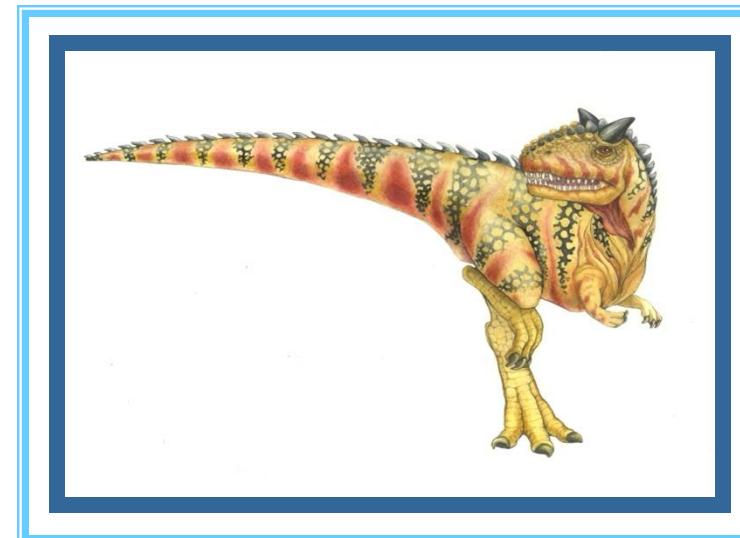
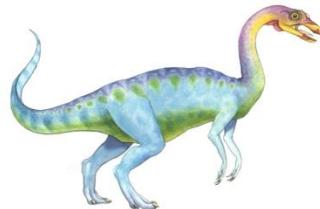


→ real memory

Chapter 10: Virtual Memory

segmentation + paging technique





Chapter 10: Virtual Memory

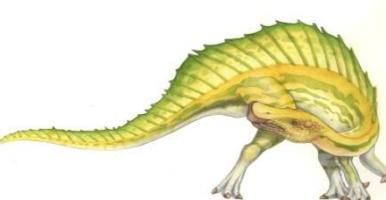
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Frame Allocation
- Thrashing
- Other Considerations

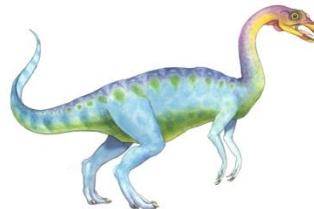
swapping → CPU utilization = 0



- OS only bring required page to the memory.
→ fork(), duplicate the address space.

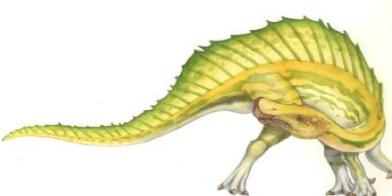
Algorithm, Swap / replace the pages in physical memory:

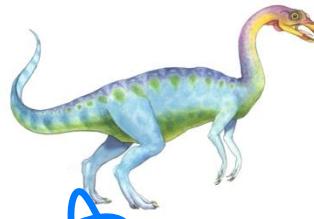




Objectives

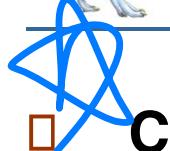
- Describe **virtual memory** and its benefits.
[physical memory] more flexible
- Illustrate how pages are loaded into memory using **demand paging**.
- Apply the ~~FIFO~~, optimal, and LRU page-replacement algorithms. *Algorithm*
(active, (additional information, may not accurate))
- Describe the **working set** of a process and explain how it is related to program locality.
Similar to CPU scheduling like cache!





Background

3.) 加入其他英雄，
2A) load other heroes!



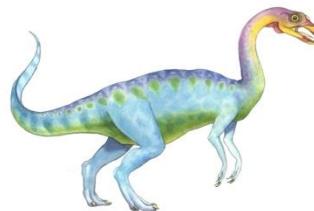
much larger!

Codes need to be in memory to execute, but not necessarily the entire program

- Error code, unusual routines. Some errors seldom, if ever, occur in practice, this code is almost never executed
 - Large data structures such as arrays, lists and tables are often allocated more memory than they need. For example, an array may be declared 100x100 elements, even though it is seldom larger than 10x10 *not everything in memory!*
 - Certain options and features of a program may be used rarely
-
- Even if the entire program is needed, it may not all be needed at the same time
 - Consider ability to execute partially-loaded programs
 - Programs no longer constrained by limits of physical memory. Programs can be written with an extremely large virtual memory address, simplifying the programming task *memory very small!*
 - Each user program could take less physical memory, more programs could be run at the same time, which increases CPU utilization (**degree of multiprogramming**) and throughput
 - Less I/O would be needed to load or swap user programs into physical memory, so each user program would run faster.

没有物理内存，
只有虚拟内存，
所以物理
内存很小！





Background

- Mapping*
- **Virtual memory** – separation of user logical memory (referred as **address space** earlier) perceived by programmers from physical memory
 - Only part of the program needs to be in memory for execution
 - Logical address space can therefore be much larger than the actual physical address space
 - Allows address spaces to be shared by several processes. For instance, system libraries can be shared by several processes
 - Allows more efficient process creation, as pages can be shared during process creation, thus speeding up the process creation
 - More programs running concurrently – increase the degree of multiprogramming
 - Less I/O needed to load or swap processes
 - Virtual memory can be implemented via:
 - Demand paging or demand segmentation – in principle, they are similar, details are different with respect to fix-sized frame/page and variable-sized segment

large address space

but only part will be loaded

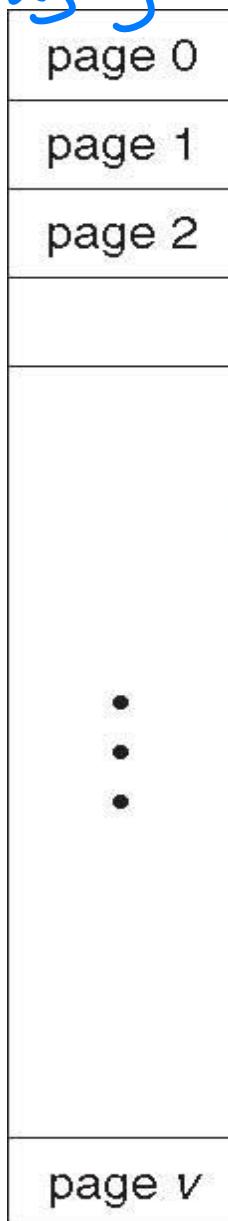




Virtual Memory That is Larger Than Physical Memory

how to implement it in details

Paging



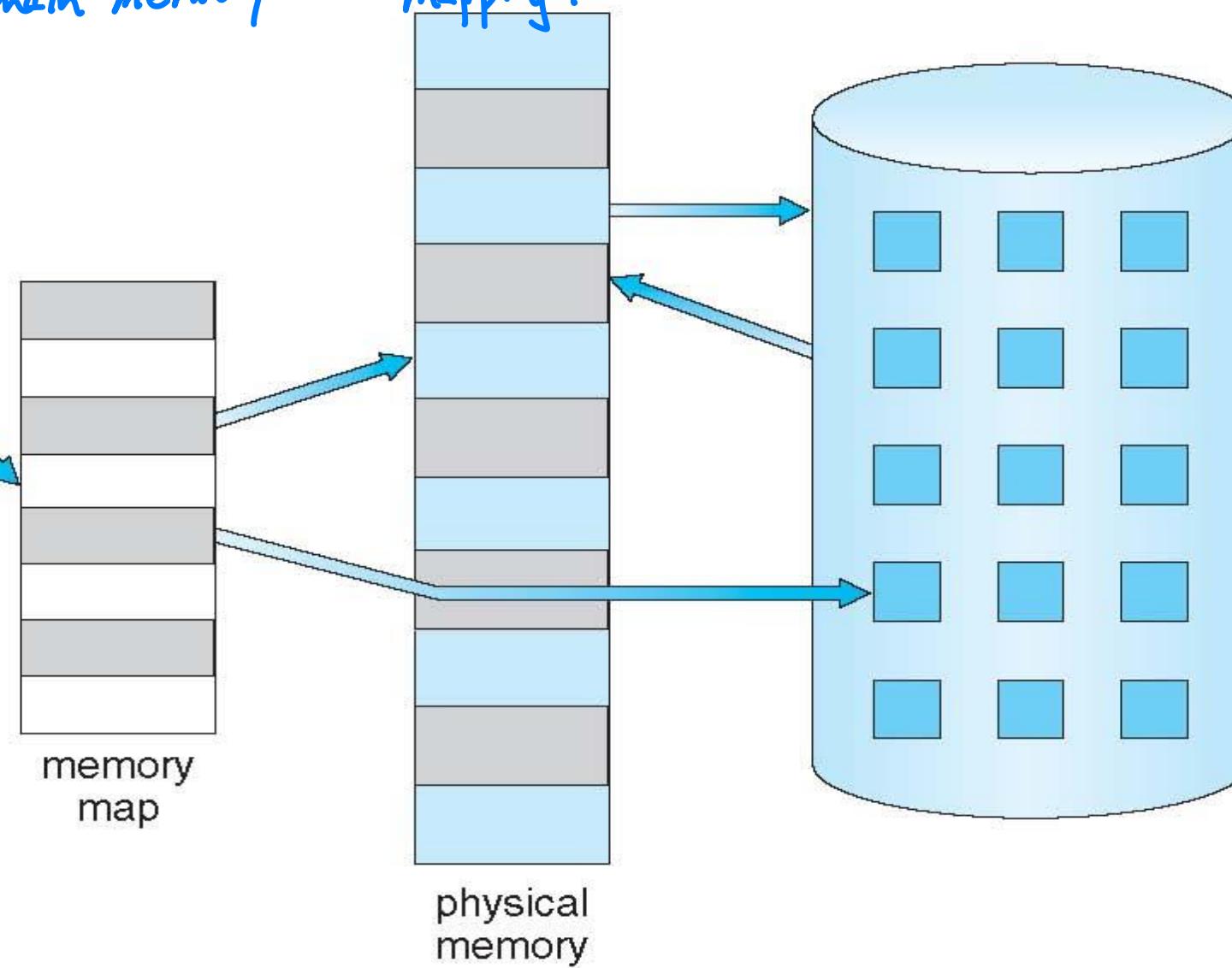
Address space
of process!

Swapping layer:

(hard disk) ↗

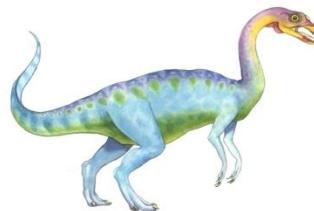
↗ to main memory!

mapping!



delay the mapping if necessary!

