Virtual Memory

Prof. Eun-Seok Ryu (esryu@skku.edu)
Multimedia Computing Systems Laboratory
http://mcsl.skku.edu
Department of Immersive Media Engineering
Department of Computer Education (J.A.)
Sungkyunkwan University (SKKU)

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- <u>Entire program code not needed at same time</u>
- Consider ability to <u>execute partially-loaded program</u>
  - Program no longer constrained by <u>limits of physical memory</u>
  - Each program takes less memory while running -> <u>more programs run at the same time</u>
    - 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

- Q: program 들은 크고 여러개이고 동시 사용하려 할 때, main memory 한계로 다 못 올리면 어떻게?
  > 가상메모리

CSLab Multimedia Computing Systems Lab.

SUNG KYUN KWAN UNIVERSITY(SKKU)

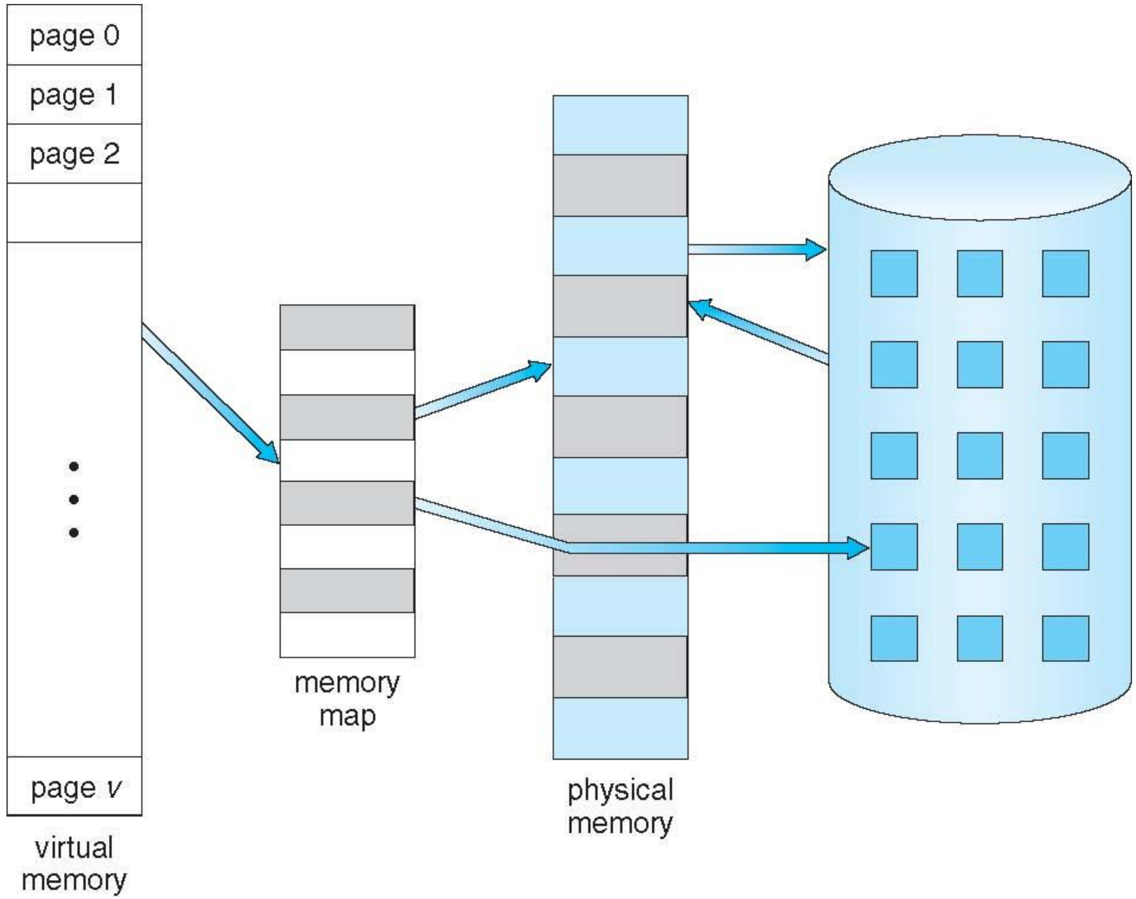
Background (Cont.)

- **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.)

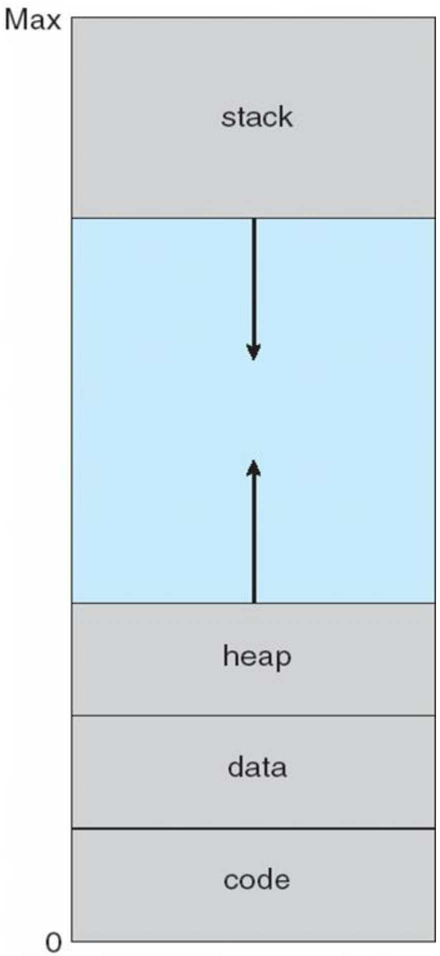
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - <u>MMU must map logical to physical</u>
- Virtual memory can be implemented via:
  - <u>Demand paging</u>
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

