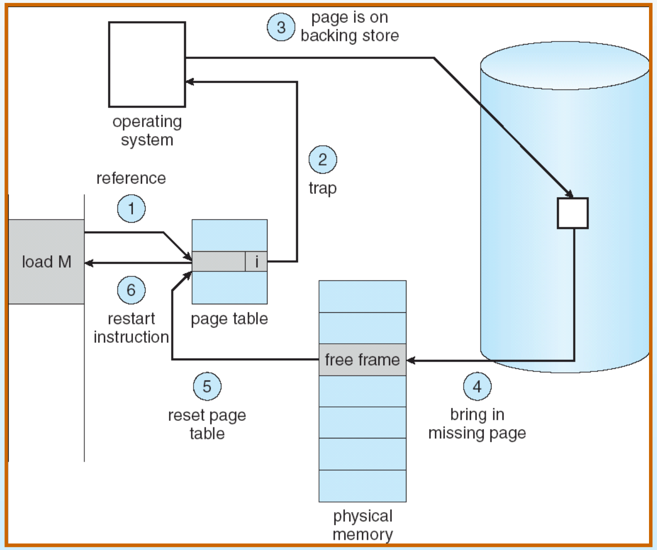
Some programs could access several new pages of memory with each instruction execution
→ poor performance
→ locality of reference results in reasonable performance from demand paging

- Restart instruction
  - Must save the state of the interrupt process
    - restart the process in **exactly** the same place
    - If page fault when writing result, restart the whole instruction
  - Problems, for instance with block move

    Solution: check locations before start or use registers



Steps in Handling a Page Fault



Performance of Demand Paging

- **Page fault rate** 0 ≤ *p* ≤ 1
  - if *p* = 0, no page faults
  - if *p* = 1, every reference is a fault
- **Effective Access Time** (EAT)

  EAT = (1 – *p*) x (memory access time)
  + *p* x (page fault overhead
  + swap page out
  + swap page in
  + restart overhead)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p x (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!
- Performance degradation < 10%
  → p < 0.0000025

Process Creation

- Virtual memory allows other benefits during process creation:
  - Copy-on-Write
  - Memory-Mapped Files (later)
- `fork()` system call creates a child process as a duplicate of its parent
  - Many child call `exec()` system call immediately after creation
  - Unnecessary code copy… waste of time

3. 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
  - Only pages that can be modified are marked CoW (not the code)

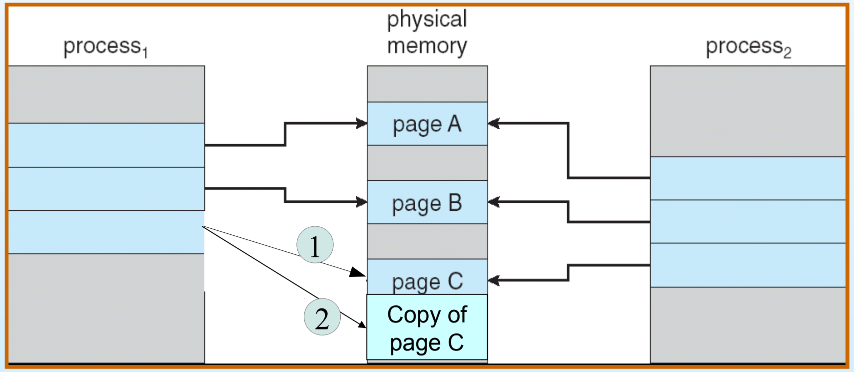
- CoW allows more efficient process creation as only modified pages are copied

- Free pages are allocated from a **pool** of zeroed-out pages

---

① Before Process 1 Modifies Page C

② After Process 1 Modifies Page C



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 that will result in minimum number of page faults
- Same page may be brought into memory several times

