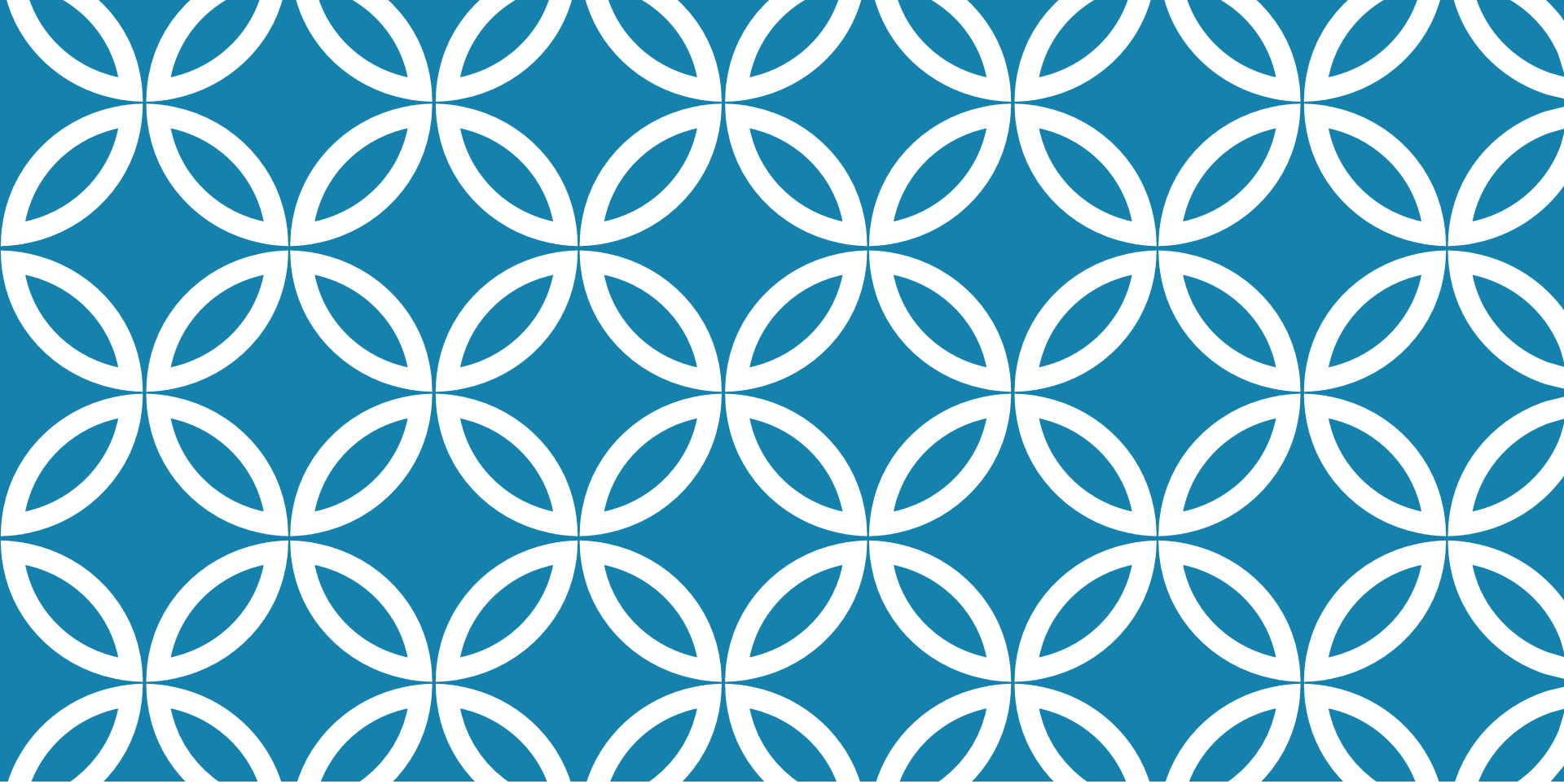


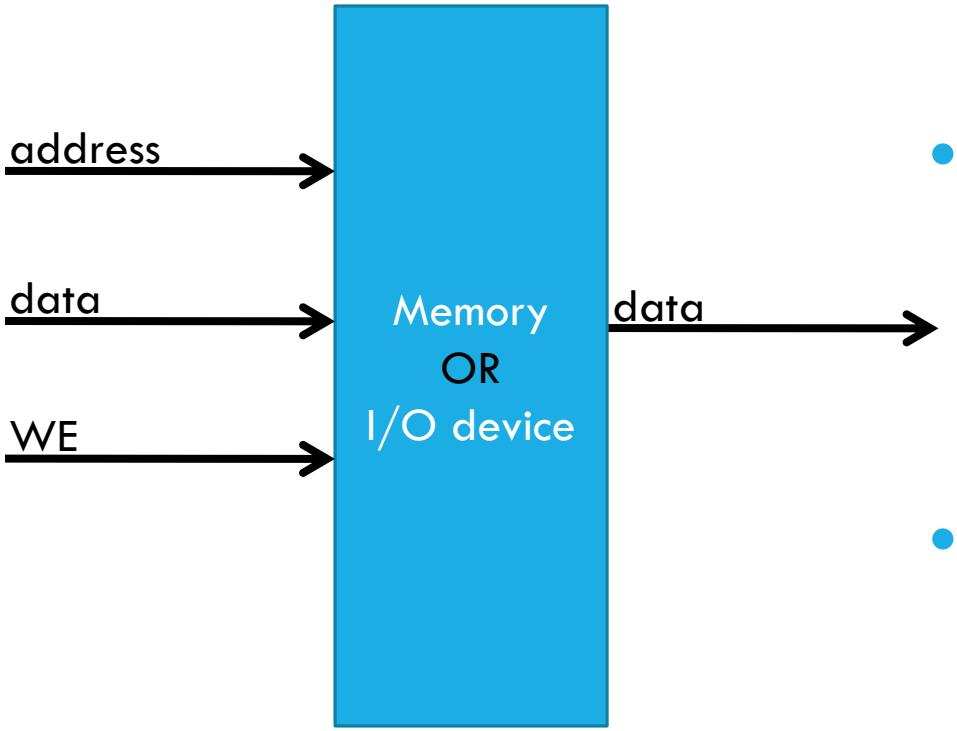
MEMORY MANAGEMENT

GOALS

1. Physical memory model
2. Address spaces: logical and physical
3. Binding of logical to physical addresses
4. Compilation – Linking revisited (dynamic linking)
5. Segmentation
6. Paging
7. Thrashing

Physical Address Space

MEMORY MODEL



Memory OR

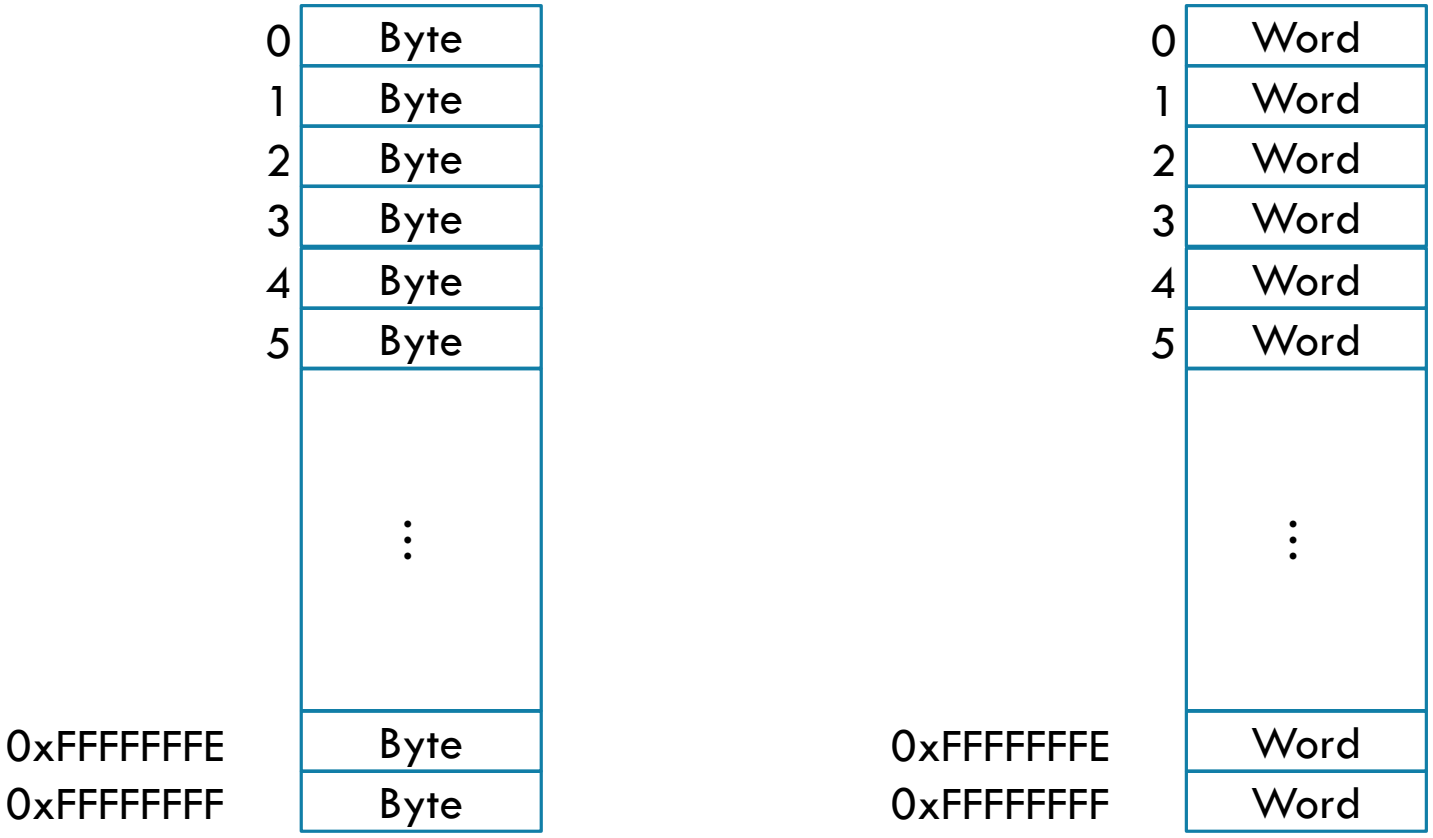
Memory-mapped I/O

- Input
  - Address
  - Data
  - WE (write-enable)
- Output
  - Data

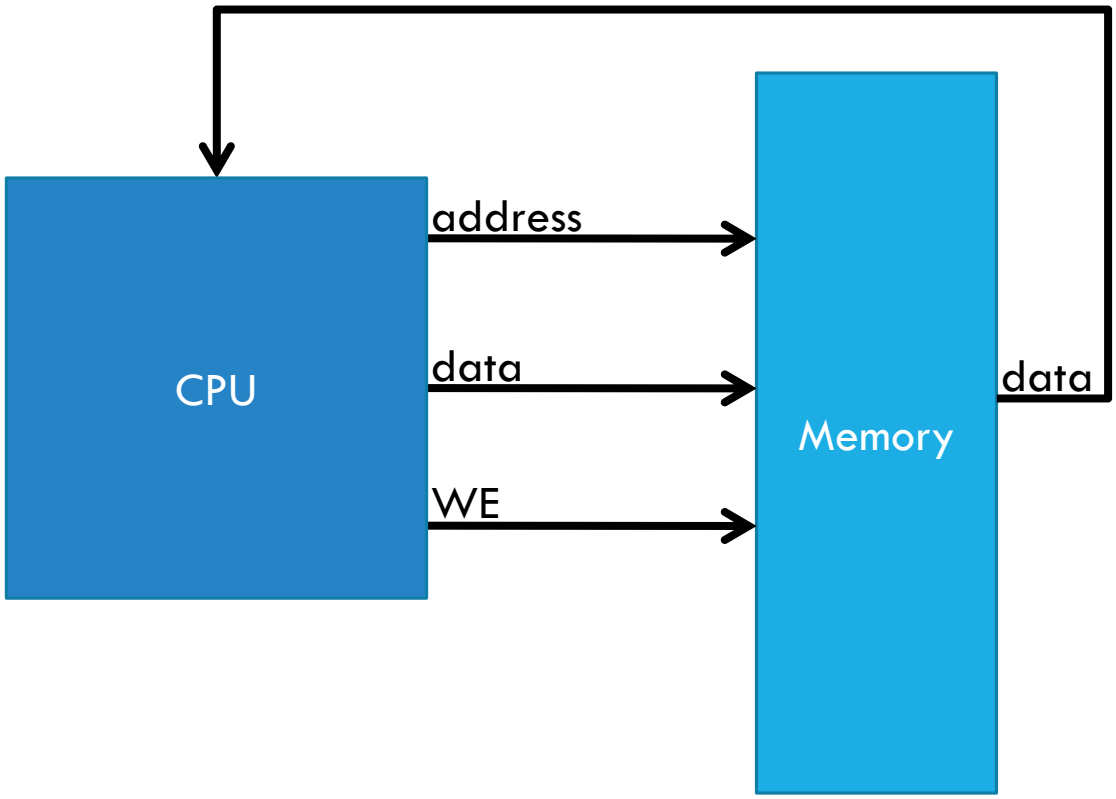
In **Memory-mapped I/O**, devices are accessed as if they were memory locations.

BYTE/ WORD ADDRESSABLE

- addressable means a unique address is associated with a fixed amount of data (bytes).



INTERFACING WITH THE CPU



Addresses generated by CPU

- **When?**
  - Compile time
  - Link time
  - Load time
  - Execution (run) time

From memory's point of view, it does not matter.

Q & A

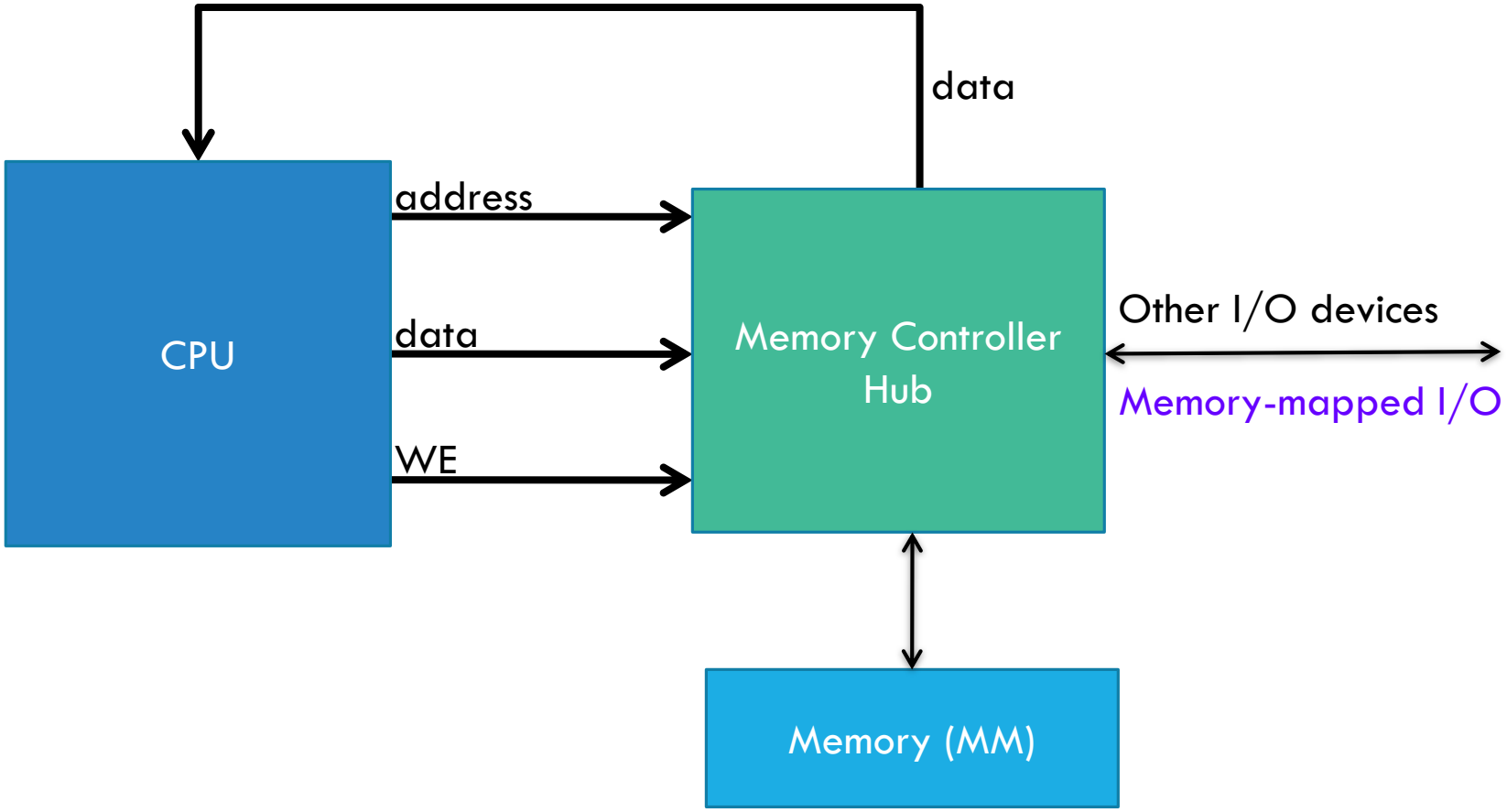
What is word size?

