Final Exam Review

Operating Systems

Yu-Ting Wu

Scope of the Final Exam

- Chapter 6/7: Synchronization
- Chapter 8: Deadlocks
- Chapter 9: Main memory
- Chapter 10: Virtual memory

Synchronization

Overview

The Goals of Synchronization

- **Concurrent access** to **shared data** may result in **data inconsistency**
  - Prevent **race condition**: The final value of the shared data depends upon which process finishes last
  - Solve the **critical section problem**

- Maintaining data consistency requires mechanism to ensure the **orderly execution** of cooperating processes

The Critical-Section Problem

- A **protocol** for processes to cooperate
- **General code section structure**
  - Only one process can be in a critical section

do {

entry section → get **entry permission**

critical section → modified **shared data**

exit section → release **entry permission**

remainder section

} while (1);

Critical-Section Requirements

- **Mutual exclusion**
  - If a process *P* is executing in its critical section (CS), no other processes can be executing in their CS
- **Progress**
  - If no process is executing in its CS and there exist some processes that wish to enter their CS, there processes cannot be postponed indefinitely
- **Bounded Waiting**
  - A **bound** must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS
- Goal: design **entry** and **exit section** to satisfy the above requirement

CS Solutions and Synchronization Tools

- Software solution
- Synchronization hardware
- Semaphore
- Monitor

Synchronization
Software Solution

Algorithm for Two Processes

- Only 2 processes P_0 and P_1
- Shared variables
  - int *turn*; // initially *turn* = 0
  - *turn* == *i* ➔ P_i can enter its critical section

```
/* Process 0 */
do {
    while (turn != 0);   ── entry section
        critical section
    turn = 1;            ── exit section
        remainder section
} while (1);

/* Process 1 */
do {
    while (turn != 1);   ── entry section
        critical section
    turn = 0;            ── exit section
        remainder section
} while (1);
```

mutual exclusion? **Y** progress? **N** bounded-wait? **Y**

Peterson's Solution for Two Processes

- Shared variables
  - int *turn*; // initially *turn* = 0
  - *turn == i* ➔ P_i can enter its critical section
  - boolean *flag[2]*; // initially *flag[0] = flag[1] = false*
  - *flag[i] == true* ➔ P_i is ready to enter its critical section

```
/* Process i */
do {
      flag[i] = true;
      turn = j;                          — entry section
      while (flag[j] && turn == j);
        critical section
      flag[i] = false;                   — exit section
        remainder section
} while (1);
```

mutual exclusion? **Y** progress? **Y** bounded-wait? **Y**

Bakery Algorithm for *n* Processes

- Before entering its CS, each process receives a **number (#)**
- **Holder of the smallest # enters CS**
- The numbering scheme always generates # in **non-decreasing order**; i.e., 1, 2, 3, 3, 4, 5, 5, 5 …
- If processes P_i and P_j receive the same #, if $i < j$, then P_i is served first
- Notation:
  - $(a, b) < (c, d)$ if
    - $a < c$ or
    - $a == c$ && $b < d$

Bakery Algorithm for *n* Processes (cont.)

get ticket

FCFS

release ticket

```
// Process i:
do {
    choosing[i] = true;
    num[i] = max (num[0], num[1], …num[n-1]) + 1;
    choosing[i] = false;
    for (j = 0; j < n; j++) {
        while (choosing[j]);
        while ((num[j] != 0) && ((num[j], j) < (num[i], i)));
    }
    critical section
    num[i] = 0;
    remainder section
}
```

Bounded waiting because processes enter CS on a **first come, first served** basis

Condition Variables

- **Condition variables (CV)** represent some condition that a **thread** can
  - **Wait on**, until the condition occurs; or
  - **Notify** other waiting threads that the condition has occurred
- Three operations on condition variables
  - **wait()** – block until another thread calls **signal()** or **broadcast()** on the CV

    pthread_cond_wait(&theCV, &someLock)
  - **signal()** – wake up one thread waiting on the CV

    pthread_cond_signal(&theCV)
  - **broadcast()** – wake up all threads waiting on the CV

    pthread_cond_broadcast(&theCV)

Synchronization
Hardware Solution

Atomic TestAndSet()

```
bool TestAndSet (bool &lock) {
        bool value = lock;
        lock = true;
        return value;
}
```

execute atomically:
return the value of “lock” and **set** “lock” to true

```
shared data: bool lock;    // initially lock = false
```

$// P_0$

```
do {
        while (TestAndSet (lock));
        critical section
        lock = false;
        remainder section
} while (1);
```

$// P_1$

```
do {
        while (TestAndSet (lock));
        critical section
        lock = false;
        remainder section
} while (1);
```

mutual exclusion? **Y** progress? **Y** bounded-wait? **N**

Atomic Swap()

Enter CS if lock == false

```
shared data: bool lock;    // initially lock = false
```

```
// P0
do {
    key0 = true;
    while (key0 == true)
        Swap(lock, key0);
    critical section
    lock = false;
    remainder section
} while (1);
```

```
// P1
do {
    key1 = true;
    while (key1 == true)
        Swap(lock, key1);
    critical section
    lock = false;
    remainder section
} while (1);
```

mutual exclusion? **Y** progress? **Y** bounded-wait? **N**

Synchronization

Semaphores

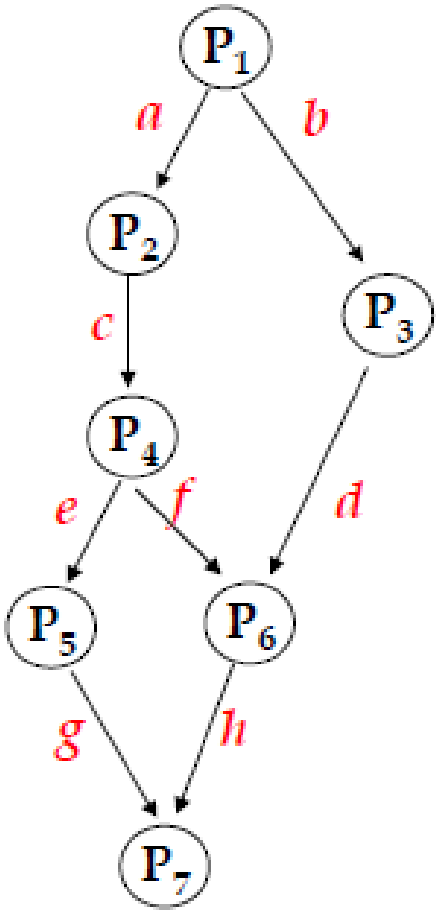
Semaphores

- A tool to generalize the synchronization problem
- More specifically
  - A **record** of **how many units of a particular resource is available**
    - If # record = 1 ➔ binary semaphore, **mutex lock**
    - If # record > 1 ➔ counting semaphore
  - Accessed only through 2 **atomic** operations: **wait** & **signal**

- Implementation
  - Spinlock
  - Data structure with queue

Semaphore Example

- Initially, all semaphores are 0
- Begin
  - P1: S1; signal(a); signal(b);
  - P2: wait(a); S2; signal(c);
  - P3: wait(b); S3; signal(d);
  - P4: wait(c); S4; signal(e); signal(f);
  - P5: wait(e); S5; signal(g);
  - P6: wait(f); wait(d); S6; signal(h);
  - P7: wait(g); wait(h); S7;
- End



Synchronization

Monitors

Monitor

- A **high-level language** construct
- The representation of a monitor type consists of
  - Declaration of **variables** whose values define the state of an instance of the type
  - **Procedures/functions** that implement operations on the type
- The monitor type is similar to a **class in O.O language**
  - A procedure within a monitor can access only **local variable** and the **formal parameters**
  - The local variables of a monitor can be used only by the local procedures
- But, the monitor ensures that **only one process at a time can be active within the monitor**

Monitor Condition Variables

- To allow a process to **wait within** the monitor, a condition variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait()** and **signal()**

  x.wait();

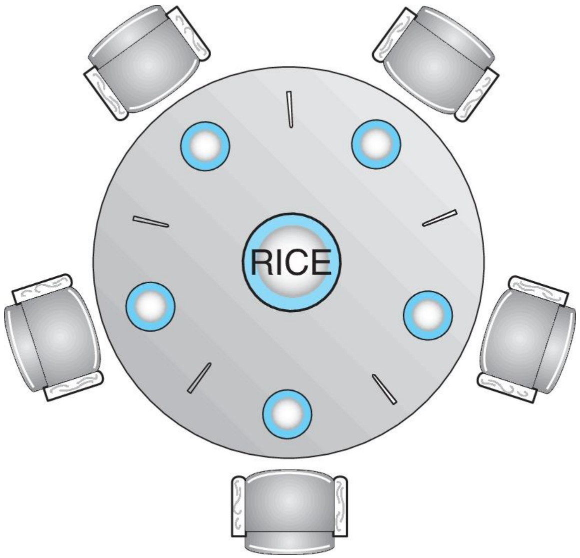
  means that the process invoking this operation is suspended until another process invokes it

  x.signal();

  resumes exactly one suspended process. If no suspended, then the signal operation **has no effects (in contrast, signal always change the state of a semaphore)**

Dining-Philosophers Problem

- **5 persons** sitting on 5 chairs with **5 chopsticks**
- A person is either thinking or eating
  - **thinking**: no interaction with the rest 4 persons
  - **eating**: need 2 chopsticks at hand
  - a person **picks up 1 chopstick at a time**
  - done eating: **put down both chopsticks**



Dining Philosophers Example

```
monitor dp {
    enum { thinking, hungry, eating } state[5];   // current state
    condition self[5];        // delay eating if can't obtain chopsticks

    void pickup(int i);       // pickup chopsticks
    void putdown(int i);      // putdown chopsticks
    void test(int i);         // try to eat

    void init() {
        for (int i = 0 ; i < 5 ;  i++)
            state[i] = thinking;
    }
}
```

Deadlocks

Overview

Deadlock Problem

- A set of blocked processes each **holding** some resources and **waiting** to acquire a resource held by another process in the set

- Example:
  - 2 processes and semaphores A and B
    - P_1 (hold B, wait A): **wait (A)**, signal (B)
    - P_2 (hold A, wait B): **wait (B)**, signal (A)

- Example:
  - Dining philosophers’ problem

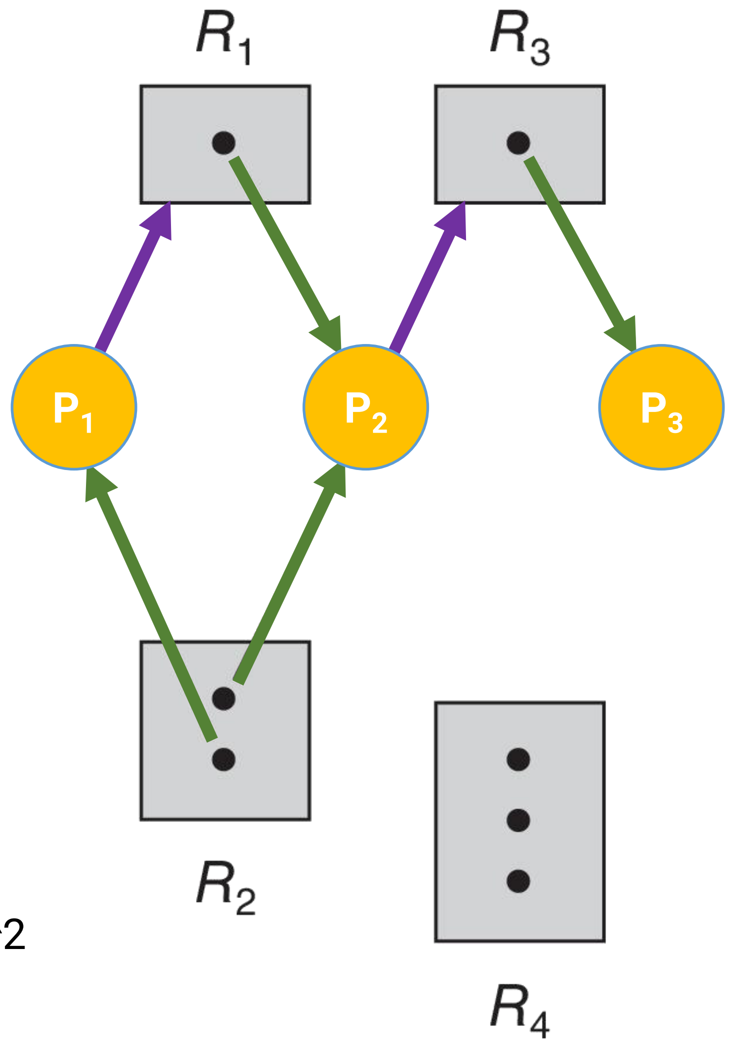
Necessary Conditions

- **Mutual exclusion**
  - Only 1 process at a time can use a resource
- **Hold and wait**
  - A process holding some resources and is waiting for another resource
- **No preemption**
  - A resource can be only released by a process **voluntarily**
- **Circular wait**
  - There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that $P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n \rightarrow P_0$

Resource-Allocation Graph

- 3 processes, $P_1 \sim P_3$
- 4 resources, $R_1 \sim R_4$
  - R_1 and R_3 each has one instance
  - R_2 has two instances
  - R_4 has three instances
- **Request edges**
  - $P_1 \rightarrow R_1$: P_1 requests R_1
- **Assignment edges**
  - $R_2 \rightarrow P_1$: one instance of R_2 is allocated to P_1

➔ P_1 is **holding on** an instance of R_2 and **waiting for** an instance or R_1