---

4. Page Replacement

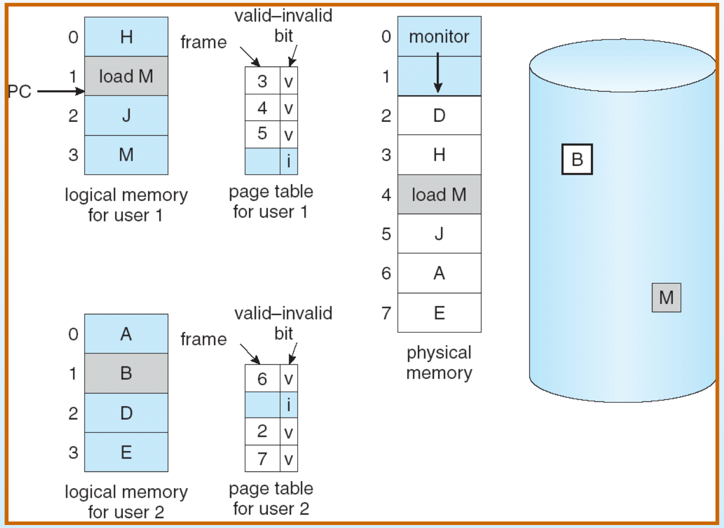
- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement

- 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
  - Frame allocation algorithm
  - Page replacement algorithm

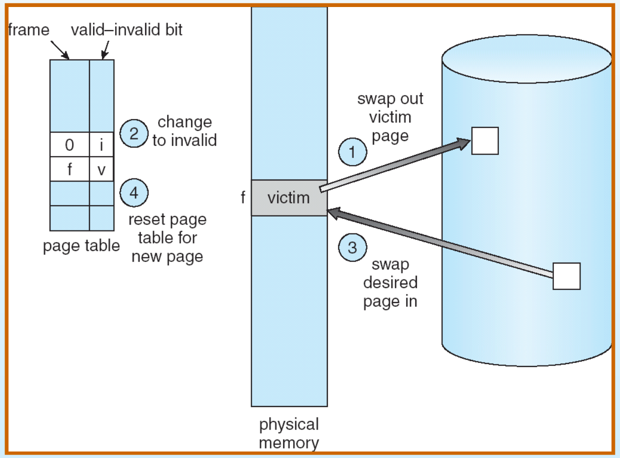
Need For Page Replacement



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
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Restart the process

Page Replacement



Page Replacement Algorithms

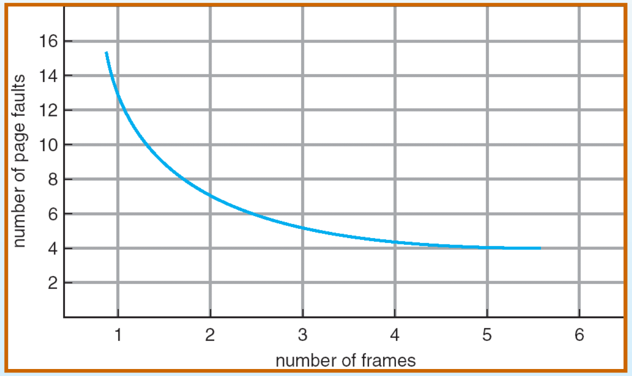
- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
- In all our examples, the reference strings are

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

and

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

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

- Reference string: 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 ?

