Virtual Memory III

Jo, Heeseung

Today's Topics

What if the physical memory becomes full?

- Page replacement algorithms

How to manage memory among competing processes?

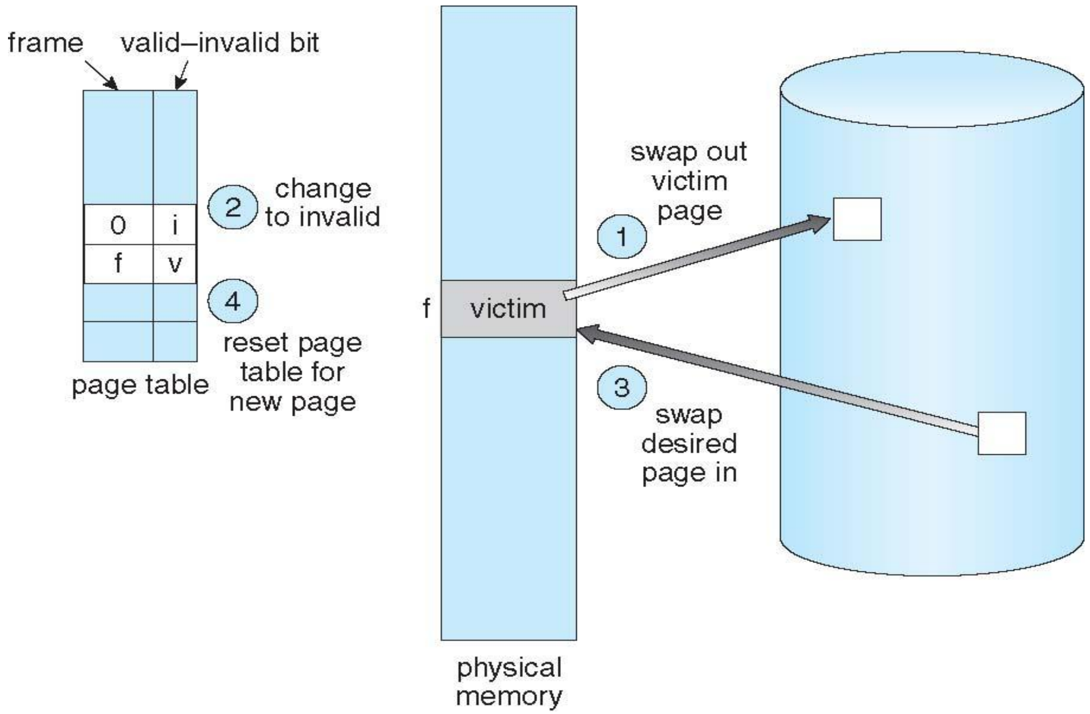
Advanced virtual memory techniques

- Shared memory
- Copy on write
- Memory-mapped files

Page Replacement (1)

Page replacement

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of memory
- At some point, the process has used all of the page frames it is allowed to use (No free frame)
- When this happens, the OS must replace a page for each page faulted in
  - It must evict a page to free up a page frame
- The page replacement algorithm determines how this is done



Page Replacement (2)

Evicting the best page

- The goal of the replacement algorithm is to reduce the fault rate by selecting the best victim page to remove
- The best page to evict is the one never touched again
  - As process will never again fault on it
- "Never" is a long time, so picking the page closest to "never" is the next best thing

Belady's proof

- Evicting the page that won't be used for the longest period of time minimizes the number of page faults

가장 오랜 기간동안 쓰이지 않은 애를 Evicting.

Belady's Algorithm

Optimal page replacement

- Replace the page that will not be used for the longest time in the future
- The lowest fault rate

Problem

- Have to predict the future
- Why is Belady's useful? – Use it as a yardstick!
  - Compare other algorithms with the optimal to gauge room for improvement
  - If optimal is not much better, then algorithm is pretty good
  - Otherwise, algorithm could be better
  - Lower bound depends on workload

FIFO (1)

First-In First-Out

- Obvious and simple to implement
  - Maintain a list of pages in order they were paged in
  - On replacement, evict the one brought in longest time ago
- Why might this be good?
  - Maybe the one brought in the longest ago is not being used
- Why might this be bad?
  - Maybe, it's not the case
  - We don't have any information either way
- FIFO suffers from "Belady's Anomaly"
  - The fault rate might increase when the algorithm is given more memory

p.m ↑ 빈공간 ↑ page fault ?

FIFO (2)

Example: Belady's anomaly

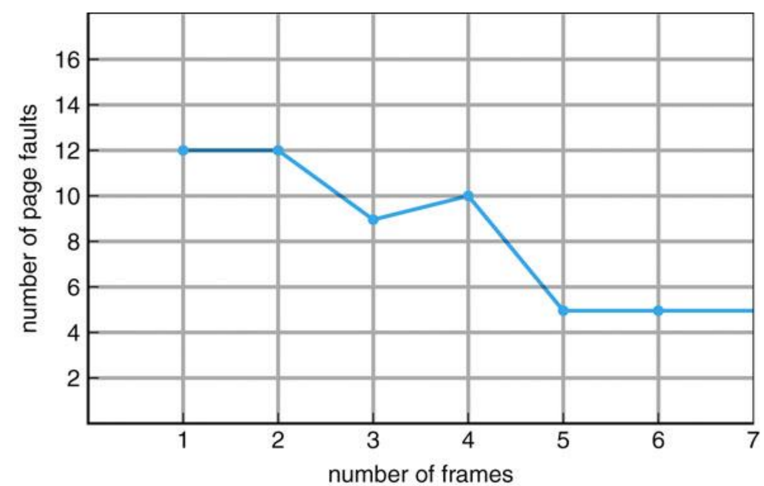
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames: 9 faults

page fault 후 1이 들어옴.
↳ p.m에 없어서

old	1	2	3	4	1	2	5
	2	3	4	1	2	5	3
new	3	4	1	2	5	3	4

- 4 frames: 10 faults

old	1	2	3	4	5	1	2
	2	3	4	5	1	2	3
	3	4	5	1	2	3	4
new	4	5	1	2	3	4	5



LRU (1)

Least Recently Used

- LRU uses reference information for better replacement decision
  - Idea: past experience gives us a guess of future behavior
  - On replacement, 퇴거하다 evict the page that has not been used for the longest time in the past
  - LRU looks at the past, Belady's wants to look at future
- Implementation
  - Counter implementation: put a timestamp
  - Stack implementation: maintain a stack
- We need an approximation (heuristic)

