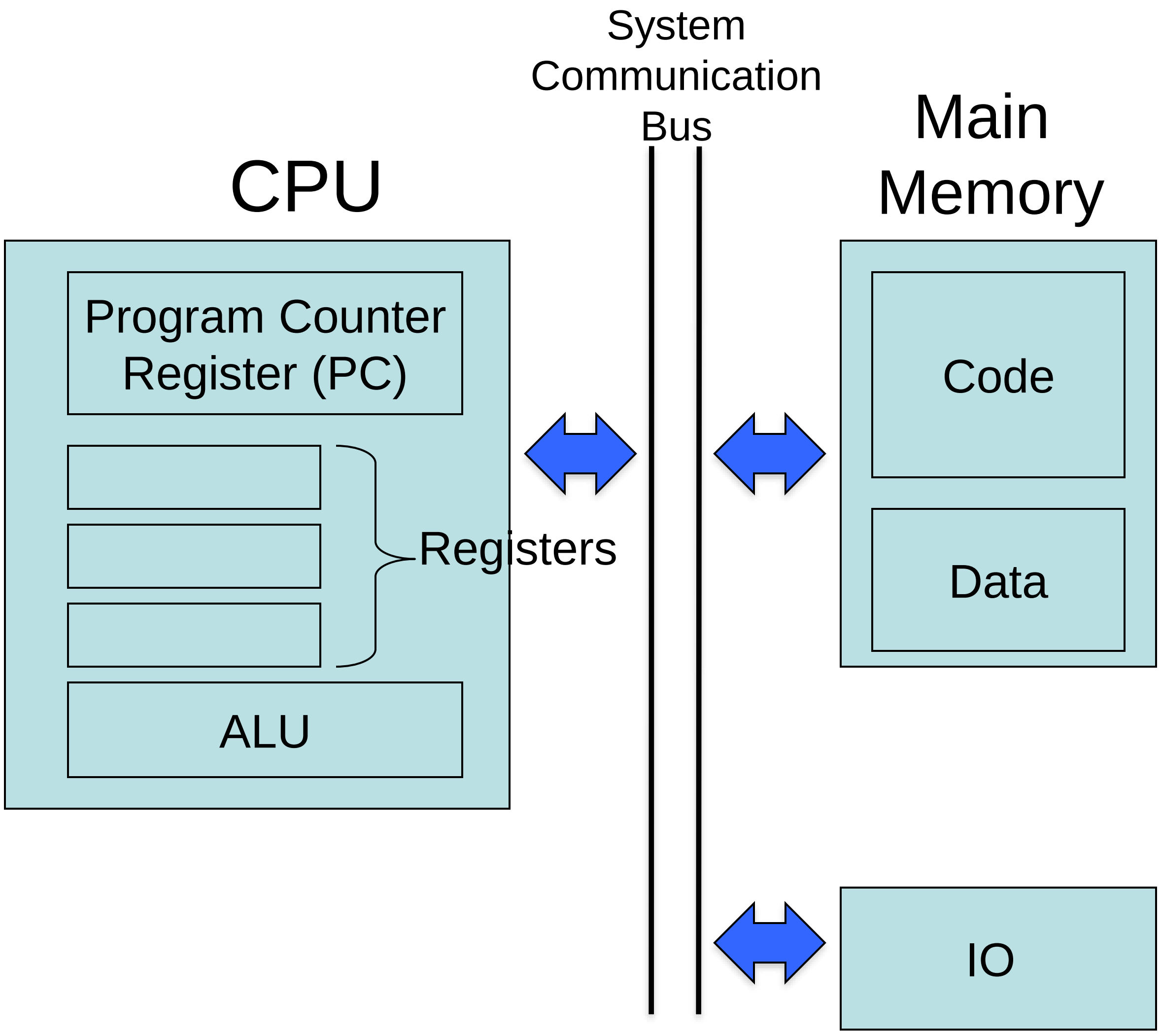
slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

CSCI 3753 Operating Systems
Summer 2020

Christopher Godley
PhD Student
Department of Computer Science
University of Colorado Boulder

Memory Management



CPU

Program Counter Register (PC)

Registers

ALU

System Communication Bus

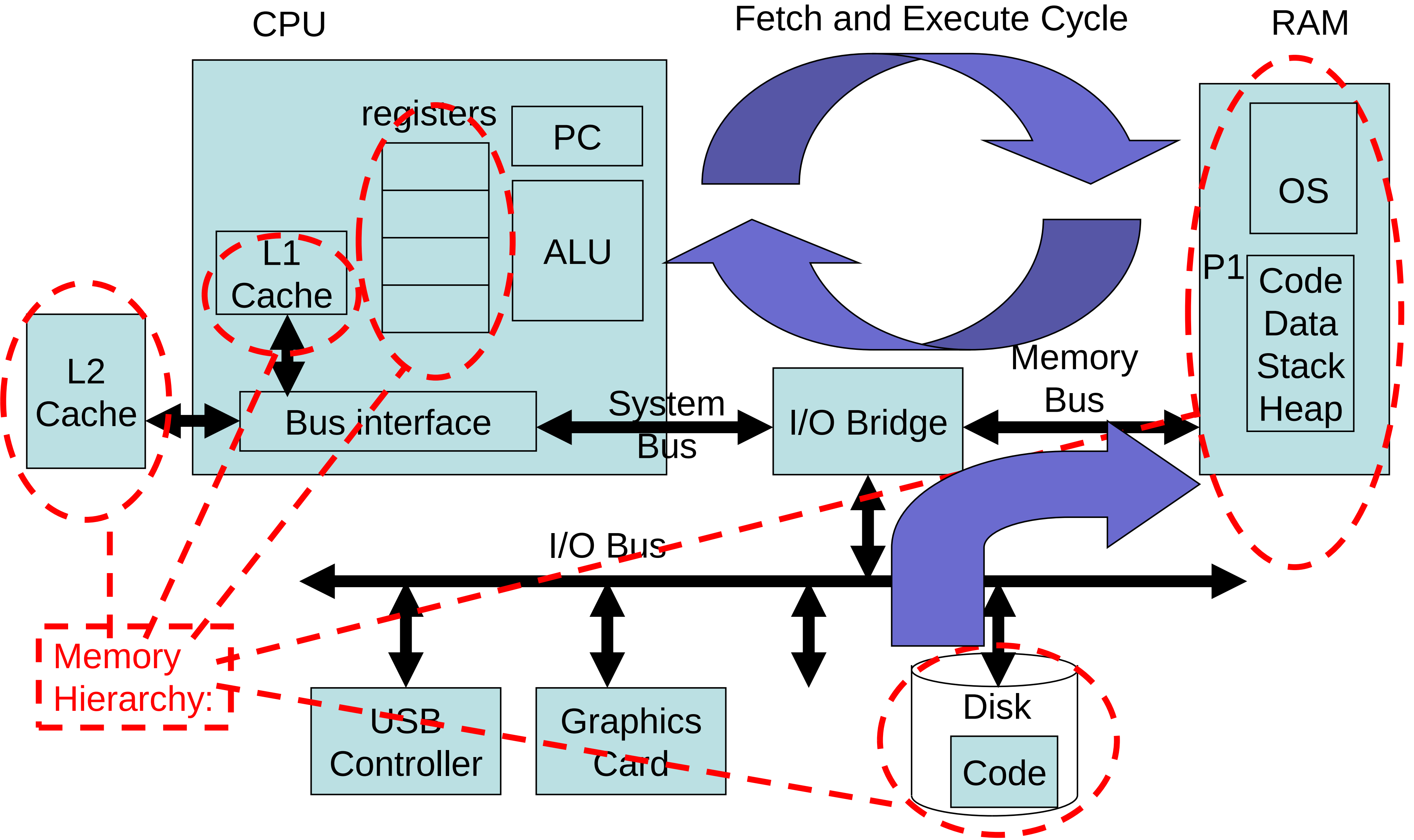
Main Memory

Code

Data

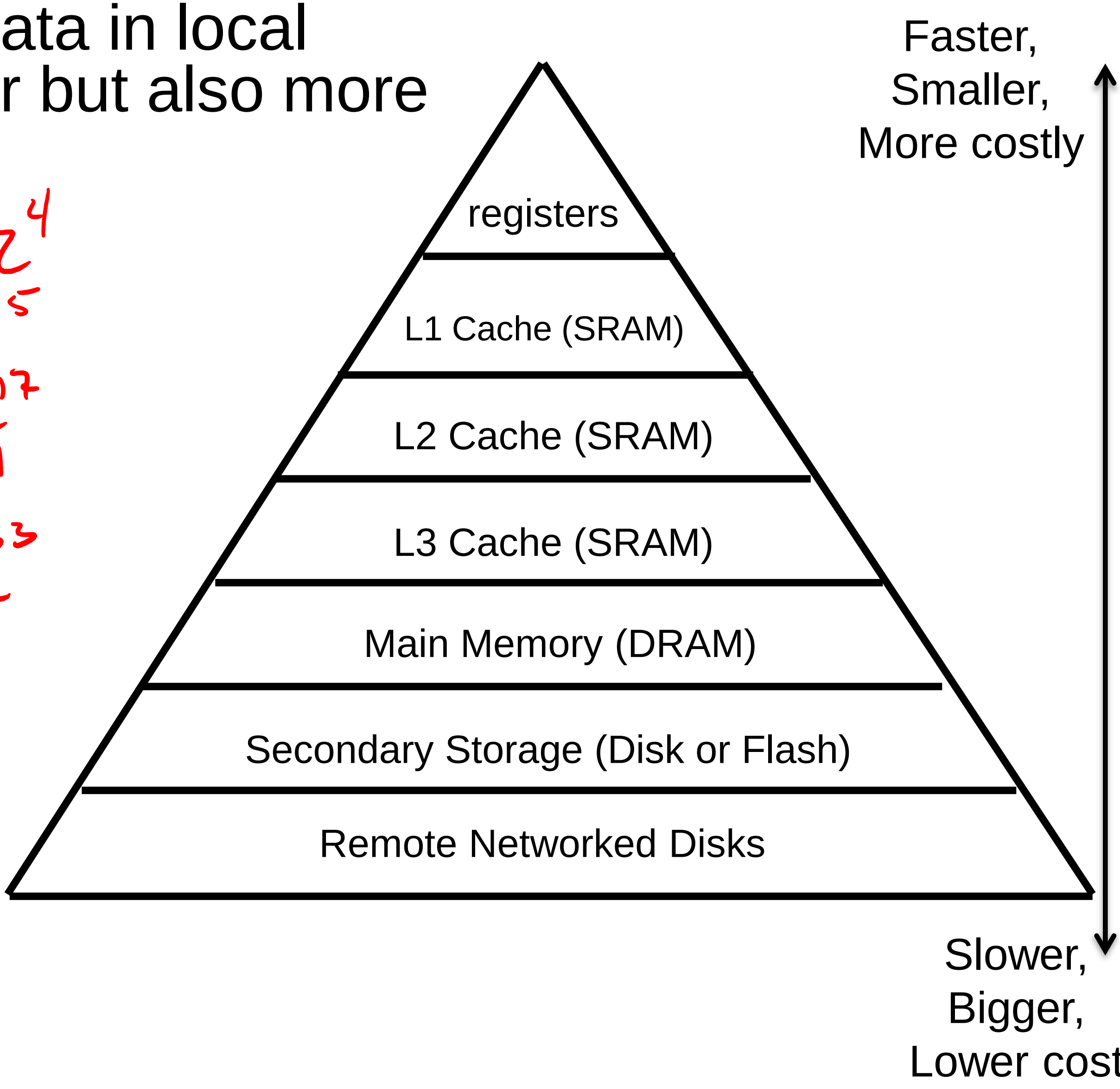
IO

Memory Management



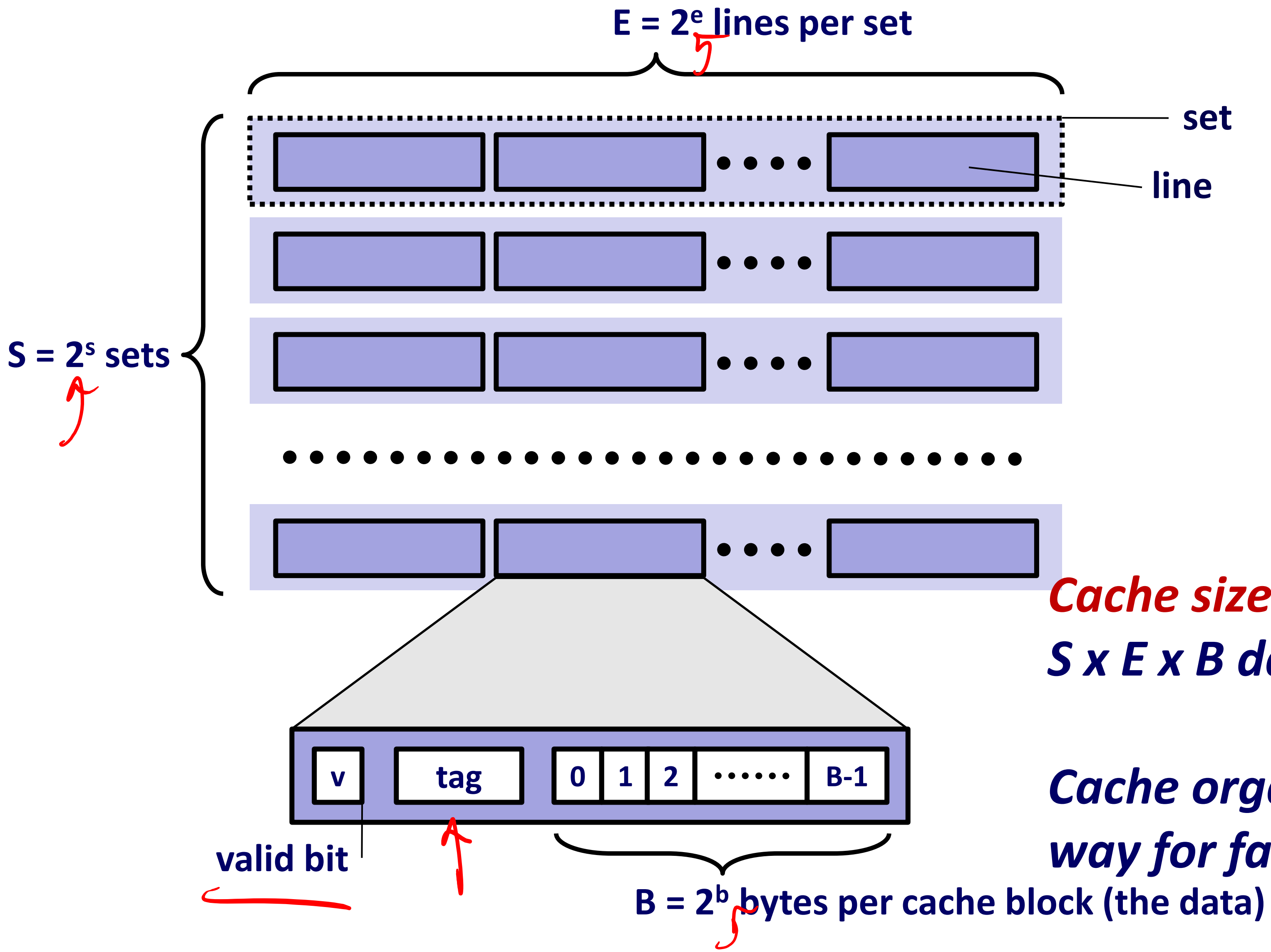
Memory Hierarchy

- cache frequently accessed instructions and/or data in local memory that is faster but also more expensive
  - Register < 1 cycle (16 B) 2^{4}
  - L1 = 1 cycle (~32 KB) 2^{15}
  - L2 = 3 cycles (~512 kB) 2^{17}
  - L3 = 10 cycles (2MB) 2^{21}
  - RAM = 200 cycles (GB) 2^{33}
  - Permanent storage:
    - Flash = 100k cycles (GB)
    - Disk =1M cycles (TB)
    - Network = 1B cycles (PB)



General Cache Organization (S, E, B)

For L1 hardware caches, access must be fast, so organize as follows:



$E = 2^e$ lines per set

set

line

$S = 2^s$ sets

v

tag

0 1 2 ······ B-1

valid bit

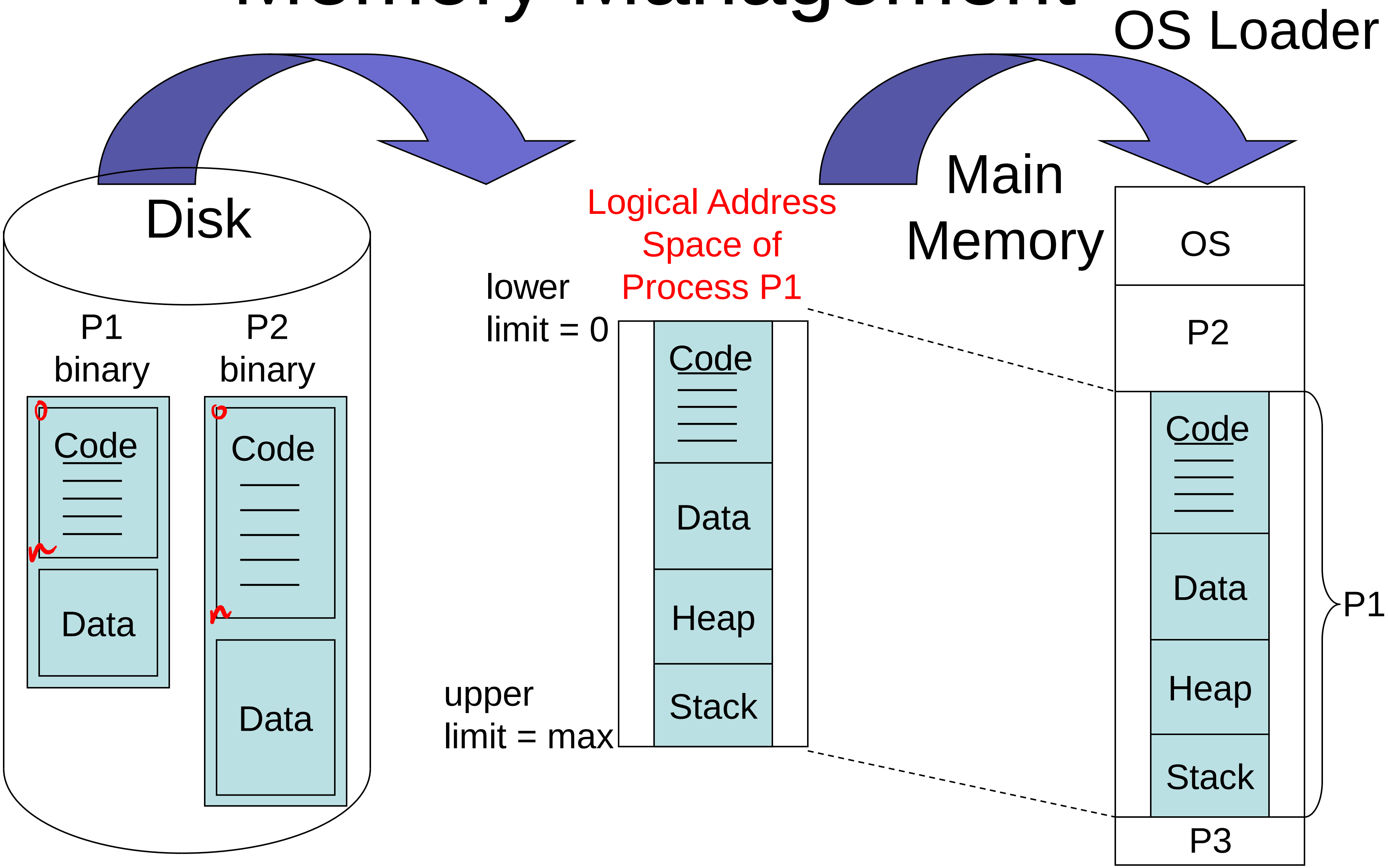
$B = 2^b$ bytes per cache block (the data)

Cache size:

S x E x B data bytes

Cache organized this way for fast access

Memory Management



OS Loader

Disk

P1 binary

0

Code

n

Data

P2 binary

0

Code

n

Data

Logical Address Space of Process P1

lower limit = 0

Code

Data

Heap

Stack

upper limit = max

Main Memory

OS

P2

Code

Data

Heap

Stack

P1

P3

Memory Management

- In the previous figure, want newly active process P1 to execute in its own *logical address space* ranging from 0 to max
  - It shouldn’t have to know exactly where in physical memory its code and data are located
  - This decouples the compiler from run-time execution
  - There needs to be a mapping from logical addresses to physical addresses at run time
    - memory management unit (MMU) takes care of this.