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

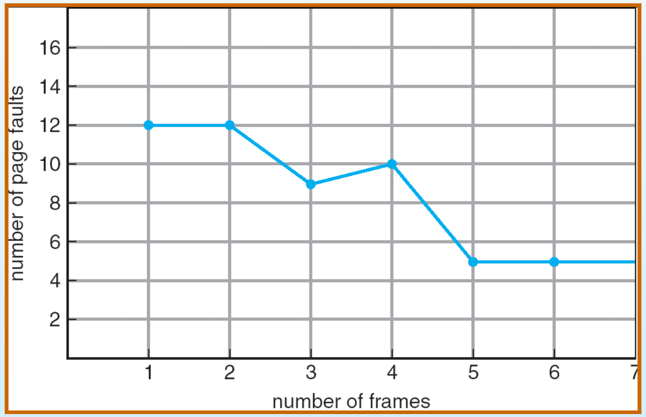
- 4 frames

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

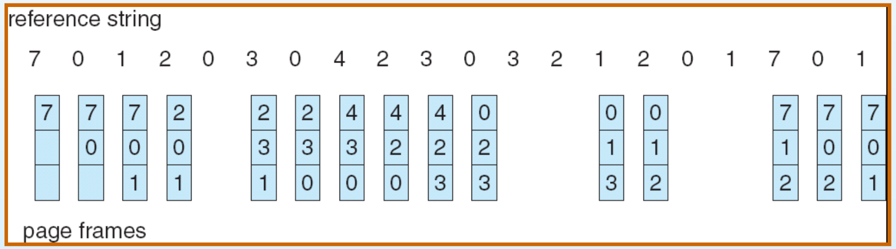
10 page faults

- Belady's Anomaly: more frames → more page faults

FIFO Illustrating Belady's Anomaly



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

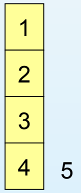
page frames

15 faults

Optimal Algorithm

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

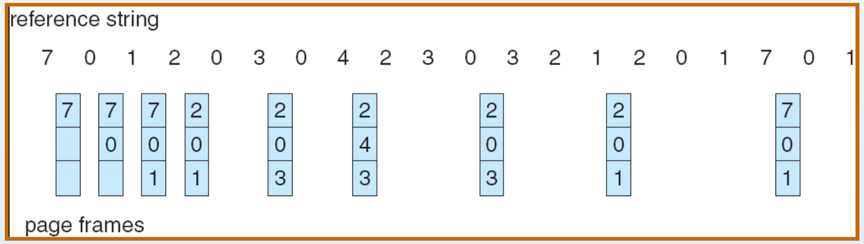
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



6 page faults

- 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

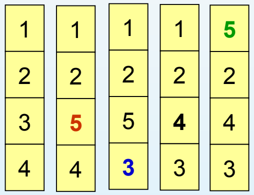
7	7	7	2	2	2	2	2	7
	0	0	0	0	4	0	0	0
		1	1	3	3	3	1	1

page frames

9 faults (vs 15 for FIFO)

Least Recently Used (LRU) Algorithm

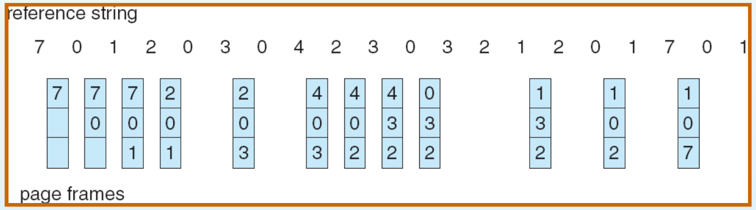
- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**



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

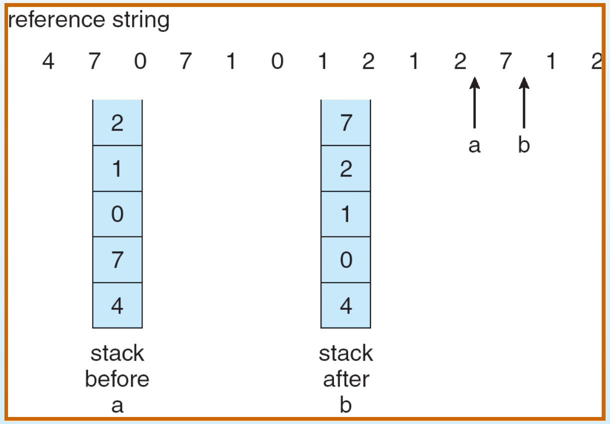


12 faults (vs 15 for FIFO)

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
- Neither implementation (counter or stack) conceivable without hardware support
  - Interrupt to update clock or stack
    - Slow every memory reference by a factor at least 10
  - Overhead that cannot be tolerated

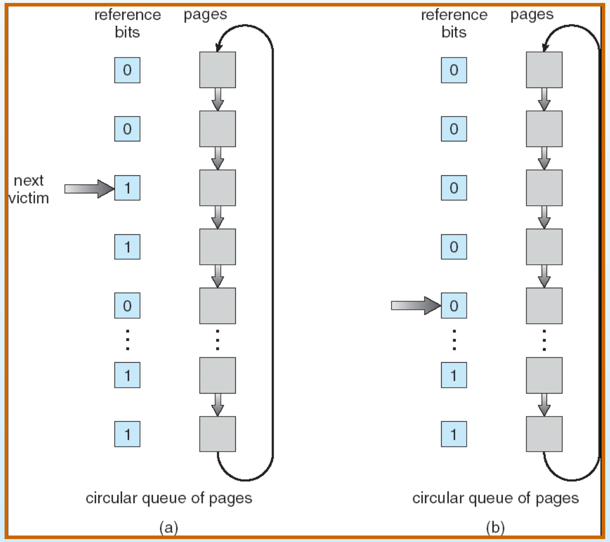
Use Of A Stack to Record The Most Recent Page References



