Chapter 3: Memory Management

3.2. Virtual Memory

GV: Nguyễn Thị Thanh Vân

Outline

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Other Considerations

Objectives

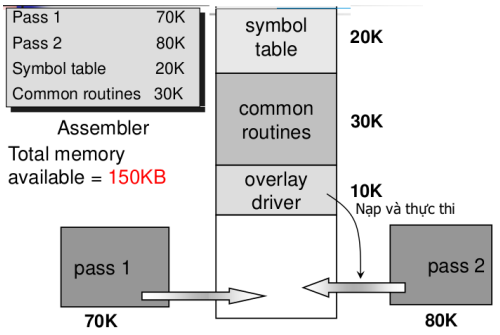
- Define virtual memory and describe its benefits.
- Illustrate how pages are loaded into memory using demand paging.
- Apply the FIFO, optimal, and LRU page-replacement algorithms.
- Describe the working set of a process and explain how it is related to program locality.
- Describe how Linux, Windows 10, and Solaris manage virtual memory.
- Design a virtual memory manager simulation in the C programming language.

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 the 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

Overlay technology

- **Overlay**
  - overlay is a technique to run a program that is bigger than the size of the physical memory
  - By keeping only those instructions and data that are needed at any given time.
  - Divide the program into modules in such a way that not all modules need to be in the memory at the same time.

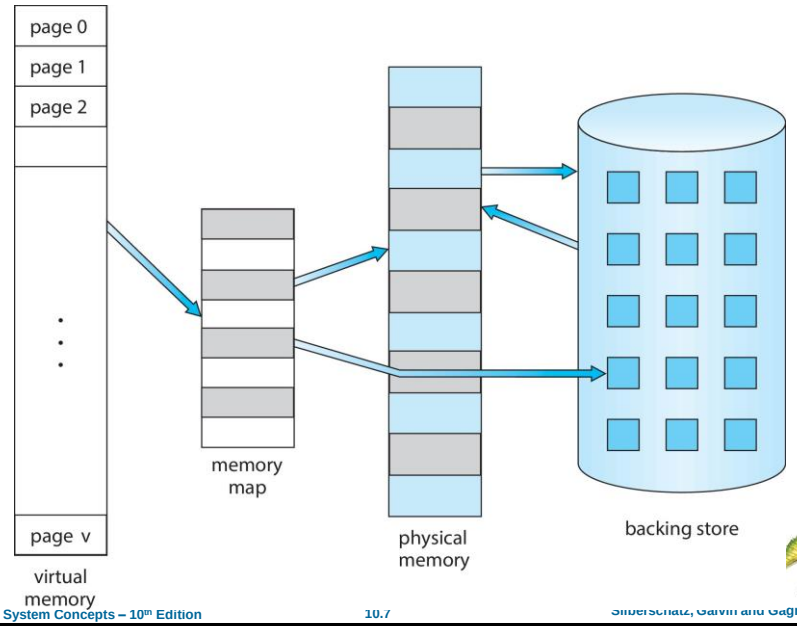


- Solution: vrtual memory – auto technology: reduce Programmer's work

Virtual memory

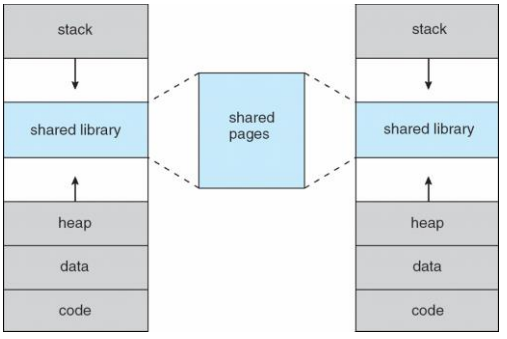
- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes
- **Virtual 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



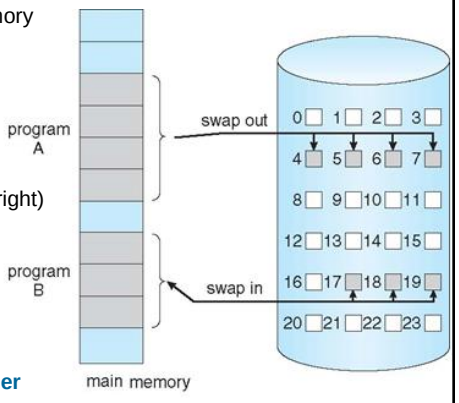
Virtual-address Space

- Usually design logical address space for the 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…
- 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



