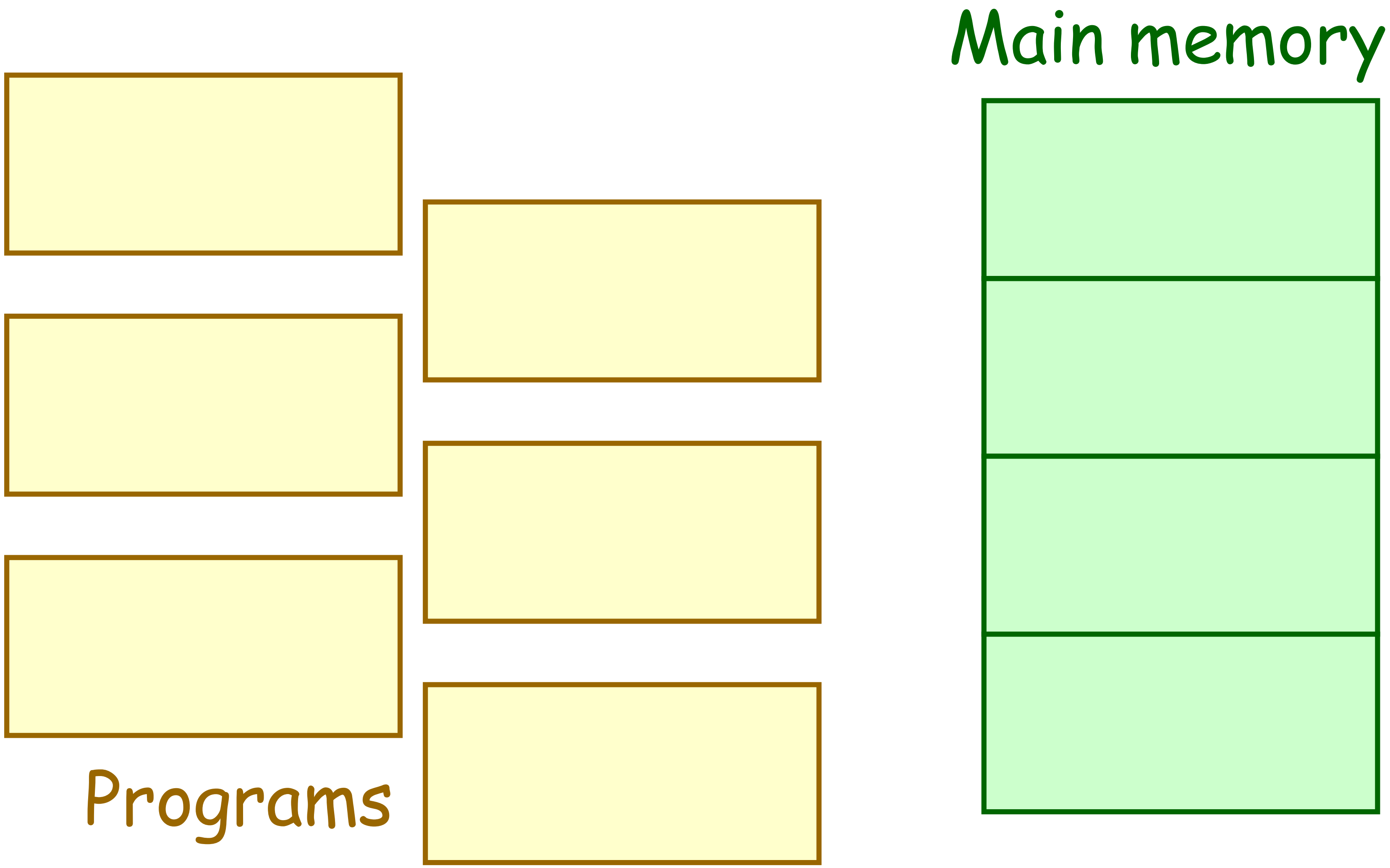
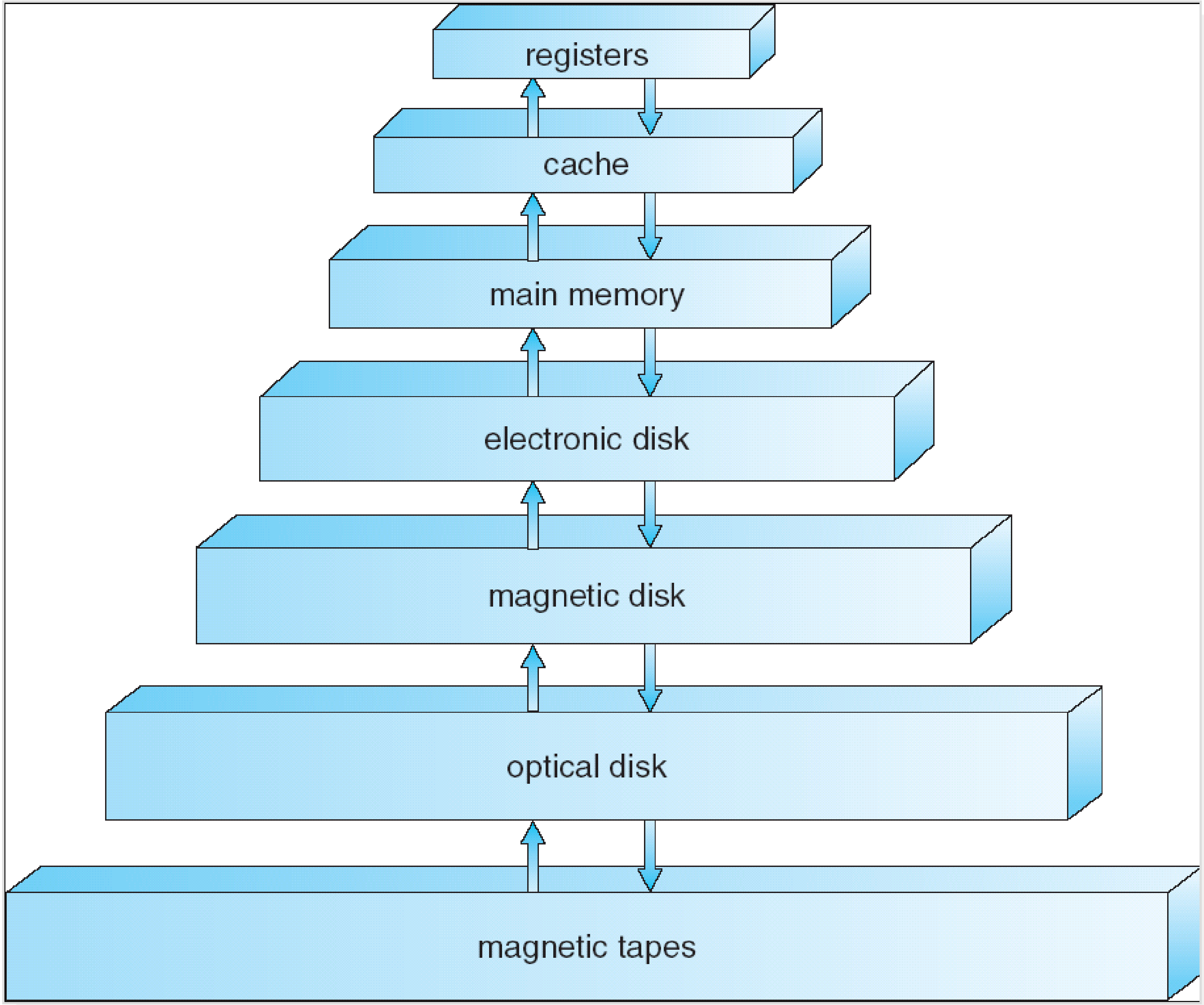
# Problem

- Memory is small, for I can not afford a bigger one.
- However, I have so many big programs to run at same time.

# Problem



# Storage-Device Hierarchy

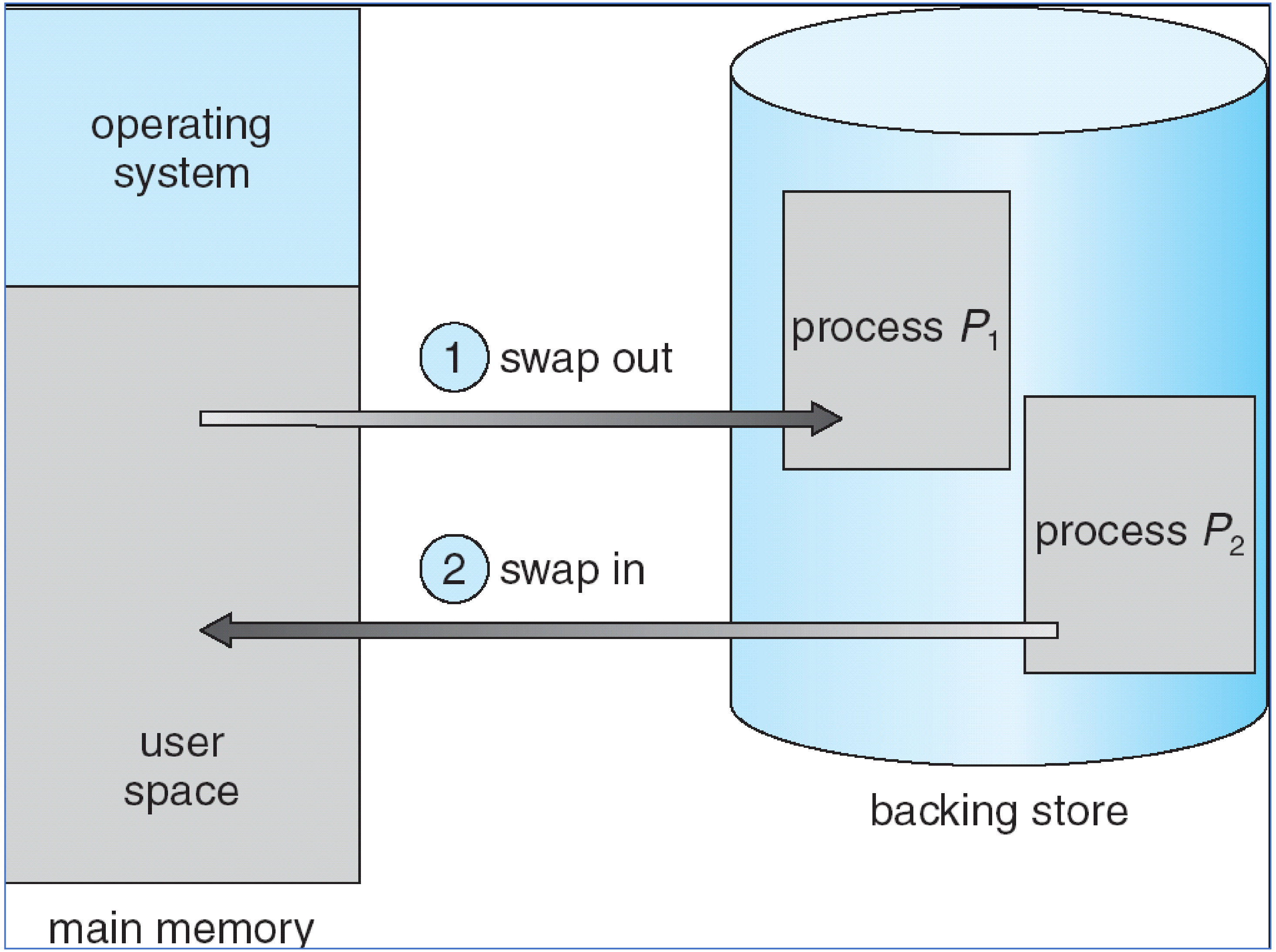


# Swapping

# Swapping

- A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

# Schematic View of Swapping



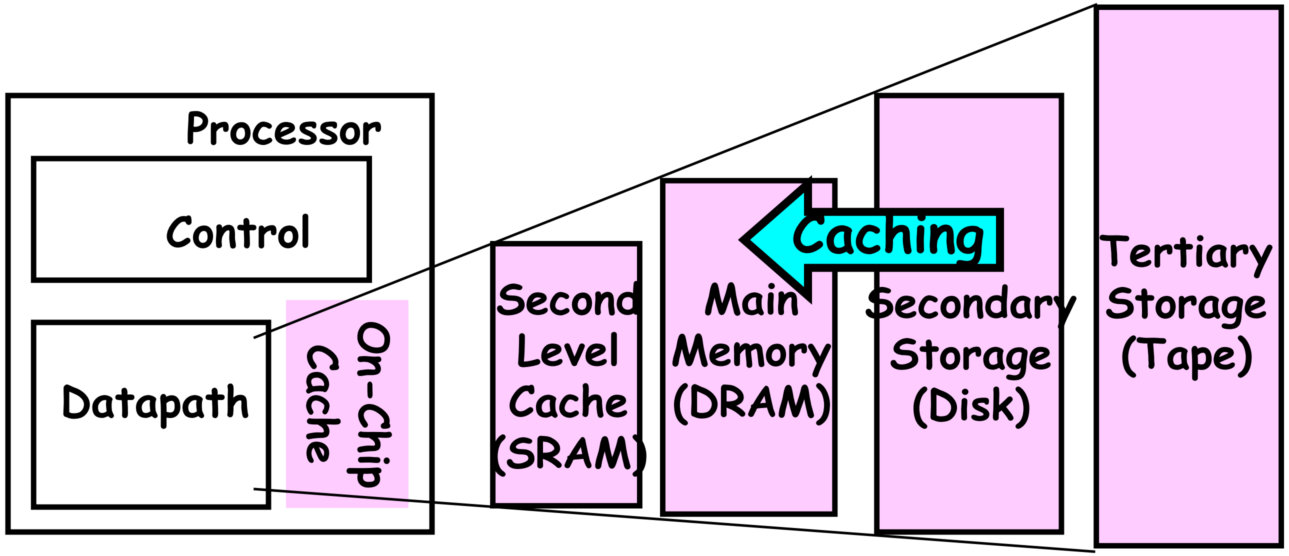
# Swapping

- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Extreme form of Context Switch: Swapping
  - In order to make room for next process, some or all of the previous process is moved to disk
  - This greatly increases the cost of context-switching
- Desirable alternative?
  - Some way to keep only active portions of a process in memory at any one time
  - Need finer granularity control over physical memory

