Chapter 9: Virtual-Memory Management

Prof. Li-Pin Chang

CS@NYCU

Chapter 9: Virtual Memory

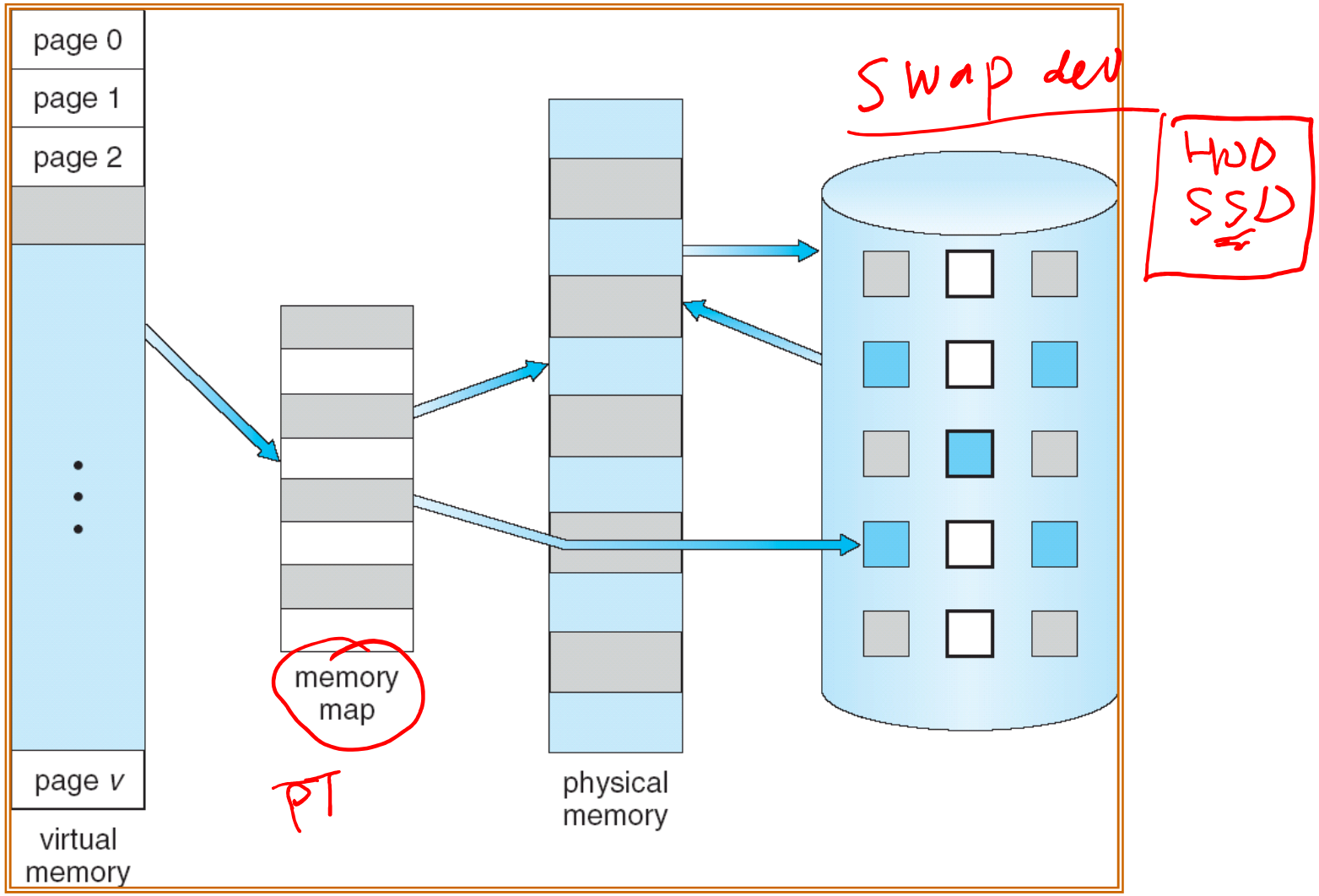
- Demand Paging
- Copy-on-Write
- Page Replacement
- Thrashing
- Allocation of Frames
- Performance Issues
- Swapping
- Memory-Mapped Files
- Kernel Memory Allocation
- Operating System Examples

DEMAND PAGING

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 for more efficient process creation
  - Improves the degree of multiprogramming
  - Allows address spaces to be shared by several processes
- Virtual memory is usually implemented by demand paging

Virtual Memory That is Larger Than Physical Memory

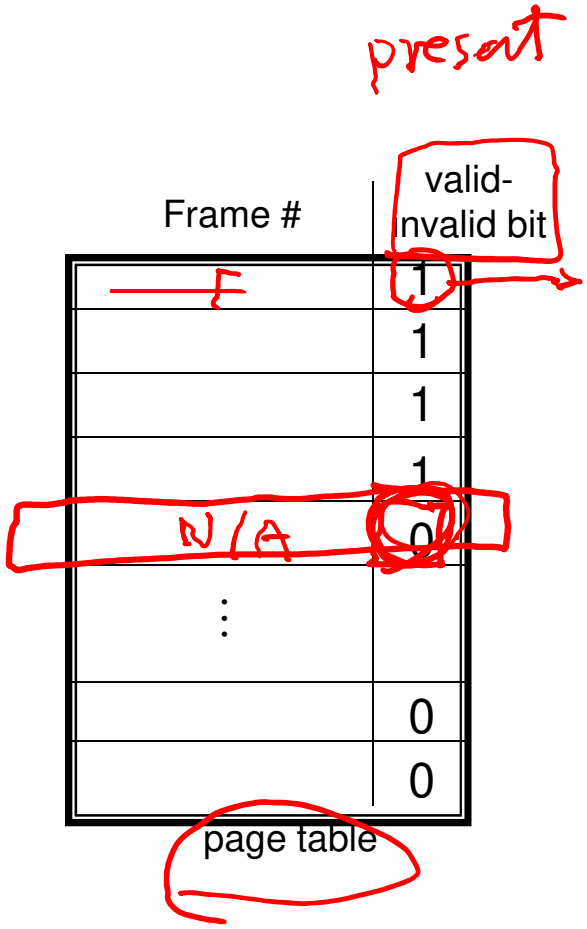


Demand Paging

- Logical memory space can be larger than physical memory space
- Bring a page into memory only when it is needed
  - Less memory needed
  - Faster response
  - More users/processes
  - Less I/O needed
- Page is needed ⇒ reference to it (load or store)
  - In-memory ⇒ normal reference
  - not-in-memory ⇒ bring to memory

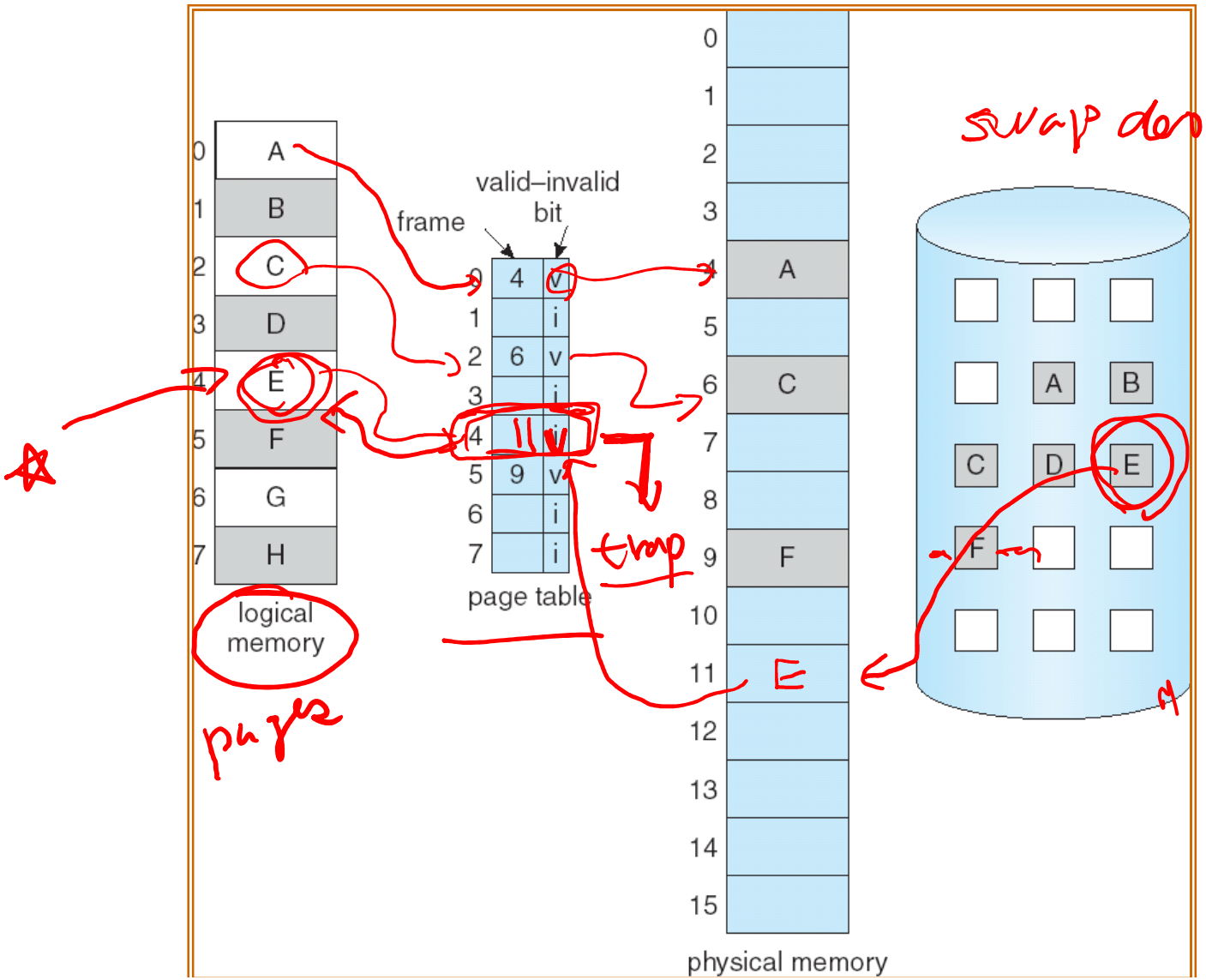
Valid-Invalid Bit of Page Table Entry

- With each page table entry a valid–invalid bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially valid–invalid but is set to 0 on all entries
- During address translation, if valid–invalid bit in page table entry is 0 ⇒ page fault

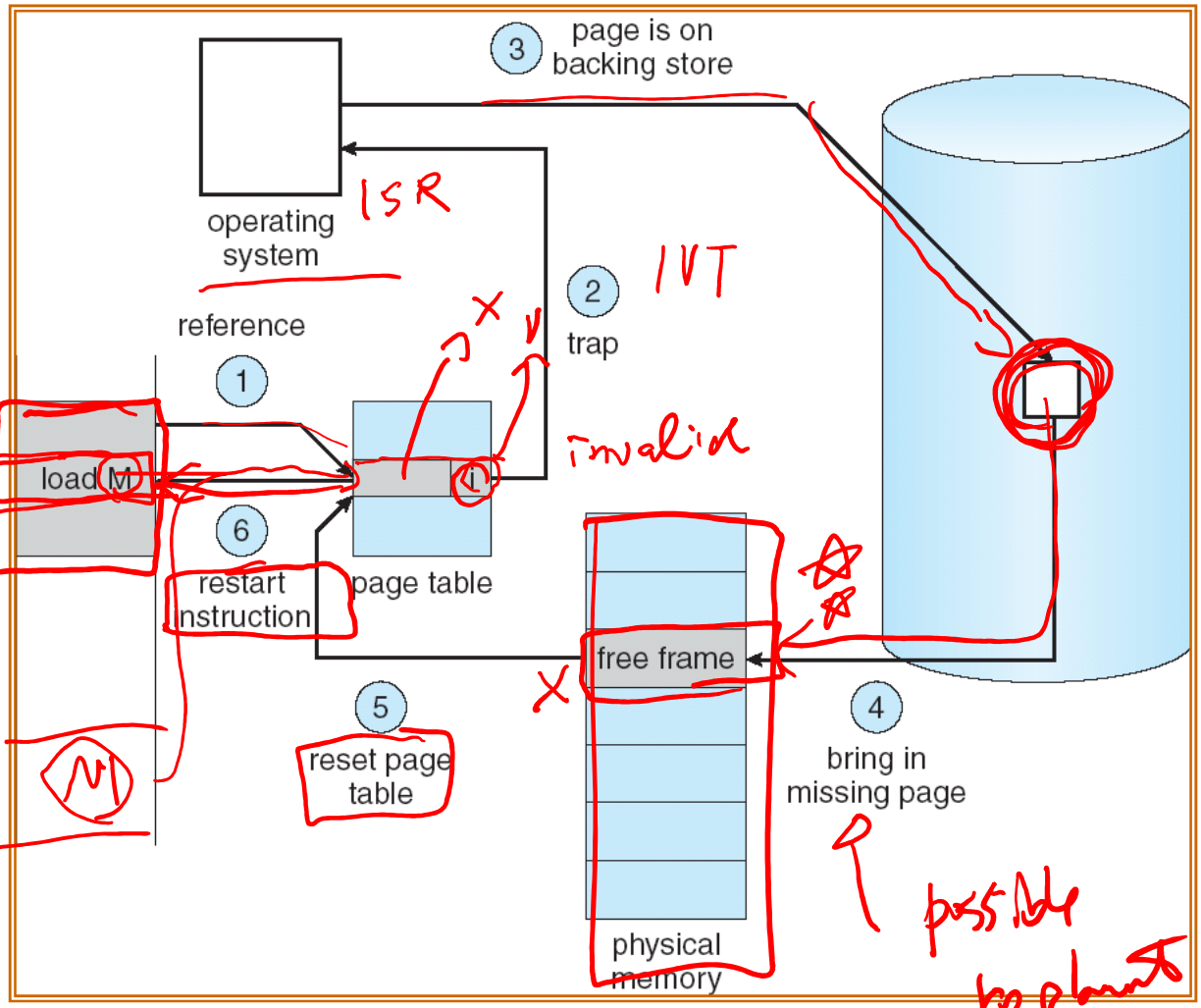


trap → OS

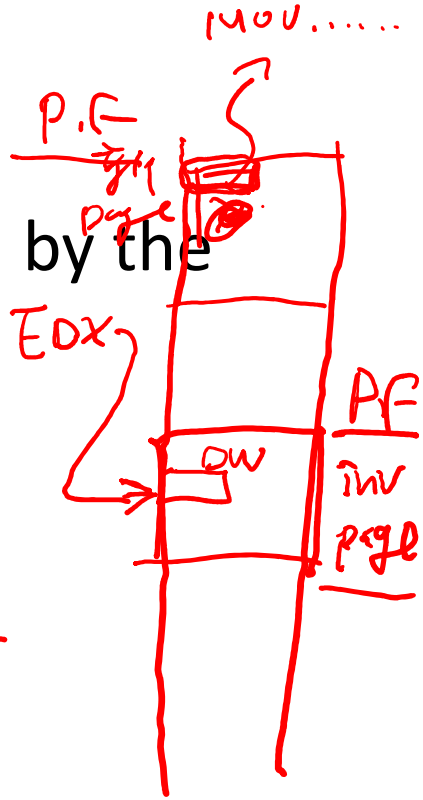
Page Table When Some Pages Are Not in Main Memory



Steps of Handling a Page Fault



1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
   a. Wait in a queue for this device 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 to some other user (CPU scheduling, optional).
7. Receive an interrupt from the disk I/O subsystem (I/O completed).
8. Save the registers and process state for the other user (if step 6 is executed).
9. Determine that the interrupt was from the disk.
10. Correct the page table and other tables to show that the desired page is now in memory.
11. Wait for the CPU to be allocated to this process again.
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction.



- Up to how many page faults will be caused by the following instruction?

```
MOV     EAX, DWORD PTR [EDX]
```

- Total 2
  - The instruction itself once and the access of memory location [EDX] the other one
  - Instructions and data do not span across page boundaries

Performance of Demand Paging

- Page Fault Rate $0 \leq p \leq 1.0$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault

- Effective Access Time (EAT)

$$EAT = (1 - p) \times \text{memory access}$$
$$+ p \times (\text{page fault overhead}$$
$$+ [\text{swap page out }]$$
$$+ \text{swap page in}$$
$$+ \text{restart overhead})$$

Suppose that the page-fault service time is 8 ms, including disk I/O and all necessary memory access

A memory access takes 200ns (TLB hit+TLBmiss)

The page-fault ratio is p

The EAT is

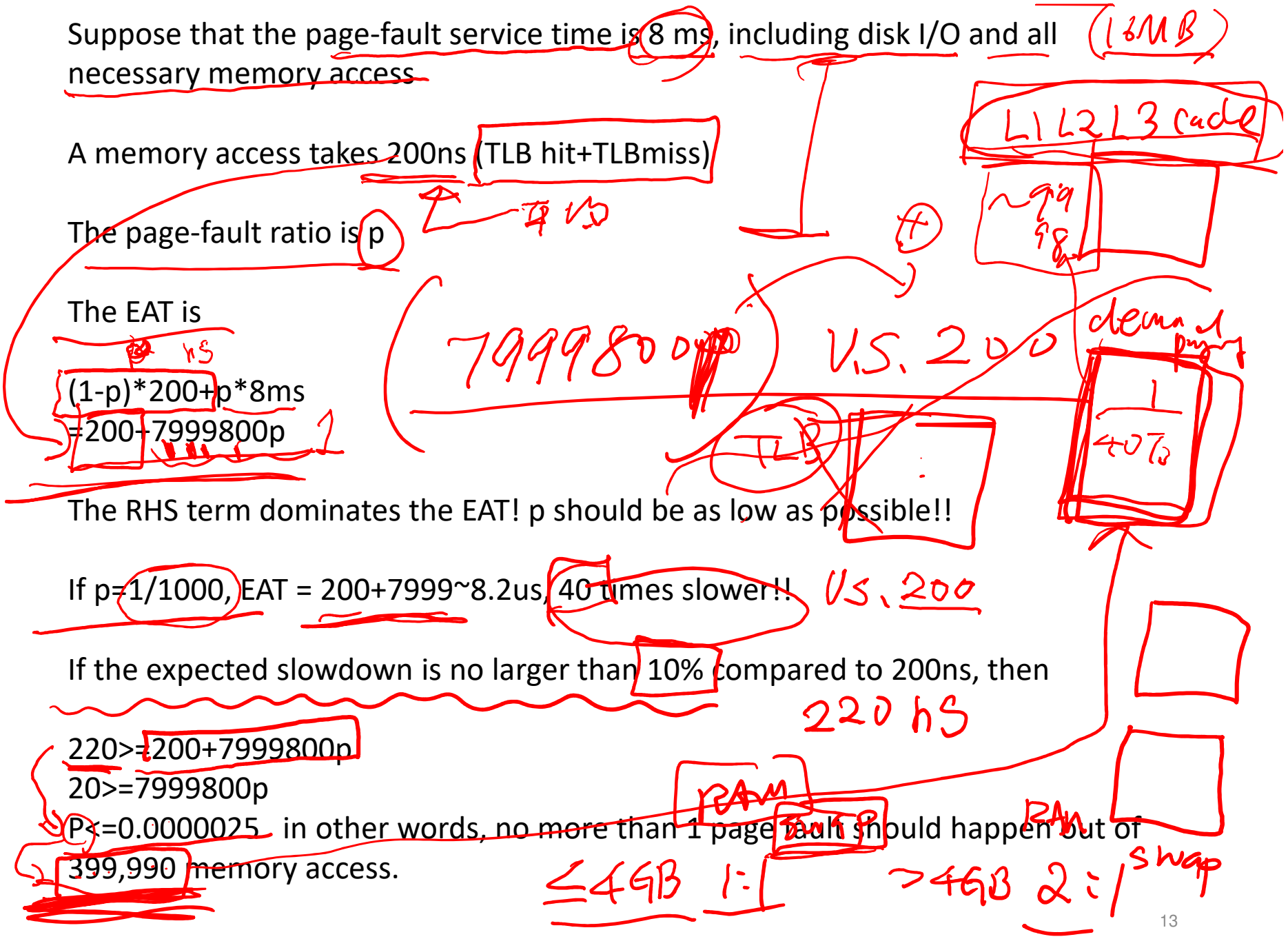
(1-p)*200+p*8ms
=200+7999800p

The RHS term dominates the EAT! p should be as low as possible!!