- Usually the width of integer registers used inside the CPU. The width of the data lines is also an indicator.

ADDRESS SPACE

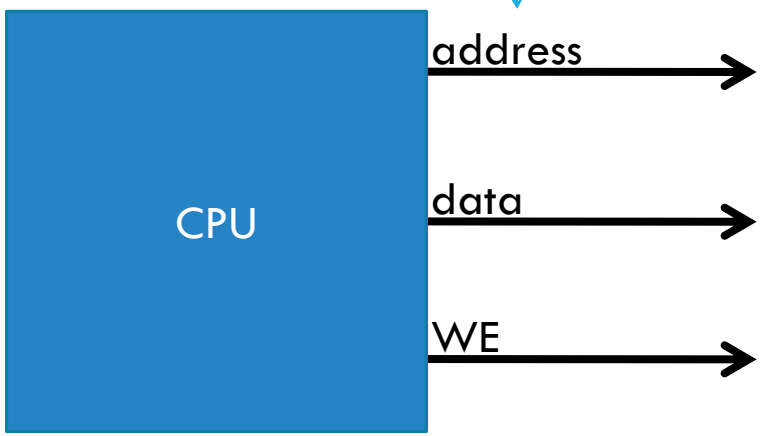
- A range of numbers (that's it?!)
- Limited only by address width
  - 32 bits can generate addresses between 0 to 2^{32}-1.
  - If memory is byte-addressable, what is the largest memory size possible for a 32-bit machine?

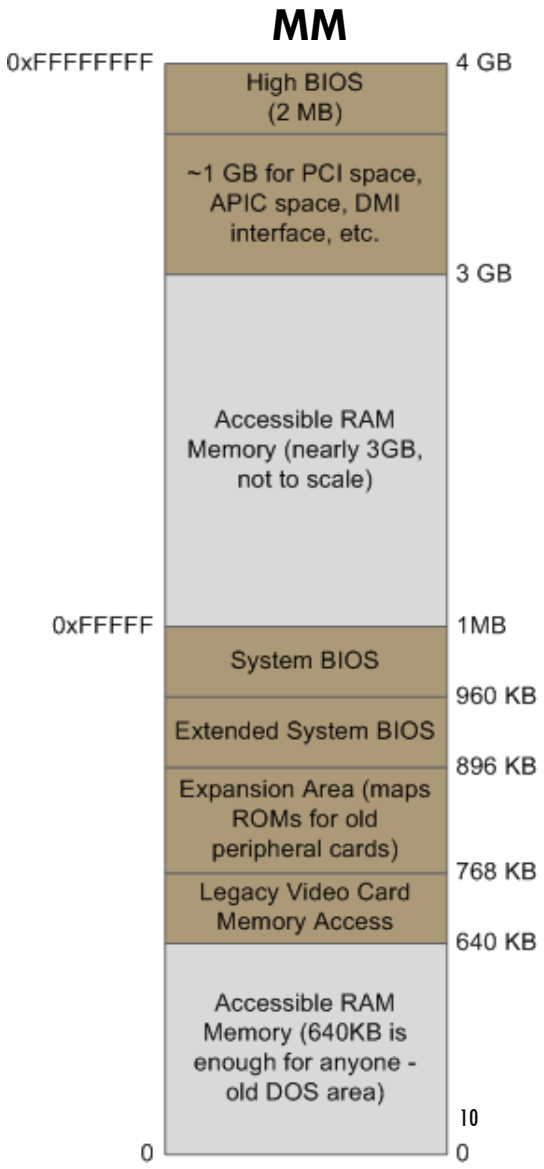
PHYSICAL ADDRESS SPACE SETUP – I



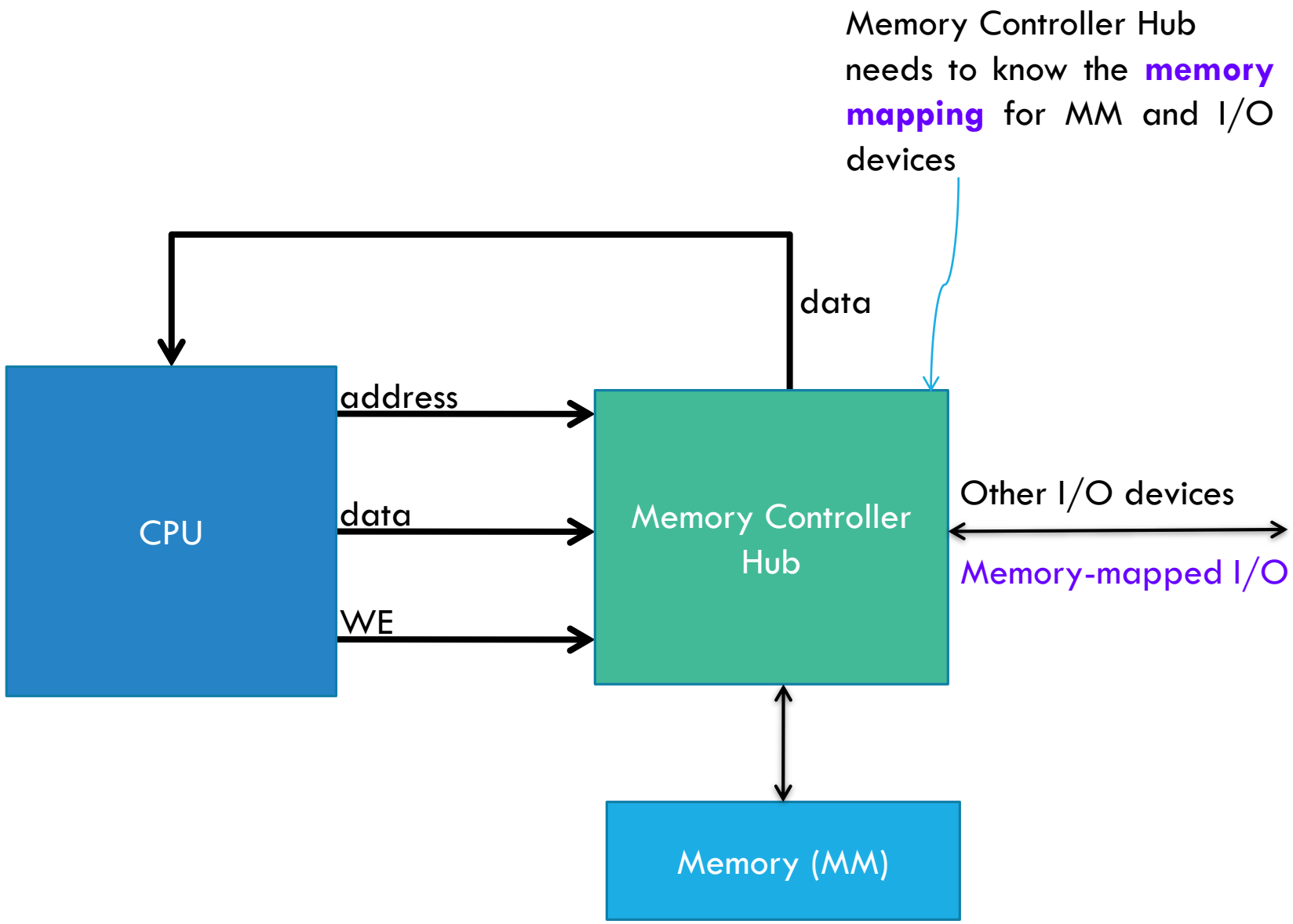
PHYSICAL ADDRESS SPACE (MAPPING)

All addresses issued by the CPU outwards are **physical addresses** (actually, **logical addresses** ‼)





PHYSICAL ADDRESS SPACE SETUP – II



Logical Address Space

RUN TWO PROCESSES OF THIS CODE. WILL THE PRINTOUT BE THE SAME?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // standard POSIX header file
#include <sys/wait.h> // POSIX header file for 'wait' function
int grace[1000];

int main()
{
  int res;
  char game[200];

  /* int fork(); */
  res=fork();

  if(res==0)
  {
    printf("\n\nInside Child process....\n");
    printf("=====================================\n");
    printf("address of global variable: [ %p ]\n", &grace[0]);
    printf("address of local variable: [ %p ]\n", &game[0]);
    printf("address of main() function: [ %p ]\n\n", main);
  }
  else
  {
    int status;
    wait(&status); /*parent wait for child to complete*/
    printf("Inside Parent process....\n");
    printf("=====================================\n");
    printf("address of global variable: [ %p ]\n", &grace[0]);
    printf("address of local variable: [ %p ]\n", &game[0]);
    printf("address of main() function: [ %p ]\n\n", main);
  }
  return 0;
}
```

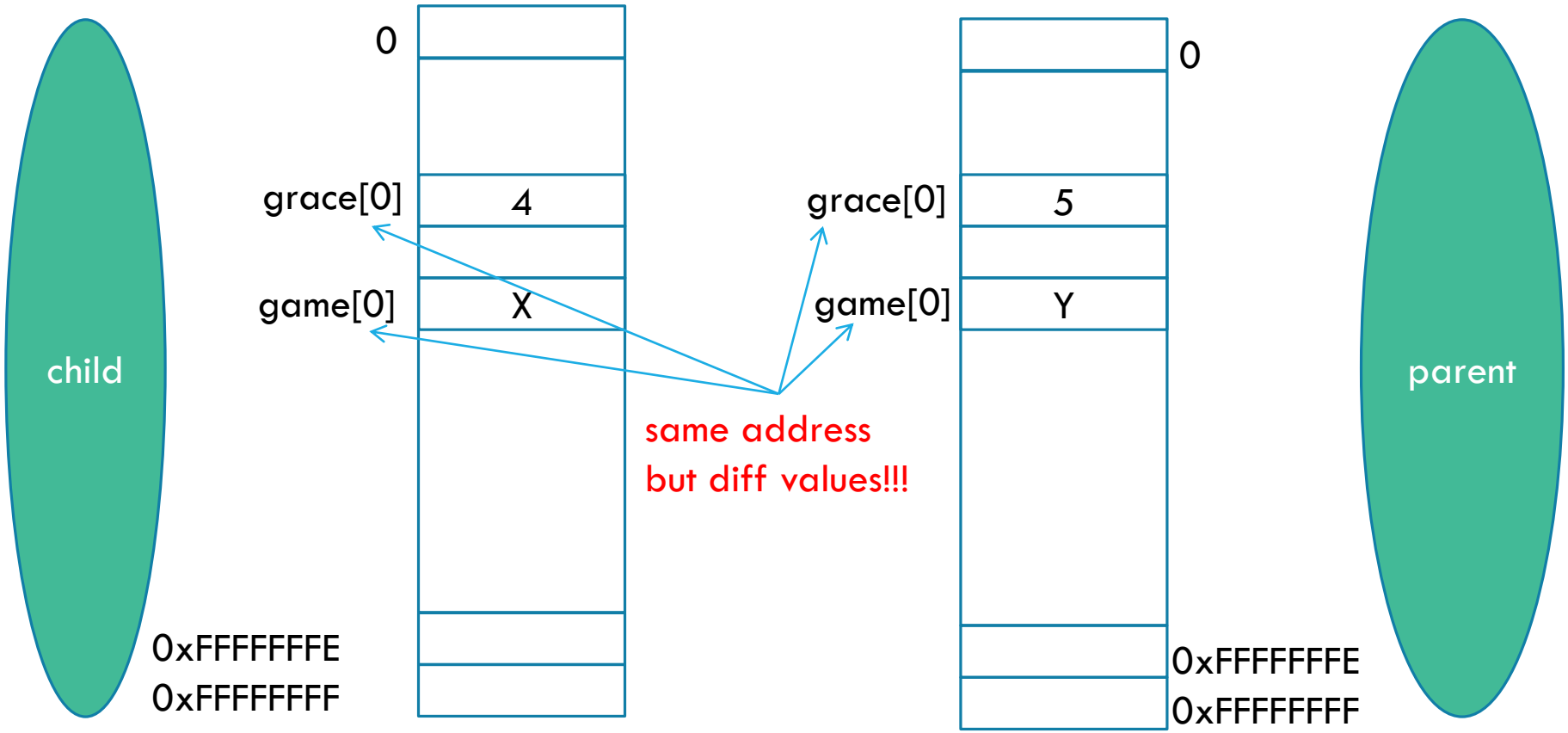
```
Inside Child process....
=======================================
address of global variable: [ 0x404080 ]
address of local variable: [ 0x7ffdcca042c0 ]
address of main() function: [ 0x401156 ]

Inside Parent process....
=======================================
address of global variable: [ 0x404080 ]
address of local variable: [ 0x7ffdcca042c0 ]
address of main() function: [ 0x401156 ]
```

RUN TWO PROCESSES OF THIS CODE. WILL THE PRINTOUT BE THE SAME?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // standard POSIX header file
#include <sys/wait.h> // POSIX header file for 'wait' function
int grace[1000];

int main()
{
  int res;
  char game[200];

  /* int fork(); */
  res=fork();

  if(res==0)
  {
    grace[0] = 4;
    game[0] = 'X';
    printf("\n\nInside Child process....\n");
    printf("=====================================\n");
    printf("address of global variable: [ %p ], its data: [ %d ]\n", &grace[0], grace[0]);
    printf("address of local variable: [ %p ], its data: [ %c ]\n", &game[0], game[0]);
  }
  else
  {
    int status;
    wait(&status); /*parent wait for child to complete*/
    grace[0] = 5;
    game[0] = 'Y';
    printf("\n\nInside Parent process....\n");
    printf("=====================================\n");
    printf("address of global variable: [ %p ], its data: [ %d ]\n", &grace[0], grace[0]);
    printf("address of local variable: [ %p ], its data: [ %c ]\n\n", &game[0], game[0]);
  }

   return 0;
}
```

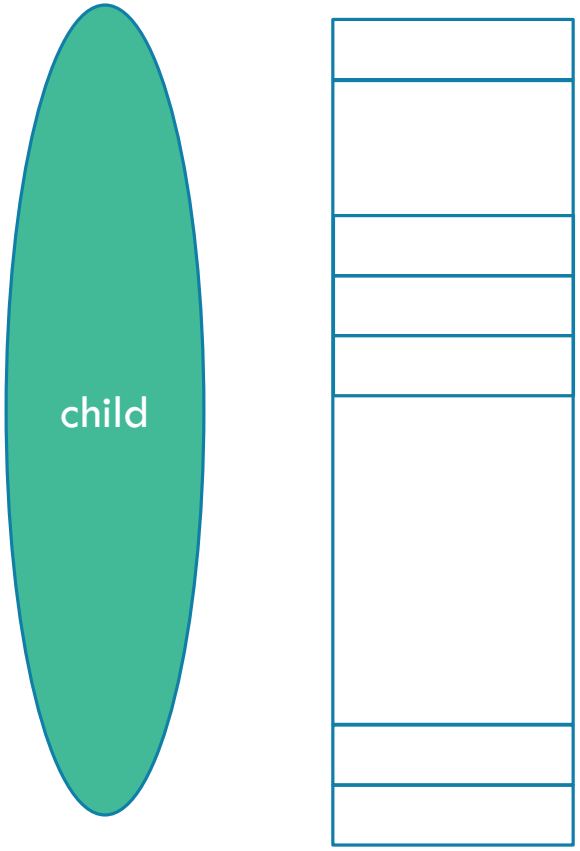
```
Inside Child process....
========================================
address of global variable: [ 0x404080 ], its data: [ 4 ]
address of local variable: [ 0x7ffddecace70 ], its data: [ X ]
Inside Parent process....
========================================
address of global variable: [ 0x404080 ], its data: [ 5 ]
address of local variable: [ 0x7ffddecace70 ], its data: [ Y ]
```

ADDRESS SPACES OF PROCESSES

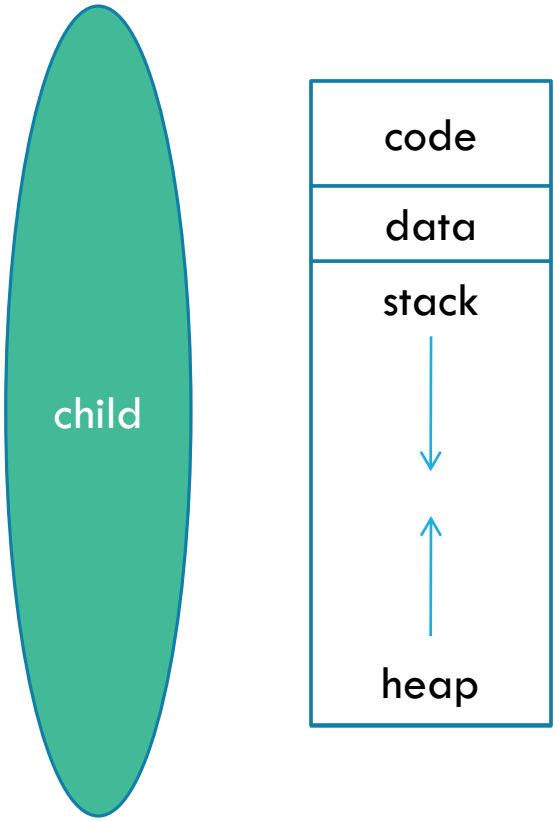


LOGICAL ADDRESS SPACE

- **Every process thinks that it owns the entire main memory.**
  - What each process sees is what we call **logical address space.**
- **Physical addresses** are **entirely hidden** from processes.
- Logical addresses need to be **translated** (mapped) into physical addresses.