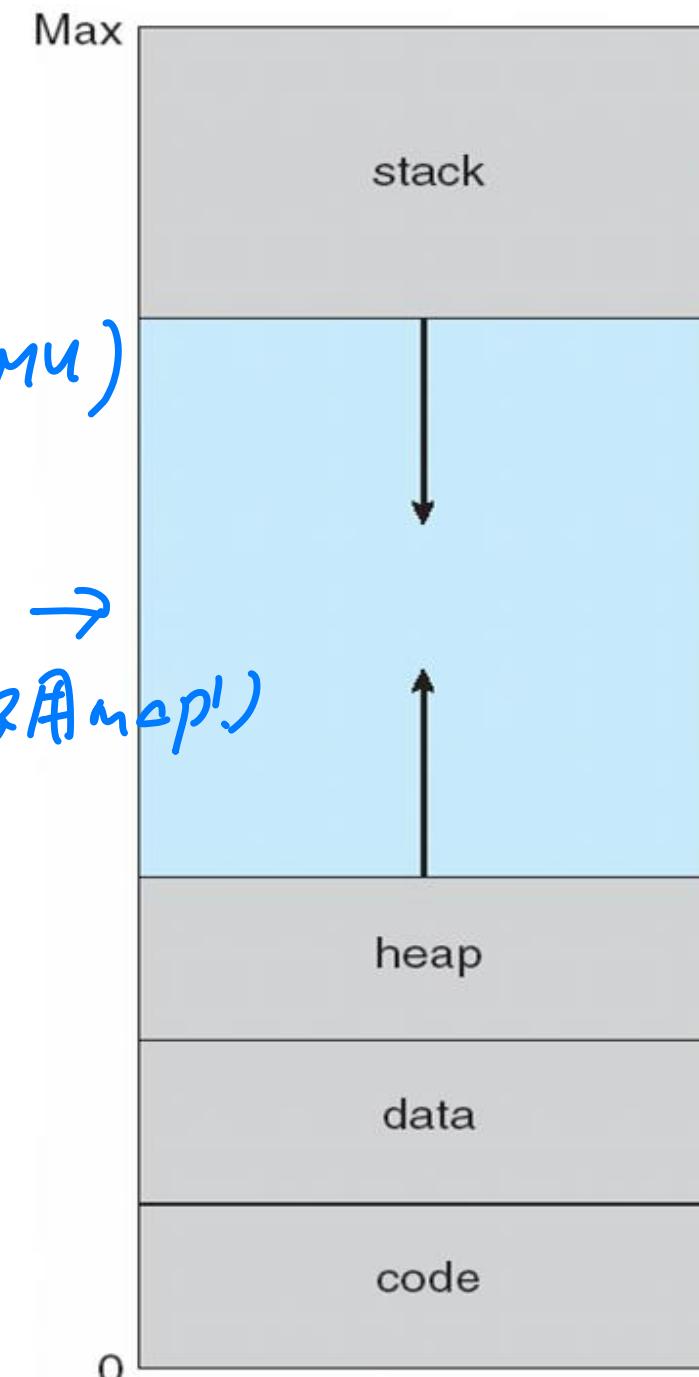




Every process

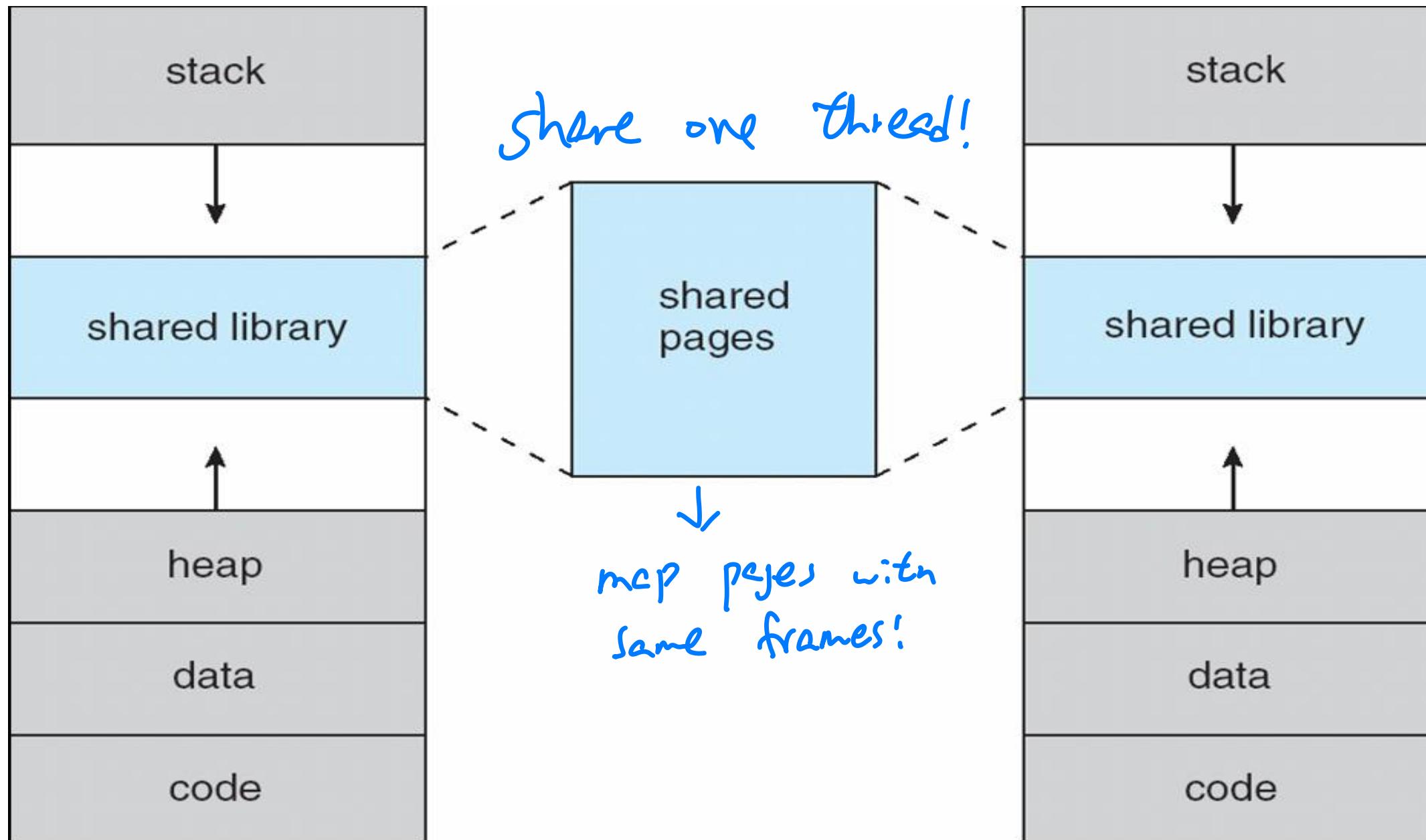
Virtual-address Space

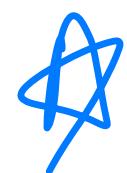
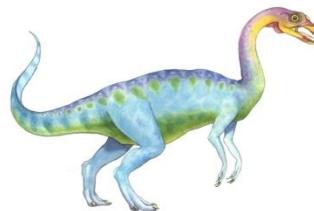
- **Virtual address space** – logical view of how process is stored in memory
 - Usually start at address 0, **contiguous** addresses until the end of address space
 - help! □ Meanwhile, physical memory organized in frames
 - MMU must map logical to physical address (*done by MMU*)
- Heap can grow upward in memory, used in dynamic memory allocation. Stack can grow downward in memory through successive function calls
- The large blank space (or hole) between the heap and stack is part of the virtual address space but will *(这些不分配!)* require actual physical pages (space) only if the heap or stack grows.
- Enables sparse address spaces with holes left for growth, dynamically linked libraries, and etc
- System libraries can be shared via mapped into virtual address space
- Pages **can be shared** during process creation with the `fork()` system call, **speeding up process creation**





Shared Library Using Virtual Memory



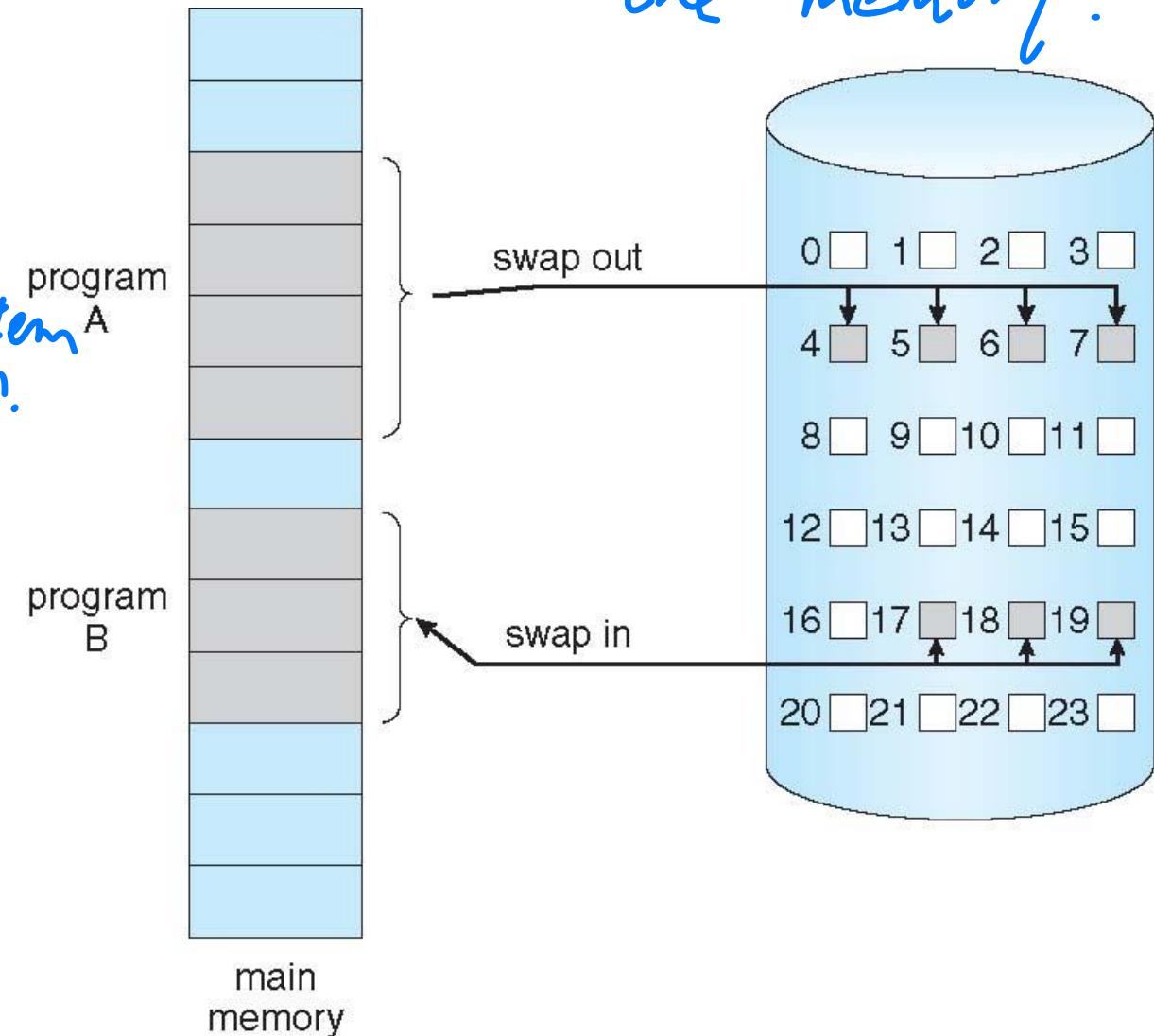


Demand Paging

- Could bring entire process into memory at load time
- Or bring a page into memory only when the page is needed or referenced
 - Less I/O needed, no unnecessary I/O
 - Less memory needed
 - Faster response
 - More users to be running
- Similar to paging system with swapping (diagram on right)
- Page is needed => reference to it
 - invalid reference (illegal memory address) => abort (cancel!)
 - not-in-memory => bring to memory
- **Lazy swapper** – never swaps a page into memory unless page is needed
 - Swapper that deals with pages is a pager

≈ paging system with swap!

≠ bring entire program to the memory!



≠, put into secondary storage!

Access page → check whether in the memory!!!





Basic Concepts

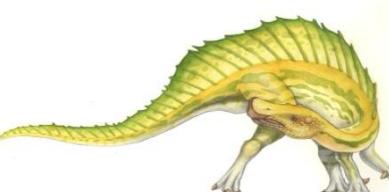
previous) only do address translation, now

- **Pager** brings in only those “needed” pages into memory
- How to determine the set of pages brought inside memory?
 - Need new MMU functionality to implement demand paging
- If pages needed are already memory-resident!
 - No difference from non demand-paging MMU
- If pages needed are not memory resident
 - Need to detect and load the page into memory from a secondary storage device
 - Without changing program behavior
 - Without programmer needing to change code – application not aware of this

can do some new functions!
enable demand paging!

check TLB, fast memory access!

Done by bst some hardware!





Valid-Invalid Bit

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

whether
the page
is valid
or invalid

| Frame # | valid-invalid bit |
|---------|-------------------|
| | v |
| | v |
| | v |
| | v |
| | i |
| | |
| | i |
| | i |

page table

: invalid, page fault → trigger you to do page swapping!

MMU need implement new logic!

; index'.

= page number.

important! frame number
physical address'.

→ not in memory!

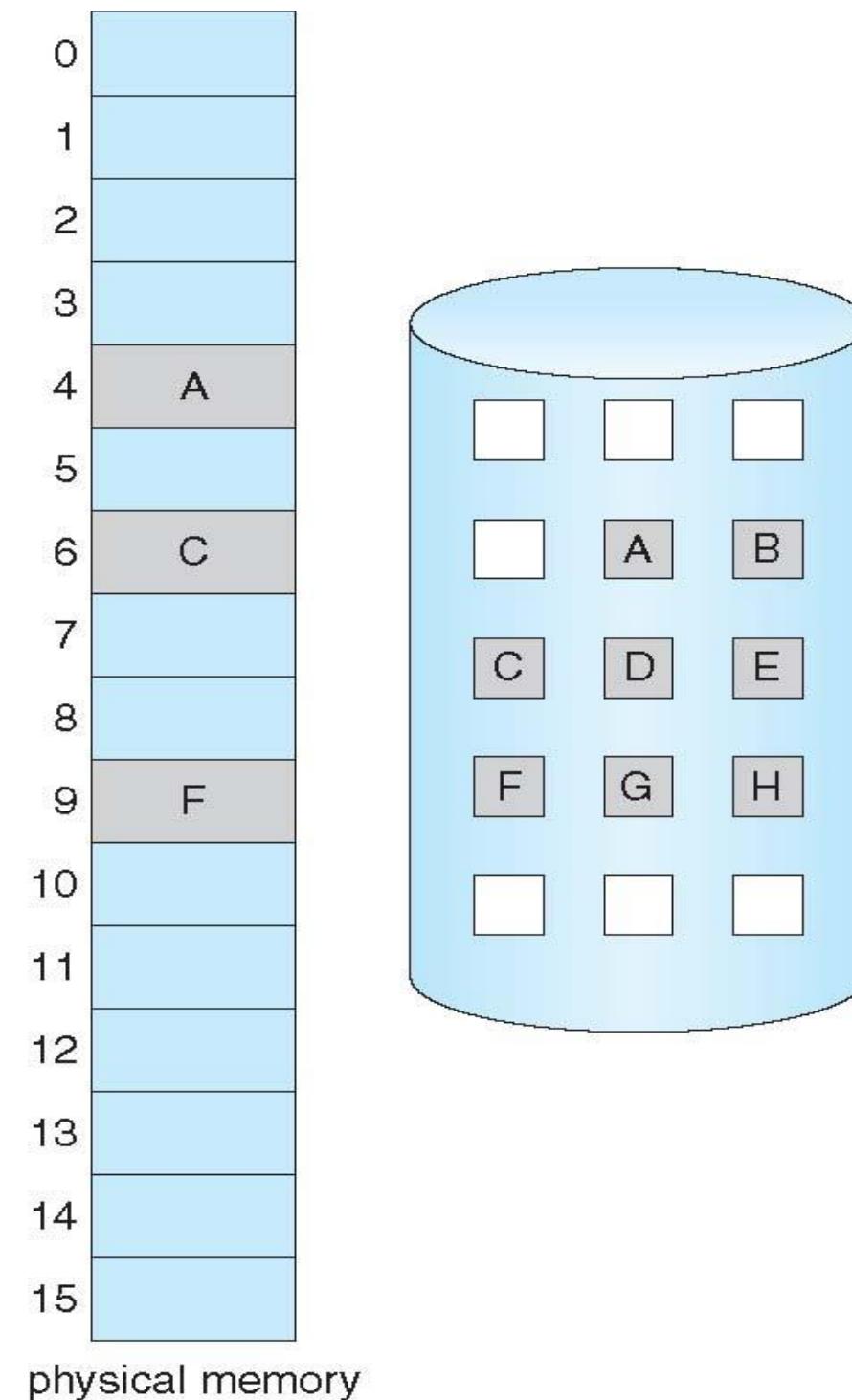
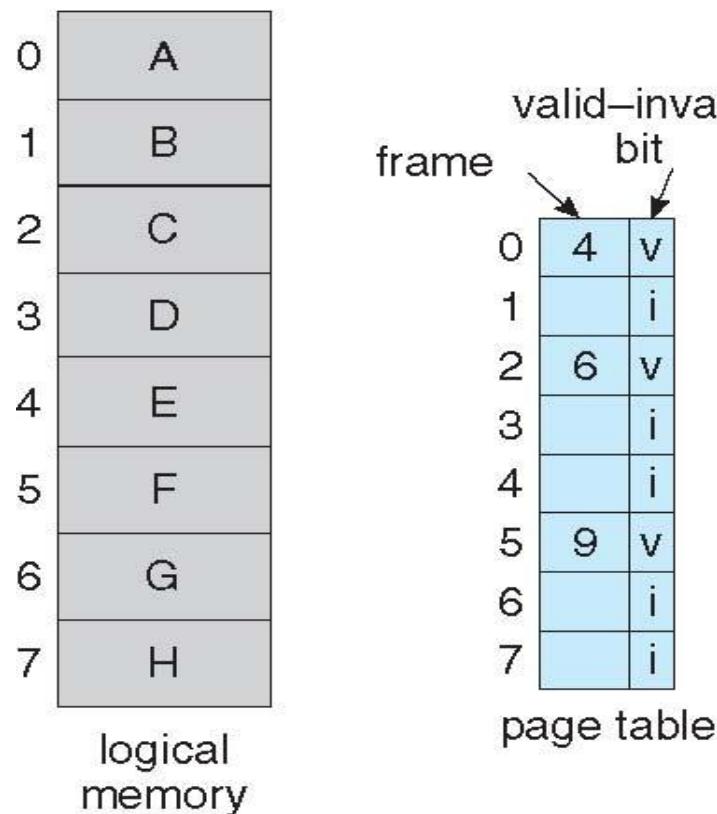
- During address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Initially everything is invalid





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 the corresponding page table entry to decide:
 - Invalid reference (illegal address) \Rightarrow abort
 - Just not in memory
2. Get an empty frame if any \rightarrow Fetch **Empty frames!**
3. Swap the page (on the secondary storage) into **the frame** via scheduled disk operation **phys(c)**
4. Update the corresponding entry in page table **invalid \rightarrow valid!** **free frame from memory!**
5. Reset page table to indicate this page is now in memory,
Set validation bit = **v**
6. Restart the instruction (depending on CPU scheduling) that caused the page fault

rerun the instruction!