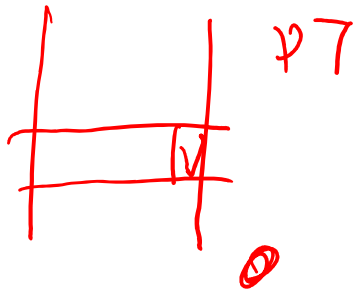
If p=1/1000, EAT = 200+7999~8.2us, 40 times slower!!

If the expected slowdown is no larger than 10% compared to 200ns, then

220>=200+7999800p
20>=7999800p
P<=0.0000025 in other words, no more than 1 page fault should happen out of 399,990 memory access.

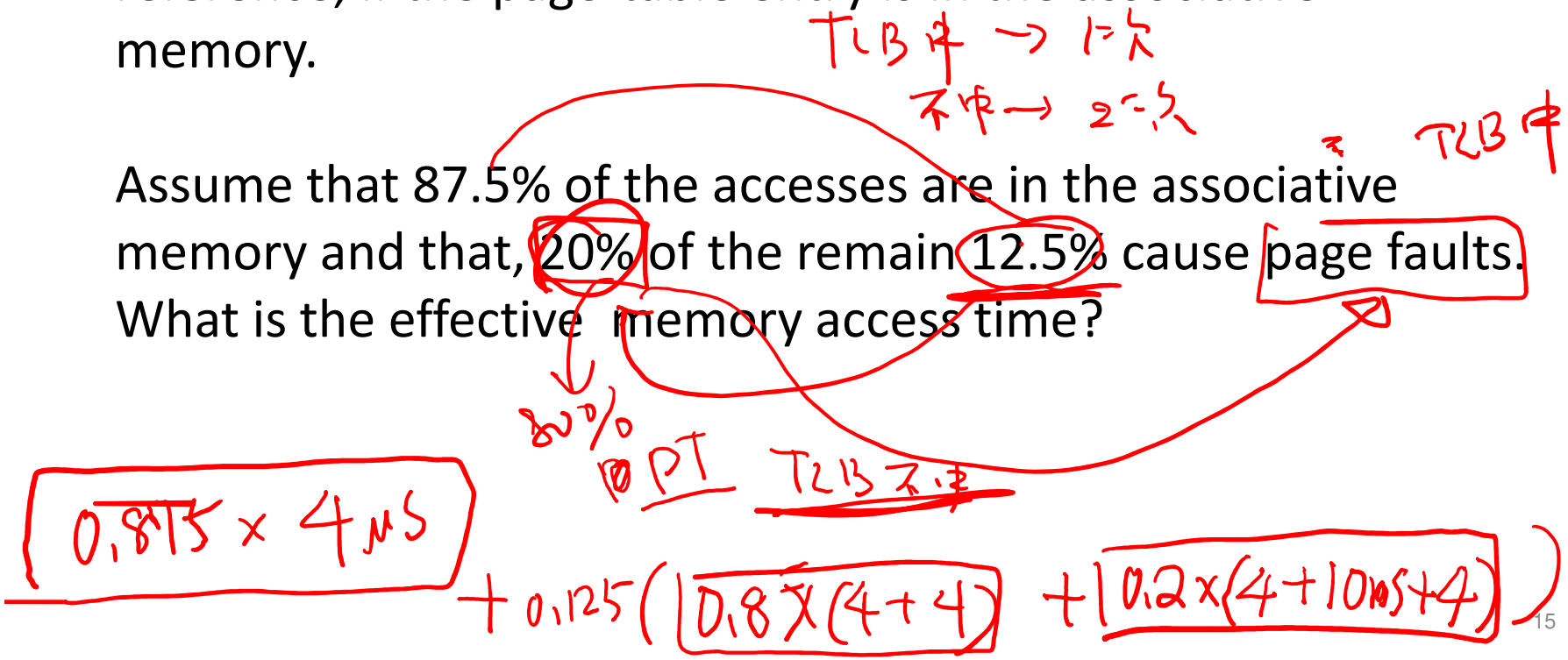


- If TLB hit
  - won't be a page fault
  - TLB access
- If TLB miss
  - If not a page fault
    - TLB access + page table access + TLB update
  - If a page fault
    - The page is not in memory
    - TLB access + page table access + page fault handling + TLB update



**Consider a demand-paging system with a paging disk that has an average access and transfer time of 10 milliseconds. Addresses are translated through a page table in main memory, with an access time of 4 microsecond per memory access. Thus, each memory reference through the page table takes two accesses. To improve this time, we have added an associative memory that reduces access time to one memory reference, if the page-table entry is in the associative memory.

Assume that 87.5% of the accesses are in the associative memory and that, 20% of the remain 12.5% cause page faults. What is the effective memory access time?



PAGE REPLACEMENT

11:13 am

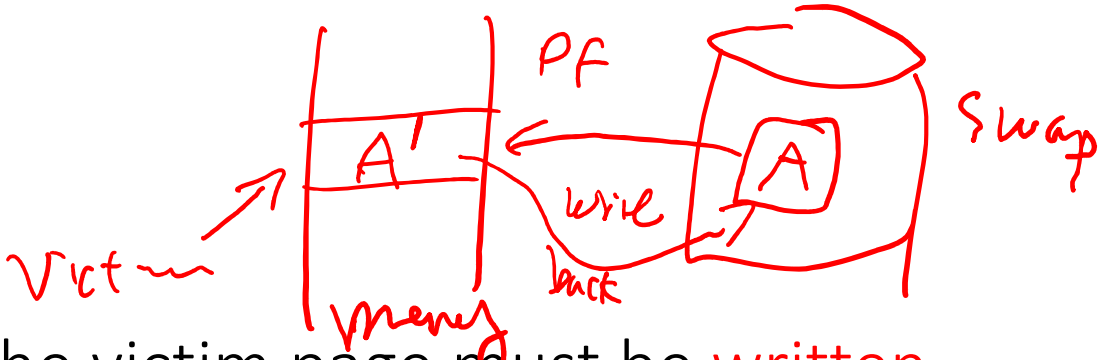
Page Replacement

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Basic Page Replacement

- Find the location of the desired page on disk
- 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
- Read the desired page into the free frame
- Update the page and frame tables
- Restart the process

Page Replacement

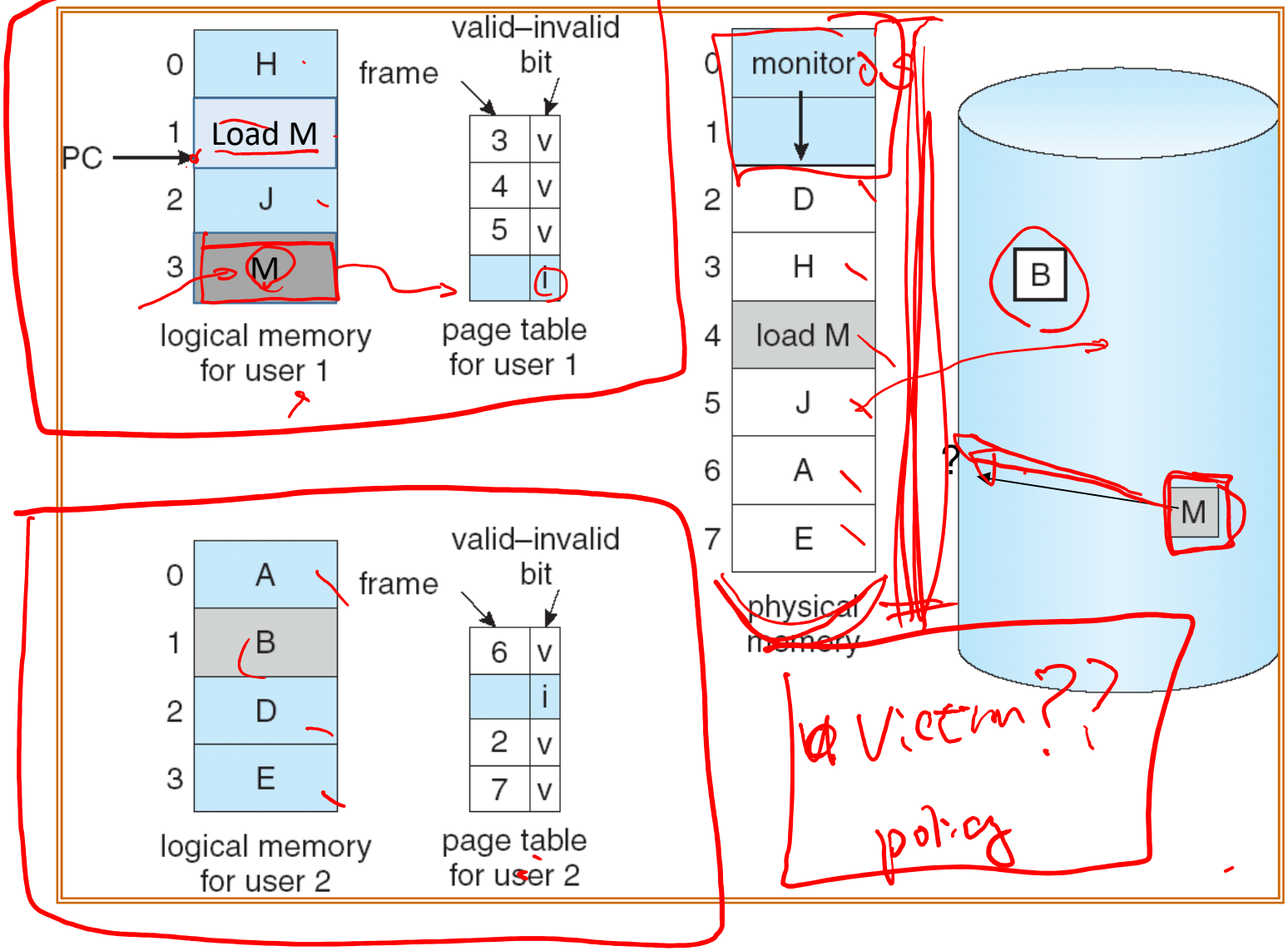


- To replace a page, the victim page must be written back to the backing store. It may double the time of page-fault handling

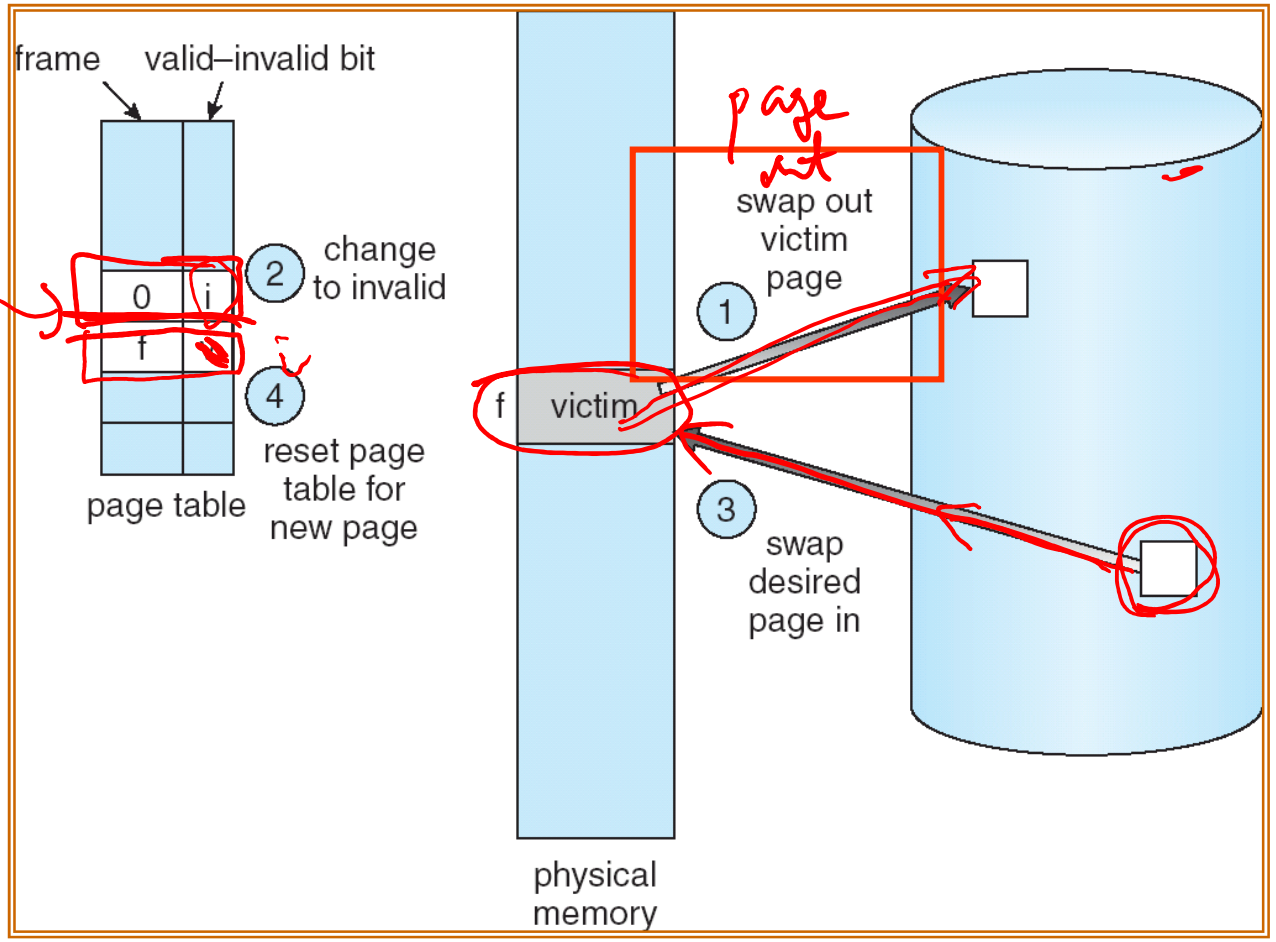
- Replace a page that is unlikely to be used in the near future to reduce the overhead of reading a page from disk
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk

MMU

Need For Page Replacement



Page Replacement (mechanism)



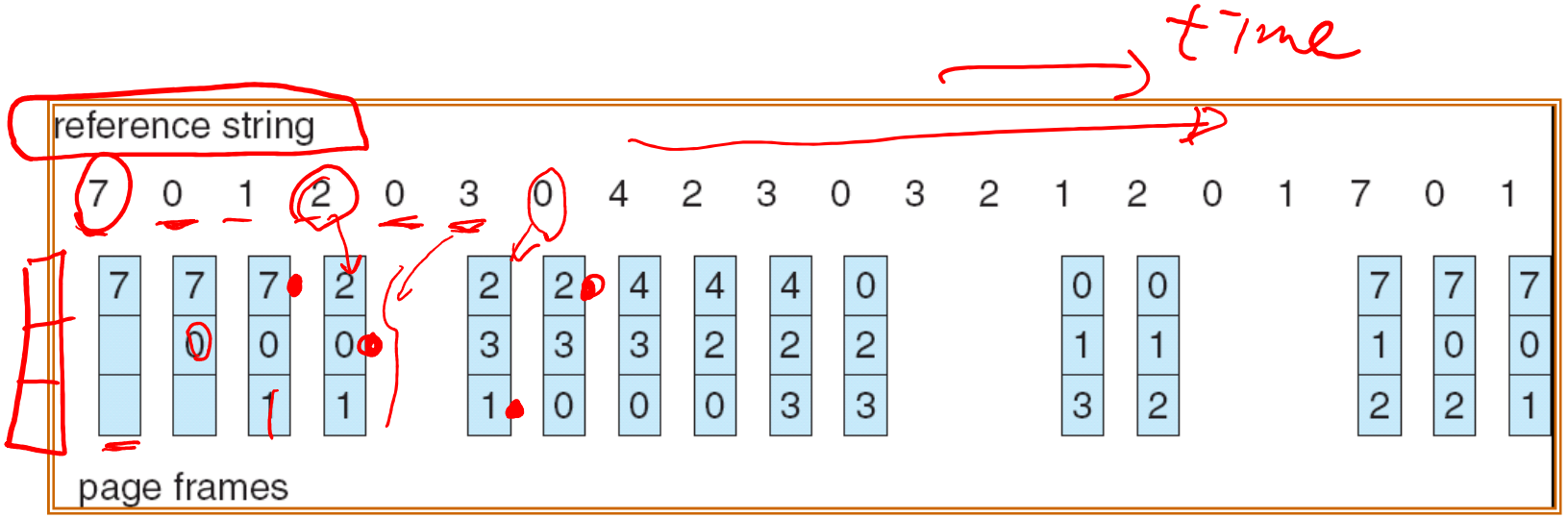
Reference string

0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105

Page size=100B

1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

FIFO Page Replacement

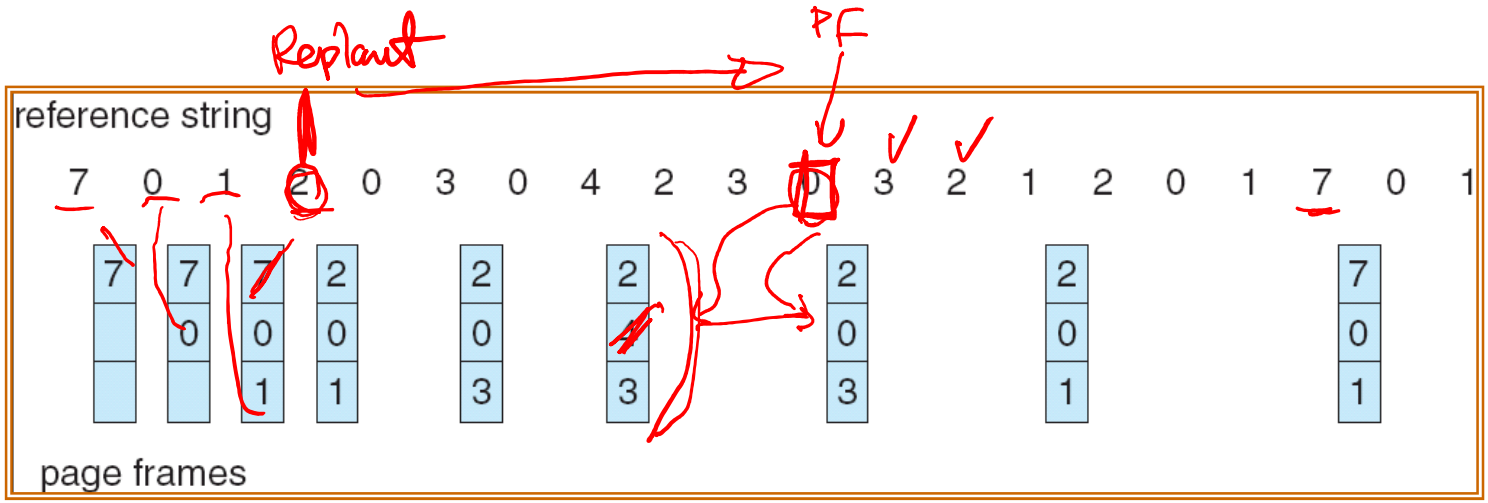


3F.