# Locality!

# Solution

- Modern programs require a lot of physical memory
  - Memory per system growing faster than 25%-30%/year
- But they don’t use all their memory all of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user’s code to be in memory
- Solution: use main memory as cache for disk



# Caching Concept

- Cache: a repository for copies that can be accessed more quickly than the original
  - Make frequent case fast and infrequent case less dominant
- Caching underlies many of the techniques that are used today to make computers fast
- Only good if:
  - Frequent case frequent enough and
  - Infrequent case not too expensive
- Important measure: Average Access time = (Hit Rate x Hit Time) + (Miss Rate x Miss Time)

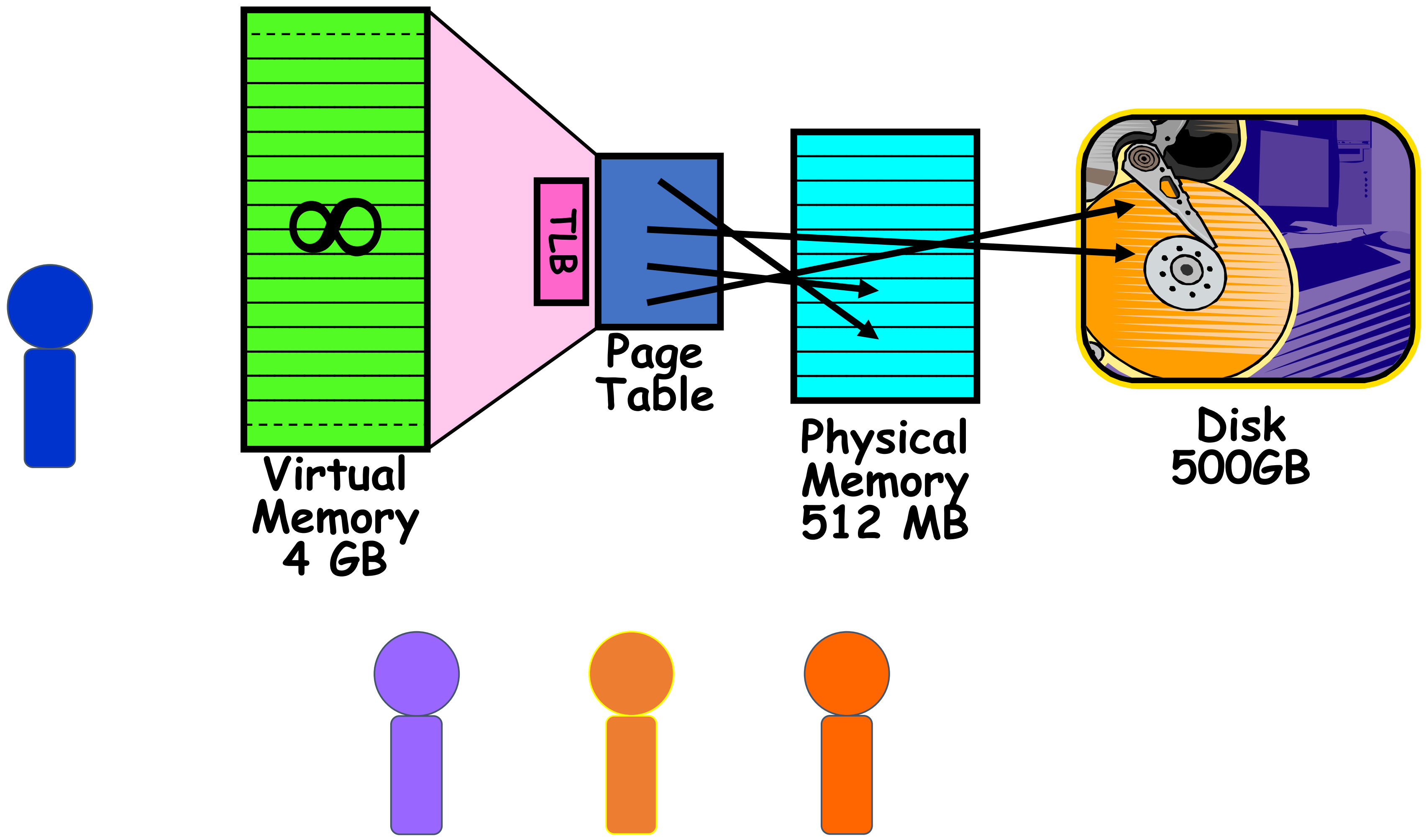
# Virtual Memory

# Background

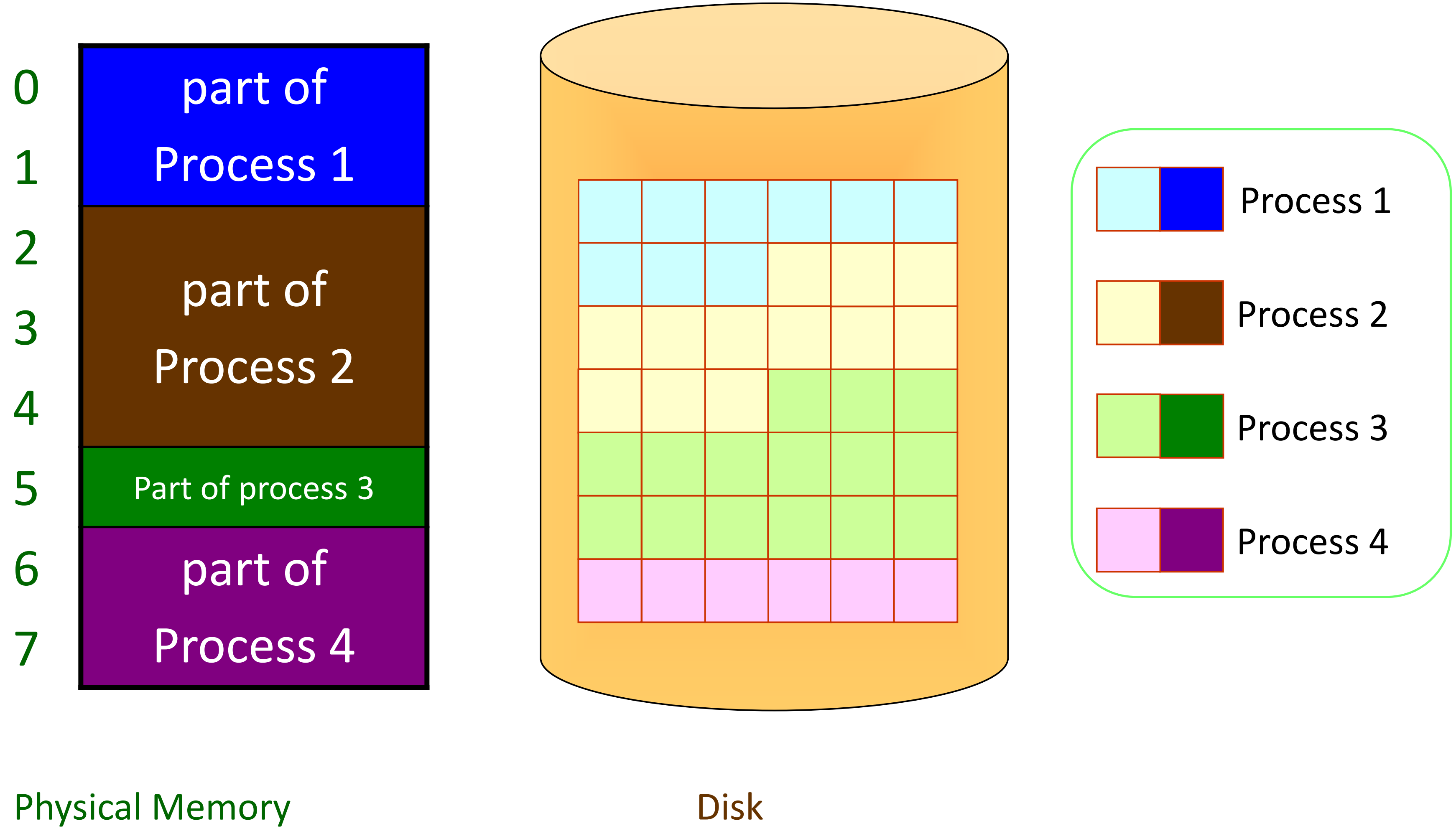
# Background

- **Virtual memory** – separation of user logical memory from physical memory.
- Virtual memory is a technique
  - Allows the execution of processes that may not completely in memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes

# Illusion of Infinite Memory



# Virtual memory



# Illusion of Infinite Memory

- Disk is larger than physical memory $\Rightarrow$
  - In-use virtual memory can be bigger than physical memory
  - Combined memory of running processes much larger than physical memory
    - More programs fit into memory, allowing more concurrency

# Benefits: (both system and user)

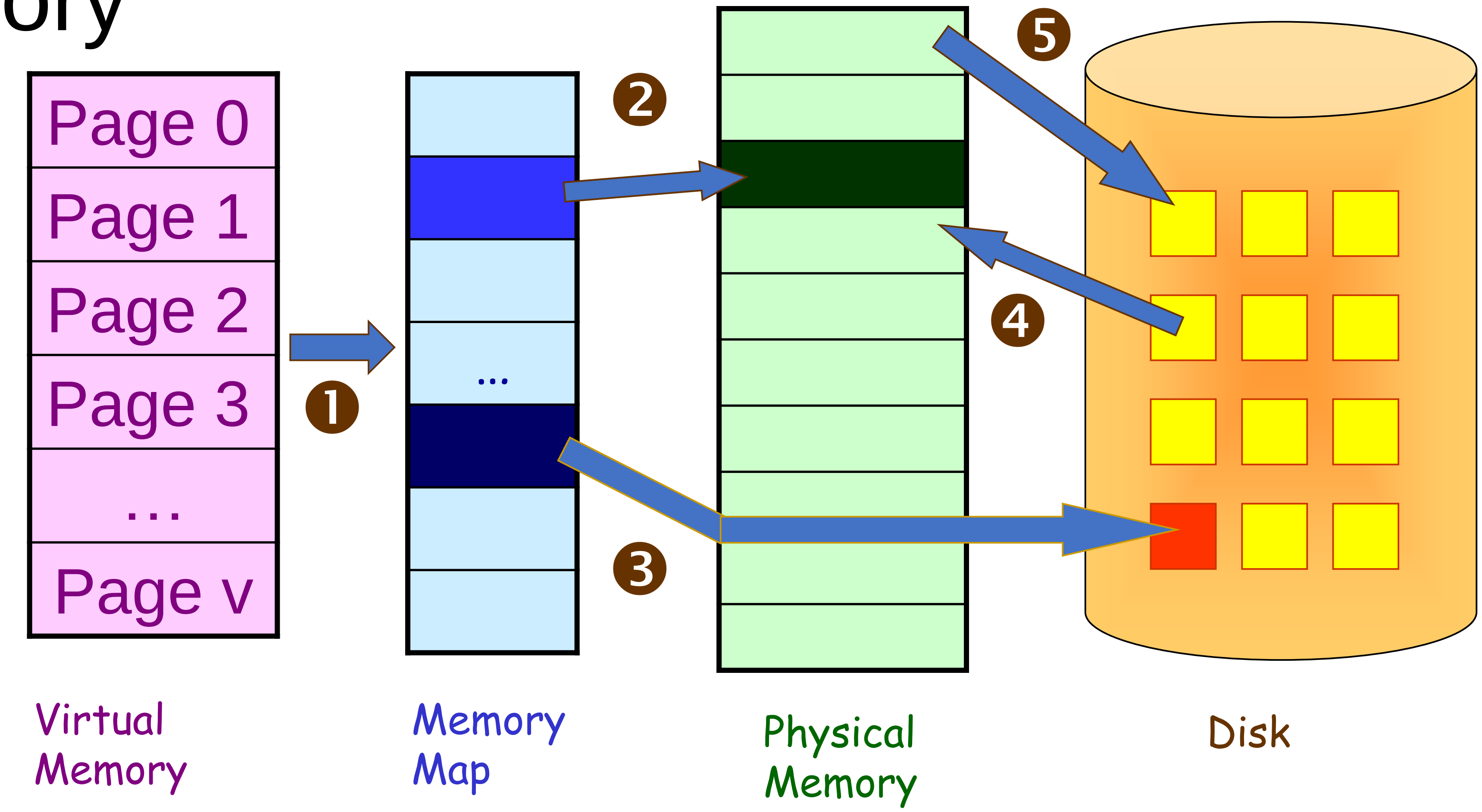
- Run a extremely large process
- Raise the degree of multiprogramming degree and thus increase CPU utilization
- Simplify programming tasks
  - Free programmer from concerning over memory limitation
- Programs run faster (less I/O would be needed to load or swap)