근사

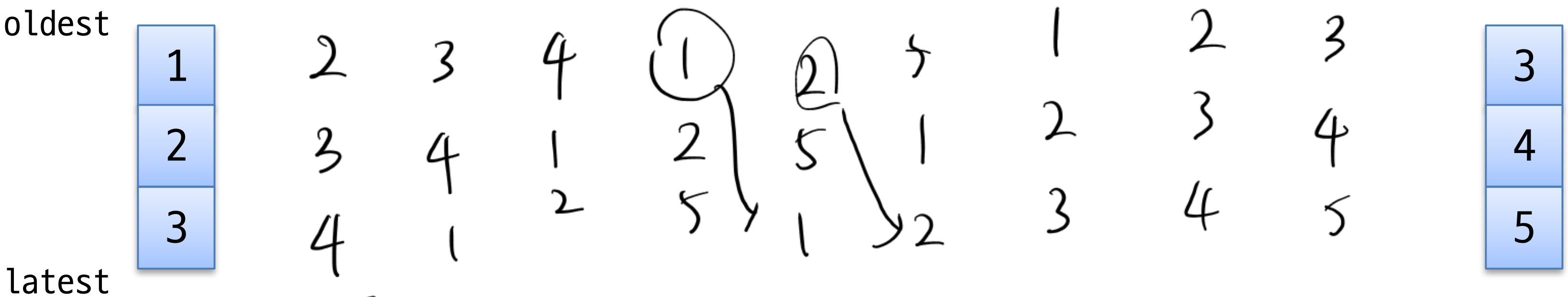
FIFO → 가장 오래전에 들어온 애를 뺌.

LRU → 가장 오랫동안 안 쓰인 애를 뺌.

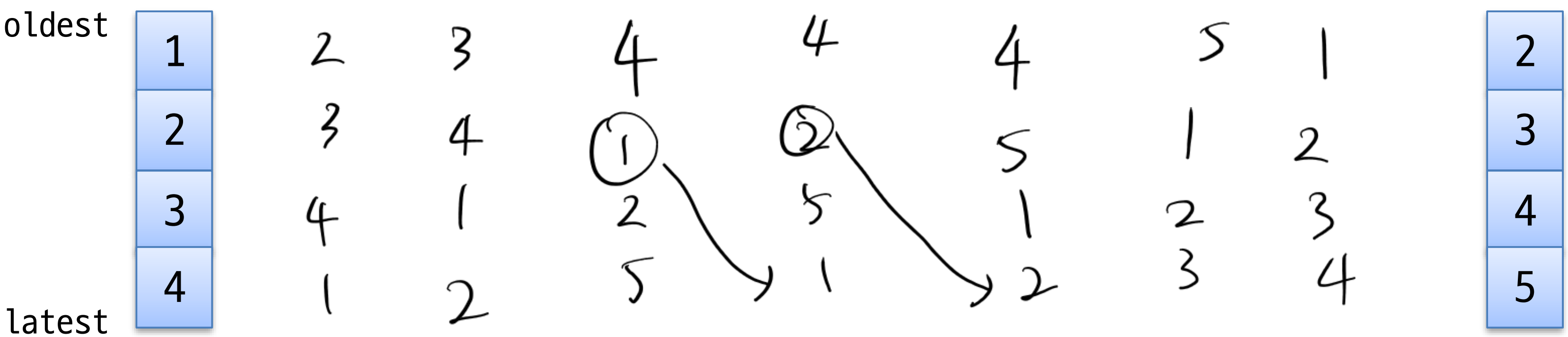
LRU (2)

Example:

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames: ?? faults 10



- 4 frames: ?? faults 8



LRU (3)

Approximating LRU

	1	1	1	2	20
counter	V	R	M	Prot	Page Frame Number (PFN)

- Many LRU approximations use the PTE reference (R) bit
  - R bit is set whenever the page is referenced (read or written)

M bit = modify (write) 됐는지 아닌지.

- Counter-based approach
  - Keep a counter for each page
  - At regular intervals, for every page, do:
    - If R = 0, increment the counter (hasn't been used)
    - If R = 1, zero the counter (has been used)
    - Zero the R bit

```
if (R ==0)
    counter++;
else if (R== 1)
    Counter = 0;
```

  - The counter will contain the number of intervals
  - The page with the largest counter is the least recently used
  - Memory overhead for every page

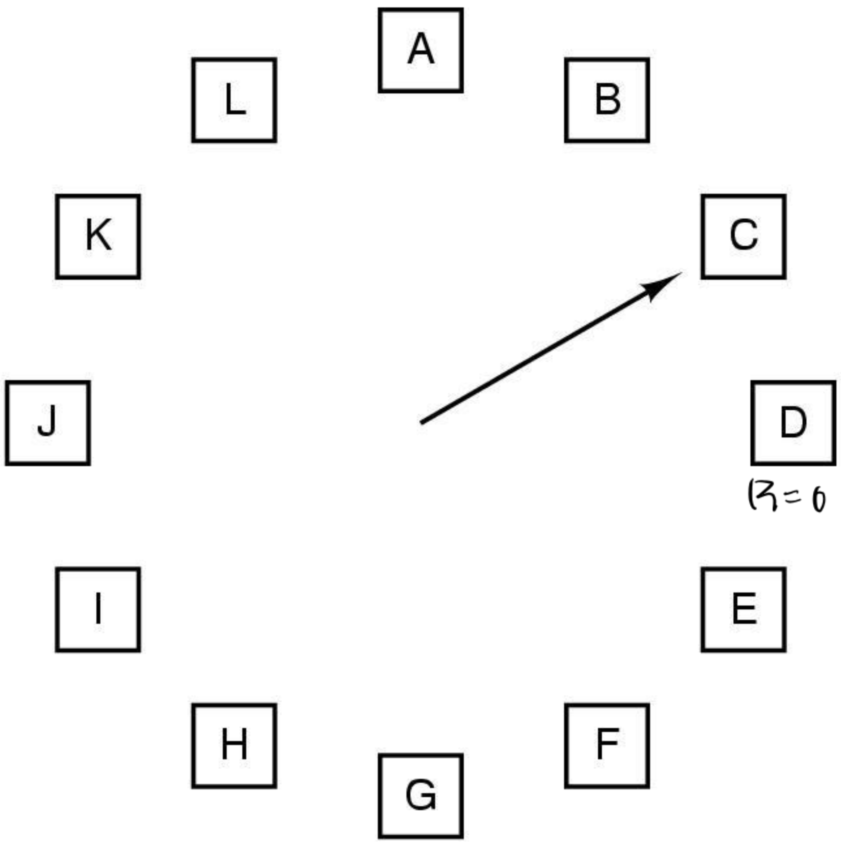
제일 큰 counter 값을 가진 page가 최소 최근에 사용.