15 page faults

希望↓↓↓

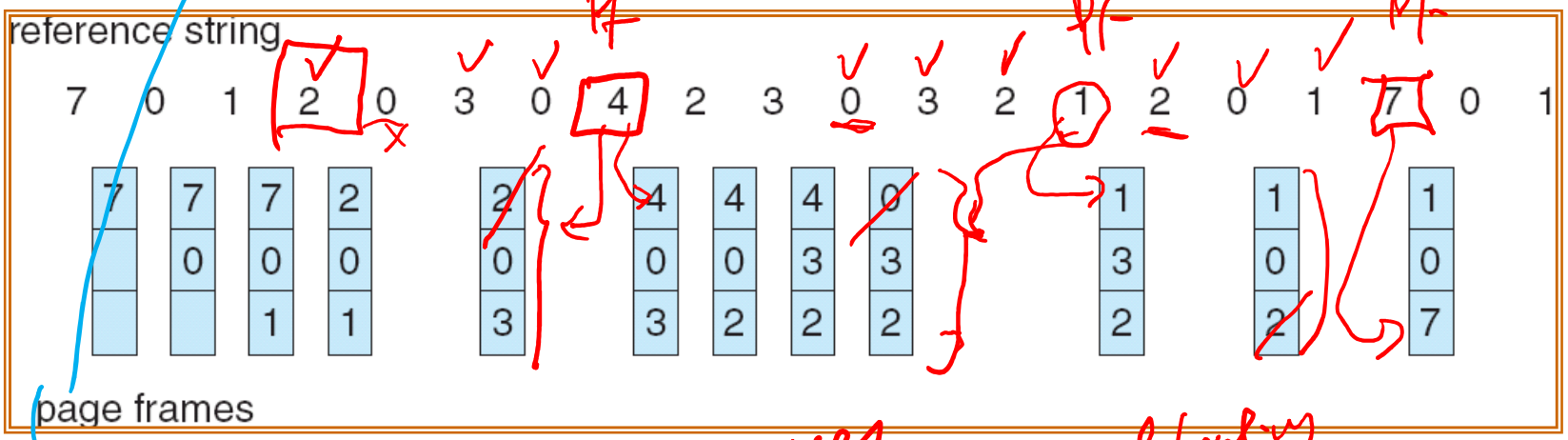
Optimal Page Replacement (OPT)



9 page faults.

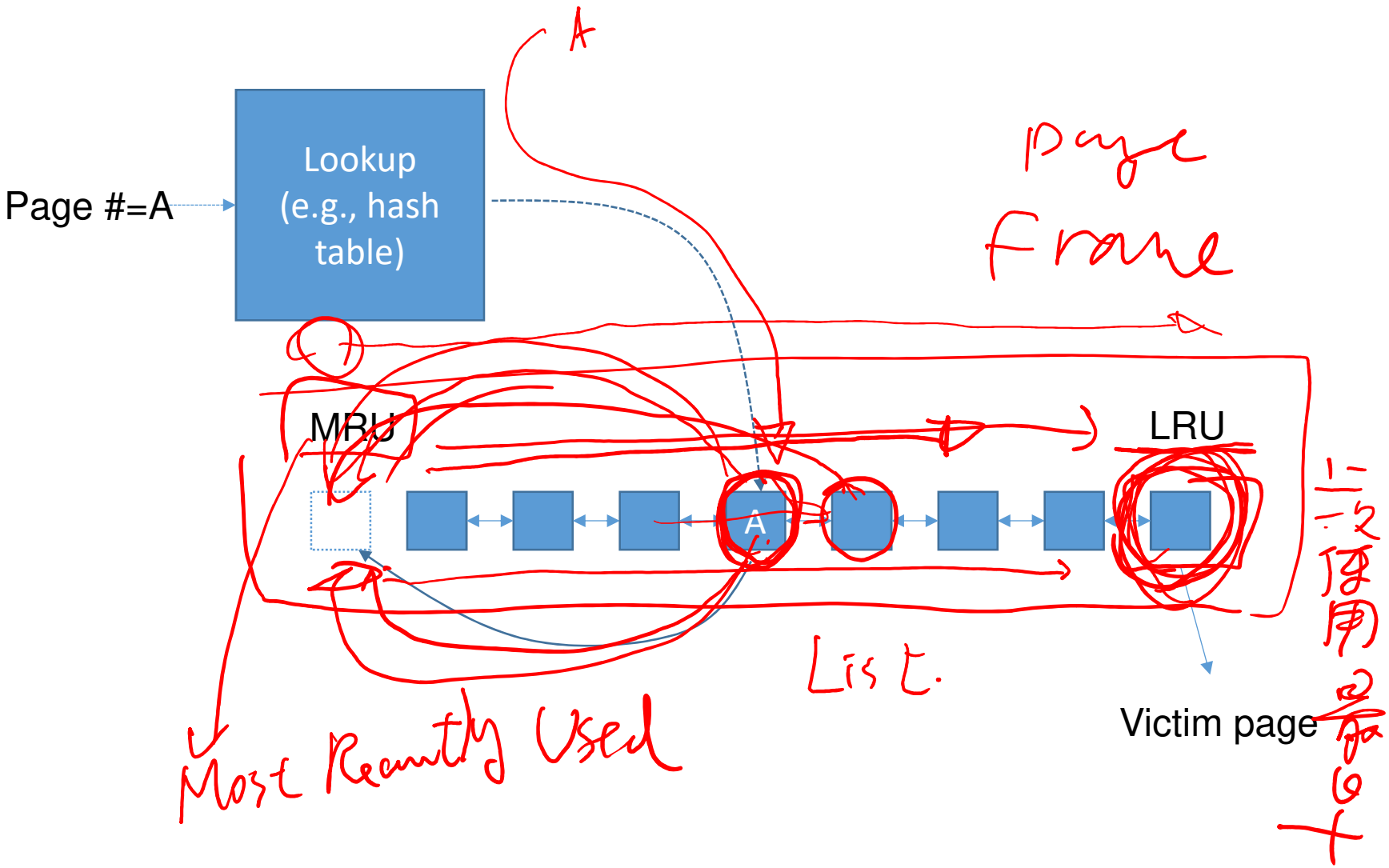
OPT is not applicable to real systems, however.

Least-Recently Used (LRU)

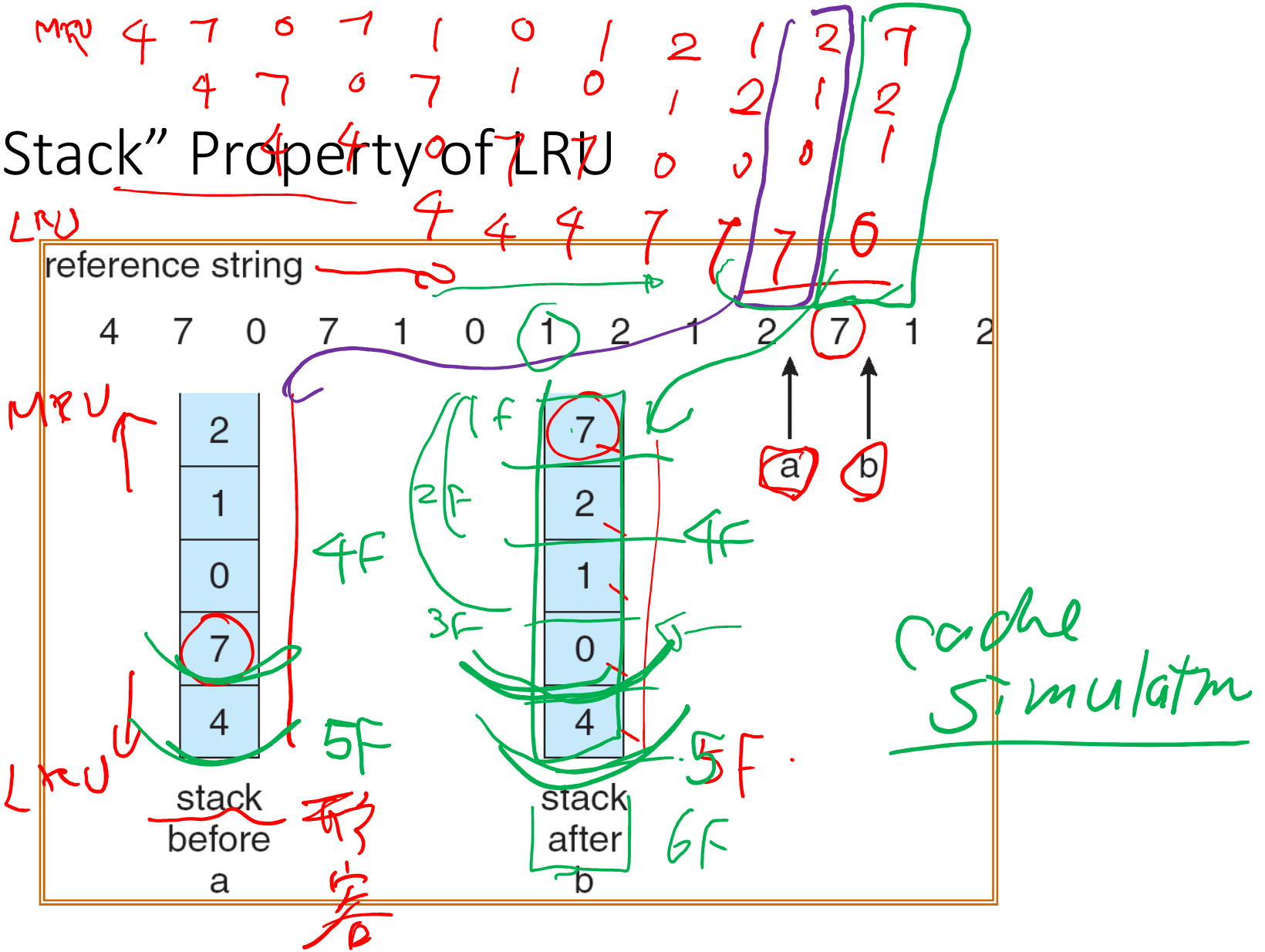


12 page faults.

An LRU Implementation



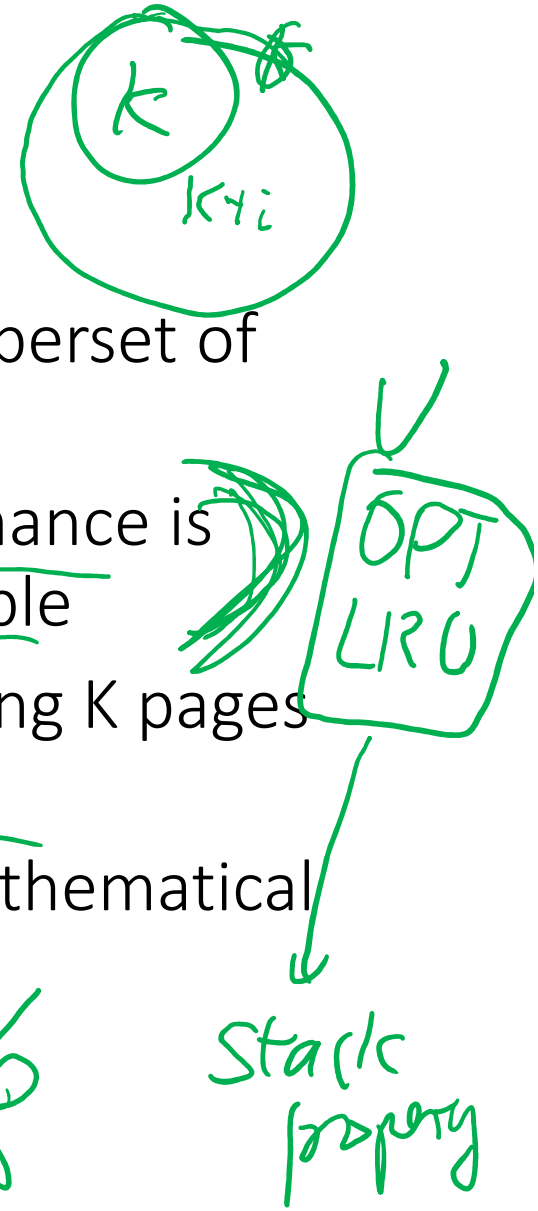
The "Stack" Property of LRU



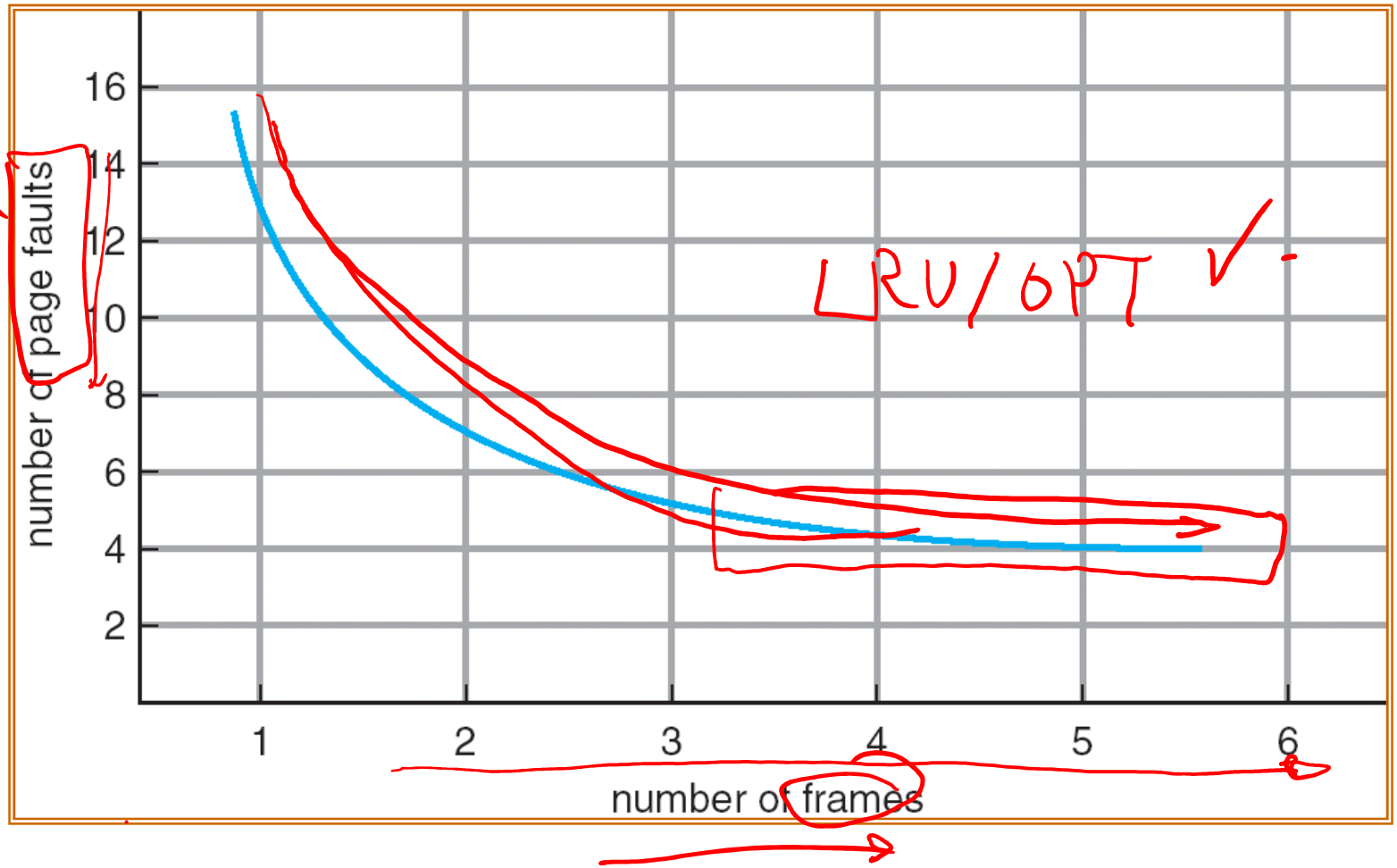
See what happens if there are four frames only.

The "Stack" Property of LRU

- The page set of LRU with K pages is a superset of that of LRU with K-1 pages
- This property suggests that LRU performance is always better as more frames are available
- Also useful in cache simulation. Simulating K pages get results of K, K-1, K-2….
- The stack property can be proved by mathematical induction



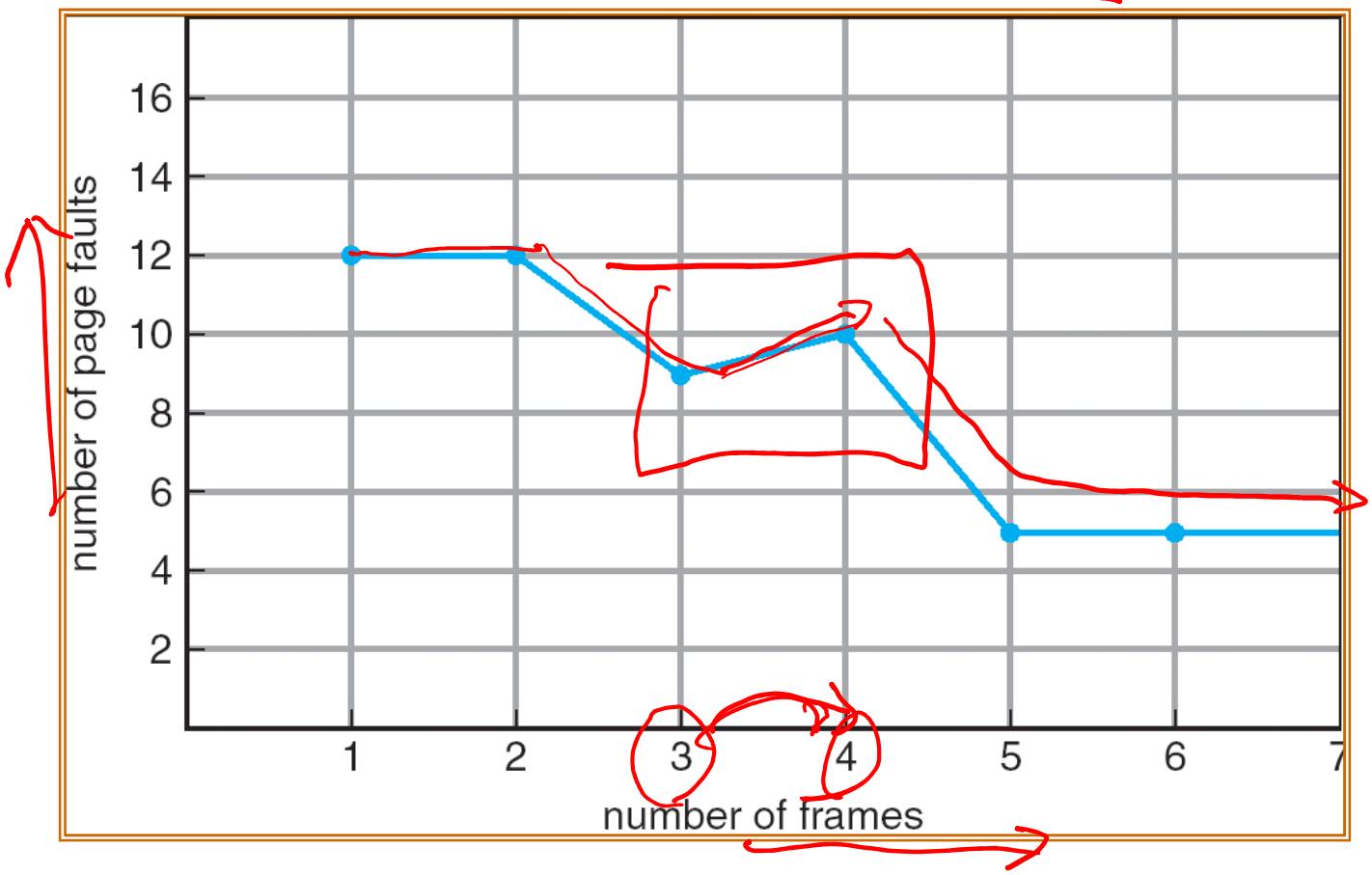
Trends of Page Fault # vs. Frame



The margin quickly saturates beyond a sweet spot

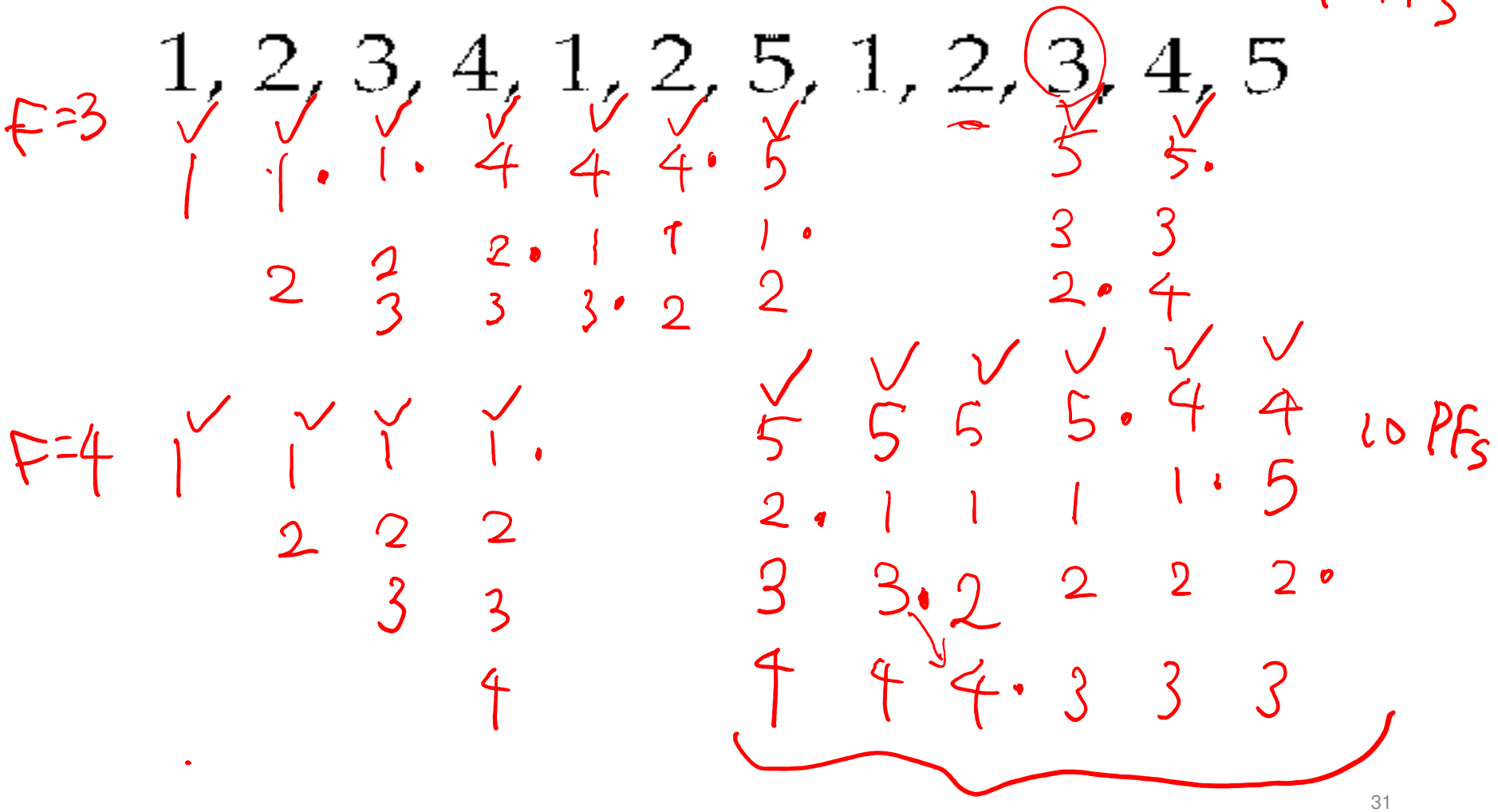
However, FIFO suffers from Belady's Anomaly