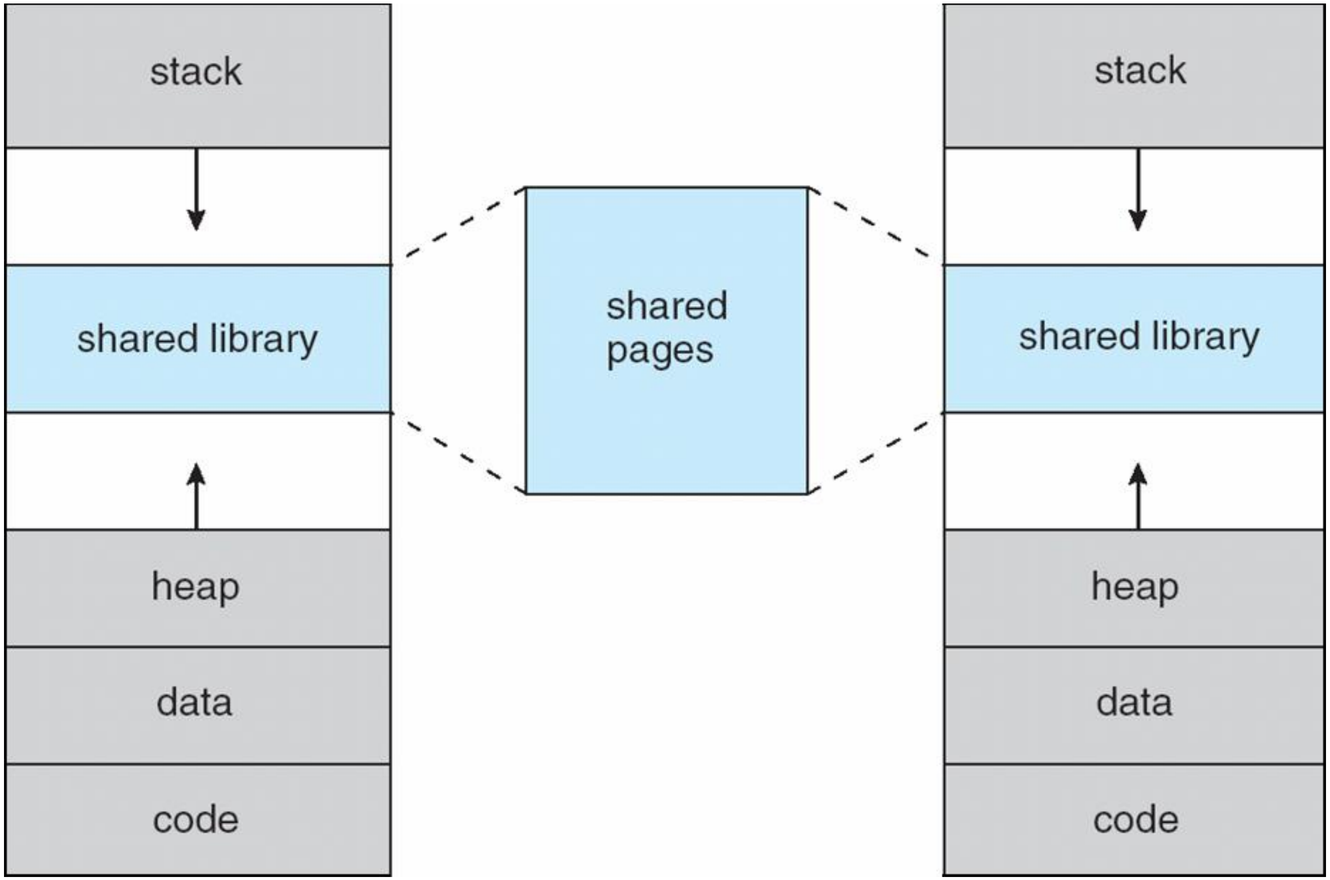


Virtual-address Space

- 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



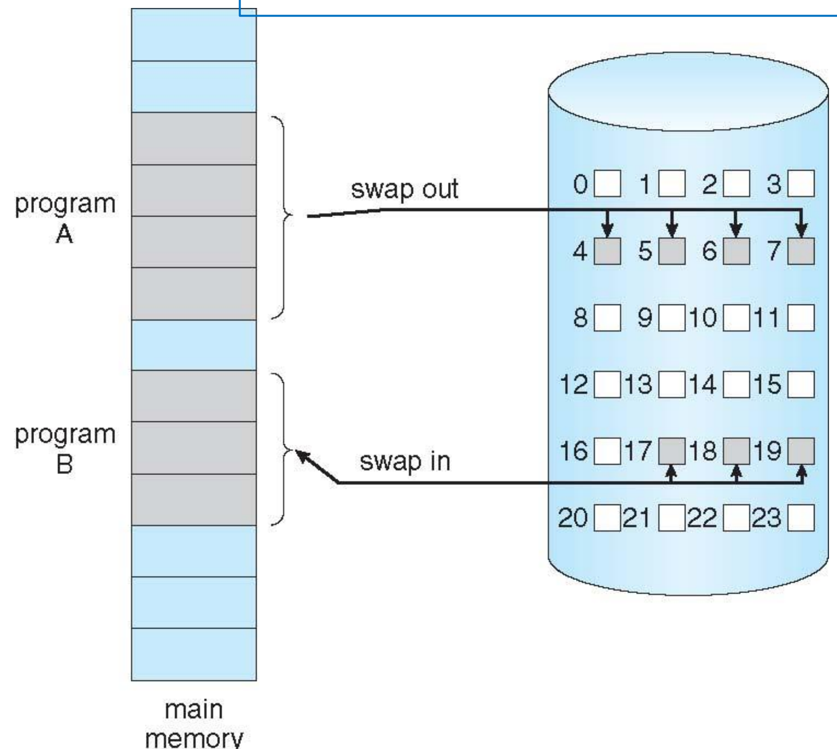
Shared Library Using Virtual Memory



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 ⇒ 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**

- Demand Paging: 필요한 페이지만 메모리에 올리고, 필요하지 않은 페이지는 디스크에 저장하여 메모리를 절약하는 방법



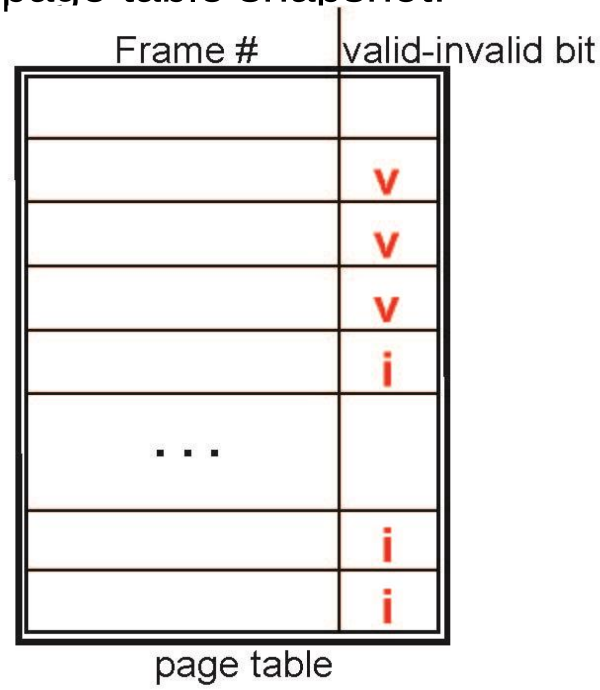
- Q: Page단위로 swap in/out될 때, 필요한 page가 main memory에 있는지 빠르게 알 수 있는가?