PROCESS'S LOGICAL ADDRESS SPACE



- When are the logical addresses of each section decided?
- What is the relationship between logical address space and the physical address space? (i.e., get one from the other)

PROCESSING OF A PROGRAM (STATIC LINKING)

Source program

- Symbolic address
  - int count;

Compiler binds

- Relocatable address
  - 20 bytes from the starting address 90400 of this code module

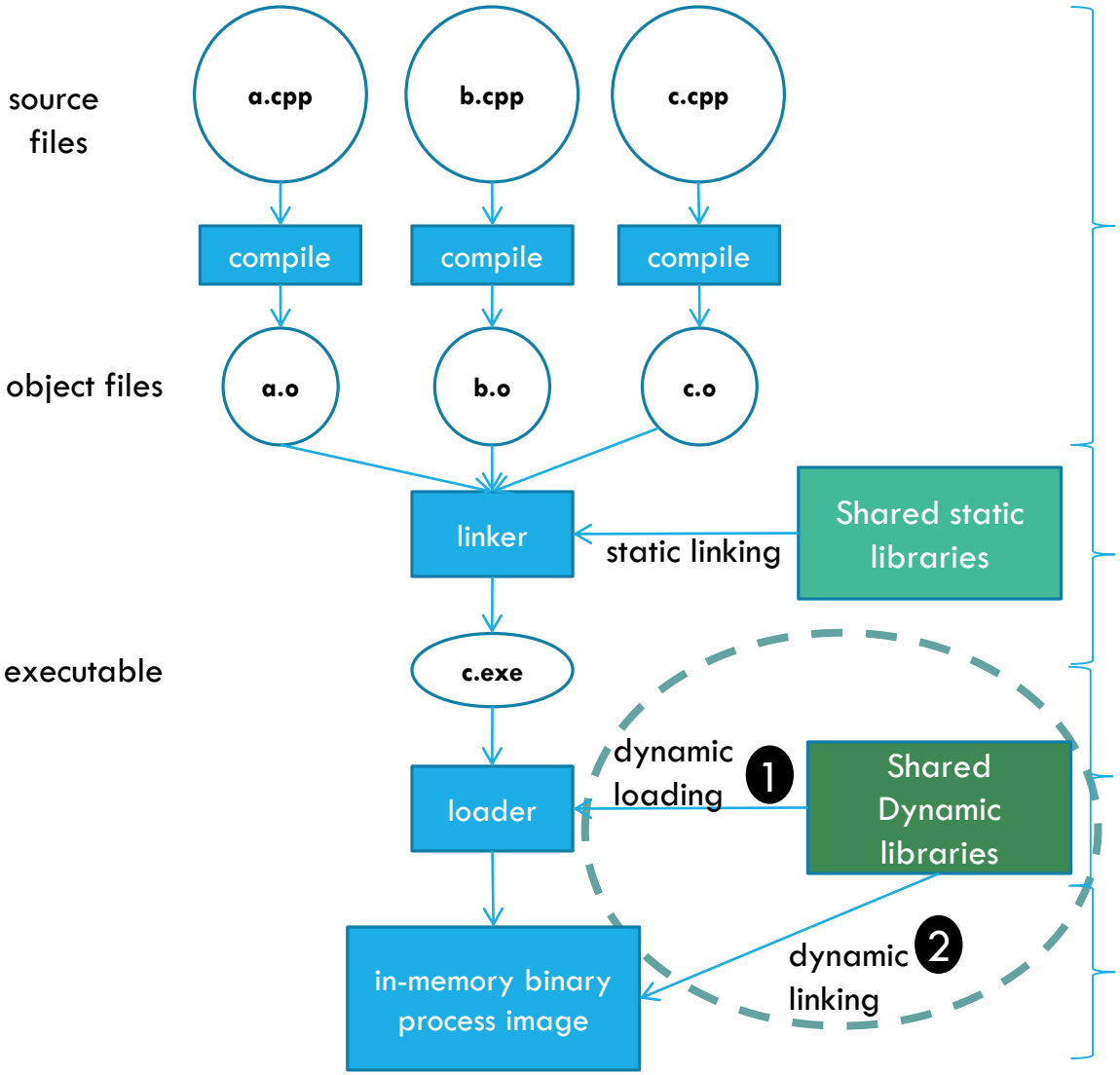
linker/loader binds

- Absolute address
  - 90420

PROCESSING OF A PROGRAM

Know at compile time where the process will reside in memory ?

- YES - absolute code
- NO - relocatable code



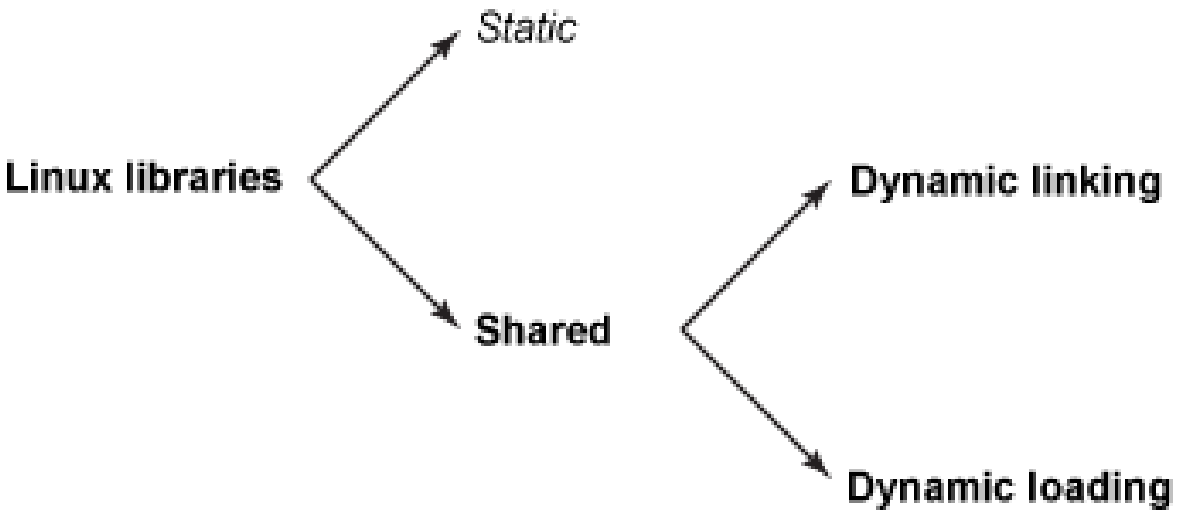
compile time

link time

load time

execution time (run time)

Linking Libraries in LINUX



Dynamic Loading: Suppose our program that is to be executed consists of various modules.

- we load the main module first and then during execution, we load some other modules **only when it is required** and the execution cannot proceed further without loading it.

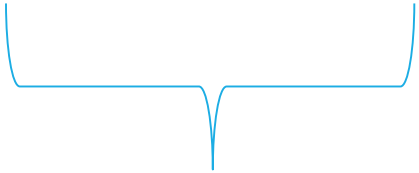
Dynamic Linking: Suppose our program has some functions whose definition is present in some system library.

- during execution when the function gets called, we load that system library into the main memory and **link** the **function call** inside our program with the **function definition** inside the system library.

Binding Logical Address Space to Physical Address Space

COMPILE/LINK-TIME BINDING

(Physical addresses) = (Logical addresses)



absolute address

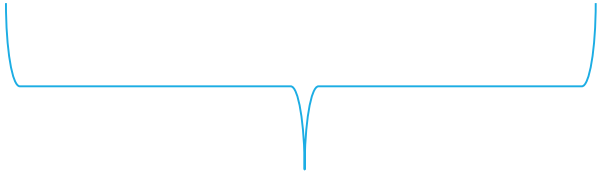
- Pros/Cons?
  - Fast
  - Need to recompile

LOAD-TIME BINDING

- Addresses are relative to some base address in physical memory.

  (physical address) = (base) + (logical address)

  - Pros/Cons?
    - need to reload



relocatable address

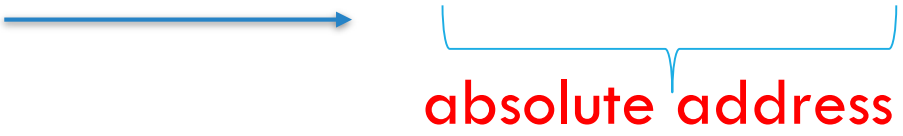
- Programs can be loaded anywhere in physical memory.

- Program can only be loaded if there is a **contiguous block of free physical memory** available large enough to hold the program and data.

EXECUTION-TIME (RUN-TIME) BINDING

- The physical address is computed in hardware at **runtime** by the *memory management unit (MMU)*.

  (logical address) mapped to **(physical address)**

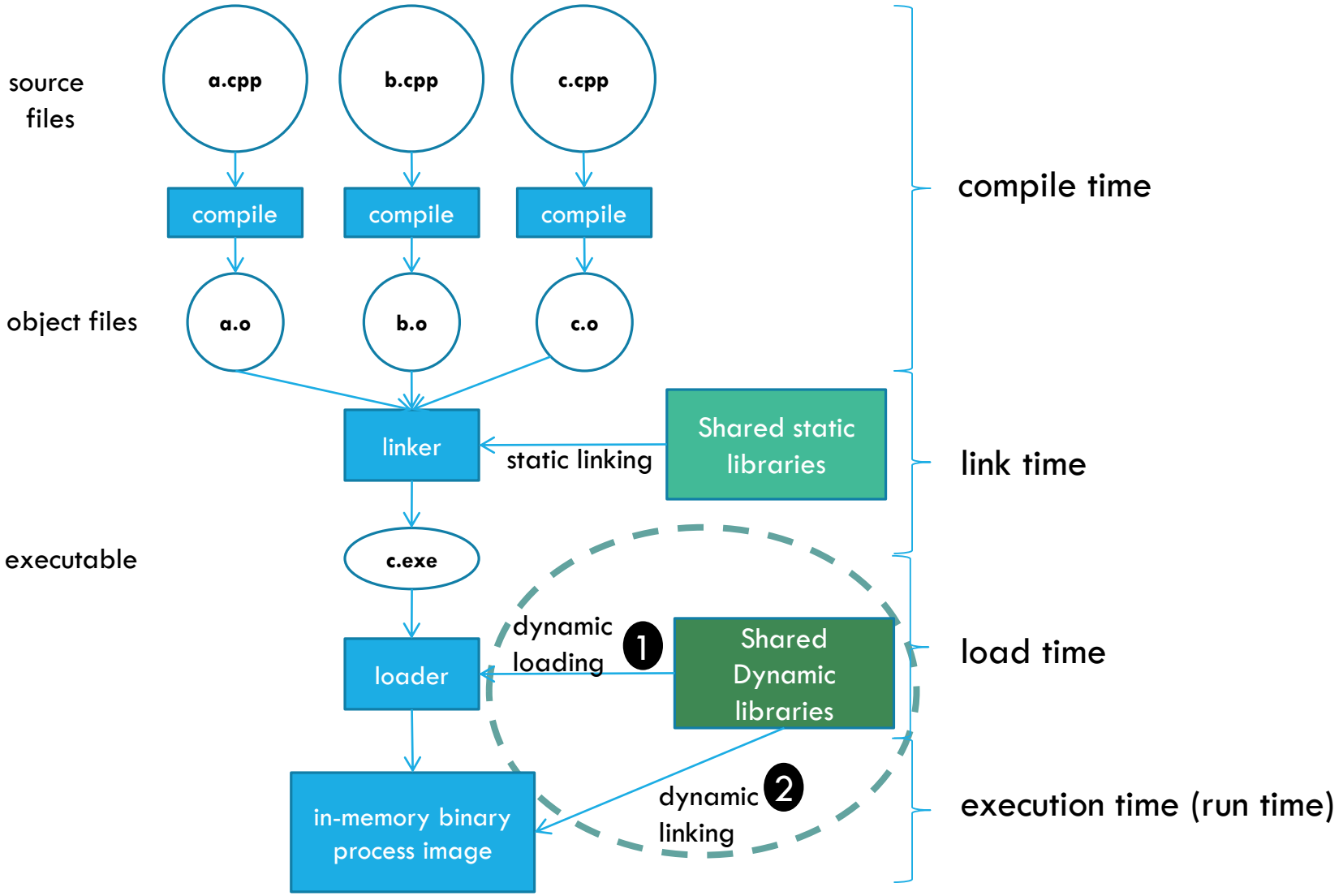
  

  - The mapping is not necessarily linear (details will be given later)

- Program may be relocated during execution (even after it is loaded).

- Program **does not require** a contiguous block of free physical memory.
  - Used by most modern OS's

MODERN SOLUTION: SHARED LIBRARIES



MODERN SOLUTION: SHARED LIBRARIES

Static libraries have the following disadvantages:

- **Duplication** in the stored executables (code redundancy)
- **Duplication** in the running executables (memory usage redundancy)
- Minor bug fixes of system libraries require each application to explicitly relink

Modern solution: Shared Libraries

- Object files that contain code and data that are loaded and linked into an application ***dynamically***, at either *load-time* or *run-time*
- Also called: dynamic link libraries, DLLs, .so files

SHARED LIBRARIES (CONT.)

Dynamic linking can occur when the executable is first loaded and run (**load-time linking**).

- Common case for Linux, handled automatically by the dynamic linker (**ld-linux.so**).
- Standard C library (**libc.so**) usually dynamically linked.

Dynamic linking can also occur after program has begun its execution (**run-time linking**).

- In Linux, this is done by calls to the **dlopen()** interface.
  - Distributing software.
  - High-performance web servers.
  - Runtime library interpositioning.

Shared library routines can be shared by multiple processes.

- More on this when we learn about virtual memory

DYNAMIC LINKING AT LOAD-TIME—CREATING A SHARED LIBRARY

Step 1: Write the Library Code

- mylib.c

```c
// mylib.c

int add(int a, int b) {
    return a + b;
}
```

Step 2: Compile the Shared Library

- gcc -shared -o libmylib.so -fPIC mylib.c

Step 3: Set the Library Path

- ***One way:*** Placing the library “libmylib.so” in one of the standard library directories like "/usr/lib" or "/usr/local/lib.“
- cp libmylib.so /usr/lib/

Step 4: Write the Program Using the Shared Library

- main.c

```c
// main.c
#include <stdio.h>

// Declare the external function from the shared library
extern int add(int a, int b);

int main() {
    int result = add(5, 3);
    printf("Result: %d\n", result);
    return 0;
}
```

Step 5: Compile the Program

- gcc -o myprogram main.c –lmylib

Step 6: Run the Program

- ./myprogram

DYNAMIC LINKING AT RUN-TIME

```
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>

int x[2] = {1, 2};
int y[2] = {3, 4};
int z[2];

int main()
{
    void *handle;
    void (*addvec)(int *, int *, int *, int);
    char *error;

    /* Dynamically load the shared library that contains addvec() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
```

```
$gcc -shared -o libvector.so -fPIC addvec.c
```

dll.c

DYNAMIC LINKING AT RUN-TIME (CONTD.)