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



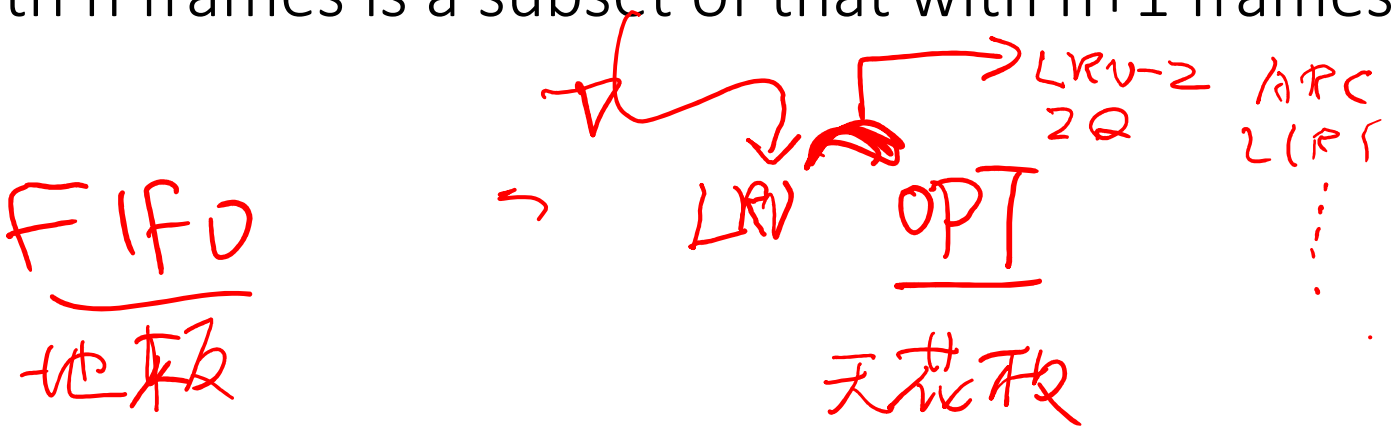
Adding more frames unexpectedly defragds the performance

Reproducing Belady's Anomaly 微觀.



Belady's Anomaly

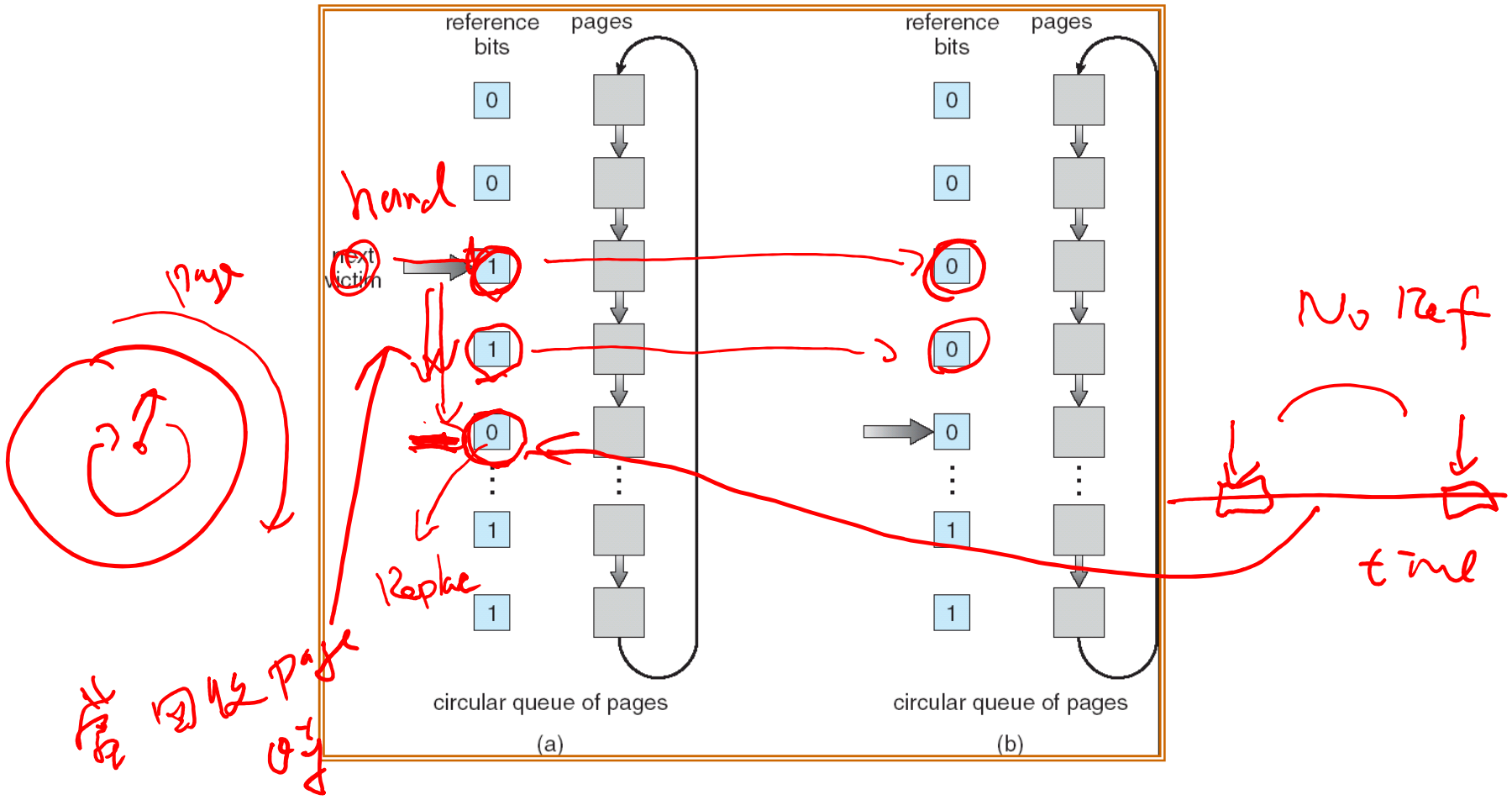
- FIFO suffers from Belady's anomaly
- LRU and OPT do not
- LRU and OPT are "stack algorithms", i.e., the page set with n frames is a subset of that with n+1 frames



LRU Approximation Algorithms

- Reference bit
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace the one which is 0 (if one exists). We do not know the order, however
- 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 (i.e., give a second chance)
    - replace next page (in clock order), subject to same rules

Second-Chance (clock) Page-Replacement Algorithm



It degrades into the FIFO policy in the worst case

Enhanced Second-Chance Algorithm

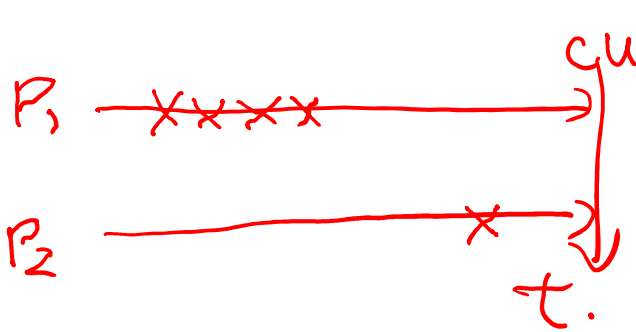
(ref, dirty)

1. (0, 0) neither recently used nor modified—best page to replace
2. (0, 1) not recently used but modified—not quite as good, because the page will need to be written out before replacement
3. (1, 0) recently used but clean—probably will be used again soon
4. (1, 1) recently used and modified—probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

- The first page encountered at the lowest nonempty class is replaced
- This method considers to reduce I/O costs.

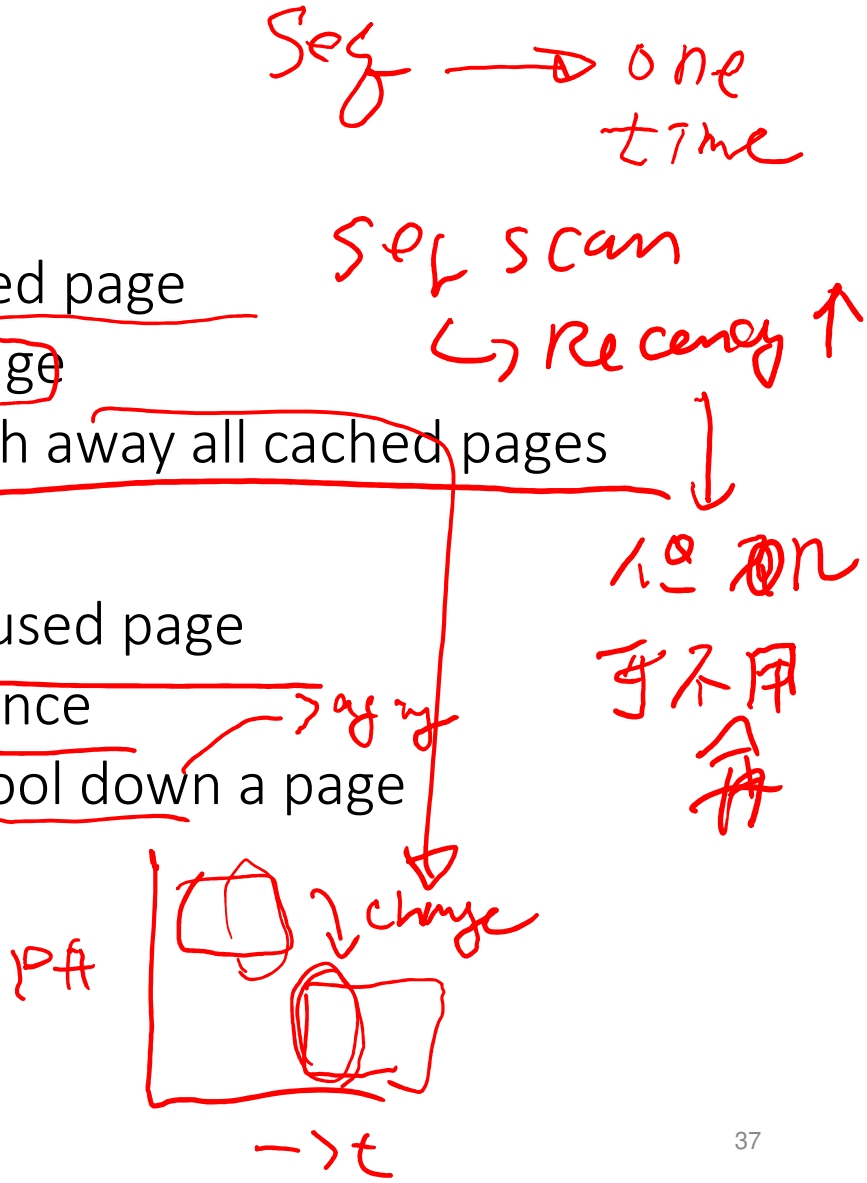
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
- LFU Algorithm: replaces page with smallest count
  - Periodically bit shift for counters (aging)
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used
  - Preserving pages newly brought in



Recency vs. Frequency

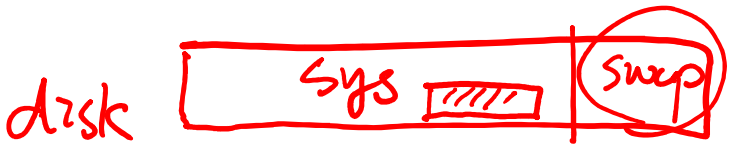
- LRU
  - Replace the least recently used page
  - Fast response to locality change
  - Sequential reference will wash away all cached pages
- LFU
  - Replace the least frequently used page
  - Resistant to sequential reference
  - Need time to warm up and cool down a page



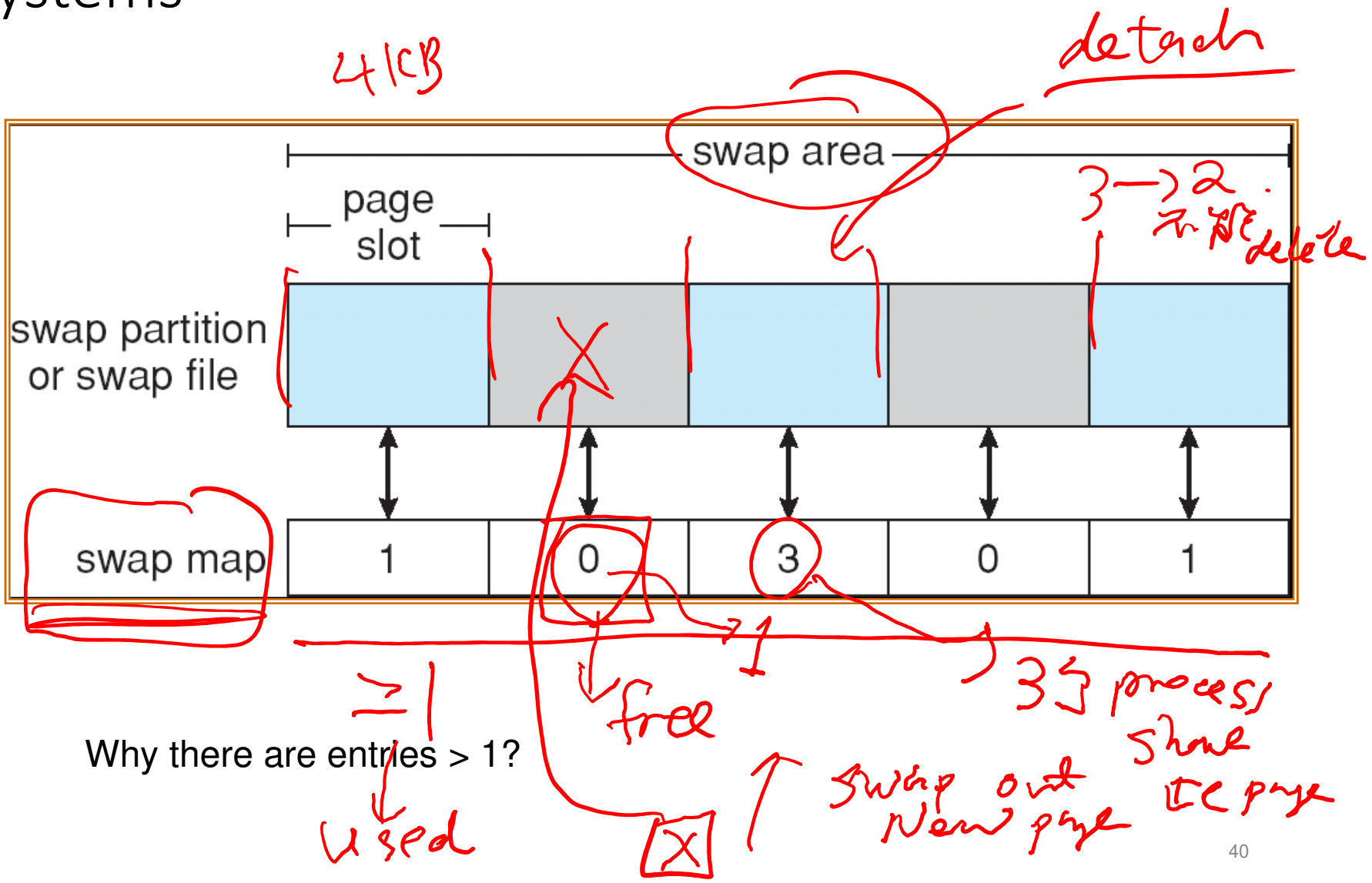
Swap Space Management

Swap-Space Management

- Swap-space — Virtual memory uses disk space as an extension of main memory
- Swap-space can be carved out of the normal file system (Windows/Linux) or can be in a separate disk partition (Linux)
- Swap-space management
  - Kernel uses swap maps to track swap-space use
  - 4.3BSD allocates swap space when process starts; holds text segment (the program) and data segment
  - Solaris 2 allocates swap space only when a page is forced out of physical memory, not when the virtual memory page is first created