Steps in Handling a Page Fault

want to translate
logical address

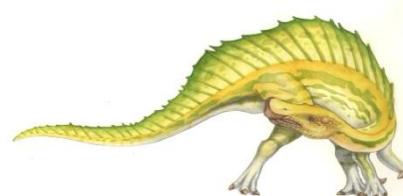
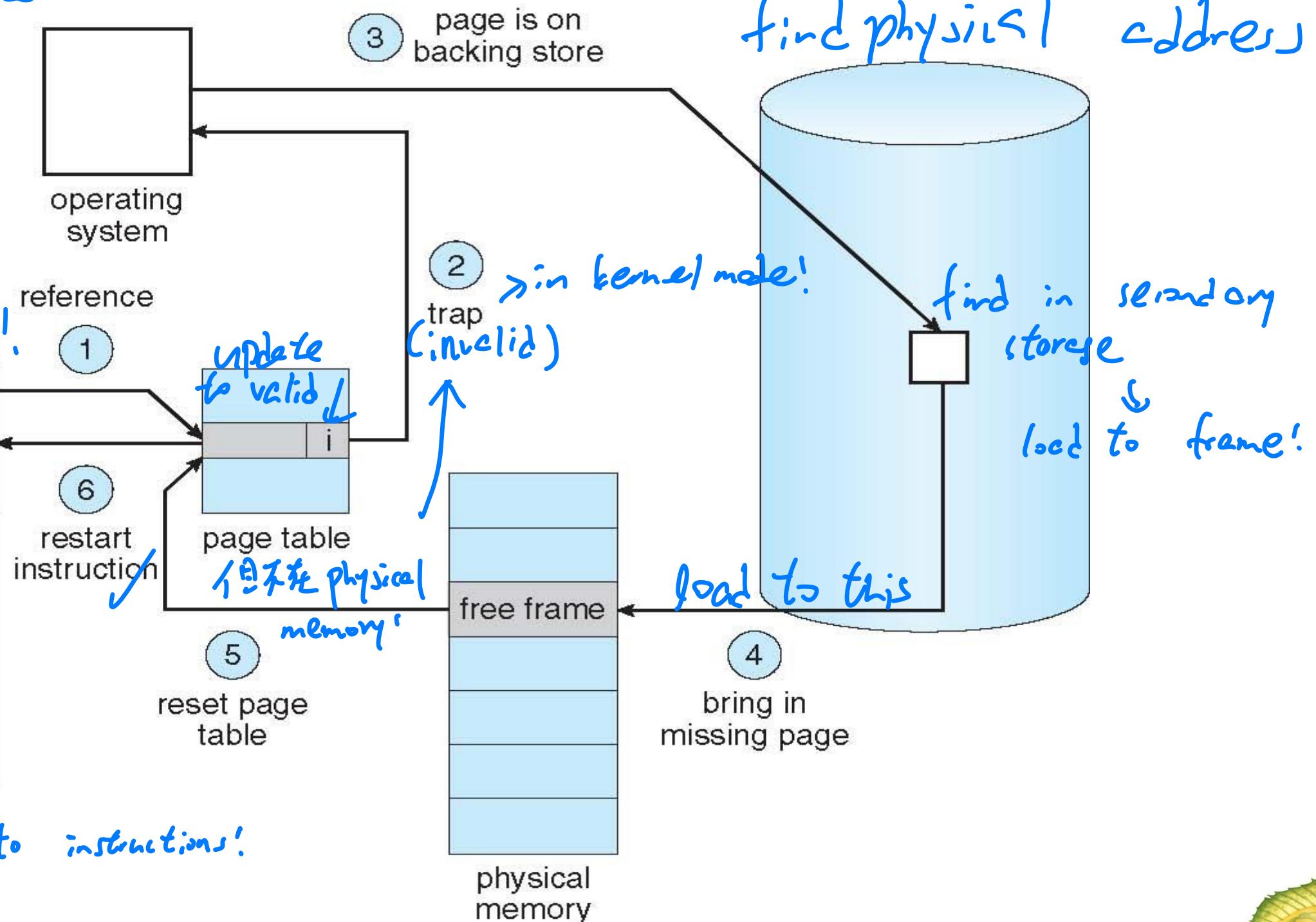
to physical
address

CPU
instructions!

find
using some page!

fetch to instructions!

Enable multiprogramming, overhead!





Aspects of Demand Paging

usually not use this

- The **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 the first access *bring Y_i frame, update the table!*
 - This is referred as **Pure demand paging**
- A given instruction could access multiple pages -> result in multiple page faults
 - Consider fetch and decode of instruction which adds two numbers from memory and stores result back to memory
 - Pain caused by page faults decreases after process starts running for some time because of *locality of memory reference*
- Usually, in order to minimize the initial and potentially high page fault, the OS **pre-page** some pages into the memory before the start of a process execution.
I need this
- **Hardware support** is needed for demand paging
 - Page table with valid / invalid bit as indication
 - Secondary memory (swap device with swap space) for page in and page out
 - Instruction restart

CPU do this!





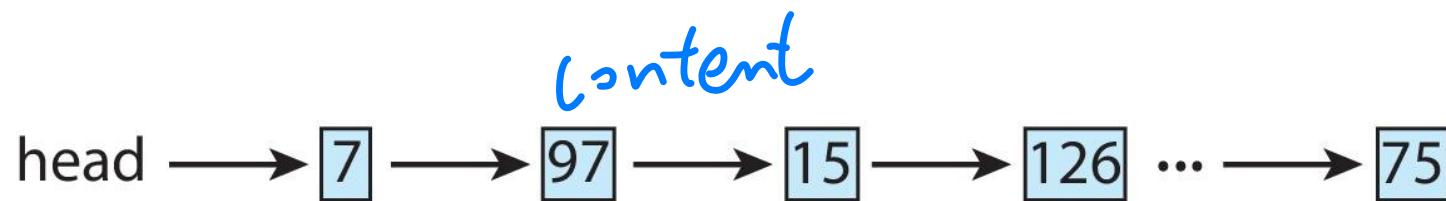
Free-Frame List

find available space in main memory

又見!

before bringing content!

- When a page fault occurs, the operating system must bring the desired page from the secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests (a kernel data structure that only OS can access)



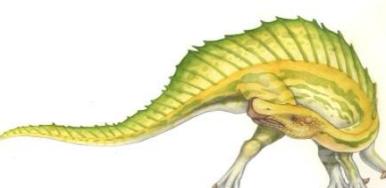
不會清 data
隨時 allocate
to new data!

fill 0 to all memory regions

clean!

- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames zeroed-out before being (re)-allocated.
 - The technique of writing zeros into a page before it is made available to a process - thus erasing their previous contents or to keep any old data from being available to the process
 - Consider the potential security implications of not clearing out the contents of a frame before reassigning it. (memory leak!)
- When a system starts up, all available memory is placed on the free-frame list.

demanding page → find one in this list!





Performance of Demand Paging

□ Steps in **Demand Paging** – handle page faults

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 (if available) in physical memory:
 1. Wait in a queue for this device until the read request is serviced
 2. Wait for the device seek and/or latency time *(DMA)* *(next chapter)*
 3. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU core to some other processes
7. Receive an interrupt from the disk I/O subsystem (I/O completed) *If DMA is finished*
8. Save the registers and process state for the other process (depending on the CPU scheduling)
9. Determine that the interrupt was from the disk
10. Update the page table and other tables to show 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

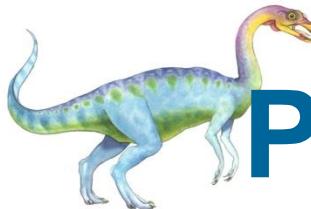
it's overhead !!.

how to avoid ?

DMA

If DMA is finished





Performance of Demand Paging (Cont.)

- very slow!
- There are three major tasks in page-fault service time:
 - ✓ Service the interrupt – careful coding might result in several hundred instructions
 - Read the page – lots of time (accessing the secondary storage, typically a hard disk)
involve secondary storage! Very slow!!! Degrade performance!
 - Restart the process – a small amount of time
- The page switch time will probably be close to 8 milliseconds (for a typical hard disk)
CPU:ns
Very large value in computer science!!
- Page Fault Rate $0 \leq p \leq 1$
 - if $p = 0$ no page faults
 - if $p = 1$, every reference is a fault

Interrupt!
] Extreme case! Goal: ↓ page switch time!
- Effective Access Time (EAT)
$$\text{EAT} = \frac{(1 - p) \times \text{memory access}}{+ p (\text{page fault overhead})} + \text{swap page out} + \text{swap page in} + \text{restart overhead}$$

page fault rate
write page to secondary storage
back victim
(more time!!!)
3 TLB: Always LRU!
traditional RA,
LRU page replacement才有!!!





Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
large number!!!
- EAT = $(1 - p) \times 200 + p (8 \text{ milliseconds})$
 $= (1 - p) \times 200 + p \times 8,000,000$
 $= 200 + p \times 7,999,800$

$$P = \frac{1}{1000}$$

- If one access out of 1,000 causes a page fault, then
EAT = 8.2 microseconds.

This is a slowdown by a factor of 40!! ≈ 41

- If want performance degradation < 10 percent
 - $220 > 200 + 7,999,800 \times p$
 $20 > 7,999,800 \times p$
 - $p < .0000025$
 - < one page fault in every 400,000 memory accesses
- Fortunately, the memory locality usually satisfies this, as each memory miss brings an entire page

same page 有多行 - 行 instruction!

2是100, TLB is not used \Rightarrow
*① Access memory for page table
② Translate address for real data
 \Rightarrow Involve 2 memory access!*

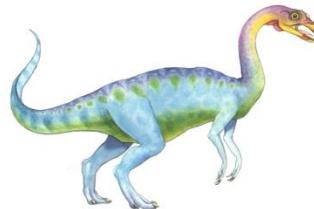
*Page fault:
Very high overhead*

好慢!!!

large performance degradation!!!

*Von Neumann Architecture
code 多数連續*





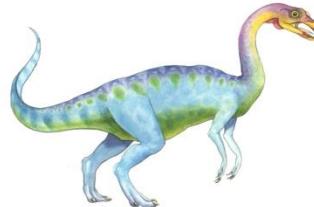
Copy-on-Write

Copy!

- Traditionally, `fork()` works by creating a **copy** of the parent's address space for the child, duplicating the pages belonging to the parent
- **Copy-on-Write** (COW) allows both parent and child processes to initially **share** the same **pages in memory** *how to implement COW?*
 - These shared pages are marked as **copy-on-write** pages, meaning that if either process writes to a shared page, a copy of the shared page must be created for that process. In that case, the non-modified page is left with the other process with no sharing
- **COW** allows more efficient process creation as only modified pages are copied or duplicated
- This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process *save memory!*
 - Considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary anyway

*↓
Waste!*





Before and After Process 1 Modifies Page C

for $\ell \in \mathbb{C}$

ℓ_{rec}^L)

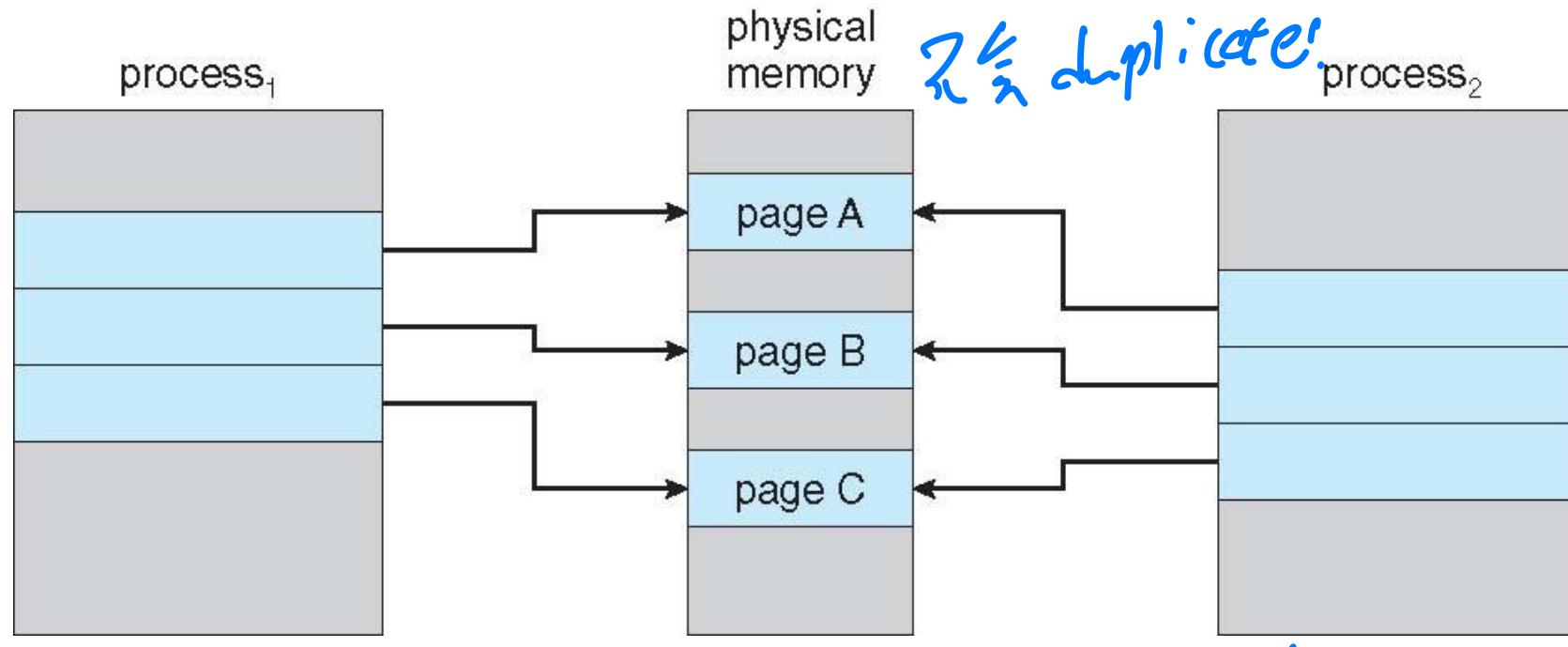
2-Fluor

沒有 (nw will)

be relieved
after new child
arrives!

cal exe
waste!!!