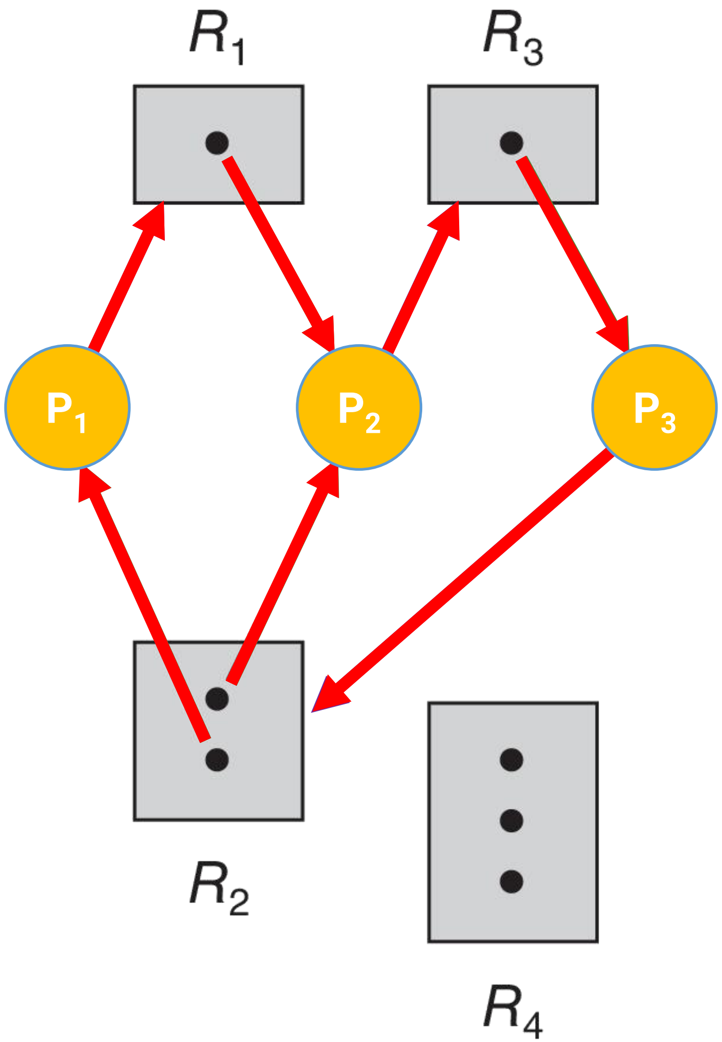
Resource-Allocation Graph w/ Deadlock

- If the graph contains a **cycle**, a deadlock **may** exist
- In the example
  - P_1 is waiting for P_2
  - P_2 is waiting for P_3

  ➔ P_1 is also waiting for P_3

  - Since P_3 is waiting for P_1 or P_2, and they both waiting for P_3

  ➔ **Deadlock !**

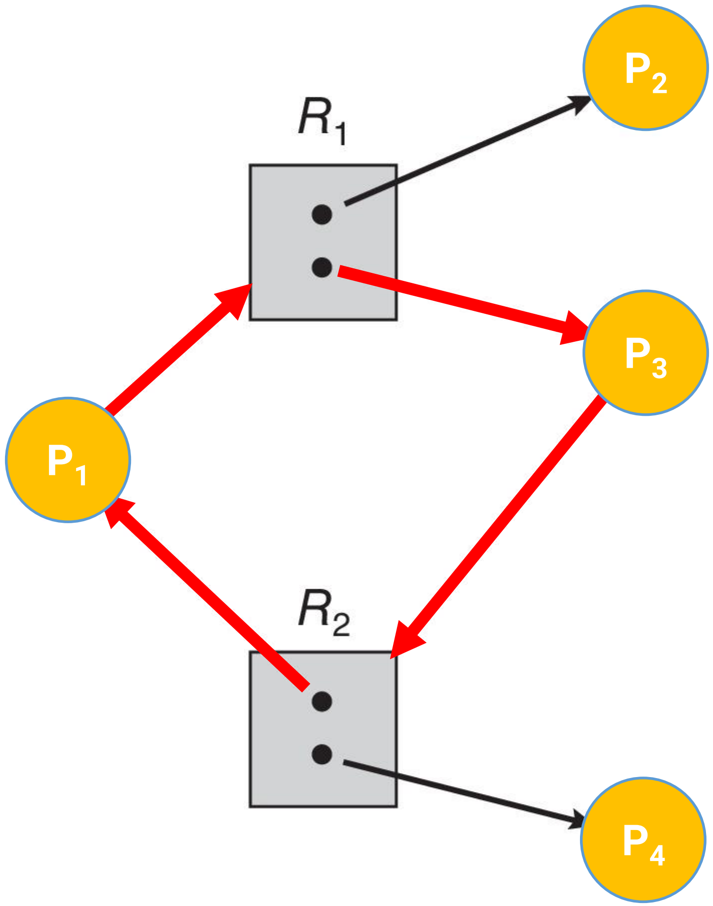


RA Graph w/ Cycle but NO Deadlock

- If the graph contains a **cycle**, a deadlock **may** exist
- In the example
  - P_1 is waiting for P_2 or P_3
  - P_3 is waiting for P_1 or P_4

  - Since P_2 and P_4 wait for no one

➔ **No Deadlock between P_1 and P_3**



Handling Deadlocks

- Ensure the system will **never** enter a deadlock state
  - **Deadlock prevention**: ensure that at least one of the **4 necessary conditions** cannot hold
  - **Deadlock avoidance**: **dynamically** examines the resource-allocation state before allocation
- Allow to **enter a deadlock state** and then **recover**
  - **Deadlock detection**
  - **Deadlock recovery**
- **Ignore the problem** and pretend that deadlocks never occur in the system
  - **Used by most operating systems, including UNIX**

Deadlocks

Prevention

Deadlock Prevention

- **Mutual exclusion (ME):** do not require ME on sharable resources
  - E.g. there is no need to ensure ME on read-only files
  - However, some resources are not shareable (e.g. printer)

- **Hold and wait:**
  - When a process requests a resource, it does not hold any resource
  - Pre-allocate all resources before executing
  - **Resource utilization is low; starvation is possible**

Deadlock Prevention (cont.)

- **No preemption:**
  - When a process is waiting on a resource, all its holding resources are preempted
    - E.g. P_1 request R_1, which is allocated to P_2, which in turn is waiting on R_2 ($P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2$)
    - **R1 can be preempted and reallocated to P1**
  - **Applied to resources whose states can be easily saved and restored later**
    - E.g. CPU registers and memory
  - It cannot easily be applied to other resources
    - E.g. printers and tape drives

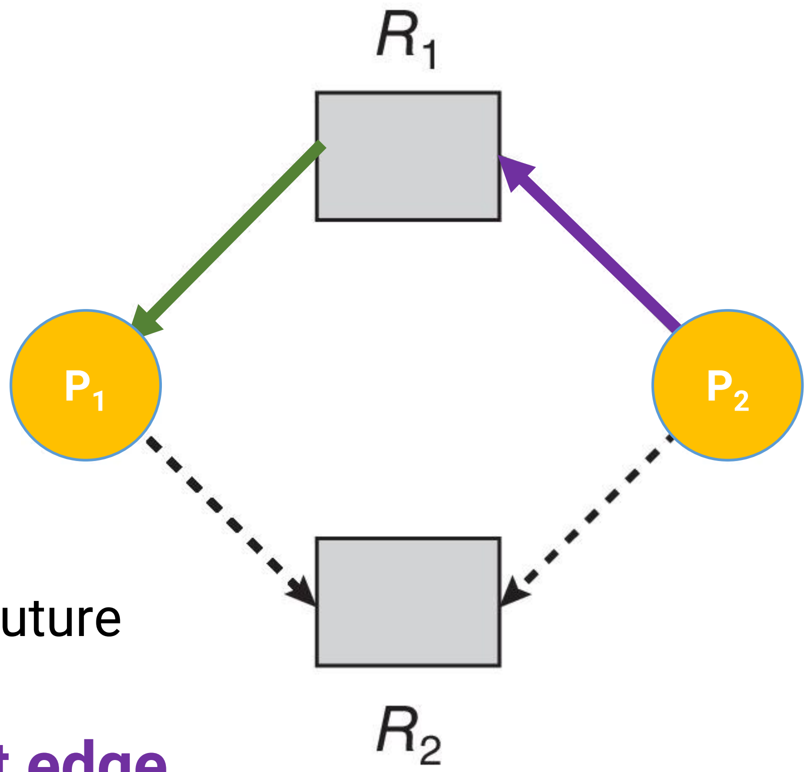
Deadlock Prevention (cont.)

- **Circular wait:**
  - Impose a **total ordering** of all resource types
  - A process requests resources in an increasing order
    - Let R = {R_0, R_1, … R_n} be the set of resource types
    - **When request R_k, should release all R_i, $i \geq k$**
- Example
  - F (disk drive) = 5, F(printer) = 12
  - A process must request disk drive before printer
- Proof: counter-example does not exist
  - $P_0(R_0) \rightarrow R_1, P_1(R_1) \rightarrow R_2, \ldots, P_n(R_n) \rightarrow R_0$ ← P_n **holds on** R_n, **waiting for** R_0
  - Conflict: $R_0 < R_1 < R_2 < \ldots R_n < R_0$

Deadlocks
Avoidance

Resource-Allocation Graph Algorithm

- **Request edges**
  - $P_i \rightarrow R_j$: P_i **is waiting** for resource R_j
- **Assignment edges**
  - $R_j \rightarrow P_i$: Resource R_j **is allocated** and held by P_i

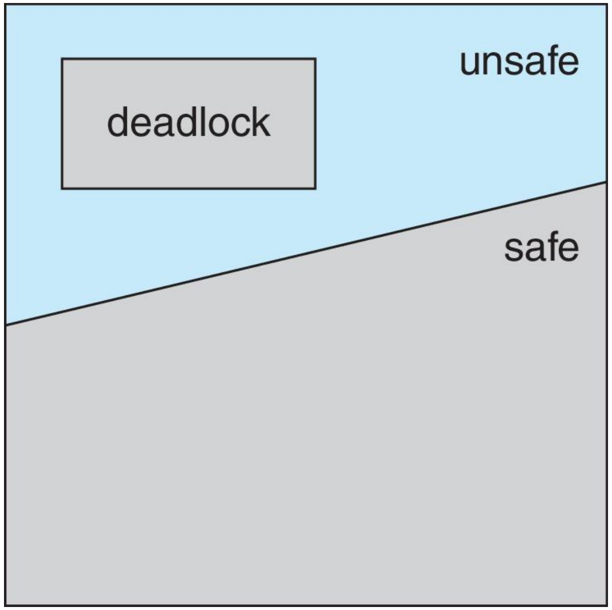


- **Claim edge**
  - Process P_i **may** request R_j in the future
- **Claim edge** converts to **request edge**
  - When a resource **is requested** by process
- **Assignment edge** converts back to a **claim edge**
  - When a resource is released by a process

Safe / Unsafe State

- **Safe state:** a system is in a safe state if there exists **a sequence of allocations** to satisfy requests by all processes
  - This sequence of allocations is called **safe sequence**
- **Safe state ➔ no deadlock**
- **Unsafe state ➔ possibility of deadlock**
- **Deadlock avoidance ➔**

  ensure that a system never enters an unsafe state



Banker's Algorithm

- Use for **multiple instances** of each resource type

- **Banker's Algorithm**
  - Use a general safety algorithm to **pre-determine** if any **safe sequence** exists after allocation
  - **Only proceed the allocation if safe sequence exists**
- **Safety algorithm**
  1. Assume processes need **maximum** resources
  2. Find a process that can be satisfied by free resources
  3. Free the resource usage of the process
  4. Repeat to step 2 until all processes are satisfied

Banker's Algorithm Example

- Total instances: A: 10, B: 5, C: 7
- Available instances: A: 3, B: 3, C: 2

	Max			Allocation			Need (Max – Alloc.)		
	A	B	C	A	B	C	A	B	C
P_0	7	5	3	0	1	0	7	4	3
P_1	3	2	2	2	0	0	1	2	2
P_2	9	0	2	3	0	2	6	0	0
P_3	2	2	2	2	1	1	0	1	1
P_4	4	3	3	0	0	2	4	3	1

- Safe sequence ?

Deadlocks

Detection

Multiple Instance for Each Resource Type

- Total instances: A: 7, B: 2, C: 6
- Available instances: A: 0, B: 0, C: 0

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	0
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

- The system is in a safe state ➔ $<P_0, P_2, P_3, P_1, P_4>$
  - ➔ No deadlock

Multiple Instance for Each Resource Type

- Total instances: A: 7, B: 2, C: 6
- Available instances: A: 0, B: 0, C: 0

	Allocation			Request		
	A	B	C	A	B	C
P_0	0	1	0	0	0	0
P_1	2	0	0	2	0	2
P_2	3	0	3	0	0	**1**
P_3	2	1	1	1	0	0
P_4	0	0	2	0	0	2