The Data Structures for Swapping on Linux Systems



Applications of Demand Paging

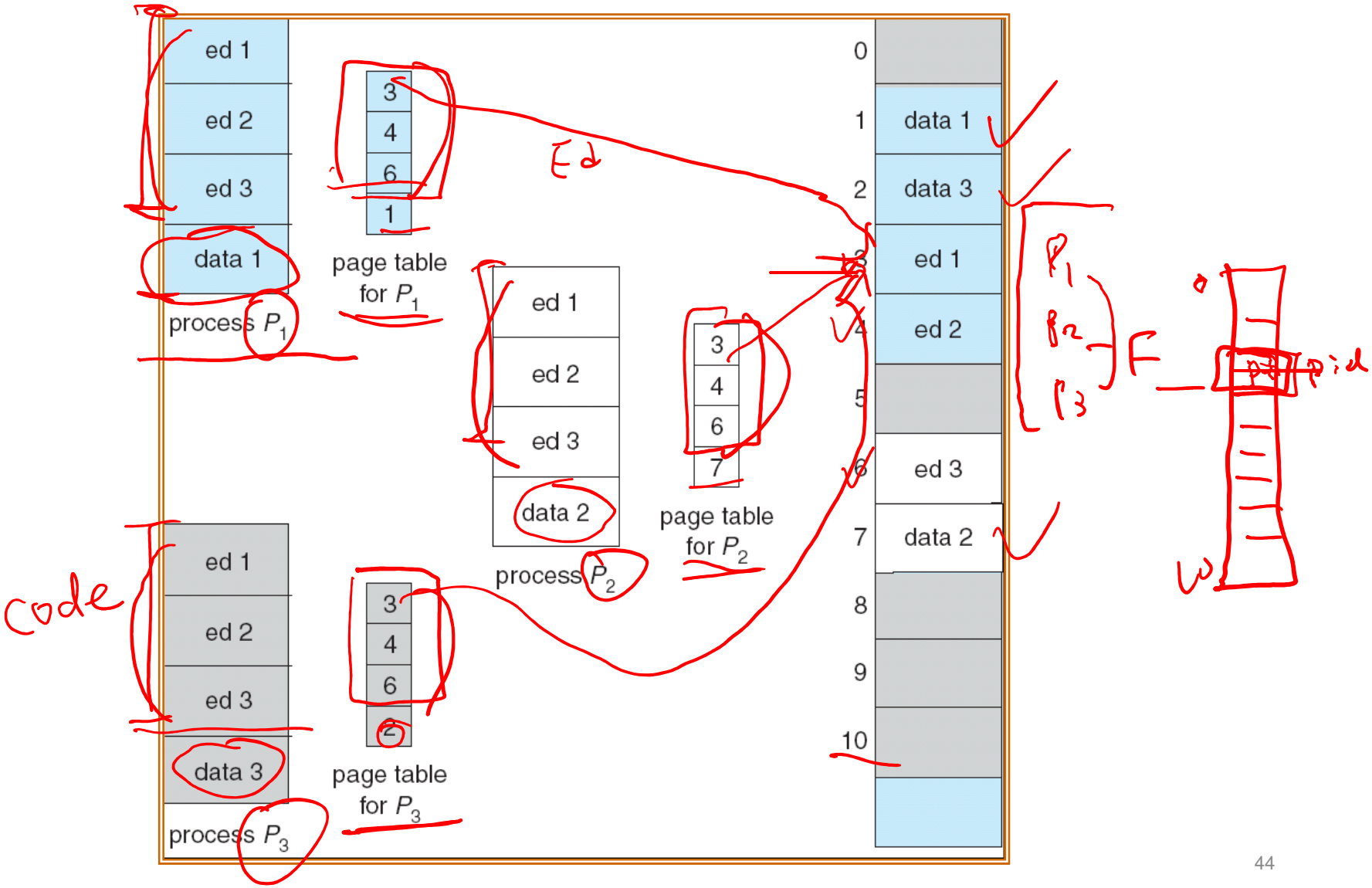
OS Features based on Demand Paging

- Page Sharing
  - Two processes share a memory region
  - Shared memory can be mapped to a file
- Copy-on-write
  - A fast implementation of memory duplication between processes
  - Pages are shared until being modified
  - Efficient fork()
- Memory-mapped file
  - Memory region backed by a regular file, not the swap space

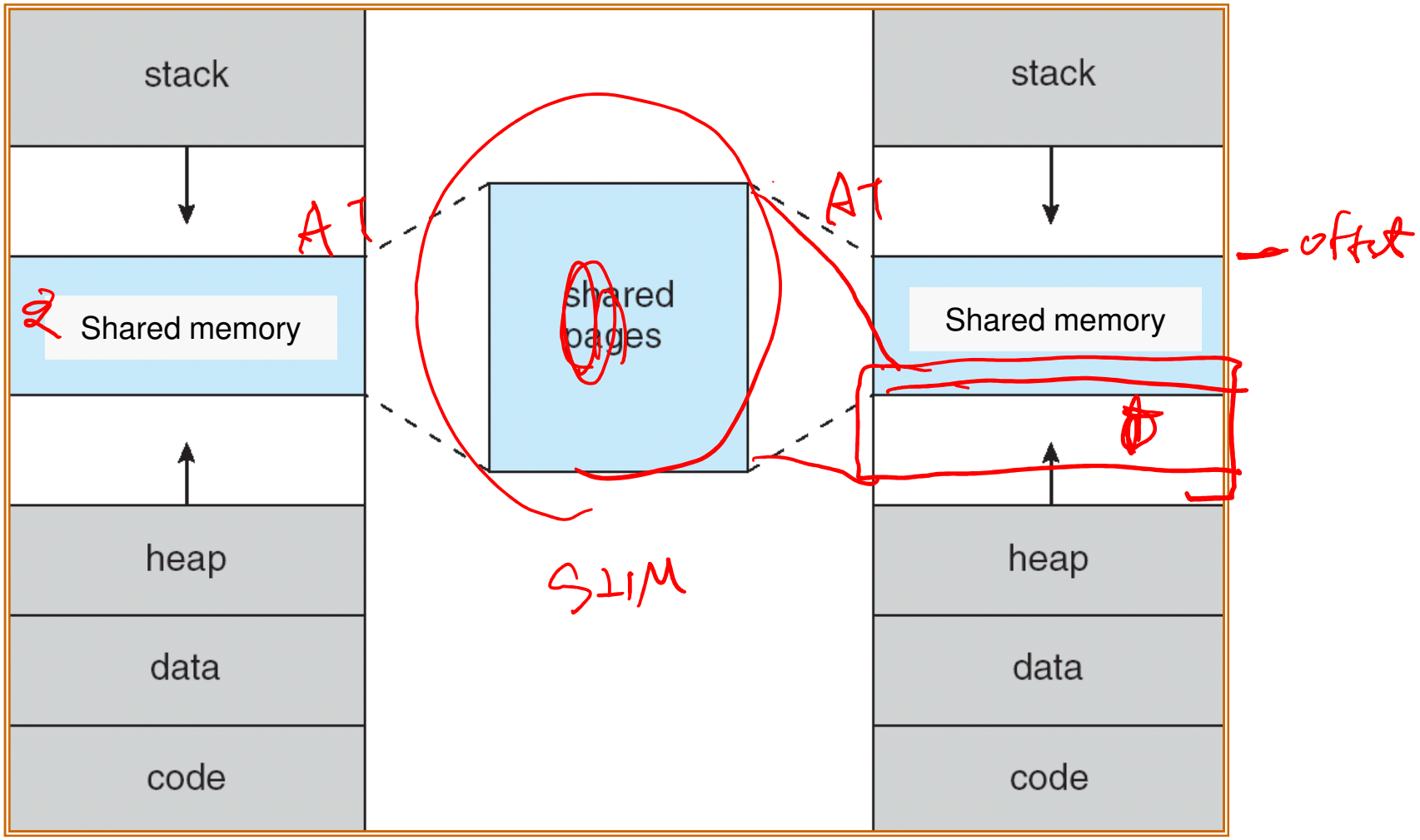
Page sharing

- Shared code
  - Dynamic linking-loading libraries (e.g., libc)
  - Kernel code
  - Handled by OS
- Shared memory
  - Creating a piece of shared memory: shmget()
  - Mapping a piece of shared memory to process address space: shmat()

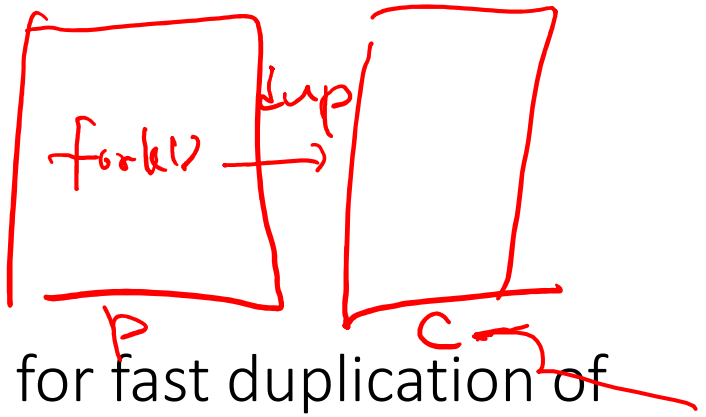
Shared Code


ed 1
ed 2
ed 3
data 1
process P_1
3
4
6
1
page table for P_1
ed 1
ed 2
ed 3
data 2
process P_2
3
4
6
7
page table for P_2
ed 1
ed 2
ed 3
data 3
process P_3
3
4
6
2
page table for P_3
0
1
2
3
4
5
6
7
8
9
10
data 1
data 3
ed 1
ed 2
ed 3
data 2

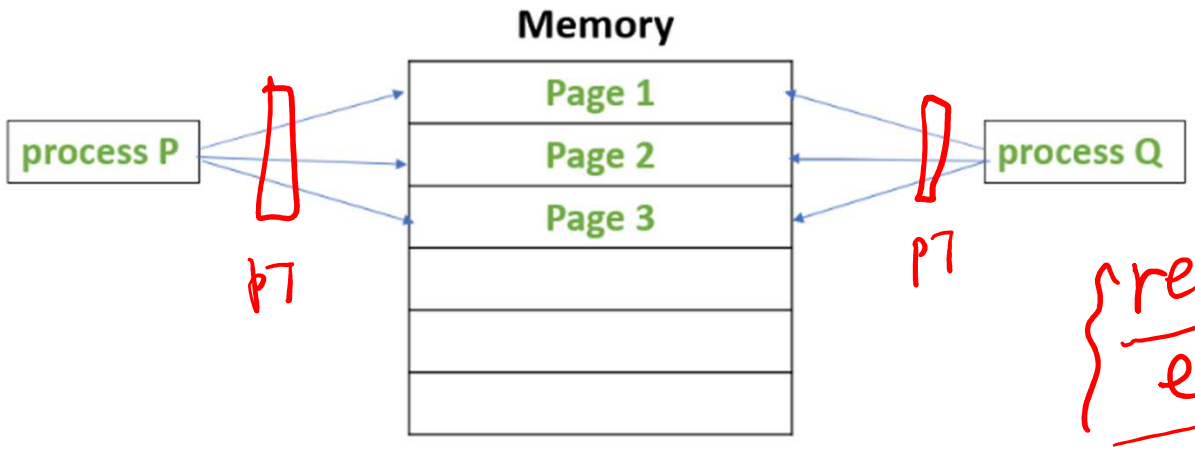
Shared Memory



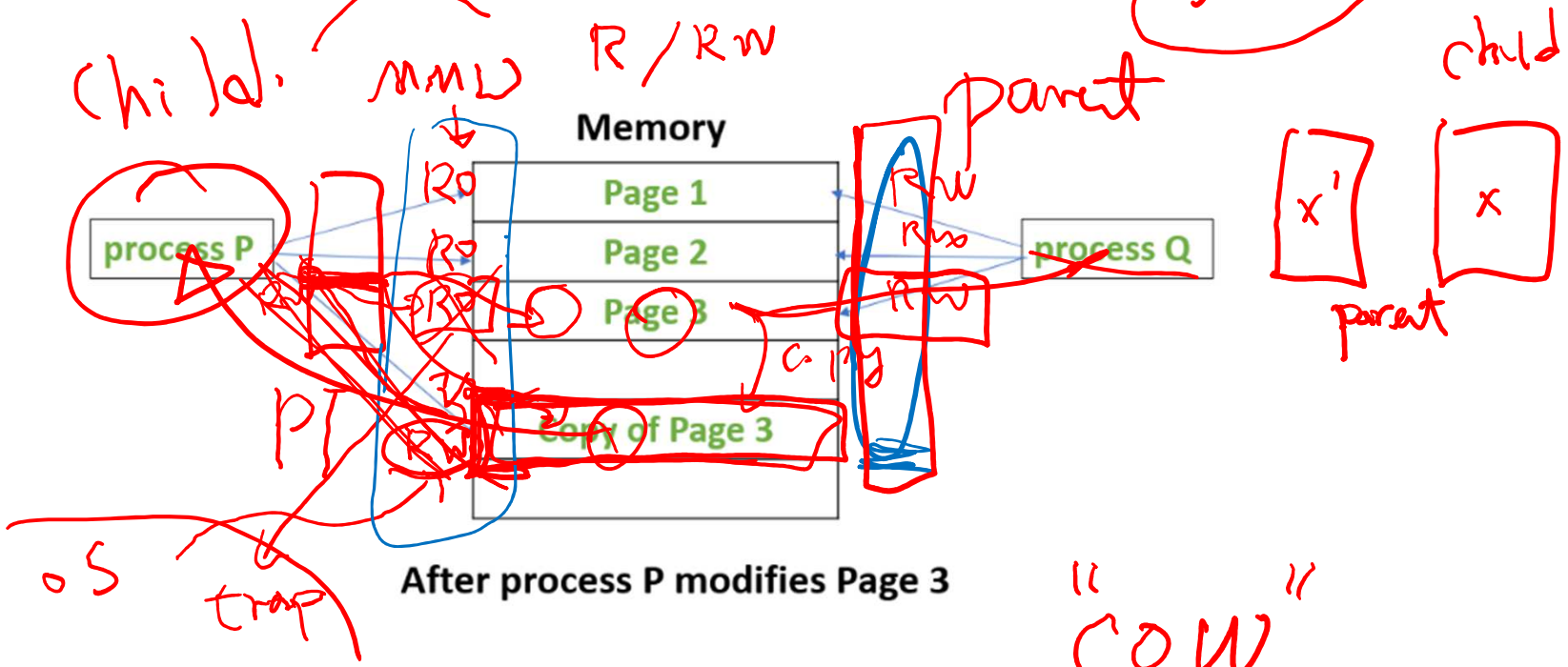
Copy-on-Write



- An extension to shared page; for fast duplication of memory
- 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, a copy of the page is then created
- A RO/RW bit for each page is necessary to trap writes to shared page; write to the page cause an exception/trap
- COW allows more efficient process creation as only modified pages are copied



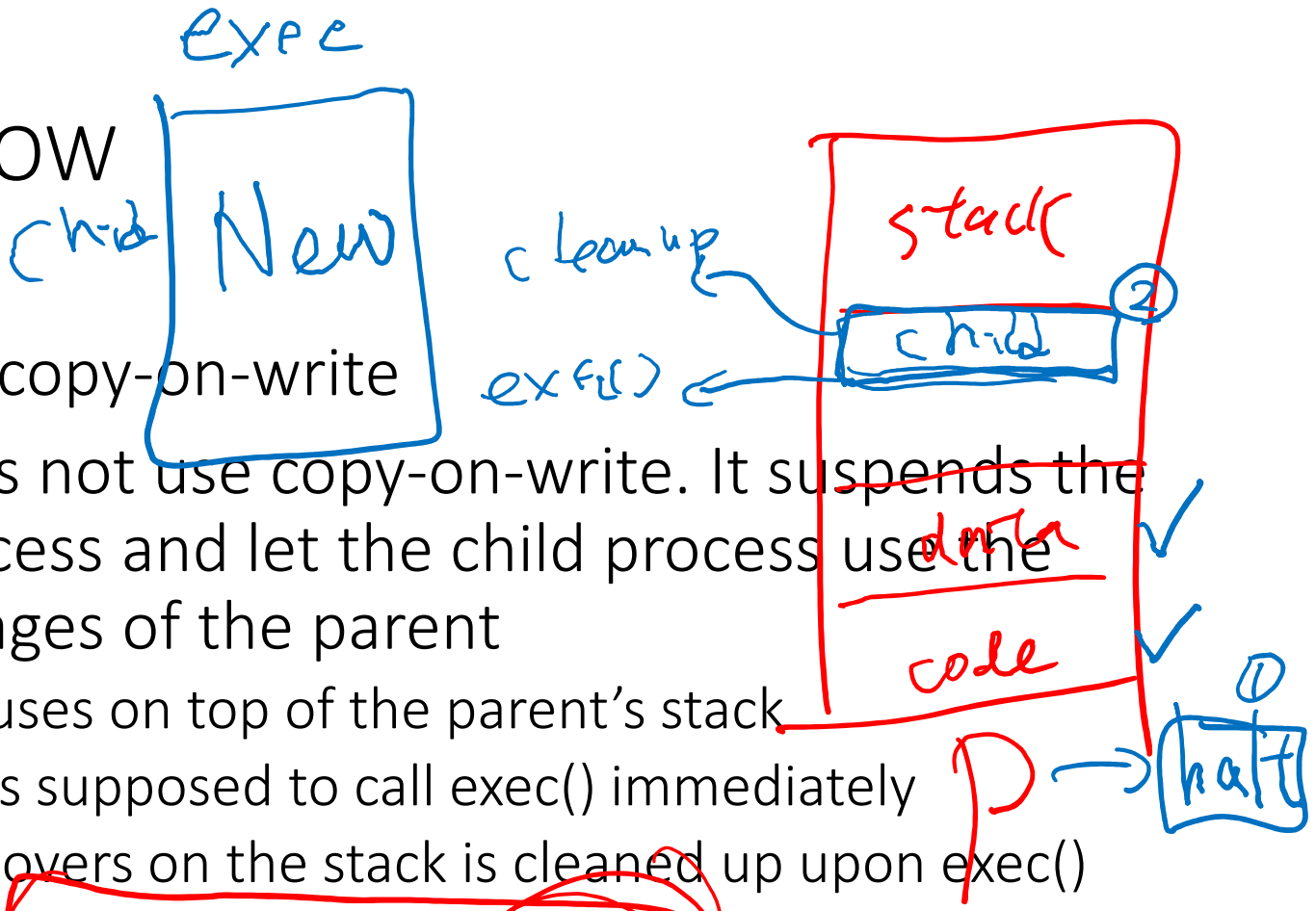
Before process P modifies Page 3



After process P modifies Page 3

https://www.geeksforgeeks.org/copy-on-write/

fork() and COW

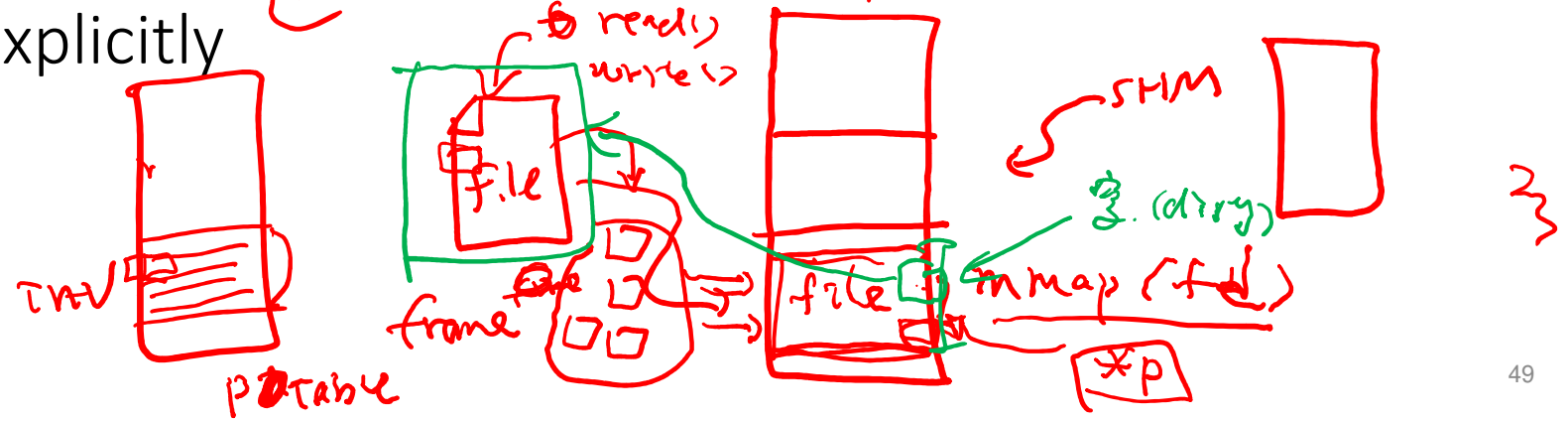


- fork() uses copy-on-write
- vfork() does not use copy-on-write. It suspends the parent process and let the child process use the memory pages of the parent
  - The child uses on top of the parent's stack
  - The child is supposed to call exec() immediately
  - Child's leftovers on the stack is cleaned up upon exec()
- vfork() is for CPUs without a MMU. Otherwise, fork() with COW is efficient enough