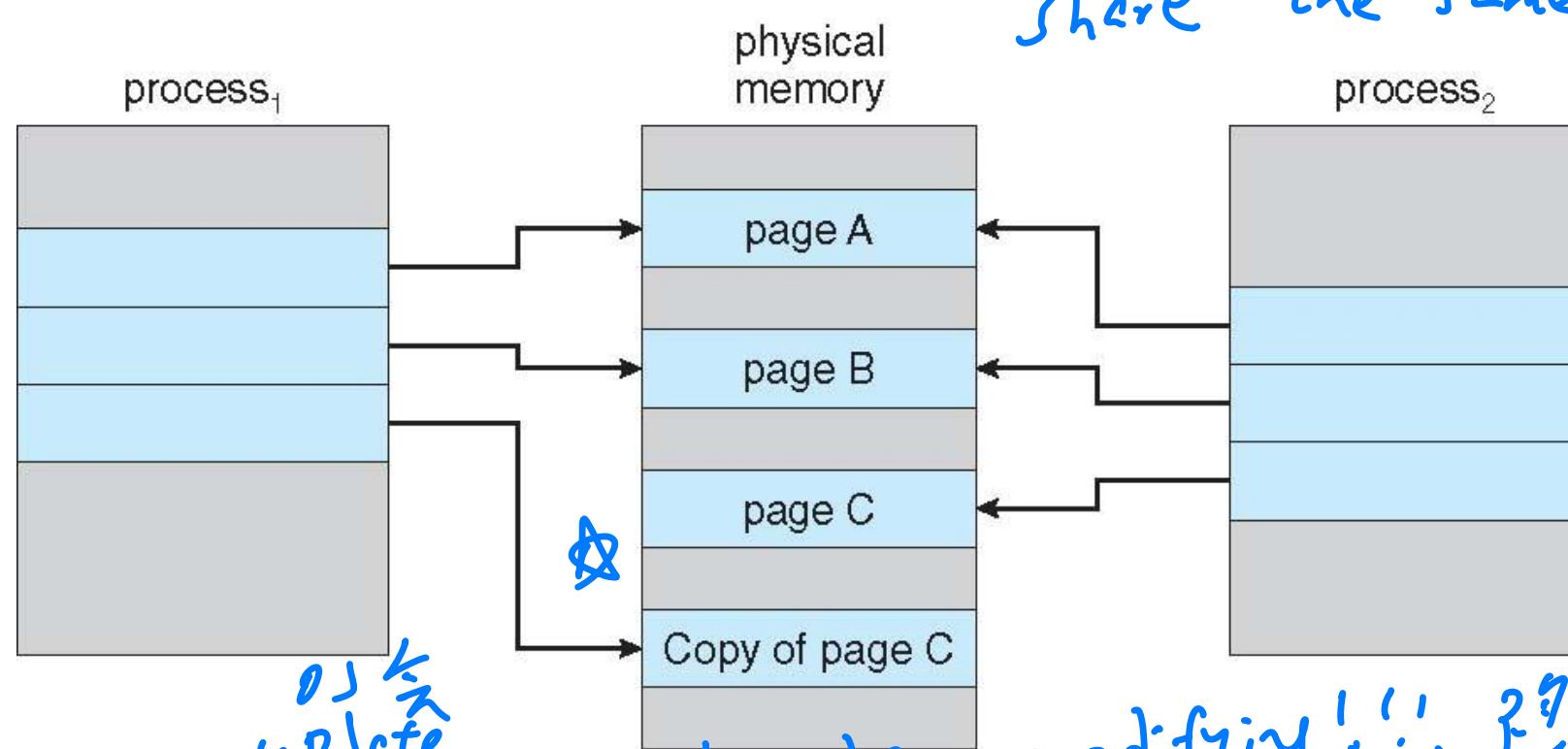
Linux we si
optimize de
performance!



No duplicate process

share the same

2 pages,
when change,
then os
will copy
the new page!



OBJX
update
copy page!
- 10th Edition

! only when

modifying ! ! !. Run 3-A update!





no frame, swap!

What Happens if There is no Free Frame?

Why?

lots of physical memory occupied!

Memory can save cycles!
important problem!

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc.
- How much memory to allocate to each process? - **frame-allocation** algorithm



脑子不转弯了! Bring it on!

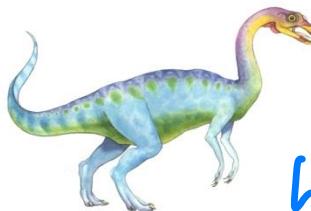
Given process,
how many frames should be allocated?

- Page replacement – find some page in memory, but not really in use, page it out
 - Algorithms – terminate the process? swap out the entire process image? replace the page? how to find?
 - Performance – want an algorithm which will result in the minimum number of page faults 竟然又用，bring again !!! :(Grr! went fault rate very slow, program performance much degrade!
 - This needs to be transparent to a process or program execution
should be done by OS!
- Noticing that it may be inevitable that same page may be brought into memory several times

X 100% page in page out, this page will not back in the memory again!

Make free-frame!





what involve?

problem

- Prevent over-allocation of memory by modifying page-fault service routine to include page replacement *program request free frame, but no!*
(Add bits of b.t!) cause program cannot continue running!
- Use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written back to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be supported on a smaller physical memory *much larger! 虽然不大*

processes working together
require pages to run!



whole memory system is
busy in page in page out



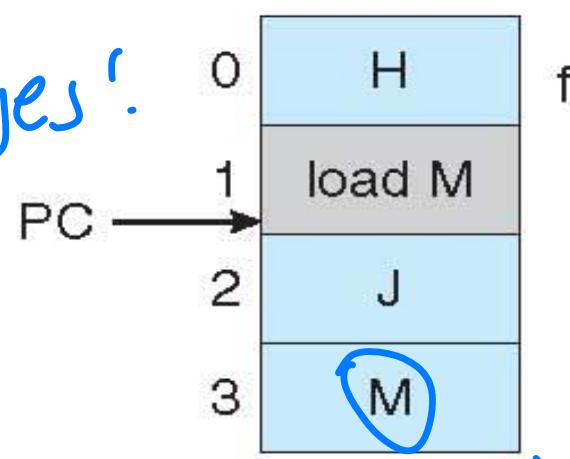
CPU cannot do other stuff!





Need For Page Replacement

each want
4 pages!



frame valid-invalid bit

| | |
|---|---|
| 3 | v |
| 4 | v |
| 5 | v |
| | i |

page table for user 1

not in main memory

trap
no free frame

choose memory!

| 0 | A |
| 1 | B |
| 2 | D |
| 3 | E |

logical memory for user 2

frame valid-invalid bit

| | |
|---|---|
| 6 | v |
| 2 | v |
| 7 | v |
| | i |

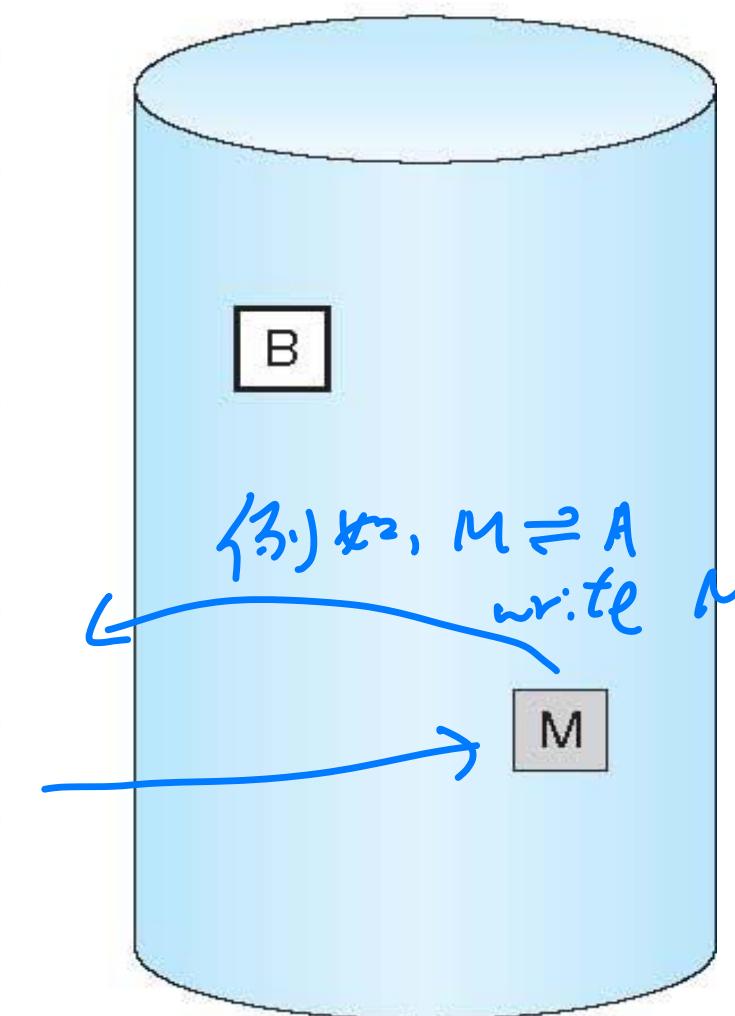
page table for user 2

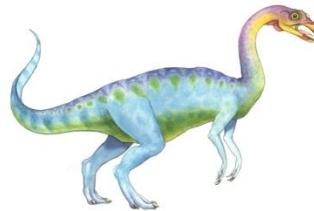
by others

full!

| 0 | monitor |
| 1 | |
| 2 | D |
| 3 | H |
| 4 | load M |
| 5 | J |
| 6 | A |
| 7 | E |

physical memory





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 if “dirty” (modified since last time it was brought into the memory)
Some time to write to disk!
3. Bring the desired page into the (newly) free frame; update the page table
4. Continue the process by restarting the instruction that caused the trap

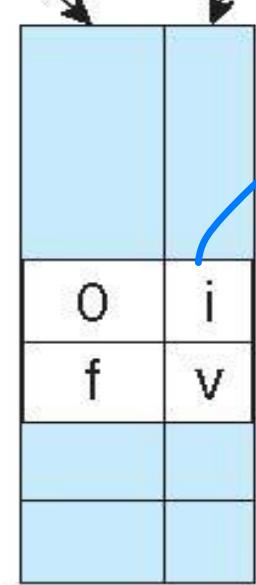
Note now potentially two page transfers for a page fault – which can significantly increase EAT





Page Replacement

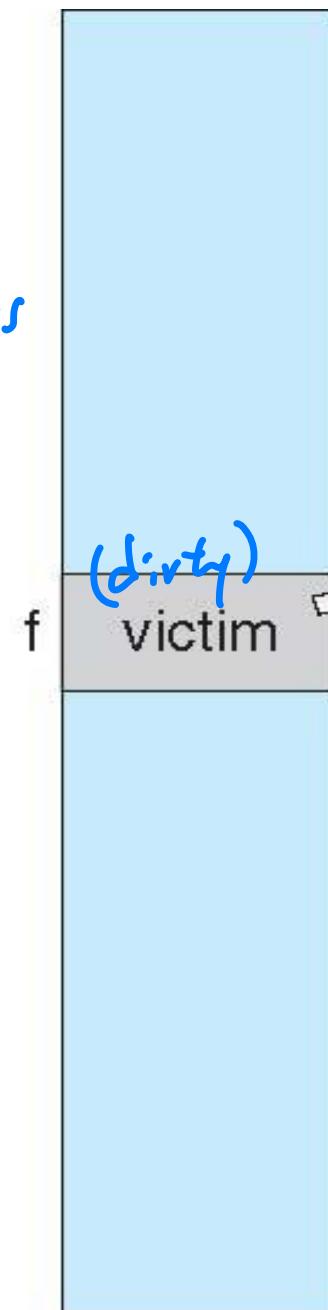
frame valid-invalid bit



page fault
interrupt to OS

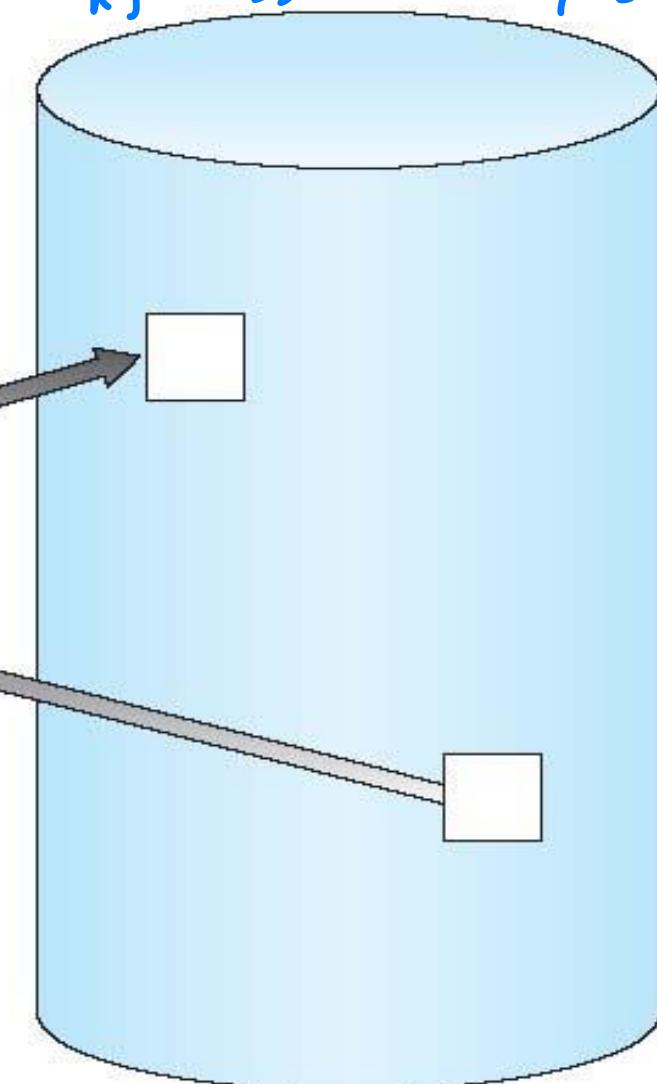
change to invalid

2
4
reset page table for new page



physical memory

Access secondary storage



two I/O



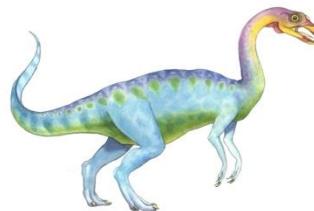


Page and Frame Replacement Algorithms

2 Directions

- Page-replacement algorithm
 - Decide which frames to replace – the objective is to minimize the page-fault rate
Decide who is unlucky guy
- Frame-allocation algorithm determines - how many frames allocated to each process – which is decided by how a process accesses the memory - locality





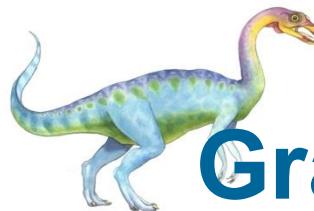
不用 physical address, very long!