- If P_2 requests (0, 0, 1) ➔ no safe sequence can be found

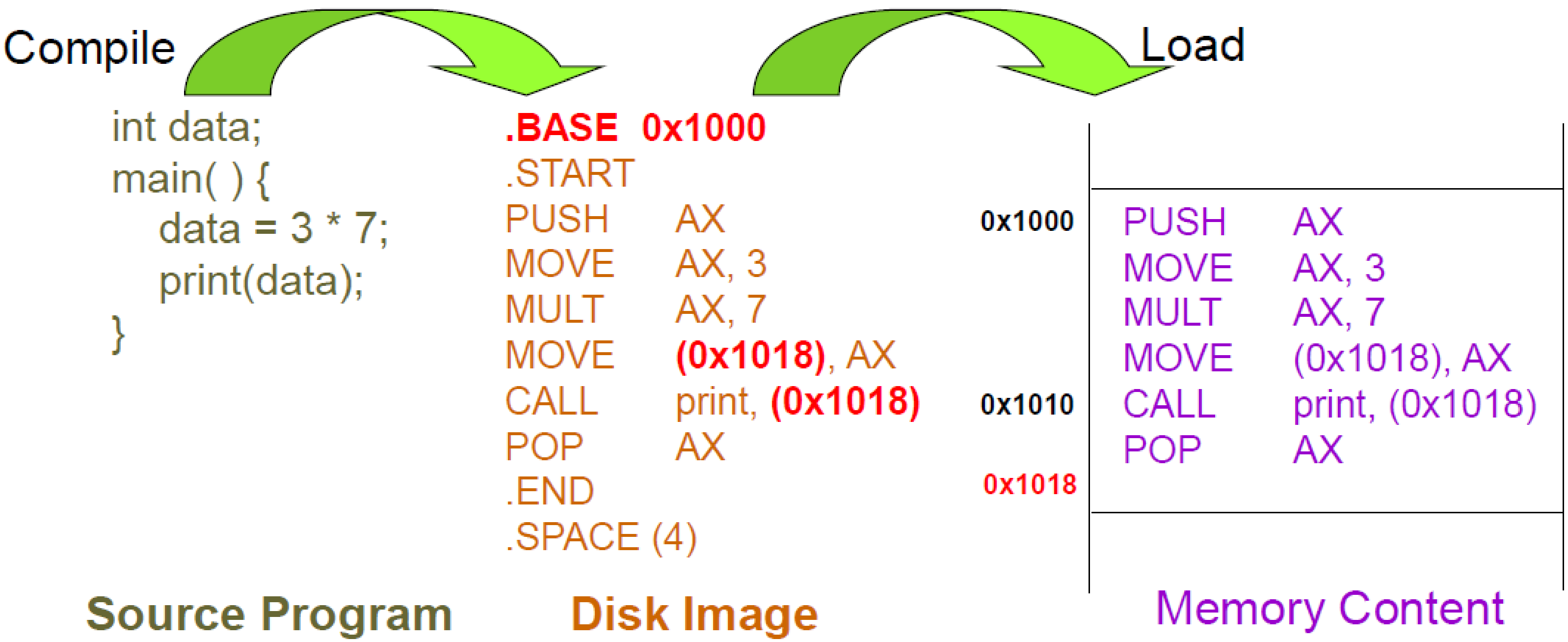
  ➔ The system is deadlocked

Main Memory

Address Binding

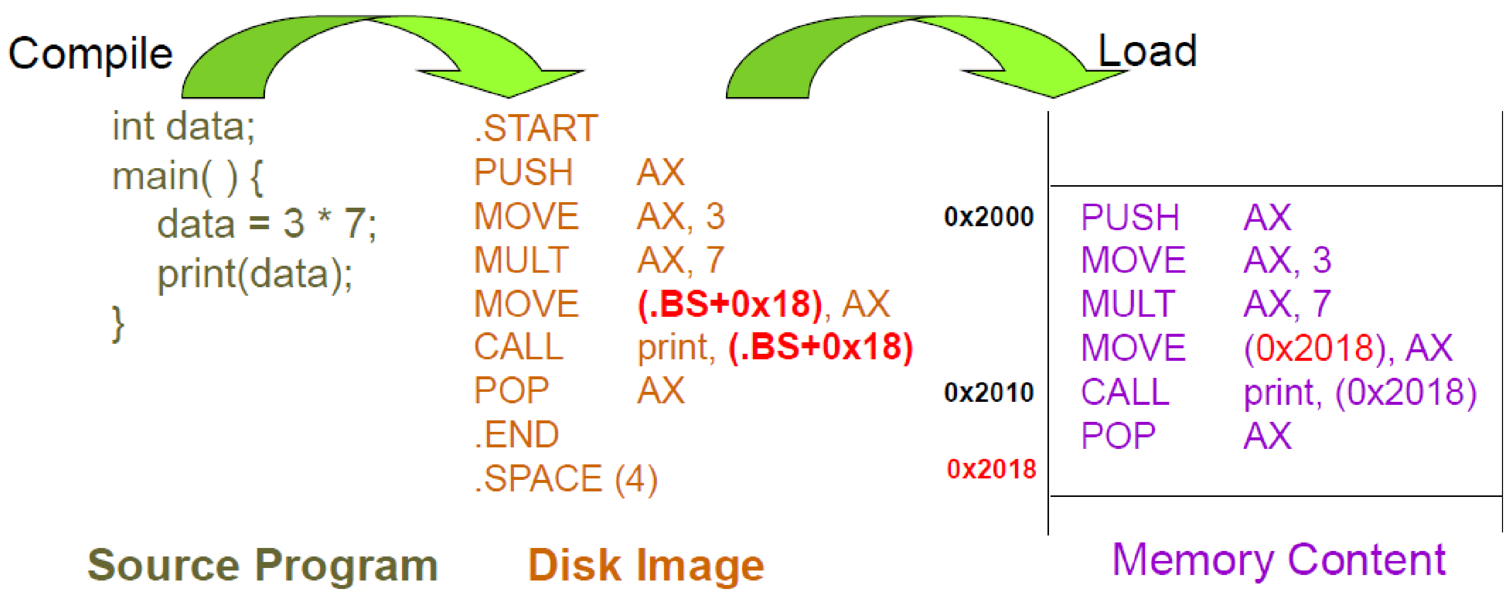
Address Binding: Compile Time

- Program is written as symbolic code
- Compiler translates symbolic code into **absolute code**
- If starting location changes ➔ **recompile**



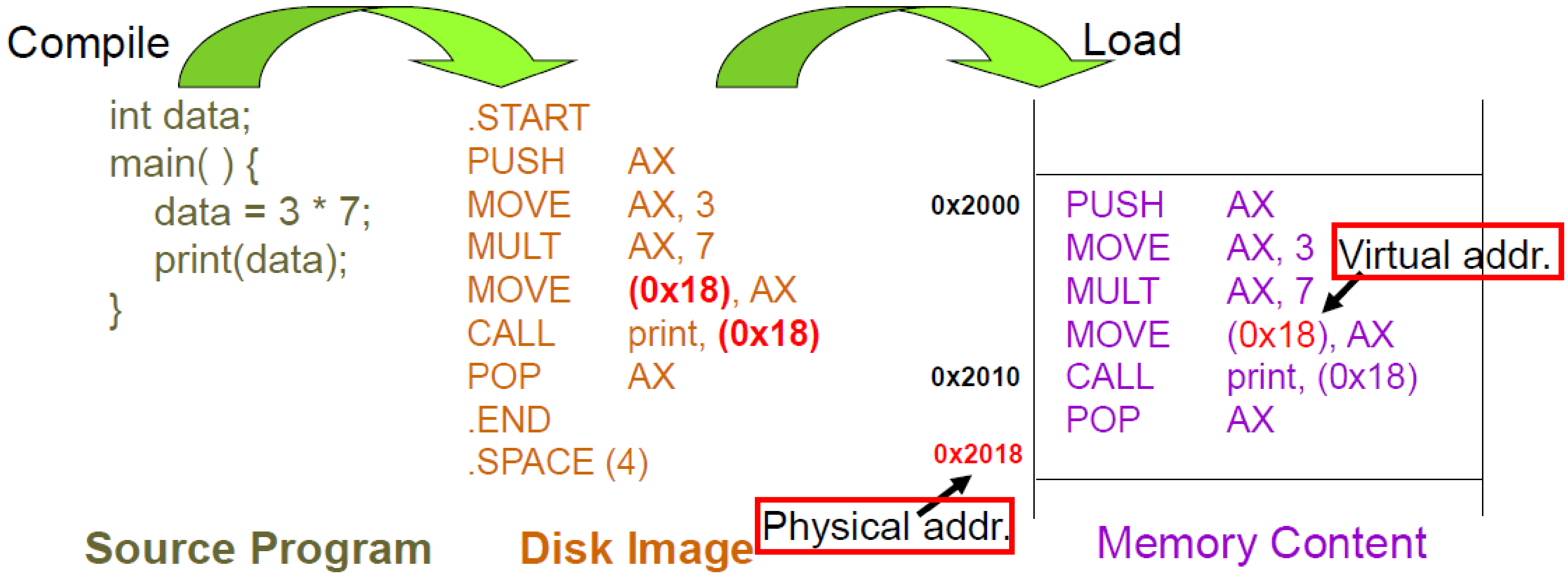
Address Binding: Load Time

- Compiler translates symbolic code into **relocatable code**
- **Relocatable code**
  - Machine language that can be run from any memory location
  - If starting location changes ➔ **reload the code**



Address Binding: Execution Time

- Compiler translates symbolic code into **logical-address (i.e. virtual-address) code**
- **Special hardware (i.e. MMU)** is needed for this scheme
- Most general-purpose OS use this method



Logical v.s. Physical Address

- **Logical address** – generated by CPU
  - a.k.a virtual address
- **Physical address** – seen by the memory module

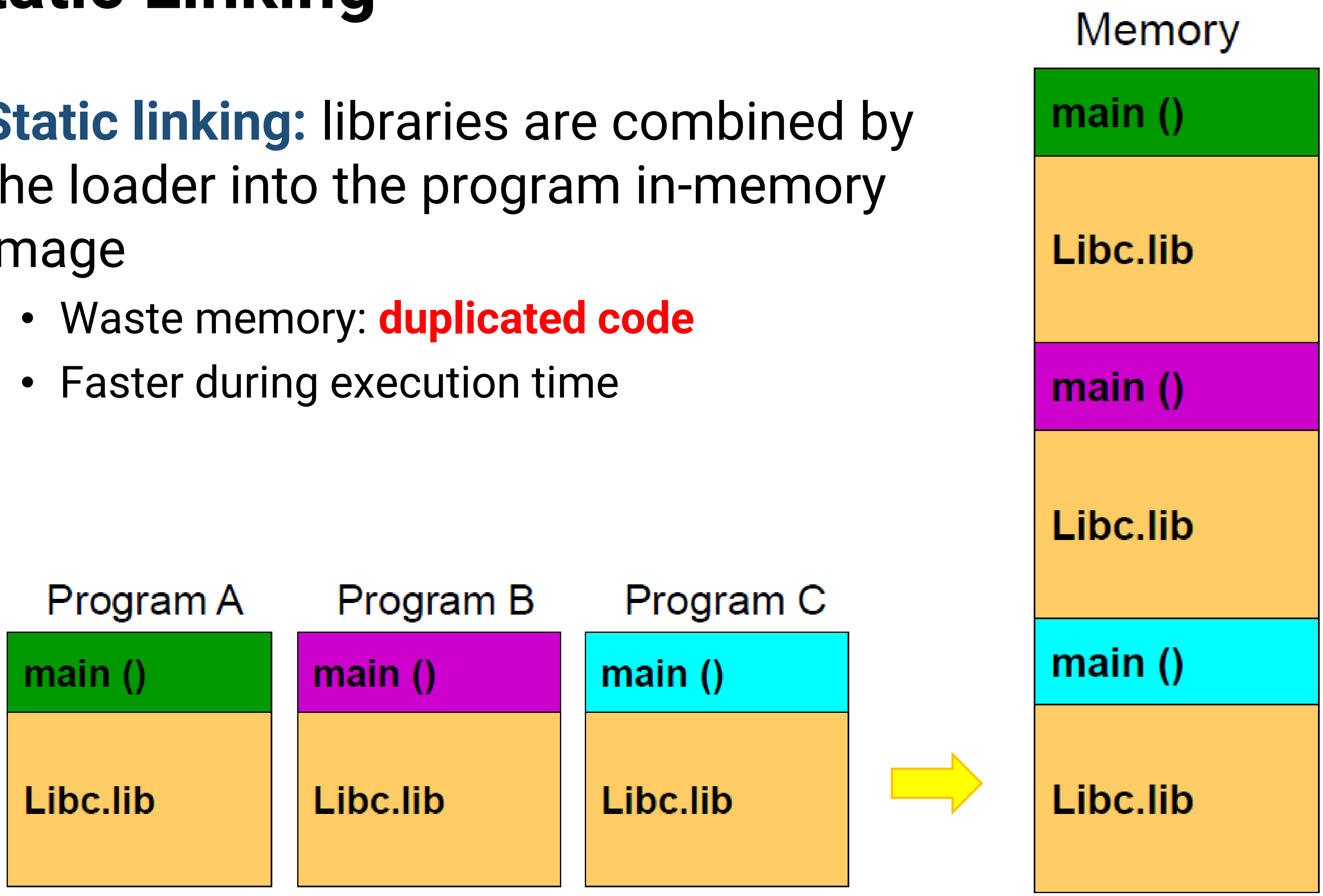
- **Compile-time** and **load-time** address binding
  - Logical address = physical address
- **Execution-time** address binding
  - Logical address ≠ physical address

Main Memory

Static / Dynamic Linking

Static Linking

- **Static linking:** libraries are combined by the loader into the program in-memory image
  - Waste memory: **duplicated code**
  - Faster during execution time



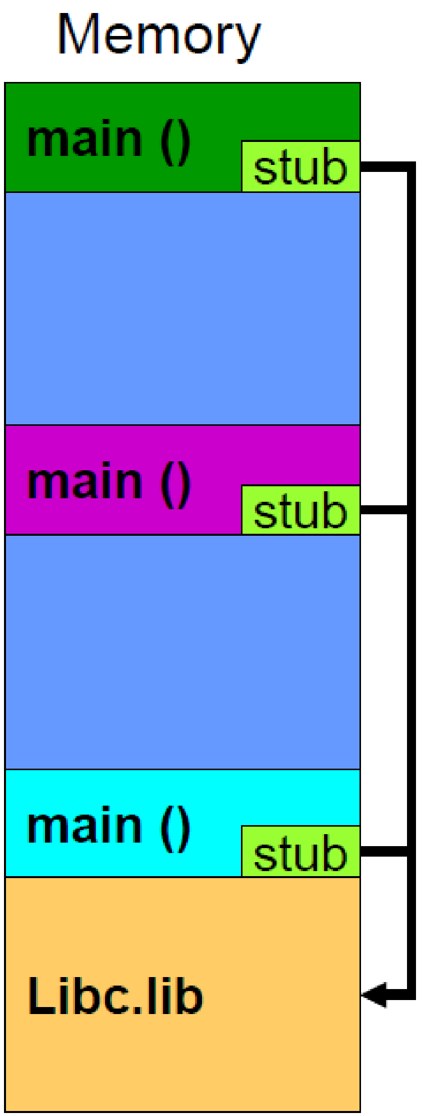
Dynamic Linking

- **Dynamic linking:** linking postponed **until execution time**
  - **Only one code copy** in memory and shared by everyone
  - **A stub** is included in the program in-memory image for each lib reference
  - **Stub call**