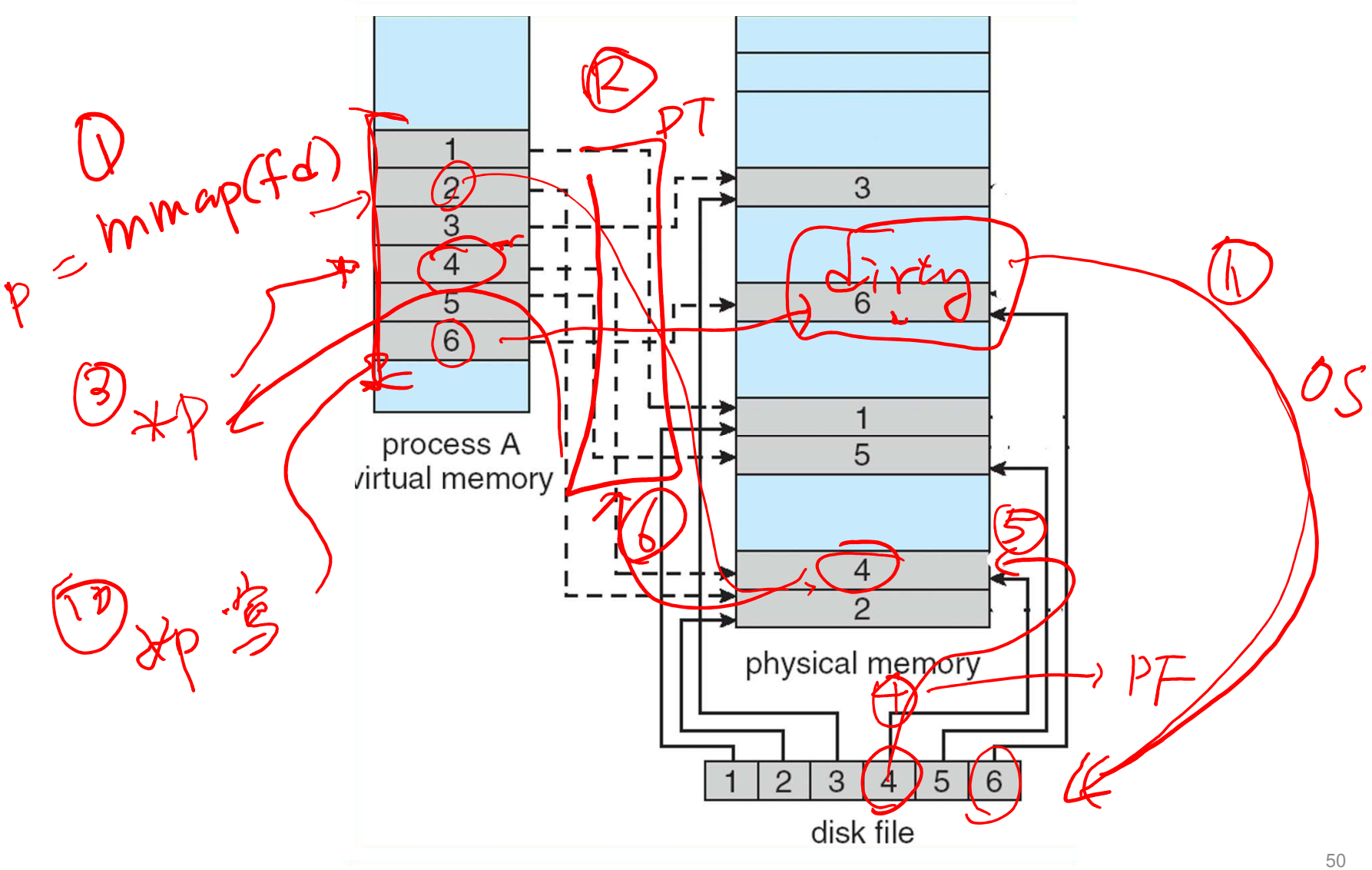
fork() → exec()

Memory-Mapped Files

- A segment of virtual memory that is linearly mapped to a disk file
- Reading the memory segment triggers page faults, which brings a page-sized portion of the file through demand paging
- Writing the memory segments makes pages dirty; dirty pages are flushed to disk file in background or explicitly

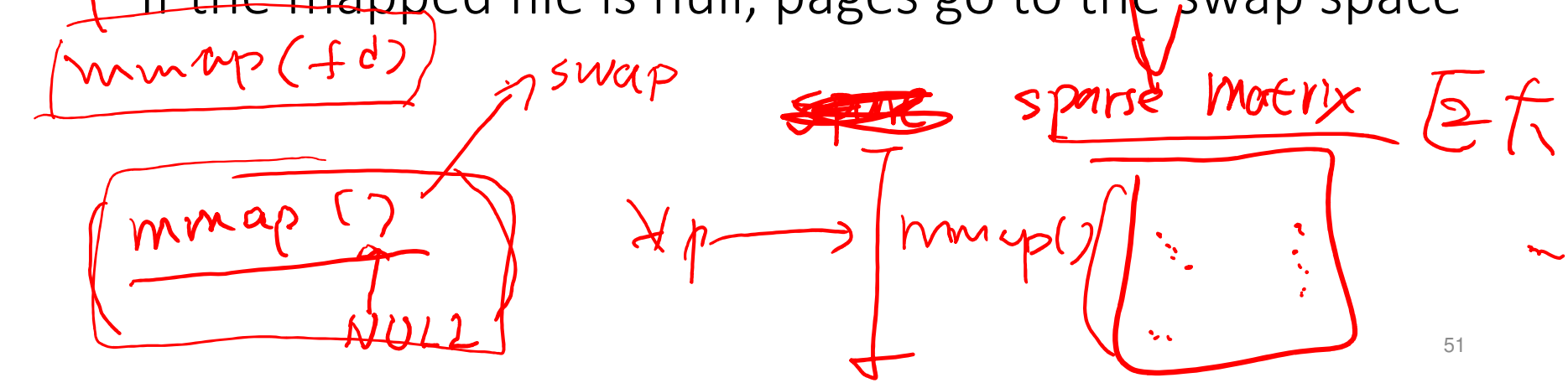


Memory Mapped Files



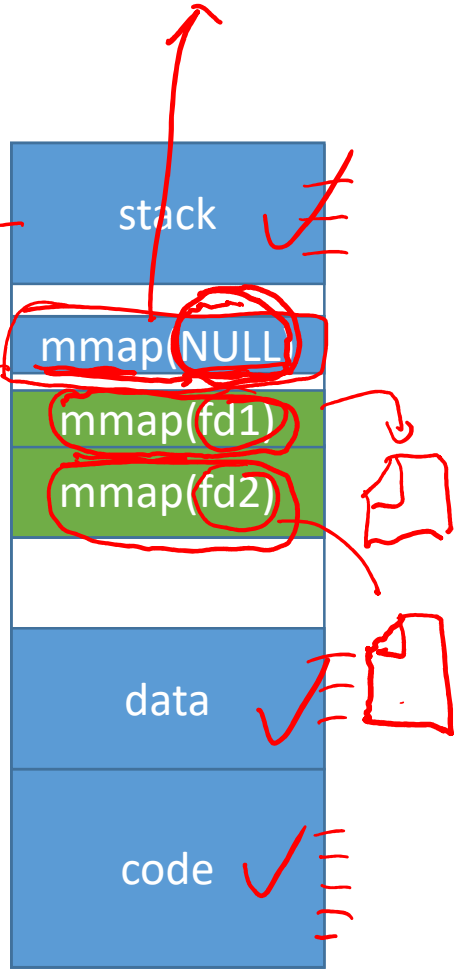
Memory-Mapped Files

- Simplifies file access by treating file I/O through memory rather than read() write() system calls
  - Simple byte-level memory operations
  - No need to enter the kernel
- Shared memory can be mapped to a file as a memory-mapped file
- If the mapped file is null, pages go to the swap space



Mapping of Pages

- Anonymous pages belong to memory segments of processes, such as data, stack, and code segments
- File-mapped pages belong to memory regions that are mapped to a file
- Anonymous pages are backed by the swap device; file-mapped pages are backed by files
- They share the same mechanism for paging in and paging out

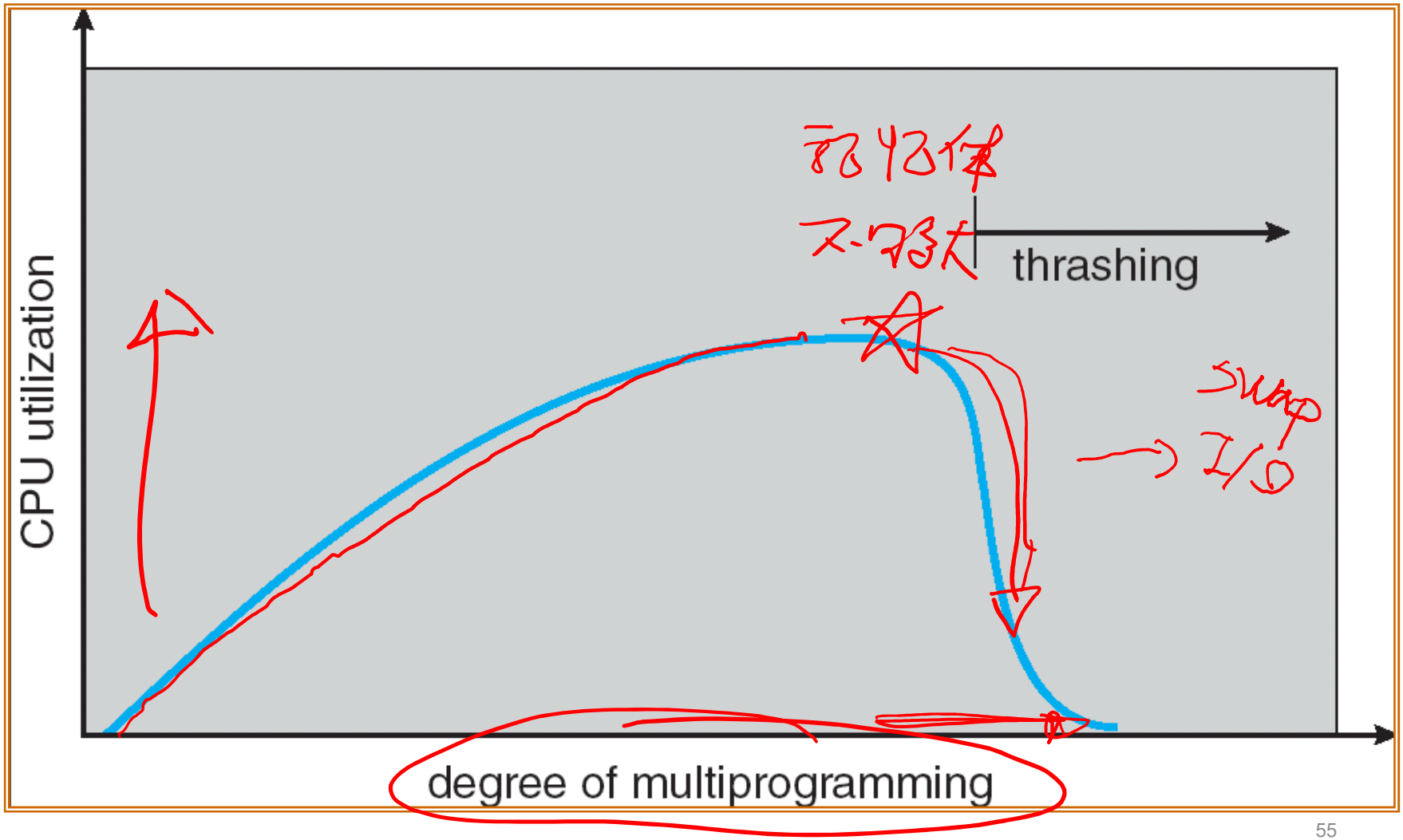


PERFORMANCE ISSUES

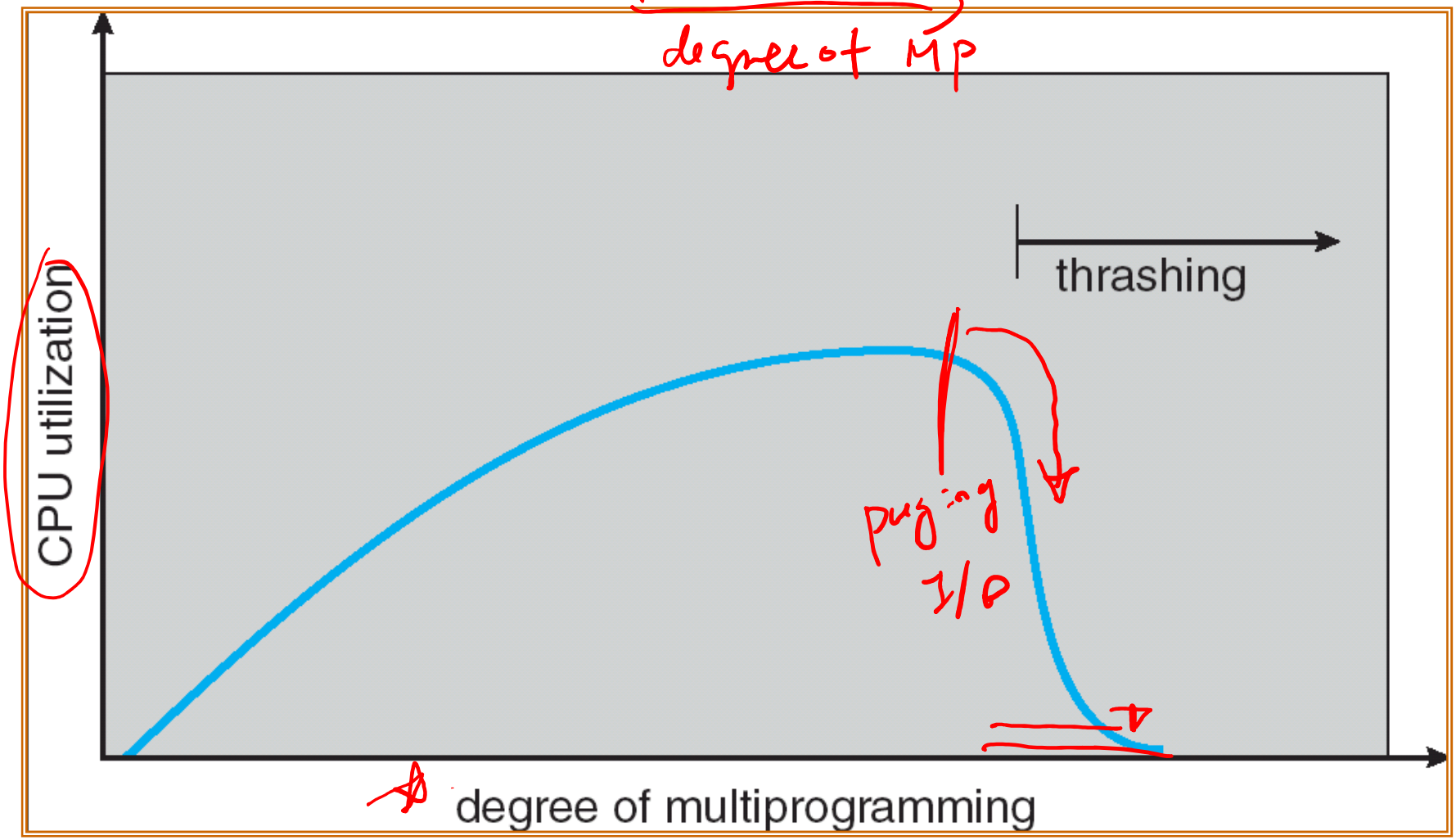
Thrashing

- If a process does not have "enough" pages, the page-fault rate will be very high. This leads to:
  - a low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process is added to the system
- Thrashing ≡ a process that is busy swapping pages in and out

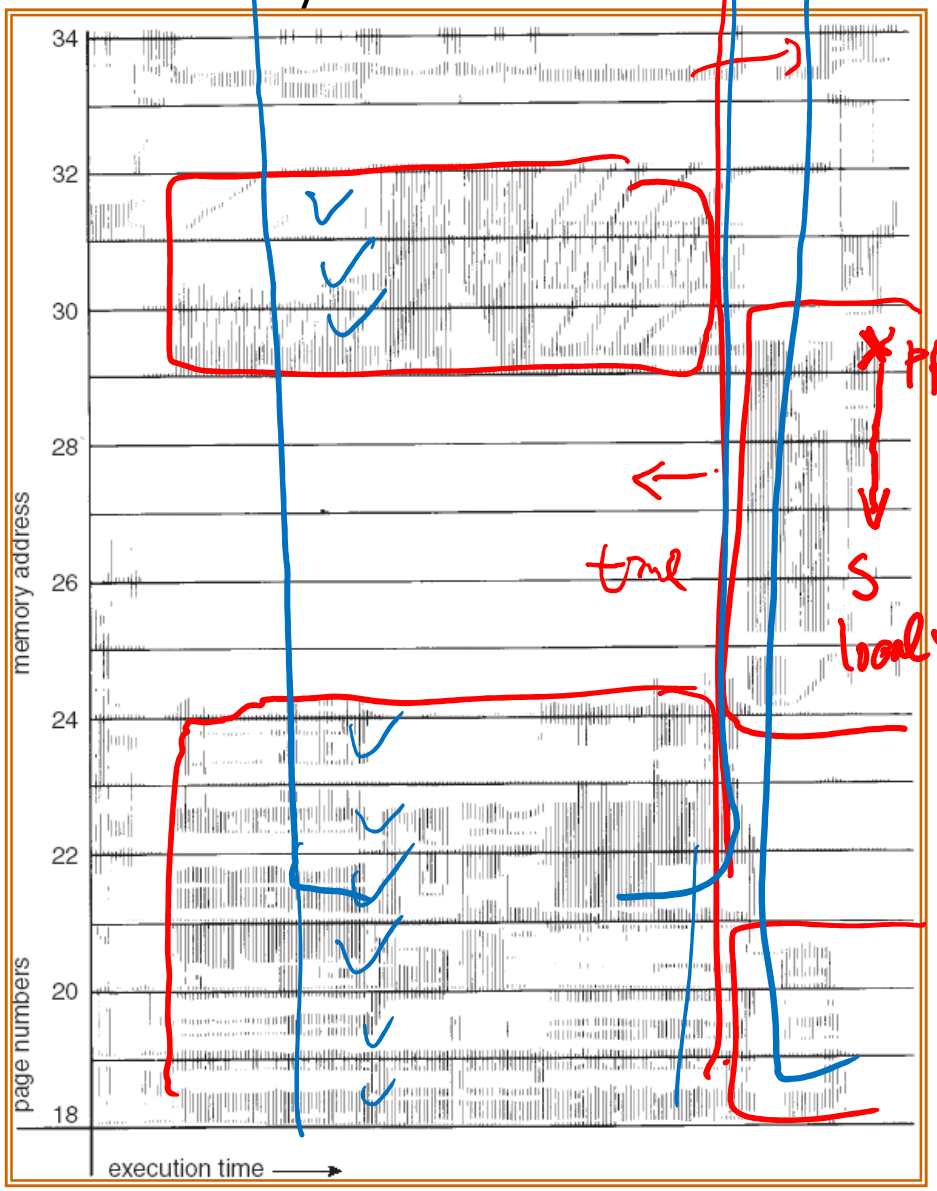
Thrashing (Cont.)



Thrashing (Cont.)