Page and Frame Replacement Algorithms

- Evaluate algorithms by running on a particular string of memory references - **reference string** and computing the number of page faults on that string
 - First, for a given page size, we need to consider only the **page number**, rather than the complete physical memory address.
 - If we have a reference to a page **p**, then any references to page **p** that **immediately follow** will never cause a page fault. Page **p** will be in memory after the first reference, so the immediately following references will not have page fault.
- For example, for a particular process, we might record the following address sequence:
first page 0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
- At 100 bytes per page, this sequence is reduced to the following reference string:
*1, 4, 1, 6, 1, 6, 1, 6, 1
? bring to main memory!*
- In all our examples, the reference string is
7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1 *fixed string!*

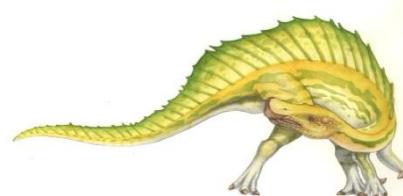
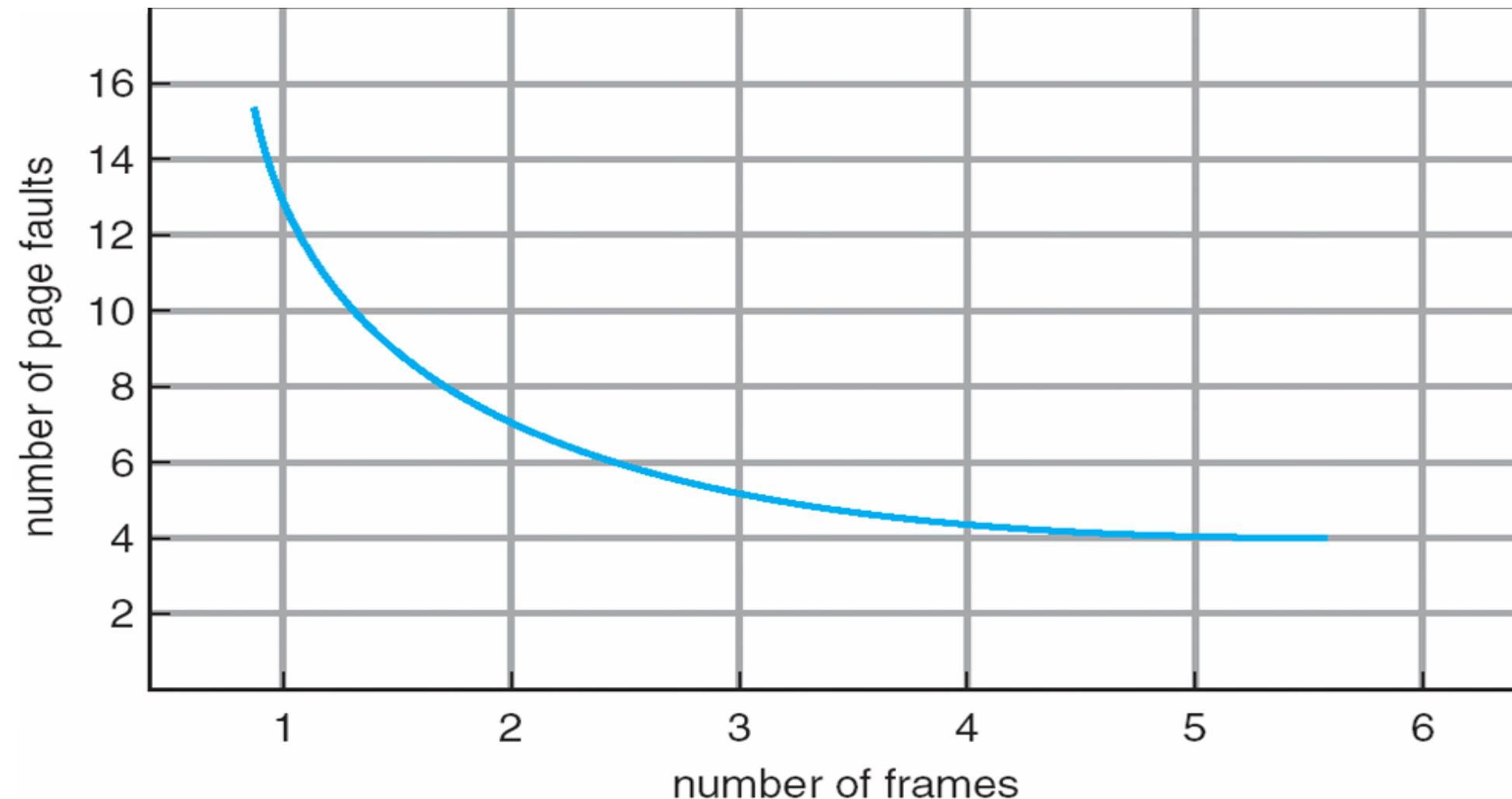
As few page as possible!

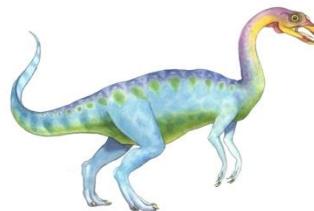




Graph of Page Faults Versus The Number of Frames

frames↑ \Rightarrow page faults↓

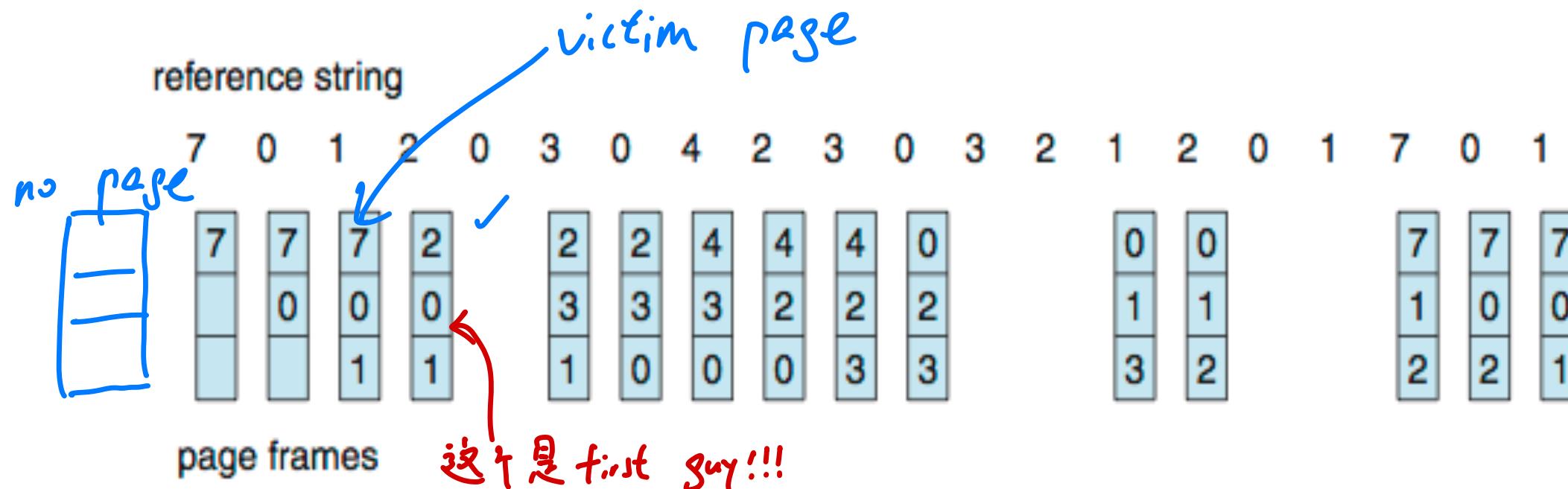




First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time for the process)

Similar to CPU scheduling



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** Even worse result!!!
- How to track ages of pages? – implementation easy
 - Just use a FIFO queue (the time this page brought into the memory)

Dqueue the structure is ok!