    ➔ check if the referred lib is in memory

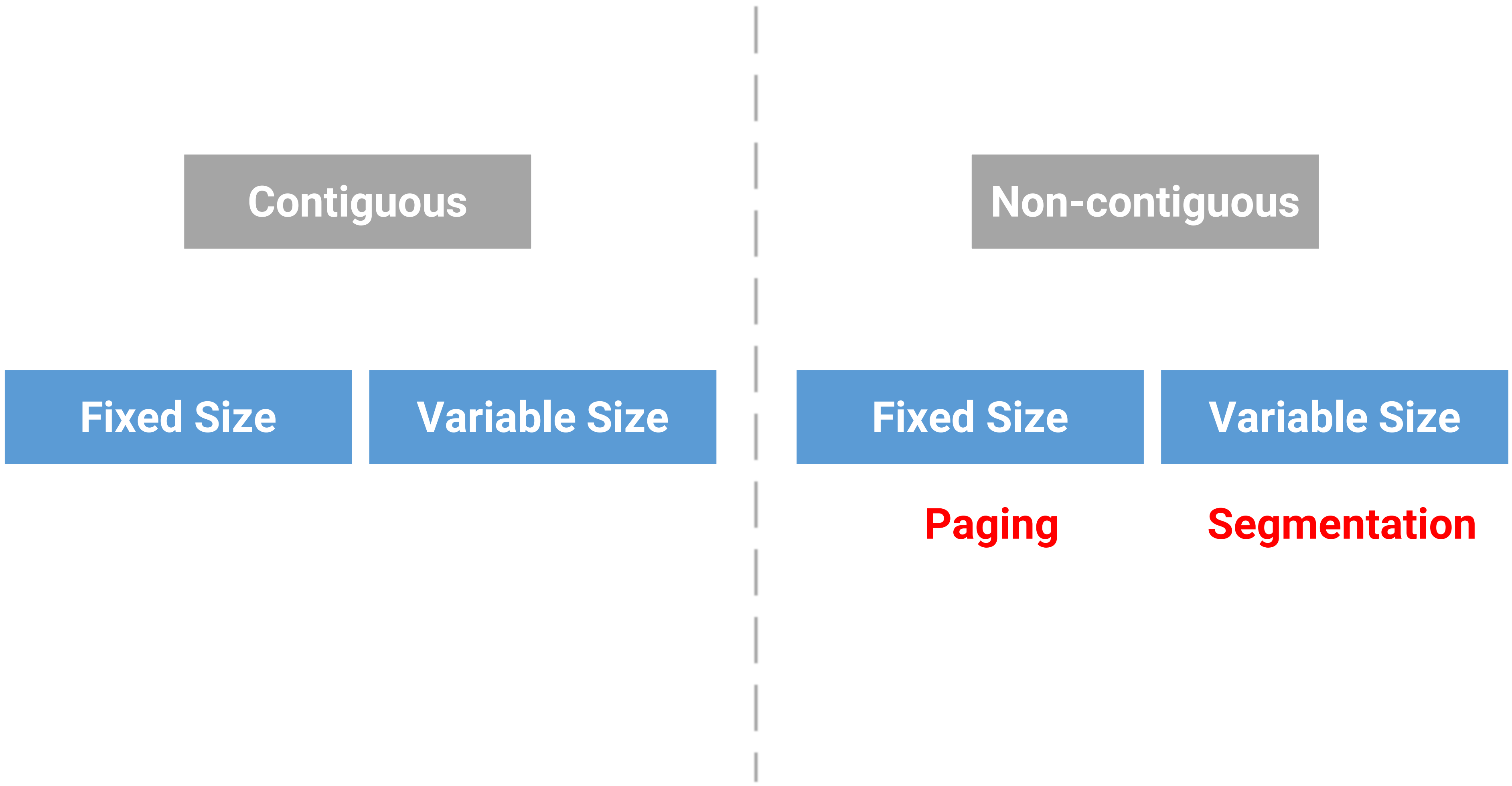
    ➔ if not, load the lib

    ➔ execute the lib



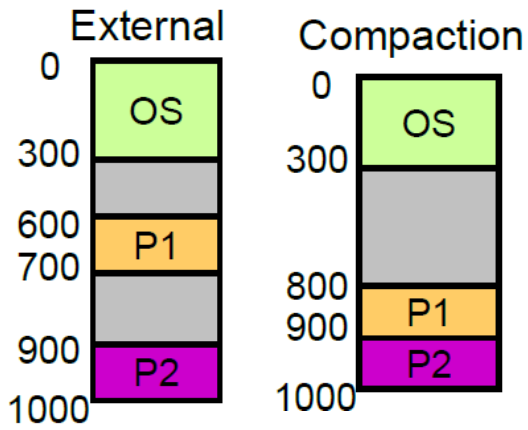
Main Memory
Contiguous Allocation

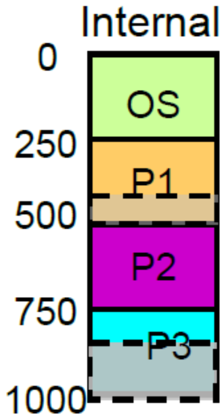
Classification of Memory Allocation



Fragmentation

- **External fragmentation**
  - Total free memory space is big enough to satisfy a request but is not contiguous
  - Occur in **variable-size allocation**
  - Solution: **compaction**
    - Shuffle the memory contents to place all free memory together in one large block at execution time
    - Only if the binding is done at execution time
- **Internal fragmentation**
  - Memory that is internal to a partition but is not being used
  - Occur in **fixed-partition allocation**





Main Memory

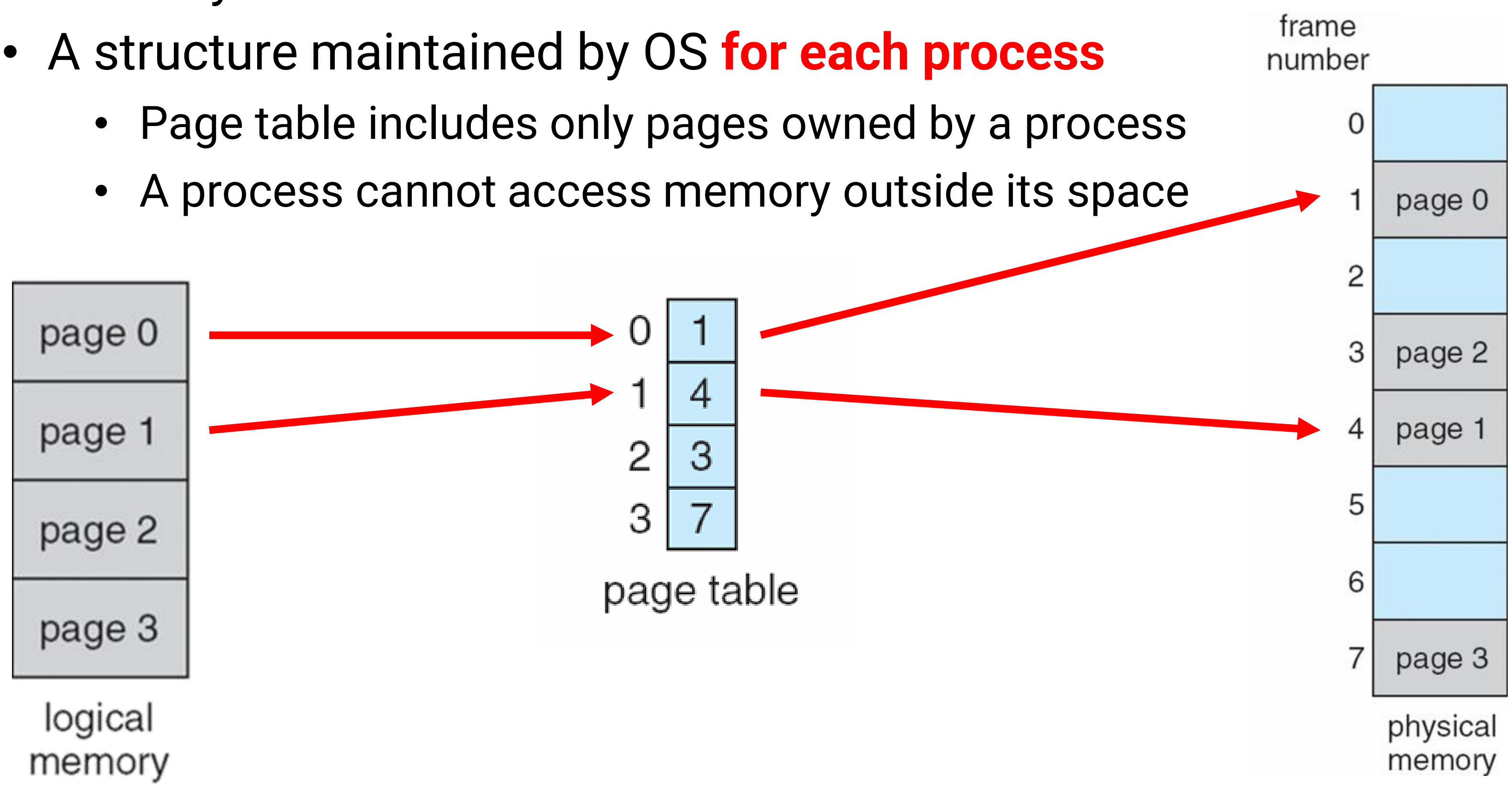
Non-Contiguous Allocation: Paging

Paging Concept

- Method
  - Divide **physical memory** into fixed-size blocks called **frames**
  - Divide **logical address** space into blocks of the **same size** called **pages**
  - To run a program of *n* pages, need to find *n* free frames and load the program
  - **Must keep track of free frames**
  - Set up a **page table** to translate logical to physical addresses
- Benefit
  - Allow the physical-address space of a process to be **noncontiguous**
  - Avoid external fragmentation
  - Limited internal fragmentation
  - Provide **shared memory / pages**

Paging Example

- **Page table**
  - Each entry maps to the **base address of a page** in physical memory
  - A structure maintained by OS **for each process**
    - Page table includes only pages owned by a process
    - A process cannot access memory outside its space



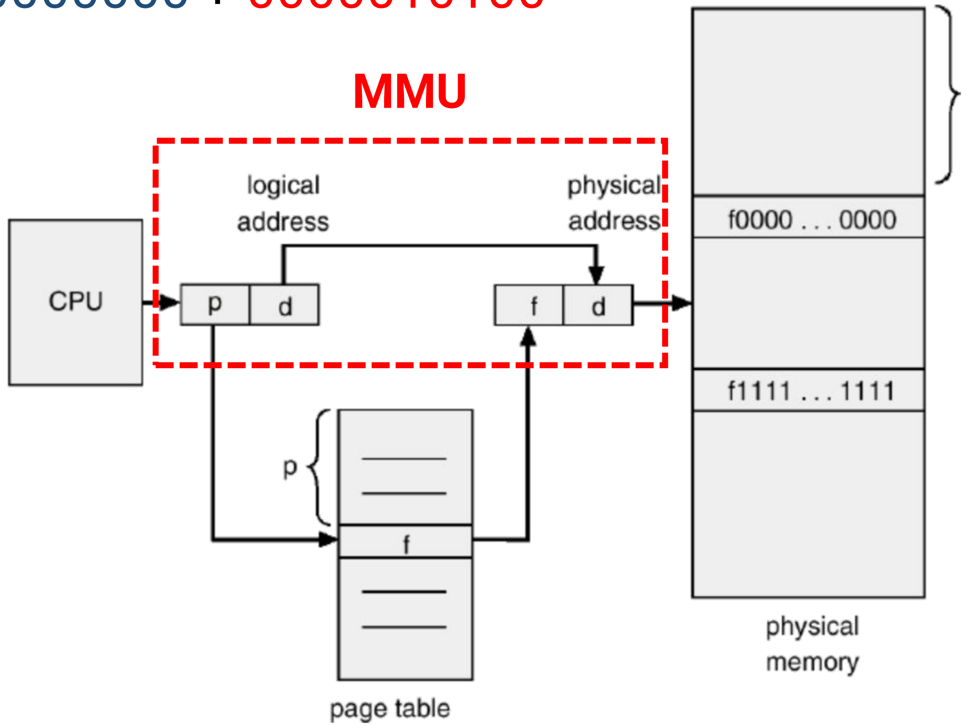
Address Translation Scheme

- Logical address is divided into two parts
  - **Page number (p)**
    - Used as an **index into a page table** which contains **base address** of each page in physical memory
    - *N* bits means a process can allocate at most 2^N pages

      ➔ ***2^N* x page size memory size**
  - **Page offset (d)**
    - Combined with base address to define the physical memory address that is sent to the memory unit
    - ***N* bits means the page size is *2^N***
- **Physical address** = **page base address** + **page offset**

Address Translation Architecture

- If page size is 1KB (2^10) and page 3 maps to frame 5
- Given 13 bits logical address (p = 3, d = 20), what is the physical address?
  - 5 * (1KB) + 20 = 1010000000000 + 0000010100 = 1010000010100



Address Translation

- Total number of pages does not need to be the same as the total number of frames
  - **Total # pages** determines the logical memory size of a process
  - **Total # frames** depending on the size of physical memory
- E.g.: Given 32 bits logical address, 36 bits physical address, and 4KB page size, what does it mean?
  - Number of bits for page offset: 4KB page size = 2^{12} bytes ➔12
  - Number of bits for page number: 2^{20} pages ➔ 20 bits
  - Page table size: $2^{32} / 2^{12} = 2^{20}$ entries
  - Max program memory: 2^{32} = 4GB
  - Number of bits for frame number: 2^{24} frames ➔ 24 bits
  - Total physical memory size: 2^{36} = 64GB

Page / Frame Size

- The page (frame) size is defined by hardware
  - **Typically, a power of 2**
  - Ranging from 512 bytes to 16 MB / page
  - 4KB / 8KB page is commonly used

- Internal fragmentation?
  - Larger page size ➔ More space waste

- But **page sizes cannot be too small**
  - Memory, process, and data sets have become larger
  - Need to keep page table small
  - Fewer access means better I/O performance

Implementation of Page Table

- Page table is kept **in memory**
- **Page-table base register (PTBR)**
  - The **physical memory address** of the page table
  - The PTBR value is stored in **PCB** (Process Control Block)
  - Changing the value of PTBR during the context switch
- With PTBR, each memory reference results in **2 memory reads**
  - One for the page table and one for the real address
- The 2-access problem can be solved by
  - **Translate Look-aside Buffers (TLB)** (HW) which is implemented by **Associative memory** (HW)

Effective Memory-Access Time

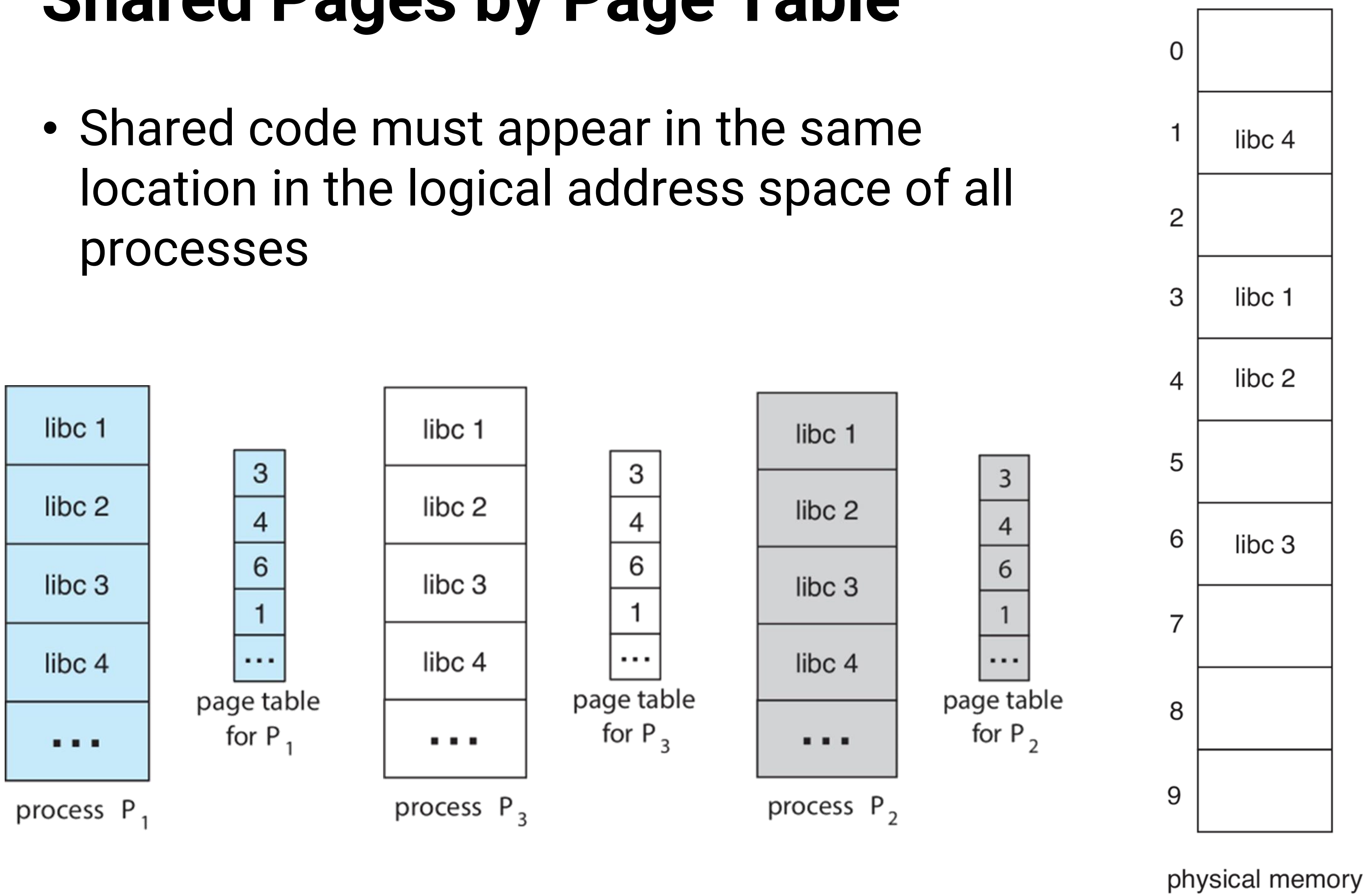
- 20 ns for TLB search
- 100 ns for memory access
- Effective Memory-Access Time (**EMAT**)
    - 70% TLB hit-ratio:

    ➔ EMAT = 0.70 x (**20 + 100**) + (1 – 0.70) * (**20 + 100 + 100**) = 150 ns
    - 98% TLB hit-ratio:

    ➔ EMAT = 0.98 x 120 + 0.02 x 220 = 122 ns

Shared Pages by Page Table

- Shared code must appear in the same location in the logical address space of all processes



Page Table Memory Structure

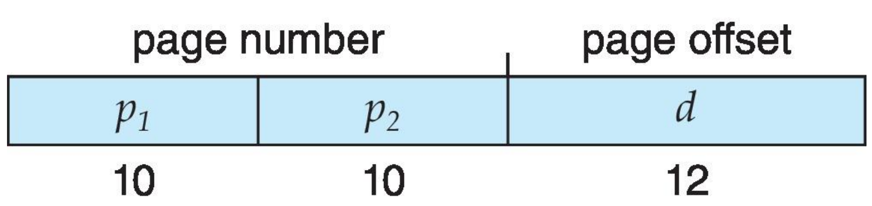
- Page table could be huge and difficult to be loaded
  - 4GB (2^{32}) logical address space with 4KB (2^{12}) page

    ➔ 1 million (2^{20}) page table entry
  - Assume each entry need 4 bytes (32 bits)

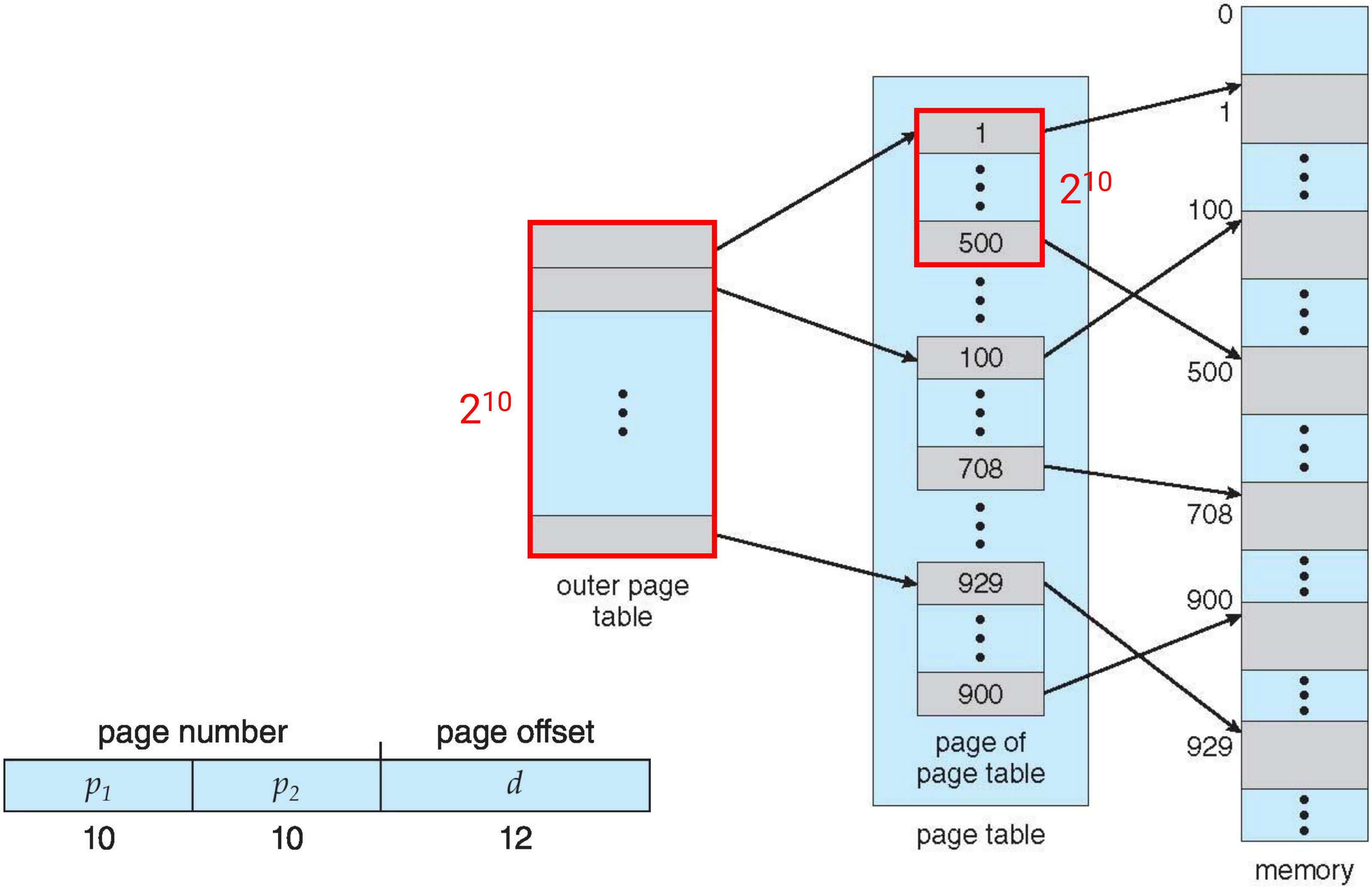
    ➔ Total size = 4MB
  - Need to break it into several smaller page tables, better within a single page size (i.e. 4KB)
  - Or reduce the total size of page table
- Solutions
  - Hierarchical paging
  - Hash page table
  - Inverted page table

Hierarchical Paging

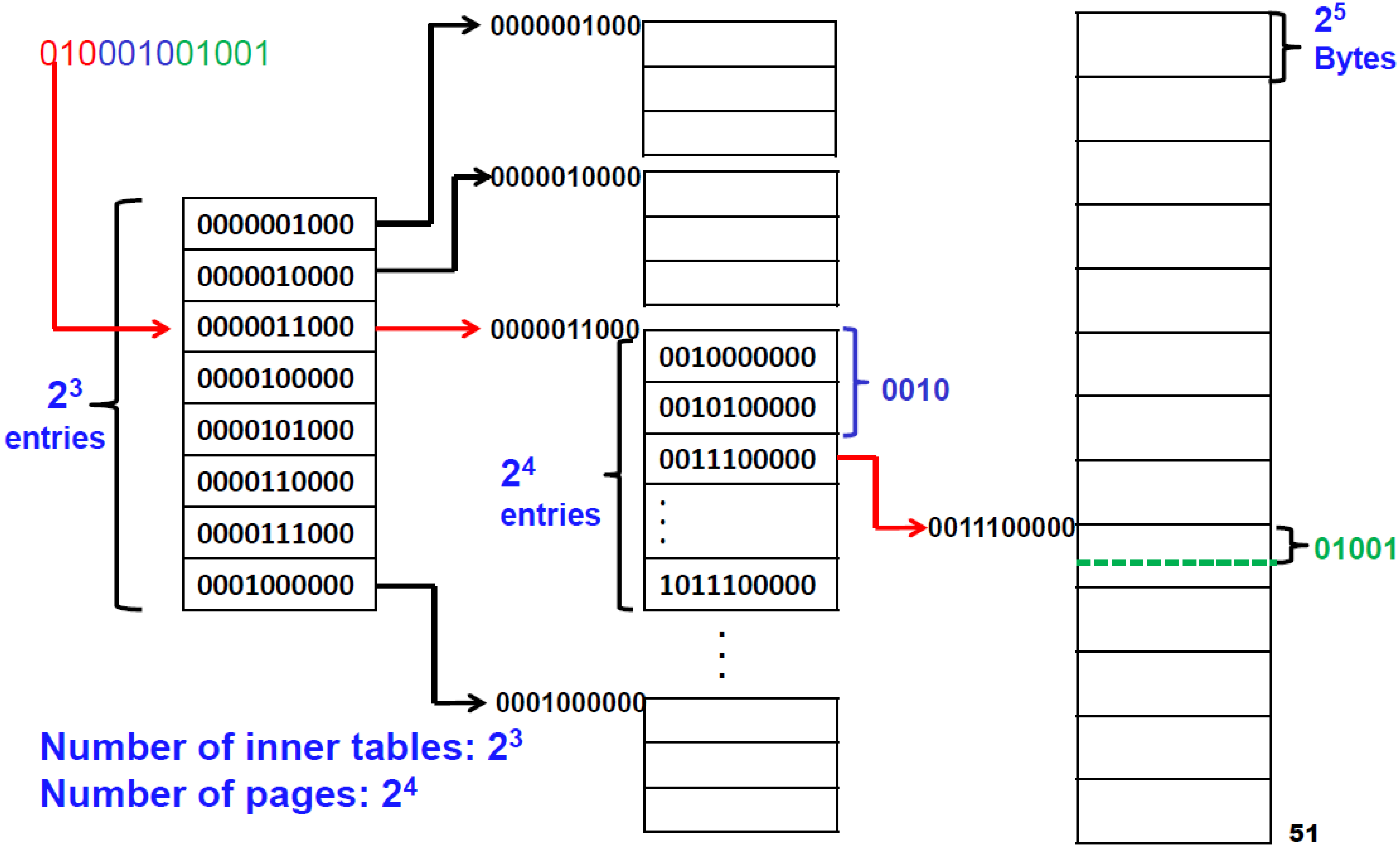
- Break up the logical address space into **multiple page tables**
  - Paged the page table
  - i.e. *n*-level page table
- Two-level paging (32-bit address with 4KB (2^{12}) page size)
  - 12-bit offset (d) ➔ 4KB (2^{12}) page size
  - 10-bit **outer** page number ➔ 1K (2^{10}) page table entries
  - 10-bit **inner** page number ➔ 1K (2^{10}) page table entries
  - 3 memory accesses



Two-Level Page Table Example



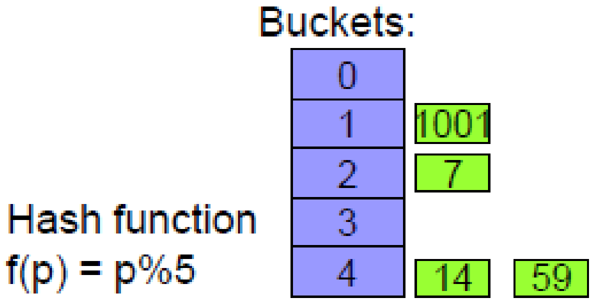
Two-Level Address Translation Example



Number of inner tables: 2^3
Number of pages: 2^4

Hashed Page Table

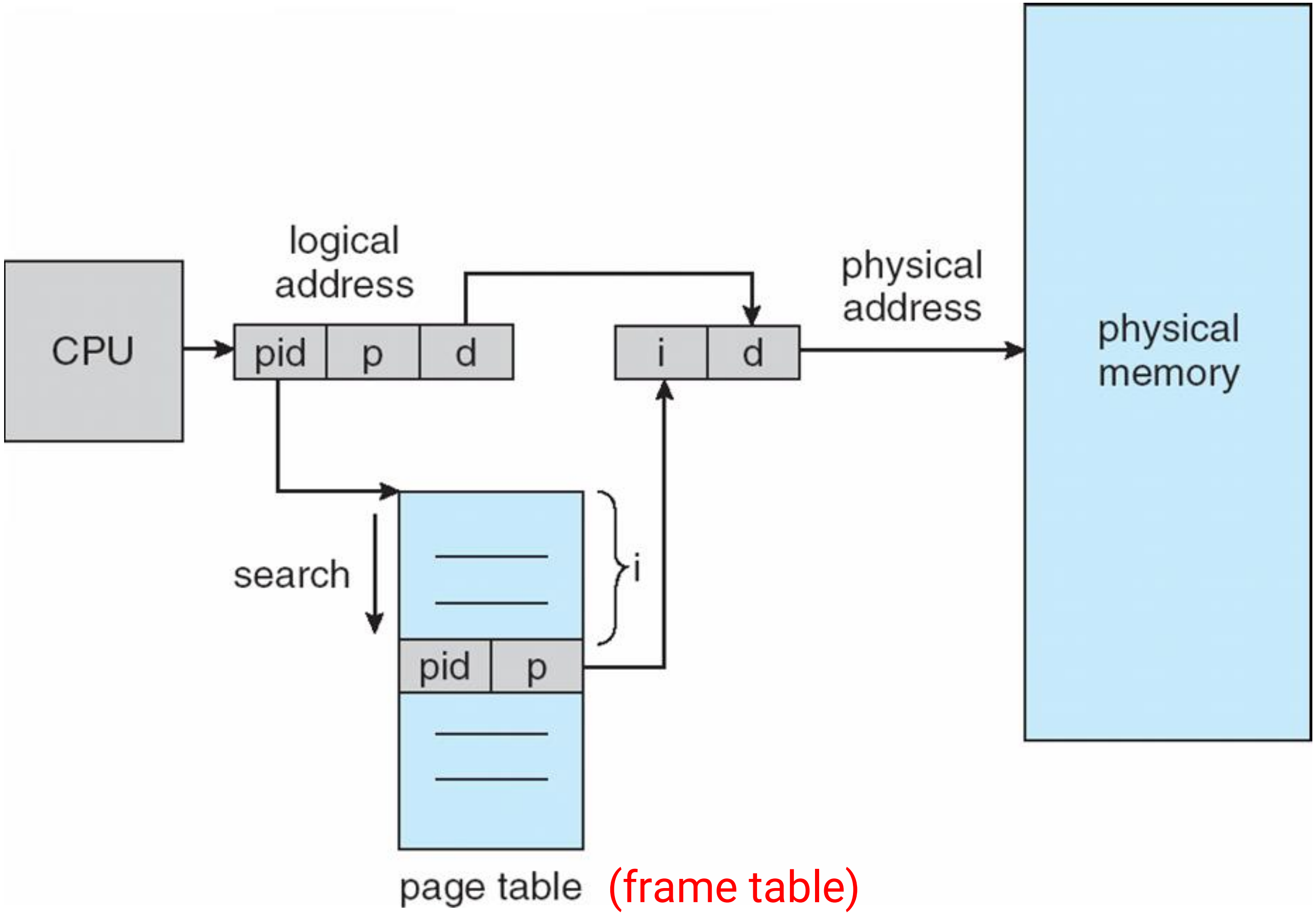
- **Commonly-used for address > 32 bits**
- Virtual page number is hashed into a hash table
- The size of the hash table varies
  - Larger hash table ➔ smaller chains in each entry
- Each entry in the hashed table contains
  - (Virtual Page Number, Frame Number, Next Pointer)
  - Pointers waste memory
  - Traverse linked list waste time and cause additional memory references