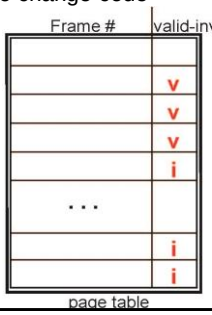
Demand Paging

- Instead of bringing the entire program into memory at load time, bring a page into memory only when it is needed => **Demand Paging**
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to **paging system with swapping** (right)
- Page reference
  - 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**

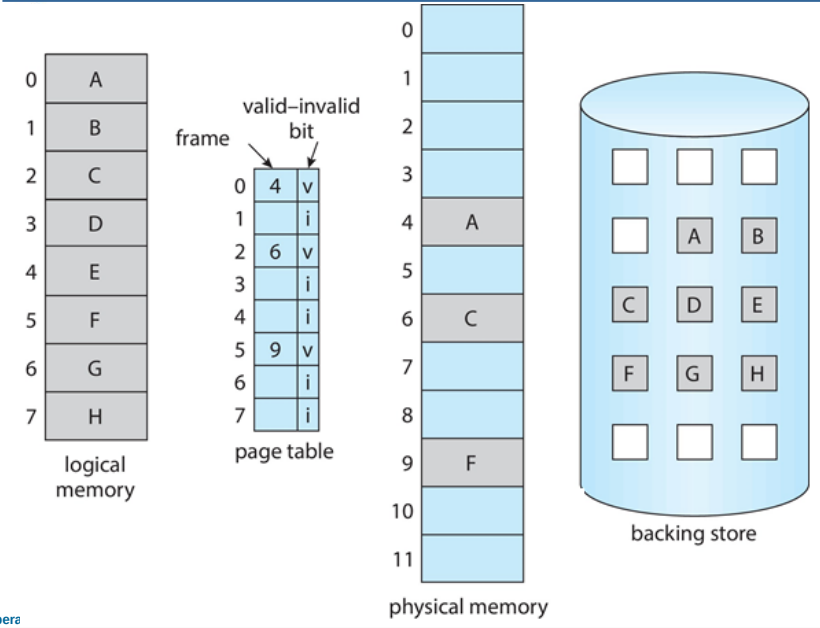


Basic Concepts

- With swapping, the pager guesses which pages will be used before swapping them out again
  - 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 and needing to change code
- Use page table with valid-invalid bit:
  - (**v** ⇒ in-memory, **i** ⇒ not-in-memory)
  - Initially valid–invalid bit is set to **i** on all entries
  - During MMU address translation,
    - if valid–invalid bit in the page table entry is **i**
    - ⇒ page fault



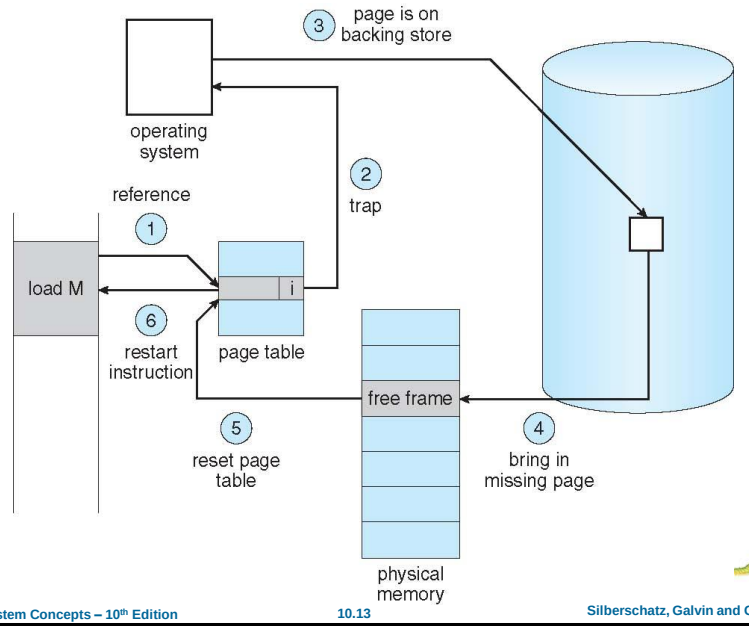
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. The first reference to a page will trap to operating system.
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
   - Found: goto step5
   - Not found: choose 1 victim, swap out victim to backing store
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory.
   - Set validation bit = **v**
6. Restart the instruction that caused the page fault

Steps in Handling a Page Fault (Cont.)