- Some architectures don't have a reference bit
  - Can simulate reference bit using the valid bit to induce faults

Second Chance (1)

Second chance or LRU clock

- FIFO with giving a second chance to a recently referenced page
- Arrange all of physical page frames
  in a big circle (clock)

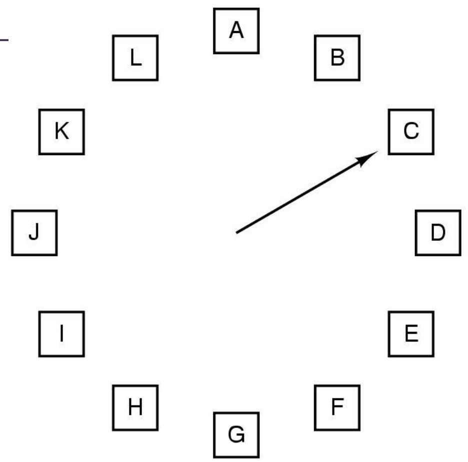


바로 빼지 말고 하나씩 뺌.

When a page fault occurs,
the page the hand is
pointing to is inspected.
The action taken depends
on the R bit:
R = 0: Evict the page
R = 1: Clear R and advance hand

R=1, 기회 1번 제공.

Second Chance (2)



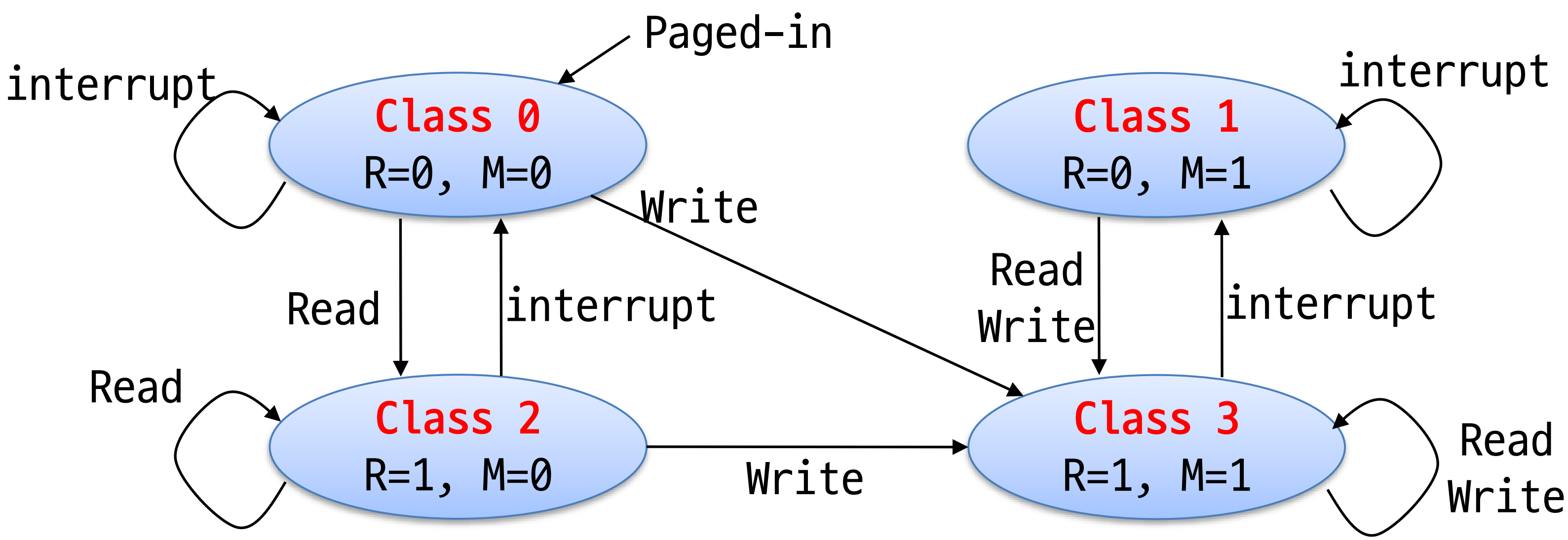
Second chance or LRU clock

- A clock hand is used to select a good LRU candidate
  - Sweep through the pages in circular order like a clock
  - If the R bit is off, it hasn't been used recently and we have a victim
  - If the R bit is on, turn it off and go to next page (second chance)
- Arm moves quickly when pages are needed
  - Low overhead if we have plenty of memory
  - If memory is large, "accuracy" of information degrades

Not Recently Used (1)

NRU or enhanced second chance

- Use R (reference) and M (modify) bits
  - Periodically, (e.g., on each clock interrupt), R is cleared, to distinguish pages that have not been referenced recently from those that have been
  - Considering modification of a page



interrupt : CPU가 프로그램을 실행하고 있을 때, I/O 장치 혹은 다른 예외상황이 발생해서 작업을 중단시키고 이걸 처리하라고 보내는 신호

Not Recently Used (2)

Algorithm

- Removes a page at random from the lowest numbered nonempty class
- It is better to keep a modified page that has not been referenced in at least one clock tick than a clean page
- Used in Macintosh

R, M bit만 check.

Advantages

- Easy to understand
- Moderately efficient to implement
- Gives a performance that, while certainly not optimal, may be adequate

LFU (1)

Counting-based page replacement

- A software counter with each page
- At each clock interrupt, for each page, the R bit is added to the counter
  - The counters denote how often each page has been referenced

Least frequently used (LFU)

- The page with the smallest count will be replaced

Most frequently used (MFU)