Memory Management

- MMU must do:

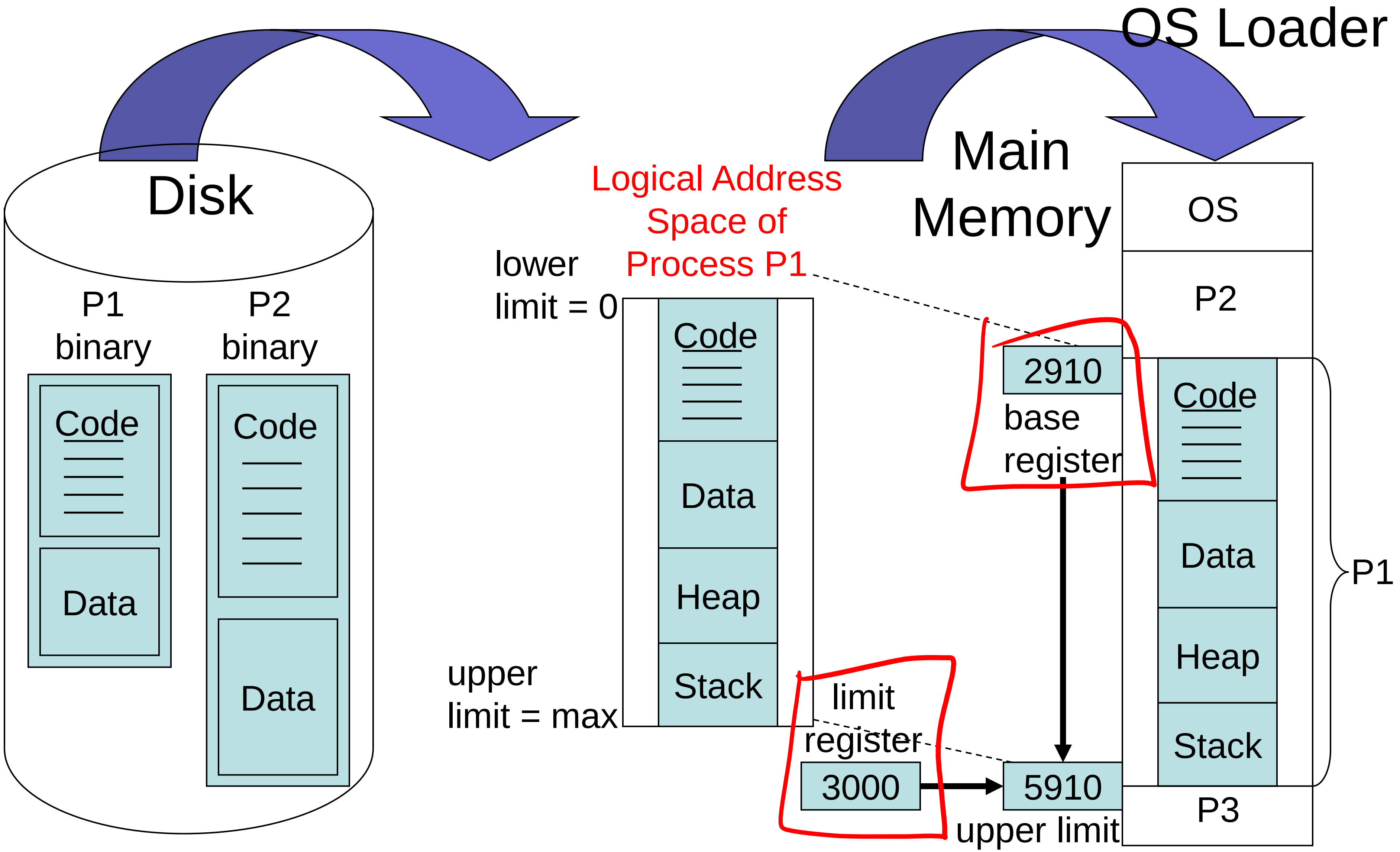
  1. Address translation: translate logical addresses into physical addresses, i.e. map the logical address space into a *physical address space*

  2. Bounds checking: check if the requested memory address is within the upper and lower limits of the address space

  One approach is:

  - *base register* in hardware keeps track of lower limit of the physical address space
  - *limit register* keeps track of size of logical address space
  - upper limit of physical address space = base register + limit register

Memory Management



OS Loader

Disk

P1 binary

Code

Data

P2 binary

Code

Data

Logical Address Space of Process P1

lower limit = 0

Code

Data

Heap

Stack

upper limit = max

Main Memory

2910

base register

limit register

3000

5910

upper limit

OS

P2

Code

Data

Heap

Stack

P1

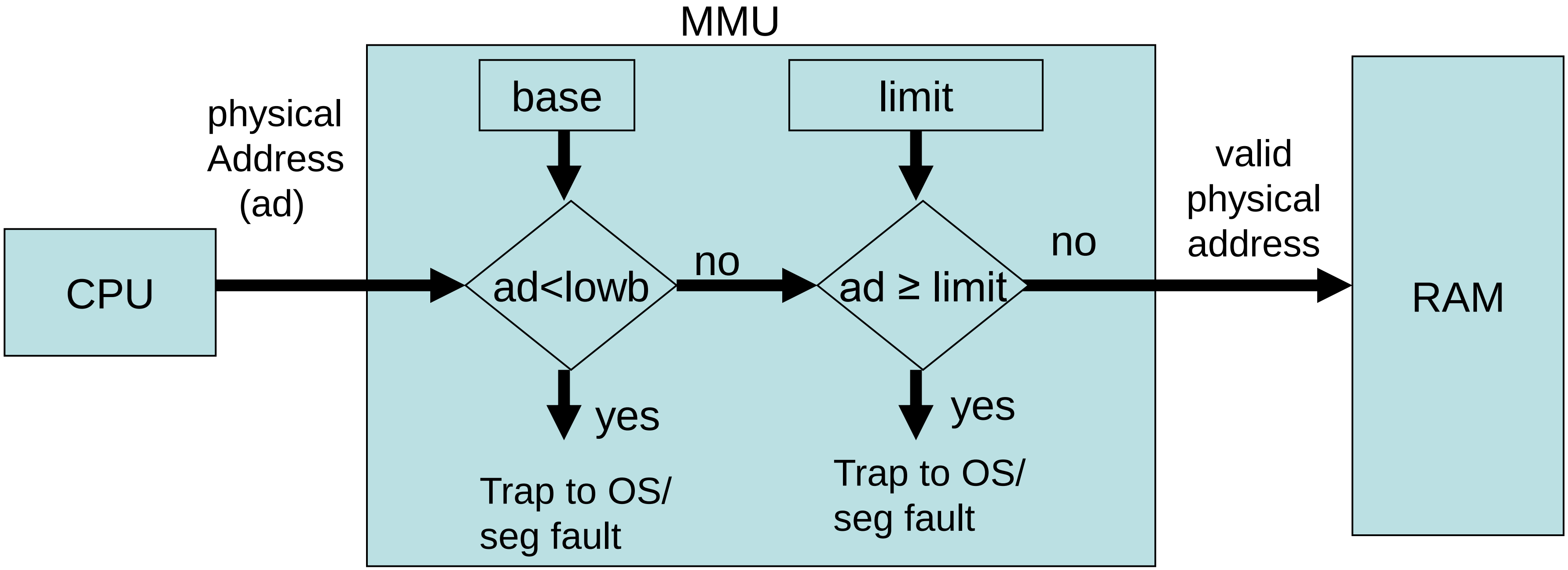
P3

Memory Management

- base and limit registers provide hardware support for a simple MMU (Memory Management Unit)
  - memory access should not go out of bounds.
  - If out of bounds, then this is a segmentation fault so trap to the OS.
  - MMU will detect out-of-bounds memory access and notify OS by throwing an exception
- Only the OS can load the base and limit registers while in kernel/supervisor mode
  - these registers would be loaded as part of a context switch

Memory Management

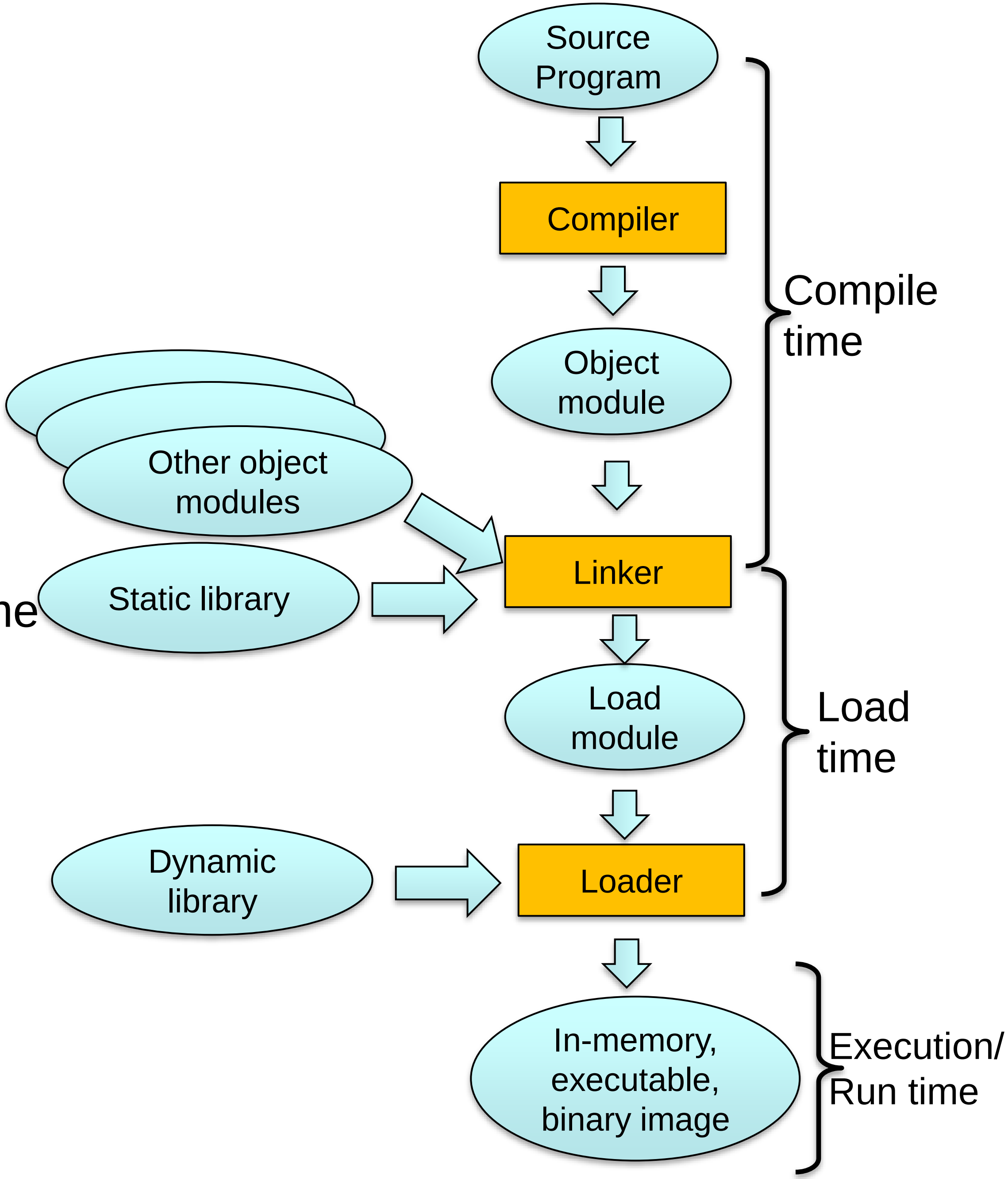
- MMU needs to check if physical memory access is out of bounds



Loading Tasks into Memory

- Using contiguous memory to hold task
- Task is stored as Code and Data on disk
- Task contains Code, Data, Stack, and Heap in memory
- Multiple tasks can be loaded in memory at the same time
- Kernel is responsible for protecting task memory from other tasks
  - Memory Management Unit(MMU) is used to validate all addresses.
  - Each task needs a base and limit register
  - Context switch must save/restore these address registers

- How does it know the addresses we want to access?
  - Binding at compile time
  - Binding at load time
  - Binding at execution time



Memory Management

- Address Binding at Compile Time:
  - If you know in advance where in physical memory a process will be placed, then compile your code with absolute physical addresses
  - Example: LOAD MEM_ADDR_X, reg1

    STORE MEM_ADDR_Y, reg2

  MEM_ADDR_X and MEM_ADDR_Y are hardwired by the compiler as absolute physical addresses

Memory Management

- Address Binding at Load Time
  - Code is first compiled in relocatable format.
  - **Replace logical addresses in code with physical addresses during loading**
  - Example: LOAD MEM_ADDR_X, reg1

    STORE MEM_ADDR_Y, reg2

At load time, the loader replaces all occurrences in the code of MEM_ADDR_X and MEM_ADDR_Y with (base+MEM_ADDR_X) and (base+MEM_ADDR_Y).

  - Once the binary has been thus changed, it is not really portable to any other area of memory, hence load time bound processes are not suitable for swapping (see later slides)

Memory Management

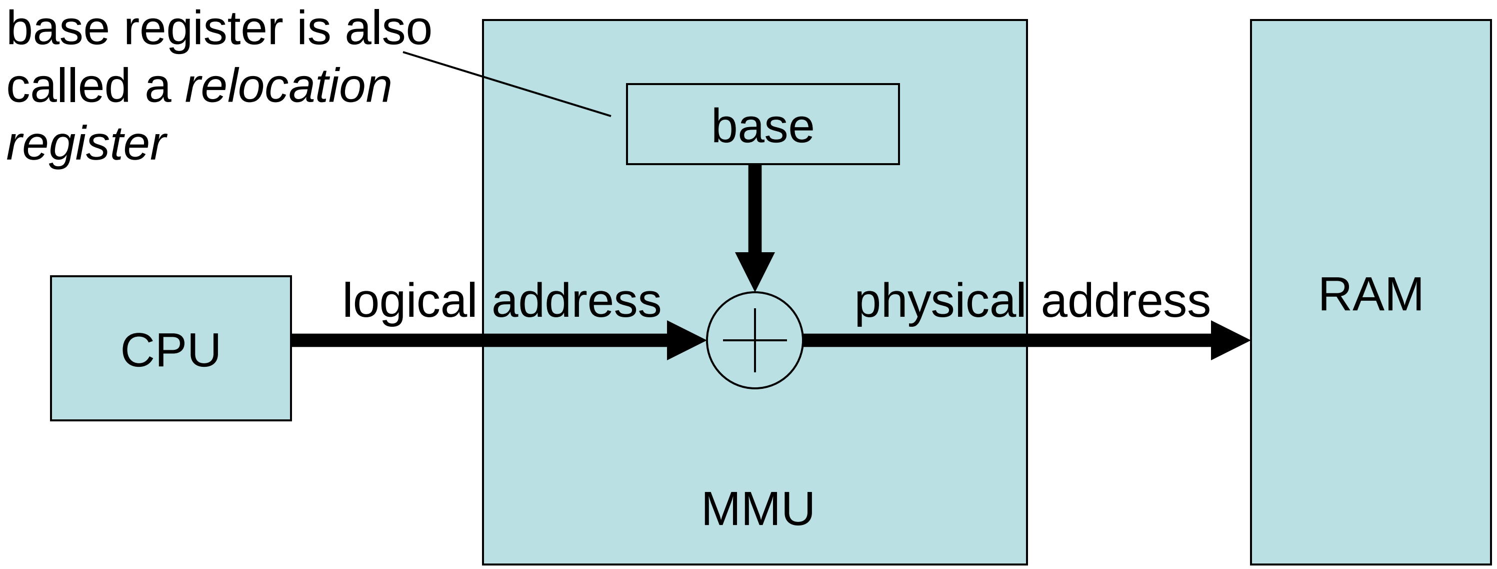
- Address Binding at Run Time (most modern OS's do this)
  - Code is first compiled in relocatable format as if executing in its own logical/virtual address space.

  - As each instruction is executed, i.e. at run time, the MMU relocates the logical address to a physical address using hardware support such as base/relocation registers.