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

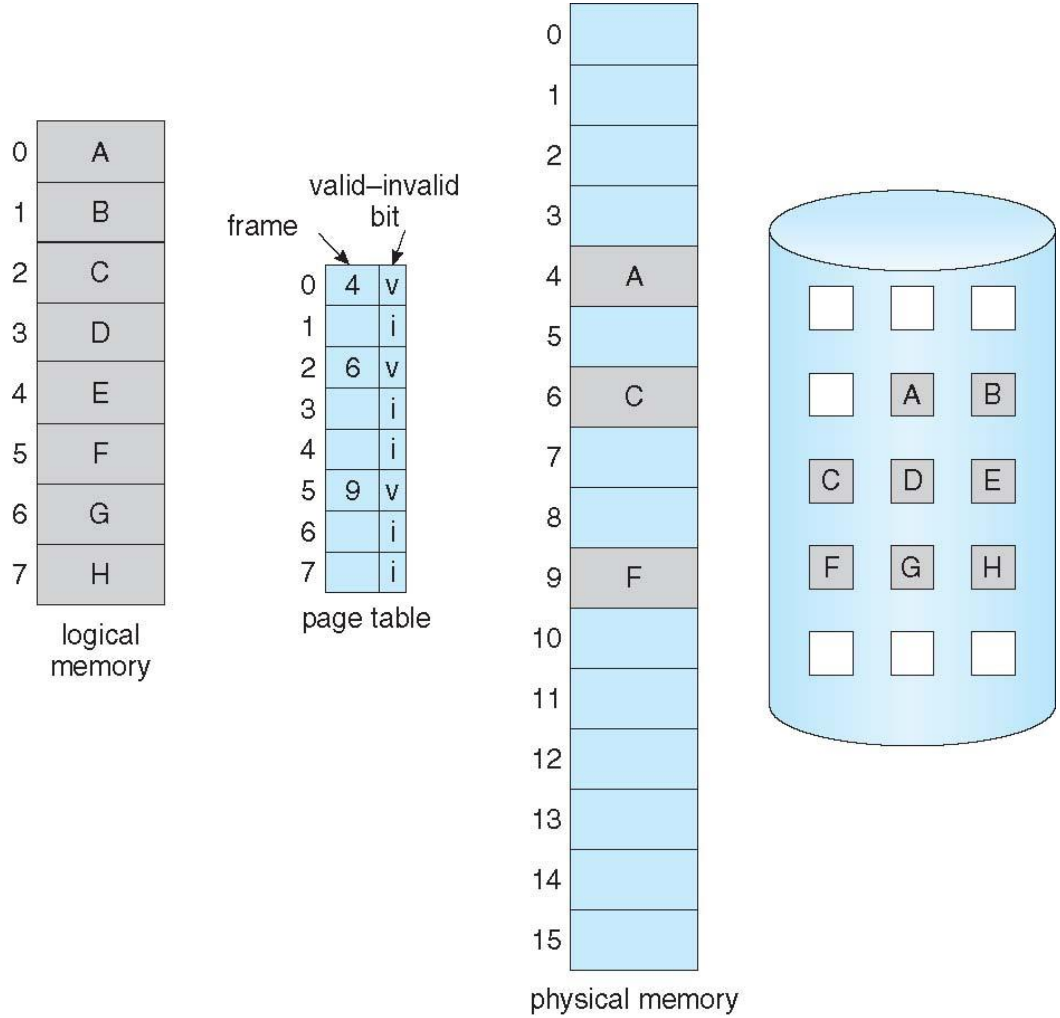
Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated ($\mathbf{v} \Rightarrow$ in-memory – **memory resident**, $\mathbf{i} \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



- 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



Page Fault

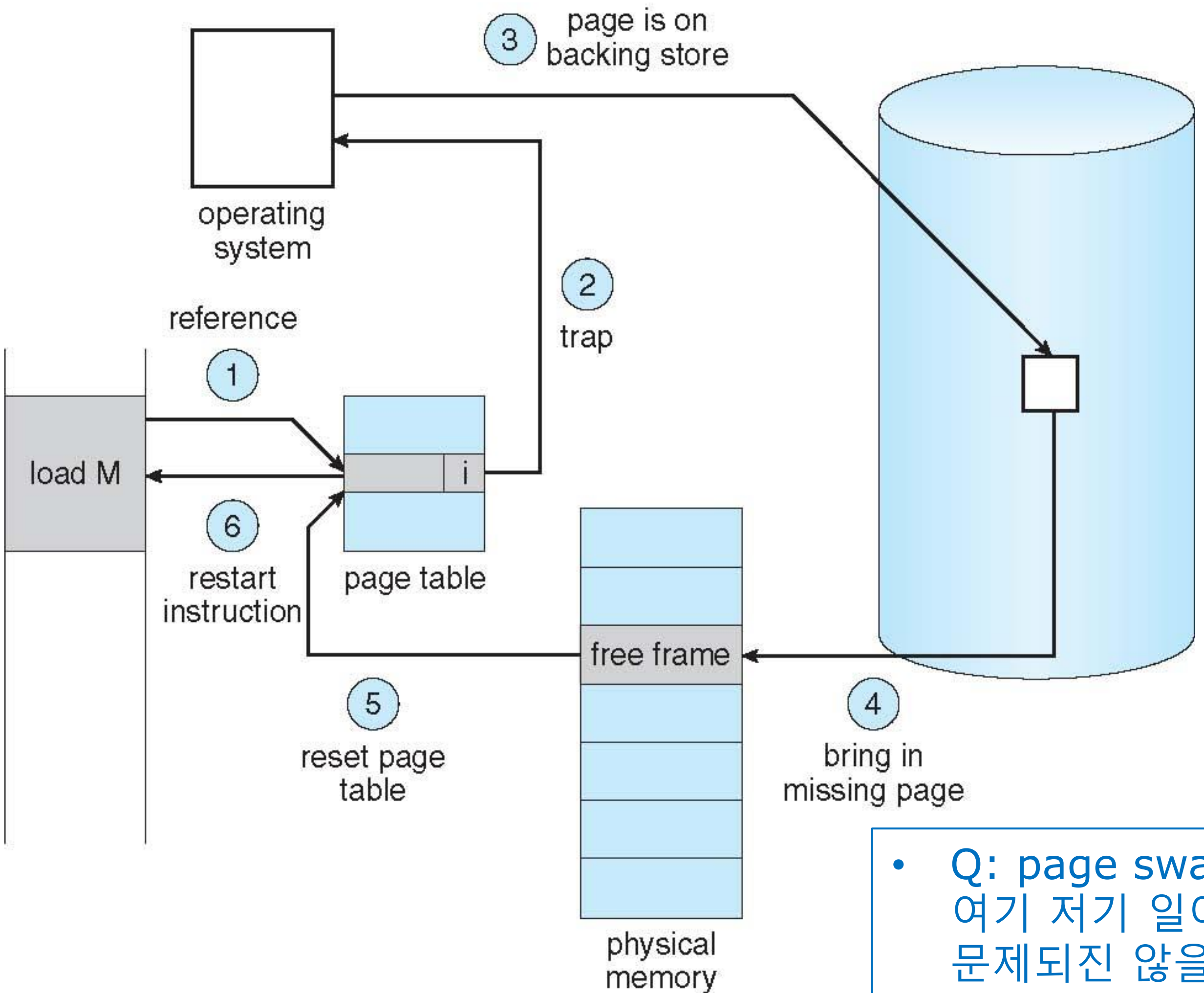
- If there is a reference to a page, first reference to that page will trap to operating system:

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find 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

- 본 과정을 뒷페이지 그림으로 설명

Steps in Handling a Page Fault



- Q: page swap in/out이 여기 저기 일어나면 효율이 문제되진 않을지?
  > locality of reference

Aspects of Demand Paging

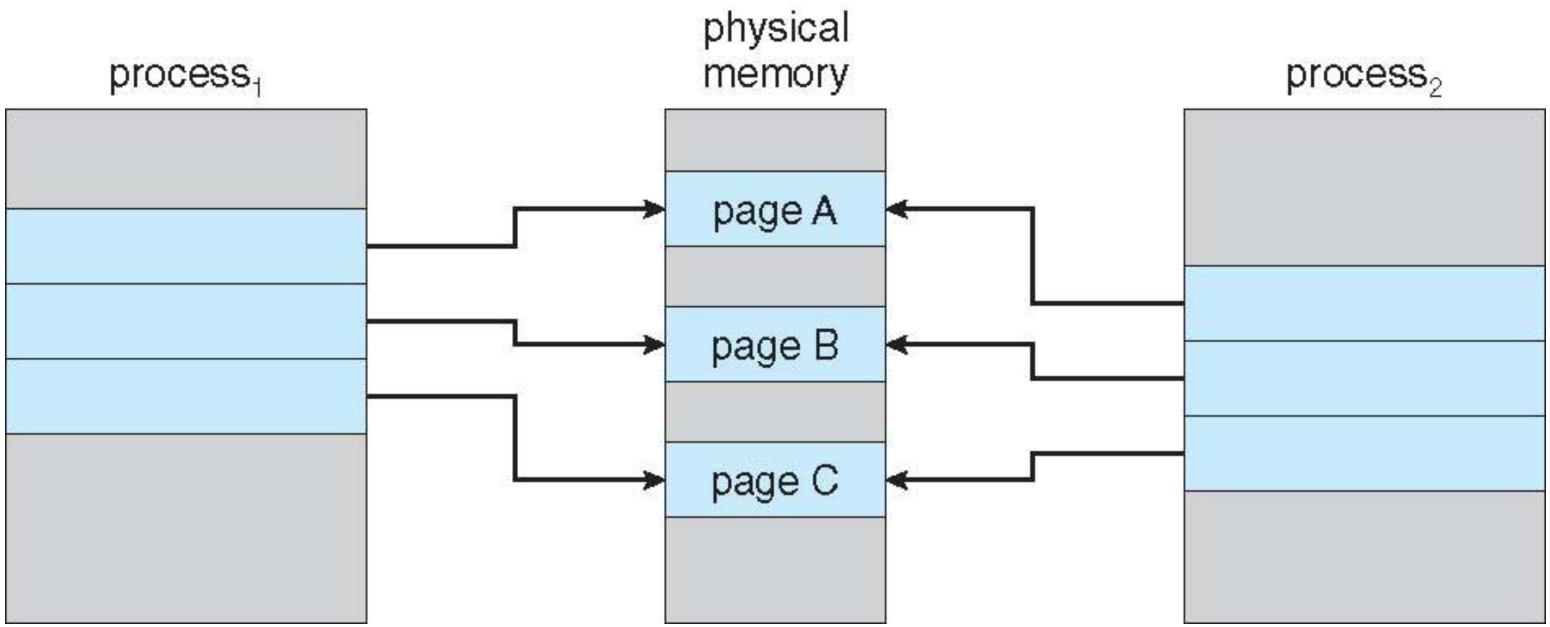
- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging** (처음엔 모두 page fault임)
- Actually, a given instruction could access 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** (메모리로의 데이터 접근이 지역성을 띔)
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

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

- Q: fork()로 생성된 parent/child process들이 page 요구가 많을 때 무언가 효율적인 방법은? > COW

Before Process 1 Modifies Page C

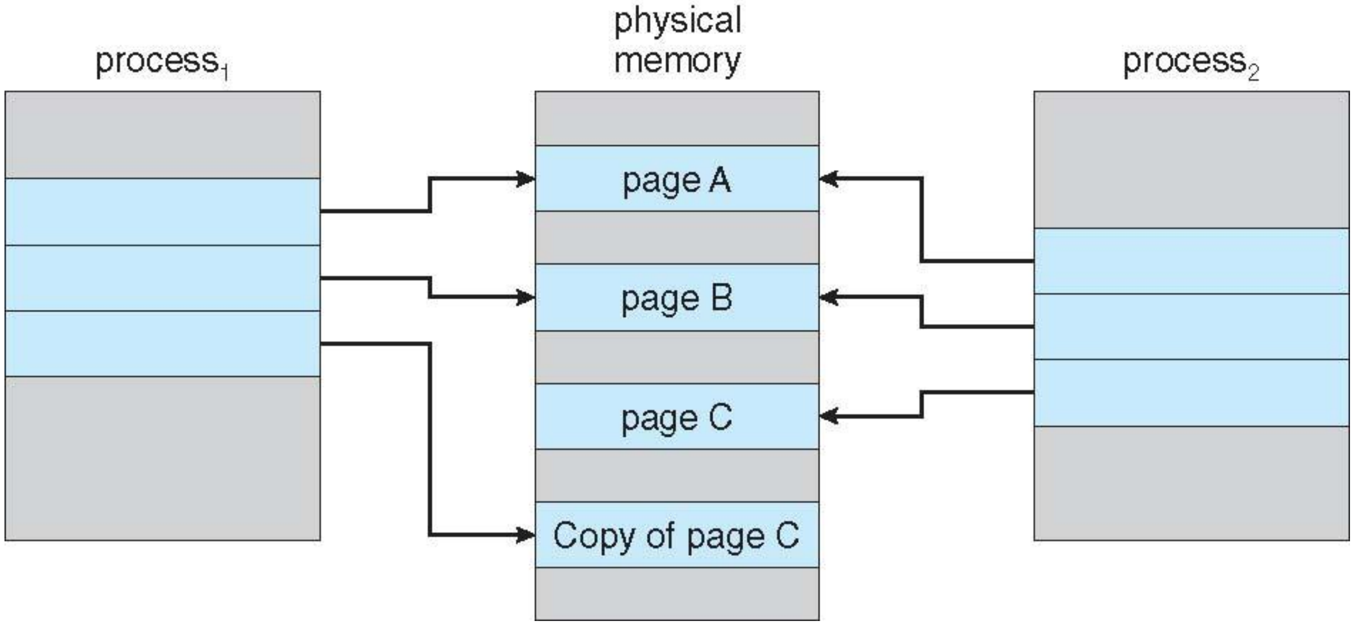


- Q: 메모리를 공유하던 child process가 page 내용을 수정하면?
- > 기본은 read only mode

Multimedia Computing Systems Lab.