# Background

- Virtual memory is commonly implemented via:
  - Demand paging
  - Demand segmentation

# Virtual Memory

- Virtual Memory: That is Larger Than Physical Memory



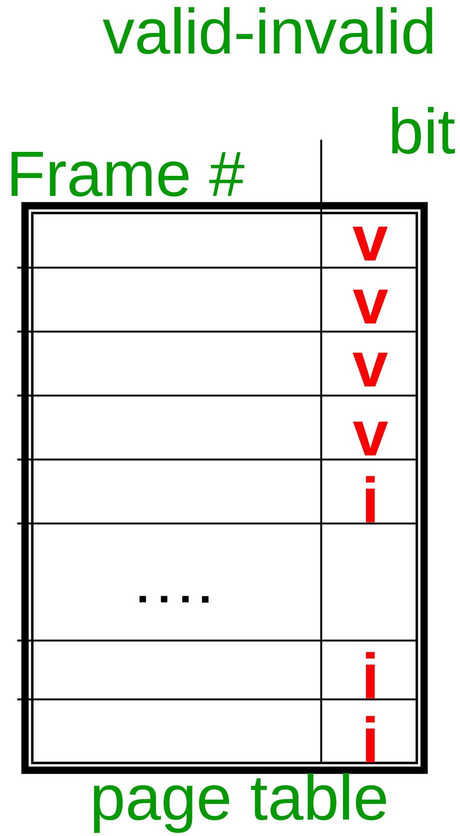
# Demand Paging

# Demand Paging

- Bring a page into memory only when it is needed
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**

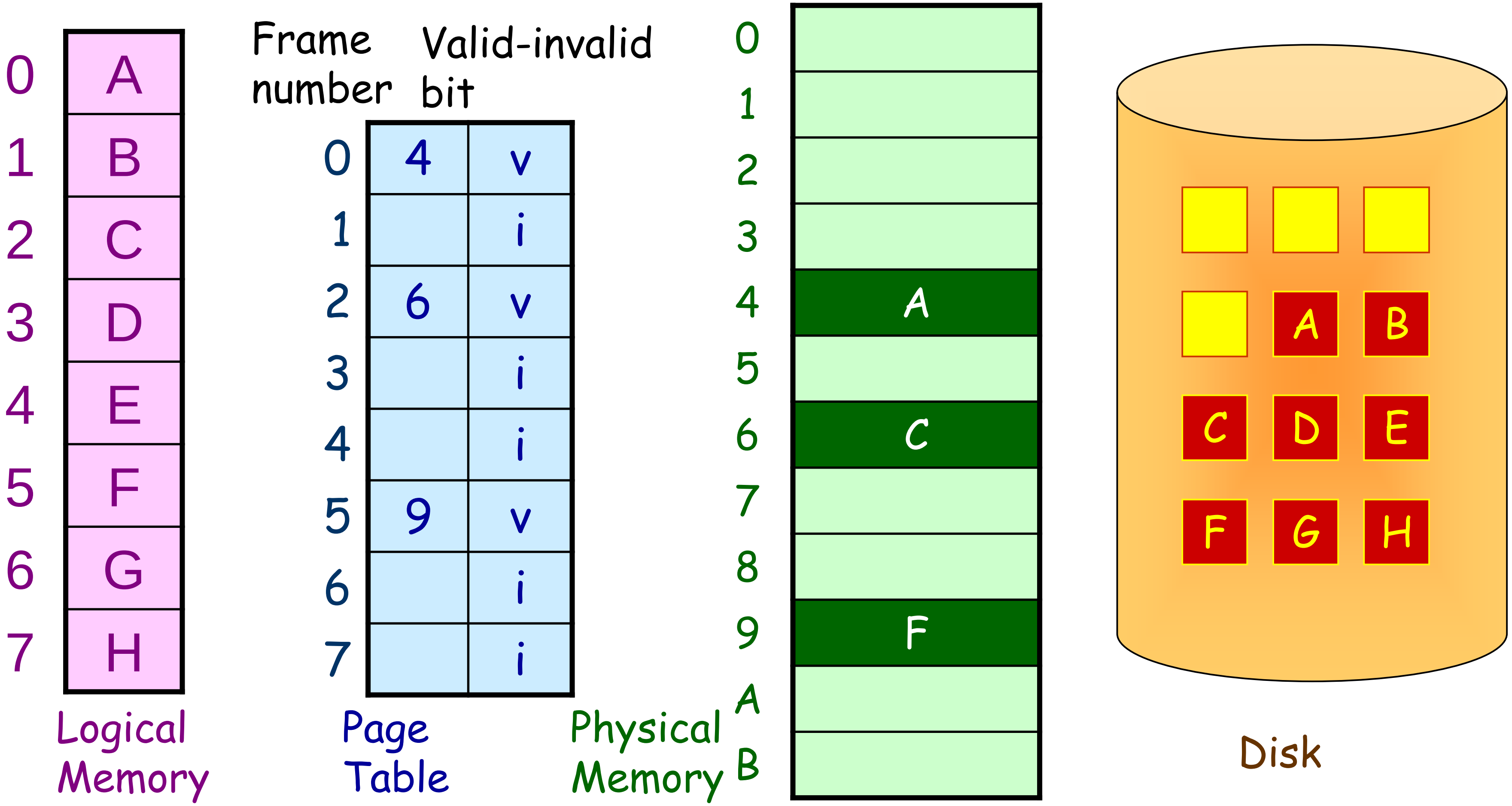
# Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - **v** ⇒ in-memory,
  - **i** ⇒ not-in-memory
- Example of a page table snapshot:



# Demand Paging

- Page Table When Some Pages Are Not in Main Memory



# Page Fault

- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:
  - During address translation, if valid–invalid bit in page table entry is **i** $\Rightarrow$ page fault

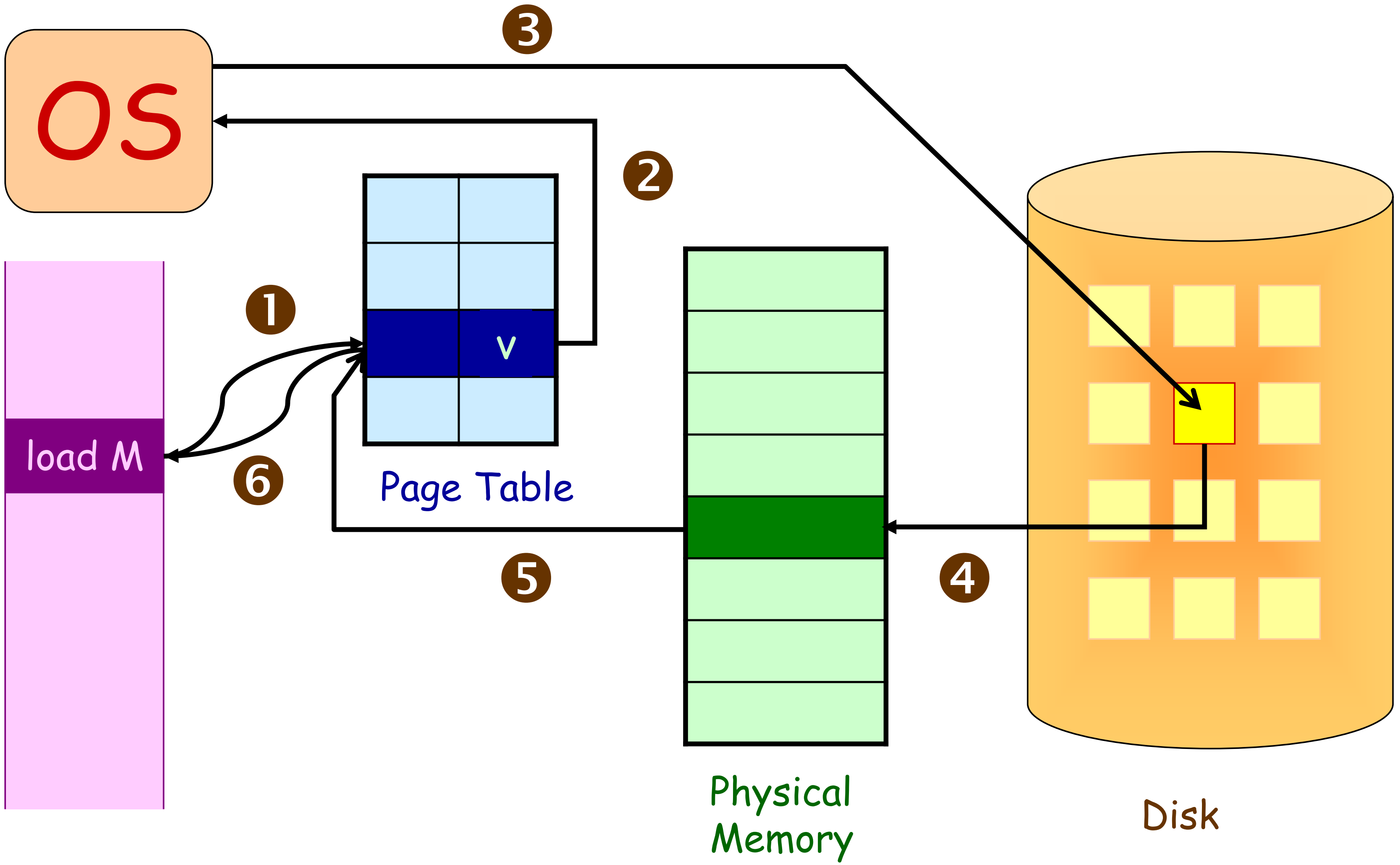
# Page Fault

- Reference to a page not in main memory will trap to operating system:

**page fault**

1. Operating system looks at the table to decide:
   ① Invalid reference $\Rightarrow$ abort
   ② Just not in memory
2. Get empty frame
3. Swap page into frame
4. Reset tables
5. Set validation bit = **v**
6. Restart the instruction that caused the page fault