  - Example: LOAD MEM_ADDR_X, reg1

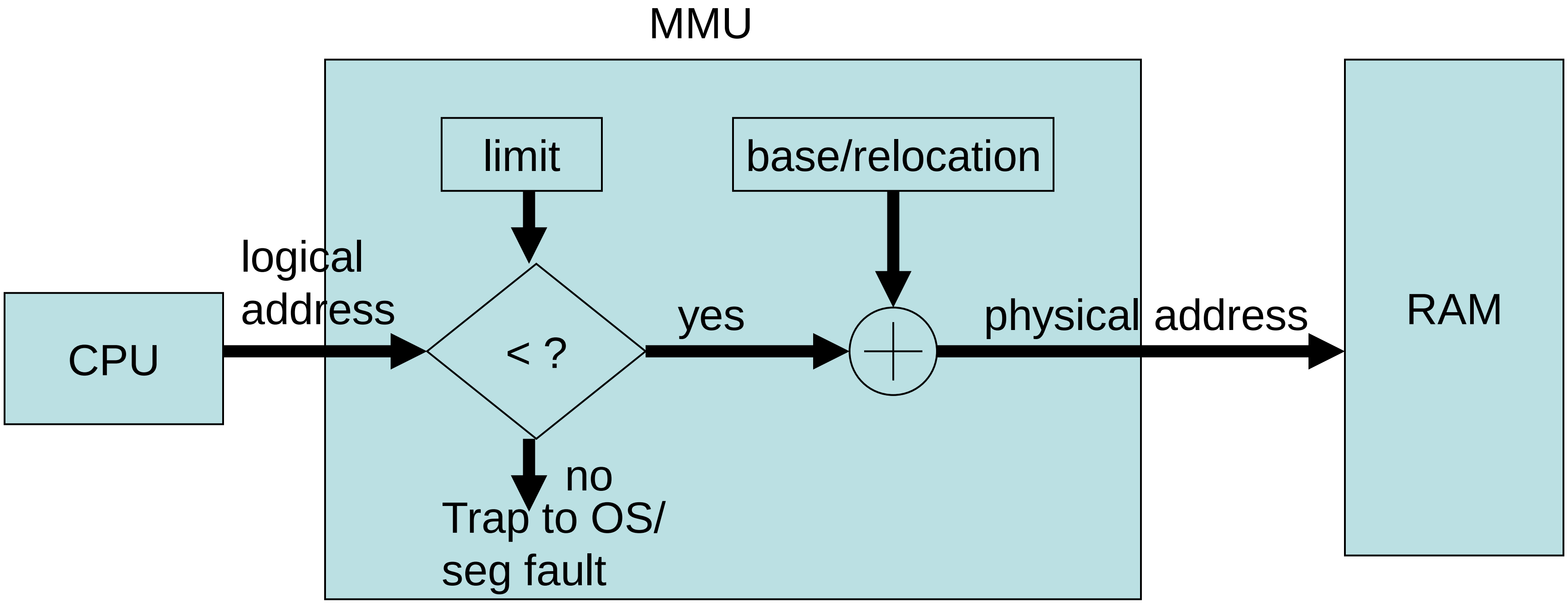
    MEM_ADDR_X is compiled as a logical address, and implicitly the **hardware MMU** will translate it to base+MEM_ADDR_X when the instruction executes

- MMU needs to perform run-time *mapping* of logical/virtual addresses to physical addresses
  - For run-time address binding,
    - each logical address is *relocated or translated* by MMU to a physical address that is used to access main memory/RAM
    - thus the application program never sees the actual physical memory - it just presents a logical address to MMU



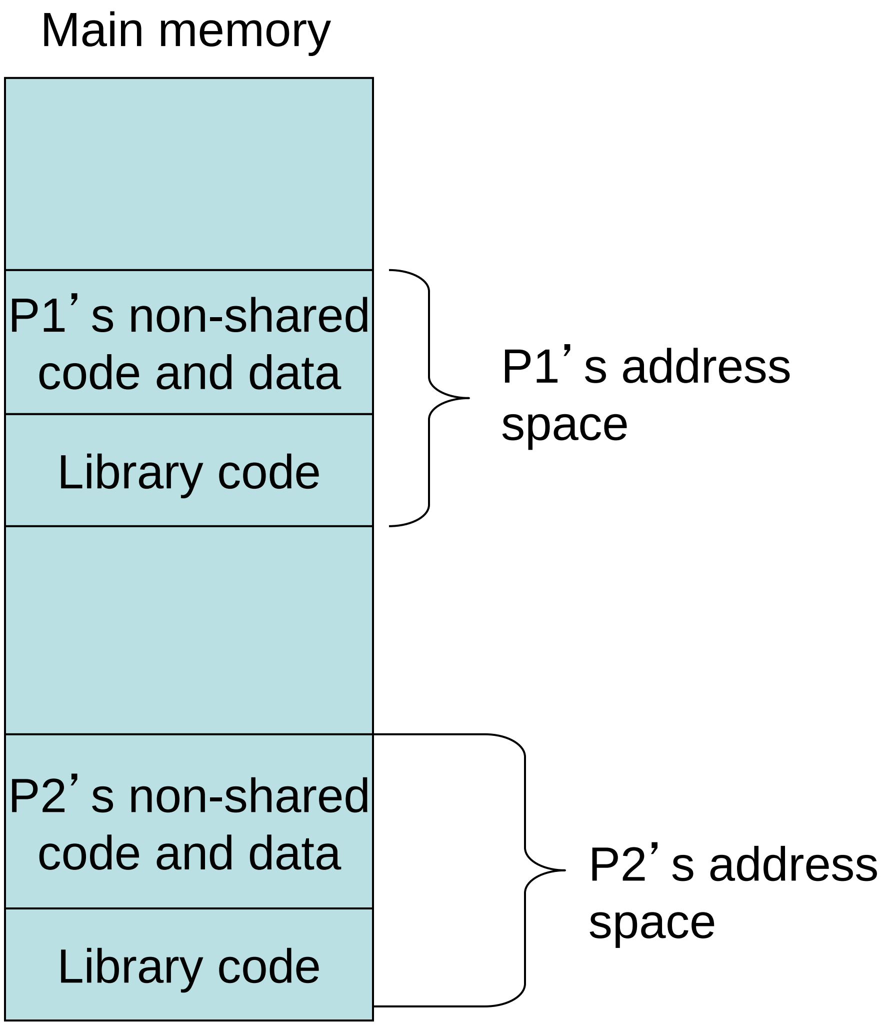
Memory Management

- Let’ s combine the MMU’s two tasks (bounds checking, and memory mapping) into one figure
  - since logical addresses can’ t be negative, then lower bound check is unnecessary - just check the upper bound by comparing to the limit register
  - Also, by checking the limit first, no need to do relocation if out of bounds



Run Time Binding with Static Linking

- Advantages of static linking:
  - Applications have access to the same library code even if it has changed recently
  - Can move to machines without the library
  - Self contained processes

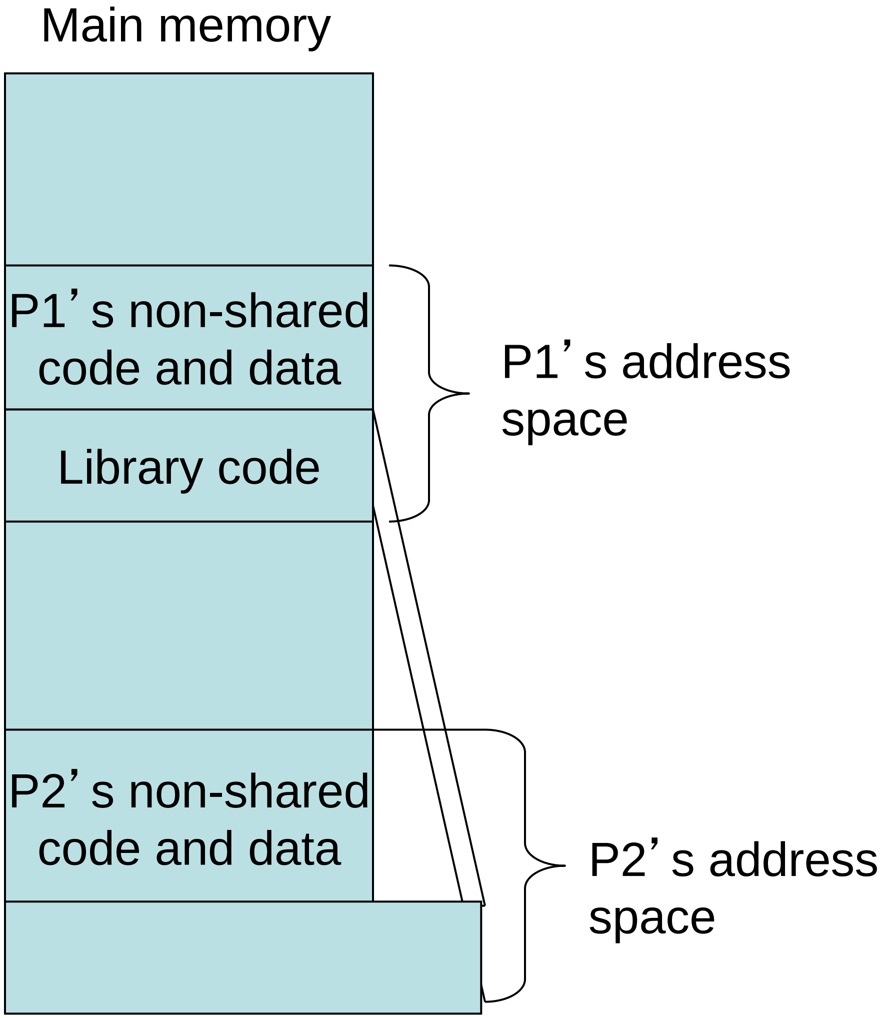


Run Time Binding

- Statically linked executable
  - Logical addresses are translated instruction by instruction into physical addresses at run time, *and the entire executable has all the code it needs at compile time through static linking*
  - Once a function is statically linked, it is embedded in the code and can’t be changed except through recompilation
    - Your code can contain outdated functions that don’t have the latest bug fixes, performance optimizations, and/or feature enhancements

Run Time Binding with Dynamic Linking

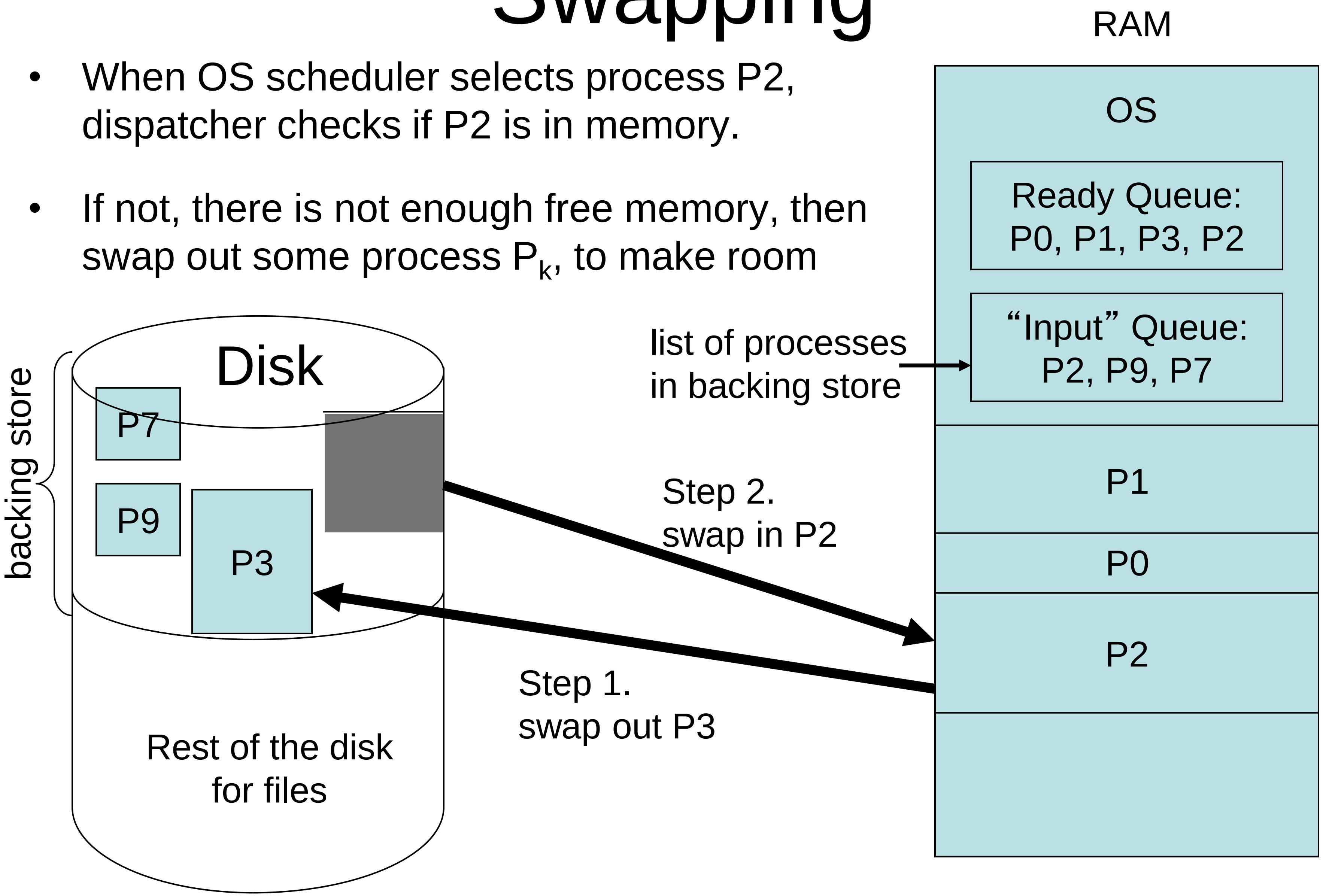
- Advantages of dynamic linking:
  - Applications have access to the latest code at run-time, e.g. most recent patched dlls,
  - Smaller size – stubs stay stubs unless activated
  - Can have only one copy of the code that is shared among all applications
    - We'll see later how code is shared between address spaces using page tables



Swapping, Fragmentation, and Segmentation

Swapping

- When OS scheduler selects process P2, dispatcher checks if P2 is in memory.
- If not, there is not enough free memory, then swap out some process P_k, to make room



Swapping

- If run time binding is used, then a process can be easily swapped back into a different area of memory.

- If compile time or load time binding is used, then process swapping will become very complicated and slow - basically undesirable

Swapping Difficulties

- Context-switch time of swapping is very slow
  - Disks take on the order of 10s-100s of ms per access
  - When adding the size of the process to transfer, then transfer time can take seconds
  - Ideally hide this latency by having other processes to run while swap is taking place behind the scenes,
    - e.g. in RR, swap out the just-run process, and have enough processes in round robin to run before swap-in completes & newly swapped-in process is ready to run
  - can't always hide this latency if in-memory processes are blocked on I/O
  - avoids swapping unless the memory usage exceeds a threshold

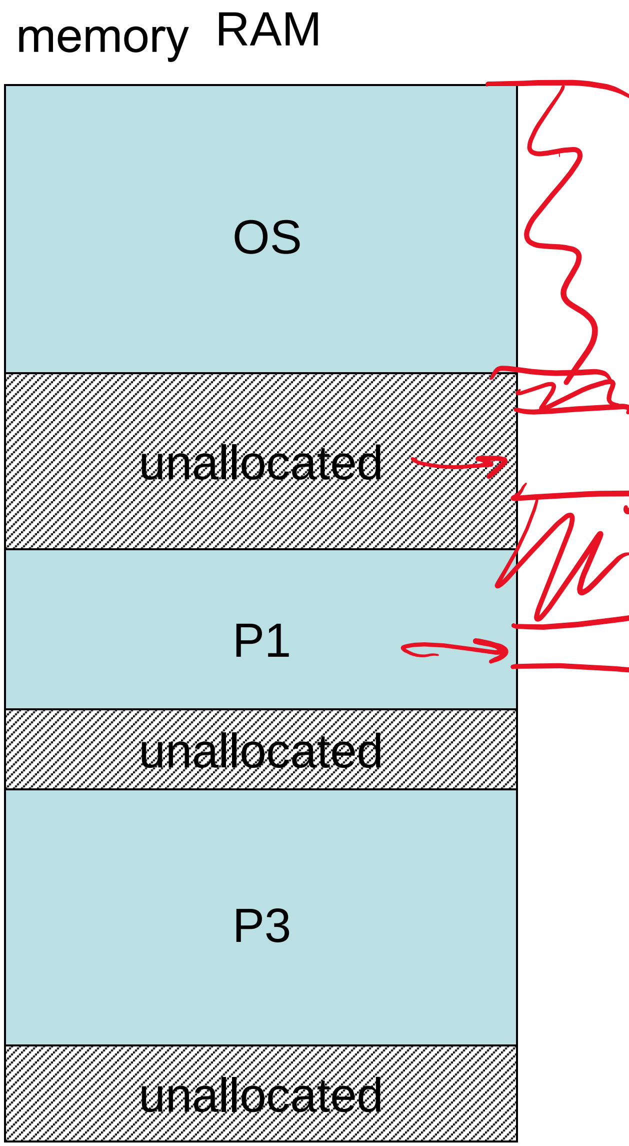
Swapping Difficulties

- swapping of processes that are blocked or waiting on I/O becomes complicated
  - one rule is to simply avoid swapping processes with pending I/O
- fragmentation of main memory becomes a big issue
  - can also get fragmentation of backing store disk
- Modern OS's swap portions of processes in conjunction with virtual memory and demand paging

Memory Allocation

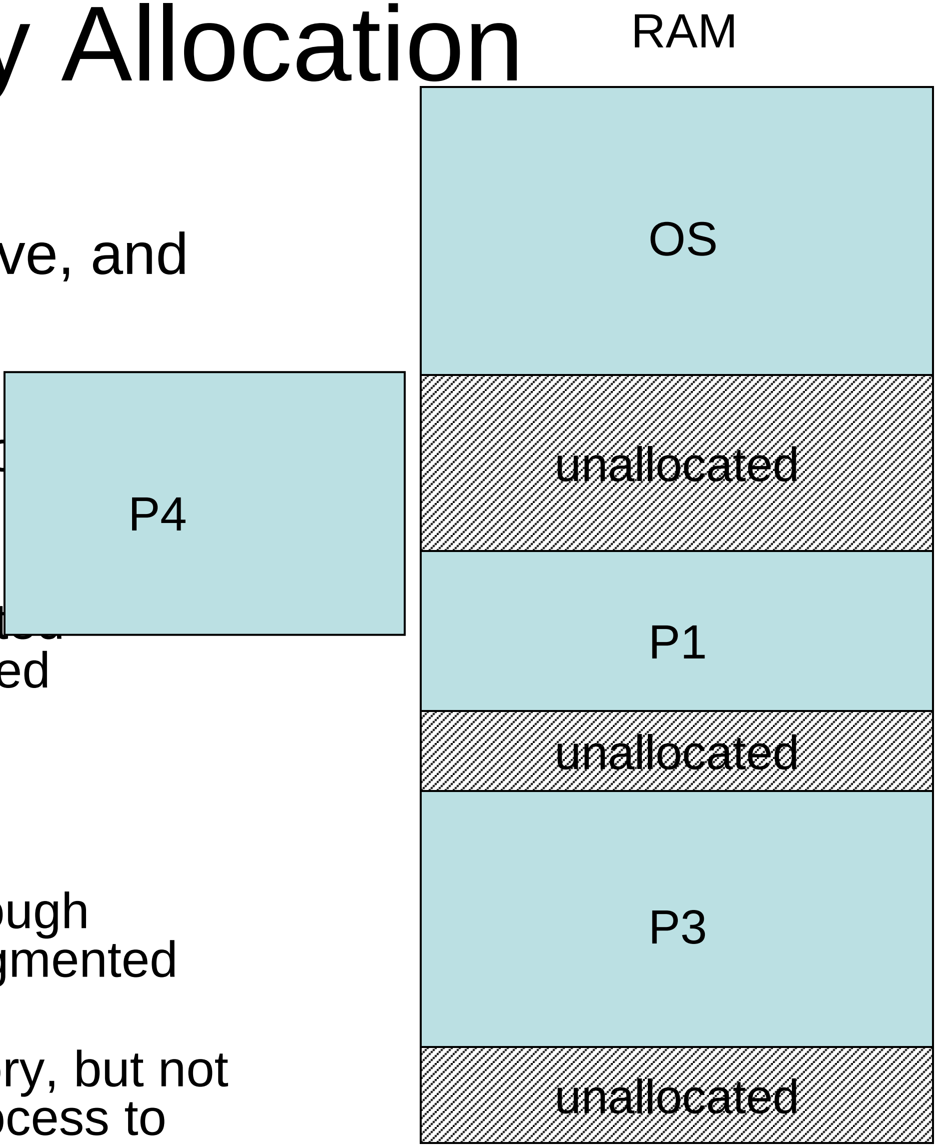
as processes arrive, they're allocated a space in main memory