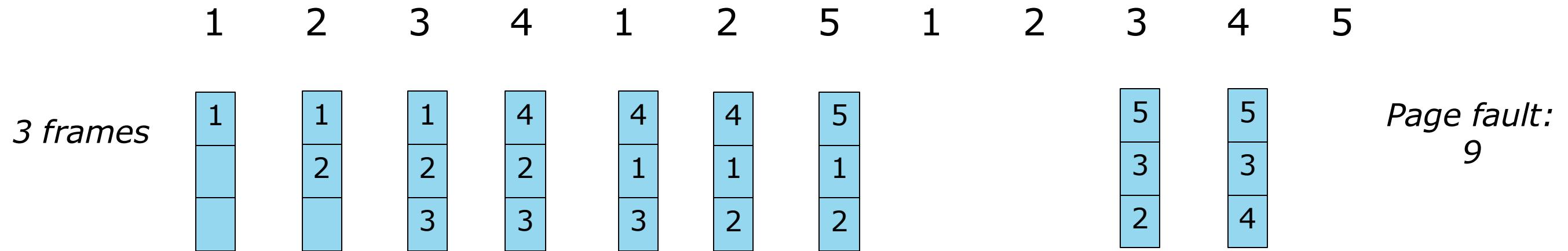
10.30



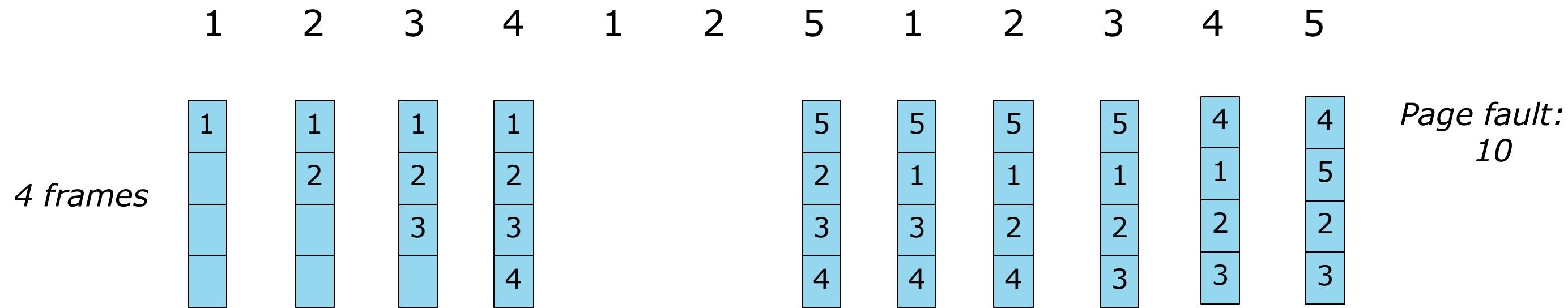


FIFO Page Replacement

reference string

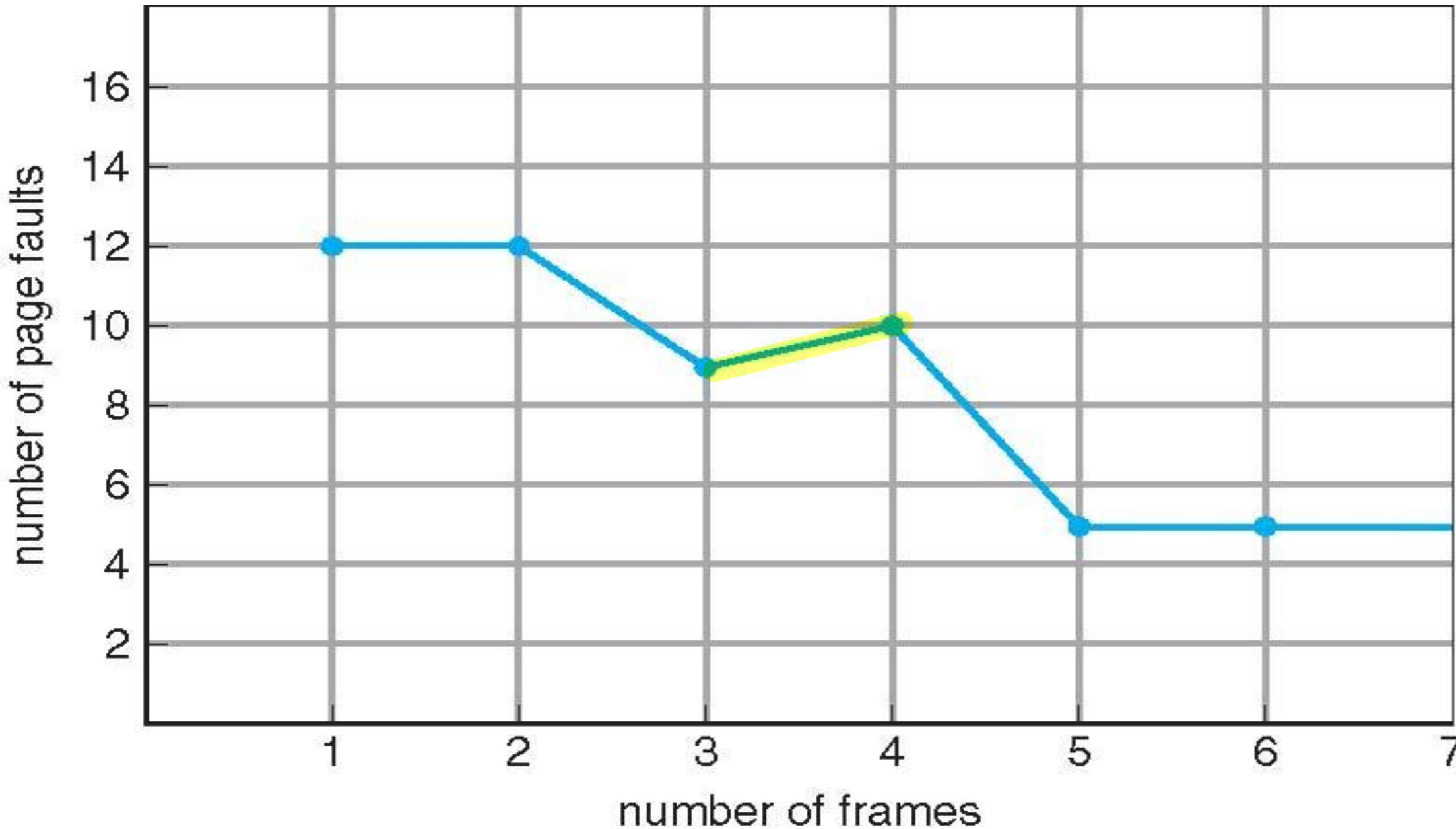


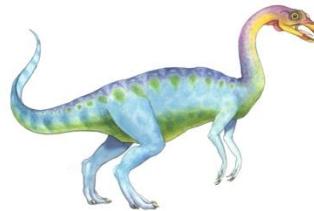
reference string





FIFO Illustrating Belady's Anomaly

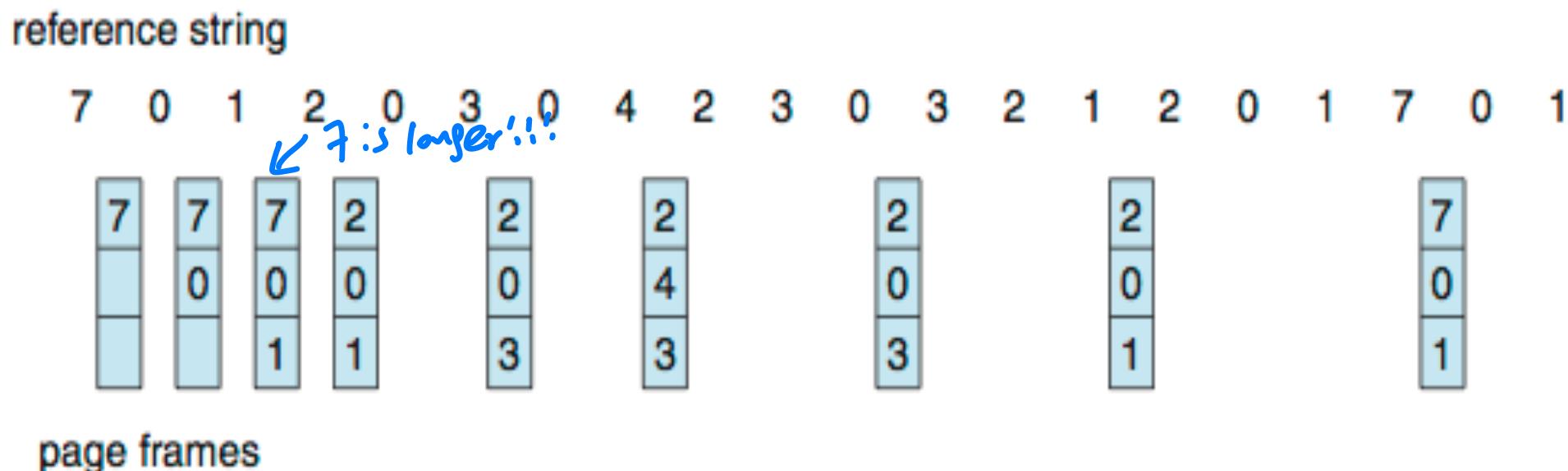


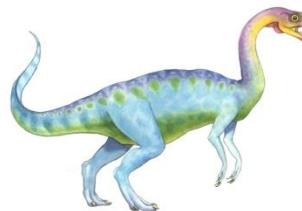


Optimal Algorithm

Difficult to implement in real world!

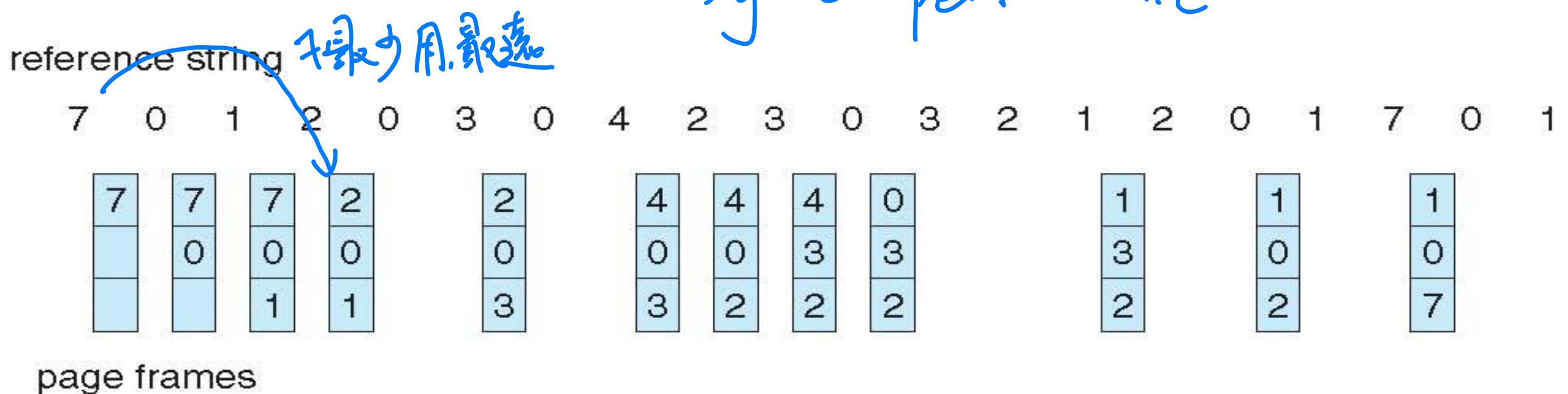
- Replace page that will not be used for **longest period of time** in the future
 - If possible, ideally select a page that will not be used at all in the future. In practice, this may not be feasible, so a page can be brought into the memory multiple times
 - 9 page fault is the optimal for the example
- How do you know this?
 - Can't read the future
- Used for measuring how well your algorithm performs as a comparison





Least Recently Used (LRU) Algorithm

- Use the past knowledge rather than future as an approximation for OPT PAST!
- Replace page that has not been used for the most amount of time
- Associate the time of last use with each page – complex, as this need to be updated with each memory reference



- 12 faults – better than FIFO but worse than OPT
- Generally a good algorithm and frequently used
- But how to implement? - An LRU page-replacement algorithm may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use.





LRU Algorithm Implementation

□ **Counters** implementation

- Every page entry adds a **time-of-use** field recording a logical clock or counter. The clock is incremented for every memory reference.
- Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page-table entry for that page.
- When a page needs to be replaced, look for the counters to find the smallest value
 - ▶ Search through table needed to find the LRU page and a write to memory (to the time-of-use field in the page table) for each memory access

Scan all pages! find least time used!

□ **Stack** implementation

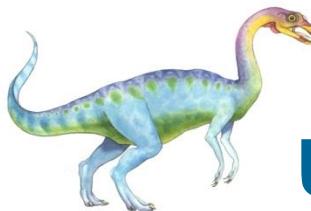
- Keep a stack of page numbers
- Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is **always at the top** of the stack, and the least recently used page is **always at the bottom**.
- This requires multiple pointers to be changed upon each reference
- Each update more expensive, but no need to search for replacement

pop → find most recent page!

tradeoffs

lots of software overhead
as lots of interrupts involved!

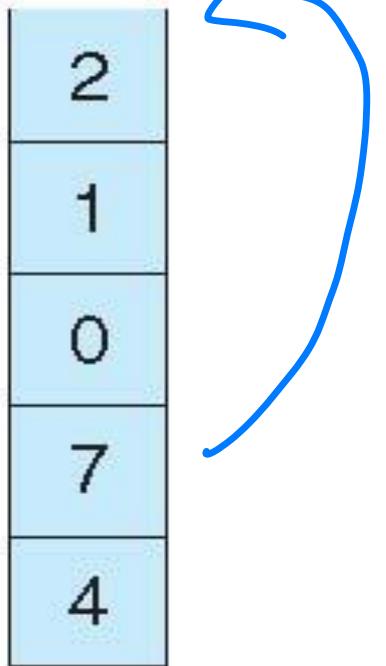




Use of A Stack to Record the Most Recent Page References

reference string

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



stack
before
a



stack
after
b

Remove 7,
place to
the top!

Pick the bottom to replace!

frame ↑ => always better performance!





LRU Algorithm Discussions

- LRU and OPT are cases of **stack algorithms** that don't suffer from Belady's Anomaly
 - A **stack algorithm** can be shown that the set of pages in memory for **n** frames is always a subset of the set of pages that would be in memory with **n+1** frames allocated
Stack algorithm!
 - For LRU replacement, the set of pages in memory would be the **n** most recently referenced pages. If the number of frames allocated is increased to **(n+1)**, these same **n** pages will still be part of the **n+1** most recently referenced, so will still be in memory
- Both the implementation of LRU A(counter and stack) requires extra hardware assistance
 - The updating of the clock fields or stack must be done for **every memory reference**
 - If we were to use an interrupt for every reference to allow software to update such data structures, it would slow every memory reference by a factor of at least ten!

table *change pointers* *MMU, special unit for LRU*

非常慢!!!





LRU Approximation Algorithms

□ Reference bit

- Each page associate a bit, initially = 0, associated with each entry in the page table
- When page is referenced (read or write), the reference bit set to 1
- Replace any with reference bit = 0 (if one exists)
 - ▶ Coarse-grained approximation, but do not know the order of use

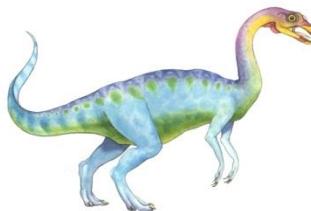
3. 没过 header:

□ Second-chance algorithm

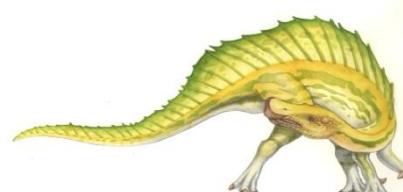
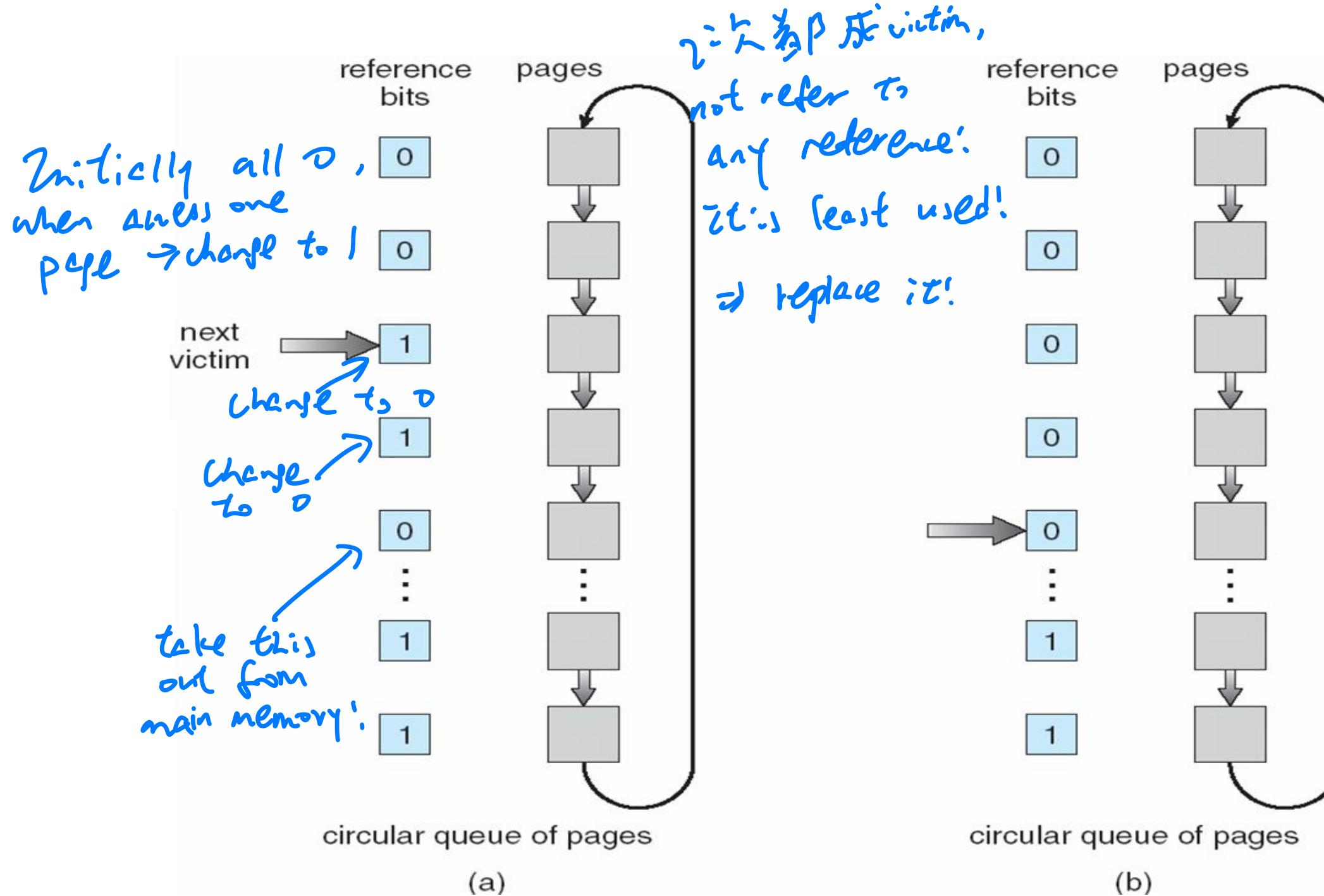
- FIFO order, plus hardware-provided reference bit - clock replacement
- If page to be replaced has 3. 没被 referencing:
 - ▶ Reference bit = 0 -> replace it (select it as the victim)
 - ▶ reference bit = 1 then:
 - set reference bit 0, leave page in memory (second chance)
 - replace next page, subject to same rules (FIFO and clock)

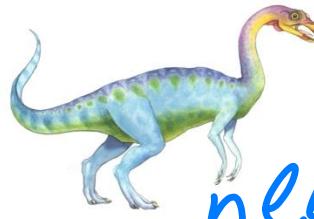
problem:
lots of 0,
cannot decide





Second-Chance (Clock) Page-Replacement Algorithm





need deep learn!

Counting Algorithms

not good
not efficient!