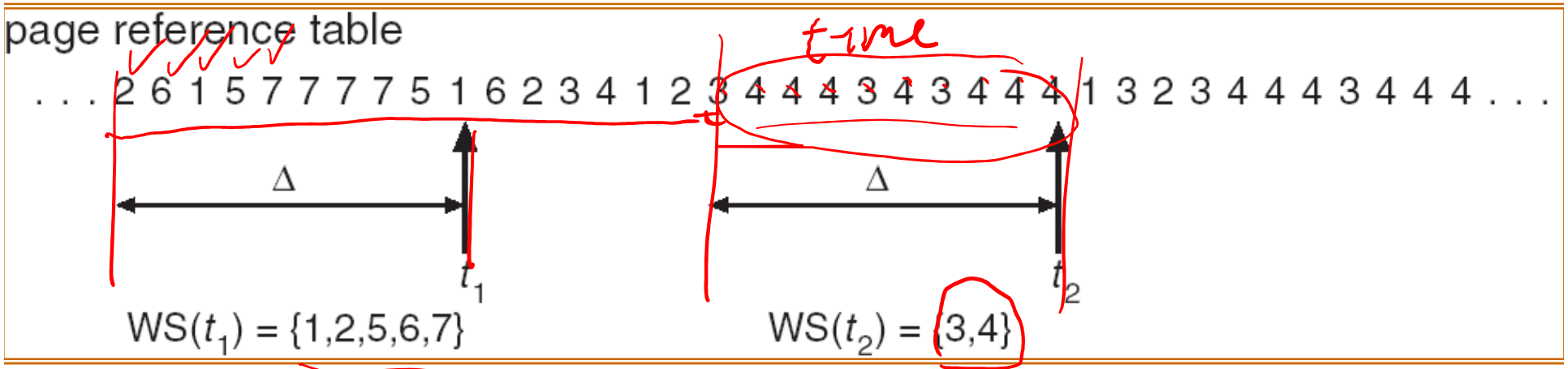
Locality In A Memory-Reference Pattern



Working-Set Model

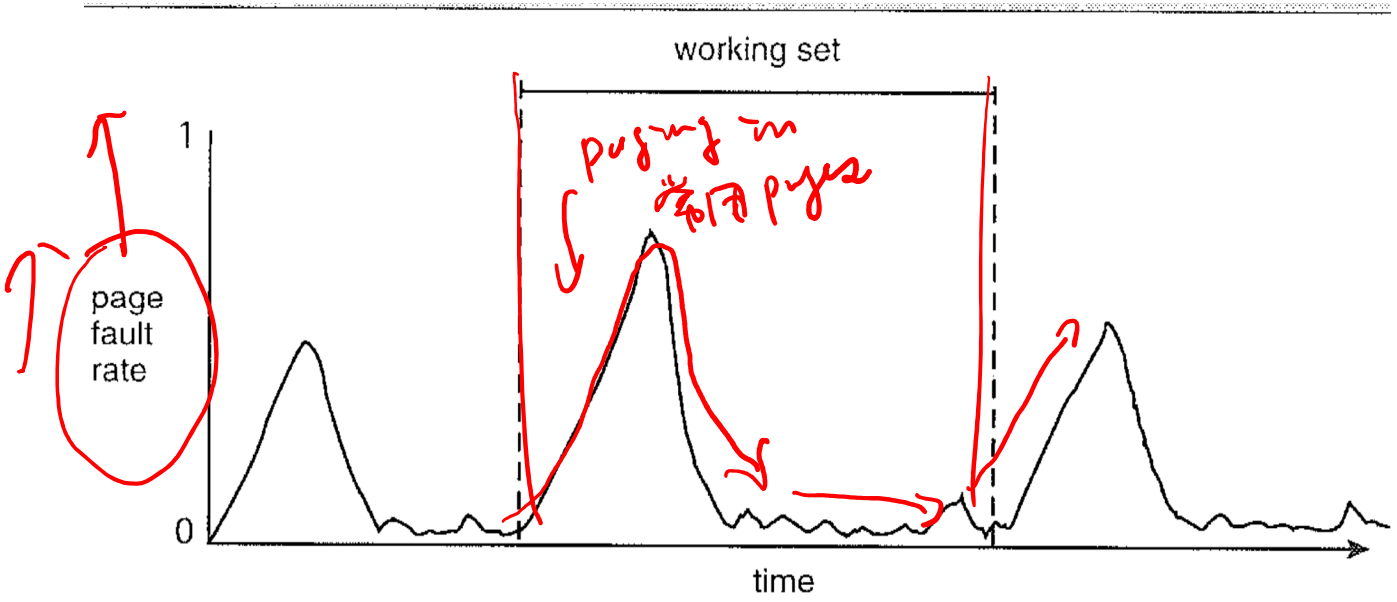
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- WSSi (working set of Process Pi) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D \equiv \Sigma$ WSSi $\equiv$ total demand frames
- if D > m $\Rightarrow$ Thrashing
  - m is the total number of frames
- Policy if D > m
  - Swap out some processes
  - De-allocate pages from processes

Working-set model



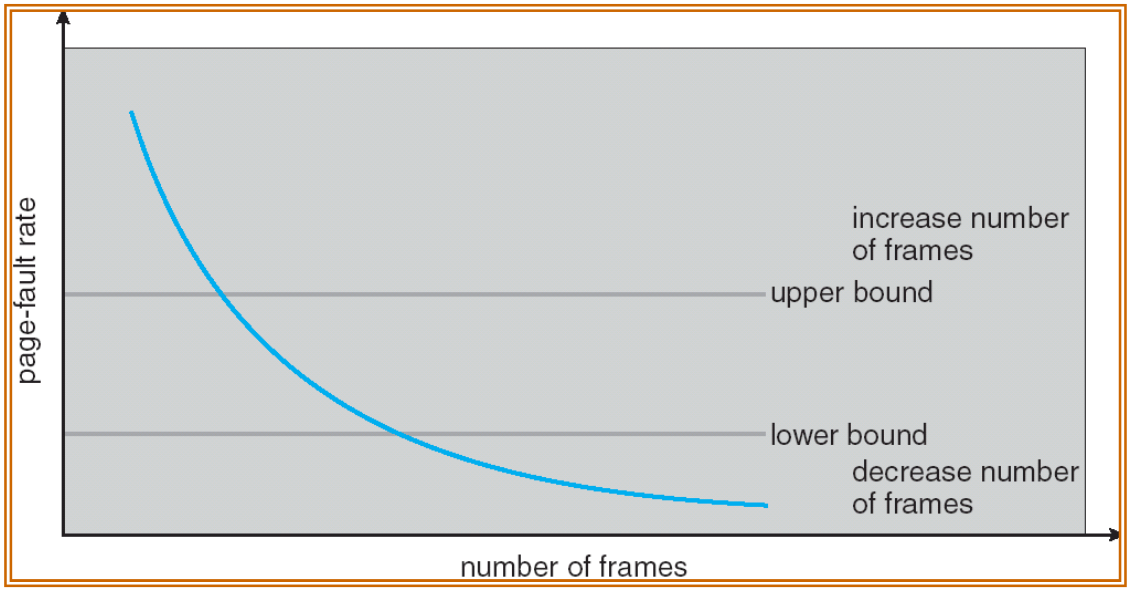
Remark: Migration of Working Sets

- Working set is a time-variant, when a process migrates from a working set to another, the page fault rate also increases (but not thrashing)



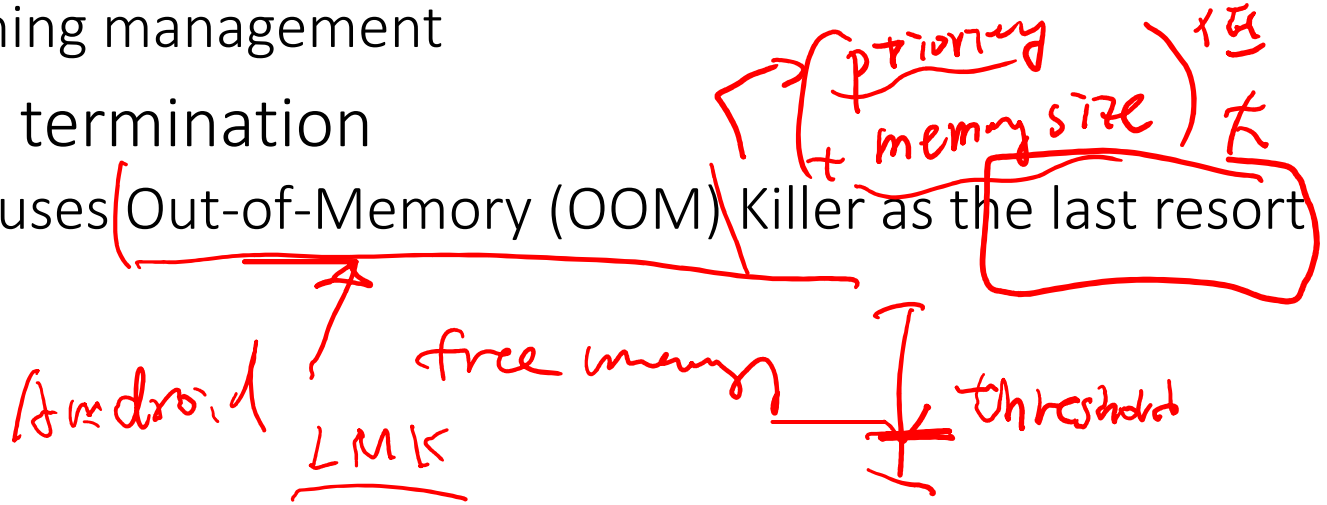
Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
- If actual rate too low, process gains frame
  - Swap in more processes
- If actual rate too high, process loses frame
  - Swap out some processes or discard some pages

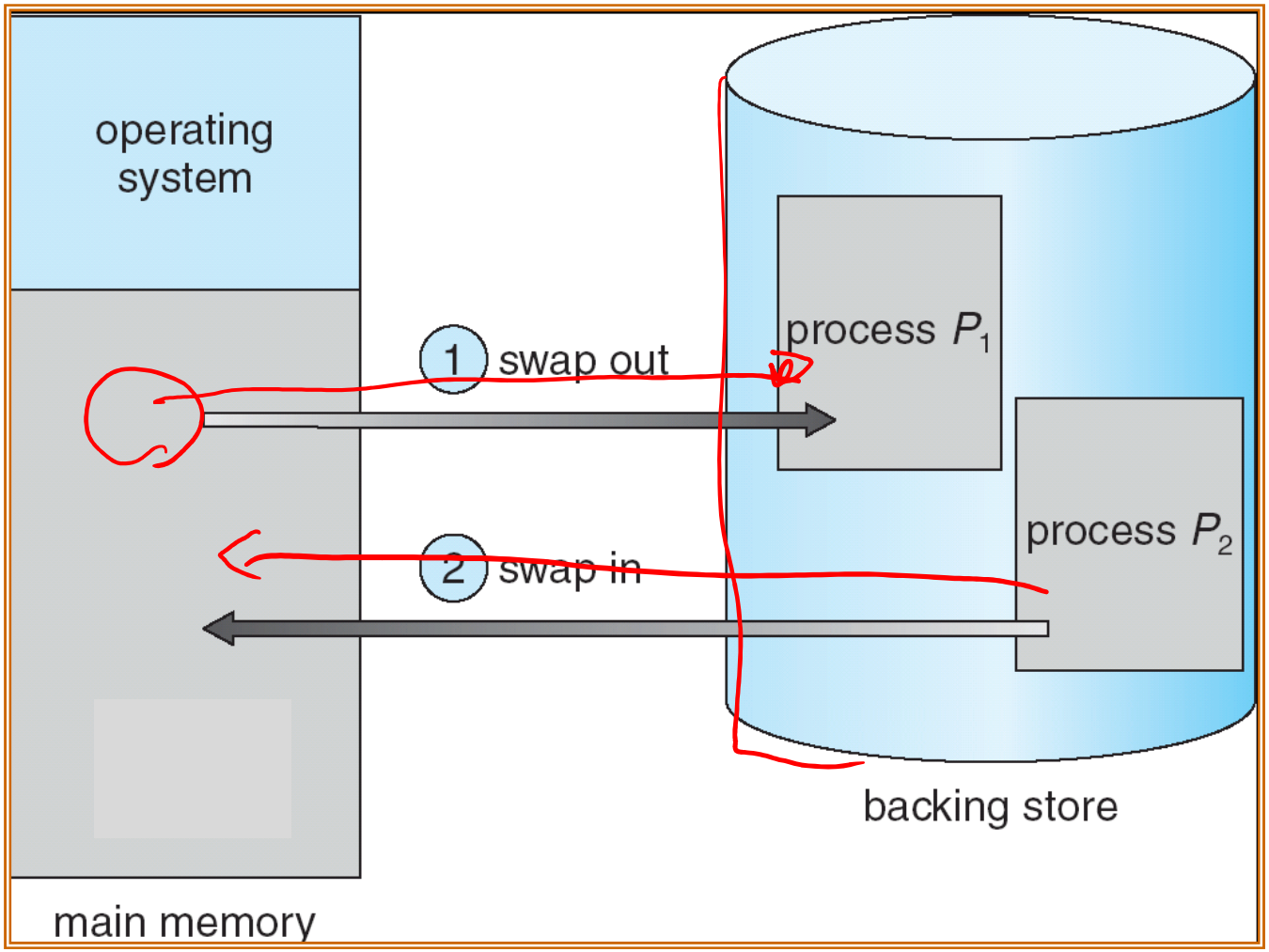


Thrashing Management

- Swapper
  - AKA Midterm scheduler; process-based swapping
  - A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution
  - Memory of a swapped-out process are released; useful to thrashing management
- Process termination
  - Linux uses Out-of-Memory (OOM) Killer as the last resort

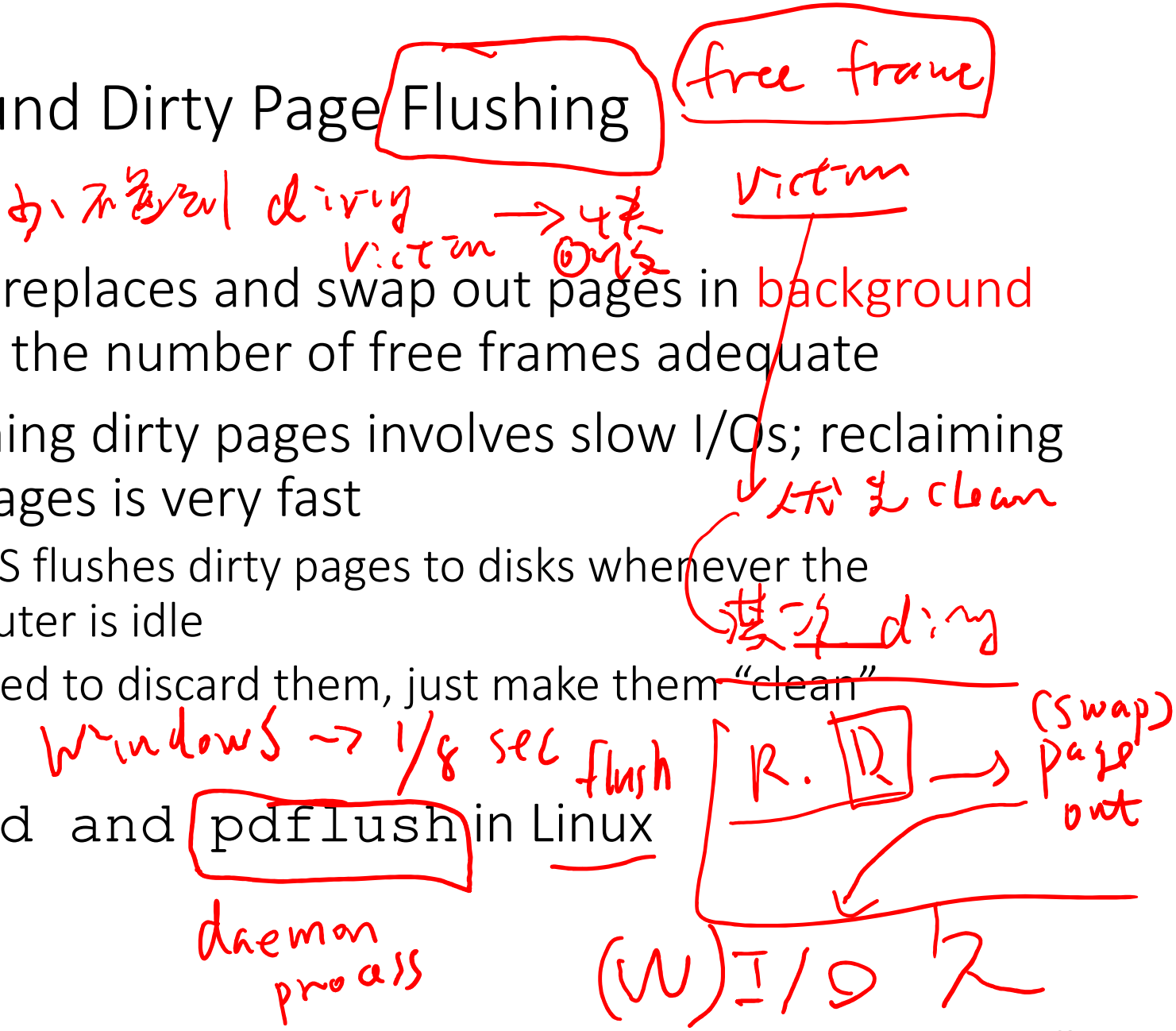


Schematic View of Process Swapping



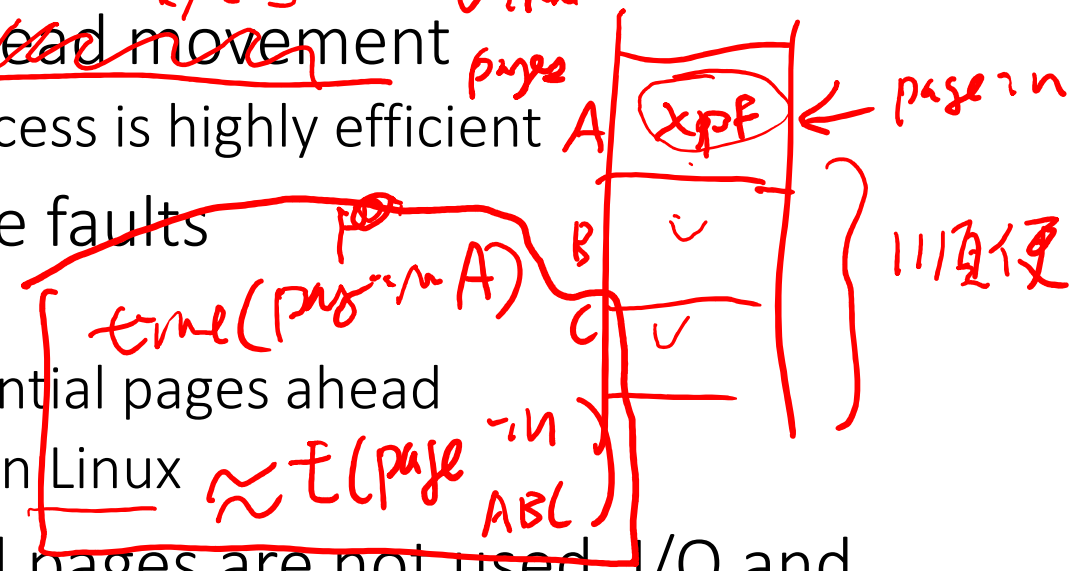
Background Dirty Page Flushing

- The OS replaces and swap out pages in background to keep the number of free frames adequate
- Reclaiming dirty pages involves slow I/Os; reclaiming clean pages is very fast
  - The OS flushes dirty pages to disks whenever the computer is idle
  - No need to discard them, just make them "clean"
- `kswapd` and `pdflush` in Linux



Pre-paging (Pre-fetching)

- On a page fault, read a number of pages ahead of the requested page
- Pros 1: Less disk head movement
  - Sequential disk access is highly efficient
- Pros 2: Fewer page faults
  - Spatial locality
  - Fetch a few sequential pages ahead
  - 128 KB=32 pages in Linux
- Cons: If pre-paged pages are not used, I/O and memory were wasted



Program Structure

- Program structure
  - Int[128,128] data;
  - Each row is stored in one page
  - Program 1