- Allocation Strategies:
  - *best fit* - find the smallest chunk that is big enough
    - This results in more and more fragmentation
  - *worst fit* - find the largest chunk that is big enough
    - this leaves the largest contiguous unallocated chunk for the next process
  - *first fit* - find the 1st chunk that is big enough
    - This tends to fragment memory near the beginning of the list
  - *next fit* – view fragments as forming a circular buffer
    - find the 1st chunk that is big enough after the most recently chosen fragment



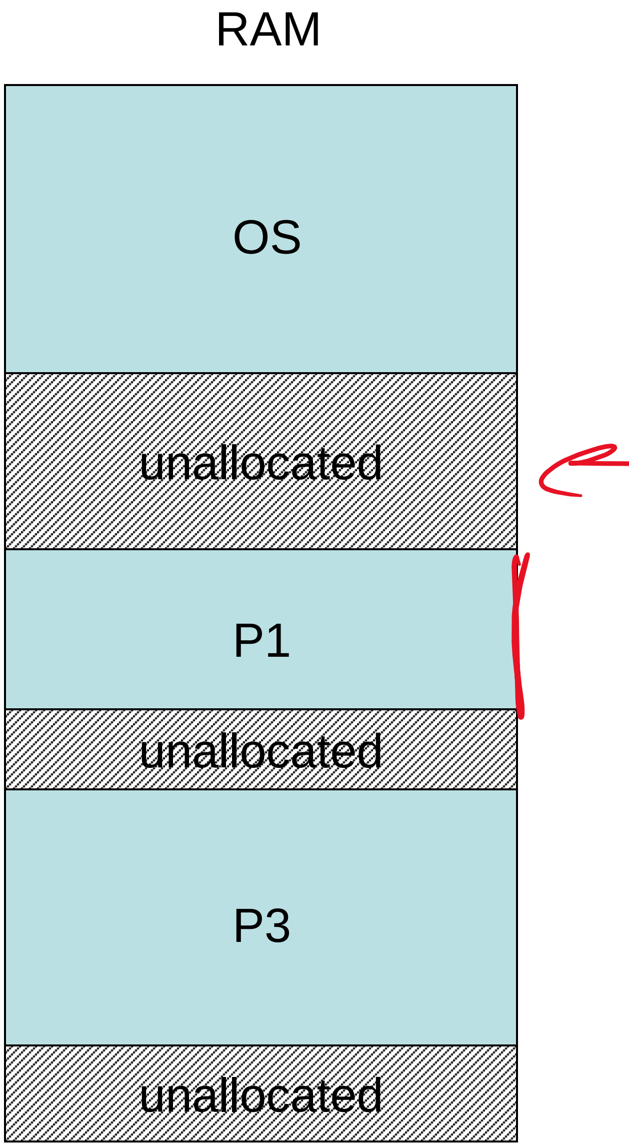
Memory Allocation

- over time, processes leave, and memory is deallocated
- results in *fragmentation* o memory
  - There are many small ch non-contiguous unallocated memory between allocated processes in memory
- for the next process,
  - OS must find a large enough unallocated chunk in fragmented memory
  - May have enough memory, but not contiguous to allow a process to load

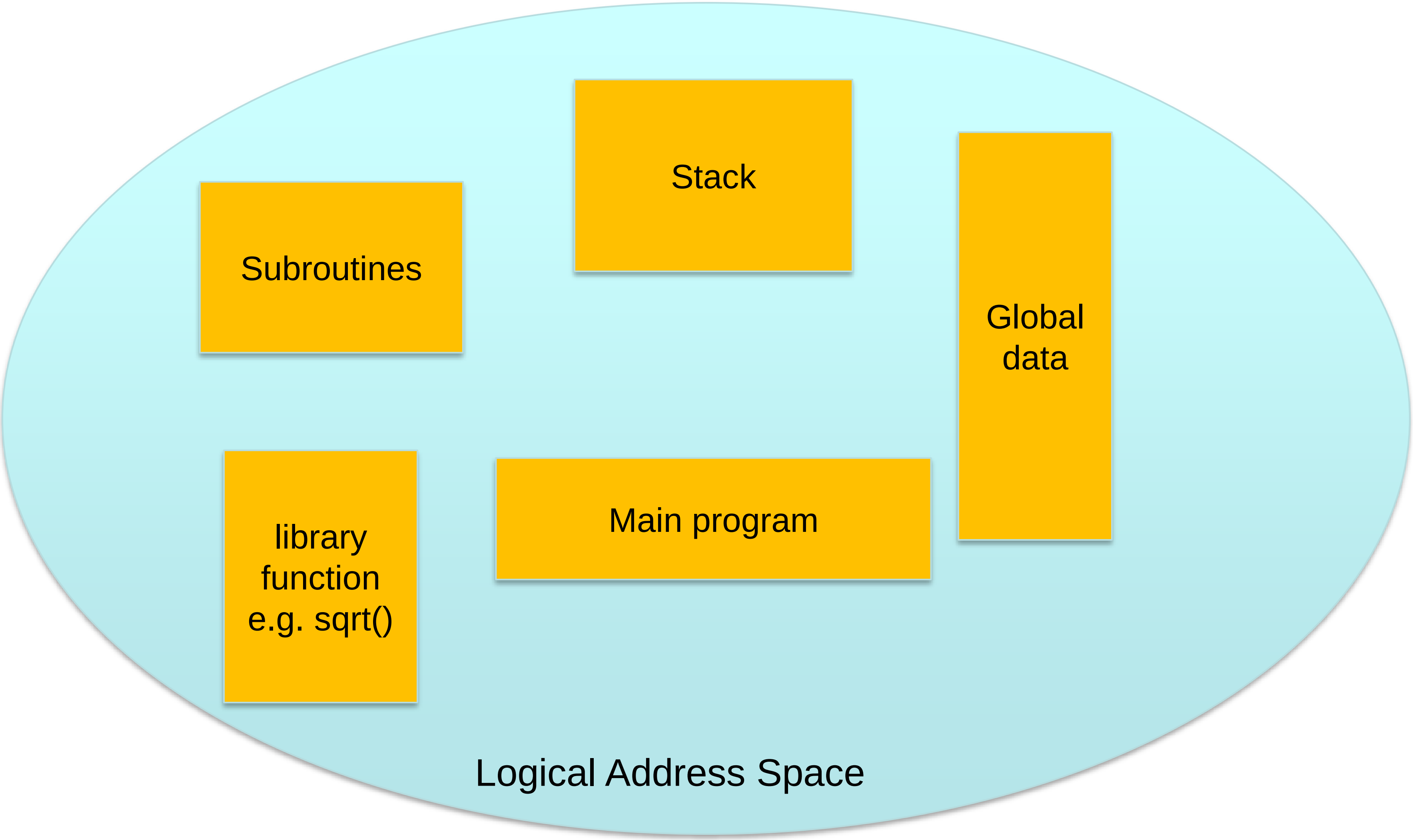


Fragmentation Problem

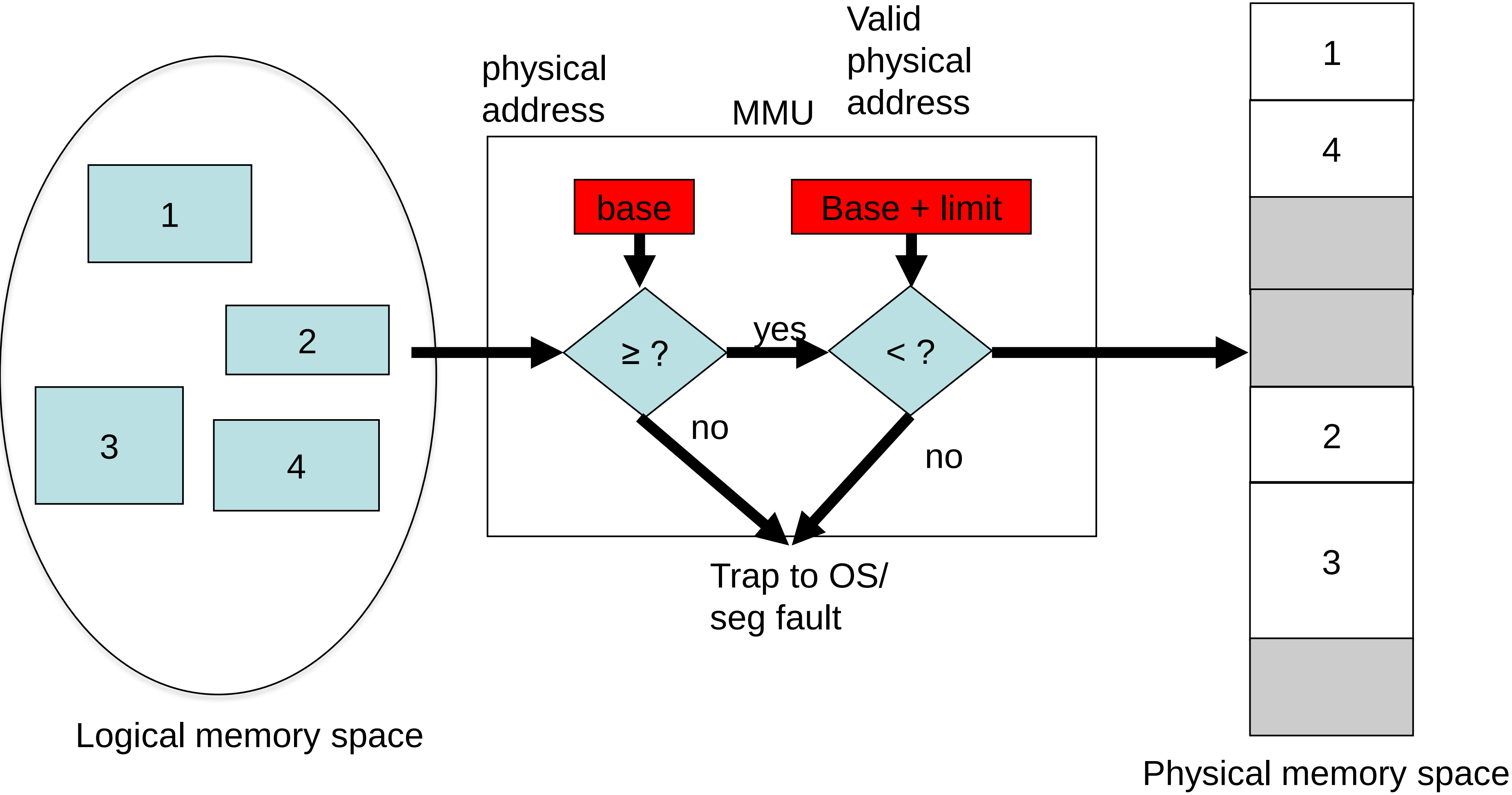
- This results in ***external fragmentation*** of main memory
  - There are many small chunks of non-contiguous unallocated memory between allocated processes in memory
- OS must find a large enough unallocated chunk in fragmented memory that a process will fit into
- De-fragmentation/Compaction
  - Algorithms for recombining contiguous segments is used, but memory still gets fragmented
  - Moving memory is expensive



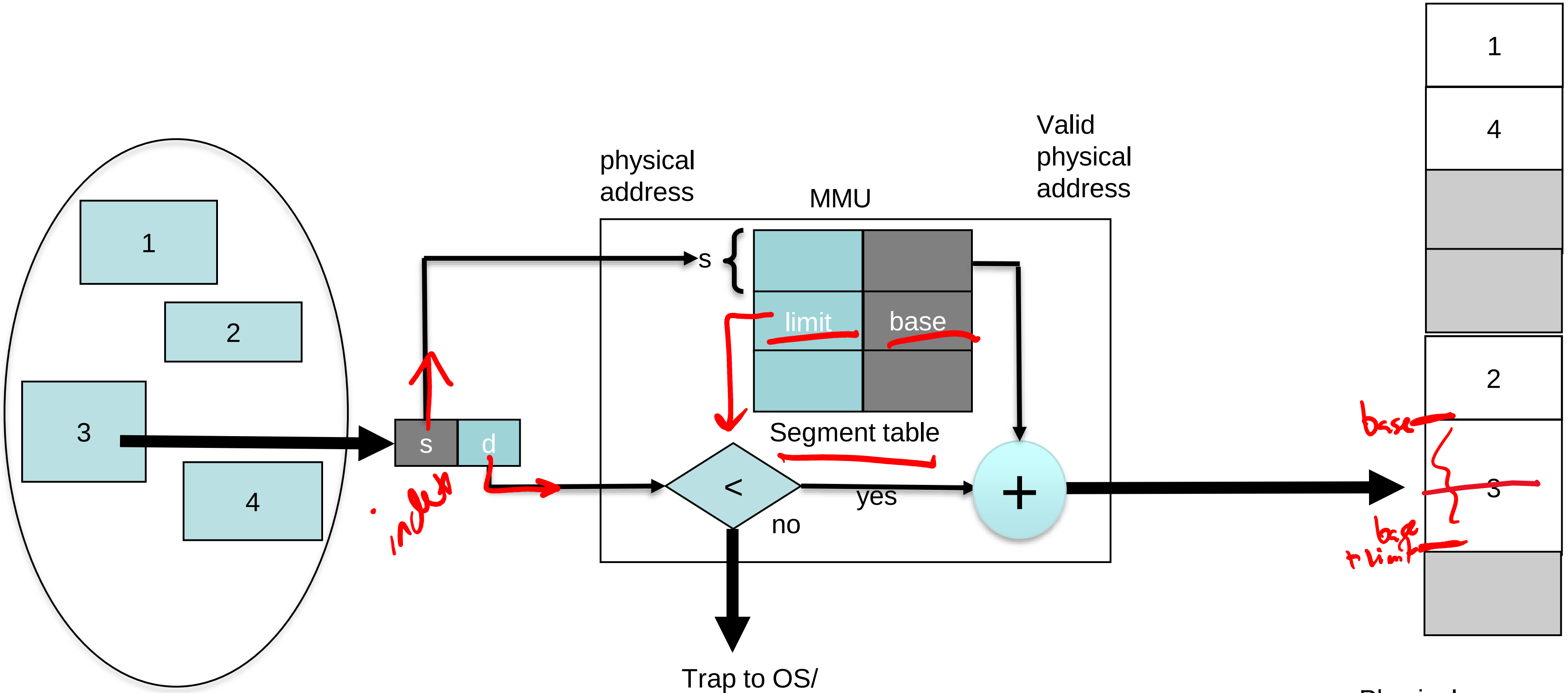
Instead of one big address space break it up into smaller segments