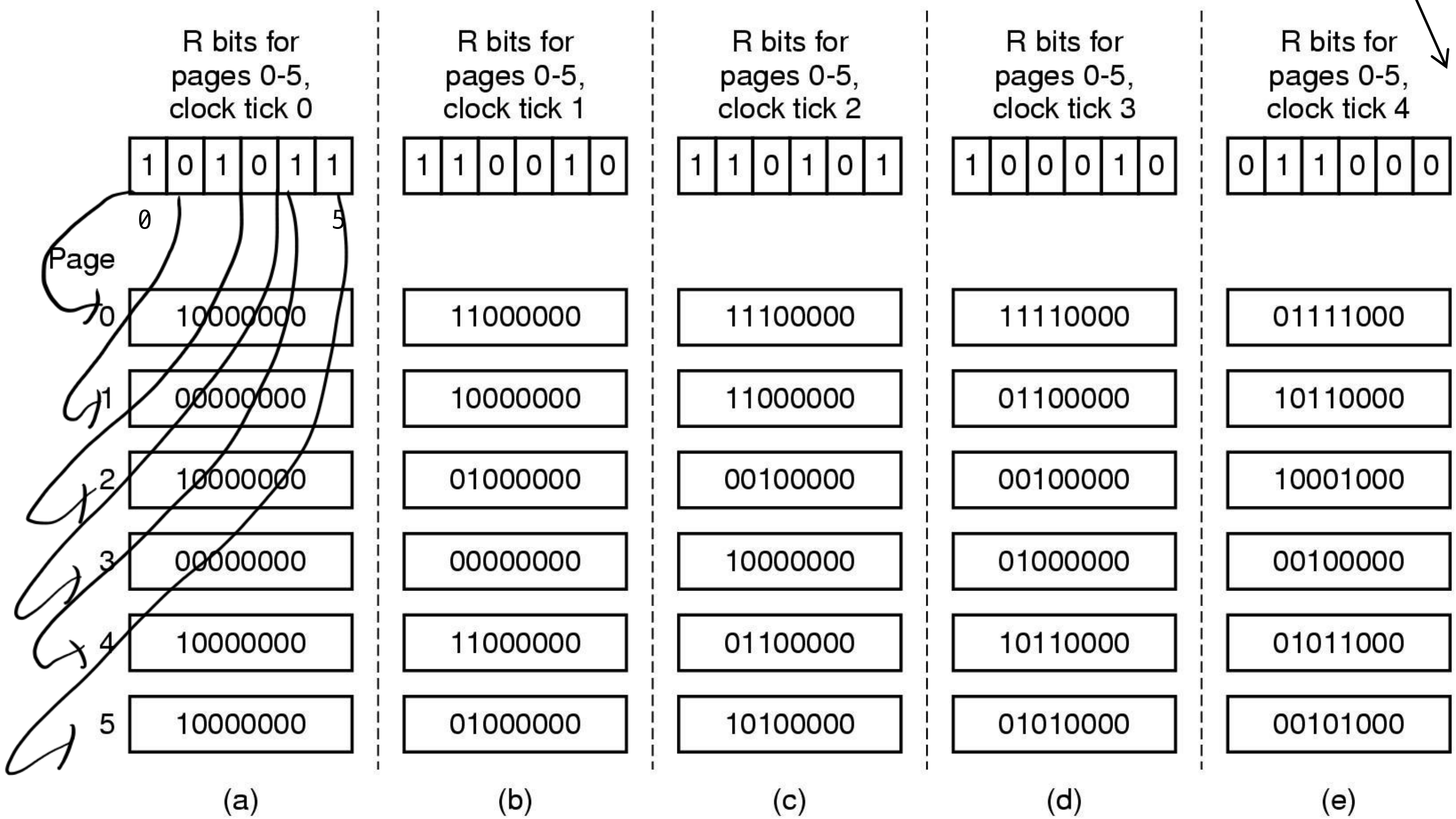
- The page with the largest count will be replaced
- The page with the smallest count was probably just brought in and has yet to be used
- A page may be heavily used during the initial phase of a process, but then is never used again

LFU (2)

At this time,
which one should be evicted?

Aging (counting)

- The counters are shifted right by 1 bit before the R bit is added to the leftmost



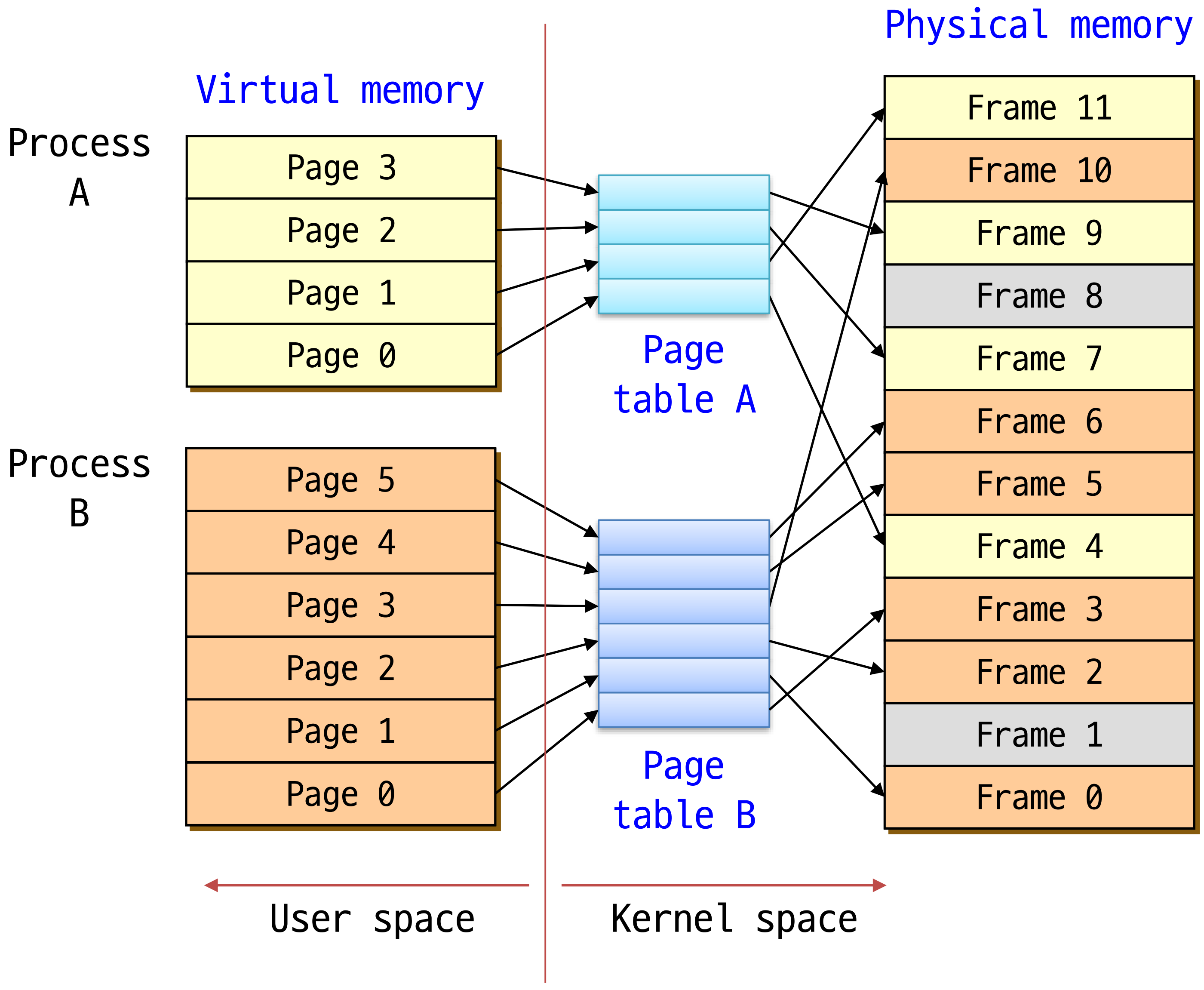
Add a R-bit after shift right the pages.