# Steps in Handling a Page Fault



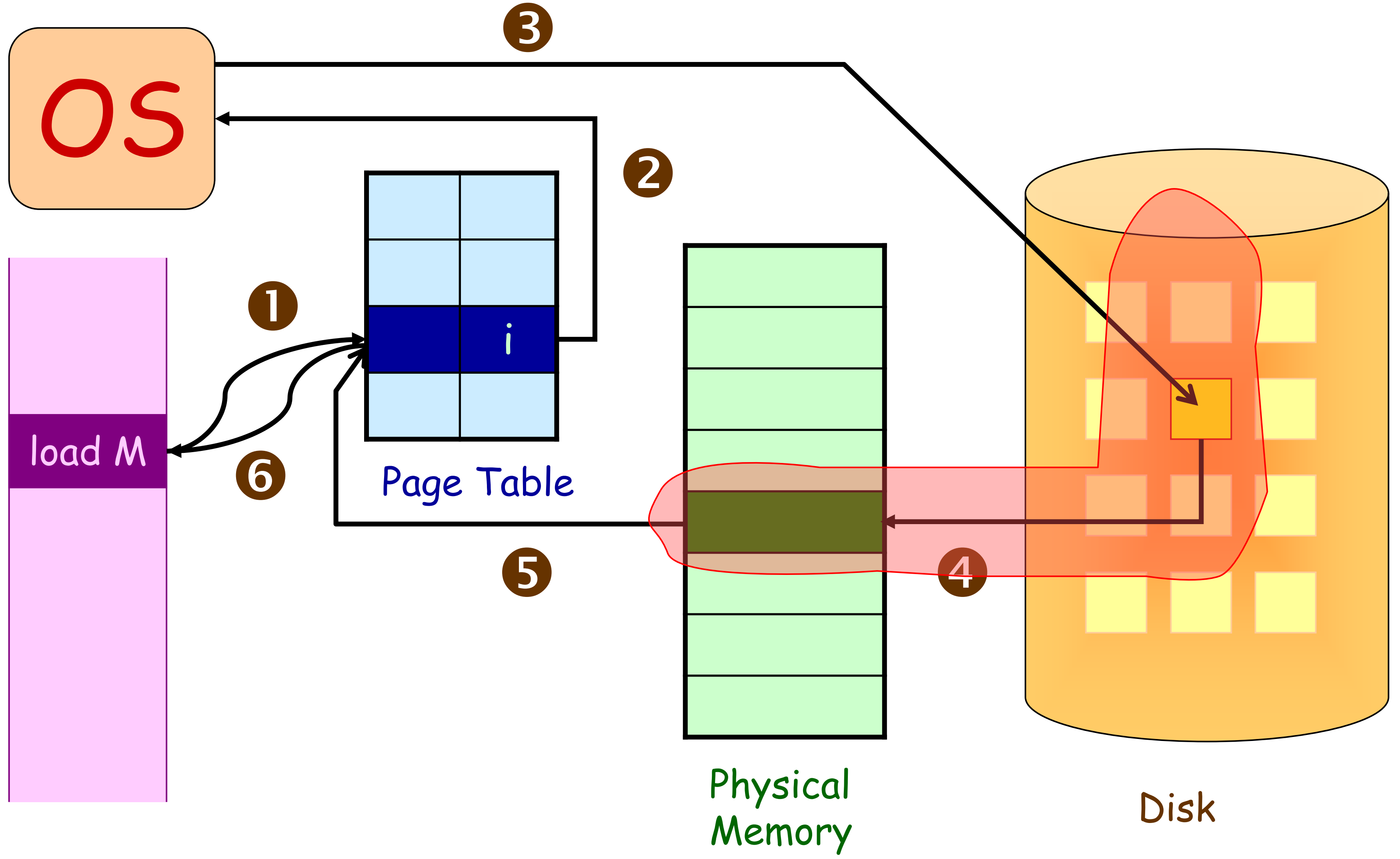
# Performance of Demand Paging

- Page Fault Rate $0 \le p \le 1.0$
  - if p = 0 no page faults
  - if p = 1, every reference is a fault
- Effective Access Time (EAT)

```
EAT = (1 – p) × memory access
        + p × ( page fault overhead
                    + swap page out
                    + swap page in
                    + restart overhead
            )
```

# Page Replacement

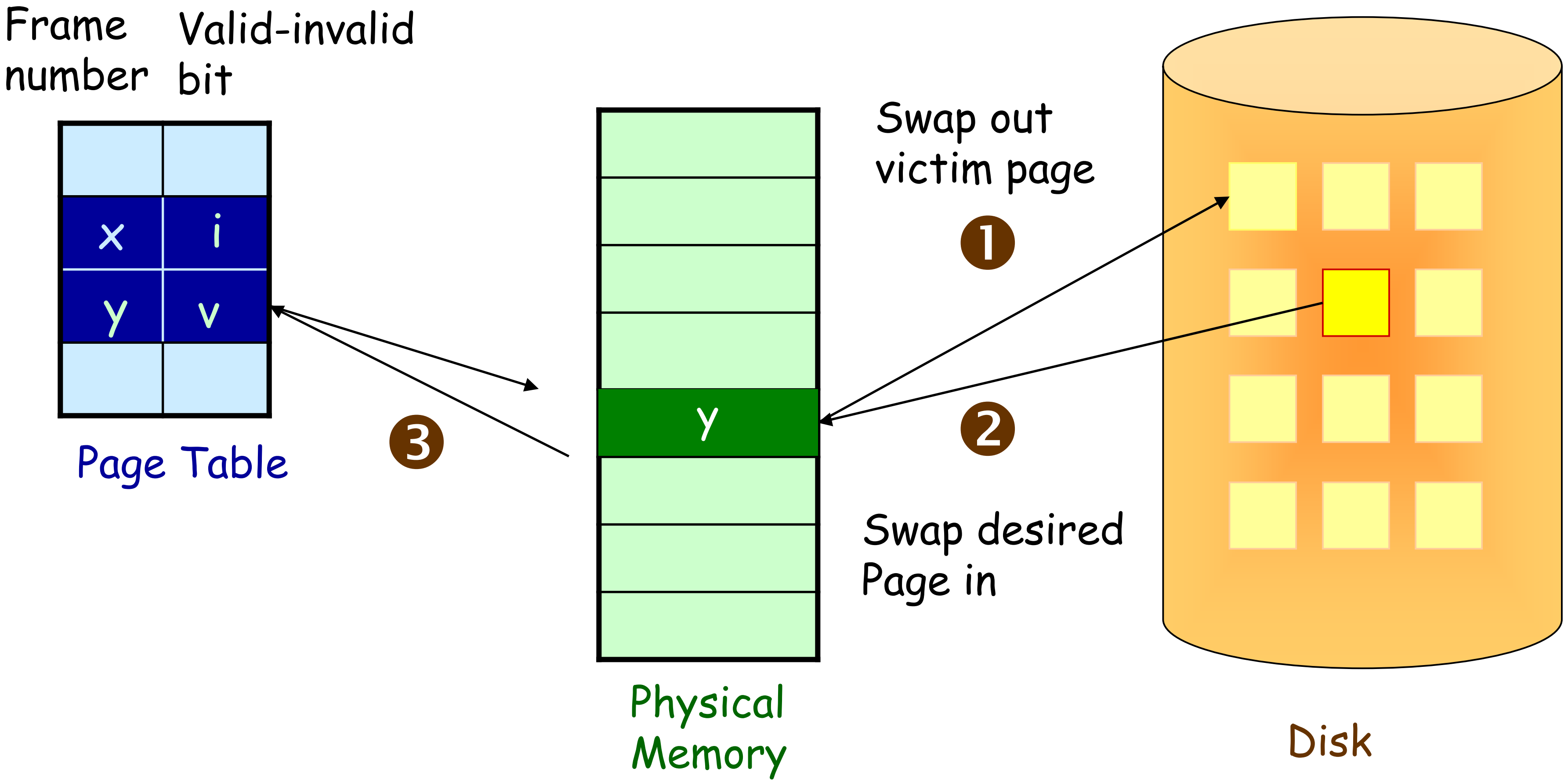
# Steps in Handling a Page Fault



# Need For Page Replacement

- Page replacement – find some page in memory, but not really in use, swap it out
  - algorithm
  - performance – want an algorithm which will result in minimum number of page faults
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk

# Page Replacement



# Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   ① If there is a free frame, use it
   ② If there is no free frame, use a page replacement algorithm to select a **victim** frame
3. Bring the desired page into the (newly) free frame; update the page table
4. Restart the process

# Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - e.g., Suppose we have 4 virtual pages, and following reference stream:
    - A B C A B D A D B C B

# Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk
- What about RANDOM?
  - Pick random page for every replacement
  - Typical solution for TLB's.  Simple hardware
  - Pretty unpredictable

# Page Replacement Policies

- What about FIFO (First In, First Out)?
  - Throw out oldest page.  Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages

# Example: FIFO

- Suppose we have 3 frames, 4 virtual pages, and following reference stream:

  A B C A B D A D B C B

- Consider FIFO Page replacement:

| Ref: Page: | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | | D | | | | C | |
| 2 | | B | | | | | A | | | | |
| 3 | | | C | | | | | | B | | |

  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

# Example: FIFO

- Reference string:

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

- 3 frames (3 pages can be in memory at a time per process)

| FIFO | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tail | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 5 | 5 | 3 | 4 | 4 |
| | | 1 | 2 | 3 | 4 | 1 | 2 | 2 | 2 | 5 | 3 | 3 |
| head | | | 1 | 2 | 3 | 4 | 1 | 1 | 1 | 2 | 5 | 5 |
| fault | X | X | X | X | X | X | X | √ | √ | X | X | √ |

# Example: FIFO

- Reference string:

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

- 4 frames

| FIFO | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tail | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |
| | | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 1 | 2 | 3 | 4 |
| | | | 1 | 2 | 2 | 2 | 3 | 4 | 5 | 1 | 2 | 3 |
| head | | | | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 1 | 2 |
| fault | X | X | X | X | √ | √ | X | X | X | X | X | X |

# Example: FIFO

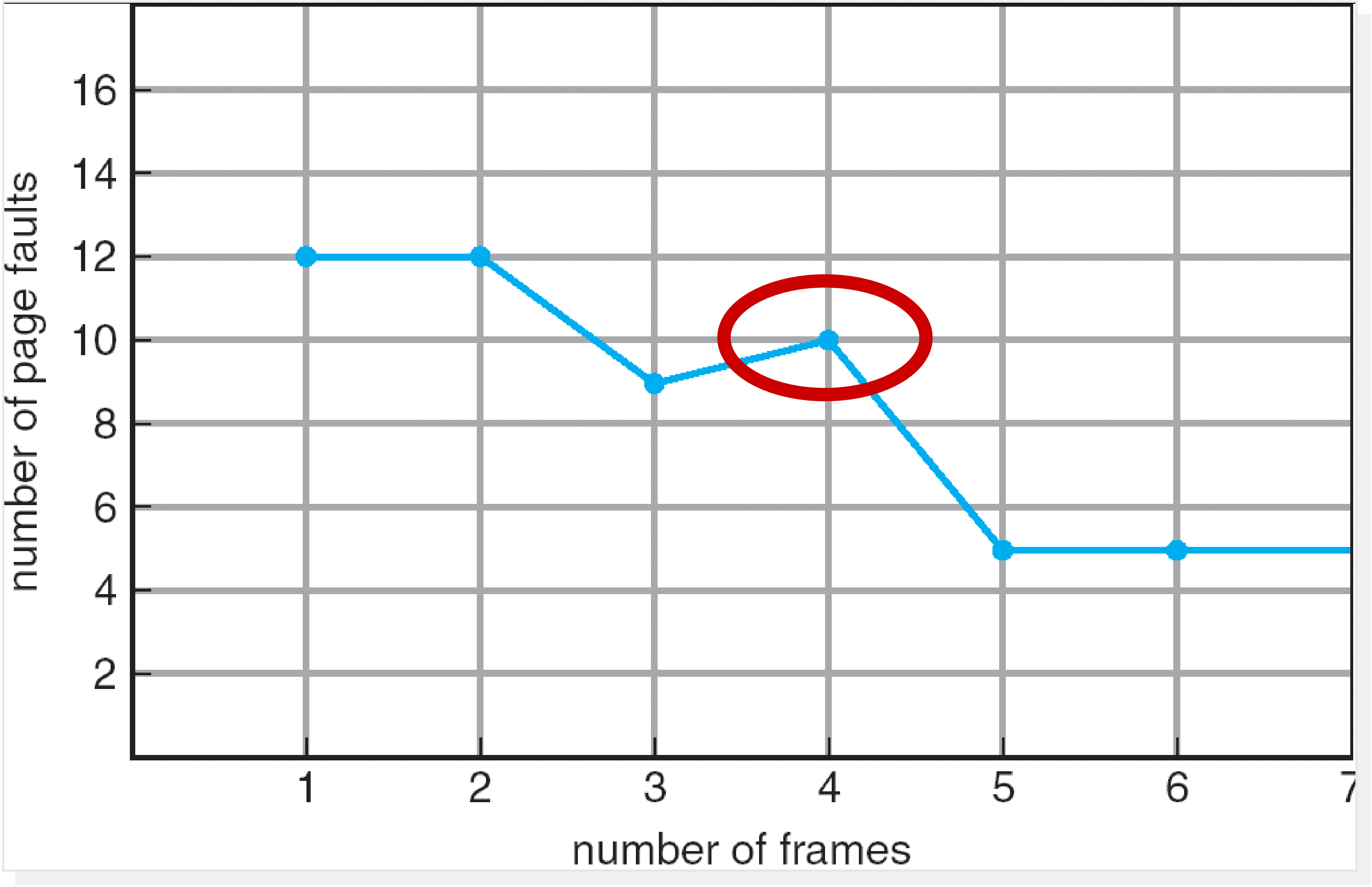
- Reference string:

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

- 3 frames
  - 9 page faults
- 4 frames
  - 10 page faults

- Belady's Anomaly: more frames ⇒ more page faults

# FIFO Illustrating Belady's Anomaly



# Page Replacement Policies

- What about MIN (Minimum) / OPT (Optimal)?
  - Replace page that won’t be used for the longest time

# Example: MIN/OPT

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider MIN Page replacement:

| Ref: / Page: | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | | | | | | C | |
| 2 | | B | | | | | | | | | |
| 3 | | | C | | | D | | | | | |

  - MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future.

# Page Replacement Policies

- What about MIN (Minimum) / OPT (Optimal)?
  - Replace page that won’t be used for the longest time
  - Great, but can’t really know future…
  - Makes good comparison case, however

# Replacement Policies (Con't)

- OPT
  - Replace page that **won't be** used for the longest time
- What about LRU(Least-Recently-Used)?
  - Replace page that **hasn't been** used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.