Inverted Page Table (Frame Table)

- Maintains **no** page table for each process
- Maintains a **frame table** for the whole memory
  - One entry for each real frame of memory
- Each entry in the hashed table contains
  - (PID, Page Number)
- Eliminate the memory needed for page tables but **increase memory access time**
  - Each access needs to search the whole frame table
  - Solution: use hashing for the frame table
- **Hard to support shared page / memory**

Inverted Page Table Address Translation



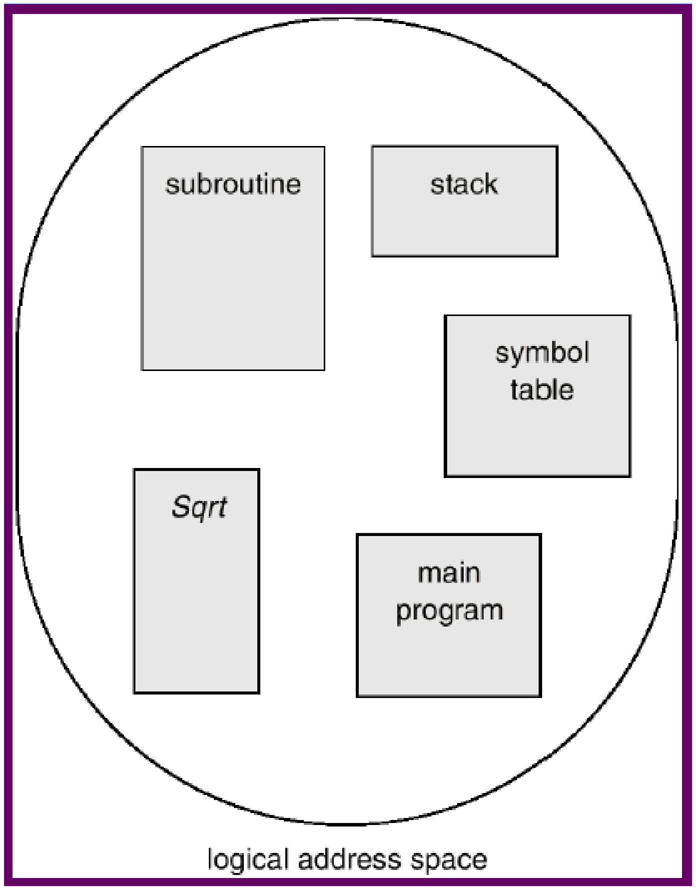
(frame table)

Main Memory

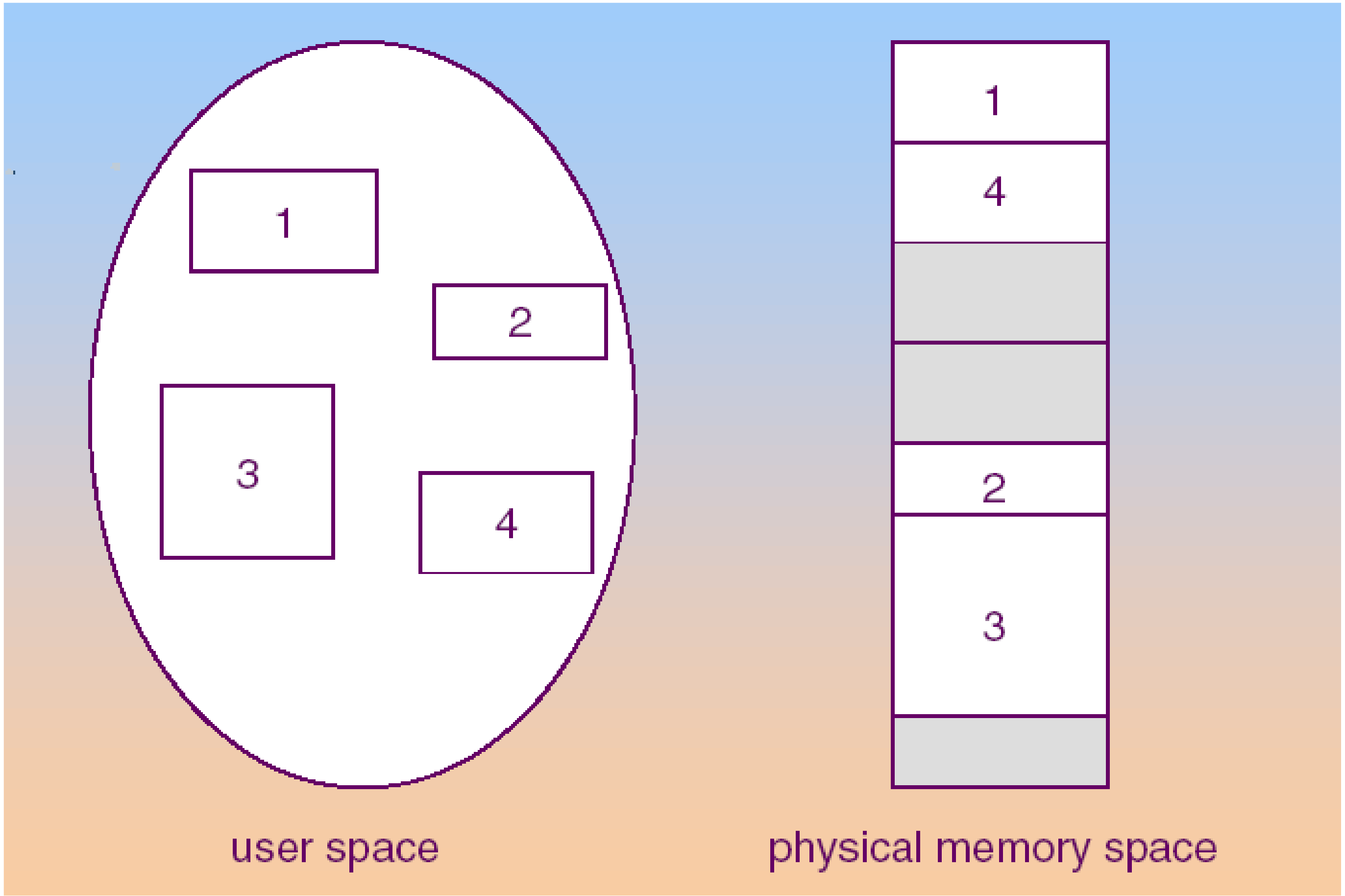
Non-Contiguous Allocation: Segmentation

Segmentation

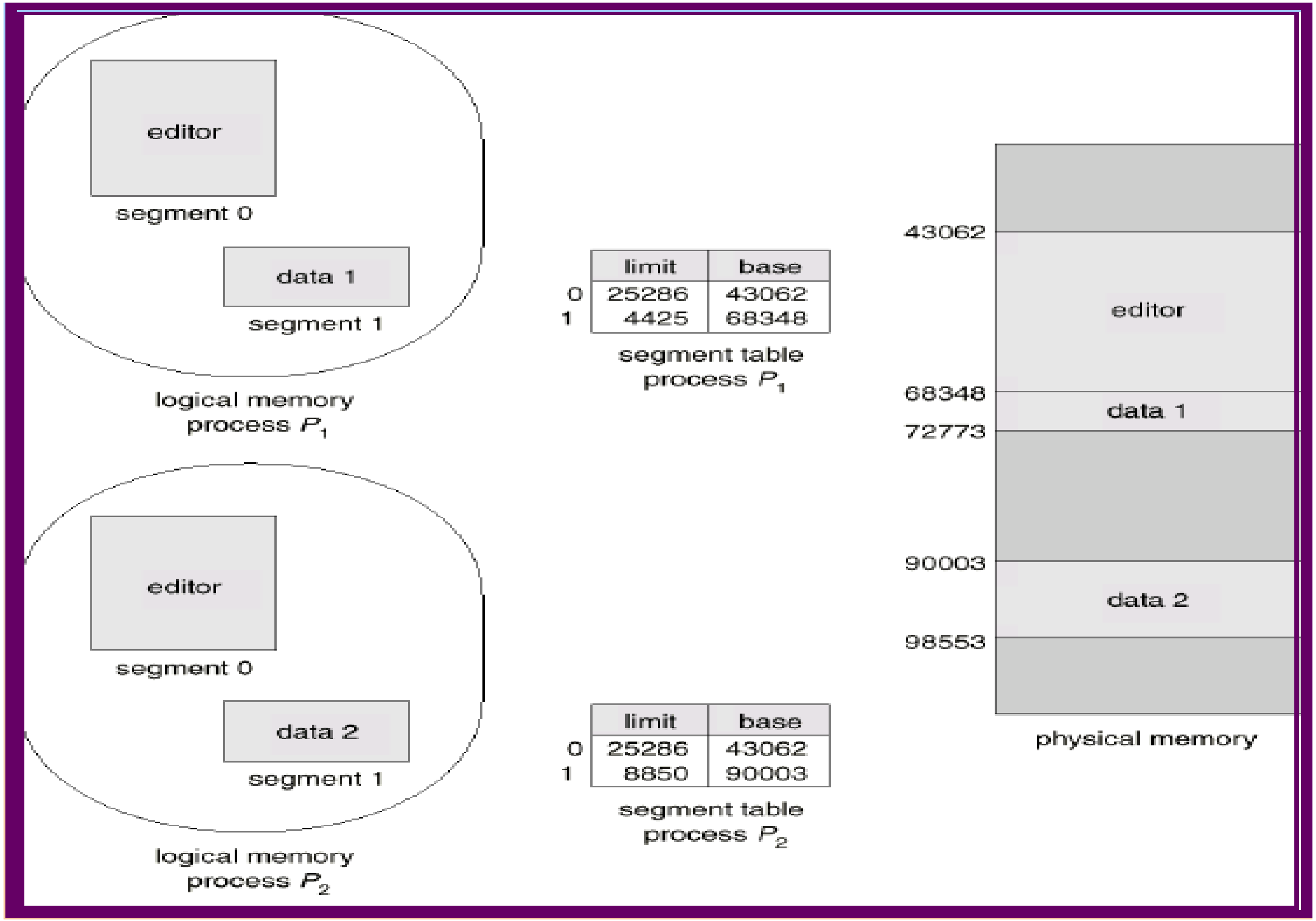
- Memory-management scheme that supports **user view of memory**
- A program is a collection of segments
- A segment is a logical unit such as
  - Main program
  - Function
  - Object
  - Local/global variables
  - Stack
  - Symbol table
  - Arrays



Logical View of Segmentation

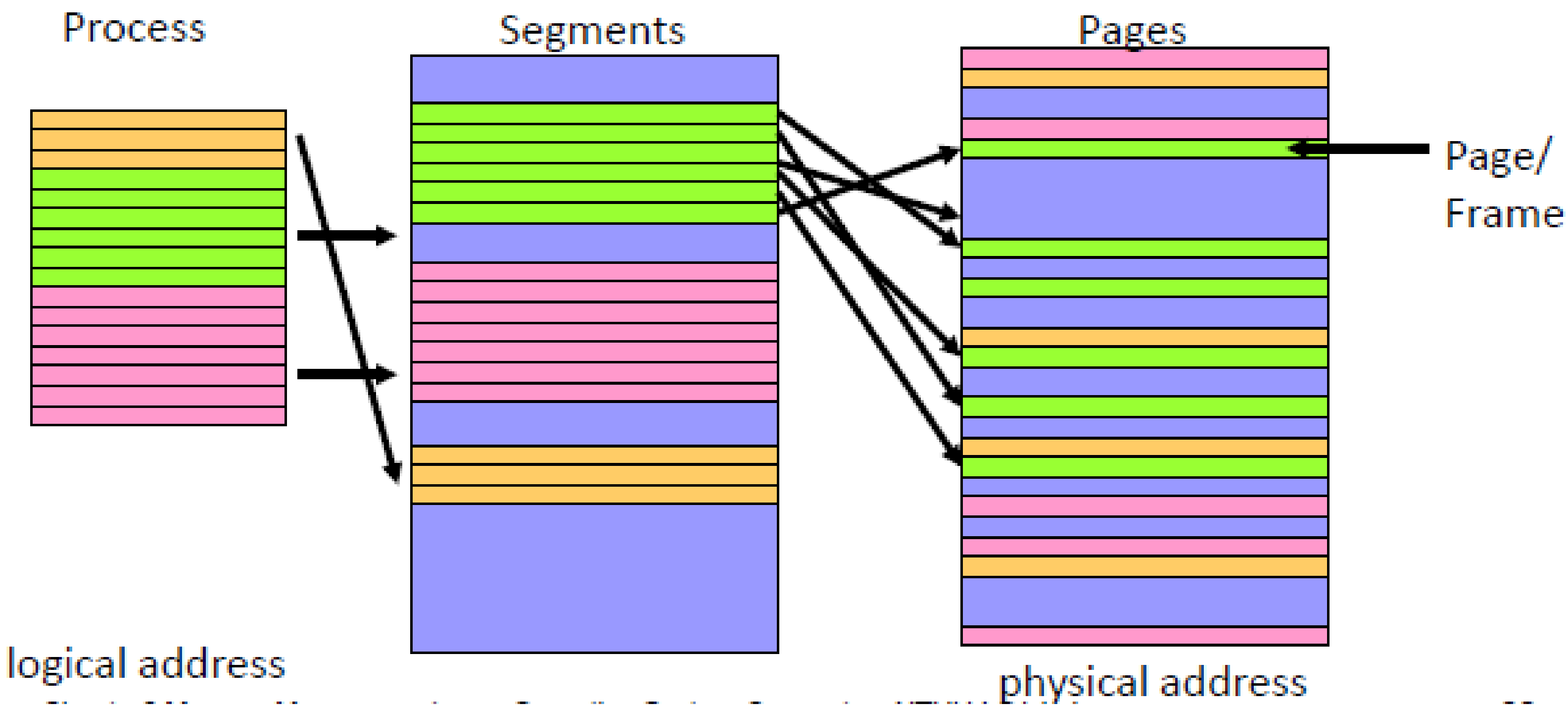


Sharing of Segments



Basic Concept

- Apply **segmentation** in **logical address space**
- Apply **paging** in **physical address space**