Steps in Handling Page Fault

1. Khi dò tìm (ref) trong bảng trang để lấy các thông tin cần thiết cho việc chuyển đổi địa chỉ, nếu nhận thấy trang đang được yêu cầu truy xuất là bất hợp lệ, cơ chế phần cứng sẽ phát sinh một lỗi trang (page fault) - một ngắt để báo cho OS
2. Hệ điều hành sẽ xử lý lỗi trang như sau : Kiểm tra truy xuất đến bộ nhớ là hợp lệ hay bất hợp lệ
   - Nếu truy xuất bất hợp lệ : kết thúc tiến trình
   - Ngược lại : đến bước 3
3. Tìm vị trí chứa trang muốn truy xuất trên đĩa. Tìm một khung trang trống trong bộ nhớ chính
   - Nếu tìm thấy : đến bước 5 0 update pg table
   - Nếu không còn khung trang trống, chọn một khung trang « nạn nhân » và chuyển trang « nạn nhân » ra bộ nhớ phụ (lưu nội dung của trang đang chiếm giữ khung trang này lên đĩa),
4. Mang trang cần truy xuất vào mem.
5. Cập nhật bảng trang tương ứng
   - Chuyển trang muốn truy xuất từ bộ nhớ phụ vào bộ nhớ chính : nạp trang cần truy xuất vào khung trang trống đã chọn (hay vừa mới làm trống).
6. Tái kích hoạt tiến trình người sử dụng.

Performance of Demand Paging

- Demand paging can significantly affect the performance of a computer system.
  - When page fault, how much time is needed to service a page fault
- Three major activities of the page-fault service time
  1. Service the interrupt – careful coding means just several hundred instructions needed
  2. Input the page from disk – lots of time
  3. Restart the process – again just a small amount of time

  1) & 3) may take from 1 to 100 microseconds each - can be reduced
- Page Fault Rate $0 \le p \le 1$, p: the probability of a page fault
  - If $p = 0$, no page faults
  - If $p = 1$, every reference is a fault
- Effective Access Time (EAT) - how much time is needed to service a page fault.

  EAT = (1 – *p*) x memory access + p x page fault time

  page fault time: page fault overhead + swap page out + swap page in

Tasks in Page fault

1. Trap to the operating system.
2. Save the registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page referencewas legal, and determine the location of the page in secondary storage.
5. Issue a read from the storage to a free frame:
   a. Wait in a queue until the read request is serviced.
   b. Wait for the device seek and/or latency time.
   c. Begin the transfer of the page to a free frame.
6. While waiting, allocate the CPU core to some other process.
7. Receive an interrupt from the storage I/O subsystem (I/O completed).
8. Save the registers and process state for the other process (if step 6 is executed).
9. Determine that the interrupt was from the secondary storage device.
10. Correct the page table to show that the desired page is now in mem
11. Wai t for the CPU core to be allocated to this process again.
12. Restore the registers, process state, and new page table, and then resume the interrupted instruction.