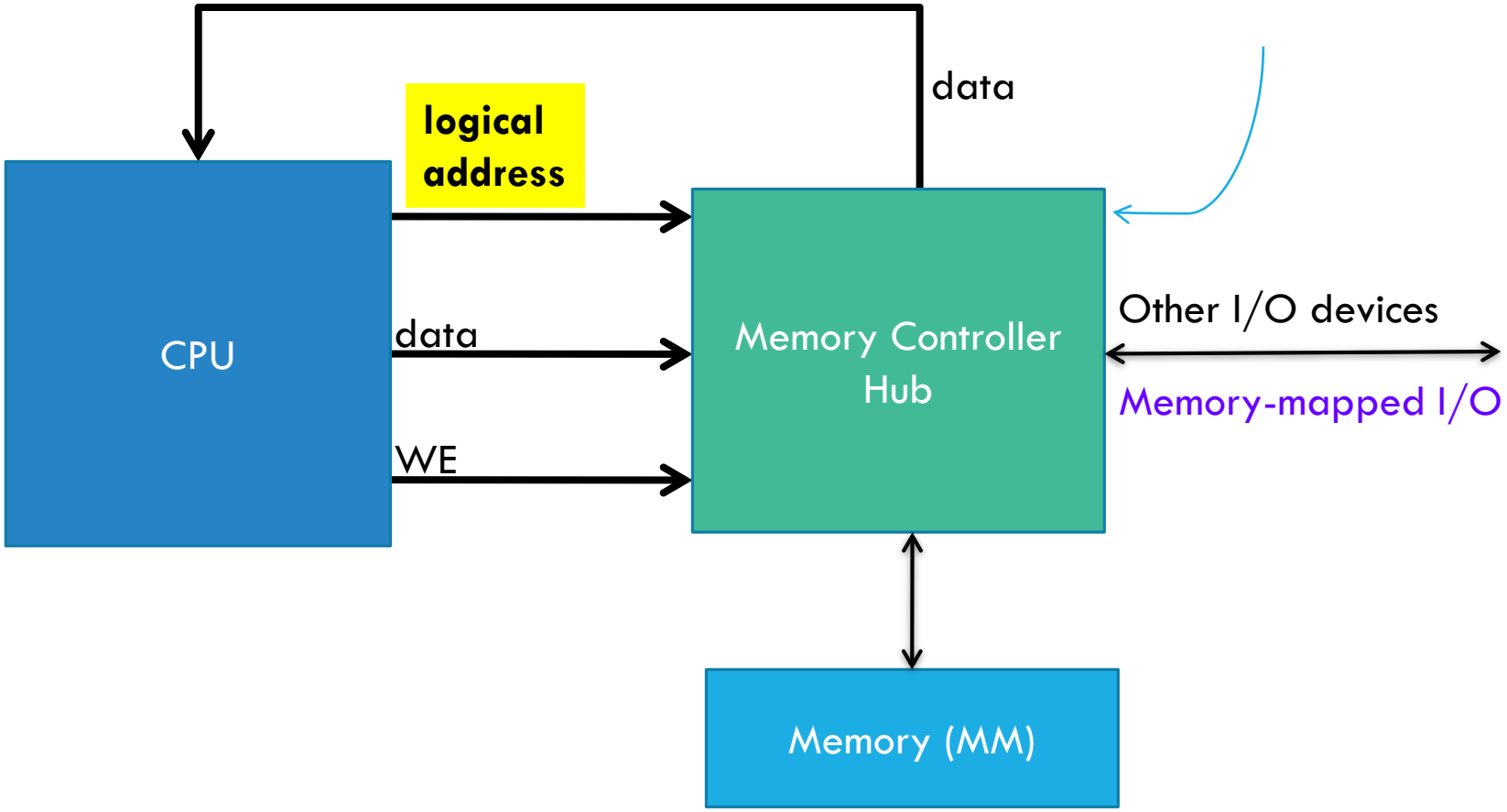
```
    . . .

    /* Get a pointer to the addvec() function we just loaded */
    addvec = dlsym(handle, "addvec");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call addvec() just like any other function */
    addvec(x, y, z, 2);
    printf("z = [%d %d]\n", z[0], z[1]);

    /* Unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

dll.c

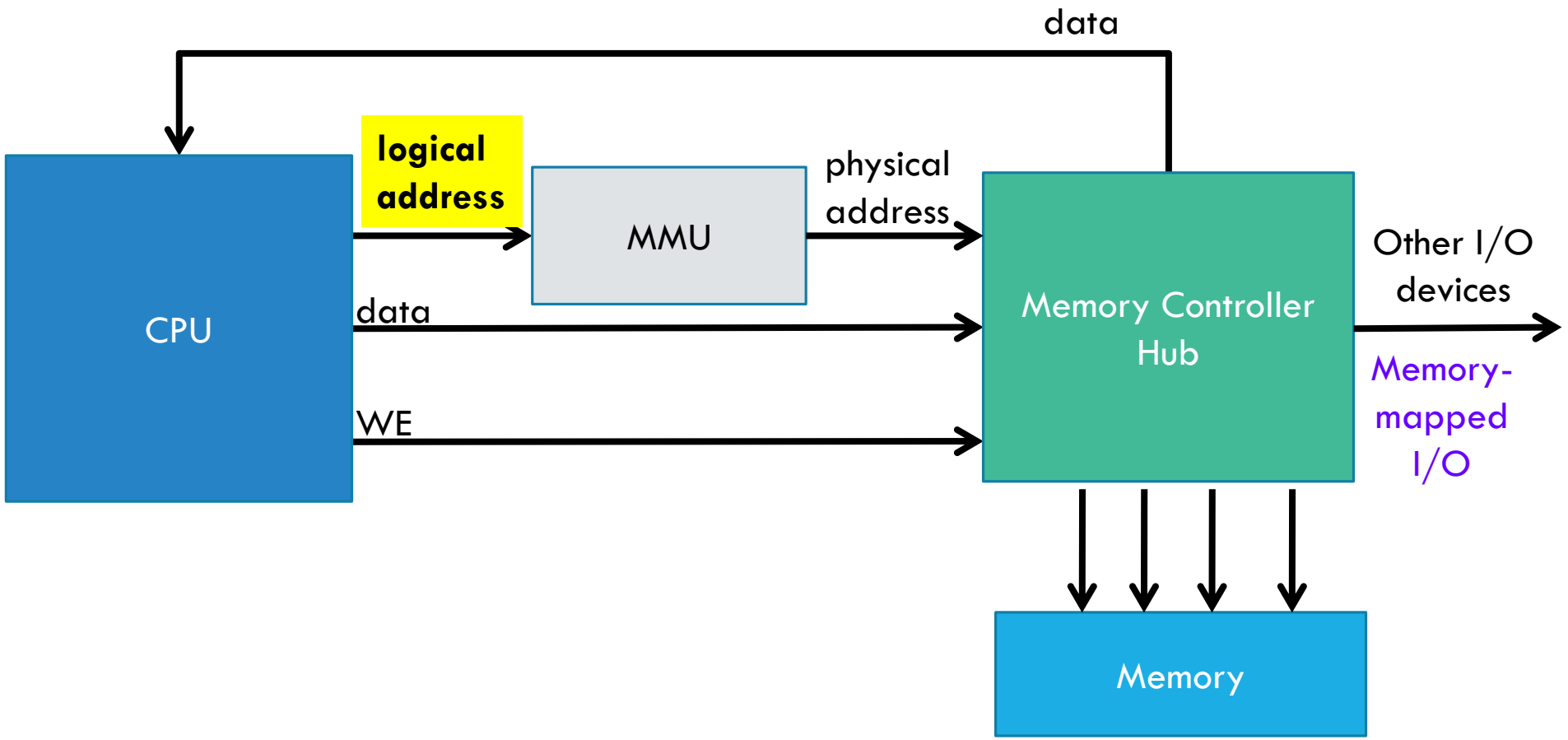
Binding Logical Address Space to Physical Address Spaces

MEMORY MANAGEMENT UNIT (MMU)

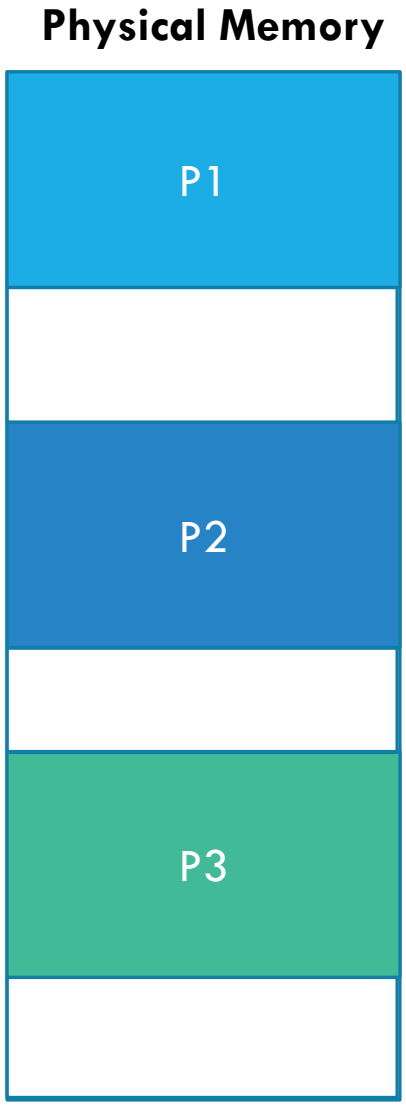
Memory Controller Hub needs to know the **memory mapping from logical to physical address** for MM and I/O devices



MEMORY MANAGEMENT UNIT (MMU)



NAÏVE IDEA: EACH PROCESS GETS A PIECE OF PHYSICAL MEMORY



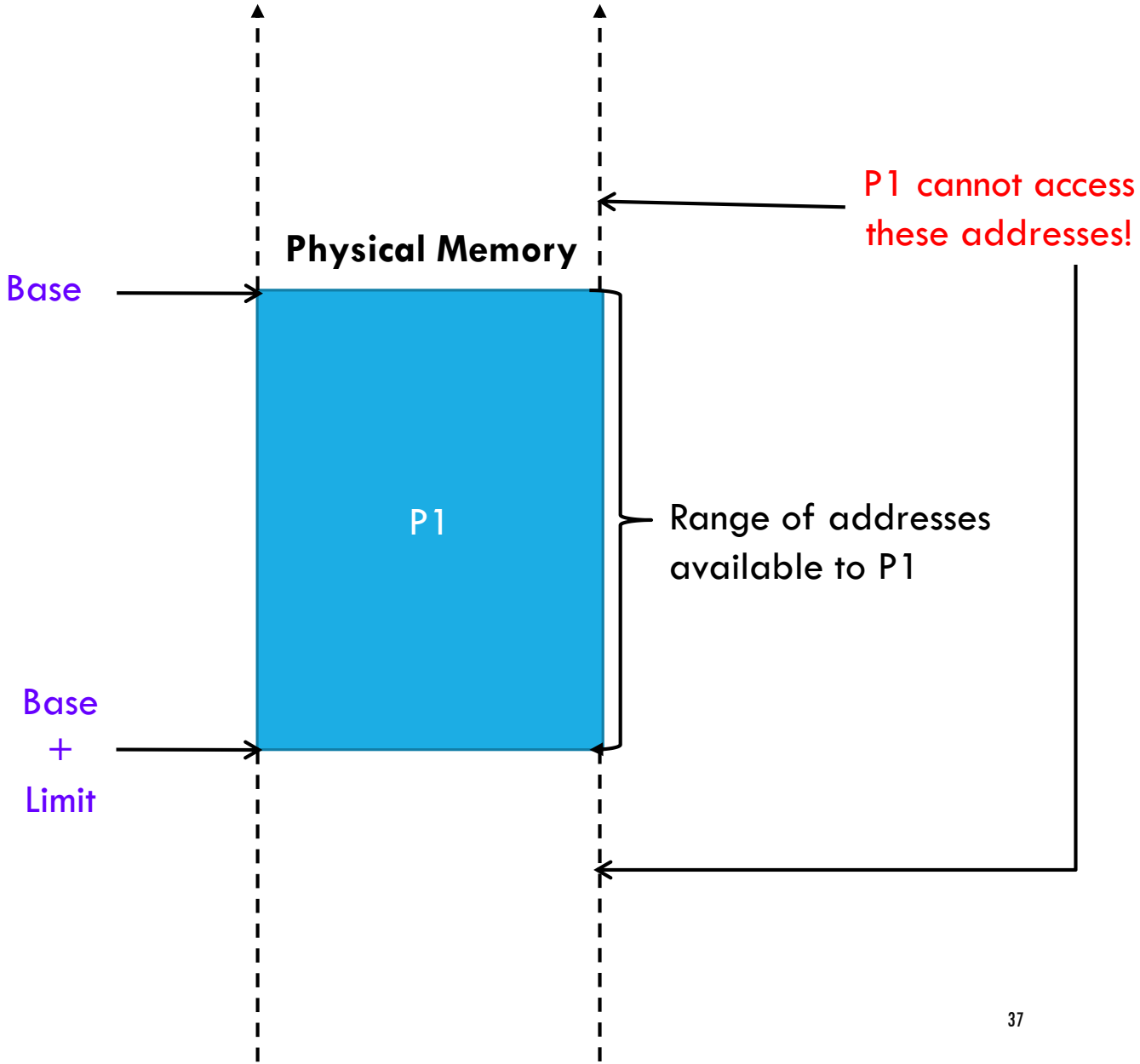
BASE AND LIMIT REGISTERS

Base Register
(Relocation Register)

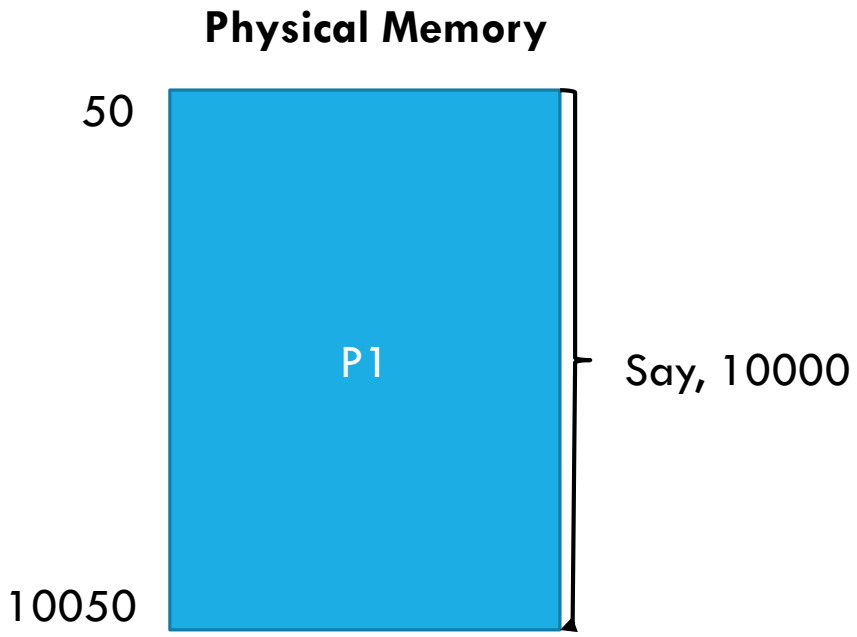
- Contains the smallest legal physical memory address of a process.

Limit Register

- Contains the range of logical addresses



P1'S LOGICAL ADDRESS SPACE



Base register : 50

Limit register : 10000

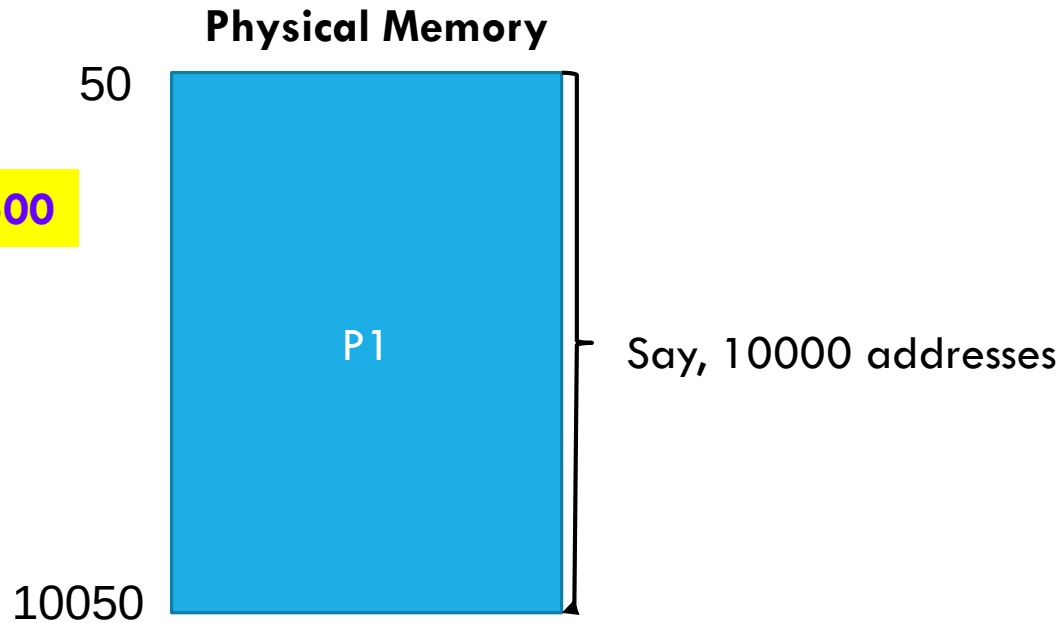
- P1's **logical address space** will range from 0 to 10000.
- P1's **physical address space** will range from 50 to 10050.

(Physical address) = Base + (Logical address)

LOGICAL ADDRESS SPACE AND CODE

Suppose the logical address of **X** is **500**