Allocation of Frames

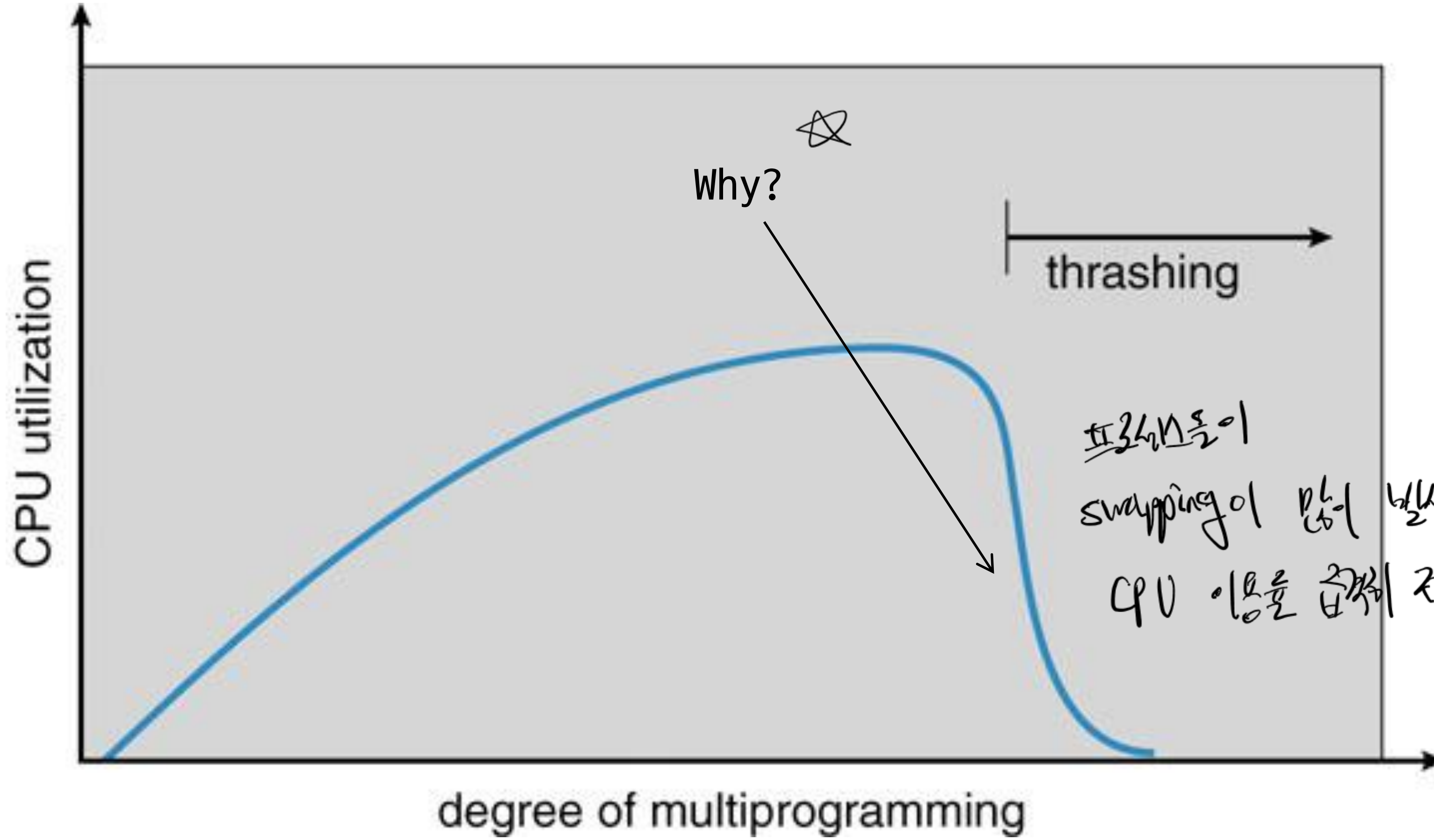
Problem

- In a multiprogramming system, we need a way to allocate physical memory to competing processes
  - What if a victim page belongs to another process?
  - How to determine how much memory to give to each process?
- Fixed space algorithms
  - Each process is given a limit of pages it can use
  - When it reaches its limit, it replaces from its own pages
  - Local replacement: some process may do well, others suffer
- Variable space algorithms
  - Processes' set of pages grows and shrinks dynamically
  - Global replacement: one process can ruin it for the rest (Linux)

Allocation of Frames



Thrashing (1)



프로세스들이
swapping이 많이 발생하여
CPU 이용률 급격히 저하.

Thrashing (2)

Thrashing

- Most of the time is spent by an OS paging data back and forth from disk (heavy page fault)
  - No time is spent doing useful work
  - The system is overcommitted
  - No idea which pages should be in memory to reduce faults
  - Could be that there just isn't enough physical memory for all processes
- Possible solutions
  - Write out all pages of a process
  - Buy more memory

Working Set Model (1)

Working set: The set of pages process currently "needs"

- Peter Denning, 1968
- A working set of a process is used to model the dynamic locality of its memory usage

Definition

- WS(t, w) = { pages P such that P was referenced in the time interval (t, t-w) }
  - t: time
  - w: working set window size (measured in page references)
- A page is in the working set only if it was referenced in the last w references

Working Set Model (2)

Working set size (WSS)

- The number of pages in the working set
  = The number of pages referenced in the interval (t, t-w)
- The working set size changes with program locality
  - During periods of poor locality, more pages are referenced
  - Within that period of time, the working set size is larger
- Intuitively, working set must be in memory to prevent heavy faulting (thrashing)

Controlling the degree of multiprogramming based on the working set:

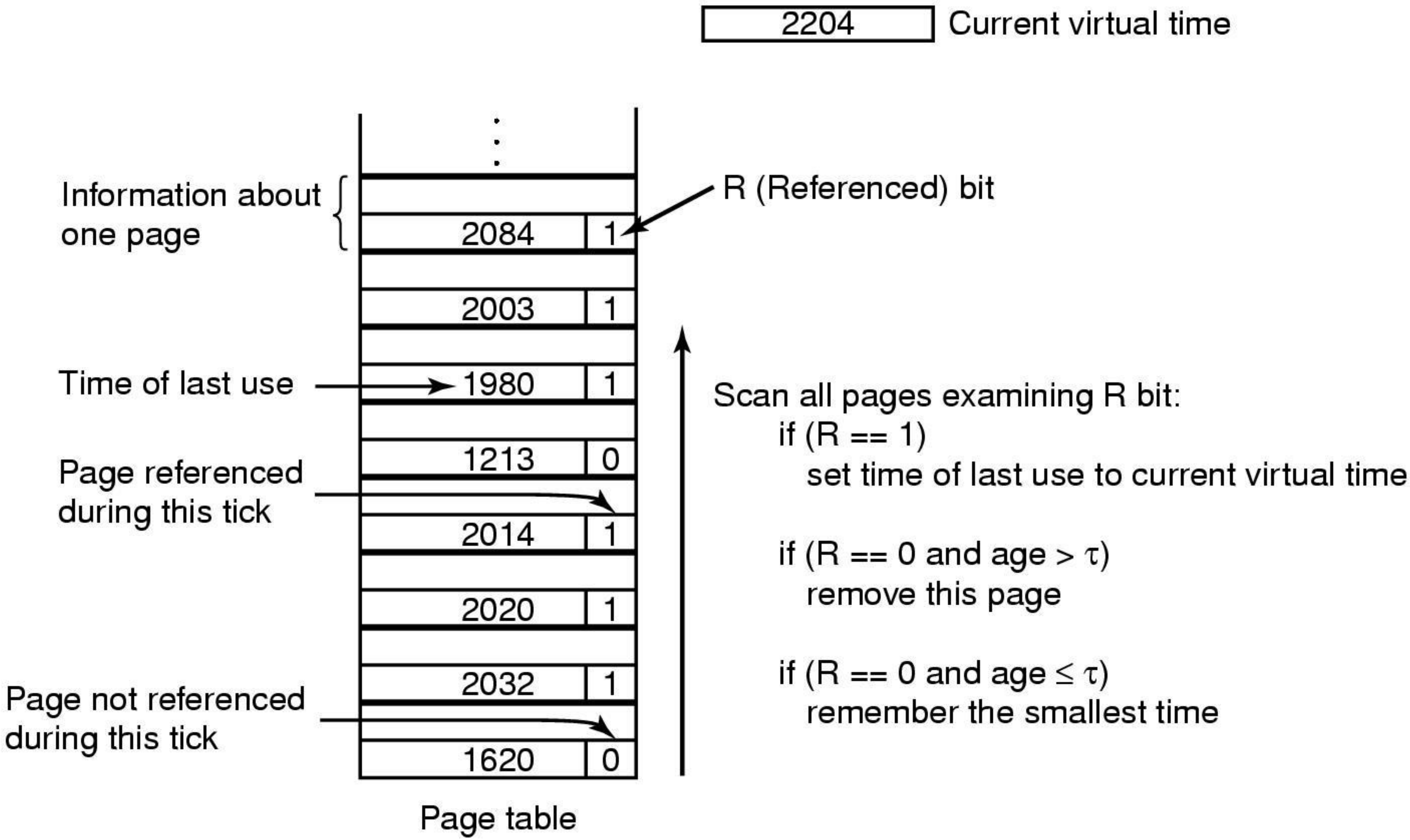
- Associate parameter WSS with each process
- If the sum of WSS exceeds the total number of frames, suspend a process
- Only allow a process to start if its WSS still fits in memory

Working Set Model (3)

Working set page replacement

- Maintaining the set of pages touched in the last k references
- Approximate the working set
  - Measured using the current virtual time: the amount of CPU time a process has actually used
- Find a page that is not in the working set and evict it
  - Associate the "Time of last use (Tlast)" field in each PTE
  - A periodic clock interrupt clears the R bit
  - On every page fault, the page table is scanned to look for a suitable page to evict
  - If R = 1, timestamp the current virtual time (Tlast = Tcurrent)
  - If R = 0 and (Tcurrent – Tlast) > t, evict the page (t: windows size)
  - If R = 0 and (Tcurrent – Tlast) < t, do nothing

Working Set Model (4)



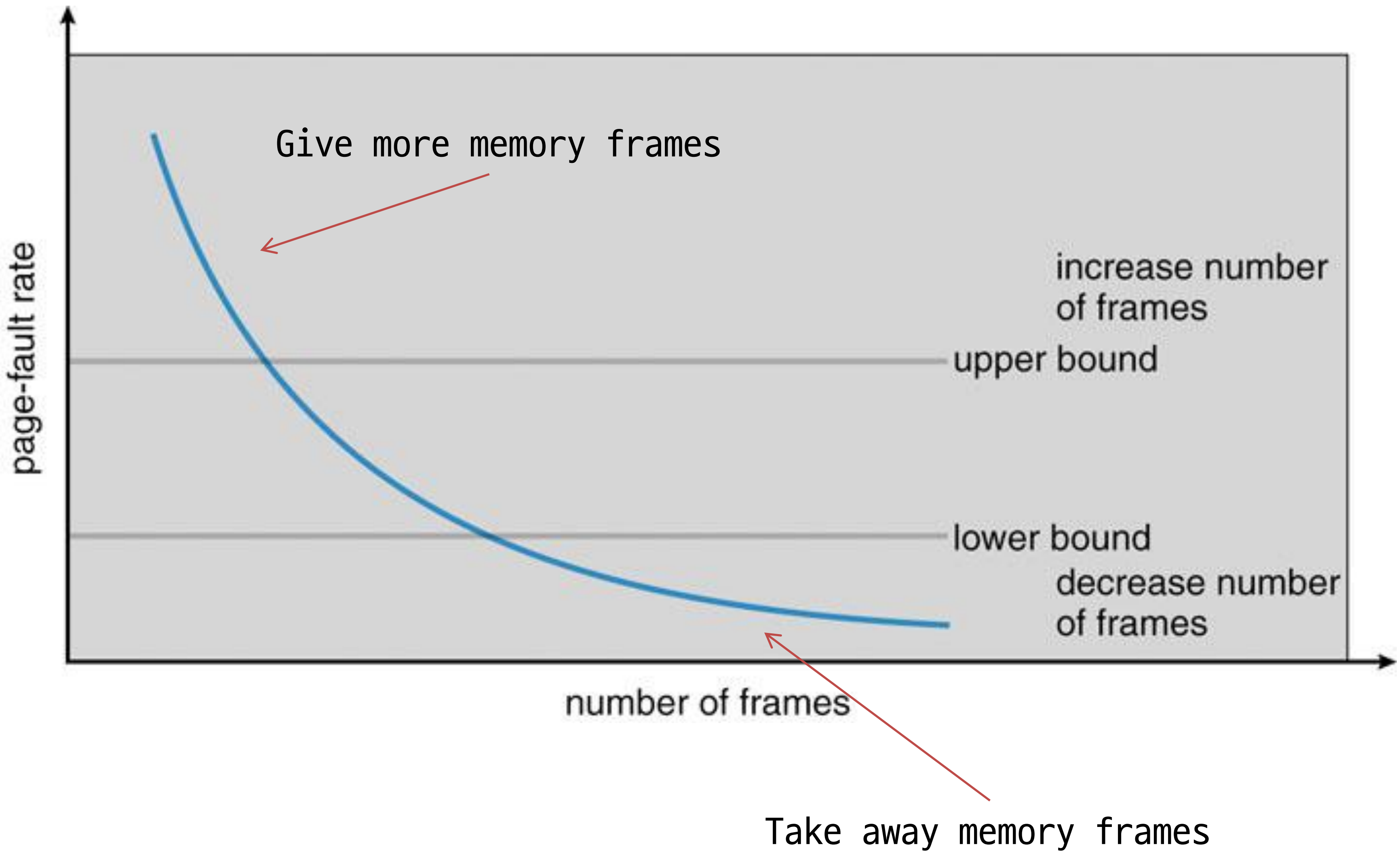
Let's assume t is 600. Then, which page should be evicted?