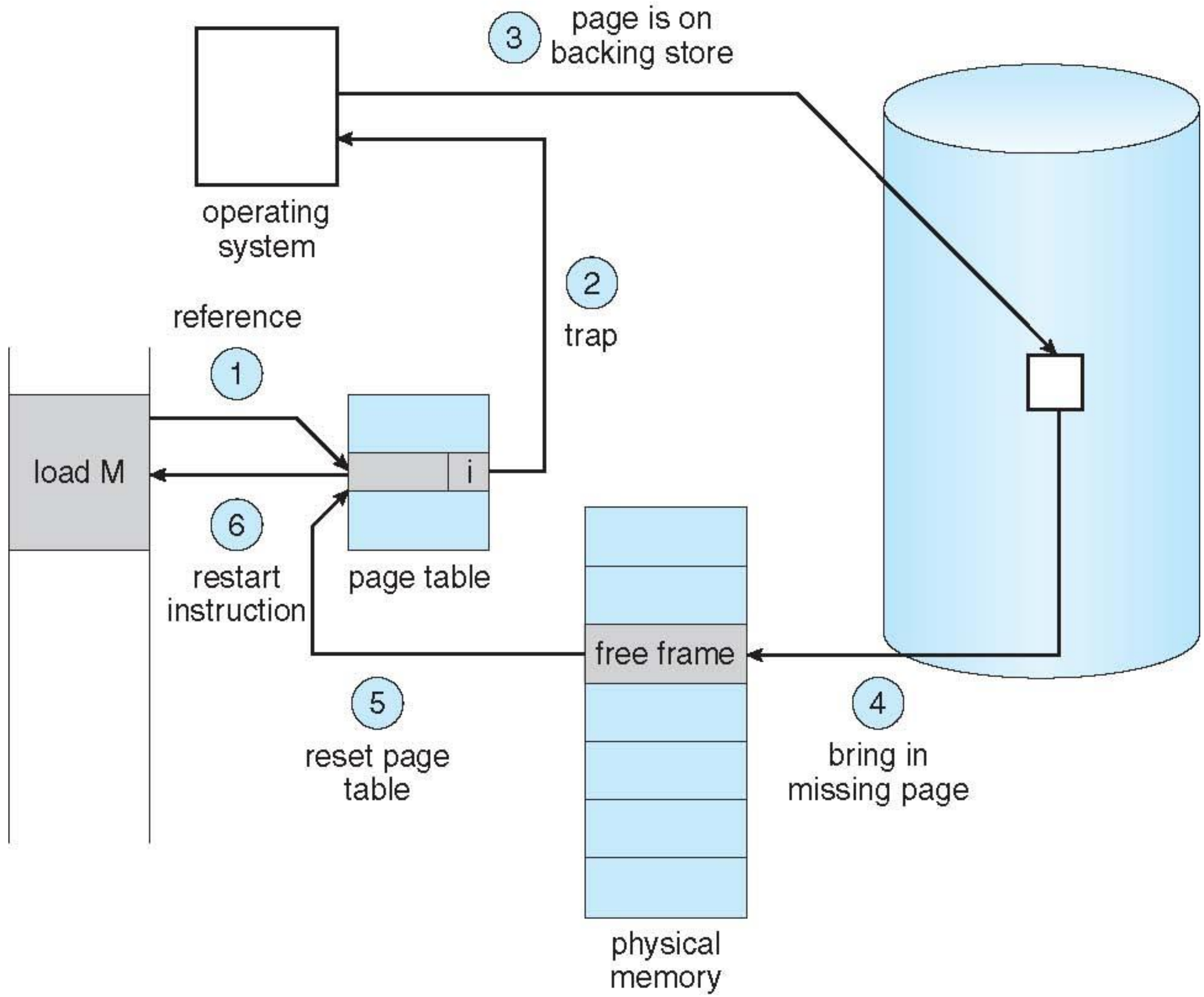


Virtual Memory
Overview

Virtual Memory

- Separation of user logical memory from physical memory
  - To run an **extremely large process**
    - Logical address space can be much larger than physical address space
  - To increase **CPU/resource utilization**
    - Higher degree of multiprogramming degree
  - To simplify programming (compiler) tasks
    - Free programmer from memory limitation
  - To **launch** programs **faster**
    - Less I/O would be needed to load or swap
- Can be implemented via
  - **Demand paging**
  - Demand segmentation (more complicated due to variable sizes)

Page Fault Handling



Virtual Memory
Process Creation

Process and Virtual Memory

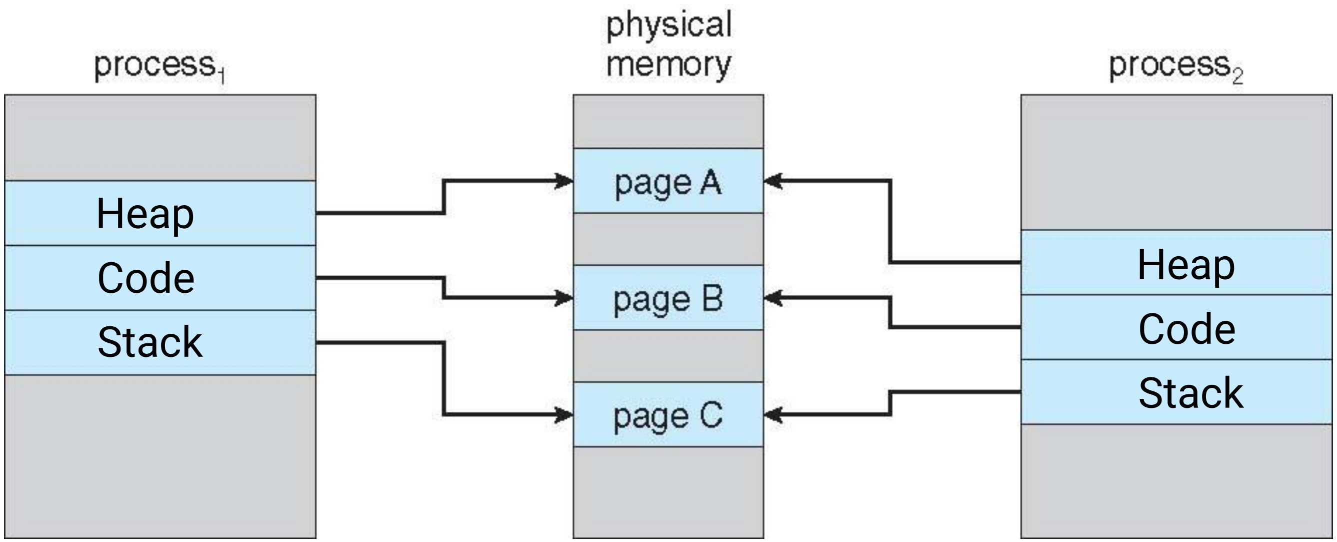
- **Demand Paging**
  - Only bring in the page containing the first instruction
- **Copy-on-Write**
  - The parent and the child process share the same frames initially, and frame-copy when a page is written

Copy-on-Write

- Allow both the parent and the child process to **share the same frames in memory**
- If either process modifies a frame, then a frame is **copied**
- Copy-on-write allows efficient process creation
- Free frames are allocated from a pool of **zeroed-out** frames (security reason)
  - The content of a frame is erased to 0

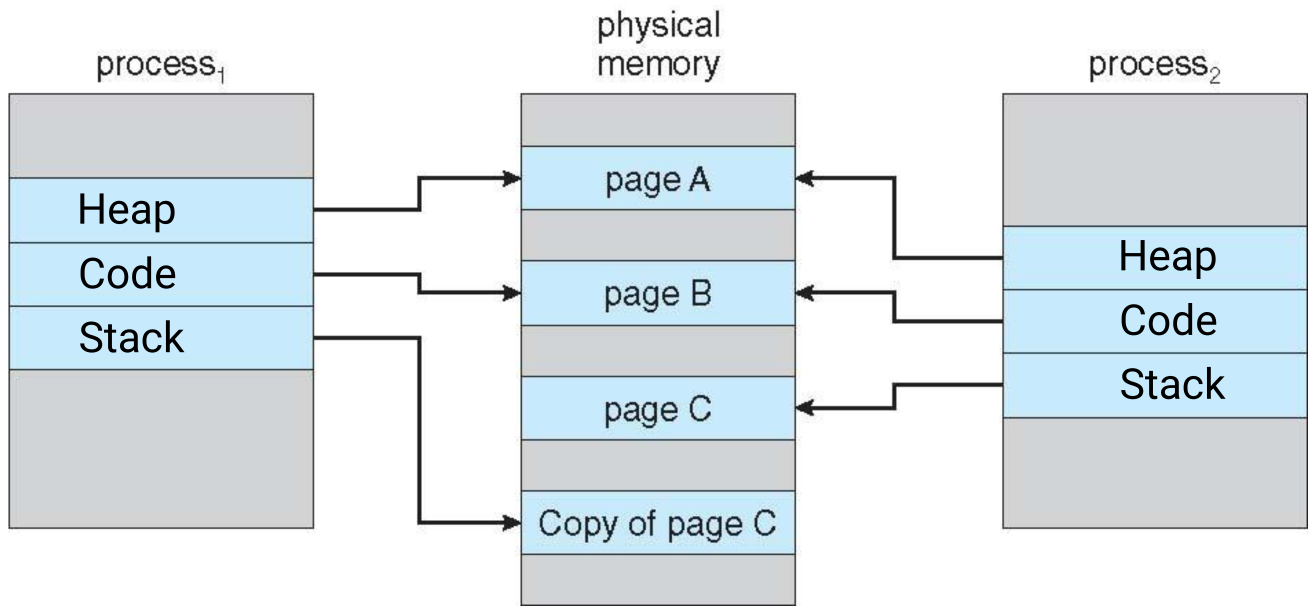
When a Child Process is Forked

```
#include <stdio.h>
void main( )
{
   int A;
   /* fork child process */
   A = fork( );

   if (A != 0) {
      /* parent process */
      int test1=0;
   }
   printf("process ends");
}
```



After a Page is Modified

```
#include <stdio.h>
void main( )
{
    int A;
    /* fork child process */
    A = fork( );

    if (A != 0) {
      /* parent process */
      int test1=0;
    }
    printf("process ends");
}
```



Virtual Memory

Page Replacement

Page Replacement Algorithms

- **Goal: lowest page-fault rate**
- Evaluation: running against a string of memory references (**reference string**) and computing the number of page faults
- Reference string example:

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

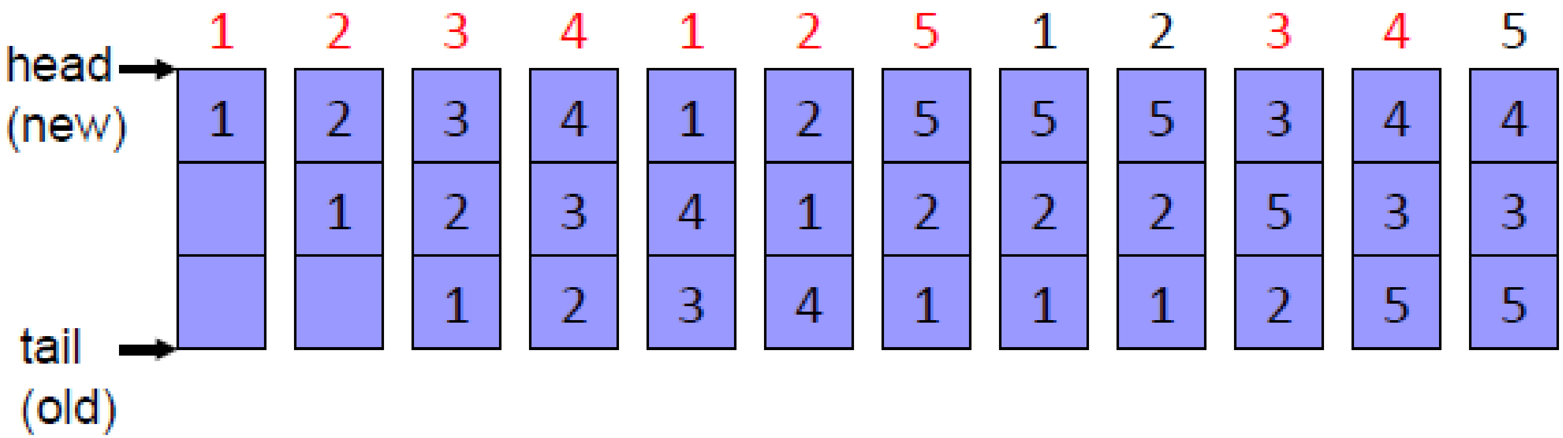
Replacement Algorithms

- FIFO algorithm
- Optimal algorithm
- LRU algorithm
- Counting algorithm
  - LFU
  - MFU

First-In-First-Out (FIFO) Algorithm

- The oldest page in a FIFO queue is replaced
- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (available memory frames = 3)

  ➔ 9 page faults



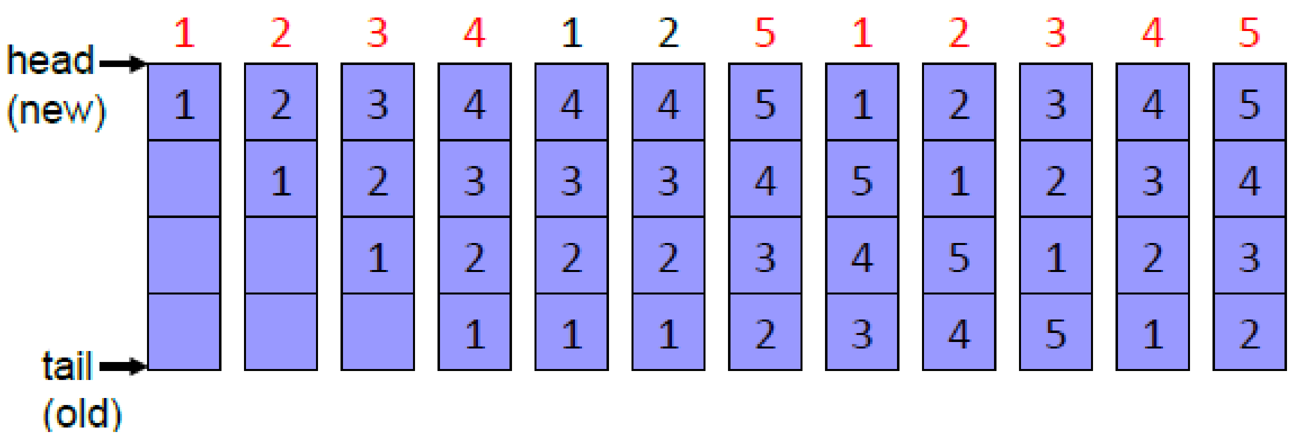
(example borrowed from Prof. Jerry Chou's slides)

FIFO Illustrating Belady's Anomaly

- Does more allocated frames guarantee less page fault?
  - Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
  - 4 frames (available memory frames = 4)

  ➔ **10** page faults !