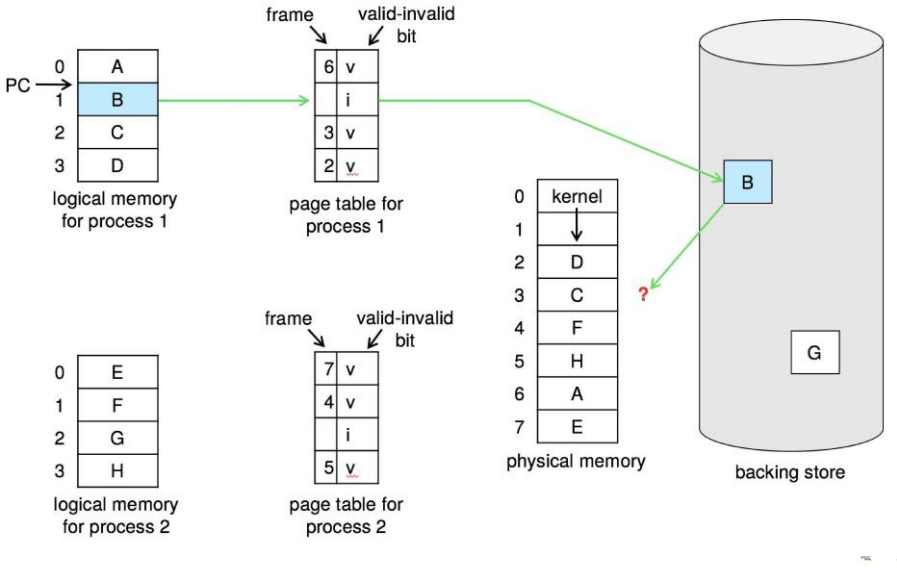
Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds =8,000,000 (ns)
- EAT = (1 – p) x 200 + p x 8,000,000 (ns)
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault (p=0.001), then
  EAT = 200 + 0.001 x 7,999,800 = 8,199.8 nanoseconds
  = 8.2 microseconds.
  This is a slowdown by a factor of 40 = $8.2*10^3/200$ (nanoseconds)
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
    - one page fault in every 400,000 (399,990) memory accesses

What Happens if There is no Free Frame?

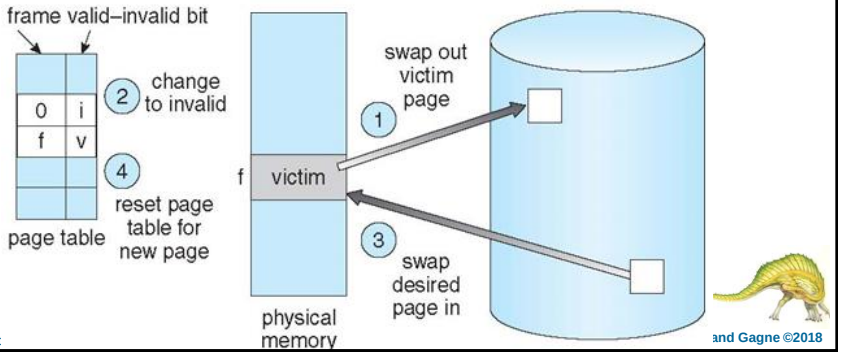
- Page is full: need Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
- Page 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
  - 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, (1)use a page repl algorithm to select a **victim frame**
   - Write victim frame to disk if dirty (only modified pages are written to disk)
   - **(2)** update page table – change to invalid
3. (3)Bring the desired page into the (newly) free frame;
   (4)update the page & frame tables
4. Continue the process by restarting the instruction that caused the trap



Page replacement

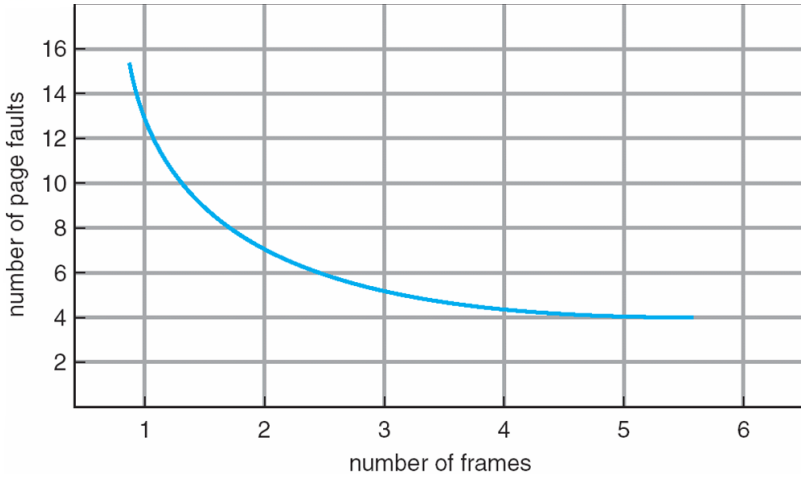
- Page replacement is basic to demand paging.
  - It completes the separation between logicalmemory and physical memory.
  - an enormous virtual memory can be provided for programmers on a smaller physical memory.
  - the size of the logical address space is no longer constrained by physical memory.
- We must solve twomajor problems to implement demand paging:
  - develop a frame-allocation algorithm and
  - develop a page-replacement algorithm.
- That is, if we have multiple processes in memory, we must decide:
  - how many frames to allocate to each process;
  - when page replacement is required,
  - we must select the frames that are to be replaced.
- Designing appropriate algorithms to solve these problems is an important task