# Example: LRU

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider LRU Page replacement:

| Ref: / Page: | A | B | C | A | B | D | A | D | B | C | B |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | | | | | | C | |
| 2 | | B | | | | | | | | | |
| 3 | | | C | | | D | | | | | |

  - 5 faults
  - Where will D be brought in? Look for page hasn't been used for the longest time
- What will LRU do?
  - Same decisions as MIN here, but won't always be true!

# When will LRU perform badly?

- **Consider the following: A B C D A B C D A B C D**
- **LRU Performs as follows (same as FIFO here):**

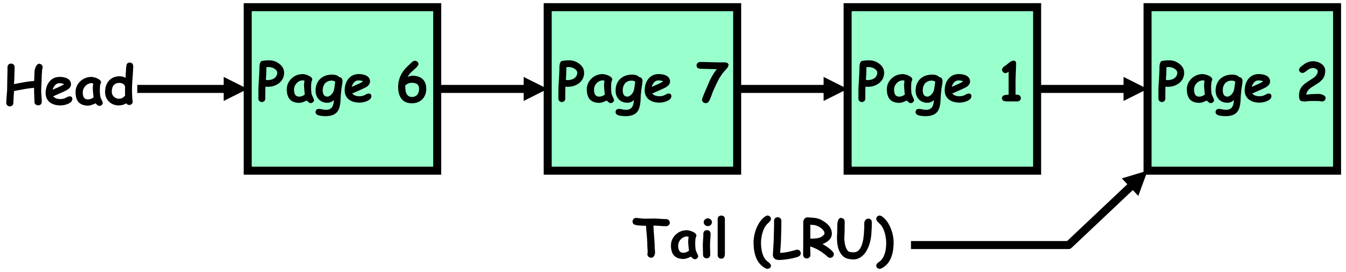
| Ref: Page: | A | B | C | D | A | B | C | D | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | D | | | C | | | B | | |
| 2 | | B | | | A | | | D | | | C | |
| 3 | | | C | | | B | | | A | | | D |

  - Every reference is a page fault!
- **MIN Does much better:**

| Ref: Page: | A | B | C | D | A | B | C | D | A | B | C | D |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | A | | | | | | | | | B | | |
| 2 | | B | | | | | C | | | | | |
| 3 | | | C | D | | | | | | | | |

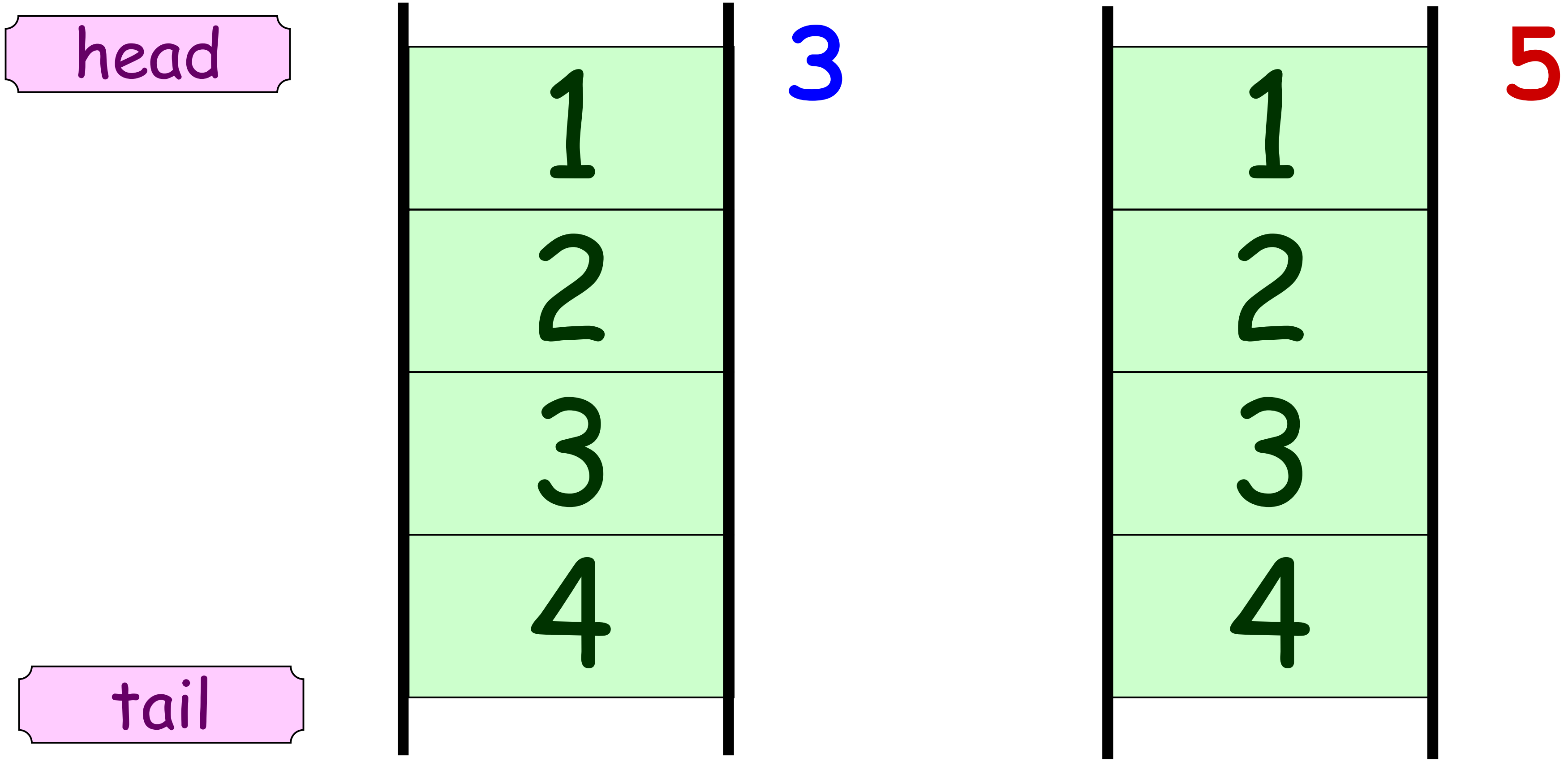
# Replacement Policies (Con't)

- How to implement LRU? Use a list!




  - On each use, remove page from list and place at head
  - LRU page is at tail
  - No search for replacement
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to determine which are to change

# Least Recently Used (LRU) Algorithm



# Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

| LRU | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| tail | 1 | 1 | 1 | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 1 | 2 |
| | | 2 | 2 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 |
| | | | 3 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 |
| head | | | | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
| Fault | X | X | X | X | √ | √ | X | √ | √ | X | X | X |

# Replacement Policies (Con't)

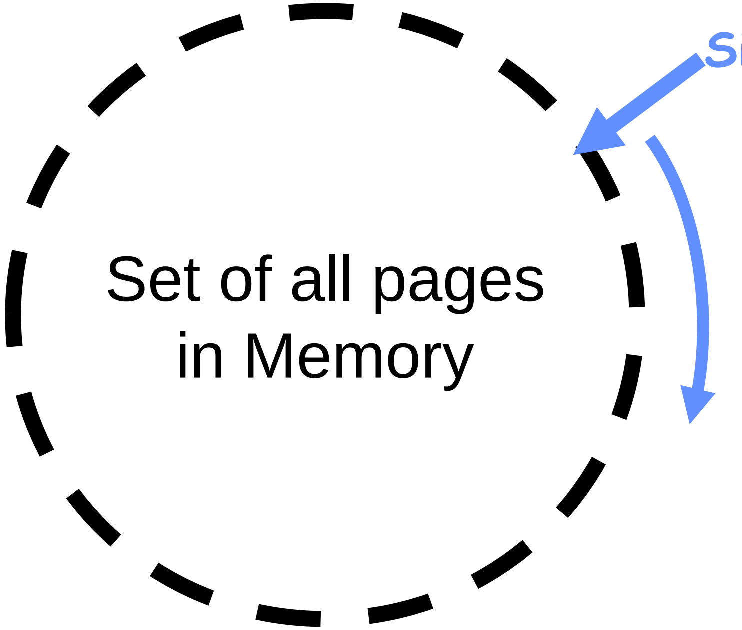
- Problems with this scheme for paging?
  - Too expensive to implement in reality for many reasons
- In practice, people approximate LRU

# 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

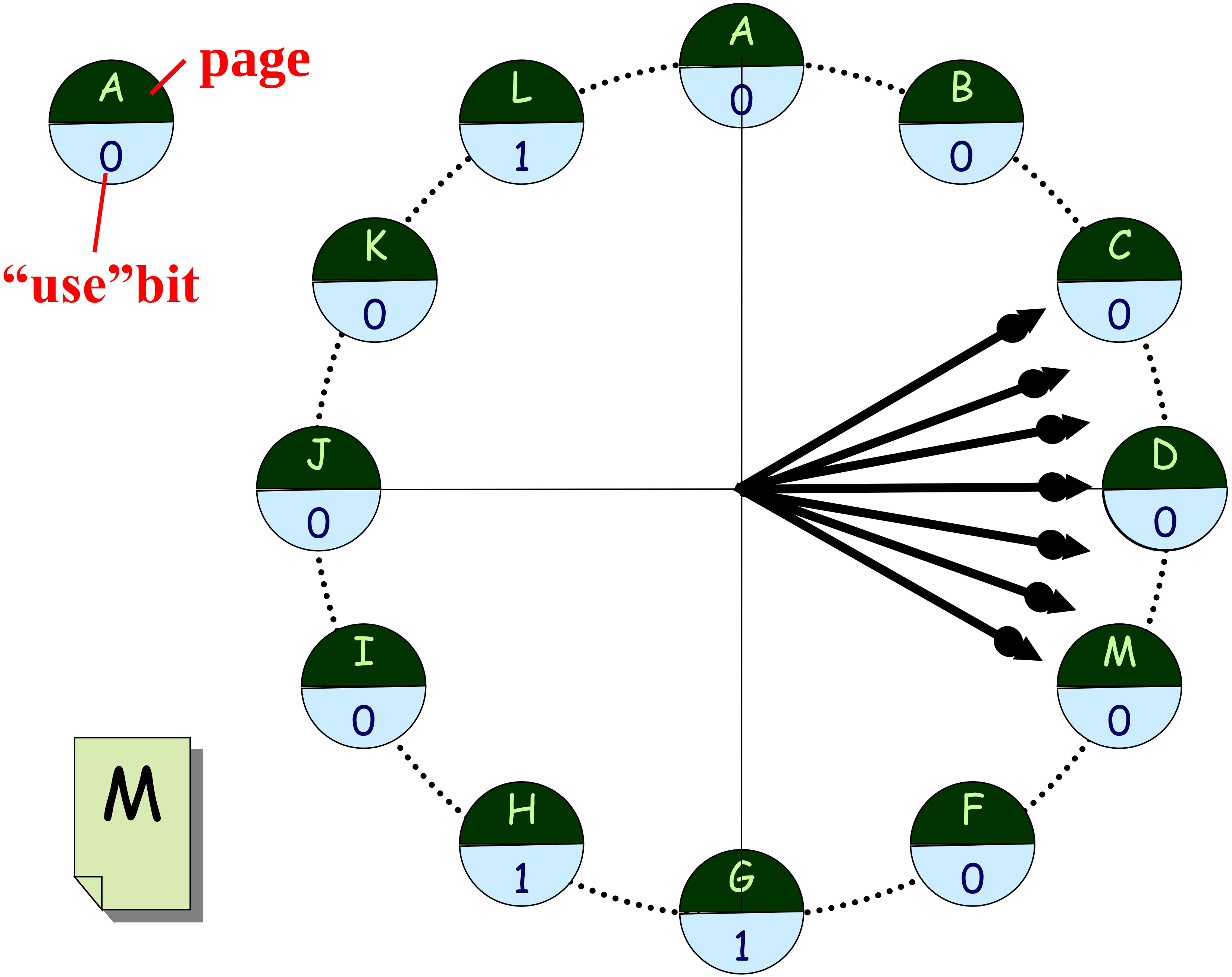
# Clock Algorithm

- Clock Algorithm: Arrange physical pages in circle with single clock hand
  - Approximate LRU (approximate to MIN)
  - Replace an old page, not the oldest page




**Single Clock Hand:**

Advances only on page fault!
Check for pages not used recently (use bit)
Mark pages as not used recently


A
0
page
"use"bit
L
1
A
0
B
0
K
0
C
0
J
0
D
0
I
0
M
0
H
1
G
1
F
0
M

# Clock Algorithm

- Details:
  - On page fault:
    - Advance clock hand
    - Check use bit:
      - 1 used recently; clear and leave alone
      - 0 selected candidate for replacement
  - Will always find a page or loop forever?
    - Even if all use bits set, will eventually loop around FIFO

# Second chance (self-study)

- Second chance
  - Need reference bit
  - If page to be replaced (in clock order) has reference bit = 1 then:
    - set reference bit 0
    - leave page in memory
    - replace next page (in clock order), subject to same rules

# $N^{th}$ Chance version of Clock Algorithm

(self-study)

- $N^{th}$ chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks use bit:
    - 1⇒clear use and also clear counter to 1(used in last sweep)
    - 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx. to LRU
    - If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - Otherwise might have to look a long way to find free page

# Enhanced Second-Chance / Not recently used

- A clean page vs. a dirty page
  - Swap which out?
- Considers both reference bit and modify bit.
  - when a page is referenced, a referenced bit is set for that page.
  - Similarly, when a page is modified (written to), a modified bit is set.