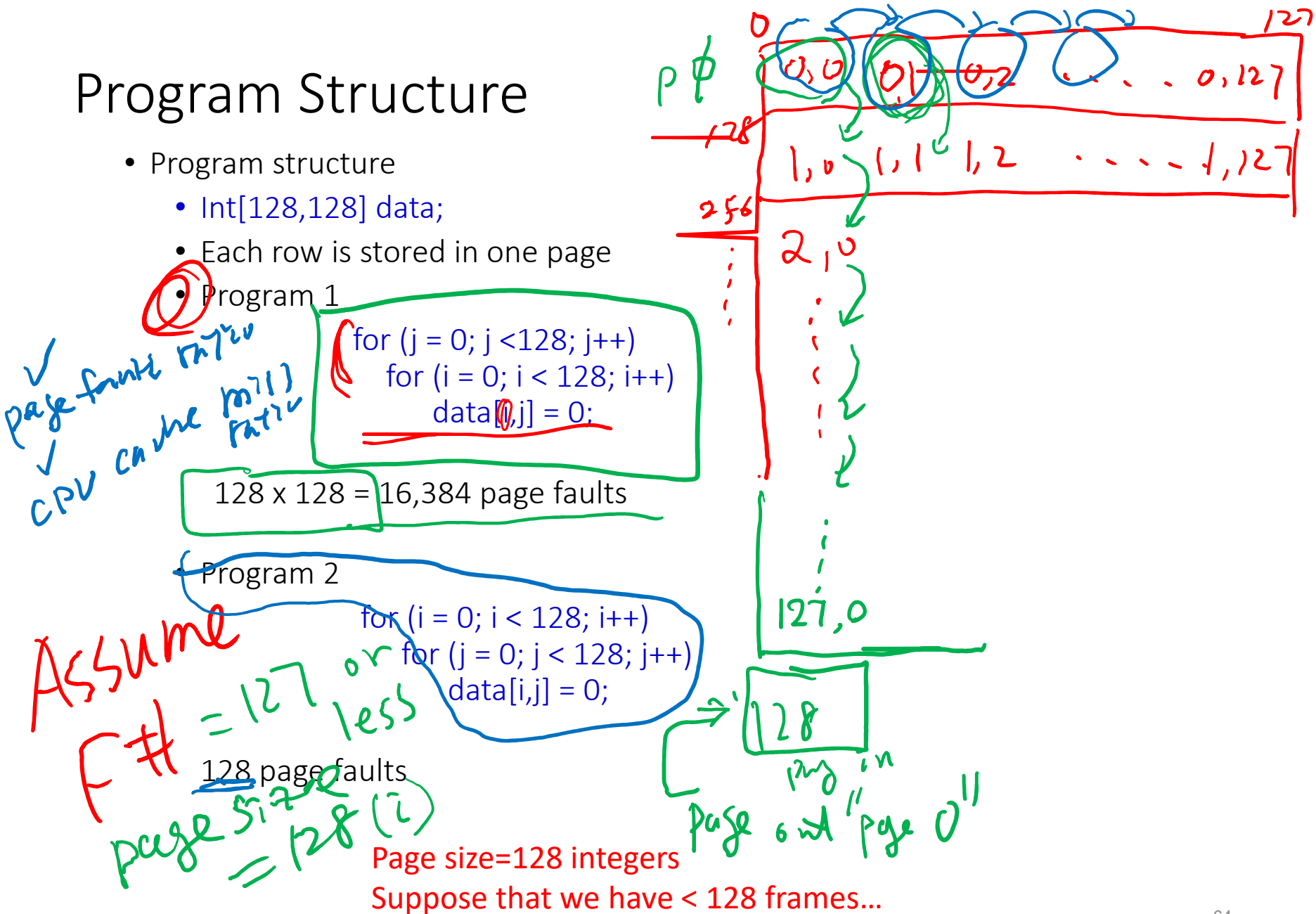
```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

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;
```

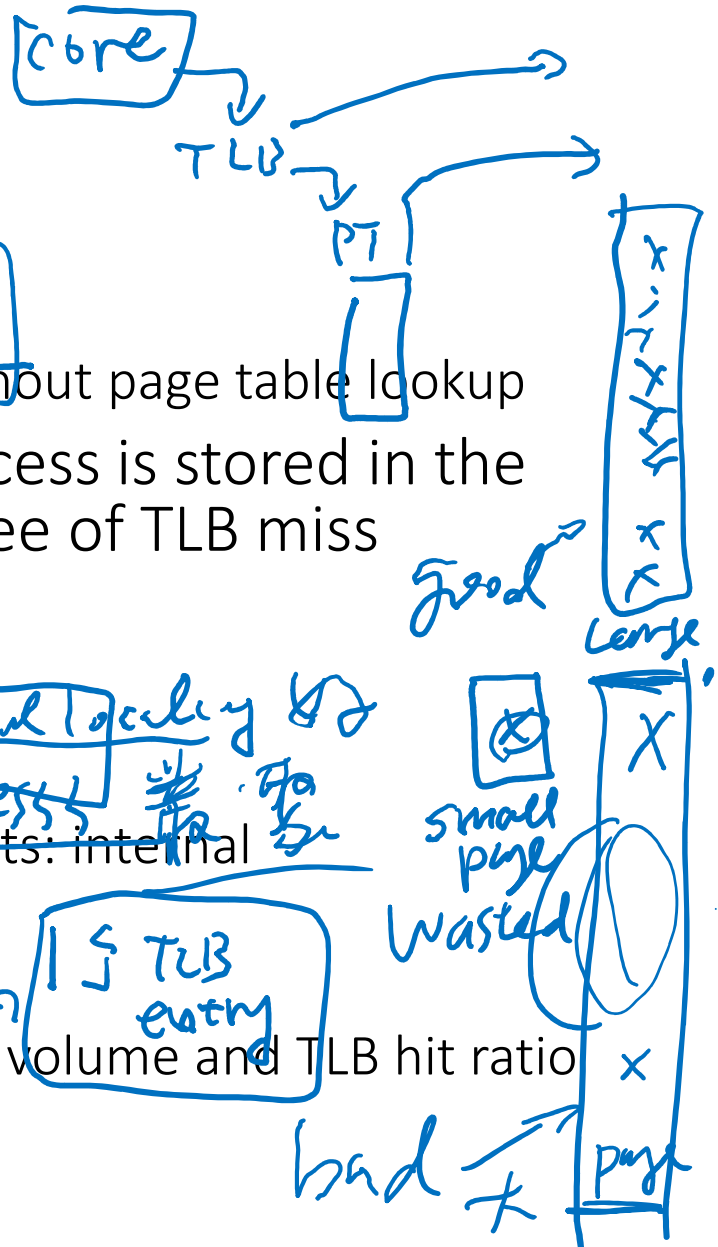
128 page faults



Page size=128 integers

Suppose that we have < 128 frames…

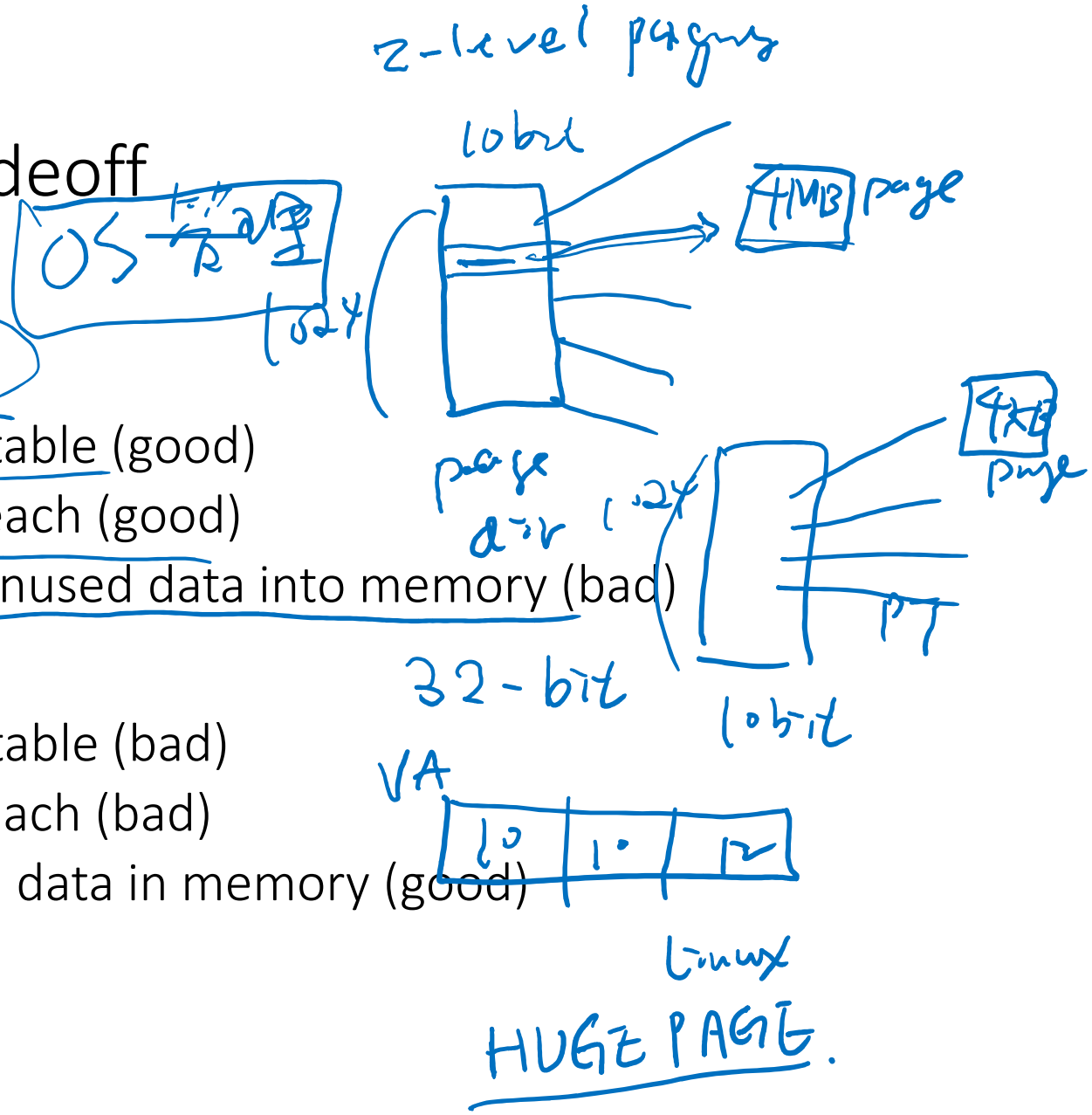
TLB Reach & Large Pages



- TLB Reach = (TLB Size) X (Page Size)
  - The amount of memory accessible without page table lookup
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of TLB miss
- Increase the TLB size
  - Impractical !! Too expensive
- Increase the Page Size
  - May have negative performance impacts: internal fragmentation, more I/O traffic, etc.
- Provide Multiple Page Sizes
  - For a good balance between I/O traffic volume and TLB hit ratio
  - IA-64 supports 4 KB and 4 MB pages

Page Size Tradeoff

- Large pages
  - Small page table (good)
  - Large TLB reach (good)
  - May bring unused data into memory (bad)
- Small pages
  - Large page table (bad)
  - Small TLB reach (bad)
  - Less unused data in memory (good)

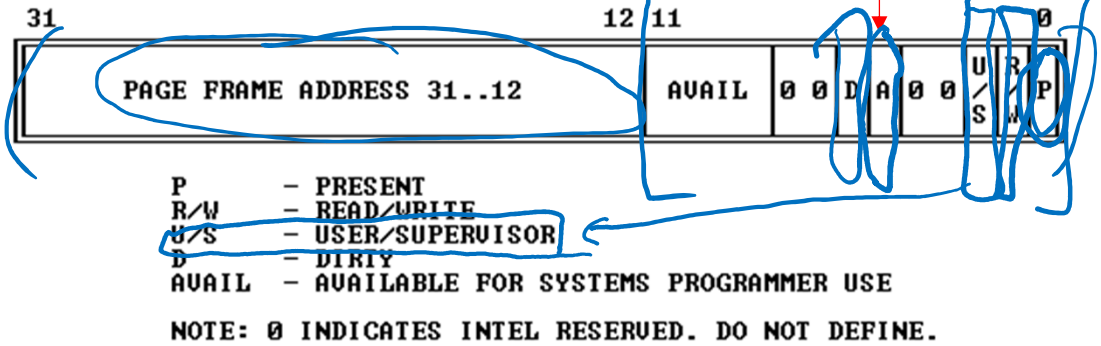


Operating System Examples

Status Bits Associated with a Page (x86)

- Valid (Present) bit: does the page present in main memory?
- RW bit: is the page read-only or read-write?
- Reference bit: is the page referenced?
- Dirty bit: is a page modified?

Figure 5-10. Format of a Page Table Entry



A: referenced

https://pdos.csail.mit.edu/6.828/2007/readings/i386/s05_02.htm#fig5-10

Linux Page Reclaiming

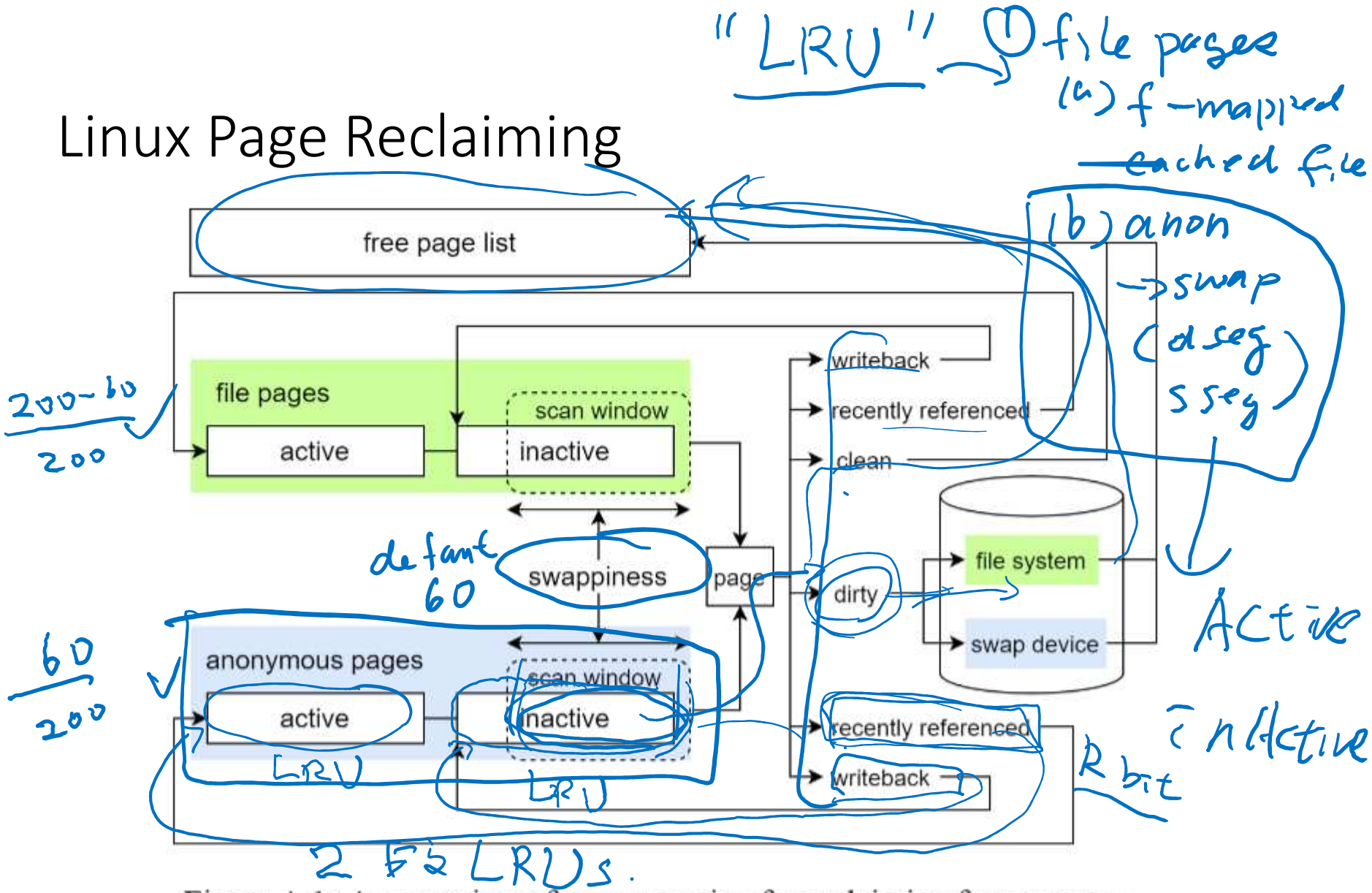
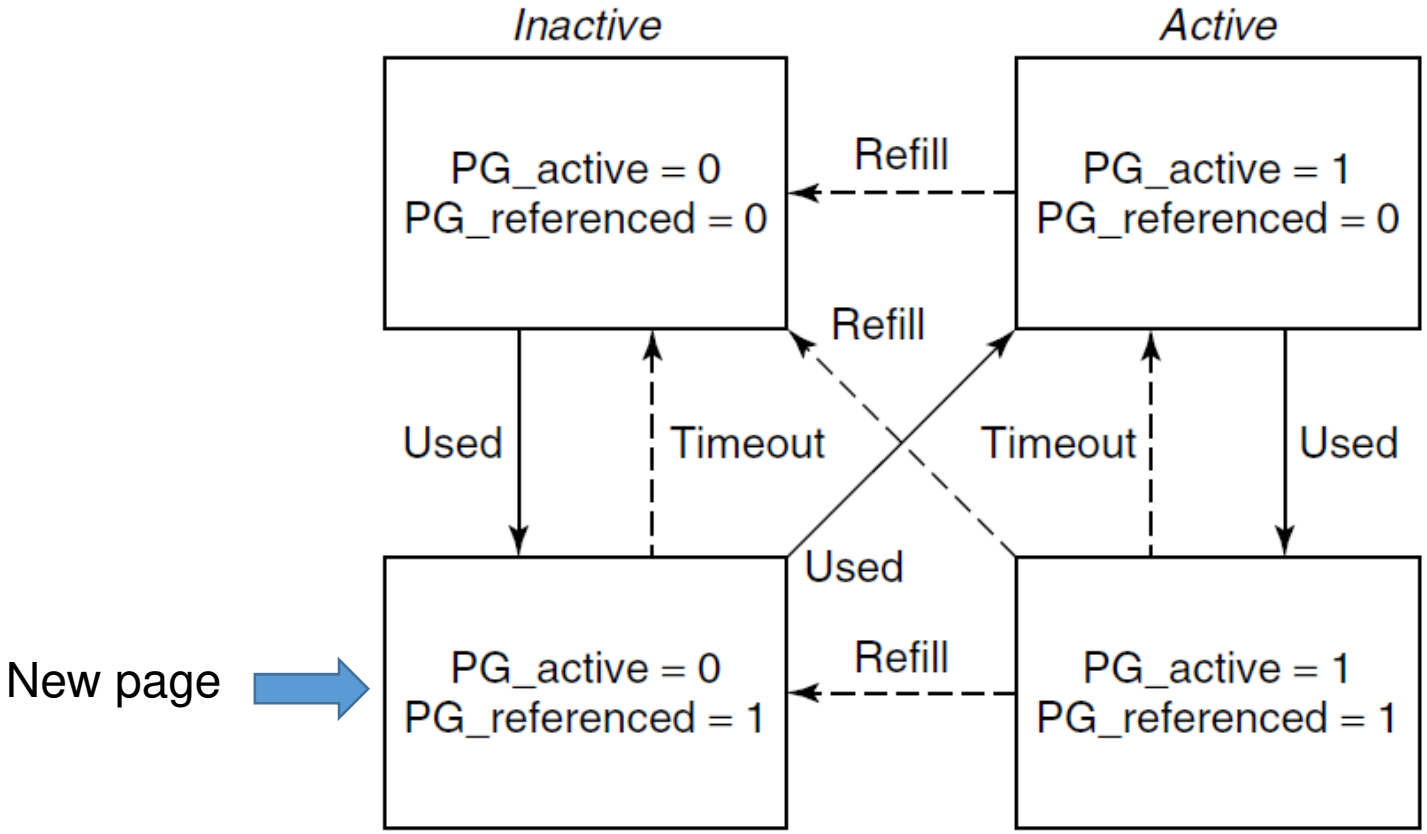


Figure A.1: An overview of page scanning for reclaiming free memory.

-王詠萱，壓縮交換:不以快閃記憶體壽命為代價之安卓裝置使用體驗改善方法，陽明交大碩士論文，2021
-Yong-Xuan Wang, Chung-Hsuan Tsai and Li-Pin Chang, "Killing Processes or Killing Flash? Escaping from the Dilemma Using Lightweight, Compression-Aware Swap for Mobile Devices," International Conference on Embedded Software (EMSOFT 2021)

The Linux Page Replacement Algorithm



PG_active: often accessed (frequency)

PG_reference: recently referenced (recency)

End of Chapter 9