```
*p = &X;
```



Address contained by `pointer` SHOULD NOT Exceed 10050 !

TRANSLATING LOGICAL ADDRESS TO PHYSICAL ADDRESS

Suppose the logical address of **X** is **500**

C code

```
*p = &X;
```

In assembly:

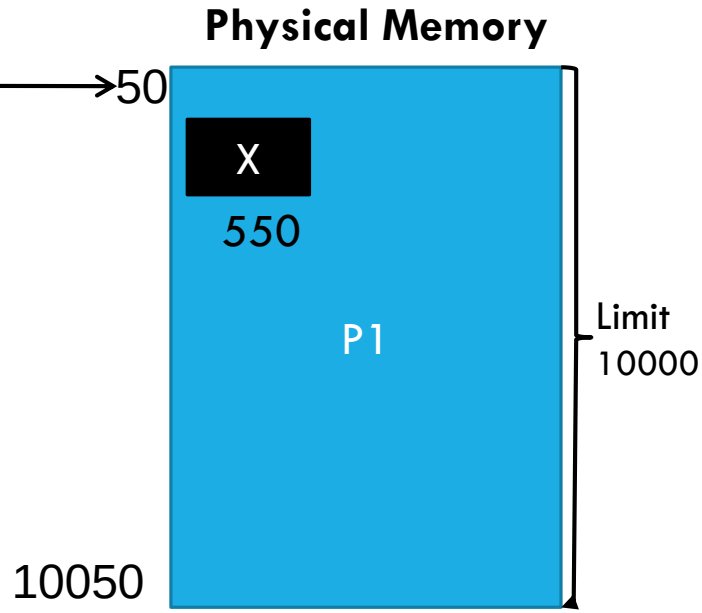
```
mov  EAX, 500
```

CPU execute this instruction & EAX holds the **logical address**!

During run time:

```
mov EBX, base
add EBX, EAX
```

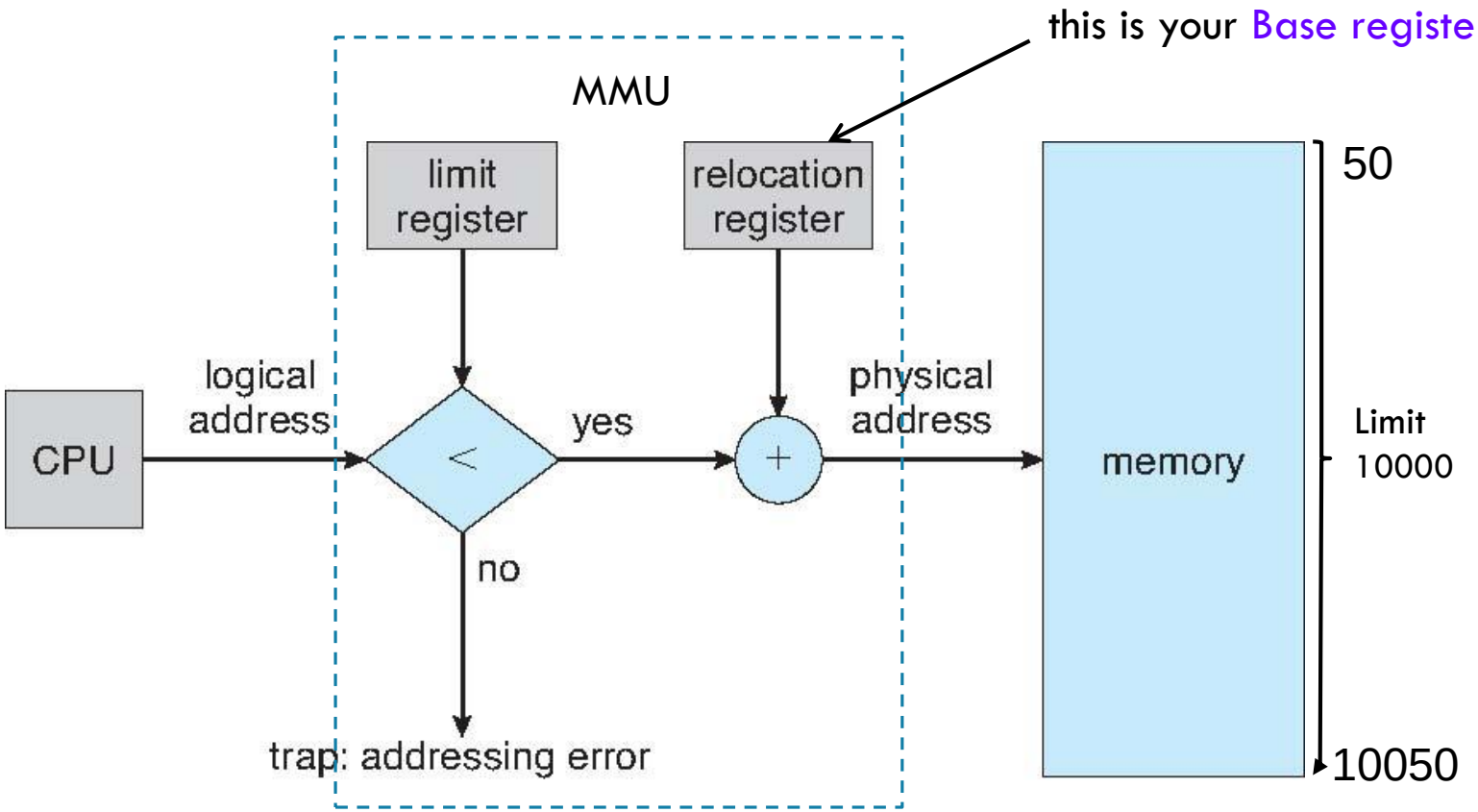
EBX stores the **physical address**



Translated by doing this in HW