SUNG KYUN KWAN UNIVERSITY(SKKU)

After Process 1 Modifies Page C



What Happens if There is no Free Frame?

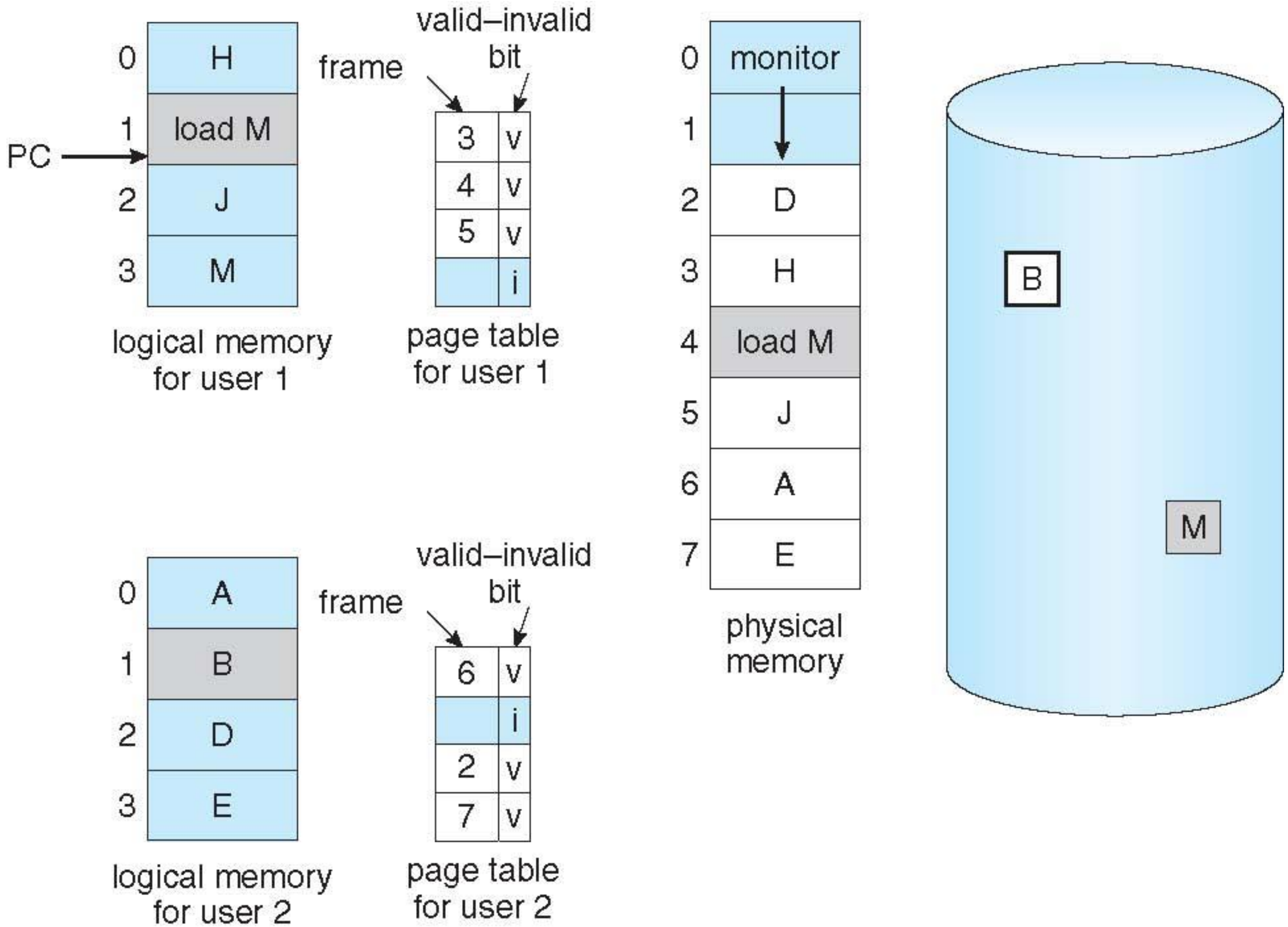
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - <u>Algorithm – terminate? swap out? replace the page?</u>
  - <u>Performance – want an algorithm which will result in minimum number of page faults</u>
- Same page may be brought into memory several times

Page Replacement

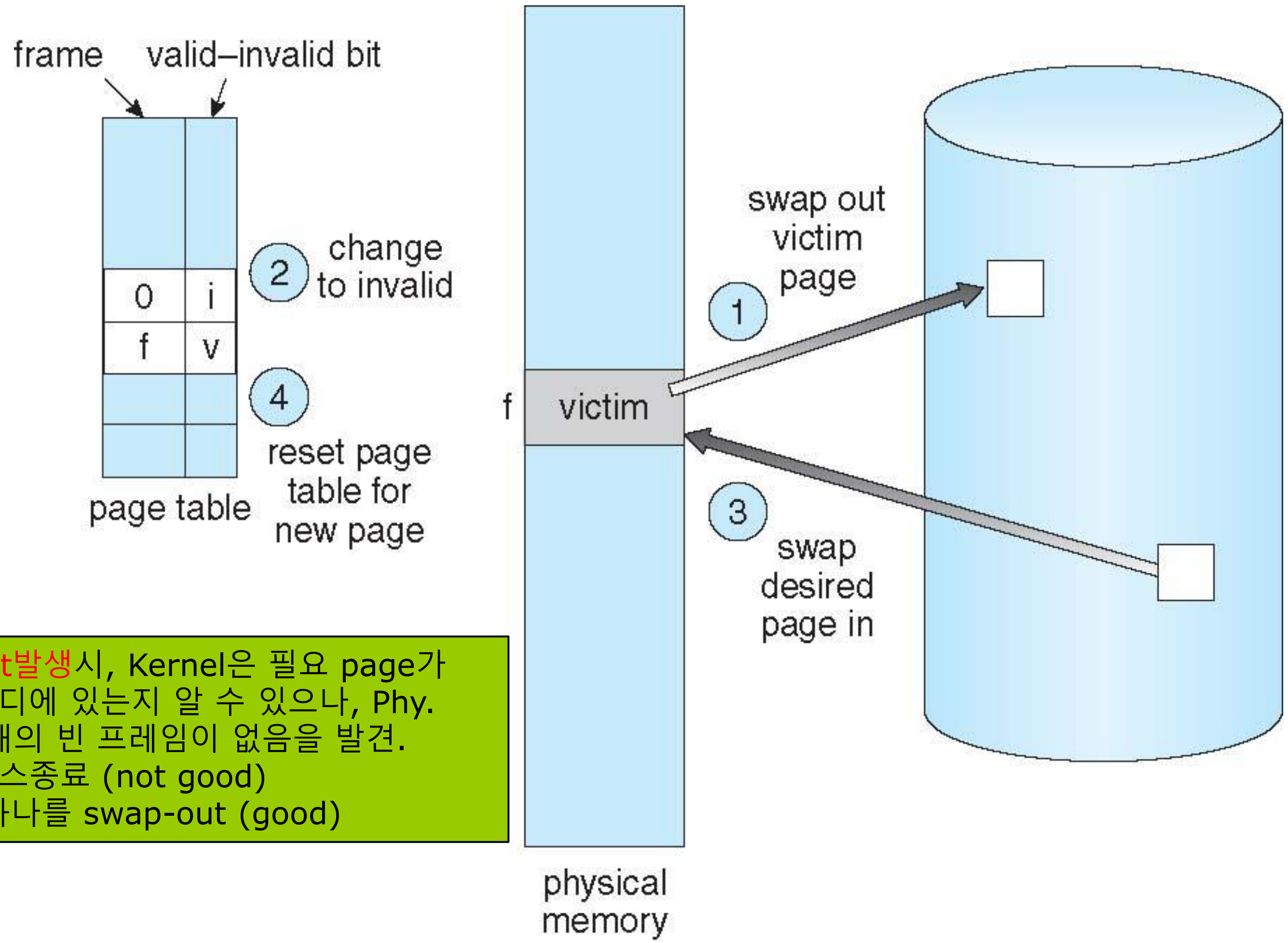
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