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace one that is 0 (if one exists)
    - We do not know the order, however
- Additional reference bit
  - Shift register to record reference bit periodically
- 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



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- **LFU Algorithm**: replaces page with smallest count

  (Least 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

  (Most Frequently Used)

---

5. Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes ?
- 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 vs Priority 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 : number of frames allocated to p_i : $a_i = (s_i / S) \times m$
  - Example with $m = 64$, $s_1 = 10$, $s_2 = 127$:
    - We obtain $a_1 = 5$, $a_2 = 59$
- Priority allocation – Use a proportional allocation scheme using priorities rather than size

Global vs. Local Allocation

- Priority allocation (cont): 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 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
- With global replacement, a process cannot control its own page fault behavior

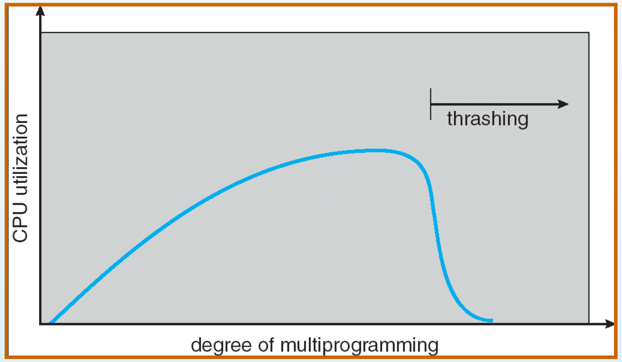
---

6. Thrashing

- If a process does not have "enough" frames, 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.)



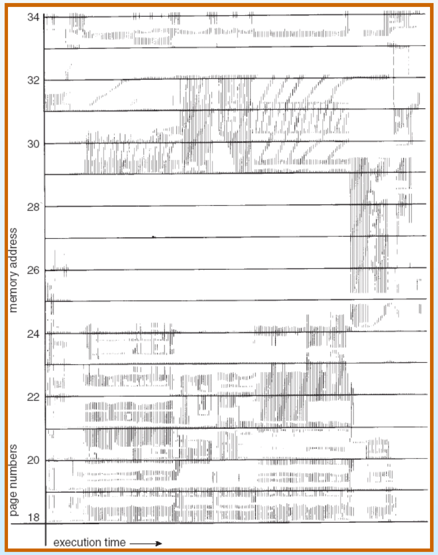
Limit the trashing by using local replacement algorithm / priority replacement :
process trashing → in paging queue most of the time → access time will increase
We need to provide a process with as many frames as it needs

How do we know how many frames it "*needs*" ?

Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- Allocate enough frames to a process to accommodate its current locality
- Why does thrashing occur?
  $\sum$ size of locality > total memory size
  Limit effects by using local or priority page replacement

Locality In A Memory-Reference Pattern



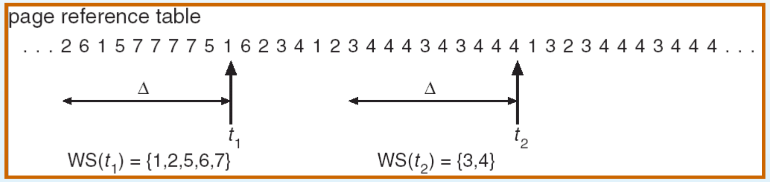
Locality model == unstated principle behind several concepts

If accesses to any type of data were random rather than patterned, caching would be useless…

Working-Set Model

- Δ = working-set window = a fixed number of page references
  Example: 10,000 instructions
- WSS_i (working-set size 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 $\Delta = \infty$: will encompass entire program
- $D = \sum WSS_i$ = total demand frames
- If D > m: Thrashing
- Policy if D > m, then suspend one of the processes

Working-set model



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 1 3 2 3 4 4 4 3 4 4 4 . . .