```
if (500 < limit)
    Physical Address = Base + 500
else
    cause a fault
```

SIMPLE CONTIGUOUS MEMORY ALLOCATION



Providing **Memory Protection with relocation and limit registers**

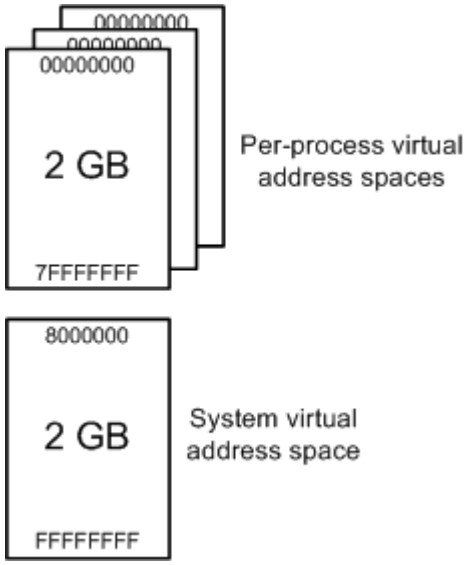
KERNEL ADDRESS SPACE

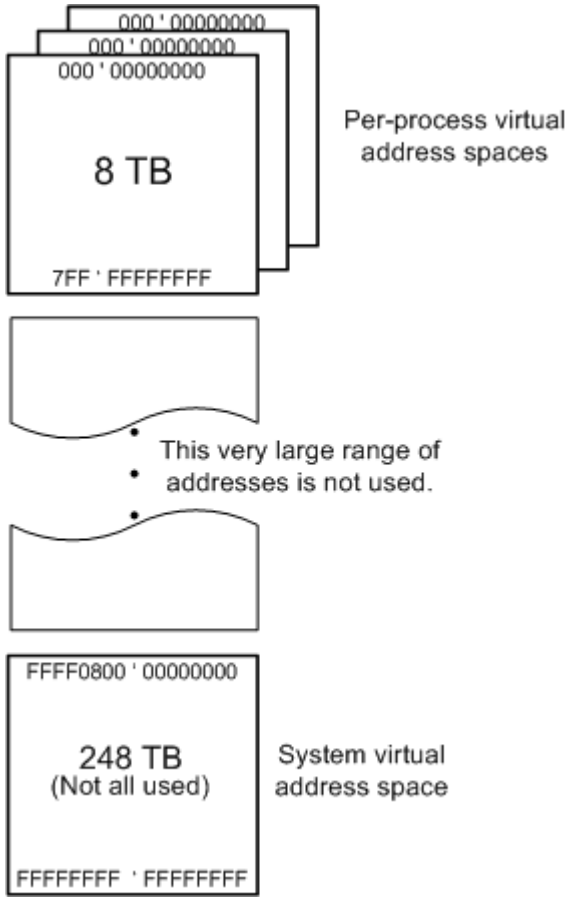
- Kernel in physical address space
  - disable MMU in kernel mode, enable MMU in user mode;
  - to access process data, the kernel must interpret page tables without hardware support;
  - OS must always be in physical memory (memory-resident).
- Kernel in separate virtual address space
  - MMU has separate state for user mode and kernel mode;
  - accessing process data is rather difficult;
  - parts of the kernel data may be non-resident.
- Kernel shares virtual address space with each process
  - use memory protection mechanisms to isolate kernel from user processes;
  - accessing process data is trivial;
  - parts of the kernel data may be non-resident

LOGICAL ADDRESSES AND KERNEL MODE

- Whose code is running in kernel mode? User process or OS?
  - OS
- Does the OS use logical address space?
  - NO. The OS "turns off" address translation
- How does OS turn off the translation?
  - Setting Base register to **0** and Limit register to **maximum** memory size.
- How about processes?
  - They can only access logical address space in user mode.

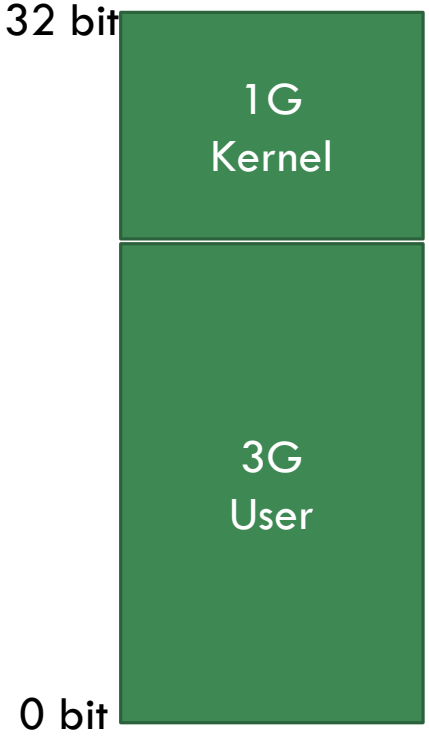
USER AND KERNEL SPACE MODEL (WINDOWS)



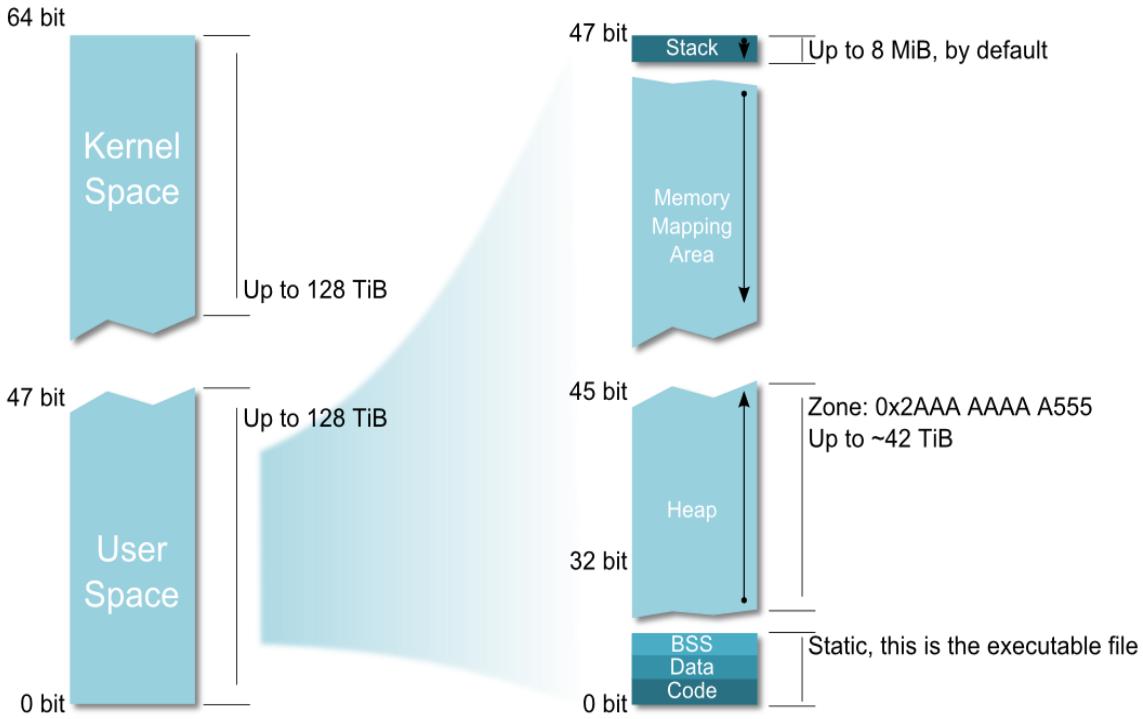


USER AND KERNEL SPACE MODEL (LINUX)

x86



x64

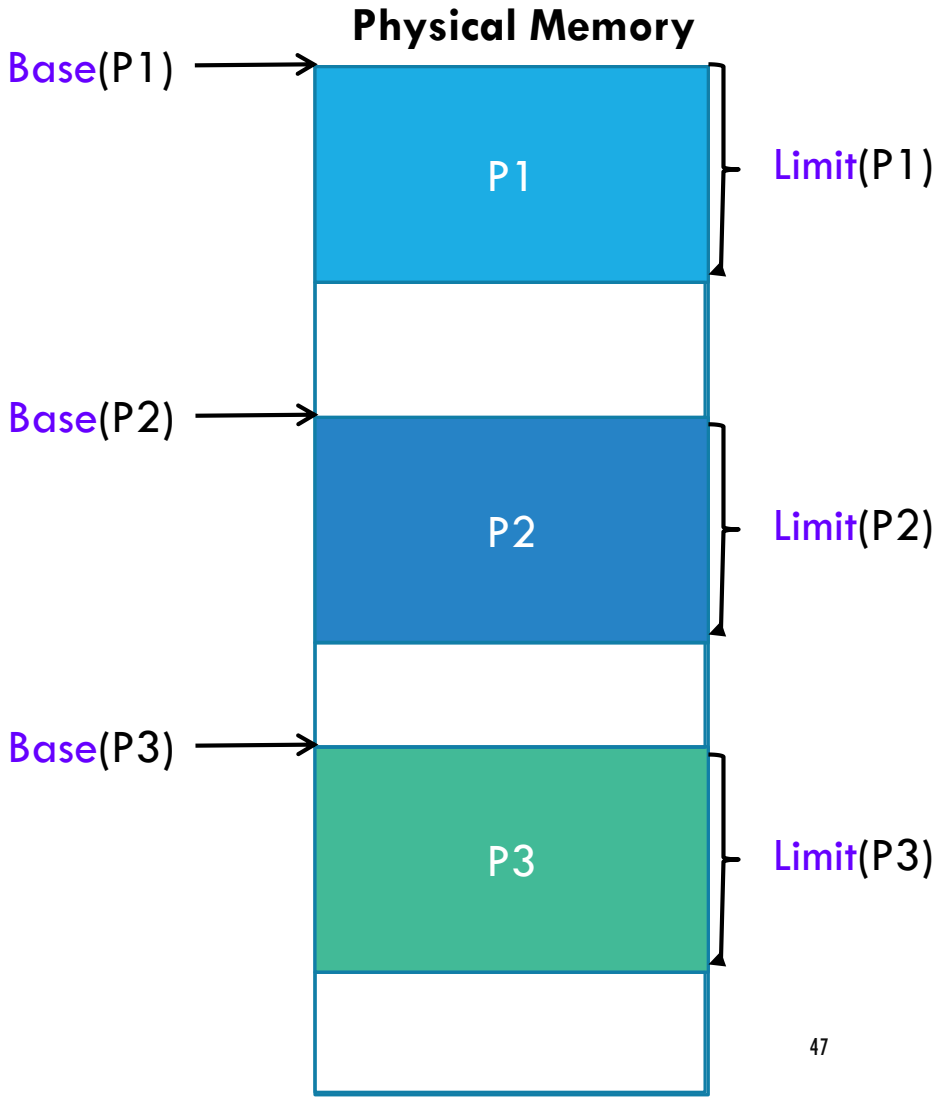


So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

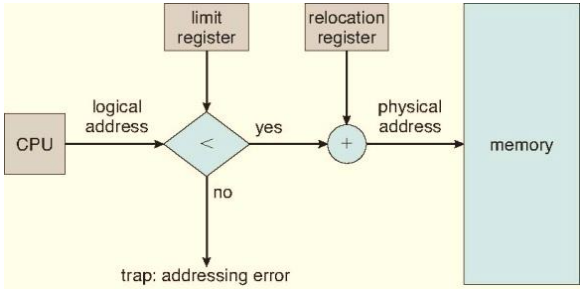
WHAT HAPPENS IN CONTEXT SWITCHING? - 1

1. Each process has a Base register and a Limit register for **mapping** its address.

2. Stored in the **PCB**.



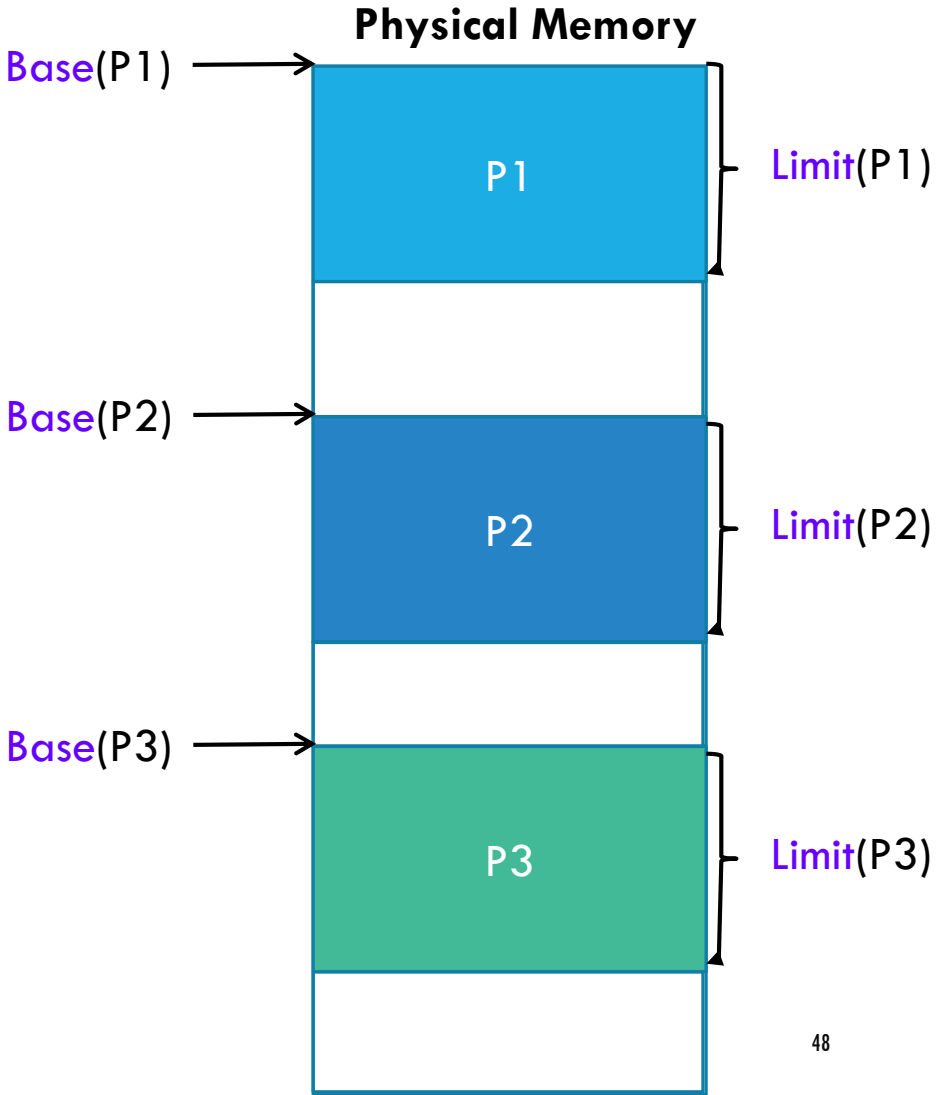
WHAT HAPPENS IN CONTEXT SWITCHING? - 2



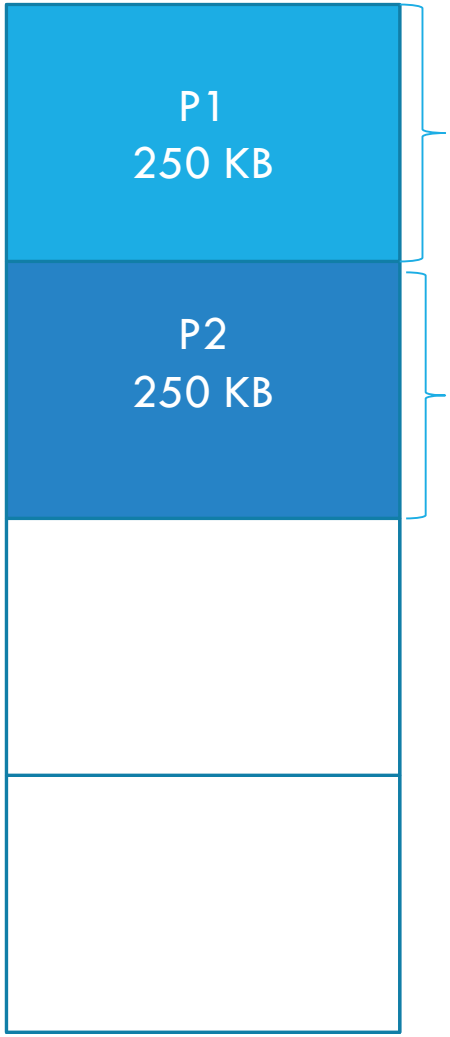
1. Suppose **P1** is running now.
   Relocation Register = Base(P1)