- Q: Page swap out을 할 때, storage에 write해야 하는데 시간이 걸리는 문제를 해결하려면?
- >dirty bit

Need For Page Replacement



Page Replacement



Page fault발생시, Kernel은 필요 page가 Disk의 어디에 있는지 알 수 있으나, Phy. Memory내의 빈 프레임이 없음을 발견.
(1)프로세스종료 (not good)
(2)Page하나를 swap-out (good)

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

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 (Effective Address Time)

- Q: page swap out/in으로 2번의 copy I/O가 필요하여 성능 문제. 해결은?

CSLab Multimedia Computing Systems Lab.

SUNG KYUN KWAN UNIVERSITY(SKKU)

Q&A

- Question: Page fault를 대응하는 I/O 타임이 너무 긴 문제 – 2번의 I/O필요.

- Ans:
  - CPU가 Page내의 어떤 정보를 update하게 되면, Page 내용이 변경되는 것임. 이를 나타내는 것이 변경비트 (Modify bit or dirty bit) - HW에 존재
  - 만일 Victim Page의 Dirty bit이 true면 메모리의 내용이 원본 디스크상의 내용과 달라짐을 의미. 이 경우 현재 내용을 Disk에 저장할 필요 있음. 하지만, false면 굳이 swap out으로 Disk에 저장할 필요 없음.
  - 따라서, 경우에 따라 Page fault의 I/O시간을 ½로 줄일 수 있음

MCSLab Multimedia Computing Systems Lab.

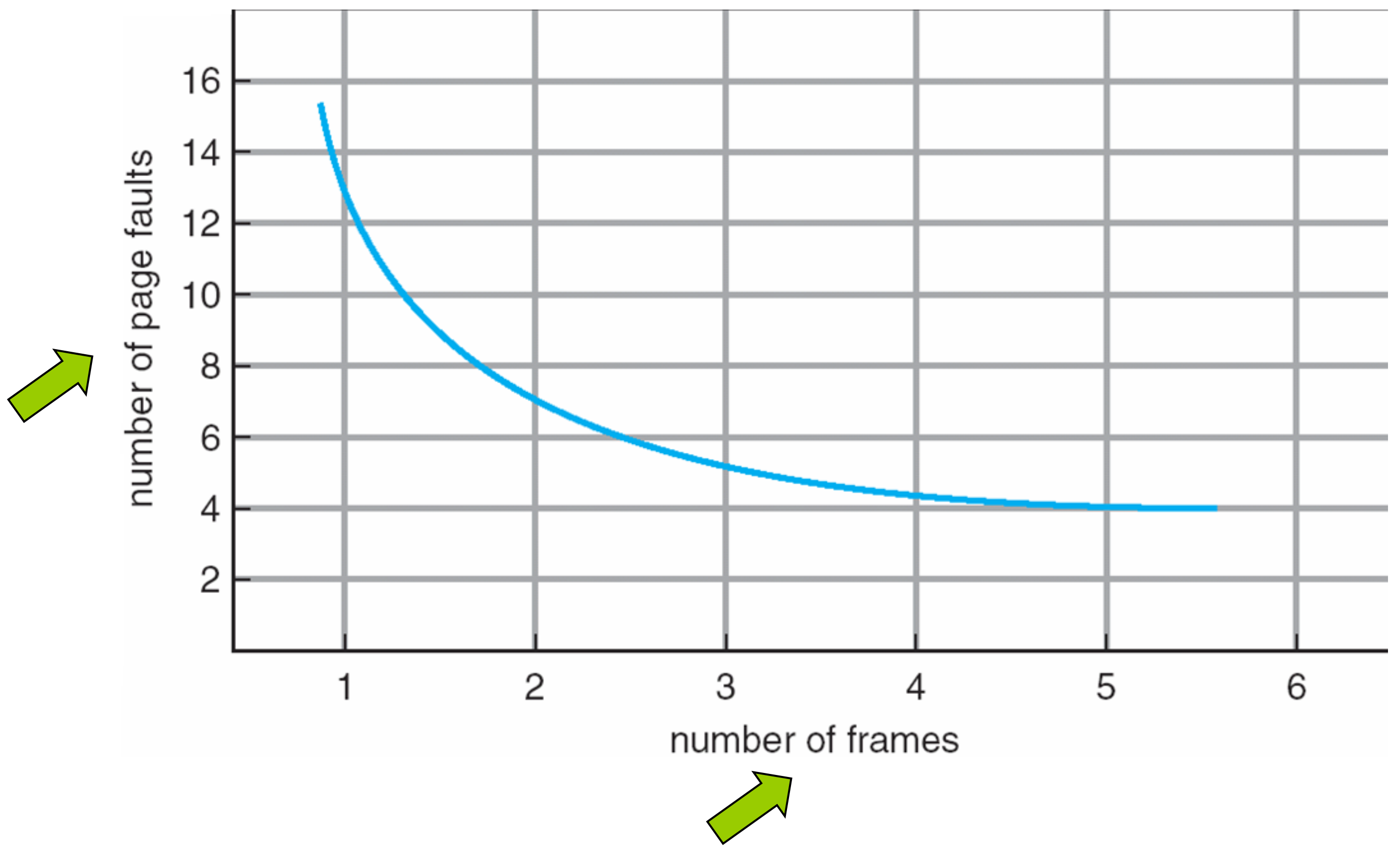
SUNG KYUN KWAN UNIVERSITY(SKKU)