PFF (1)

Page Fault Frequency

- Monitor the fault rate for each process
- If the fault rate is above a high threshold(upper bound)
  - Give it more memory, so that it faults less
  - But not always valid - FIFO, Belady's anomaly
- If the fault rate is below a low threshold(lower bound)
  - Take away memory
- If the PFF increases and no free frames are available
  - Select some process and suspend it

PFF (2)


Give more memory frames
increase number of frames
upper bound
lower bound
decrease number of frames
page-fault rate
number of frames
Take away memory frames

Advanced VM Functionality

Virtual memory tricks

- Shared memory
- Copy on write
- Memory-mapped files

https://velog.io/@anjaekk/OS-Thrashing

Shared Memory (1)

Shared memory

- Private virtual address spaces protect applications from each other
- But this makes it difficult to share data
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying
  - Read/Write (access to share data)
  - Execute (shared libraries)
- We can use shared memory to allow processes to share data using direct memory reference
  - Both processes see updates to the shared memory segment
  - How are we going to coordinate access to shared data?

Shared Memory (2)

Example

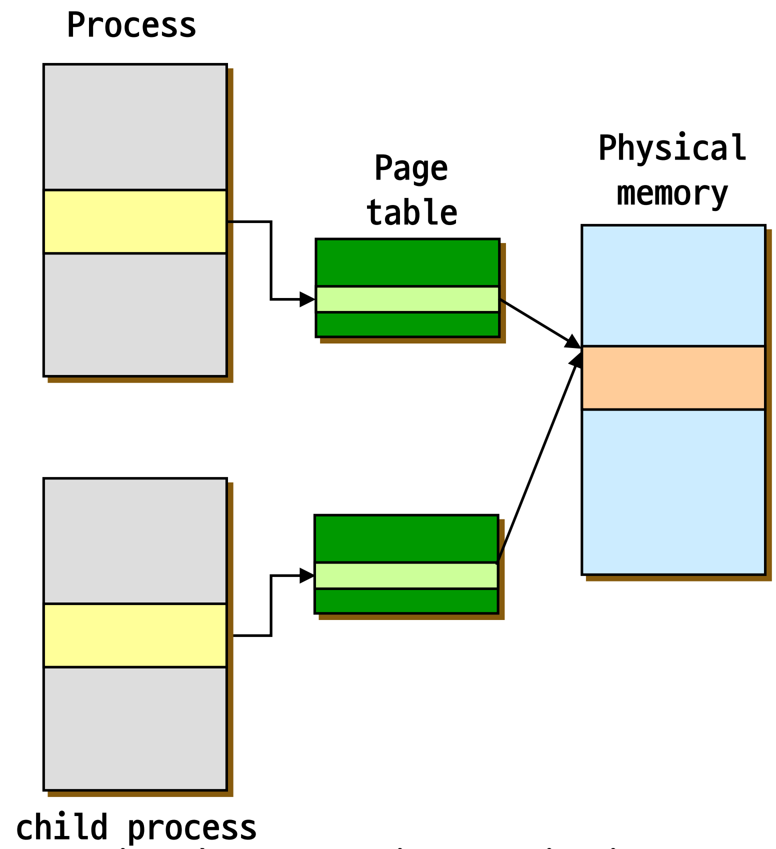
```
Process A

#include <sys/shm.h>
key_t key = 123456;
char *shared_memory;

/* shared_memory create */
shmid = shmget(key, SIZE, IPC_CREAT ¦ 0644)

shared_memory = shmat(shmid, NULL, 0)

memset(shared_memory, 0, strlen(shared_memory));   // shared_memory reset
memcpy(shared_memory, "TEST", 5);
```

```
Process B

#include <sys/shm.h>
key_t key = 123456;

shmid = shmget(key, SIZE, 0644)
shm = shmat(shmid, NULL, 0)
printf("%s\n",shm);
```

Shared Memory (3)

Implementation

- How can we implement shared memory using page tables?
  - Have PTEs in both tables map to the same physical frame
  - Each PTE can have different protection values
  - Must update both PTEs when page becomes invalid
- Can map shared memory at same or different virtual addresses in each process' address space?
  - Different: Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same: Less flexible, but shared pointers are valid



Copy On Write (1)

Process creation

- Requires copying the entire address space of the parent process to the child process
- Very slow and inefficient!

Write가 있을 때 Copy 하겠다.