   Limit Register = Limit(P1)

2. **Interrupt happens!**
   A. **P1's context is saved.**
      - Base(P1) and Limit(P1) saved.
   B. Relocation Register = **0**
   C. Limit Register = **MAX**

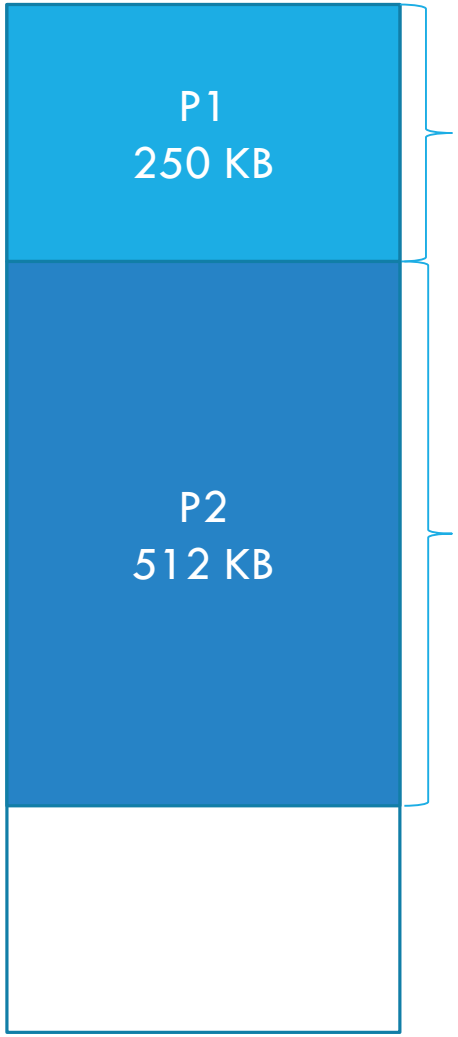
3. Scheduler decides to run **P2**
   A. **Restore P2's context**
   B. Relocation Register = Base(P2)
   C. Limit Register = Limit(P2)



FIXED-SIZE VERSUS VARIABLE SIZE MEMORY ALLOCATION

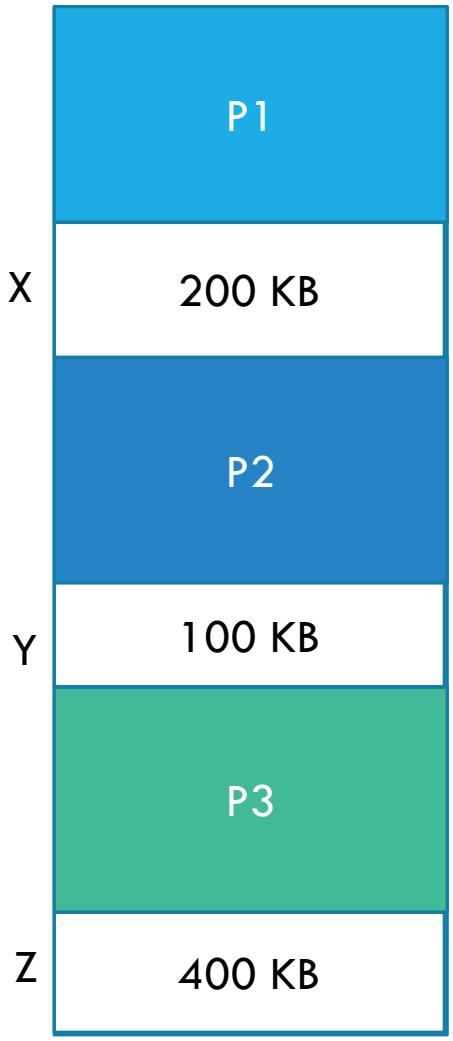


Each process has a **fixed-size** memory assigned to it



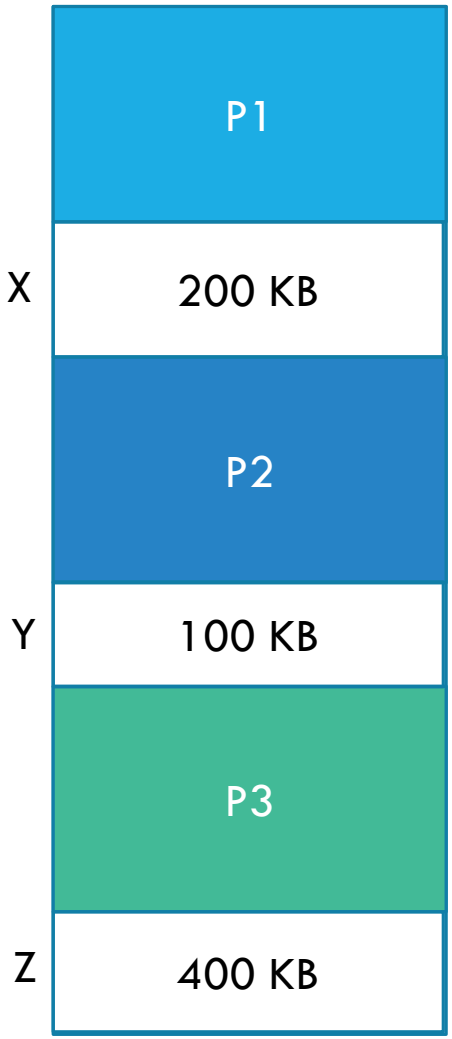
Each process has a **variable-size** memory Assigned to it

CONTIGUOUS MEMORY ALLOCATION

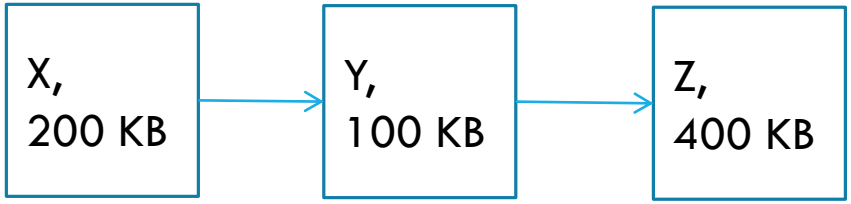


- Assign and allocate memory in a contiguous block to processes:
  - each process is loaded into a single, continuous block of physical memory
- Maintain a list of free blocks
- **Memory allocation techniques:**
  - First fit
  - Best fit
  - Worst fit

FIRST FIT

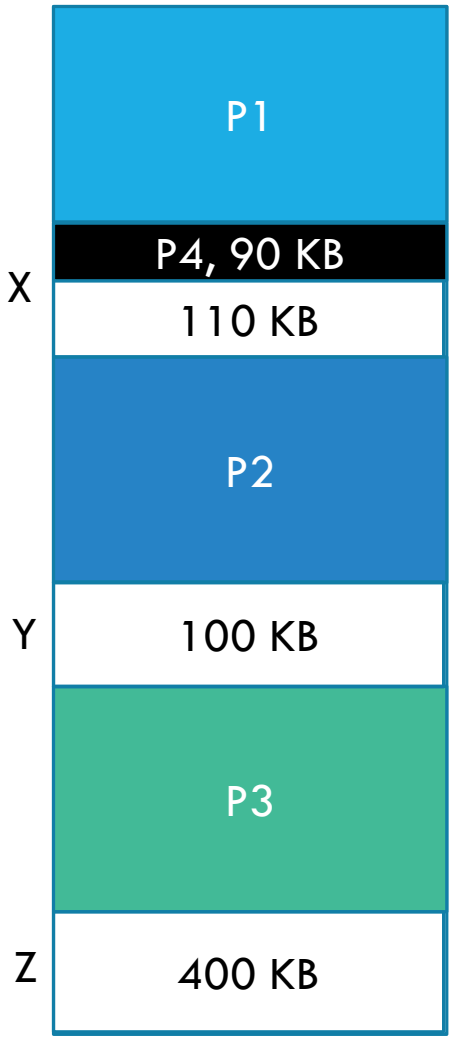


Free List

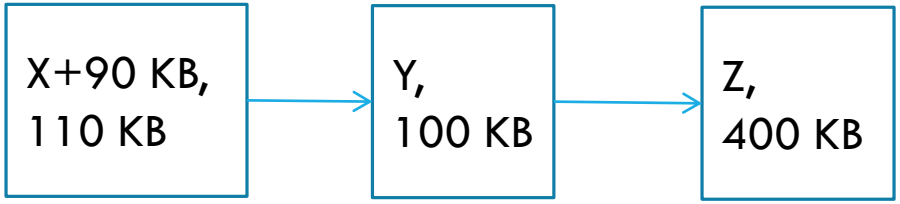


Request by P4

FIRST FIT

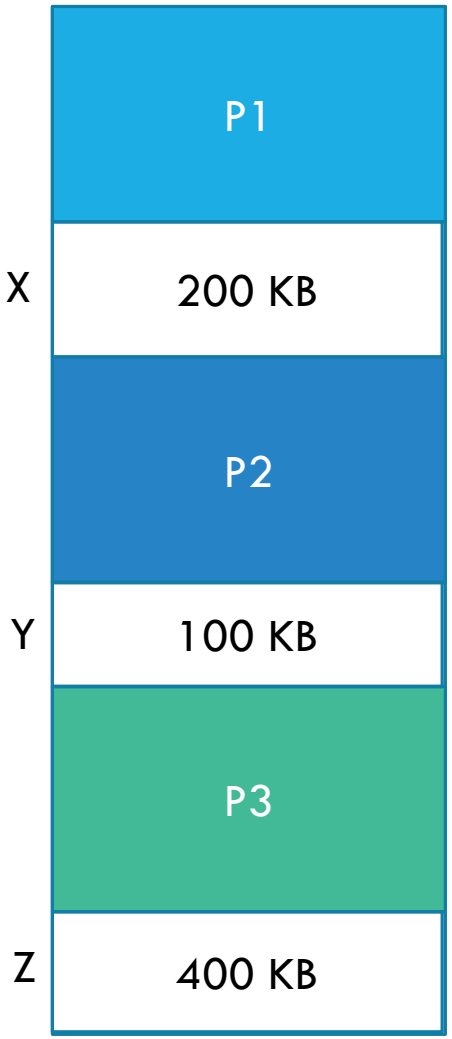


Free List

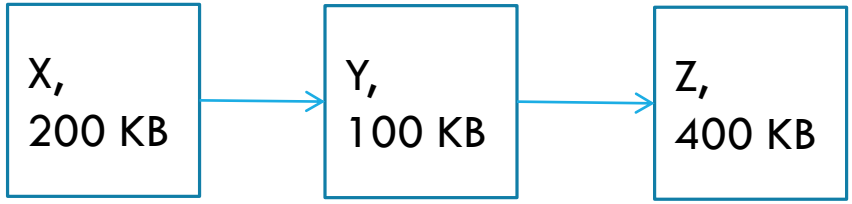


Request by P4

BEST FIT

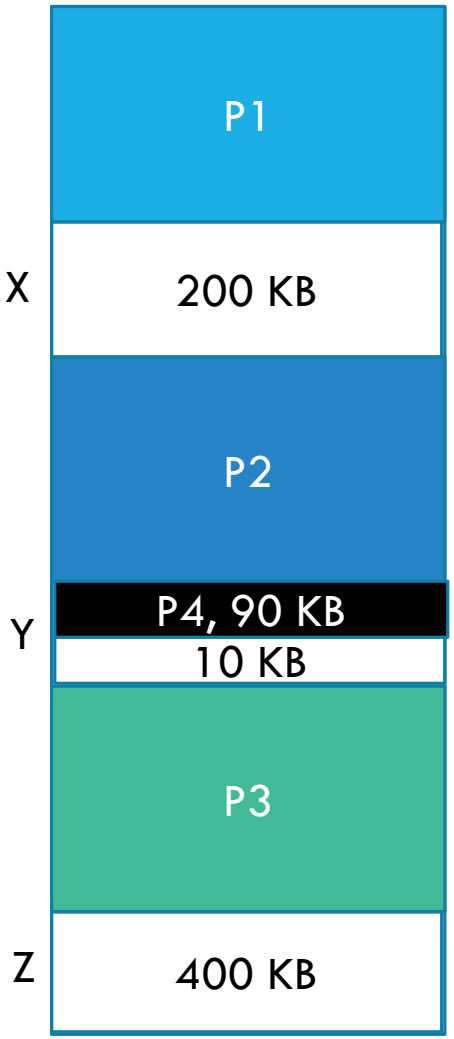


Free List

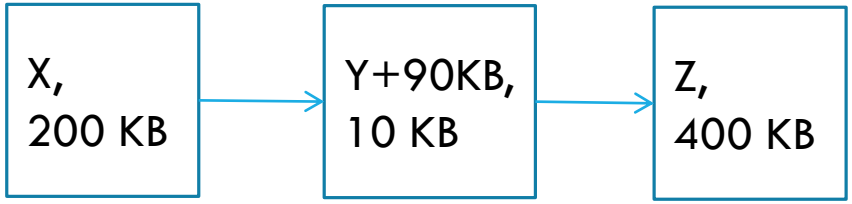


Request by P4

BEST FIT

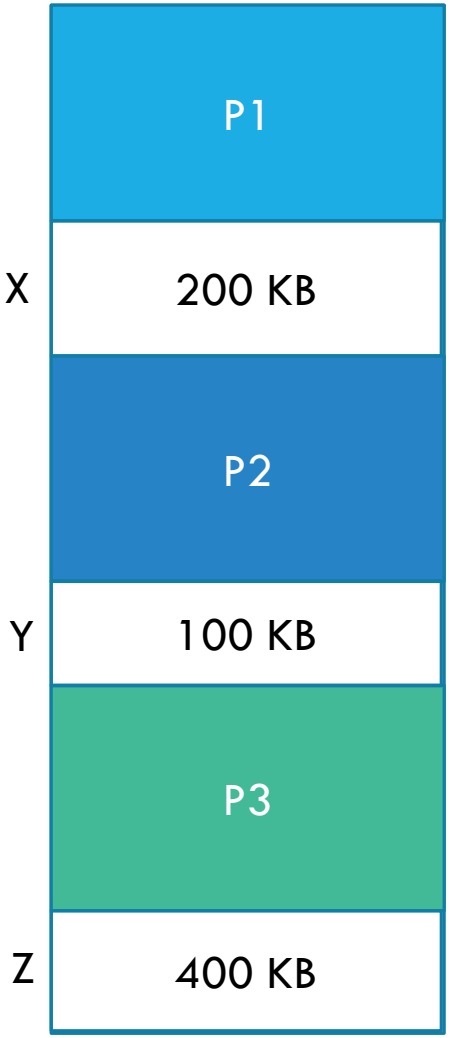


Free List

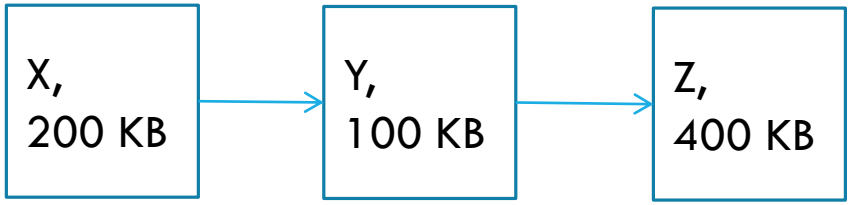


Request by P4

WORST FIT

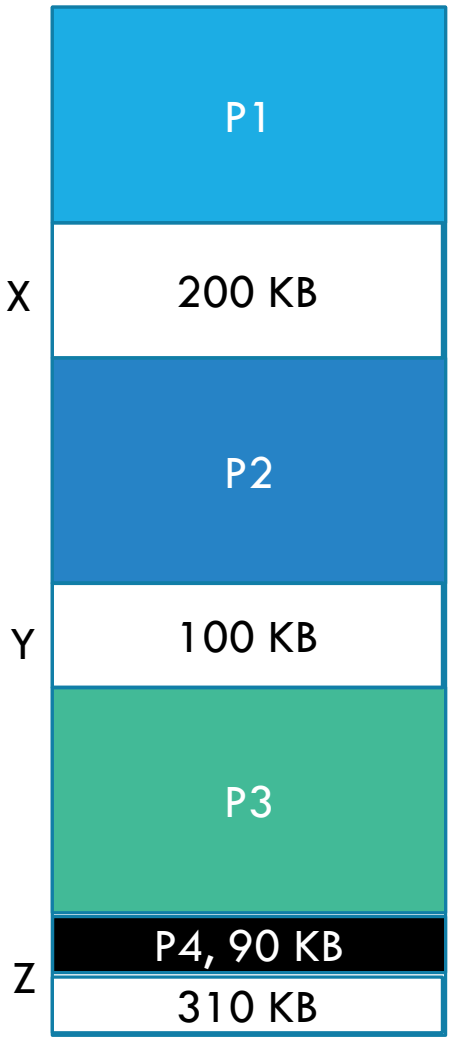


Free List

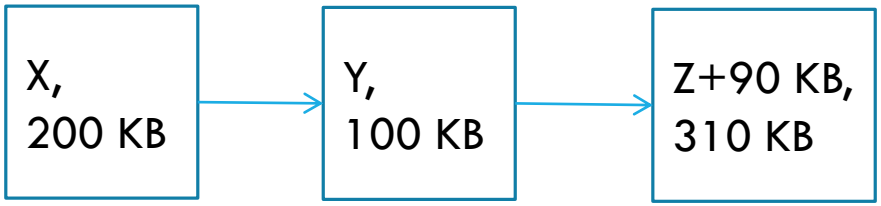


Request by P4

WORST FIT

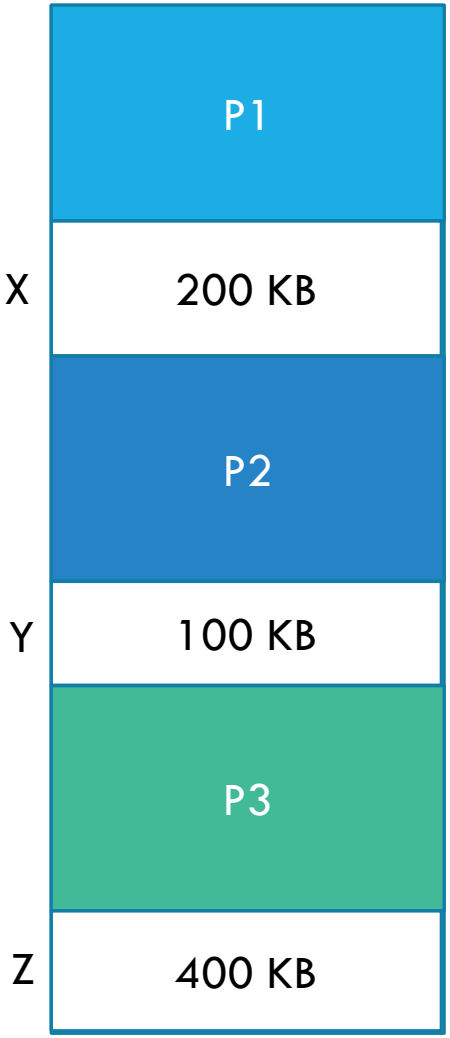


Free List



Request by P4

EXTERNAL FRAGMENTATION



Request by P4

What to do?

External Fragmentation:

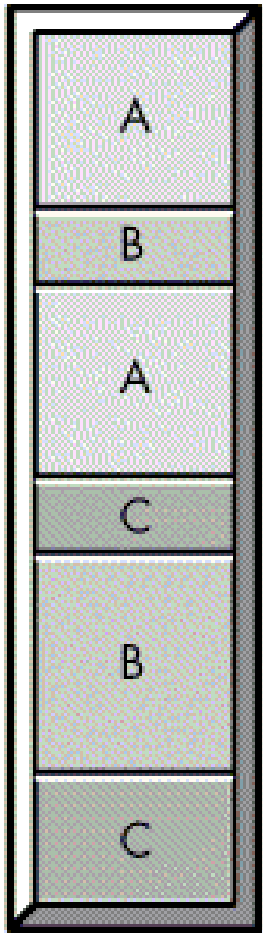
- **Allocatable memory > Requested Memory,** But no contiguous block large enough…

Proper definition:

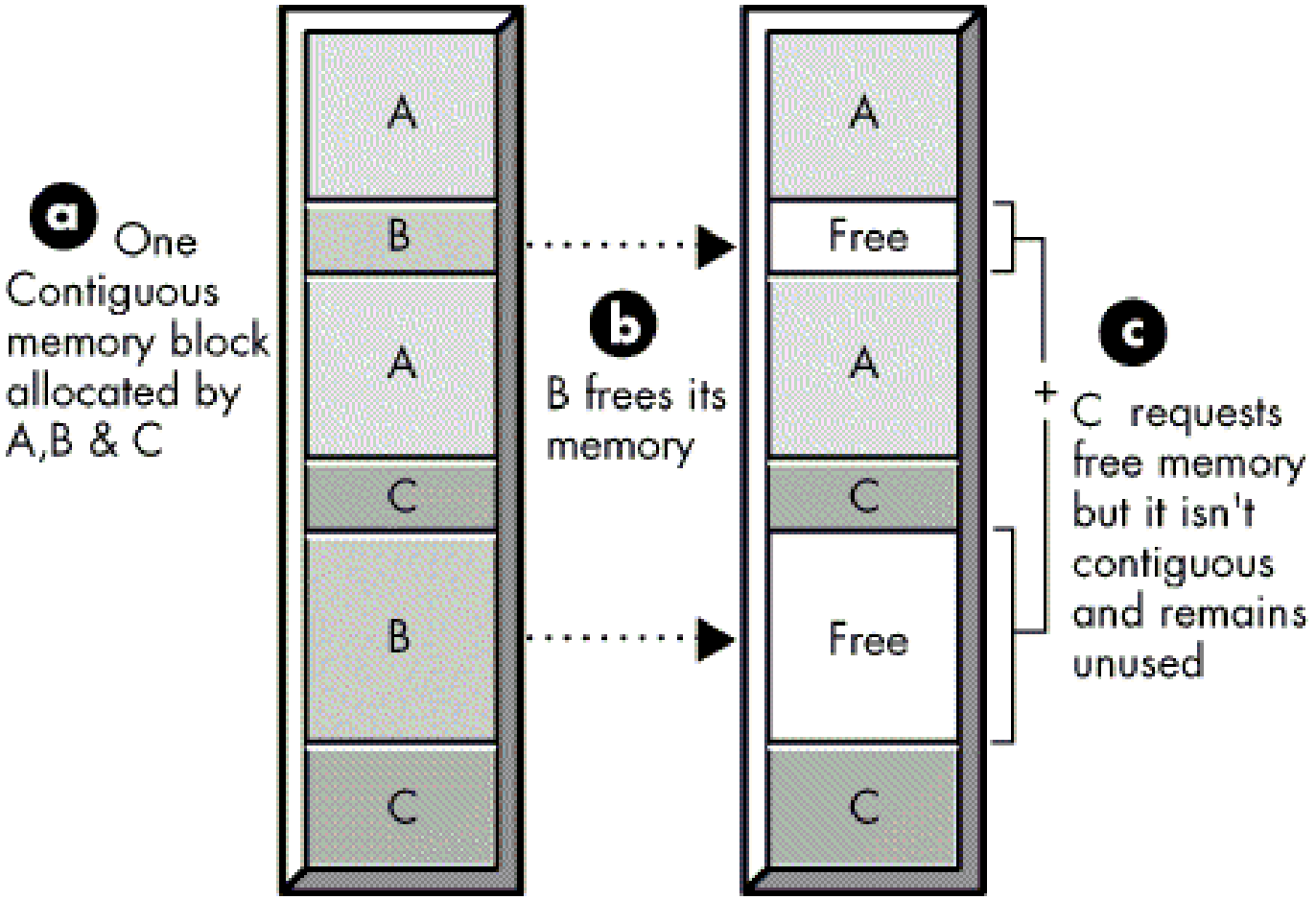
External fragments of memory exists outside allocated regions that *may* be unallocatable for a specific request within the size of free memory.