# Enhanced Second-Chance / Not recently used

| Page number | Referenced Bit | Modified Bit |
|---|---|---|
| 0 | 1 | 1 |
| 1 | 1 | 0 |
| 2 | 1 | 1 |
| 3 | 0 | 0 |
| 4 | 1 | 0 |
| 5 | 1 | 1 |
| 6 | 0 | 1 |
| 7 | 0 | 1 |

# Enhanced Second-Chance / Not recently used

- At a certain fixed time interval, the clock interrupt clears the referenced bit of all the pages, so only pages referenced within the current clock interval are marked with a referenced bit.
- When a page needs to be replaced, the operating system divides the pages into four classes:
  - Class 0: not referenced, not modified
  - Class 1: not referenced, modified
  - Class 2: referenced, not modified
  - Class 3: referenced, modified
- The NRU(Not Recently Used) algorithm picks a random page from the lowest category for removal.

# Other Page Replacement Algorithms

- Counting Algorithms
  - Keep a counter of the number of references that have been made to each page
  - **LFU Algorithm (Least Frequently Used):** replaces page with smallest count (最少使用)
  - **MFU Algorithm(Most Frequently Used)**: replaces page with biggest count
    - based on the argument that the page with the smallest count was probably just brought in and has yet to be used

# Aging Algorithm

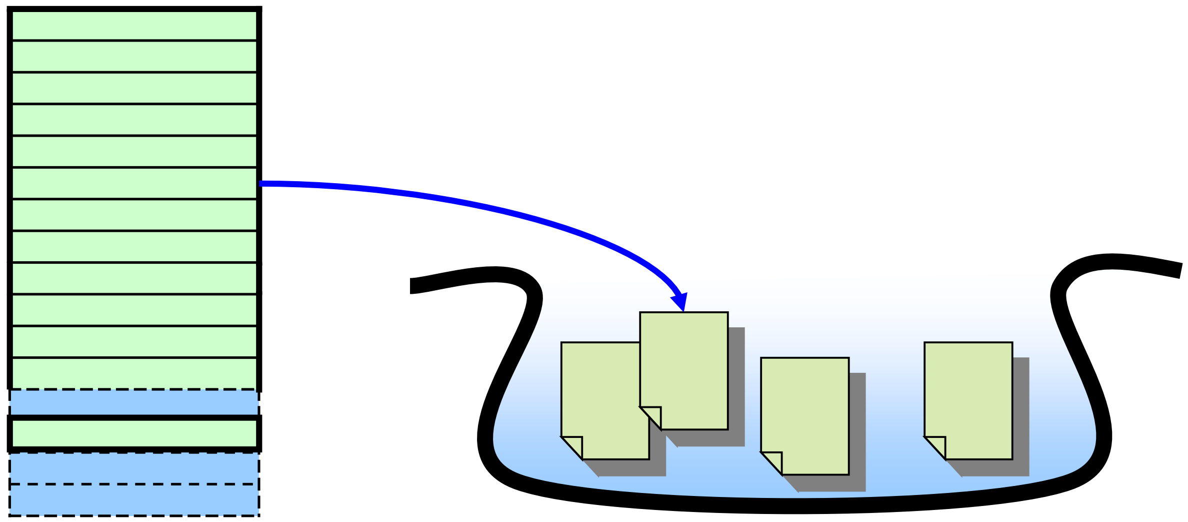
- Using reference-bits

| R bits for page 0-3 clock tick 0 | R bits for page 0-3 clock tick 1 | R bits for page 0-3 clock tick 2 | R bits for page 0-3 clock tick 3 |
|---|---|---|---|
| 1 0 1 0 | 1 1 0 0 | 1 1 0 1 | 1 0 0 0 |

| | | | | | |
|---|---|---|---|---|---|
| 0 | 00000000 | 10000000 | 11000000 | 11100000 | 11110000 |
| 1 | 00000000 | 00000000 | 10000000 | 11000000 | 01100000 |
| 2 | 00000000 | 10000000 | 01000000 | 00100000 | $\boxed{00010000}$ |
| 3 | 00000000 | 00000000 | 00000000 | 10000000 | 01000000 |

# Page Buffering Algorithm

- As an add-on to any previous algorithm.
- A **pool** of free frames is maintained.
- When a page fault occurs, the desired page is read into a free frame from the pool. The victim frame is later swapped out if necessary and put into the free frames pool.

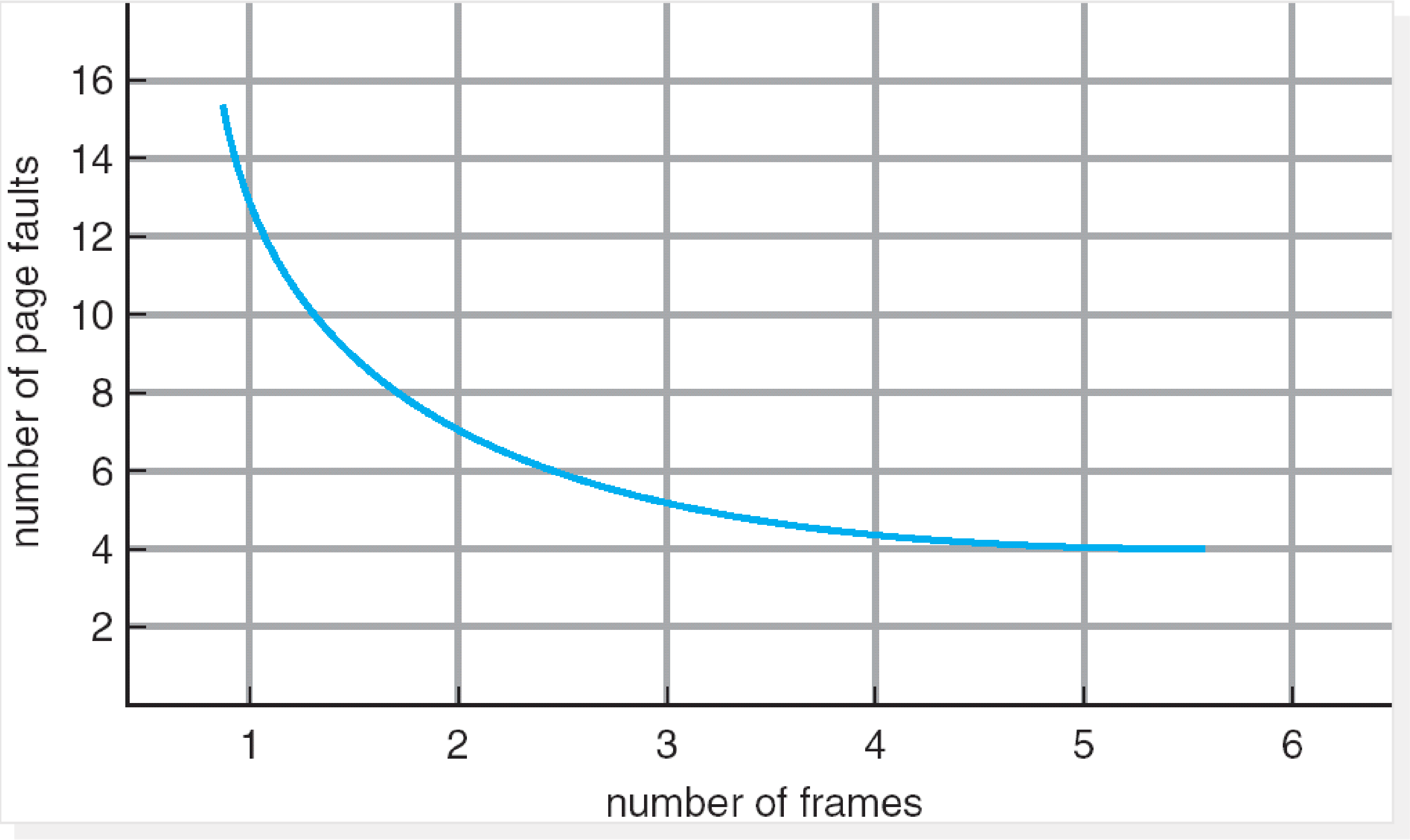


# Page Buffering Algorithm

- Advantage / disadvantage ?
  - Plus - faster.
  - Minus - less pages are in use overall.

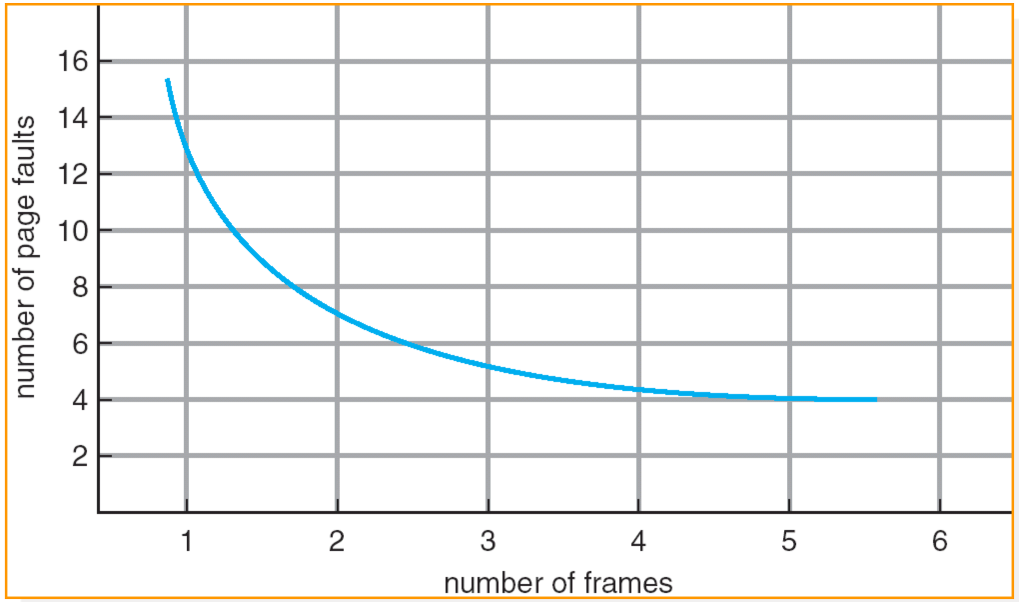
# Page Replacement

- Graph of Page Faults Versus The Number of Frames



# Graph of Page Faults Versus The Number of Frames

- One desirable property:
  When you add memory
  the miss rate goes down
  - Does this always happen?
  - Seems like it should, right?




- No: BeLady’s anomaly
  - Certain replacement algorithms (FIFO) don’t have this obvious property!

# Does adding memory reduce number of page faults?

- Not necessarily for FIFO!
- Yes for LRU and MIN
- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

# Allocation of Frames

# Allocation of Frames

- Each process needs minimum number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
  - Example: IBM 370 – 6 pages to handle Storage to Storage MOVE instruction:
    - instruction is 6 bytes, might span 2 pages.
    - 2 pages to handle from
    - 2 pages to handle to
- Two major allocation schemes
  - fixed allocation
  - priority allocation

# Fixed Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes, per process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

    $s_i$ = size of process $p_i$ and $S = \Sigma s_i$

    $m$ = total number of frames

    $a_i$ = allocation for $p_i = ( s_i/S )\times m$

# Priority Allocation

- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme

# Replacement Policies

- Possible Replacement Scopes:
  - **Global replacement** – process selects replacement frame from set of all frames; one process can take a frame from another
  - **Local replacement** – each process selects from only its own set of allocated frames