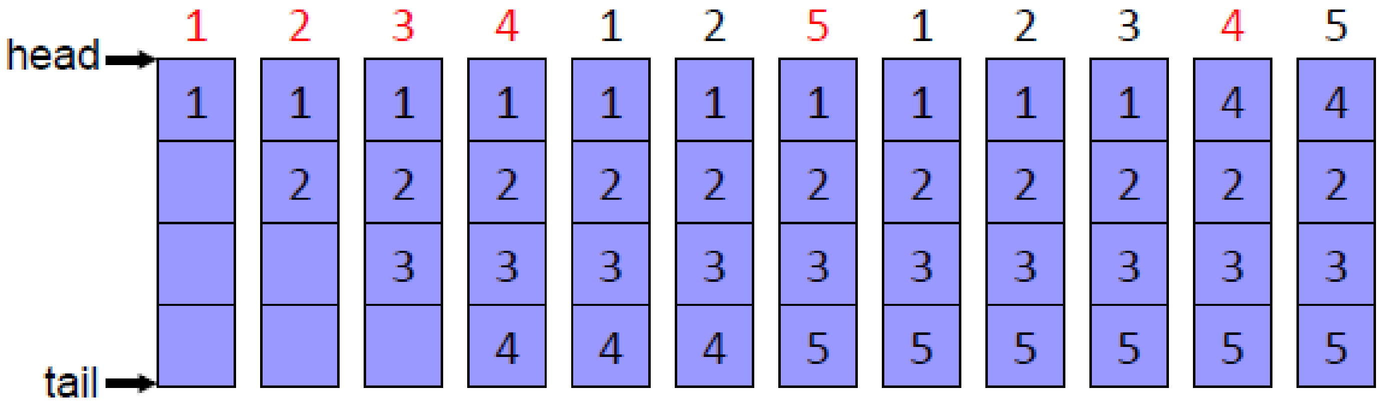
- **Belady's anomaly**
  - More allocated frames could result in more page faults



(example borrowed from Prof. Jerry Chou's slides)

Optimal (Belady) Algorithm

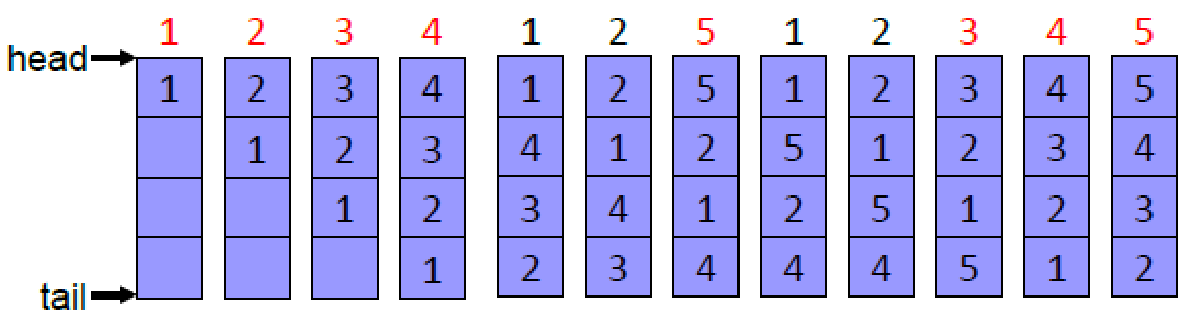
- Replace the page that will not be used for the **longest period of time**
  - Need future knowledge
- 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ➔ 6 page faults !
- In practice, we don't have future knowledge
  - Only used for reference and comparison



(example borrowed from Prof. Jerry Chou's slides)

LRU Algorithm Implementations

- Time stamp implementation
  - Page referenced: **time stamp** is copied into the counter
  - Replacement: remove the one with oldest counter
    - Linear search is required
- **Stack implementation**
  - Page referenced: move to top of the double-linked list
  - Replacement: remove the page at the bottom
  - 4 frames: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5 ➔ 8 page faults !



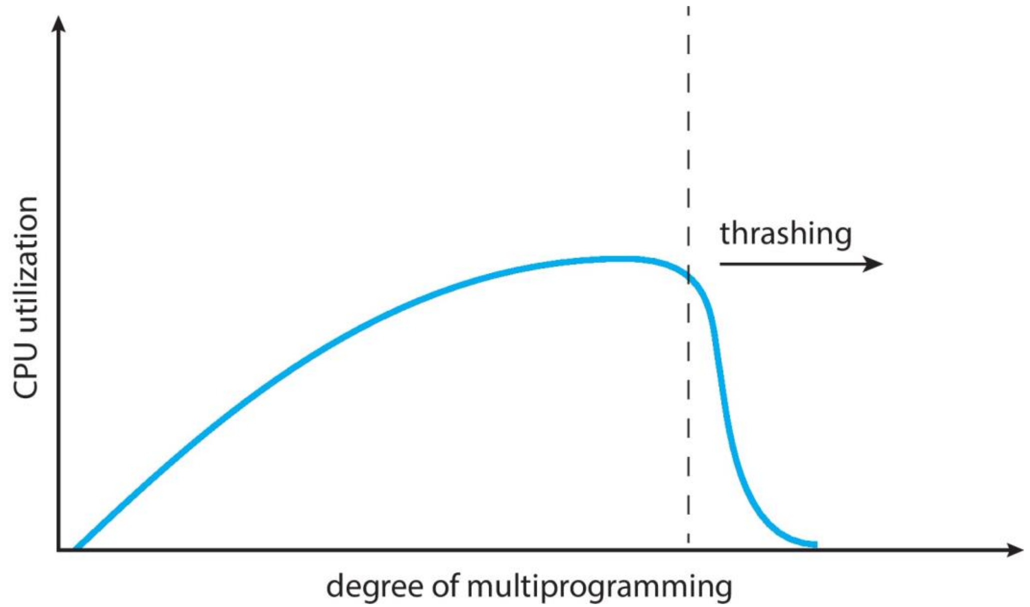
(example borrowed from Prof. Jerry Chou’s slides)

Virtual Memory

Thrashing

Definition of Thrashing

- If a process does not have enough **frames**
  - The process does not have # frames it needs to support pages in active use
    ➔ Very high paging activity
- **A process is thrashing if it is spending more time paging than executing**



Thrashing

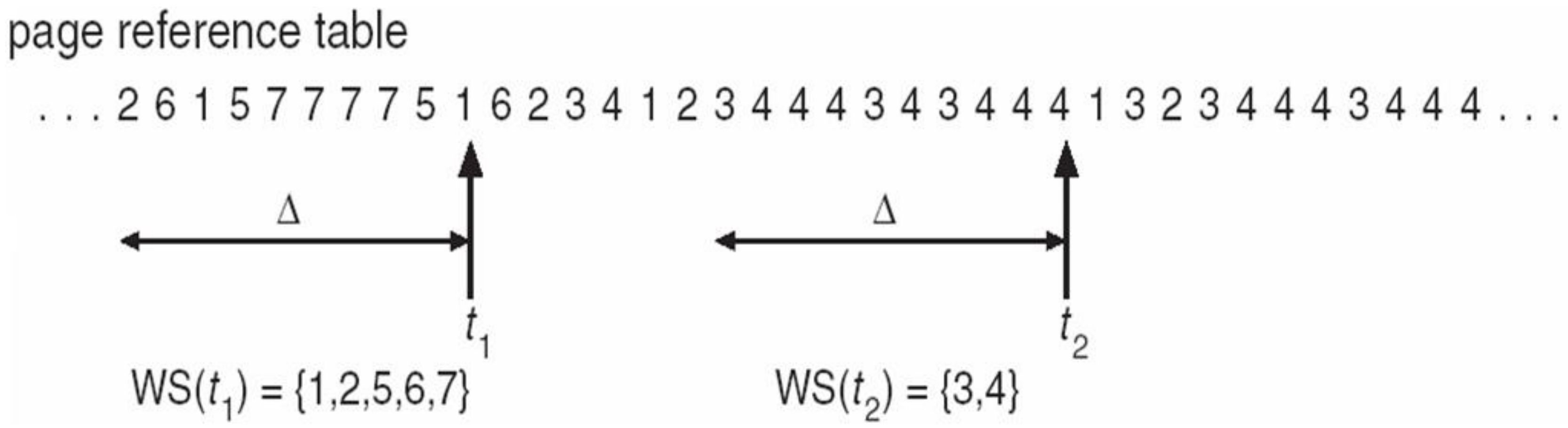
- Performance problem caused by thrashing (assume global replacement is used)
  - Processes queued for I/O to swap (page fault)
    - ➔ Low CPU utilization
    - ➔ OS increases the degree of multi-programming
    - ➔ New processes take frames from old processes
    - ➔ More page faults and thus more I/O
    - ➔ CPU utilization drops even further
- To prevent thrashing, must provide enough frames for each process
  - **Working-set model**
  - **Page-fault frequency**

Working-Set Model

- **Locality**: a set of pages that are actively used together
- Locality model: as a process executes, it moves from locality to locality
  - Program structure (subroutine, loop, stack)
  - Data structure (array, table)
- Working-set model (based on locality model)
  - Working-set **window**: a parameter Δ (delta)
  - Working-set: set of pages in most recent Δ page references (an approximation locality)

Working-Set Example

- If Δ (delta) = 10



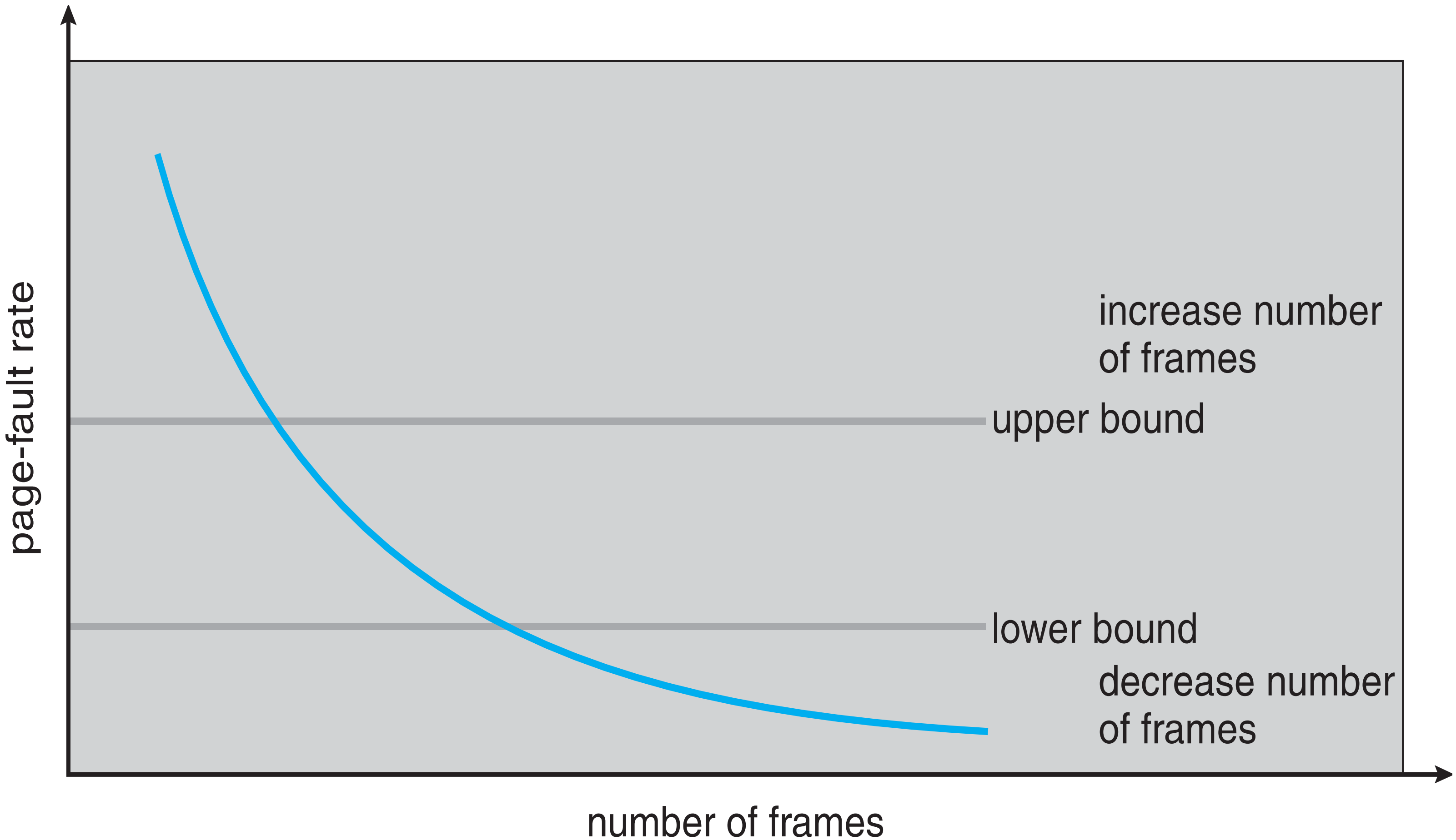
Working-Set Model (cont.)

- Prevent thrashing using the working-set size
  - WSS: working-set size for process *i*
  - $D = \sum WSS_i$ (total demand frames)
  - if D > m (available frames) ➔ thrashing
  - The OS monitors the WSS_i of each process and allocates to the process enough frames
    - if D << m, increase degree of MP
    - If D > m, suspend a process
- Prevent thrashing while keeping the degree of multiprogramming as high as possible
- Optimize CPU utilization
- **However, too expensive for tracking**

Page Fault Frequency Scheme

- **Page fault frequency** directly measures and controls the page-fault rate to prevent thrashing
  - Establish **upper** and **lower** bounds on the desired page-fault rate of a process
  - If page fault rate exceeds the upper limit
    - Allocate another frame to the process
  - If page rate falls below the lower limit
    - Remove a frame from the process

Page Fault Frequency Scheme (cont.)



That’s All !