Segmentation



Segmentation of Memory



1

2

3

4

s

d

index

physical address

MMU

s

limit

base

Segment table

<

yes

no

+

Valid physical address

Trap to OS/
seg fault

1

4

2

3

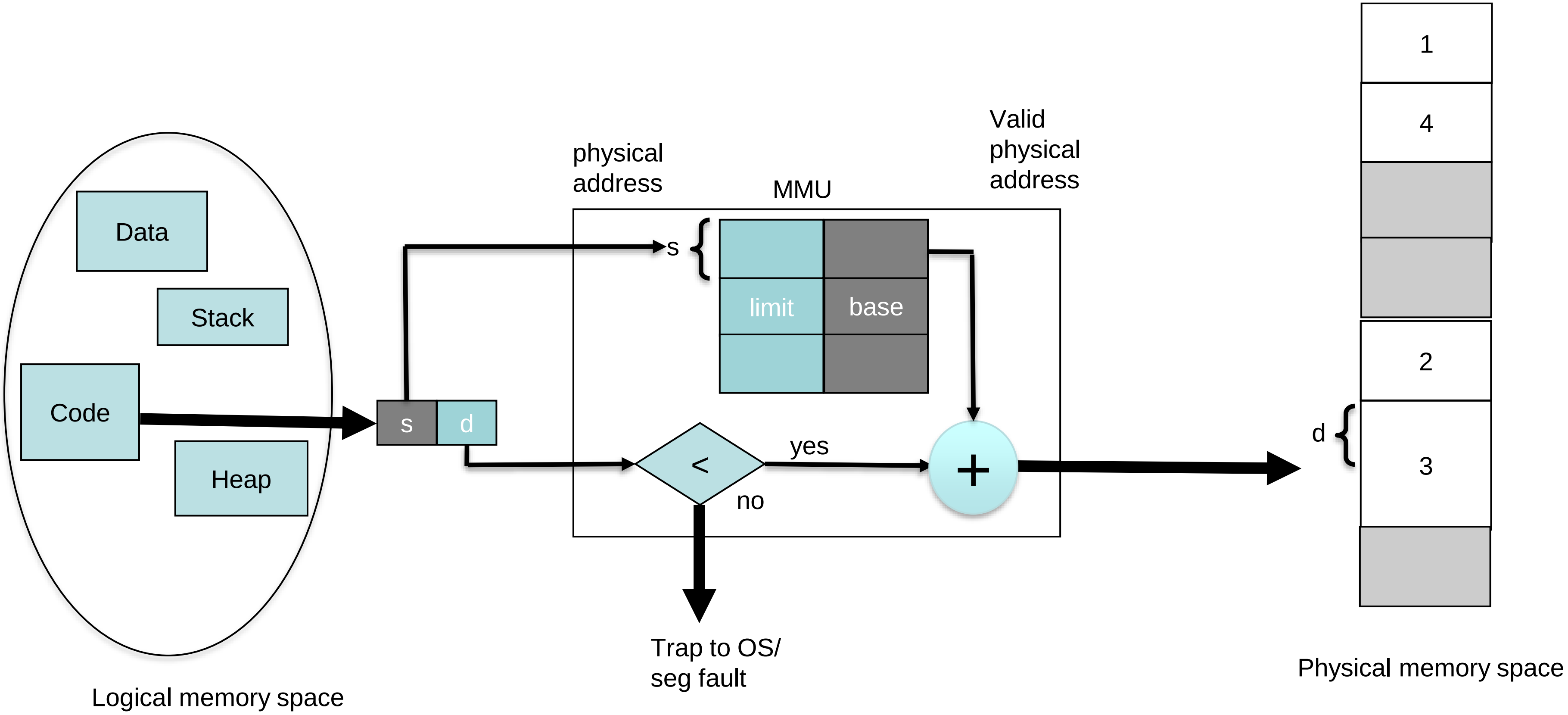
base

base + limit

Logical memory space

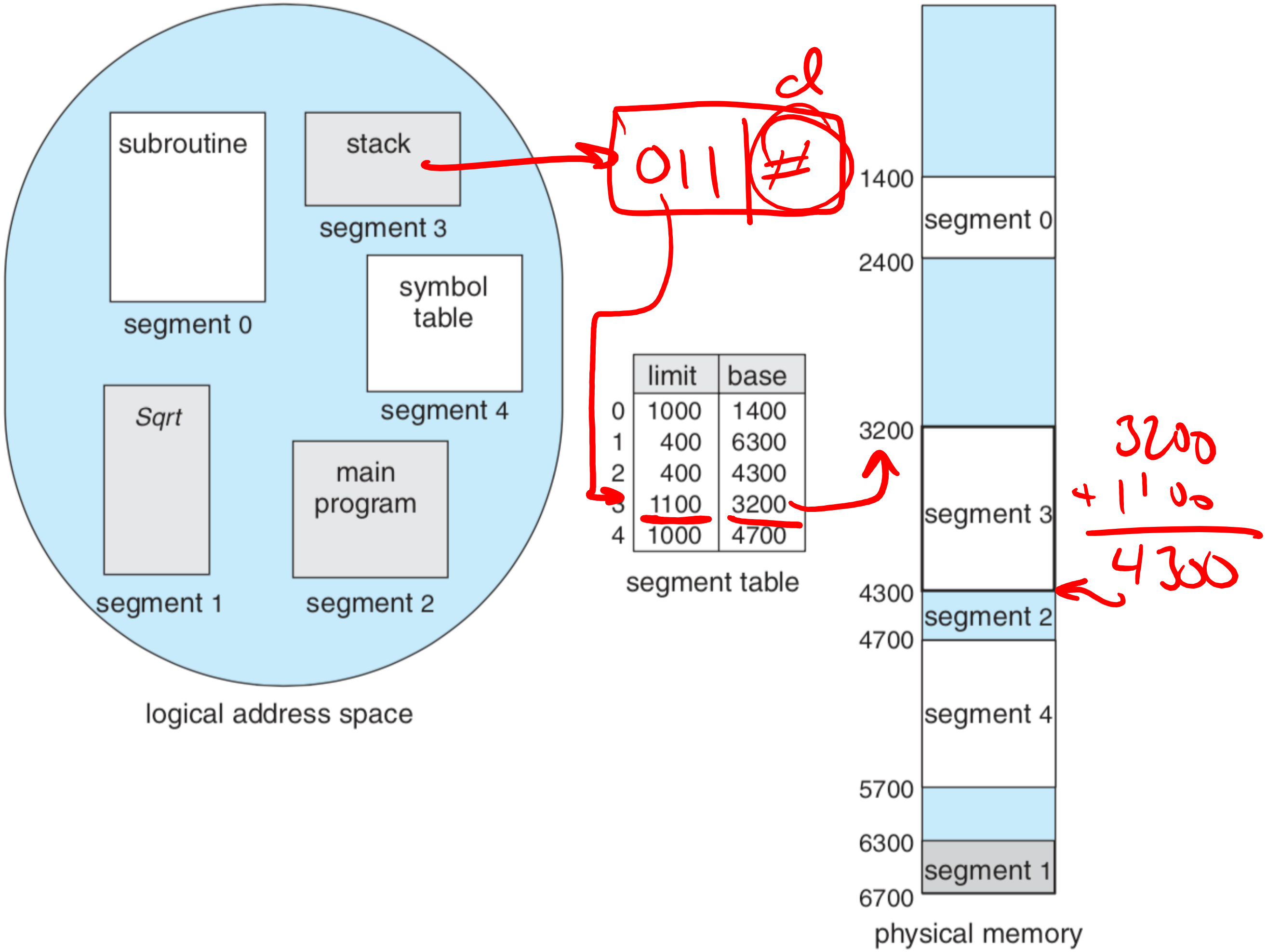
Physical memory space

Segmentation of Memory



Segmentation allows physical address of a process to be non-contiguous

Example

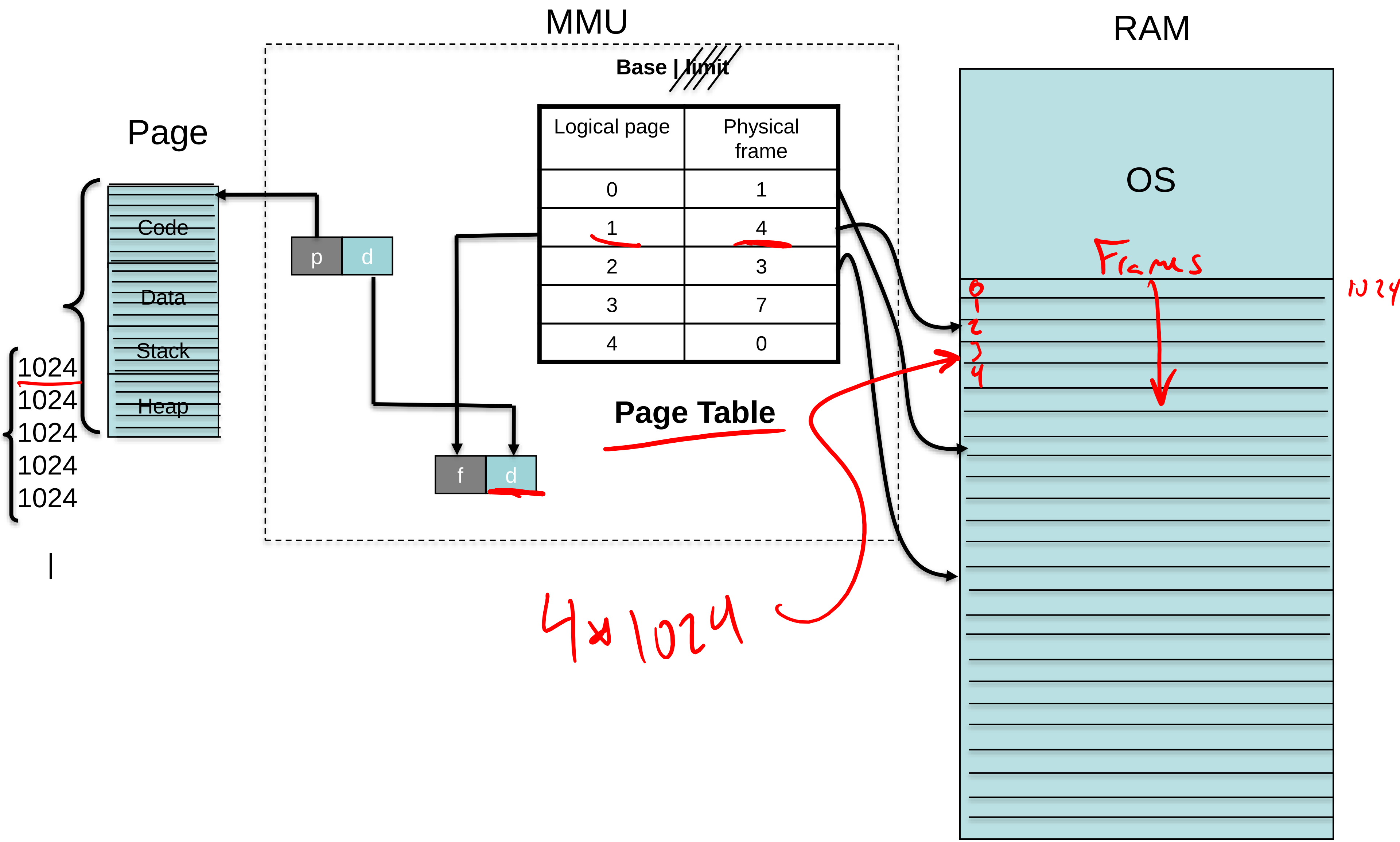


	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table

Segmentation allows physical address of a process to be non-contiguous

Better Solution to Reduce Fragmentation



Logical page	Physical frame
0	1
1	4
2	3
3	7
4	0

Paging

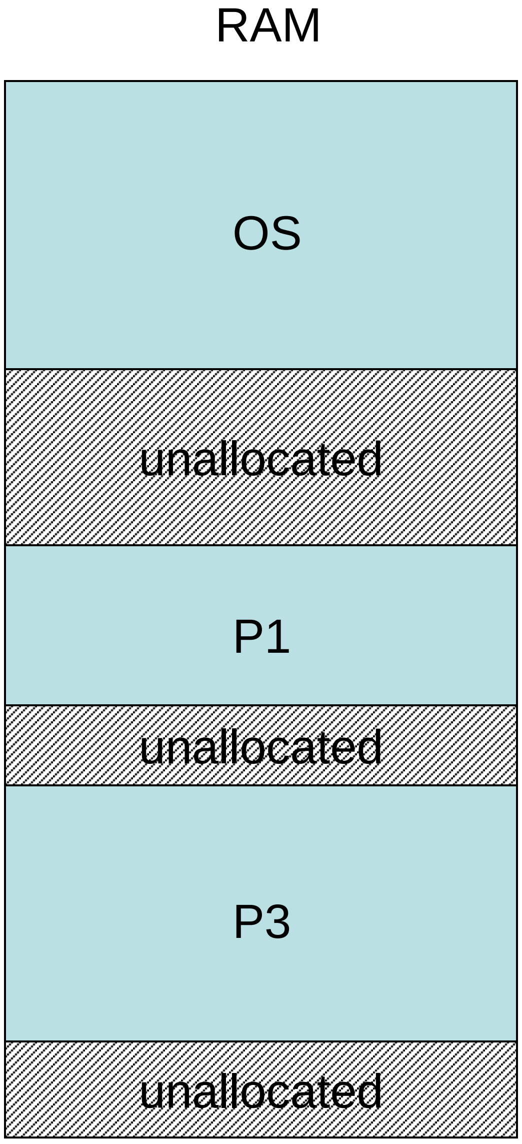
Avoiding External Fragmentation

- Over time, repeated allocation and deallocation will cause many small chunks of non-contiguous unallocated memory to form between allocated processes in memory

- Resulting in external fragmentation

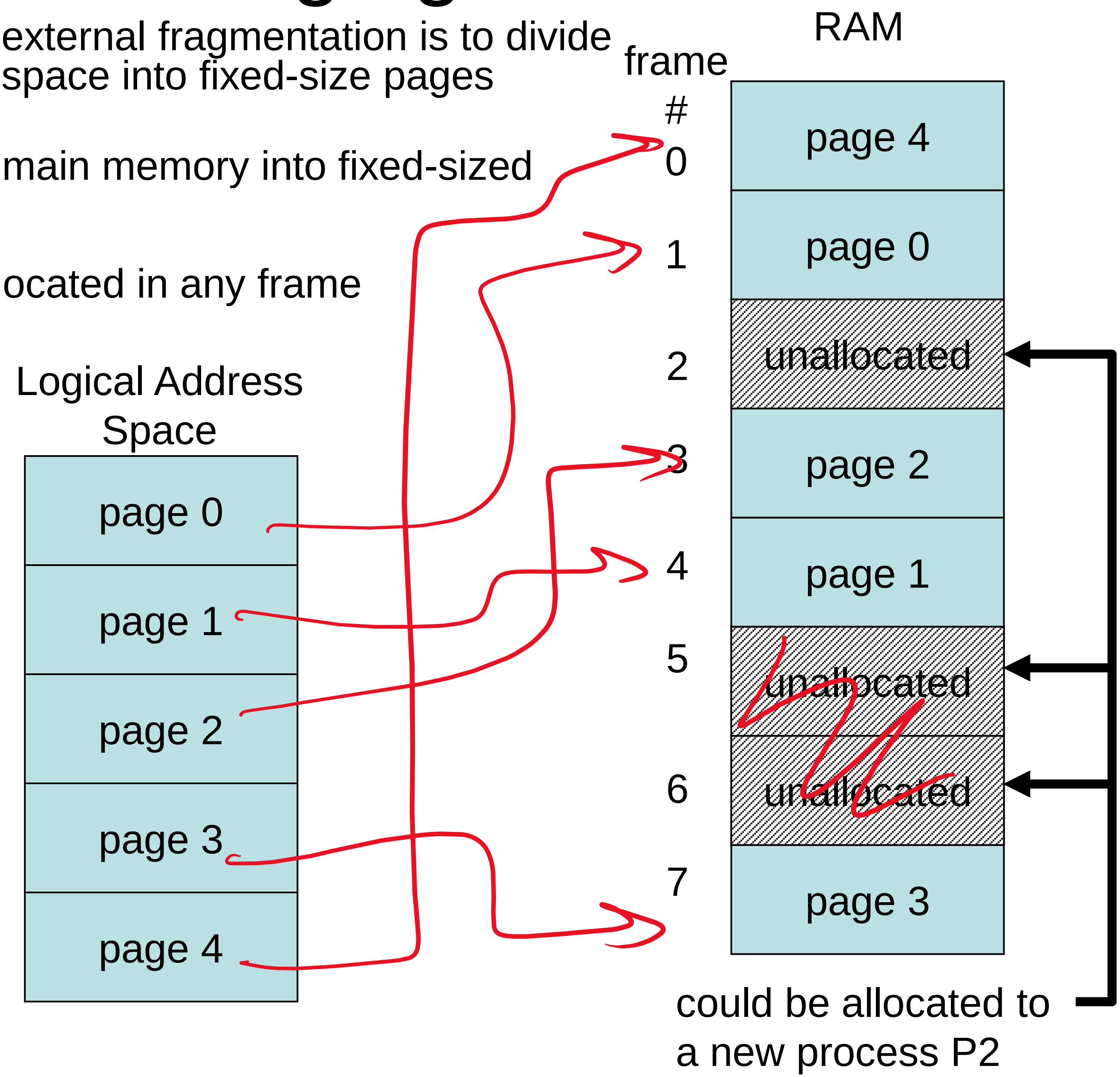
- OS must swap out current processes to create a large enough contiguous unallocated memory

- Could use de-fragmentation, but it is costly to move memory.



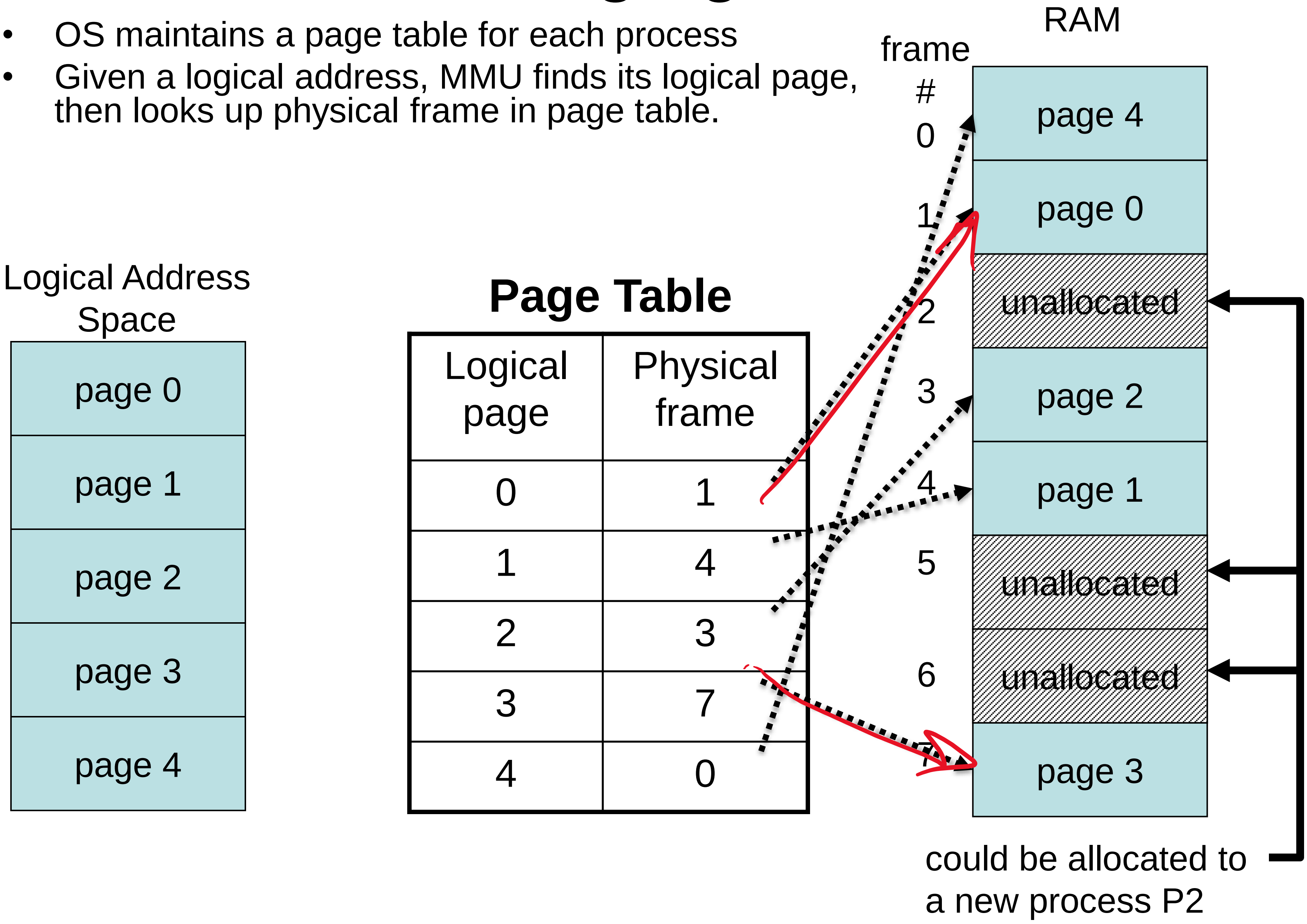
Paging

- A better solution to external fragmentation is to divide the logical address space into fixed-size pages
- We can also break main memory into fixed-sized frames
- Each page can be located in any frame



Paging

- OS maintains a page table for each process
- Given a logical address, MMU finds its logical page, then looks up physical frame in page table.