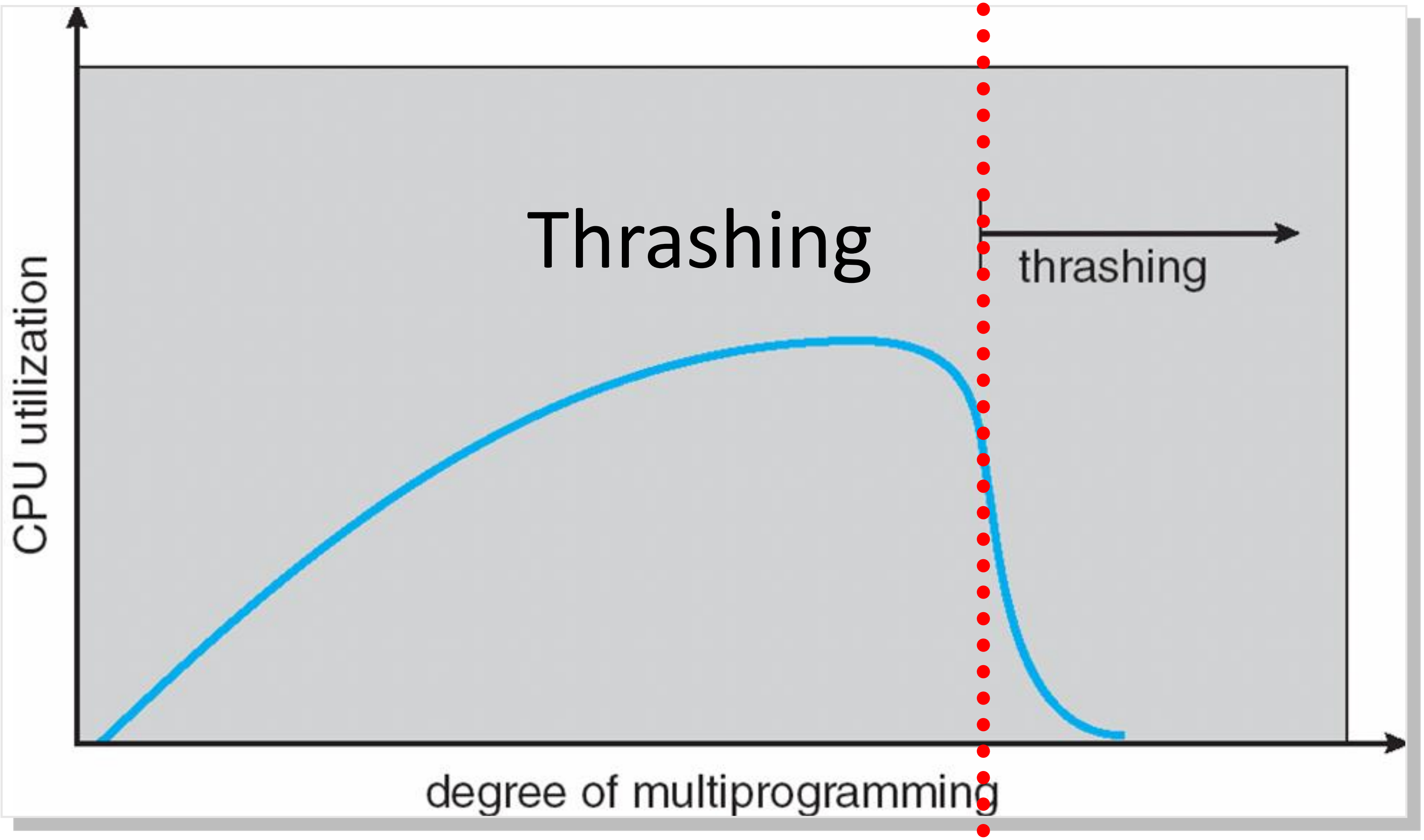
# Thrashing

# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system thinks that it needs to increase the degree of multiprogramming
  - another process added to the system
- **Thrashing** ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing ?
  - What is best response to Thrashing ?

北京交通大學

# Thrashing



# Thrashing

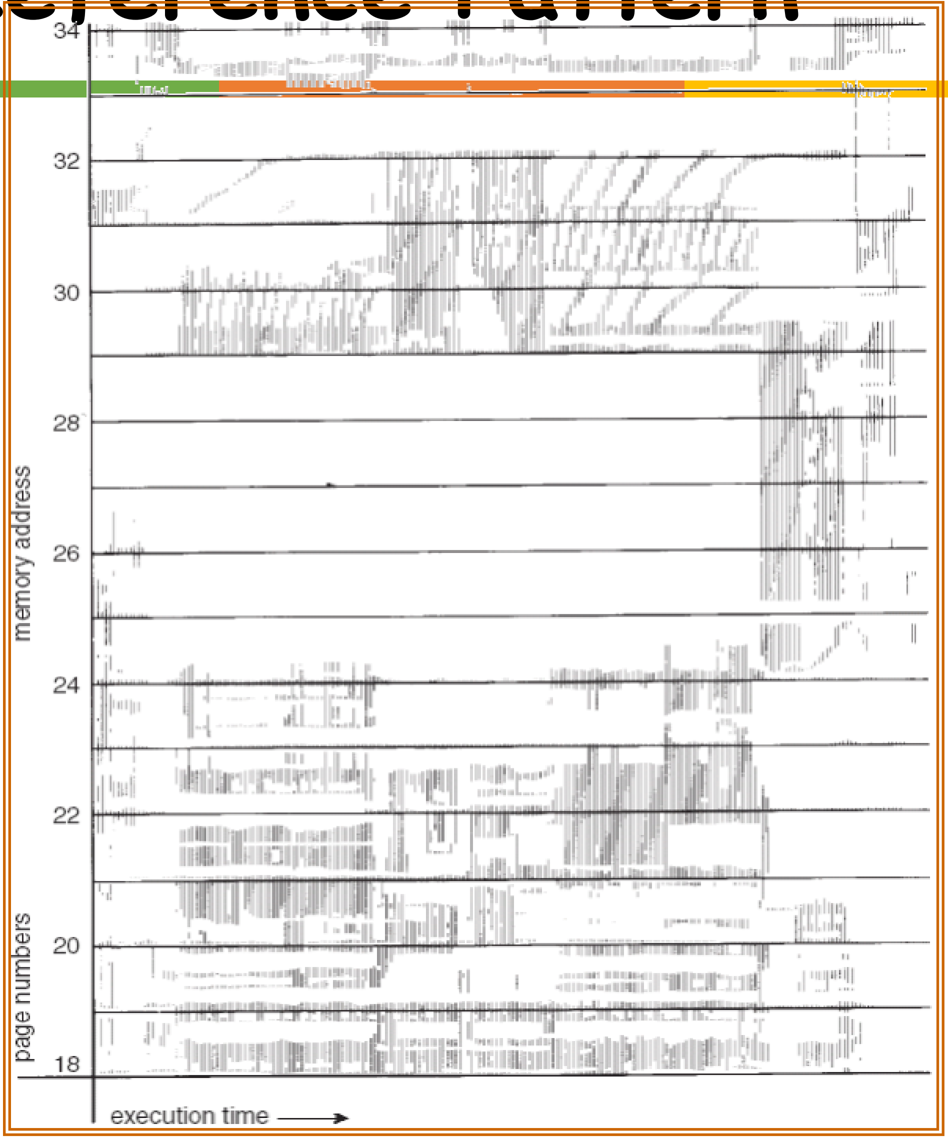
- If allocated frames < minimum number
  - ⇒ Very high paging activity
- A process is thrashing if it is spending more time paging than executing.

# Demand Paging and Thrashing

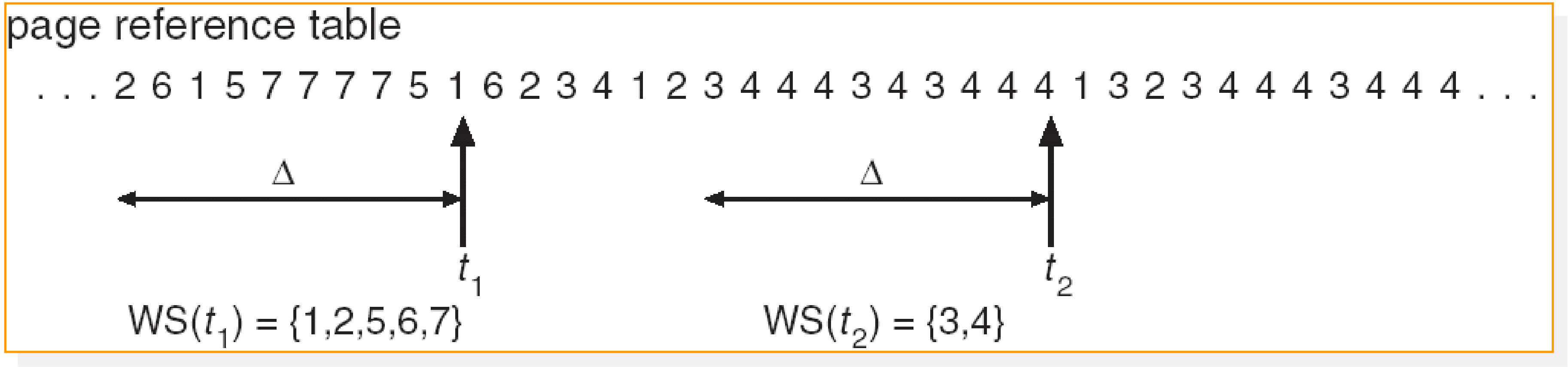
- Why does demand paging work?
  Locality model
- Why does thrashing occur?
  $\Sigma$ size of locality > total memory size

# Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "**Working Set**"
  - Working Set defines **minimum** number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing



# Working-Set Model




- $\Delta \equiv$ working-set window

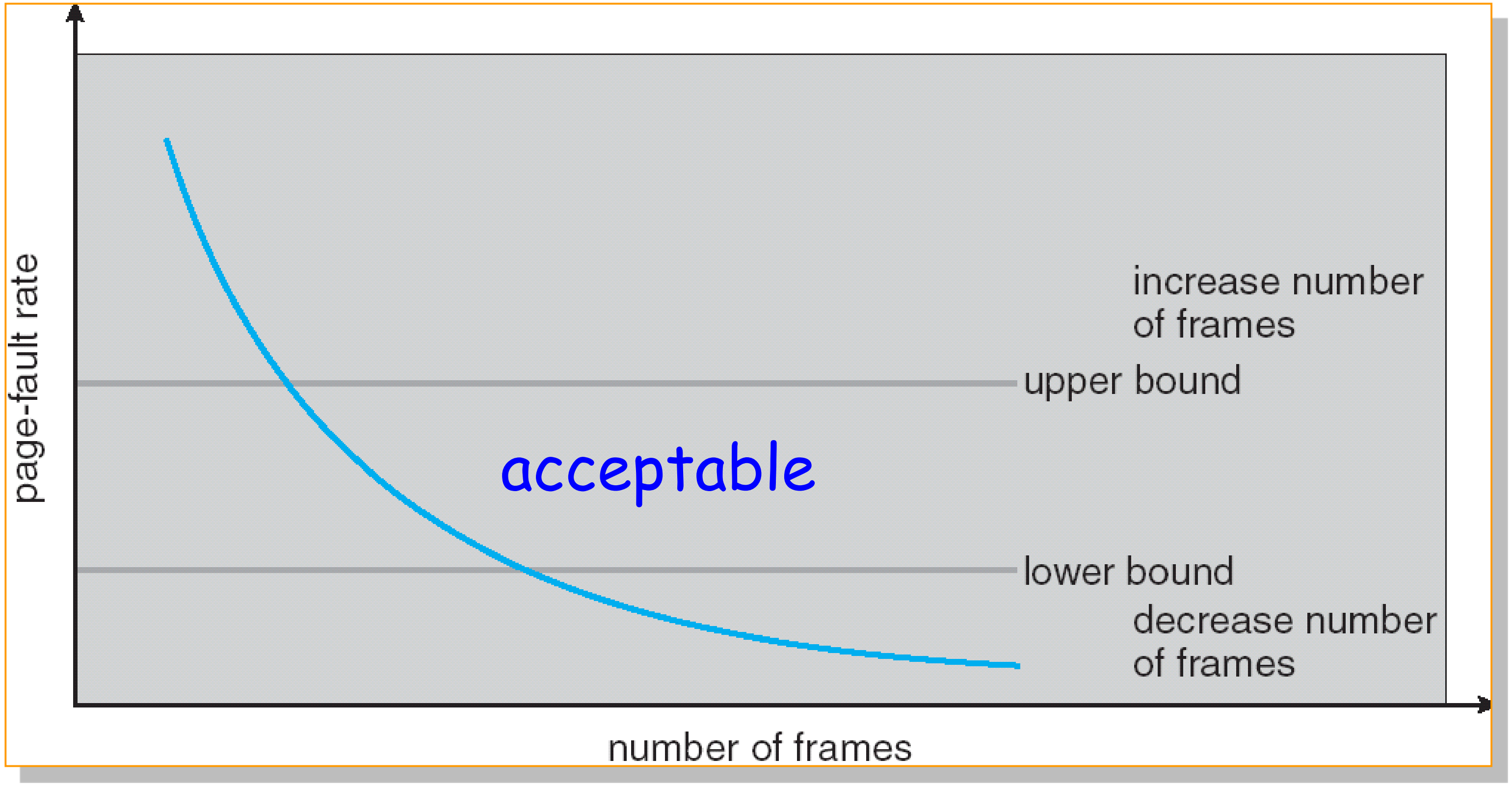
  $\equiv$ fixed number of page references