원래 복사.

vfork 사용

Solution 1: Use threads

- Sharing address space is free

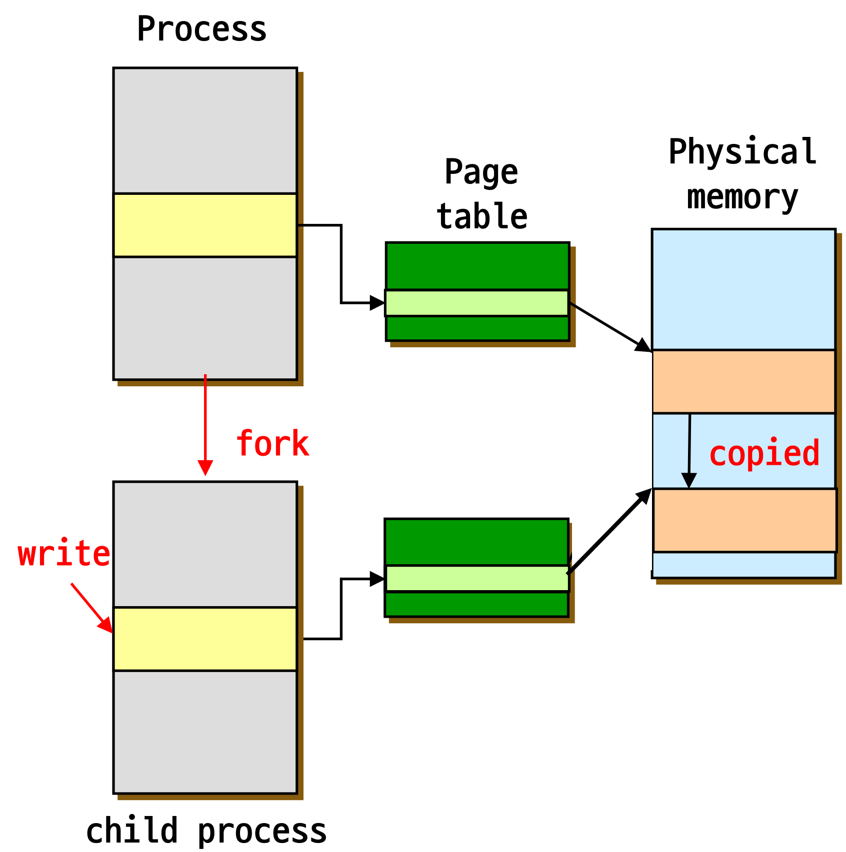
Solution 2: Use vfork() system call

- vfork() creates a process that shares the memory address space of its parent
- To prevent the parent from overwriting data needed by the child, the parent's execution is blocked until the child exits or executes a new program
- Any change by the child is visible to the parent once it resumes
- Useful when the child immediately executes exec()

Copy On Write (2)

Solution 3: Copy On Write (COW)

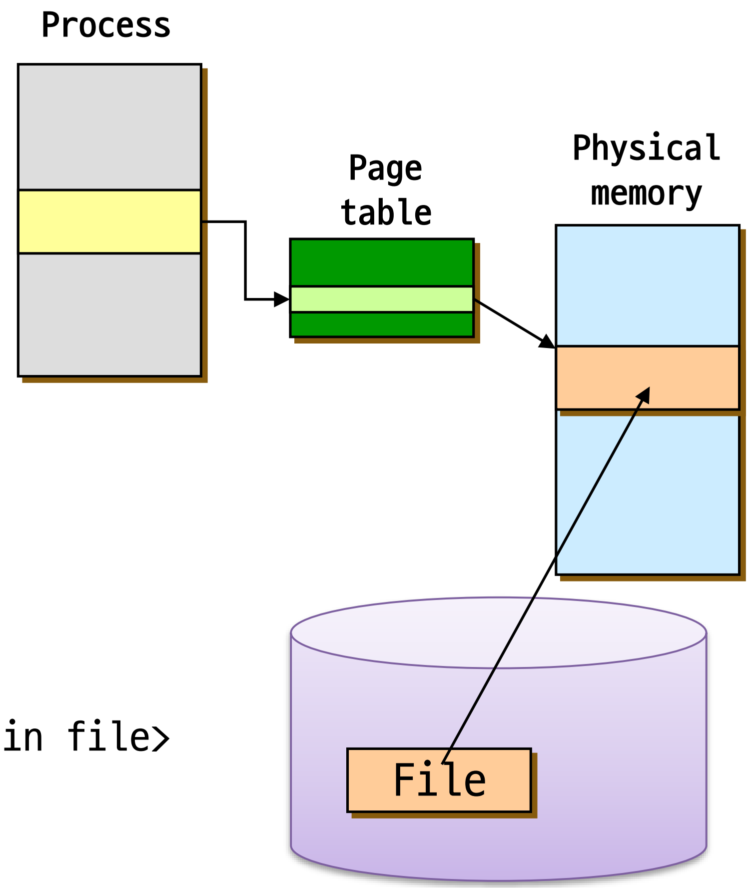
- Instead of copying all pages, create shared mappings of parent pages in child address space
- Shared pages are protected as read-only in child
  - Reads happen as usual
  - Writes generate a protection fault, trap to OS
  - OS copies the page and changes page mapping in client page table
  - Restarts write instruction



Memory-Mapped Files (1)

Memory-mapped files

- Mapped files enable processes to do file I/O using memory references
  - Instead of open(), read(), write(), close()
- mmap(): bind a file to a virtual memory region
  - PTEs map virtual addresses to physical frames holding file data
  - <Virtual address base + N> = <offset N in file>
- Initially, all pages in mapped region marked as invalid
  - Whenever invalid page is accessed, OS reads a page from file
  - When evicted from physical memory, OS writes a page to file
  - If page is not dirty, no write needed



Memory-Mapped Files (2)

Example

```
if ((fd = open("test.txt", O_RDWR)) == -1) {
    perror("open");
    exit(1);
}

addr = mmap(NULL, statbuf.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, (off_t)0);
if (addr == MAP_FAILED) {
    perror("mmap");
    exit(1);
}
close(fd);

printf("%s", addr);
strcpy(addr, buf);
sprintf(addr, "%s to modify some text ", buf);
...
...
```

메모리 매핑 해제

Memory-Mapped Files (3)

Advantages

- Uniform access for files and memory (just use pointers)
- Less copying
- Several processes can map the same file allowing the pages in memory to be shared

Drawbacks

- Process has less control over data movement
- Does not generalize to streamed I/O (pipes, sockets, etc.)

Note:

- File is essentially backing store for that region of the virtual address space (instead of using the swap file)
- Virtual address space not backed by "real" files also called "anonymous VM(page)"

anonymous page.

→ file back-up X.

Summary (1)

VM mechanisms

- Physical and virtual addressing
- Partitioning, Paging, Segmentation
- Page table management, TLBs, etc.

VM policies

- Page replacement algorithms
- Memory allocation policies

VM requires hardware and OS support

- MMU (Memory Management Unit)
- TLB (Translation Lookaside Buffer)
- Page tables, etc.

Summary (2)

VM optimizations

- Demand paging (space)
- Managing page tables (space)
- Efficient translation using TLBs (time)
- Page replacement policy (time)

Advanced functionality

- Sharing memory
- Copy on write
- Mapped files