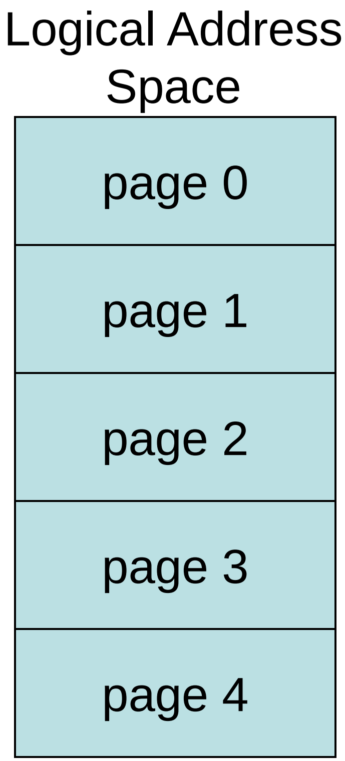


Page Table

Logical page	Physical frame
0	1
1	4
2	3
3	7
4	0

could be allocated to a new process P2

Paging



Logical Address Space

page 0

page 1

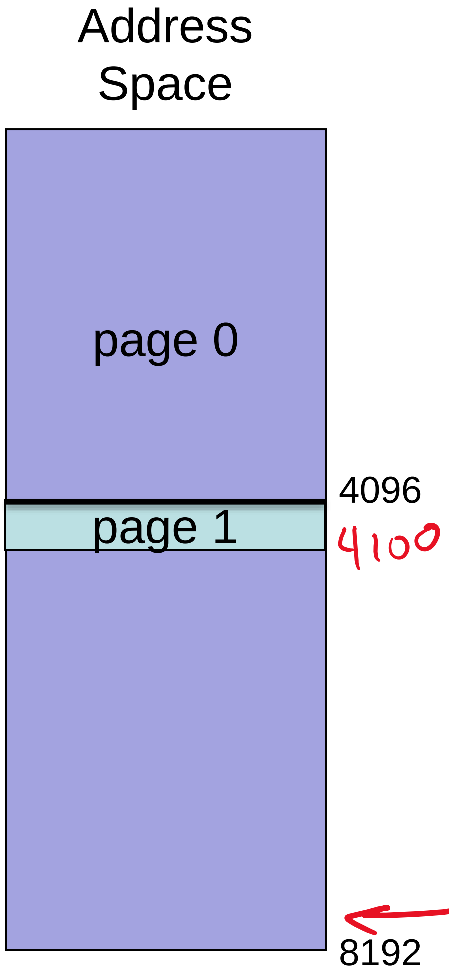
page 2

page 3

page 4

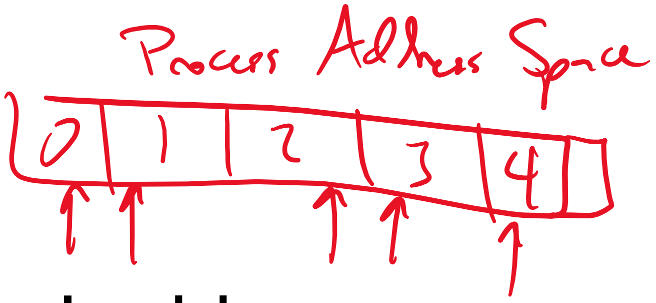
- Advantage: User’s view of memory is still as one contiguous block of logical address space
  - – MMU performs run-time mapping of each logical address to a physical address using the page table
- Typical page size is 4-8 KB
  - – Example: a 4 GB 32-bit address space with 4 KB/page (2^{12})
    => $2^{32}/2^{12}$ = 1 million entries in page table
    - Your page table would need to be >= 20 bits/ table entry (~1 MB per process)

Paging



- No external fragmentation
- But we do get some ***internal fragmentation***
  - example: suppose my process is size 4100 Bytes, and each page size is 4 KB (4096 Bytes)
    - then I have to allocate two pages = 8 KB,
    - 3999 B of 2nd page is wasted due to fragmentation that is internal to a page
- OS also has to maintain a frame table/pool that keeps track of what physical memory frames are free

Paging

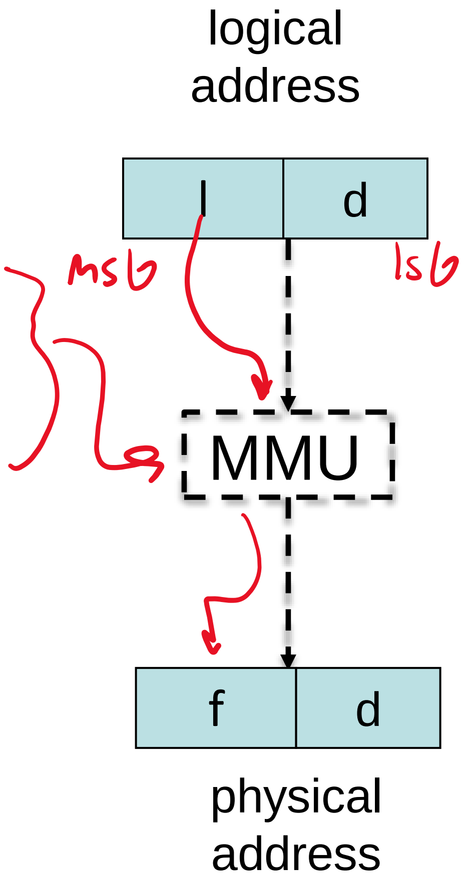


- Conceptually, every logical/virtual address can now be divided into two parts:

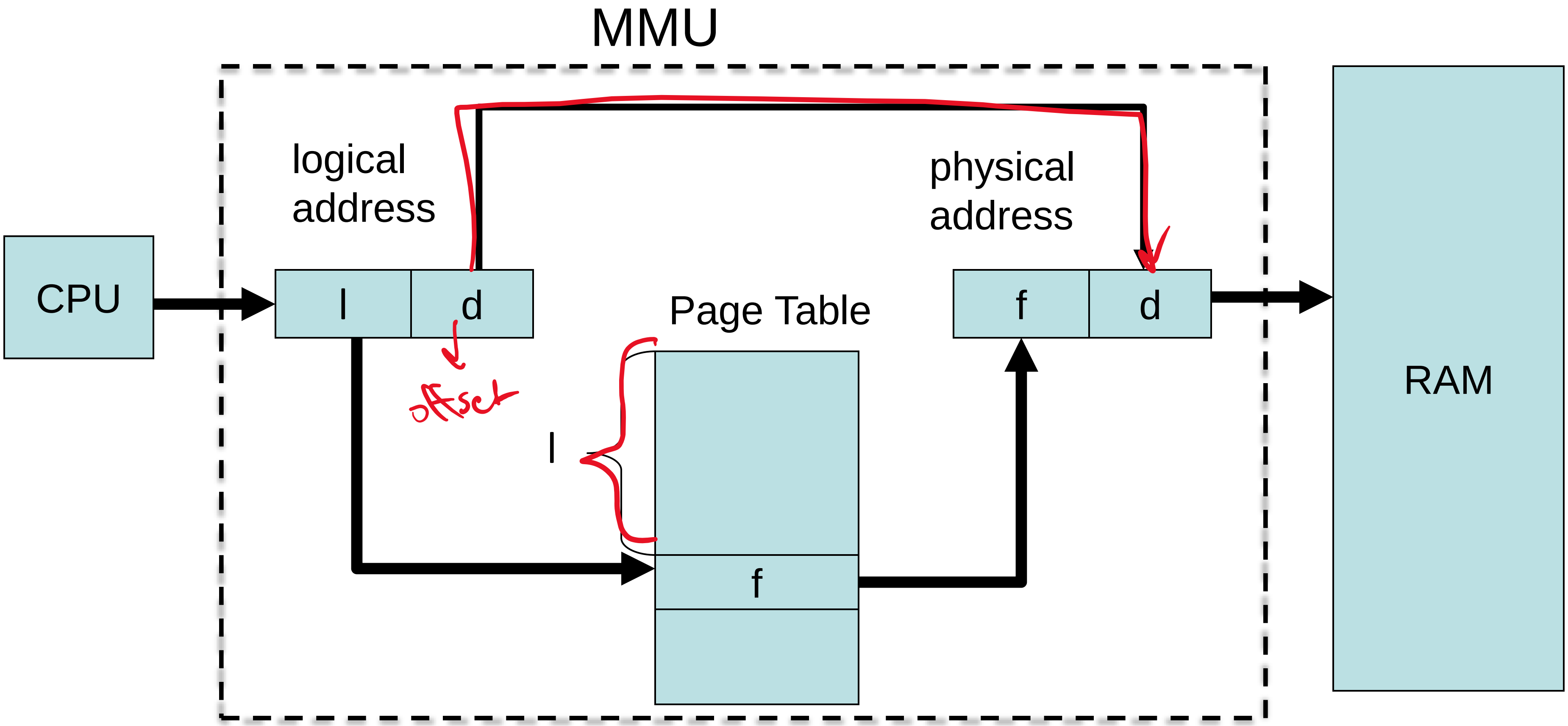
  Logical page number & Offset

  – most significant bits = logical page # ***L***,
    - Equals the virtual address / page size
    - used to index into page table to retrieve the corresponding physical frame *f*

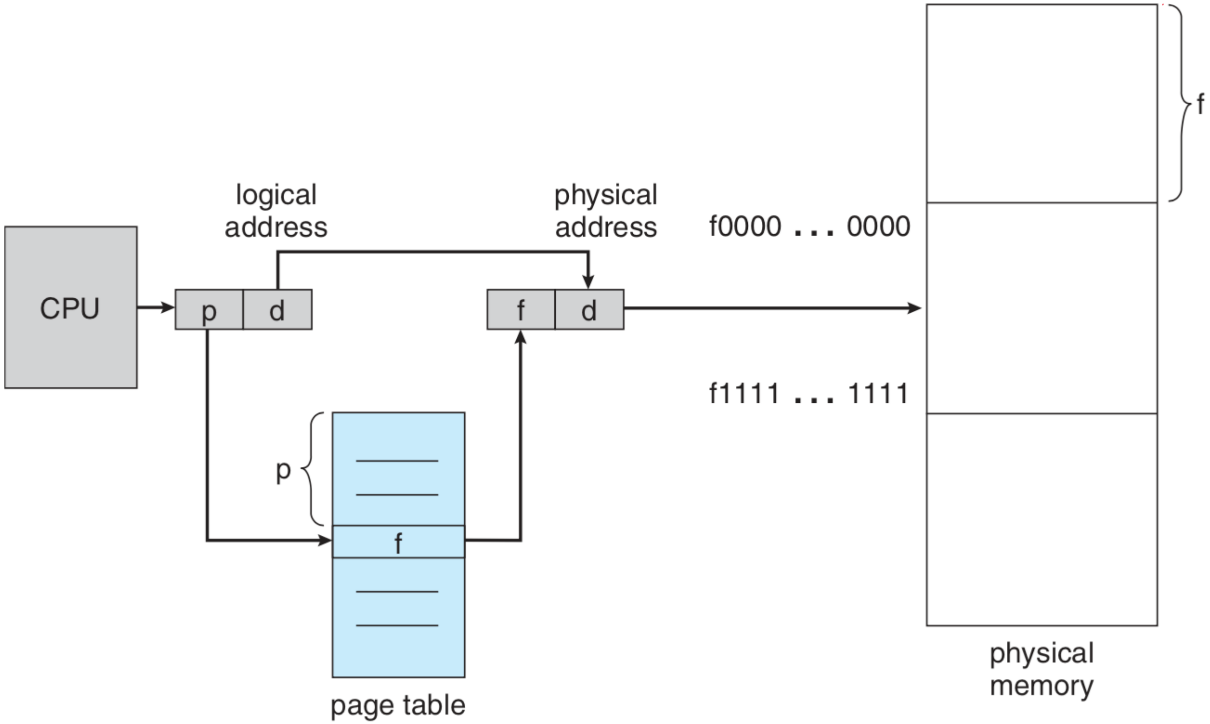
  – least significant bits = page offset ***d***

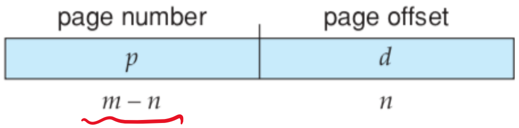


Paging


MMU
CPU
logical address
l
d
offset
Page Table
l
f
physical address
f
d
RAM

Paging

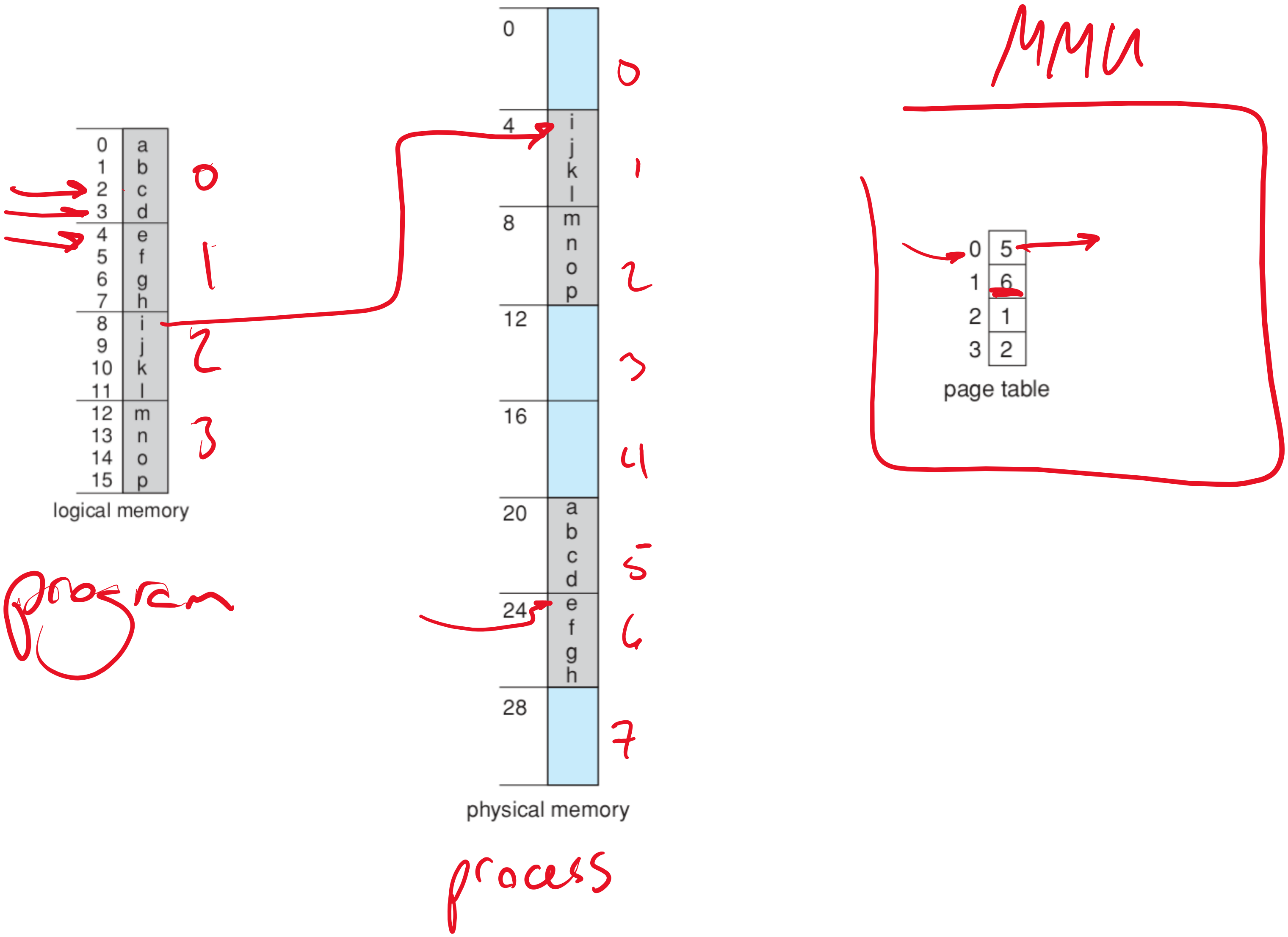




Size of the logical address space is 2^m Size of each page is 2^n

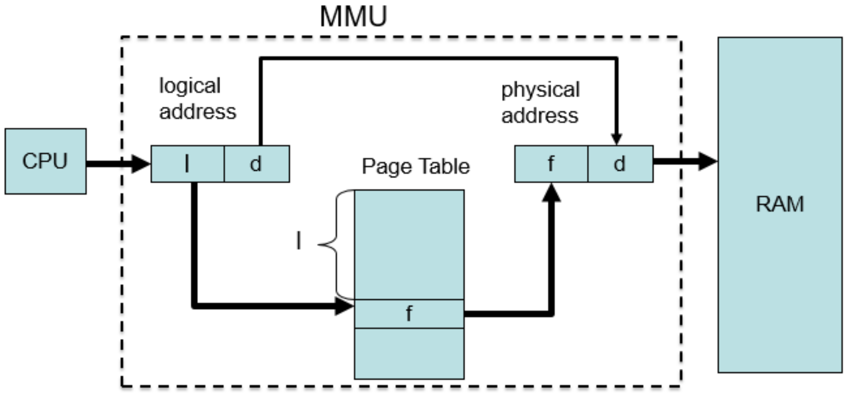
if $m = 32$, $n = 12$, $m - n = 20$

Paging


0 a
1 b
2 c
3 d
4 e
5 f
6 g
7 h
8 i
9 j
10 k
11 l
12 m
13 n
14 o
15 p
logical memory
0
1
2
3
program
0
4 i j k l
8 m n o p
12
16
20 a b c d
24 e f g h
28
physical memory
0
1
2
3
4
5
6
7
process
MMU
0 5
1 6
2 1
3 2
page table

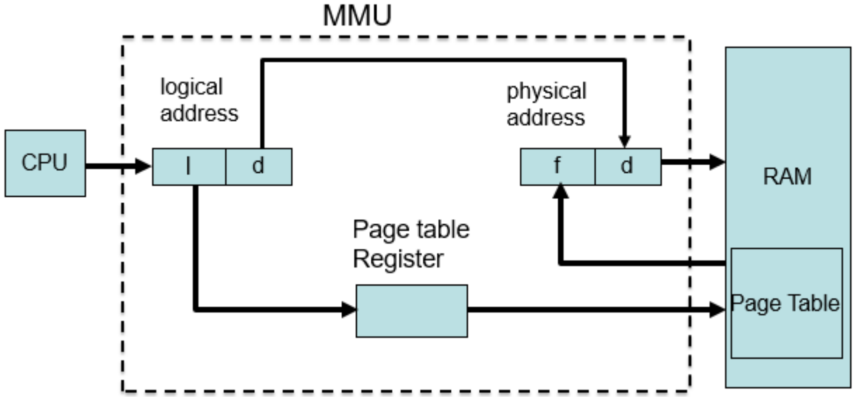
Paging

- Implementing a page table:



  - option #1: use dedicated bank of hardware registers or memory to store the page table
    - fast per-instruction translation
    - slow per context switch because the entire page table has to be reloaded for the new process
    - limited by cost (expensive hardware) to being too small - some page tables can be large, e.g. 1 million entries – too expensive