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

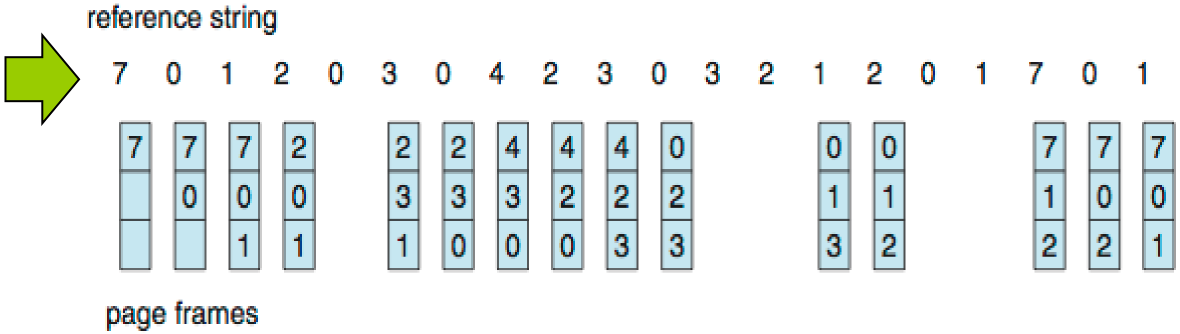
Graph of Page Faults Versus The Number of Frames



#Frames in Phy. memory

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)



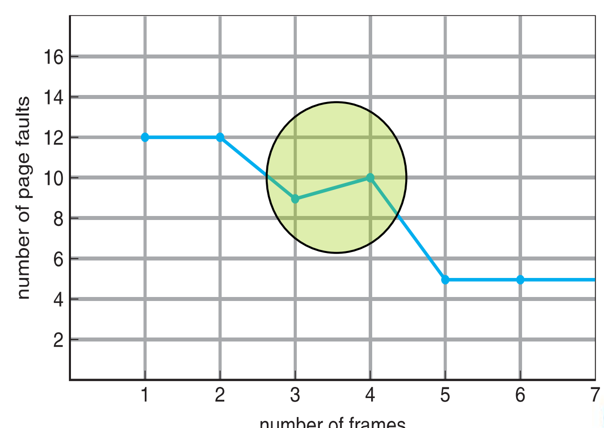
Replace oldest page slot

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

1,2,3,4,1,2,5,1,2,3,4,5
3 frames
1, 2, 3
4, 2, 3
4, 1, 3
4, 1, 2
5, 1, 2
5, 3, 2
5, 3, 4
9 times

4 frames
1, 2, 3, 4
5, 2, 3, 4
5, 1, 3, 4
5, 1, 2, 4
5, 1, 2, 3
4, 1, 2, 3
4, 5, 2, 3
10 times



더 많은 frame에 도리어 page fault가 느는 경우

CSLab Multimedia Computing Systems Lab.

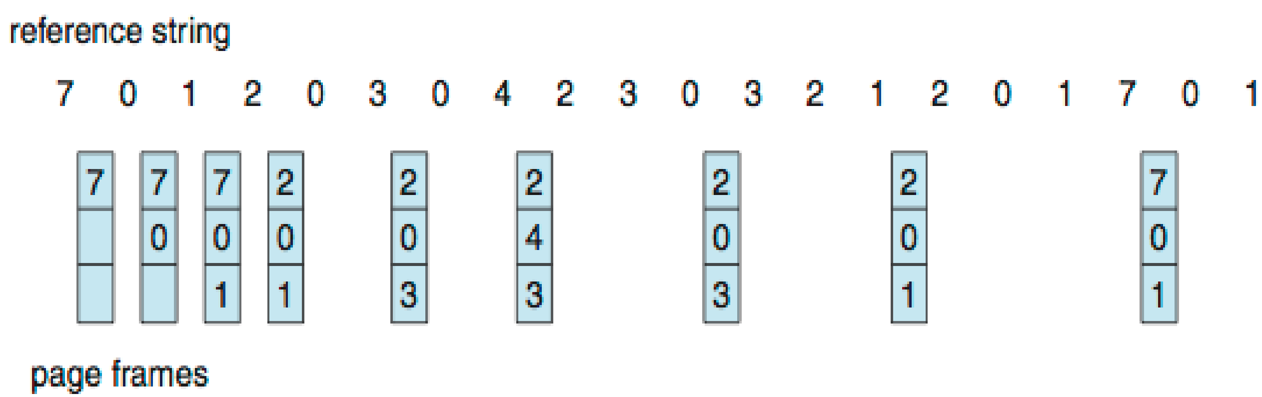
SUNGKYUNKWAN UNIVERSITY(SKKU)

Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can’t read the future
- Used for measuring how well your algorithm performs