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

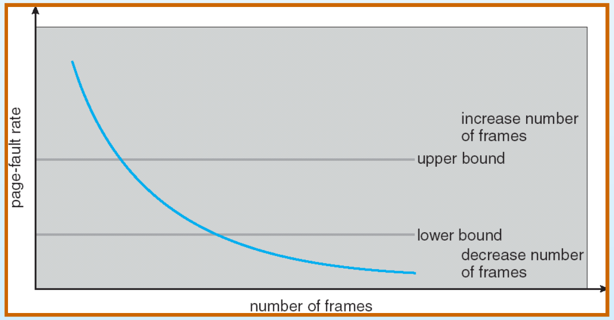
$WS(t_2) = \{3,4\}$

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts: copy and set the values of all reference bits to 0
  - If one of the bits in memory = 1: page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

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



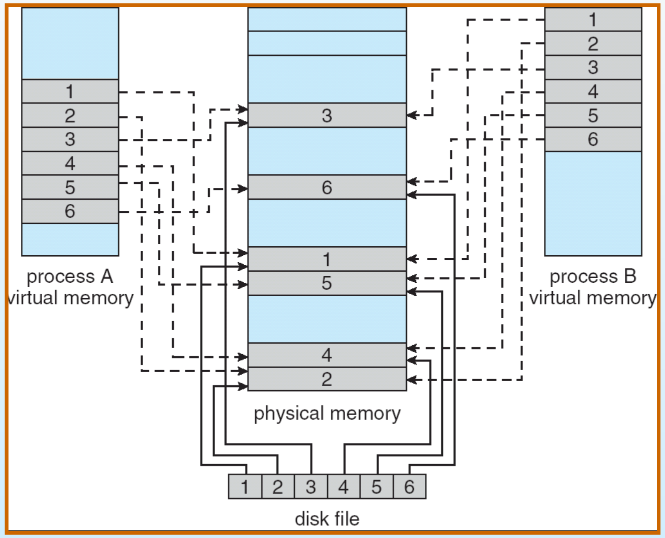
---

7. Memory-Mapped Files

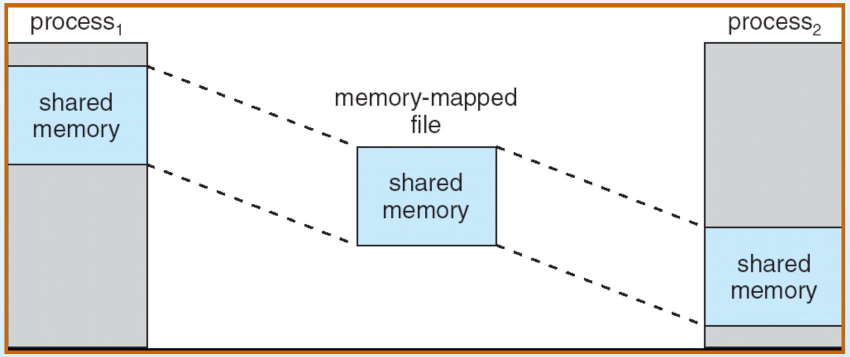
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using demand paging. A page-sized portion of the file is read from the file system into a physical page. Subsequent reads/writes to/from the file 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



8. Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes: should limit waste due to fragmentation
  - Some kernel memory needs to be contiguous

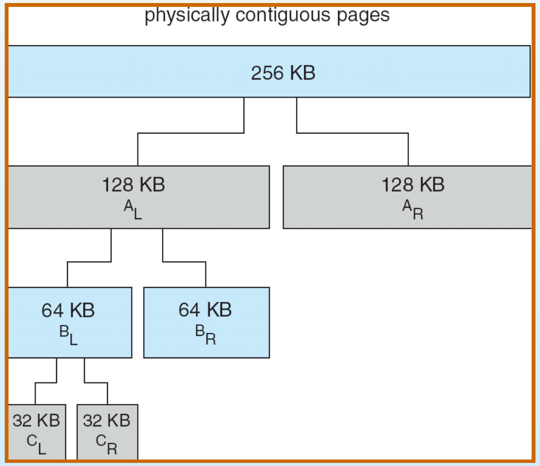
Buddy System

- 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

- Pros and cons
  - Coalescing to quickly combine adjacent buddies
  - Rounding up to next highest power of 2 causes fragmentation