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



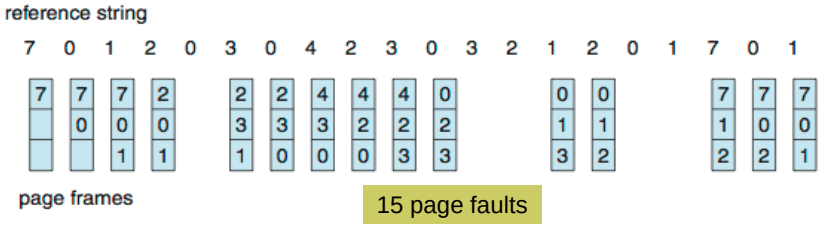
the number of frames increases, the number of page faults drops to some minimal level.

Replacement Algorithms

- First-In-First-Out (FIFO)
- Optimal Algorithm
- Least Recently Used (LRU)
- LRU Approximation
  - Additional-Reference-Bits Algorithm
  - Second-Chance Algorithm
  - Enhanced Second-Chance Algorithm
- Counting Algorithms
- Page-Fault Frequency Algorithm

First-In-First-Out (FIFO) Algorithm

- Replace page that **FIFO: the oldest page is chosen**
  - No need record the time that page is brought in
  - a FIFO queue to hold all pages inmemory: replace head of thq queue
- Ex: 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)



- FIFO is easy to understand and program. But, its performance is not always good

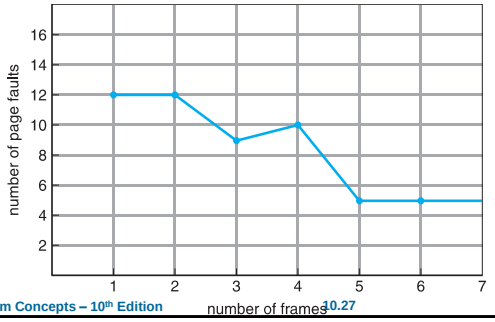
Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!

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

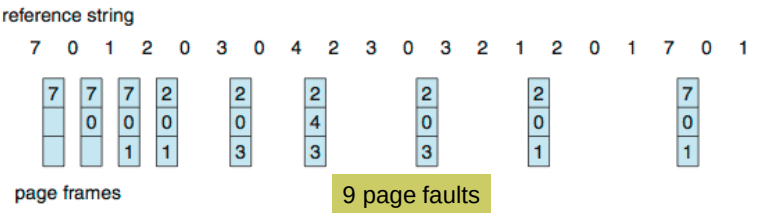
	1	2	3	4	1	2	5	1	2	3	4	5
f1	1	1	1	1			5	5	5	5	4	4
f2		2	2	2			2	1	1	1	1	5
f3			3	3			3	3	2	2	2	2
f4				4			4	4	4	3	3	3
	x	x	x	x			x	x	x	x	x	x

- Graph illustrating Belady's Anomaly



Optimal Algorithm

- Replace page that **will not be used for longest period of time**
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

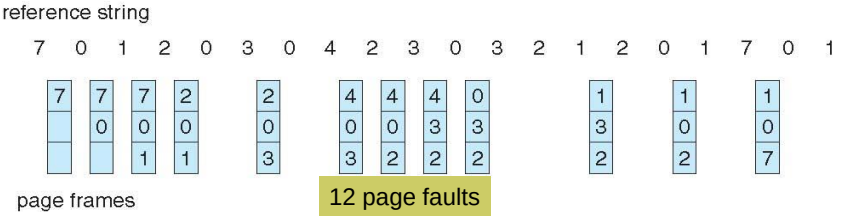


9 page faults

- How do you implement this?
  - Can't read the future reference string - dificult to implement
- Optimal is much better than a FIFO algorithm
- Optimal is an example of **stack algorithms** that don't suffer from Belady's Anomaly