앞으로 가장 오랫동안 사용되지
않을 페이지를 찾아 교체

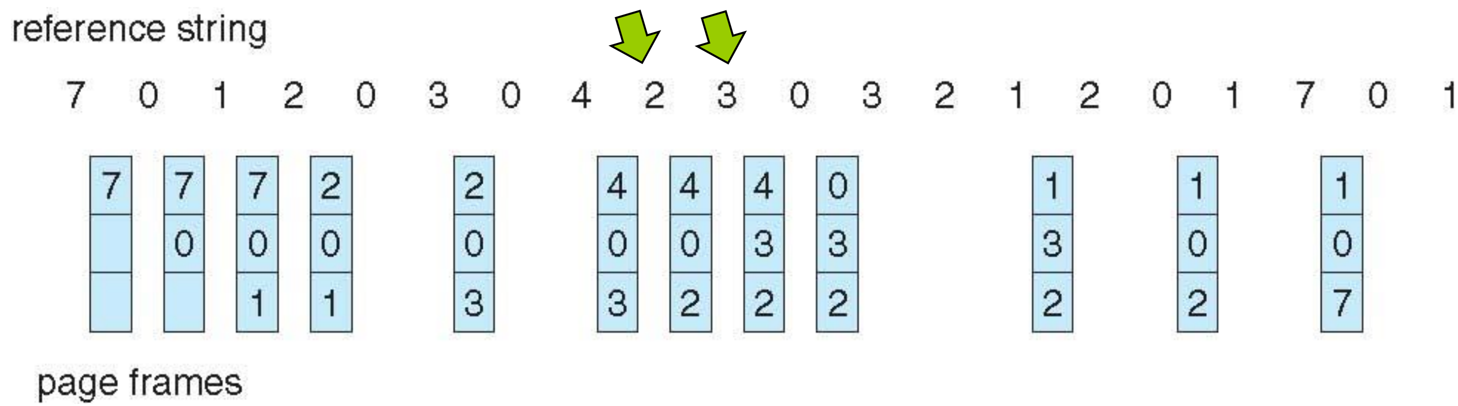
닥터 스트레인저 필요



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

가장 오랫동안 사용되지 않은 페이지를 교체



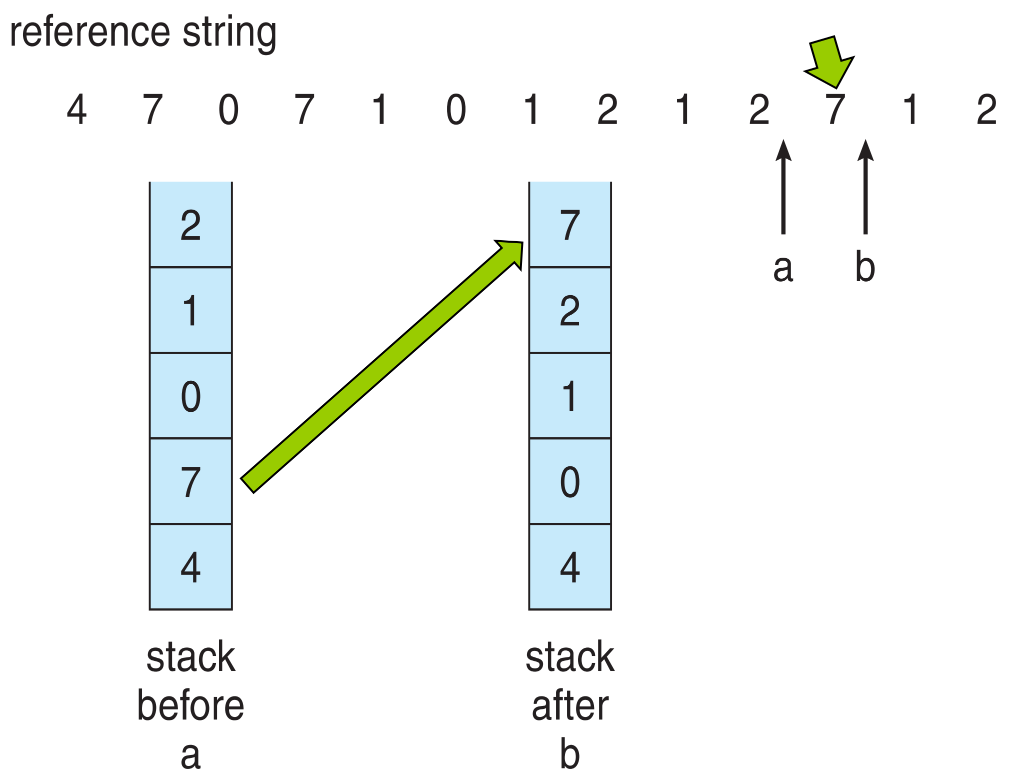
- 12 faults – better than FIFO but worse than OPT
- <u>Generally good algorithm and frequently used</u>
- But <u>how to implement?</u>

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

LRU는 HW지원 필요 2가지 방법가능
- Counters
- Stack – 페이지 참조될 때 마다 Page 번호가 Stack 중간에서 제거 되어 top에 놓이게 함. -> Bottom이 가장 오랫동안 안쓴 페이지. : Double linked list로 구현

Use Of A Stack to Record Most Recent Page References



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common

둘다 잘 안쓰임

- **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

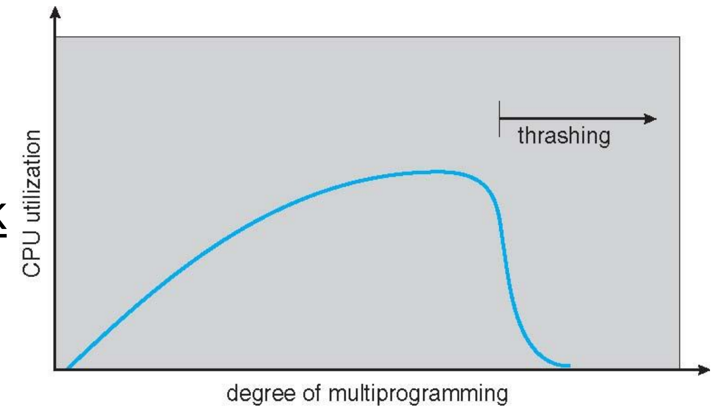
LFU 즉, 참조회수가 가장 작은 페이지를 교체: 활발한 페이지는 참조회수가 많아질 것이란 생각. 하지만, 어떤 프로세스가 초기에 한 페이지 집중적 사용 후 그 후로 사용 잘 안하면 문제. 계속 메모리에 머물게 됨.
해결책> 참조회수를 일정시간마다 right shift하여 지수적으로 영향력 감소시킴 (일종의 aging의 HW사용한 변형)

CSLab Multimedia Computing Systems Lab.

SUNG KYUN KWAN UNIVERSITY(SKKU)

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

요약(1/2)

- 가상메모리는 프로세스 전체가 메모리 내에 올라오지 않더라도 실행이 가능하도록 하는 기법
  - 사용자 프로그램이 물리 메모리 보다 커도 됨
- 각 프로세스들은 공유 라이브러리가 자신의 일부라고 생각하지만, 실제로는 라이브러리가 존재하는 물리 메모리 페이지들은 모든 프로세스들에게 공유 (라이브러리는 읽기만이 허용되는 상태)
- 요구페이징(Demand Paging) – Swapping 기법과 유사. 통상 프로세스는 보조 메모리(디스크)에 존재하나, 프로세스를 실행하고 싶으면 메모리로 읽어들임(Swap in). 단, 필요한 Page 만 읽어들임.
- Page Table의 유효비트/무효비트는 해당페이지가 메모리에 있다 없다를 의미.
- 메모리에 올라와 있지 않는 페이지에 프로세스가 접근하면, Page fault trap발생 -> Paging HW가 주소변환 과정에서 무효 비트 발견하고 OS에 trap. 유효한 참조인데 아직 Page가 메모리에 없다면 디스크로 부터 가져옴.

요약(2/2)

- Locality of Reference: 통상 프로세스들을 분석하면, 프로그램의 어느 한 특정 작은 부분만 한동안 집중적으로 참조 -> Demand Paging (필요한 부분만 Paging)은 좋은 성과
- Copy-on-Write: 예로 부모/자식 프로세스처럼 fork() 때 주소공간을 초기에 복사하는 경우, 자식들은 부모로 부터 Page들을 복사할 필요 없이 share해서 쓰다가 data를 write해서 메모리가 변경이 되는 때에만 Copy-on-write로 해당 페이지를 복사하여 write. : vfork() 등이 존재
- Page Replacement: 만약 빈 프레임이 없다면, 현재 사용되지 않는 프레임을 찾아서 그것을 비움. 그 프레임의 내용을 swap공간에 쓰고, 그 페이지가 메모리에 더 이상 존재하지 않음을 Page table 수정하여 프레임을 빈 상태로 만듦. 그 후 빈 프레임에 page fault난 page를 저장
- Page fault의 I/O를 줄이는 법: HW기반의 Dirty-bit사용 (변경 유무 확인)