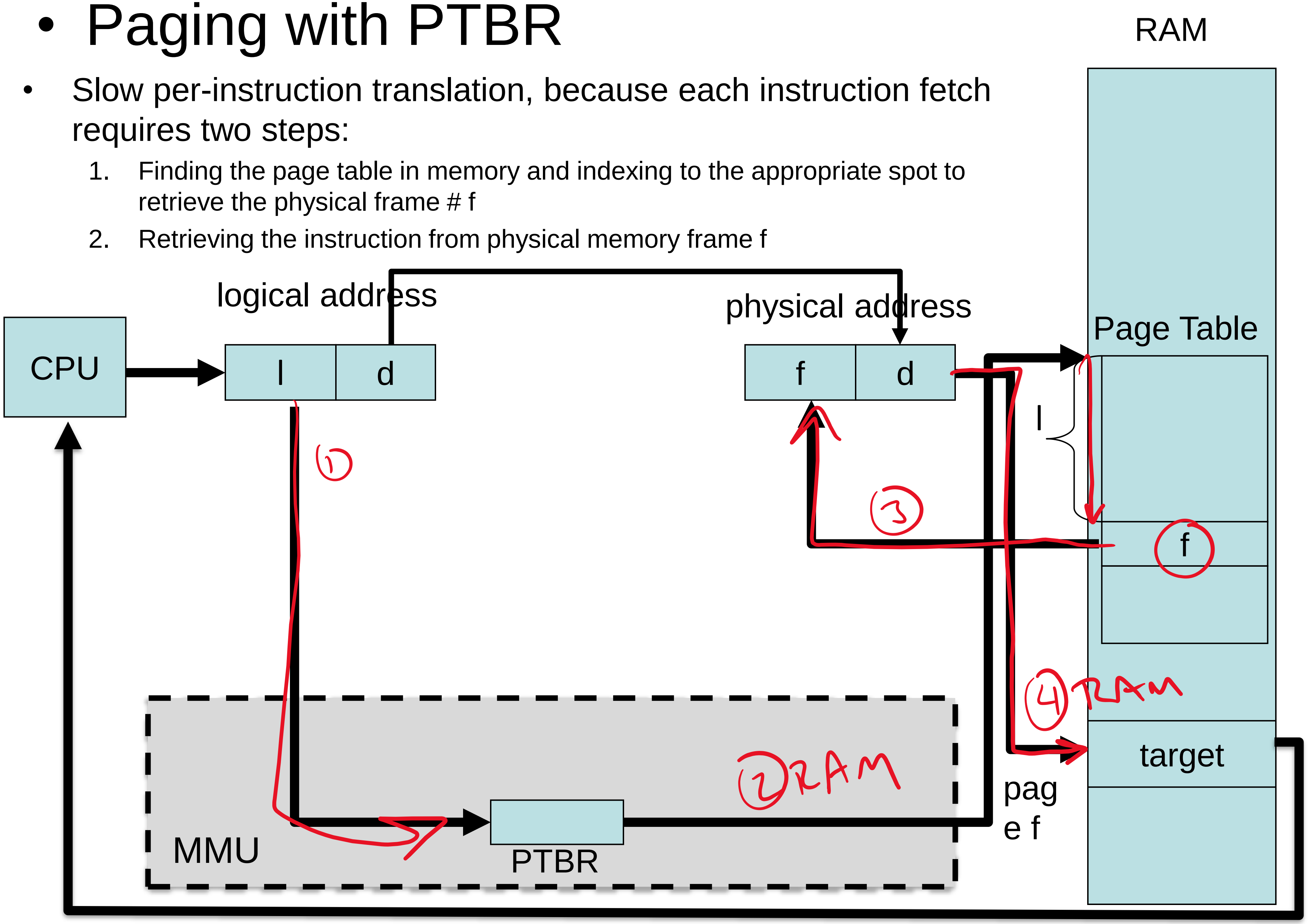
Implementing a page table



- option #2: store the page table in main memory
  - keep a pointer to the page table in a special CPU register called the Page Table Base Register (PTBR)
  - can accommodate fairly large page tables
  - fast context switch - only reload the PTBR!
  - slow per-instruction translation, because each instruction fetch requires *two steps memory access*:
    1. finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
    2. retrieving the instruction from physical memory frame f

Paging with PTBR

- Slow per-instruction translation, because each instruction fetch requires two steps:
  1. Finding the page table in memory and indexing to the appropriate spot to retrieve the physical frame # f
  2. Retrieving the instruction from physical memory frame f

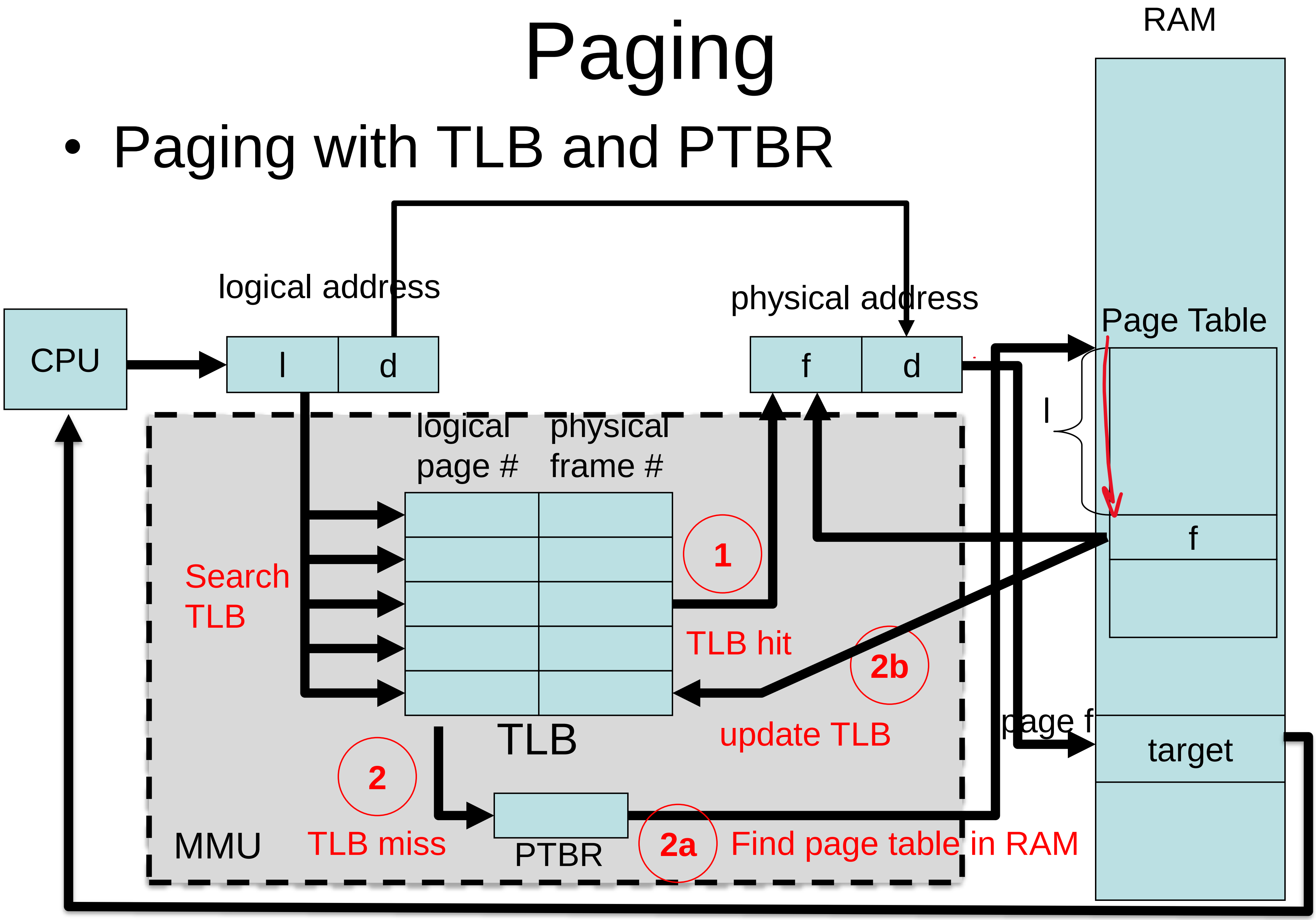


Implementing a page table

- Option #3: cache a subset of page table mappings/entries in a small set of CPU buffers called Translation-Look-aside Buffers (TLBs)
  - Fast solution to option #2's slow two-step memory access
  - Several TLB caching policies:
    - Cache the most popular or Most frequently referenced pages in TLB
    - Cache the most recently used pages

Paging

- Paging with TLB and PTBR



Return fetched code/data @ offset d inside page f

Paging and TLB Caching

- Summarize steps depicted in the graph on the last slide
- MMU in CPU first looks in TLB's to find a match for a given logical address
  1. if match found, then quickly call main memory with physical address frame f (plus offset d)
     - this is called a *TLB hit*
     - TLB as implemented in hardware does a fast parallel match of the input page to all stored values in the cache - about 10% overhead in speed

Paging and TLB Caching

2. if no match found, then this is a *TLB miss*
    a) go through regular two-step lookup procedure: go to main memory to find page table and index into it to retrieve frame #f, then retrieve what's stored at address <f,d> in physical memory
    b) Update TLB cache with the new entry from the page table
        - if cache full, then implement a cache replacement strategy, e.g. Least Recently Used (LRU) - we'll see this later

- Goal is to maximize TLB hits and minimize TLB misses

Paging and TLB Caching

- **On a context switch, the TLB entries would typically have to all be invalidated/completely flushed**
  - since different processes have different page tables
  - E.g. x86 invalidate all caches after CSwitch
- **An alternative is to include process IDs in TLB**
  - at the additional cost of hardware and an additional comparison per lookup
  - Only *TLB entries with a process ID* matching the current task are considered valid
  - E.g. DEC RISC Alpha CPU

Paging and TLB Caching

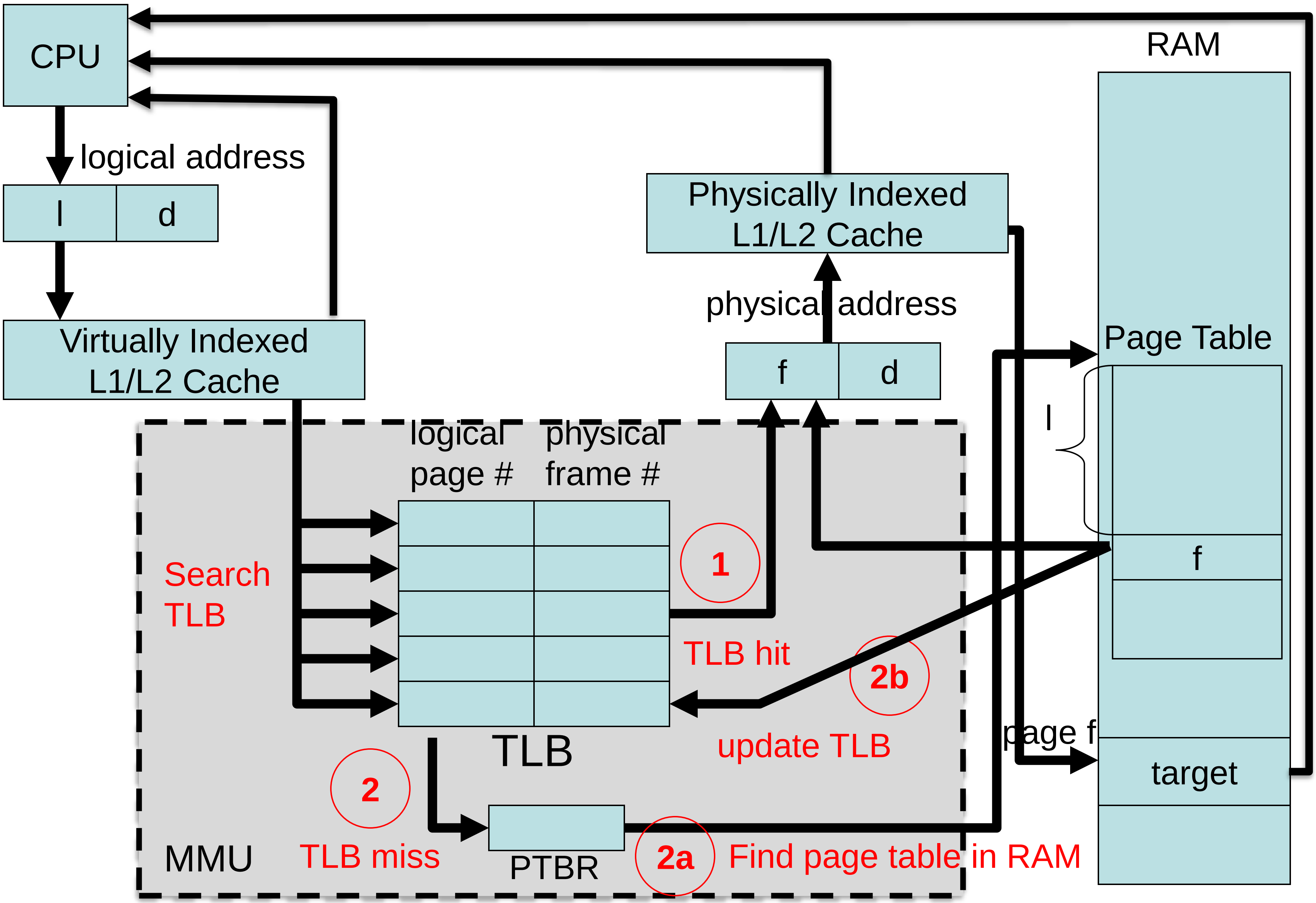
- For frequently ran processes: **Prevent frequently used pages** from being automatically invalidated in the TLBs on a context switch
  - In Intel Pentium Pro, use the **page global enable (PGE) flag** in the register CR4 and the global (G) flag of a page-directory or page-table entry
  - ARM allows flushing of individual entries from the TLB indexed by virtual address

Paging and L1/L2 Caching

- **How does MMU interact with L1 or L2 data or instruction caches?**
  - It depends on whether the items in a cache are indexed as logical (virtual) or physical (for look up purposes)

- L1/L2 data/instruction caches can store their information and be **indexed by either virtual or physical addresses**
  - If physical, then MMU must first convert virtual to physical, before the cache can be consulted – this is slow, but each entry is uniquely identifiable by its physical address
  - If virtual, then cache can be consulted quickly to see if there's a hit without invoking the MMU (if miss, then MMU must still be invoked…)

- Paging with TLB and L1/L2 Caching



Paging and L1/L2 Caching

- **A virtually indexed L1/L2 data/instruction caches introduces some problems:**
  - Homonym problem: when a new process is switched in, it may use the same virtual address V as the previous process.
    - The cache that indexes just by virtual address V will return the wrong information (cached information from the prior process).

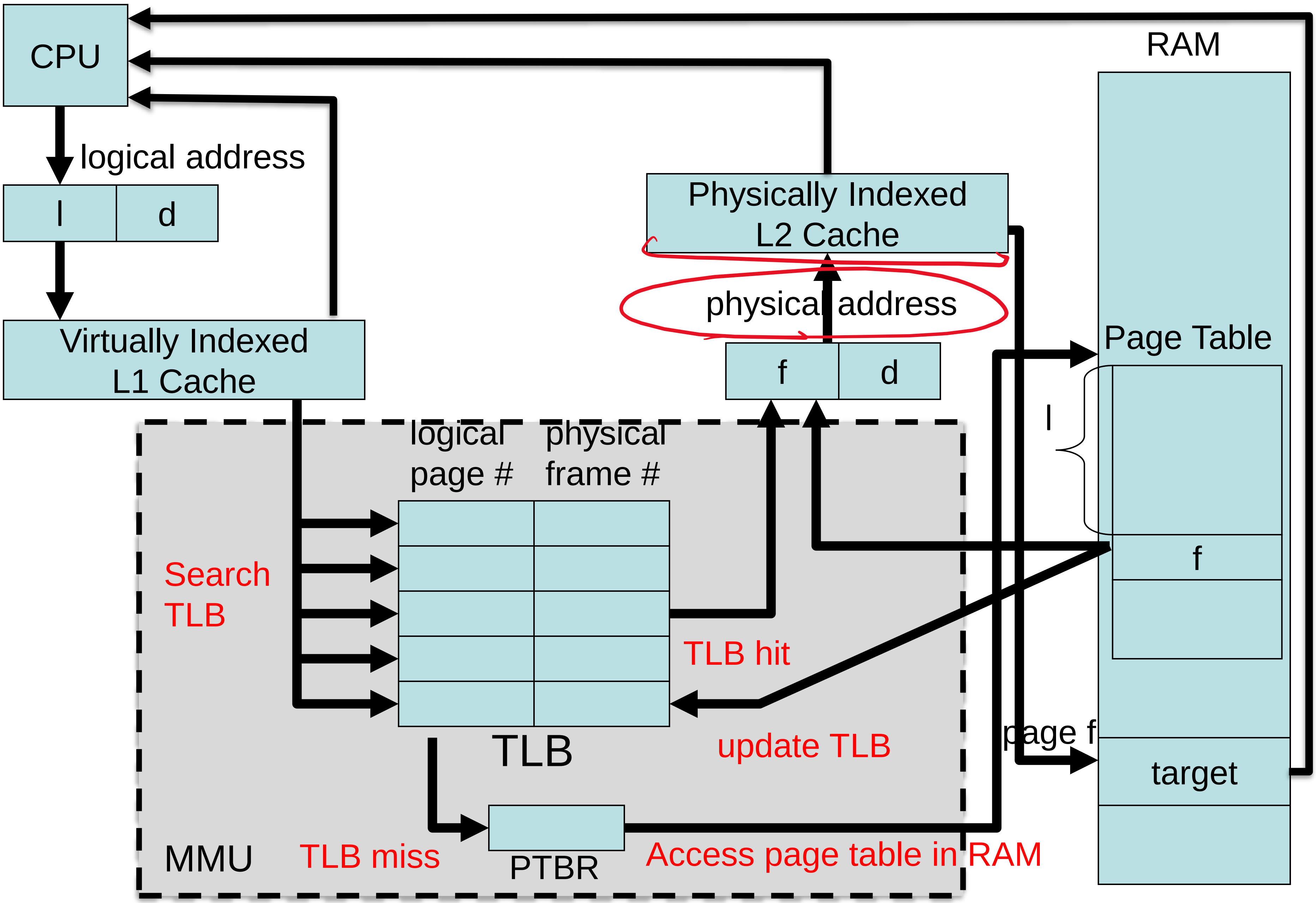
Paging and L1/L2 Caching

- Some solutions to the homonym problem:
  - 1 - Flush the cache on each context switch
    - process gets entire cache to itself, but have to rebuild cache
  - 2 - Add an address space id (process id) to each entry of the cache
    - so only data/instructions for the right process are returned for a given virtual address V
    - requires hardware support and an extra comparison
    - reduces available cache space for each process, since it has to be shared
  - 3 - Each process uses non-overlapping virtual addresses in its address space
    - unlikely, violates model that each process is compiled & executes independently in its own address space [0,MAX]
    - Violating the virtual space principle – the independence from other processes.

Paging and L1/L2 Caching

- **In practice,**
  - Most L1 caches are virtually indexed – fast
  - Most L2 caches are physically indexed
    - Each entry is unique
    - No collisions
    - This is good for code/data from shared library pages, i.e. if multiple processes share the same code/data, then it just has to be stored once in cache
  - The virtually indexed cache is essentially a small L1 cache, and the physically indexed cache is a much larger L2 cache.