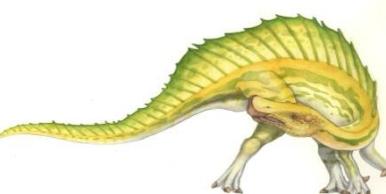
- Keep a counter of the number of references that have been made to each page
- The least frequently used (LFU) Algorithm: replaces the page with the smallest count
- The most frequently used (MFU) Algorithm: replace the page with the largest count based on the argument that the page with the smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU replacement is commonly used. The implementation of such algorithms is expensive, and they do not approximate OPT replacement well

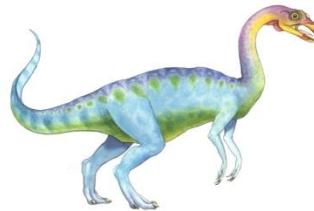
→ one counter one page! page replacement

counter, very expensive!!!

1. how to determine the victim?

2. how many pages that we need to allocate?



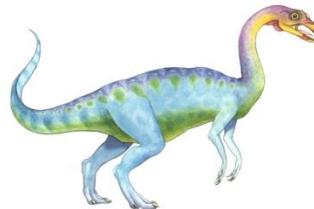


Allocation of Frames

- How do we allocate memory among different processes?
 - Does every process get the same fraction of memory, or different fractions?
 - Should we completely swap some processes out of memory?

Should we swap off the pages? minimum number of frames!
- Each process needs certain minimum number of frames in order to execute its program
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- The maximum, of course, is the total frames required for a process
- Two major allocation schemes
 - ① □ fixed allocation
 - ② □ priority allocation
- Many variations





Fixed Allocation

Easy!
Static Allocation,
the number is fixed

每一样!!!

- **Equal allocation** – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames

Just do division!

$$\frac{100}{5} = 20$$

- **Proportional allocation** – Allocate according to the size of process

- Dynamic as the degree of multiprogramming and process sizes change over the time

s_i = size of process p_i

$S = \sum s_i$ (total size)

m = total number of frames

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

$$m = 64$$

$$s_1 = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Determined in advance!





Dynamic allocation! Number can be changed!

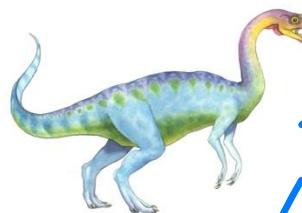
Global vs. Local Allocation

- No free frames!
- **Global replacement** – process selects a replacement frame from the set of all frames, even if that frame is currently allocated to some other process; thus, one process can take a frame from another process
 - For instance, this can be based on priority – priority allocation in which a process can preempt memory from a lower priority process → suffer from performance degradation, fault rate
 - This can result in better system throughput (不用低优先级进程的帧) ⇒ good utilization!
 - But process execution time can vary greatly, as a process cannot control its own page-fault rate. (Unfairness!). :(
 - some processes suffer from low performance
 - ↓ fault rate!
 - processes can't occupy frame (\uparrow priority)
 - **Local replacement** – each process selects from only its own set of allocated frames
 - What we used in the page replacement algorithms earlier
 - The set of pages for a process is only affected by the paging behaviour of that process
 - More consistent per-process performance
 - But possibly underutilized memory, since pages allocated to a process can not be utilized by another process, even if this page is not currently used by the process holding it

Drawbacks

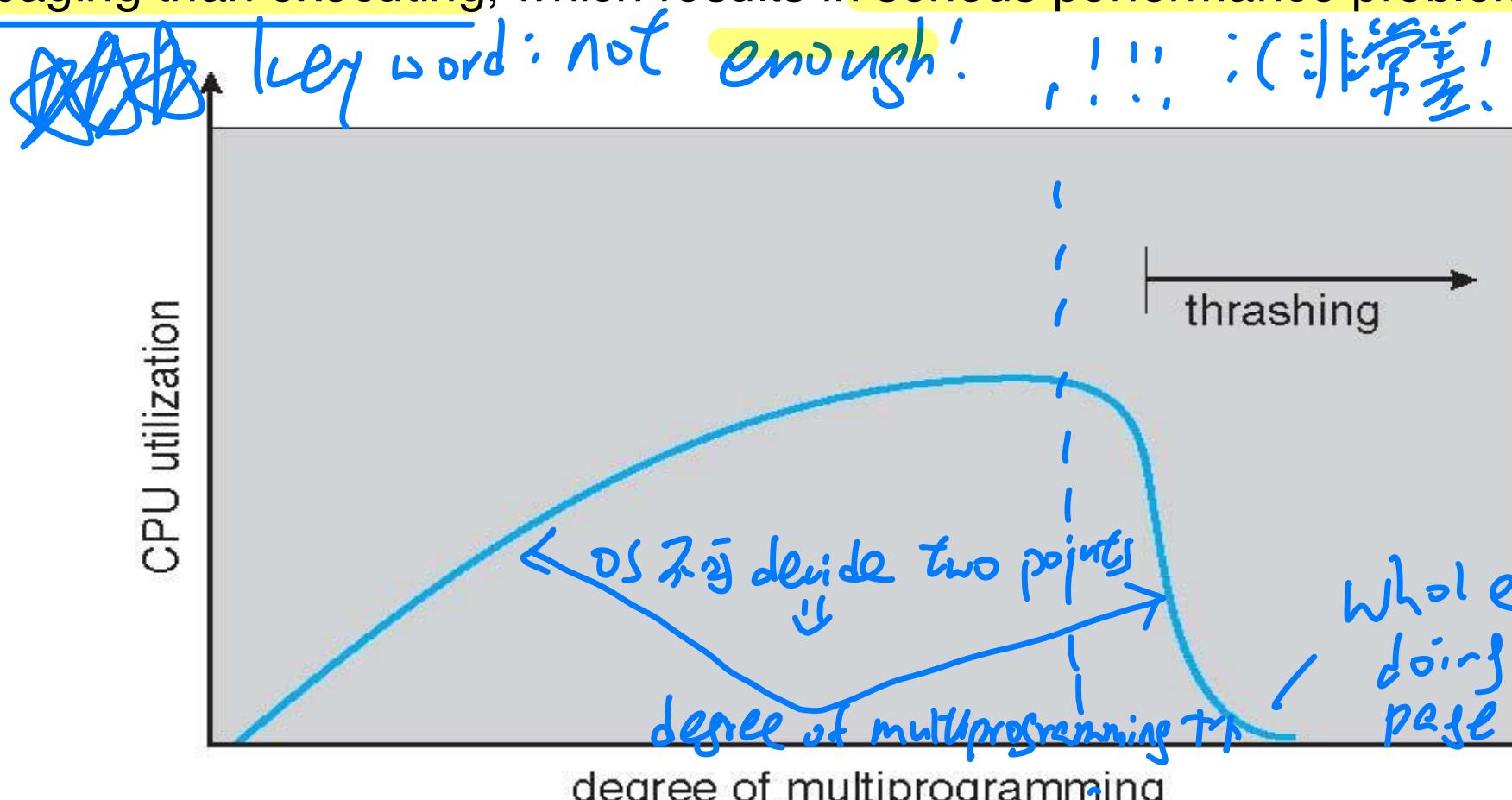
memory wastage!





Extreme case? Why this mean? Quite difficult! Thrashing

- If a process does not have "enough" pages, the page-fault rate would be very high
 - Page fault to get page, and replace an existing frame, but quickly need replaced frame back
 - This leads to low CPU utilization – OS might think "by mistakes" that it needs to increase the degree of multiprogramming in order improve the CPU utilization – aggravate the problem
- Thrashing ≡ a process or a set of processes is busy swapping pages in and out
- This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing, which results in serious performance problems



Thrashing:
determine what
is enough!





Demand Paging and Thrashing

- Why does demand paging work?

Locality model

Assumption of memory access!

Memory access have some locality pattern

- A locality is a set of pages that are actively used together. A running program is generally composed of several different localities over the time, which may overlap
- Memory access or subsequent memory access tends to stay in the same set of pages
- Process migrates from one locality to another, e.g., operating on a different set of data or call a function (different code segment)
- Localities may overlap, instructions or part of the data being manipulated

function A,
memory access

in the same page!

too big

2 同 process 2 同 locality!

- Why does thrashing occur?

Σ size of locality (of all processes) > total memory size

total frames

many frame \Rightarrow thresh

- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

All

- We can limit the effects by using local replacement, this way thrashing in one process can not steal frames from another process and cause the latter to thrash as well

the locality 太大

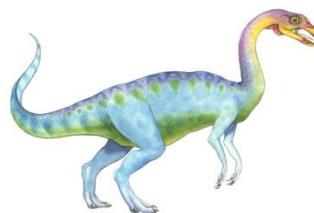
selection!

[local] : steal others, make the system \downarrow crashes :

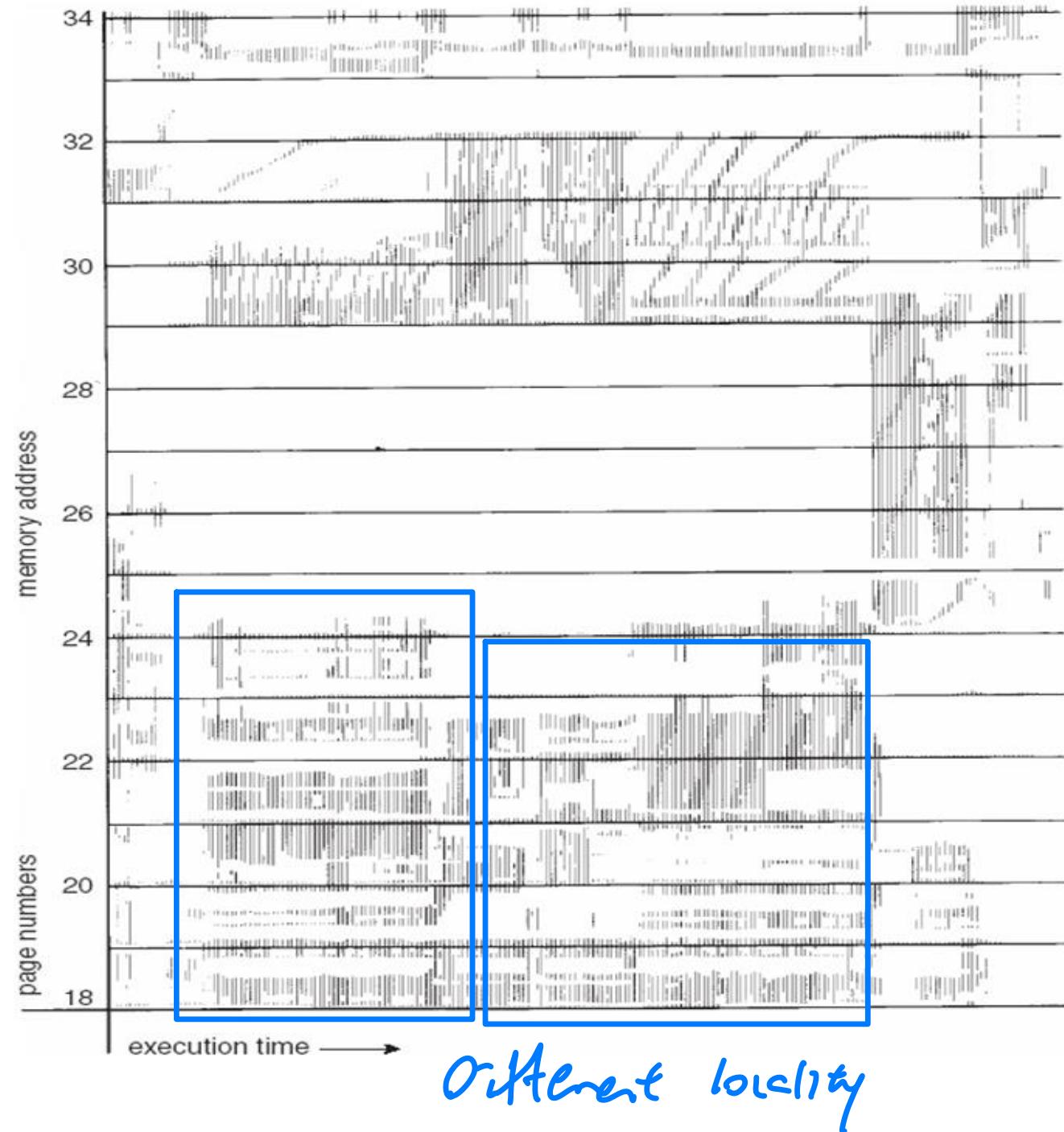
not enough
frame for
localities!

All process will steal frames!

All will have not enough frames :::



Locality In A Memory-Reference Pattern



- Recall program memory access patterns exhibit temporal and spatial locality
- The left Figure illustrates the concept of locality and how a process's locality changes over time. At time (a), the locality is the set of pages {18, 19, 20, 21, 22, 23, 24, 29, 30, 33}. At time (b), the locality changes to {18, 19, 20, 24, 25, 26, 27, 28, 29, 31, 32, 33}. Notice the overlap, as some pages (for example, 18, 19, and 20) are part of both localities.





How we can track locality in the process?

Working-Set Model

- $\Delta \equiv \text{working-set window} \equiv \text{a fixed number of page references}$

Example: 10,000 instructions

- WSS_i (working set of Process P_i) =

total number of distinctive pages referenced in the most recent Δ - this 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

working-set window

- capture too many localities.

need set good Δ

- $D = \sum WSS_i \equiv \text{total demand frames}$

- Approximation of the current locality in the system (of all processes)

Total memory:

- if $D > m \Rightarrow$ Thrashing - at least one process is short of memory

- Policy if $D > m$, then suspend or swap out one of the processes

Principle

⇒ decrease degree of multiprogramming:

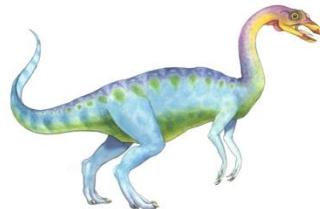
- The working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible, thus optimizes CPU utilization

- The difficulty with a **working-set model** is how to keep track of the working set.