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

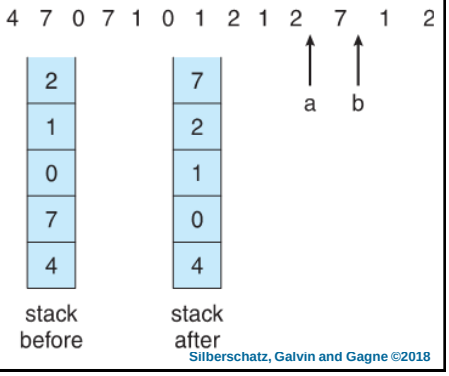


12 page faults

- LRU is better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- LRU is another example of stack algorithms; thus, it does not suffer from Belady's Anomaly
- appropriate for software or microcode implementations of LRU replacement

LRU Algorithm Implementation

- **Time-counter implementation**
  - Every page entry has a time-counter variable; every time a page is referenced through this entry, copy the value of the clock into the time-counter
  - When a page needs to be changed, look at the time-counters to find smallest value
    - Search through a table is needed
- **Stack implementation**
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - Move it to the top, ex 7
    - Requires 6 pointers to be changed
  - But each update more expensive
  - No search for replacement
- Use of a stack to record most recent page references



LRU Approximation Algorithms

- Needs special hardware
- **Reference bit**
  - With each page associate a bit, initially = 0 (by OS)
  - When page is referenced (a process executes), bit set to 1 (by HW)
- 1 entry in page table

Page#	Bit valid /invalid	Dirty bit	Reference bit

- After some time, we can determine which pages have been used and which have not been used by examining the reference bits
  - although we do not know the order of use
- When a page needs to be replaced, replace any with reference bit = 0 (if exists)
- To know the order of page, LRU appr use:
  - Additional-Reference-Bits Algorithm
  - Second-Chance **(or Clock)** Algorithm
  - Enhanced Second-Chance Algorithm

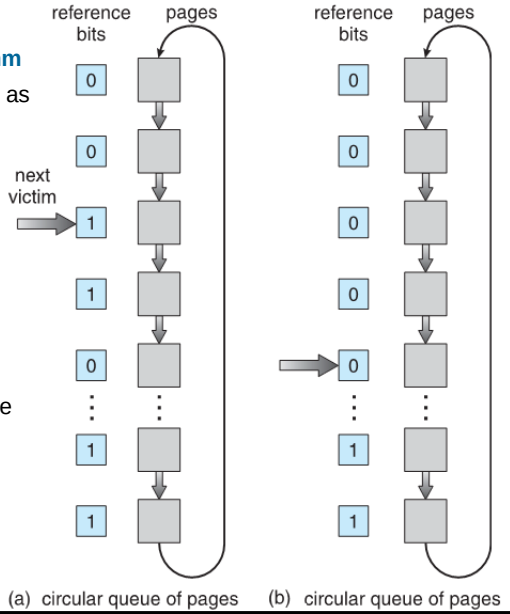
LRU Approximation Algorithms

- **Additional-Reference-Bits Algorithm**
  - keep an 8-bit byte for each page in a table - history of page use for the last 8 time periods.
  - At regular intervals (every 100 ms), a timer interrupt transfers control OS
  - OS shifts the **reference bit** for each page into the **high-order** bit of its 8-bit byte, shifting the other bits right by 1 bit and discarding the low-order bit.
  - HR=00000000: the page has not been used for 8 time periods
  - HR=11111111: A page that is used at least once in each period
  - Ex: A page with a HR1 has been used more recently than HR3

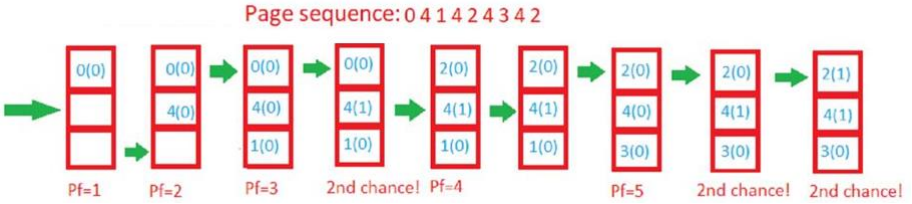
HR =11000100									
HR =11100010									
HR =01110001									