---

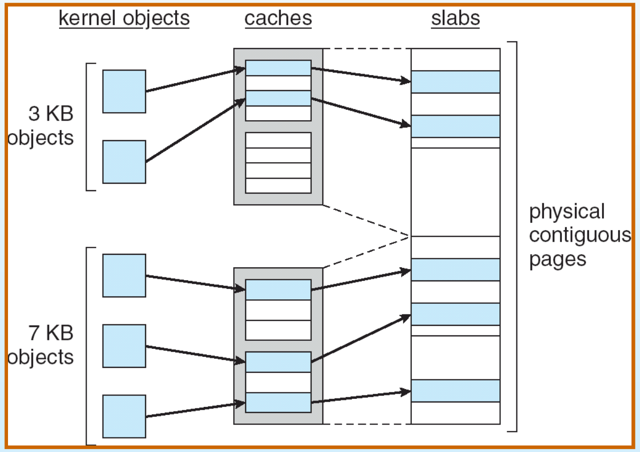
Buddy System Allocator



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

Slab Allocation



---

9. Other Issues – Prepaging

- Prepaging
  - To reduce the large number of page faults that occur at process startup
  - Prepage all or some of the pages a process will need, before they are referenced (e.g., pages from working set)
  - But if prepaged pages are unused, I/O and memory were wasted
  - Assume *s* pages are prepaged and *α* of the pages are used
    - Is cost of *s x α* saved pages faults greater or less than the cost of prepaging *s x (1- α)* unnecessary pages*?*
    - *α* close to zero (resp. one): prepaging loses (resp. win)

Other Issues – Page Size

- Page size selection must take into consideration:
  - fragmentation
  - table size
  - I/O overhead (latency + transfer rate)
  - locality
  - number of page faults

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

- 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;
```

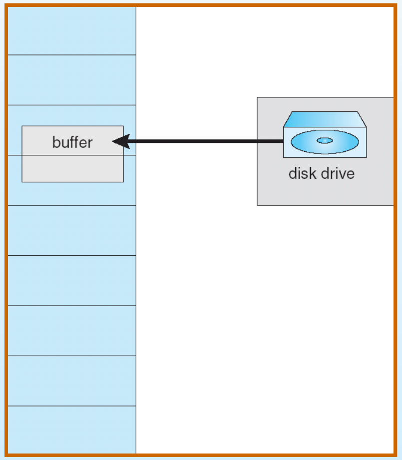
128 page faults

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



---

10. Operating System Examples
10a. Windows XP

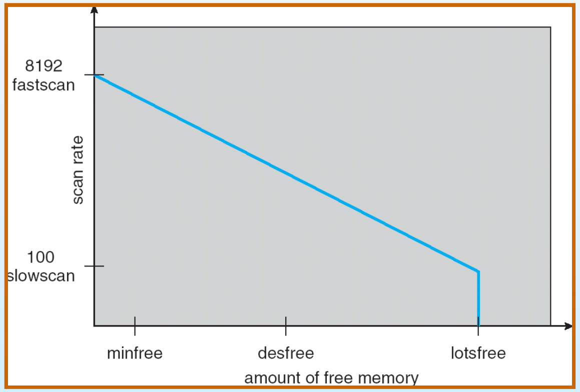
- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page.
- Processes are assigned **working set minimum** (50) and **working set maximum** (345)
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may 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

10b. Solaris

- Maintains a list of free pages to assign faulting processes
- *lotsfree* – threshold parameter (amount of free memory) to begin paging (1/64 of the size of the physical memory)
- *desfree* – threshold parameter to increasing paging
- *minfree* – threshold parameter to begin swapping
- Paging is performed by *pageout* process
- pageout scans pages using modified clock algorithm
- *scanrate* is the rate at which pages are scanned. This ranges from *slowscan* to *fastscan*
- pageout is called more frequently depending upon the amount of free memory available

---

Solaris 2 Page Scanner



Résumé

- Mémoire virtuelle: mapper un large espace d'adressage logique sur une mémoire physique plus petite – permet de faire tourner des gros processus, et d'augmenter le degré de multiprogrammation
- Pagination à la demande: table des pages, et faute de page si la page n'est pas en mémoire; mettre la page en mémoire et relancer l'instruction qui a provoqué la faute
- Remplacement de pages lorsque la mémoire est pleine; attention à l'anomalie de Belady
- Politique d'allocation des cadres de page; remplacement local (interne à un processus) ou global (avec priorité par exemple); modèle du working-set pour éviter le thrashing
- Fichiers mappés en mémoire: accès fichier = accès mémoire
- Mémoire système: allocation buddy ou slab

Exercices

1. Demand-paged memory. Page table: in registers.

* 8 millisec to service page fault if there is an empty page, or the replaced page is not modified

* 20 millisec if the replaced page is modified

* Memory access time: 100 nanosec

If the page to be replaced is modified 70% of the time, what is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanosec?

2. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?