# Working-Set Model

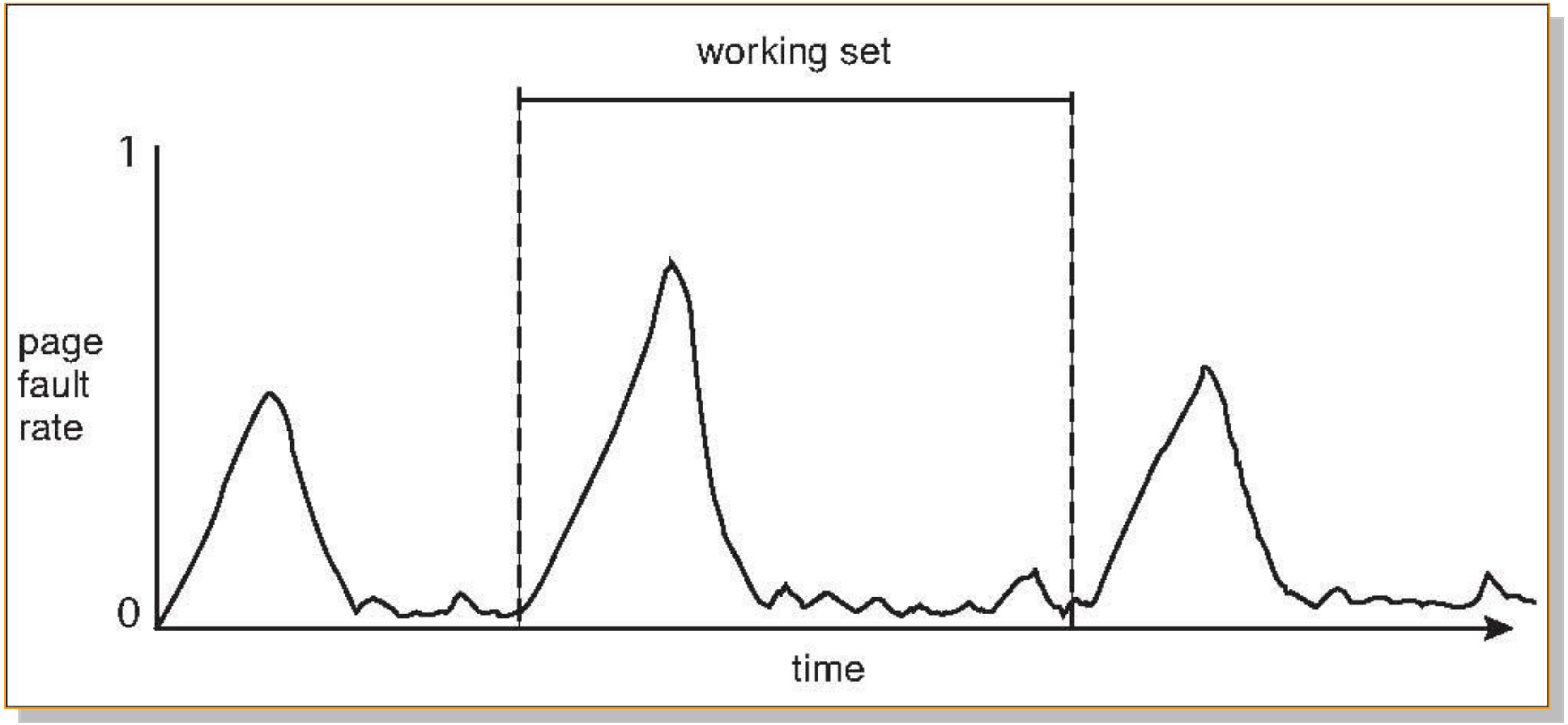
- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
- $WS_i$ (working set of Process $P_i$) = total set of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma|WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
  - Policy: if $D > m$, then suspend one of the processes
  - This can improve overall system behavior by a lot!

# Page-Fault Frequency Scheme

- Establish “acceptable” page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



# Working Sets and Page Fault Rates



# Windows XP

- Uses demand paging with **clustering**. Clustering brings in pages surrounding the faulting page
- Processes are assigned **working set minimum** and **working set maximum**
  - Default working set minimum: 50
  - Default working set maximum: 345
- Working set minimum is the minimum number of pages the process is guaranteed to have in memory
- A process may be assigned as many pages up to its working set maximum

# Windows XP

- When the amount of free memory in the system falls below a threshold, **automatic working set trimming** is performed to restore the amount of free memory
- Working set trimming removes pages from processes that have pages in excess of their working set minimum

# Other Considerations

# Prepaging

- To reduce the large number of page faults that occurs at process startup (e.g., pure demand-paging)
- Prepage all or some of the pages a process will need, before they are referenced
  - e.g., whole working set for a swapping-in process
- But if prepaged pages are unused, I/O and memory was wasted

# Page Size

- Usually, $2^{12}$(4K) ~ $2^{22}$ (4M) size
  - memory utilization (small internal fragmentation) ⇒ small size
  - minimize I/O time (less seek, latency) ⇒ large size
  - minimize number of page faults ⇒ large size
- Trend: larger

# Program Structure

- Program structure
  - `int data[128][128];`
  - 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

# Program Structure

| | | | |
|---|---|---|---|
| D[0,0] | D[0,1] | D[0,2] | D[0,3] |
| D[1,0] | D[1,1] | D[1,2] | D[1,3] |
| D[2,0] | D[2,1] | D[2,2] | D[2,3] |
| D[3,0] | D[3,1] | D[3,2] | D[3,3] |

| |
|---|
| D[0,0] |
| D[1,3] |
| D[2,3] |
| D[3,3] |

# Program Structure

北京交通大學

| | | | |
|---|---|---|---|
| D[0,0] | D[0,1] | D[0,2] | D[0,3] |
| D[1,0] | D[1,1] | D[1,2] | D[1,3] |
| D[2,0] | D[2,1] | D[2,2] | D[2,3] |
| D[3,0] | D[3,1] | D[3,2] | D[3,3] |

```
for (i = 0; i <4; i++)
   for (j = 0; j < 4; j++)
          D[i,j] = 0;
```

| |
|---|
| D[0,0] |
| D[0,1] |
| D[0,2] |
| D[0,3] |
| D[1,0] |
| D[1,1] |
| D[1,2] |
| D[1,3] |
| D[2,0] |
| D[2,1] |
| D[2,2] |
| D[2,3] |
| D[3,0] |
| D[3,1] |
| D[3,2] |
| D[3,3] |

| |
|---|
| D[3,0] |
| D[3,1] |
| D[3,2] |
| D[3,3] |

# Program Structure

北京交通大學

| | | | |
|---|---|---|---|
| D[0,0] | D[0,1] | D[0,2] | D[0,3] |
| D[1,0] | D[1,1] | D[1,2] | D[1,3] |
| D[2,0] | D[2,1] | D[2,2] | D[2,3] |
| D[3,0] | D[3,1] | D[3,2] | D[3,3] |

```
for (j = 0; j <4; j++)
   for (i = 0; i < 4; i++)
          D[i,j] = 0;
```

| |
|---|
| D[0,0] |
| D[0,1] |
| D[0,2] |
| D[0,3] |
| D[1,0] |
| D[1,1] |
| D[1,2] |
| D[1,3] |
| D[2,0] |
| D[2,1] |
| D[2,2] |
| D[2,3] |
| D[3,0] |
| D[3,1] |
| D[3,2] |
| D[3,3] |

| |
|---|
| D[3,0] |
| D[3,1] |
| D[3,2] |
| D[3,3] |

# Win32 API

- malloc
- free
- VirtualAlloc
- VirtualFree
- ZeroMemory
- GetSystemInfo
- GlobalMemoryStatus

# Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - Temporal Locality: Locality in Time
    - Spatial Locality: Locality in Space
- Demand Paging:
  - Treat memory as cache on disk
  - Cache miss $\Rightarrow$ get page from disk
- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - The cost of being wrong is high: must go to disk

# Summary

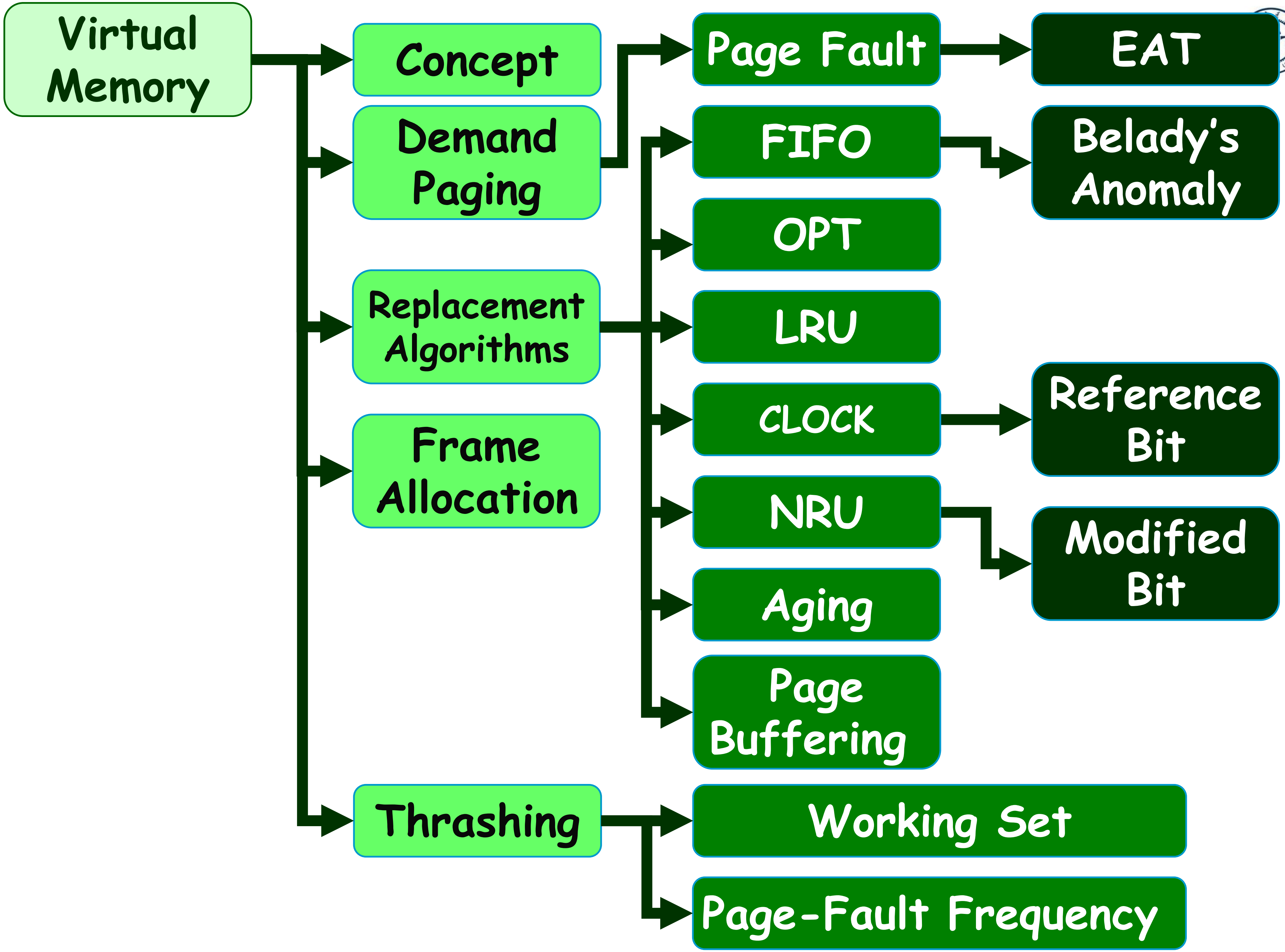
- Replacement policies
  - RANDOM
    - Pick random page for every replacement
    - Typical solution for TLB's.  Simple hardware
    - Pretty unpredictable
  - FIFO (First In, First Out)
    - Throw out oldest page.
    - Place pages on queue, replace page at end
    - Bad, because throws out heavily used pages instead of infrequently used pages
  - OPT/MIN (Optimal/Minimum)
    - Replace page that won't be used for the longest time in future
    - Great, but can't really know future…
    - Makes good comparison case, however

# Summary

- Replacement policies
  - LRU
    - Replace page used farthest in past / Replace page that hasn't be used for the longest time
    - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
    - Seems like LRU should be a good approximation to MIN.
    - Implemented LRU using a list
      - On each use, remove page from list and place at head
      - LRU page is at tail
  - Clock Algorithm: Approximation to LRU
    - Arrange all pages in circular list
    - Sweep through them, marking as not "in use"
    - If page not "in use" for one pass, than can replace
  - NRU：(Not *Recently Used*）
  - Aging Algorithm

# Summary

- Allocation of Frames
  - fixed allocation
    - Equal allocation
    - Proportional allocation
  - priority allocation
  - Global replacement / Local replacement
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn’t fit in memory
  - Need to swap out a process
- Working Set:
  - Set of pages touched by a process recently
- Page-Fault Frequency Scheme



Virtual Memory
Concept
Demand Paging
Replacement Algorithms
Frame Allocation
Thrashing
Page Fault
EAT
FIFO
Belady's Anomaly
OPT
LRU
CLOCK
Reference Bit
NRU
Modified Bit
Aging
Page Buffering
Working Set
Page-Fault Frequency