LRU Approximation Algorithms

- **Second-chance (or Clock) algorithm**
  - referred to as the **clock** algorithm, as a circular queue.
  - Generally, **FIFO**, plus hardware-provided reference bit
- **Clock** replacement (circular queue)
  - 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 (victim), subject to same rules



Ex, Second-chance algorithm



- **Initially**, all frames are empty so after first 3 passes they will be filled with {0, 4, 1} and the second chance array will be {0, 0, 0} as none has been referenced yet. Also, the pointer will cycle back to 0.
- **Pass-4:** Frame={0, 4, 1}, second_chance = {0, 1, 0} [4 will get a second chance], pointer = 0 (No page needed to be updated so the candidate is still page in frame 0), pf = 3 (No increase in page fault number).
- **Pass-5:** Frame={2, 4, 1}, second_chance= {0, 1, 0} [0 replaced; it's second chance bit was 0, so it didn't get a second chance], pointer=1 (updated), pf=4
- **Pass-6:** Frame={2, 4, 1}, second_chance={0, 1, 0}, pointer=1, pf=4 (No change)
- **Pass-7:** Frame={2, 4, 3}, second_chance= {0, 0, 0} [4 survived but it's second chance bit became 0], pointer=0 (as element at index 2 was finally replaced), pf=5
- **Pass-8:** Frame={2, 4, 3}, second_chance= {0, 1, 0} [4 referenced again], pointer=0, pf=5
- **Pass-9:** Frame={2, 4, 3}, second_chance= {1, 1, 0} [2 referenced again], pointer=0, pf=5

LRU Approximation Algorithms

- **Enhanced Second-Chance Algorithm**
  - Improve algorithm by using reference bit and modify bit (if available)
  - Take ordered pair (reference, modify):
    - (0, 0) neither recently used not modified – best page to replace
    - (0, 1) not recently used but modified – not quite as good, must write out before replacement
    - (1, 0) recently used but clean – probably will be used again soon
    - (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
- **Lease Frequently Used** (**LFU**) **Algorithm**:
  - Replaces page with smallest count

- pg5: f1=2, f2=2, f3=1, f4=1
  - Pg3 came first => remove, f3=0
  - Add pg5, f5=1
- pg6: f1=2, f2=2, f5=1, f4=1:
  - Pg4 came first => remove pg4, f4=0
  - Add pg6, f6=1

- **Most Frequently Used** (**MFU**) **Algorithm**:

- pg5: f1=2, f2=2, f3=1, f4=1
  - Remove, f1=1. Add pg5, f5=1
- pg6: f5=1, f2=2, f3=1, f4=1:
  - Remove pg2, f2=1. Add pg6, f6=1
- pg2: f5=1, f6=1, f3=1, f4=1:
  - Remove pg3, f3=0. Add pg2, f2=2
- pg1: f5=1, f6=1, f2=2, f4=1:
  - Remove pg2, f2=1. Add pg1, f1=2

Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes
  - 93 free frames and 2 processes, ?frames/process
  - => free-frame list was exhausted
  - try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into
- Strategy: Processs need minimum Number of Frames
  - is defined by the computer architecture
  - the number of frames allocated to each process decreases, the page-fault rate increases, slowing process execution.
  - when a page fault occurs before an executing instruction is complete, the instruction must be restarted.
  - Have enough frames to hold all the different pages
  - ***Maximum*** of course is total frames in the system
- Two major allocation schemes - Allocation Algorithms
  - Fixed allocation
  - Priority allocation
- Many variations

Fixed Allocation

- **Equal allocation** – all processes gets the same number of frames.
  - 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
  - Example

s_i = size of process p_i

$S = \sum s_i$

m = total number of frames

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

$m = 62$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

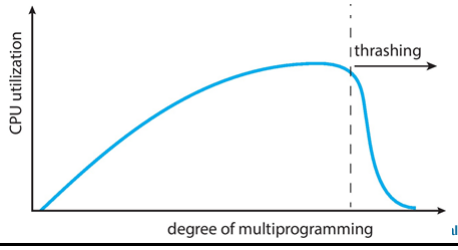
$a_2 = \frac{127}{137} \times 62 \approx 57$

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
  - Process execution time can vary greatly
  - Greater throughput so more commonly used
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory
  - What if a process does not have enough frames?