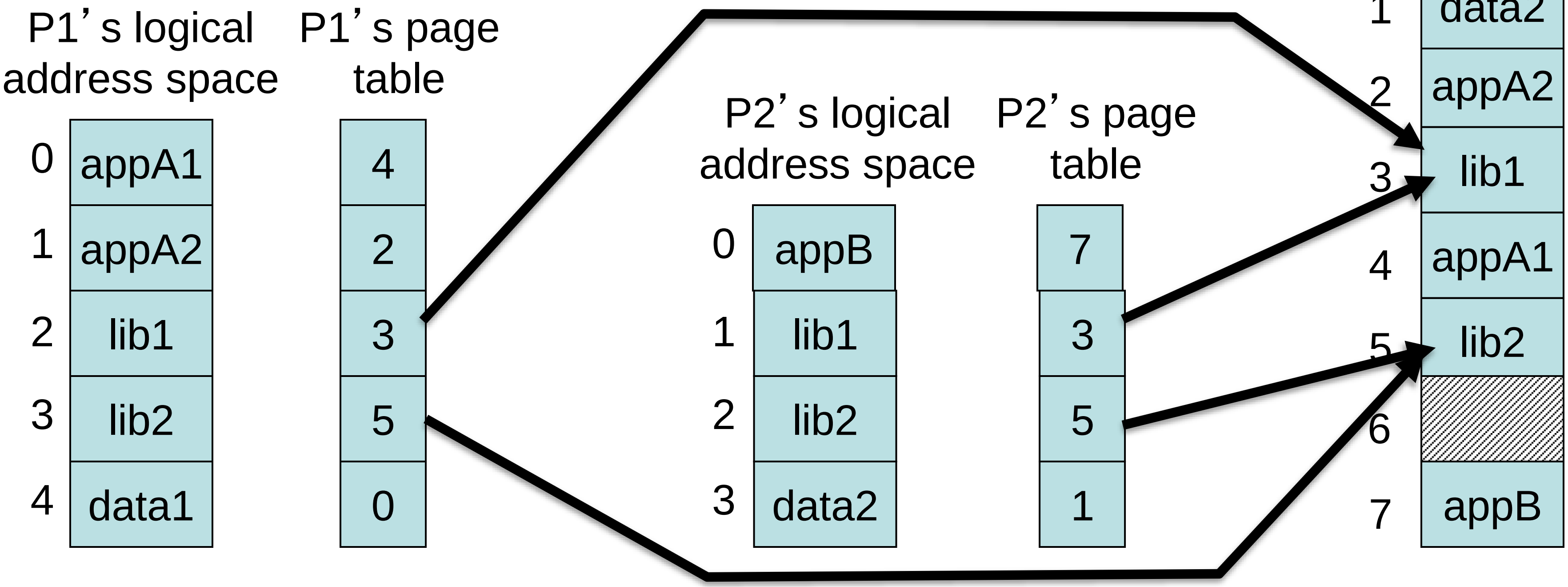
- Paging with TLB and L1/L2 Caching



Shared Pages

- Page tables can point to the *same* memory frames
  - e.g.: consider 2 app's A and B, running in 2 processes P1 and P2, sharing some C library functions consisting of 2 pages of code, lib1 and lib2, and each process has its own data

RAM



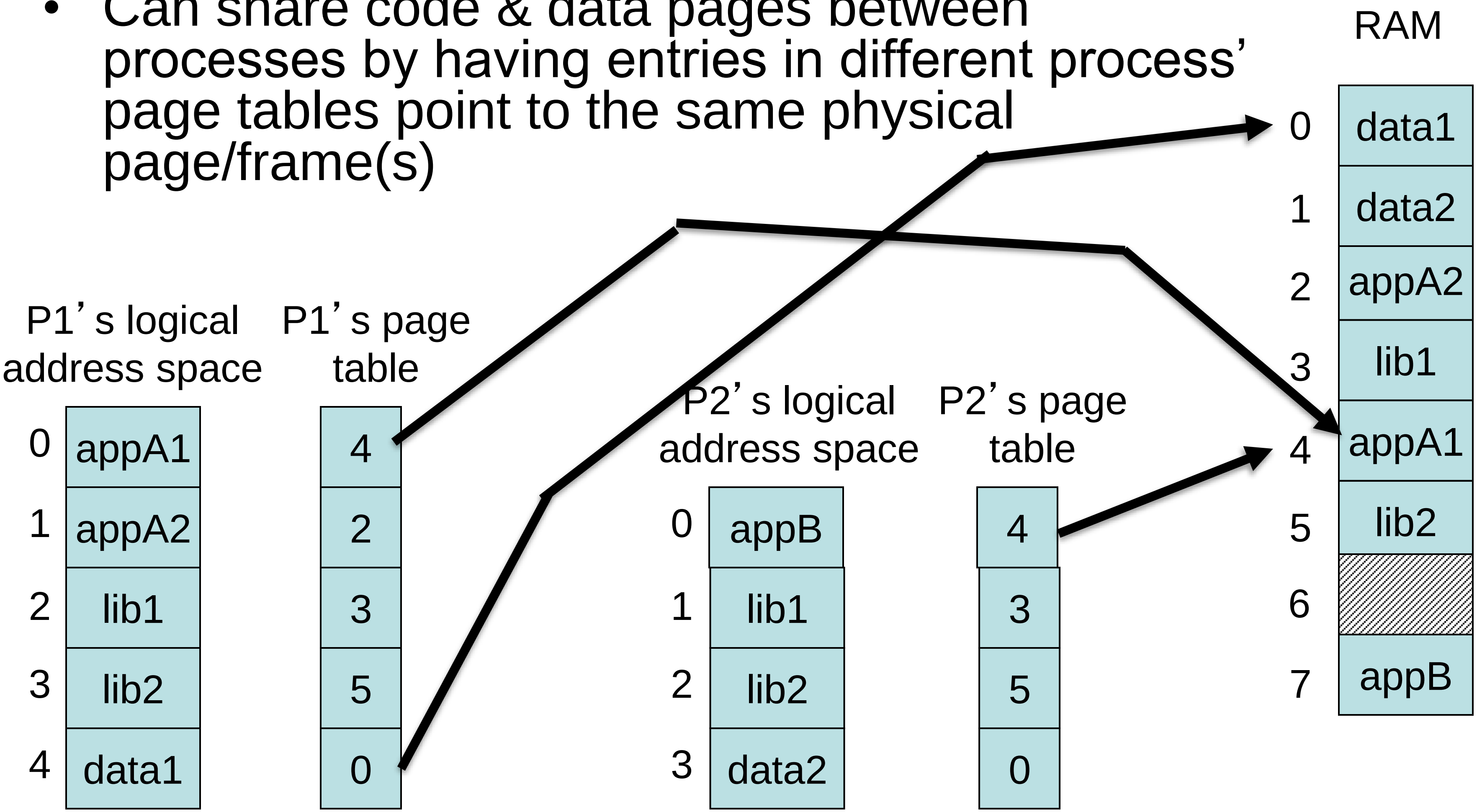
	P1's logical address space	P1's page table
0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0

	P2's logical address space	P2's page table
0	appB	7
1	lib1	3
2	lib2	5
3	data2	1

	RAM
0	data1
1	data2
2	appA2
3	lib1
4	appA1
5	lib2
6	▨
7	appB

Shared Pages

- Can share code & data pages between processes by having entries in different process’ page tables point to the same physical page/frame(s)



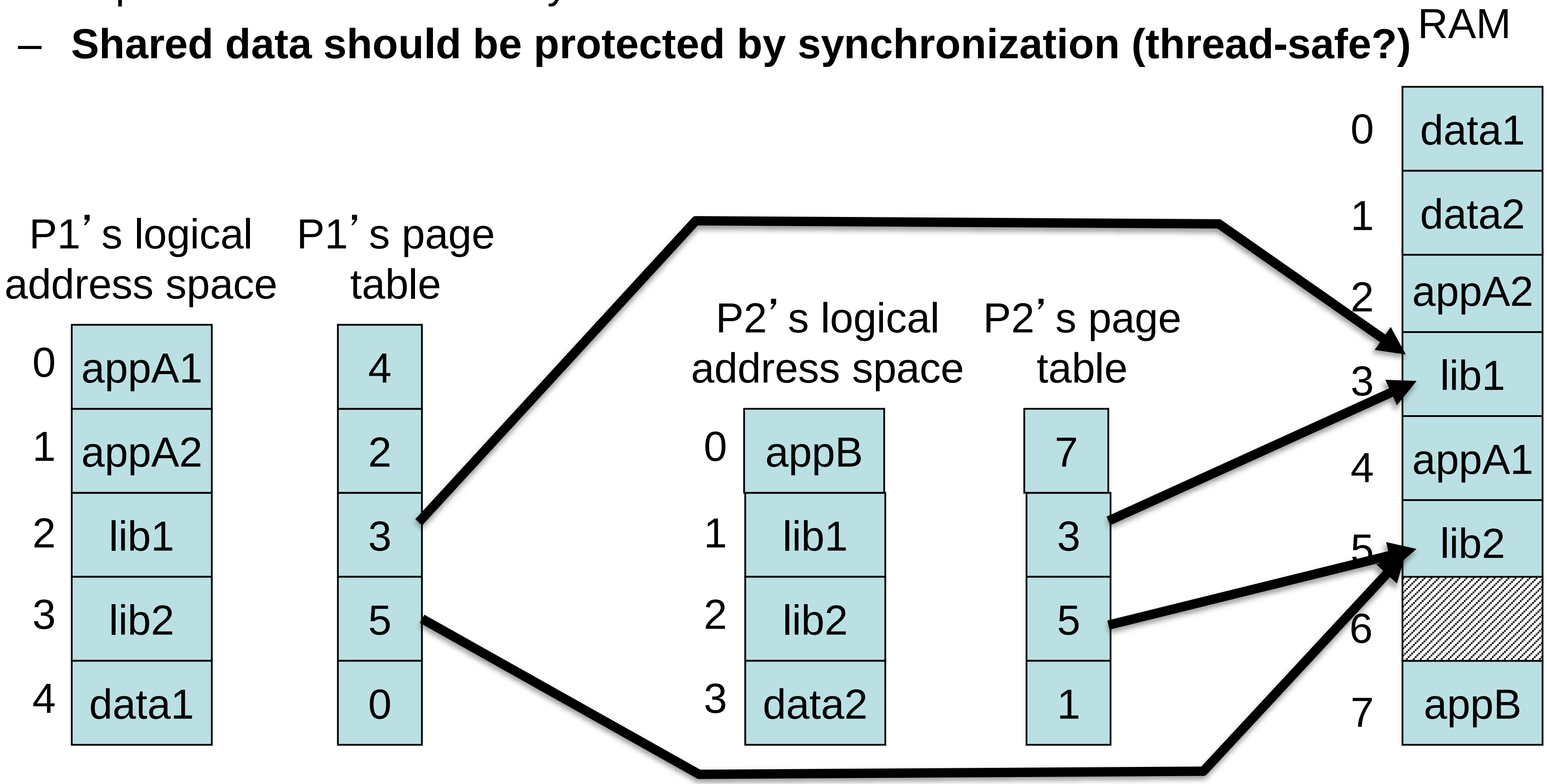
	P1’s logical address space	P1’s page table
0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0

	P2’s logical address space	P2’s page table
0	appB	4
1	lib1	3
2	lib2	5
3	data2	0

	RAM
0	data1
1	data2
2	appA2
3	lib1
4	appA1
5	lib2
6	▨
7	appB

Shared Pages

- Sharing data:
  - Two or more processes may want to share memory between them, so pointing multiple page tables to the same data pages is a way to implement shared memory
  - **Shared data should be protected by synchronization (thread-safe?)**



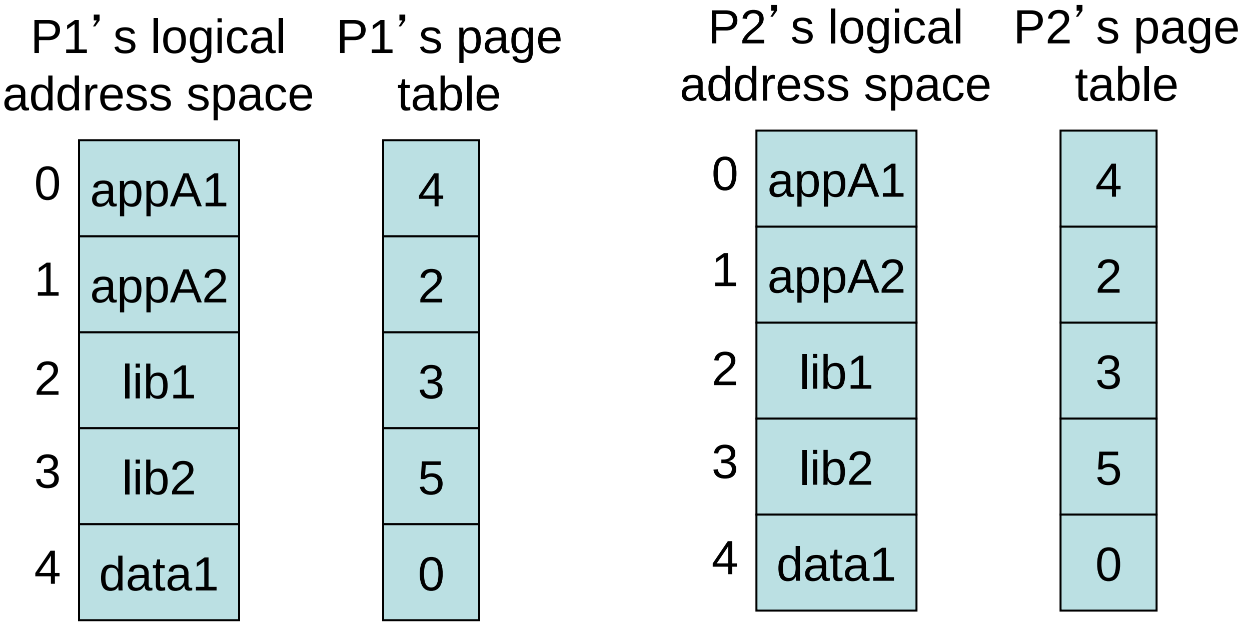
	P1’s logical address space	P1’s page table
0	appA1	4
1	appA2	2
2	lib1	3
3	lib2	5
4	data1	0

	P2’s logical address space	P2’s page table
0	appB	7
1	lib1	3
2	lib2	5
3	data2	1

	RAM
0	data1
1	data2
2	appA2
3	lib1
4	appA1
5	lib2
6	▨
7	appB

Shared Pages

- Fork()' ing a child process causes the child to have a copy of the entire address space of the parent, including code
  - Rather than duplicating all such code pages, can simply map the child's page to the point to the same set of code pages as the parent
  - This is a way to implement copy-on-write



P1' s logical address space

0	appA1
1	appA2
2	lib1
3	lib2
4	data1

P1' s page table

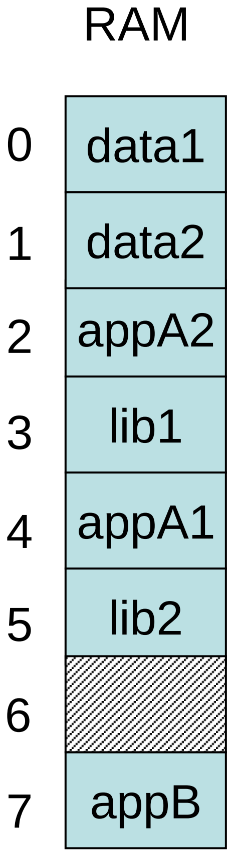
4
2
3
5
0

P2' s logical address space

0	appA1
1	appA2
2	lib1
3	lib2
4	data1

P2' s page table

4
2
3
5
0

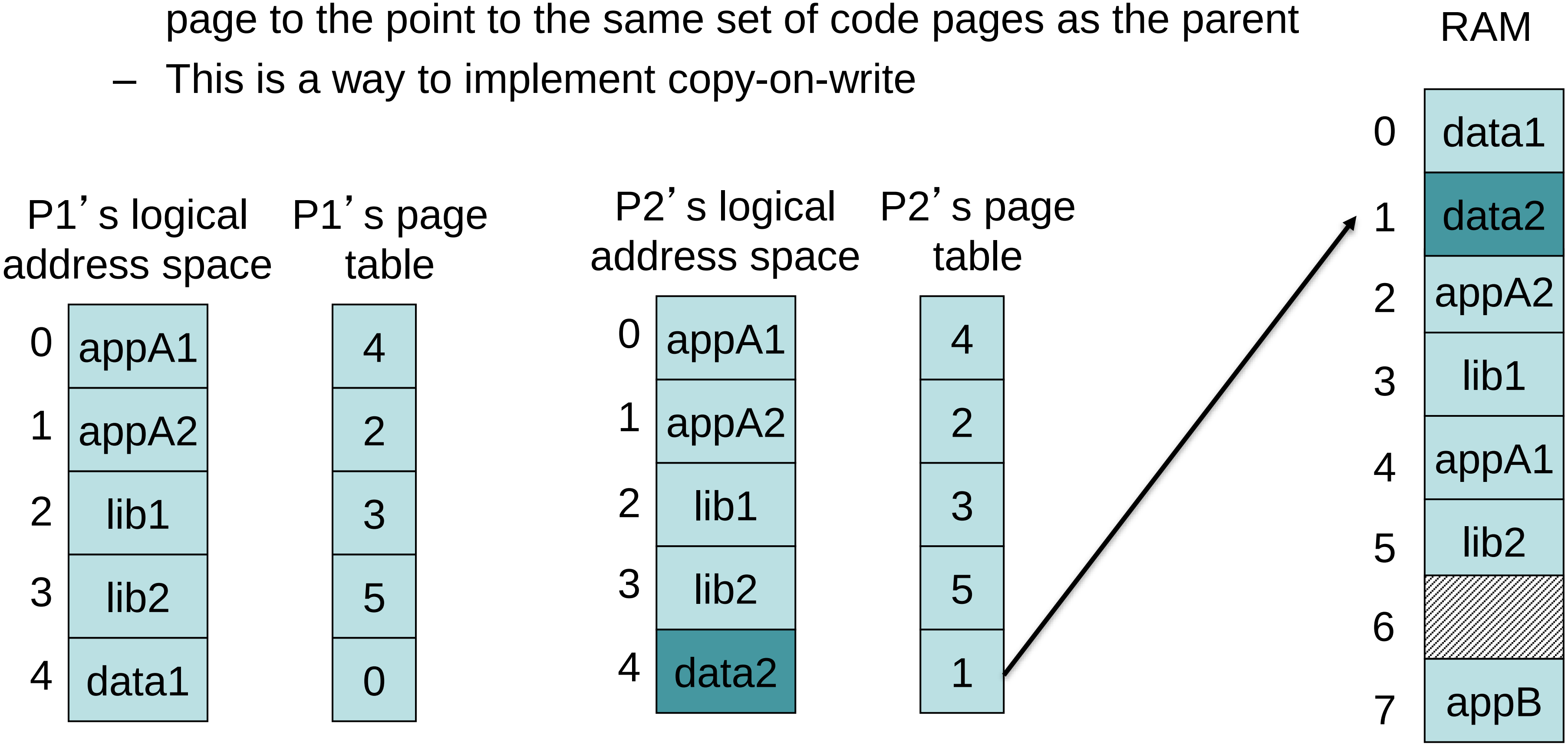


RAM

0	data1
1	data2
2	appA2
3	lib1
4	appA1
5	lib2
6	▨
7	appB

Shared Pages

- Fork()’ ing a child process causes the child to have a copy of the entire address space of the parent, including code
  - Rather than duplicating all such code pages, can simply map the child’s page to the point to the same set of code pages as the parent
  - This is a way to implement copy-on-write



P1’s logical address space

0	appA1
1	appA2
2	lib1
3	lib2
4	data1

P1’s page table

4
2
3
5
0

P2’s logical address space

0	appA1
1	appA2
2	lib1
3	lib2
4	data2

P2’s page table

4
2
3
5
1

RAM

0	data1
1	data2
2	appA2
3	lib1
4	appA1
5	lib2
6	▨
7	appB

End