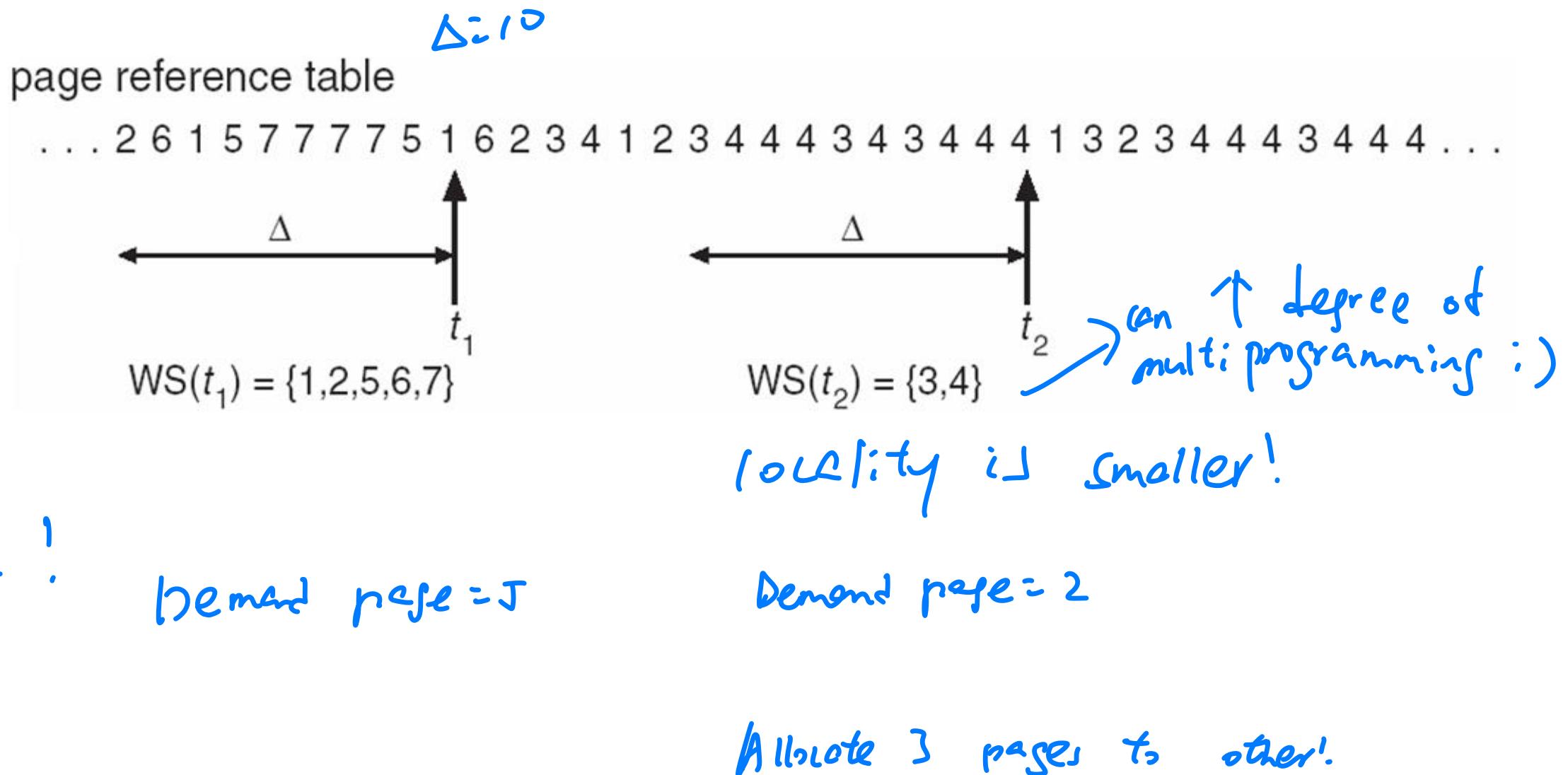
should have high complexity.
use simplified sampling algorithm

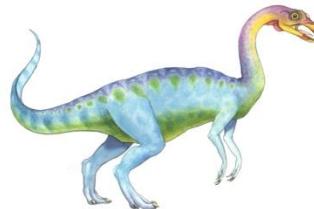
check every memory access!





Working-Set Model (Cont.)





Keeping Track of the Working Set

- It is difficult to keep track of the working set, as working-set window is a **moving window** which needs to be updated **for each memory reference** *high complexity, huge overhead! :(*
- Approximate with interval timer + a reference bit
Sampling
for every timer ✓
Timer only happens once in one Δ !
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page, (0,0), (0,1),(1,0),(1,1)
 - Whenever a timer interrupt **copy and sets the values** of all reference bits to 0
 - If one of the bits in memory = 1 \Rightarrow page in working set
- This is of course not accurate, as we cannot tell where, within an interval of 5,000, a reference occurred
fewer samples, lower overhead! ;), more inaccurate :(trade off!
- Improvement = 10 bits and interrupt every 1000 time units, more accurate but cost is higher
- Accuracy versus complexity

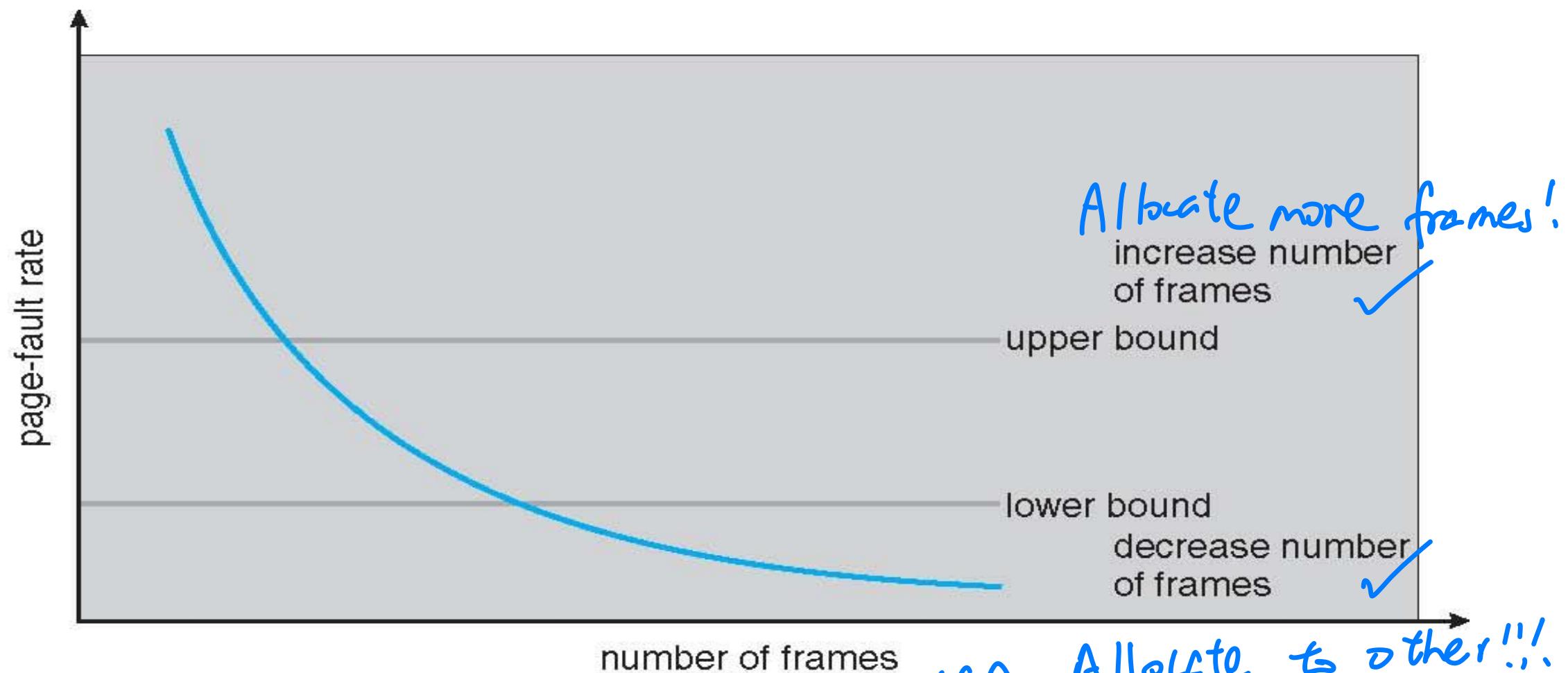




[intro | thrashing!]

Page-Fault Frequency

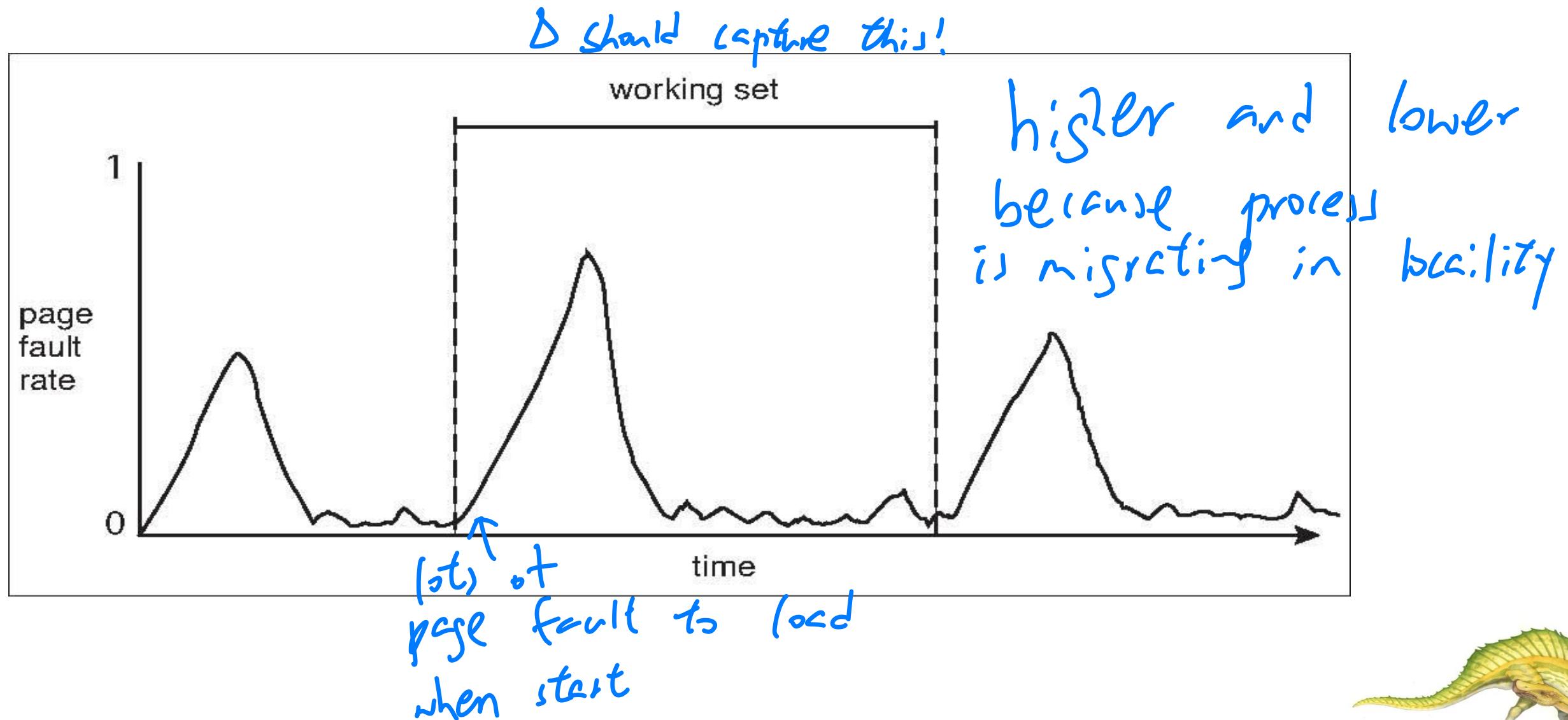
- More direct approach than WSS
- Establish “acceptable” page-fault frequency (PFF) rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame

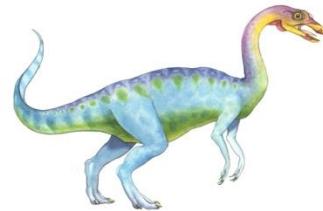




Working Sets and Page Fault Rates

- Relationship between working set of a process and its page-fault rate
- Working set changes over time
- The page-fault rate of a process transitions between peaks and valleys over time.





Other Considerations

- ① Prepaging
- ② Page size
- ③ TLB reach
- ④ Program structure

introduce in brief is okay.





Prepaging

how many pages to bring to memory when the process starts

- To reduce the large number of page faults that occurs at process startup
- Prepage some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used

Compare whether benefit is larger than overhead? final

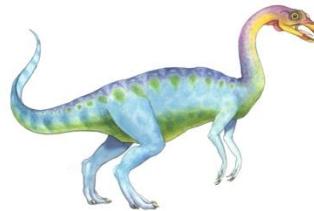
- Is cost of $s * \alpha$ save pages faults > or < than the cost of prepaging $s * (1 - \alpha)$ unnecessary pages?
- α near zero \Rightarrow prepaginaing loses

$\alpha \approx 1 \Rightarrow$ prepaginaing wins

$\alpha = 1 : \text{guess is } 100\% \text{ correct :)}$

$\alpha = 0 : \text{unlucky :(}$





Page Size

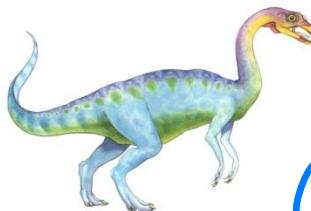
- Sometimes OS designers have a choice
 - Especially if running on custom-built CPU
- Page size selection must take into consideration a conflicting set of criteria:
 - Fragmentation – calls for smaller page size
 - Page table size – calls for larger page size
 - Resolution – isolate the memory actually be used
 - I/O overhead – larger page size requires longer I/O time
 - Number of page faults – smaller page size can increase the number of page faults
 - Locality – ideally each page should match the current locality
 - TLB size and effectiveness – larger page size improves the TLB reach
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

How to determine
page size?

small page size \Rightarrow table is small!

Read textbooks!.





Memory reach TLB!

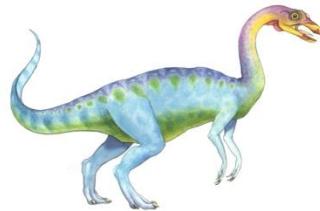
TLB Reach

→ bits of entry →
each entry points some memory!

- TLB Reach - The amount of memory accessible from the TLB
- $\text{TLB Reach} = (\text{TLB Size}) \times (\text{Page Size})$
- Ideally, the working set of each process is stored in the TLB
 - Otherwise, there might be a high degree of page faults, or the access time slows down
- Increase the Page Size ✓
 - This may lead to an increase in fragmentation as not all applications require a large page size

OS 有不同 versions : 4KB, 16KB
- Provide Multiple Page Sizes
 - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





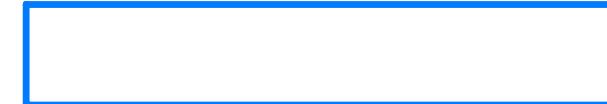
Program Structure

□ 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;
```

page →



$128 \times 128 = 16,384$ page faults

□ Program 2

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

fewer fault!

*Compiler change code!
Compiler will help you*

128 page faults



Quite Difficult :c

End of Chapter 10