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 the replaced frame back
- This leads to:
  - Low CPU utilization
  - OS thinking that it needs to increase the degree of multiprogramming
  - Another process added to the system
  - Which results in even higher page fault rate
- **Thrashing**. A process is busy swapping pages in and out (high paging activity)



Preventing thrashing

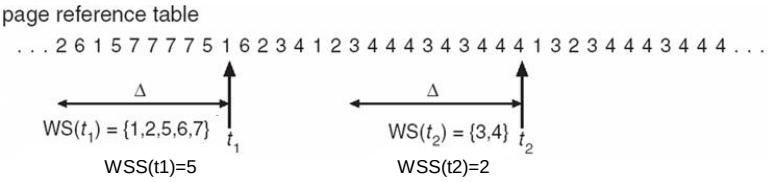
- Prevent thrashing: must provide a process with as many frames as it needs.
  - But how do we know how many frames it "needs"?
- One strategy starts by looking at how many frames a process is actually using.
- **=> Locality model**
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size

- To avoid thrashing: using a local/priority replacement algorithm
  - Calculate the Σ size of locality
  - Policy: if Σ size of locality > total memory size
    - suspend or swap out one of the processes
- Issue: how to calculate "Σ size of locality"

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references, Ex: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- Example: if Δ=10



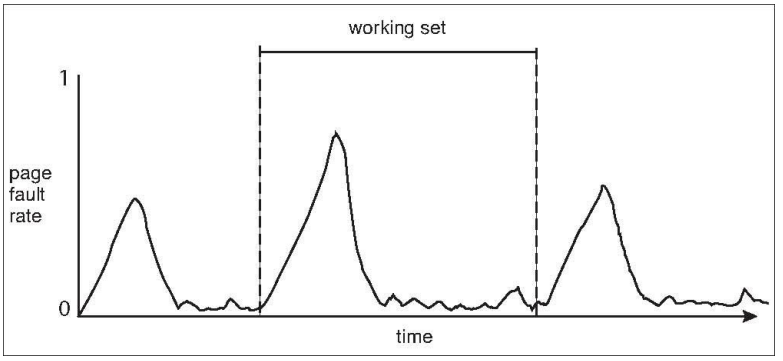
- Observation
  - if Δ too small will not include the entire locality
  - if Δ too large will include several localities
  - if $\Delta = \infty \Rightarrow$ will include entire program
- $D = \Sigma\ WSS_i \equiv$ total demand frames (process i request WSSi)
  - Approximation of locality
  - Let m = total number of frames
  - If $D > m \Rightarrow$ Thrashing, then suspend or swap out one of the processes

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page *i*
    - $B1_i$ and $B2_i$
  - Whenever a timer interrupts copy the reference to one of the B_j and sets the values of all reference bits to 0
  - If either $B1_i$ or $B2_i$ = 1, it implies that Page *i* is in the working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

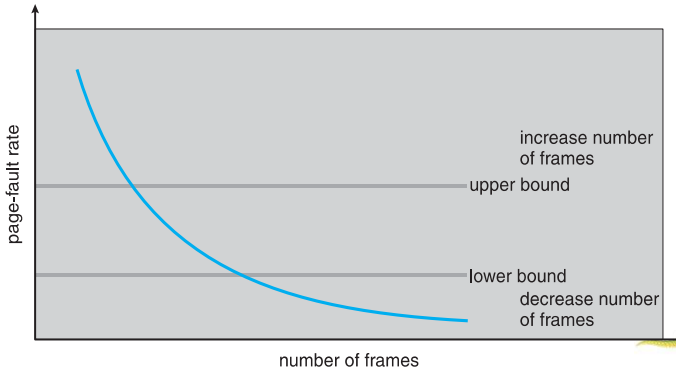
Working Sets and Page Fault Rates

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



Page-Fault Frequency Algorithm

- 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
- Example



Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

End of Chapter 3.2