EXTERNAL FRAGMENTATION

(a) One Contiguous memory block allocated by A,B & C



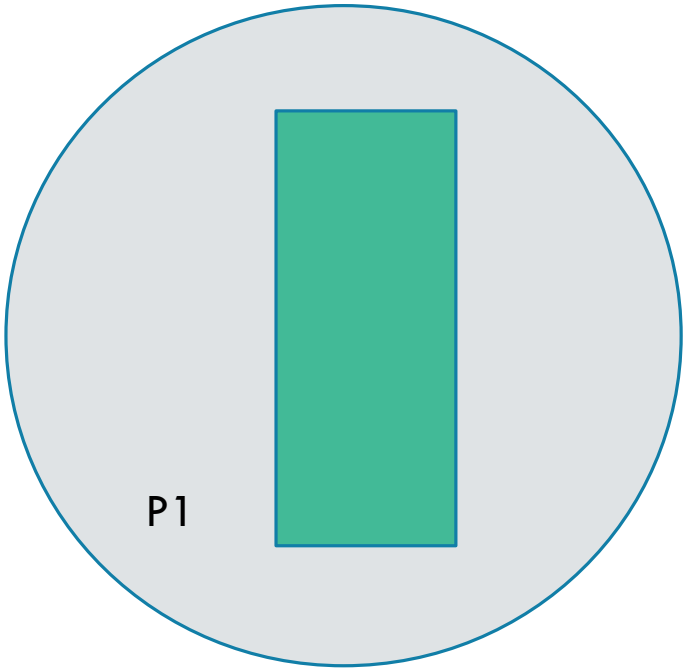
EXTERNAL FRAGMENTATION



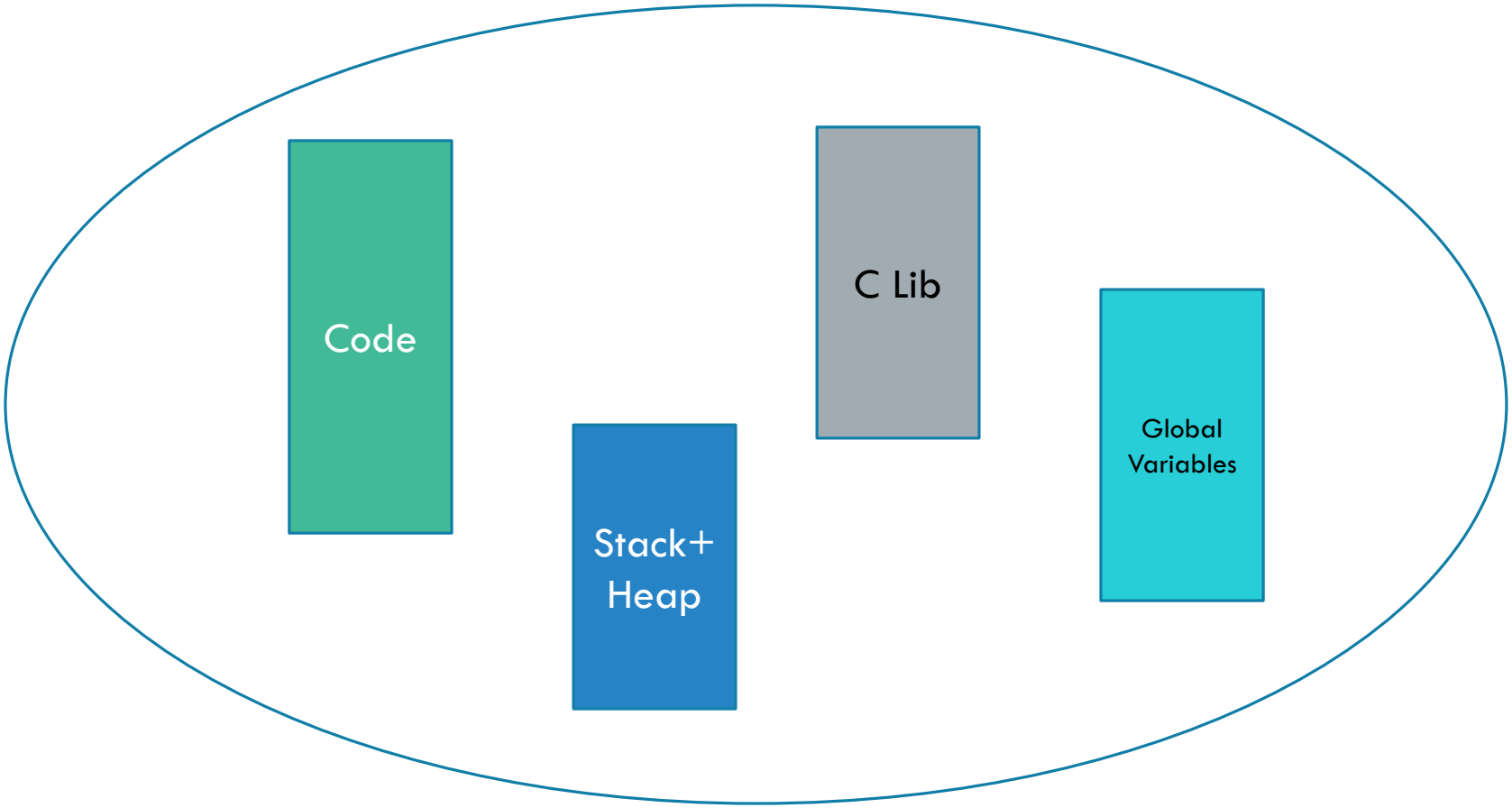
External fragmentation occurs when memory is divided into variable size partitions based on the size of processes

SEGMENTATION INTRO - I

- Programmers think of process memory allocation in terms of **"regions of different size"**
  - Text → executable code
  - Data → initialized/uninitialized global and static variables
  - Stack → function call management, local variable storage
  - Heap → dynamic memory allocation
  - Etc
- Not as just **one contiguous area !!**

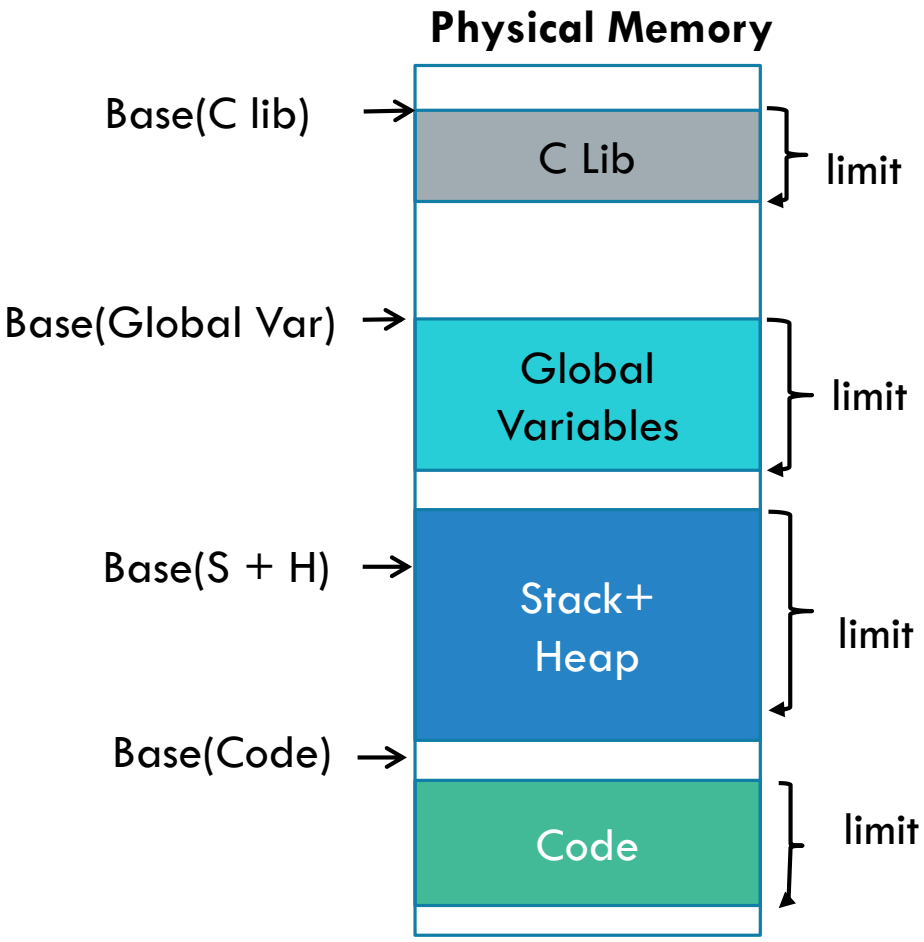


SEGMENTATION INTRO - II



Idea: Why don't we have multiple contiguous memory segments instead?

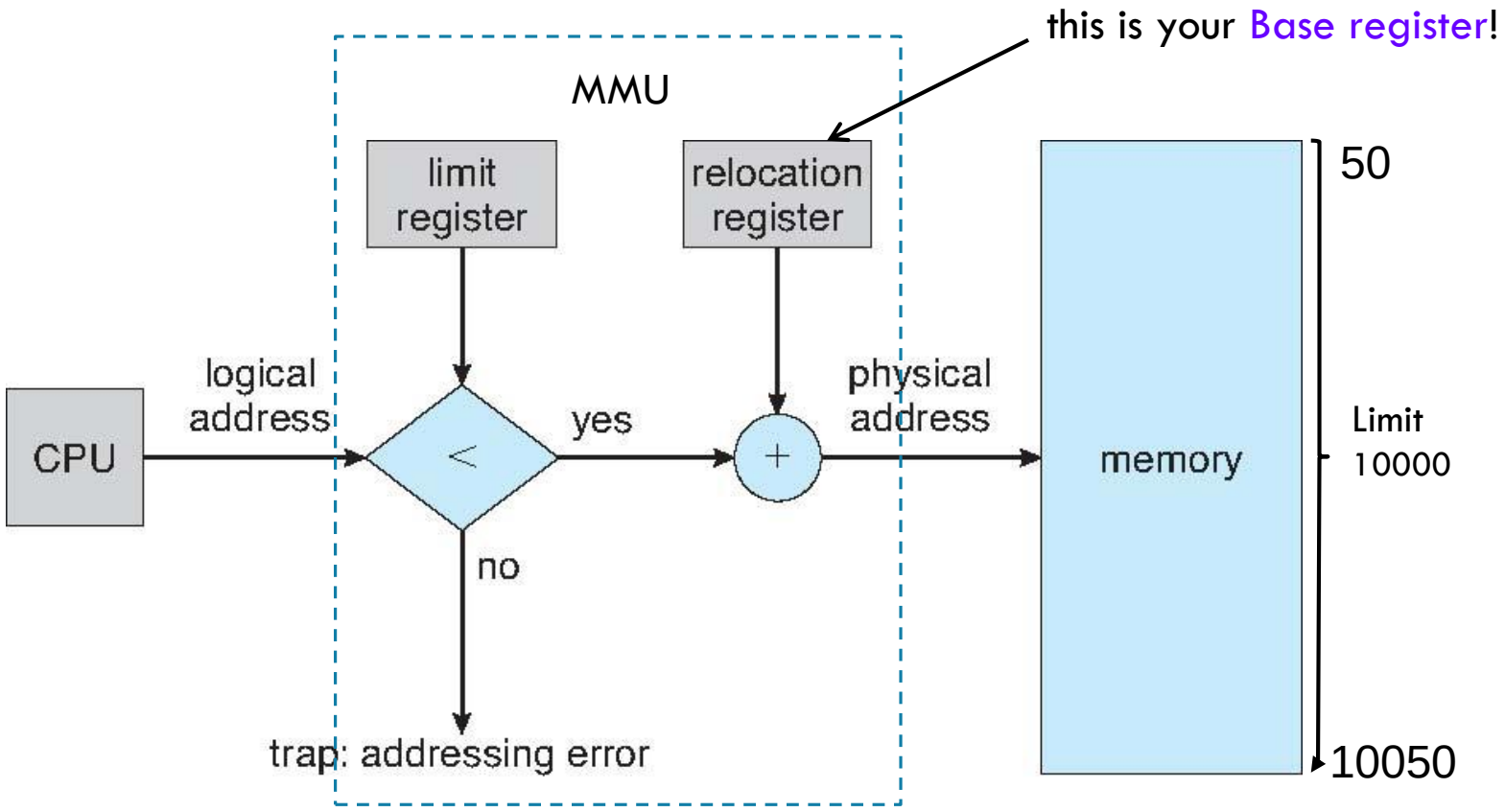
SEGMENTATION INTRO - III



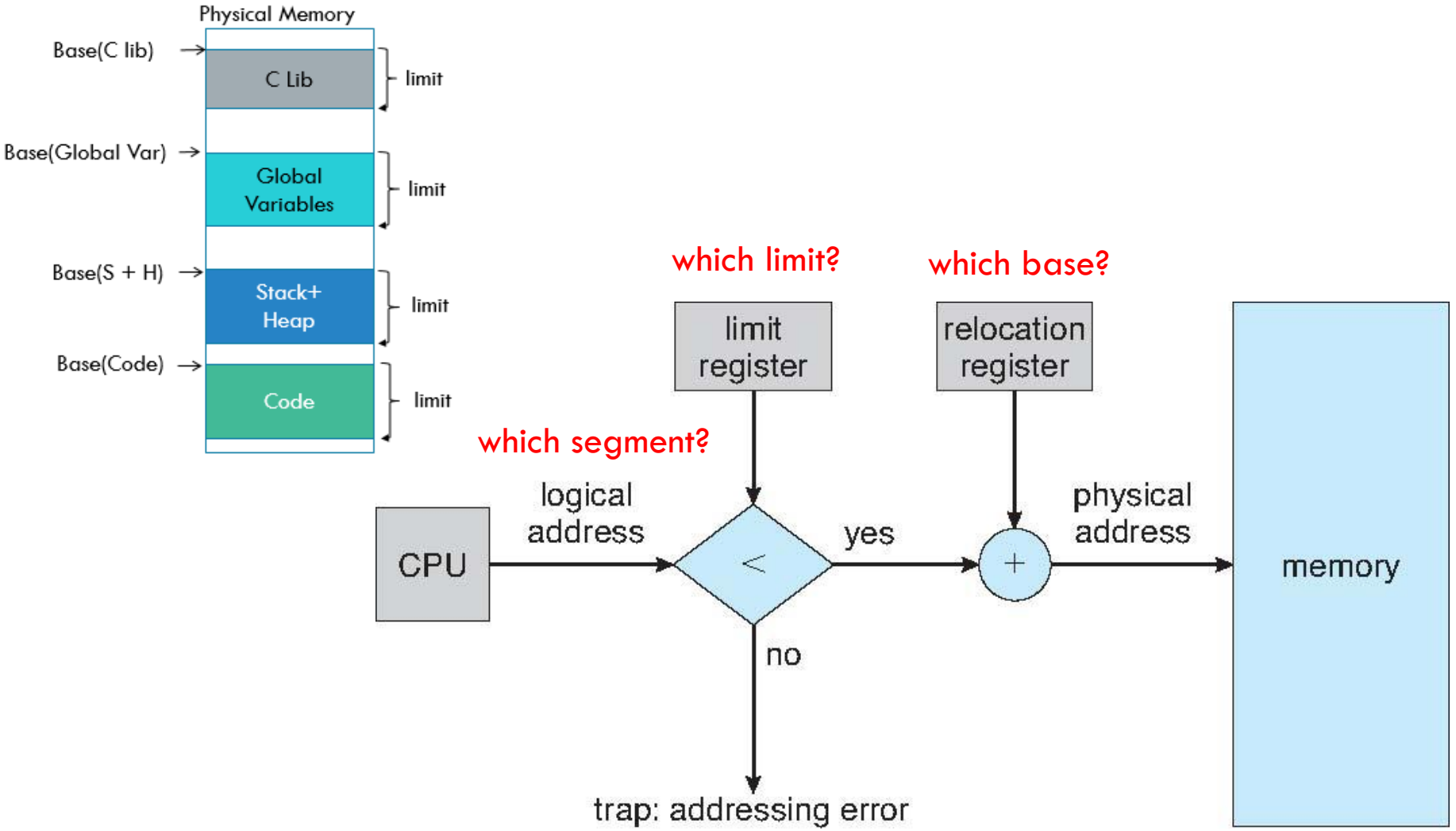
- Each process has multiple "segments" of different sizes.
  - Each segment has its own Base and Limit registers

SIMPLE CONTIGUOUS MEMORY ALLOCATION

More suitable for a **single process represented as a single segment**



DOES THIS SETUP STILL WORK FOR SEGMENTATION?



SEGMENTATION LOGICAL ADDRESS

<Segment-number, offset>

1. Instead of a **single number**, the segmentation logical address is a **tuple.**

2. Each segment has a **unique** segment number.

SEGMENT TABLE

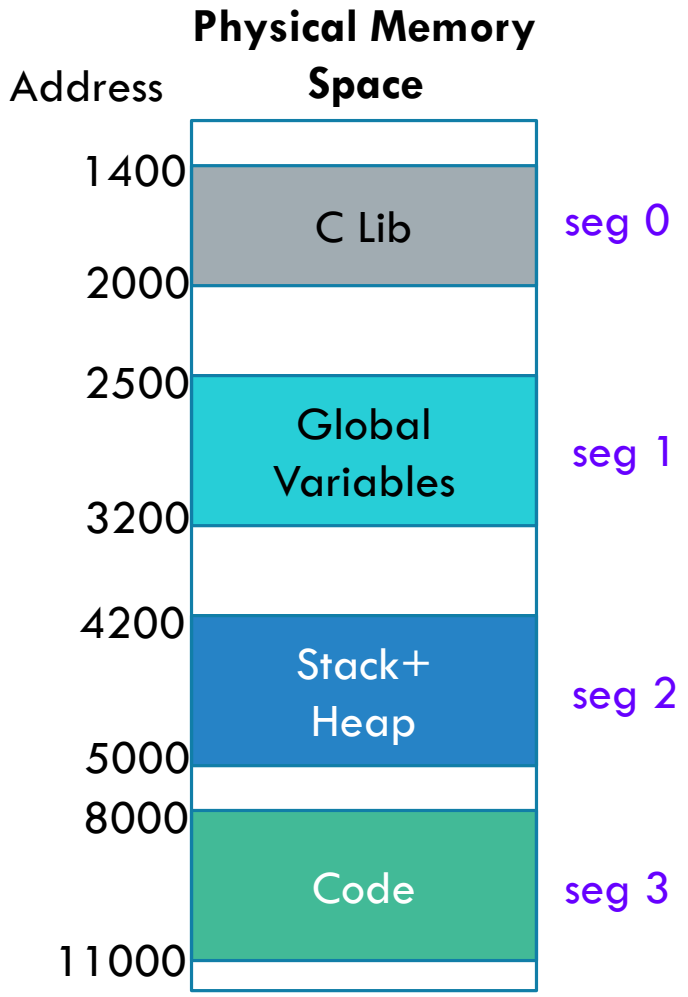
A table that records the **Limits** and **Bases** for each of the segments

Seg. No.	Limit	Base
0	600	1400
1	700	2500
2	800	4200
3	3000	8000

When a program tries to access memory using a segmentation logical address:

1. The operating system or hardware uses the Segment Number from the logical address to **index into the segment table**.
2. The **entry retrieved** from the segment table contains the Base Address and Limit for the selected segment.
3. The Offset from the logical address is then **added to the Base Address** to calculate the physical address in memory.

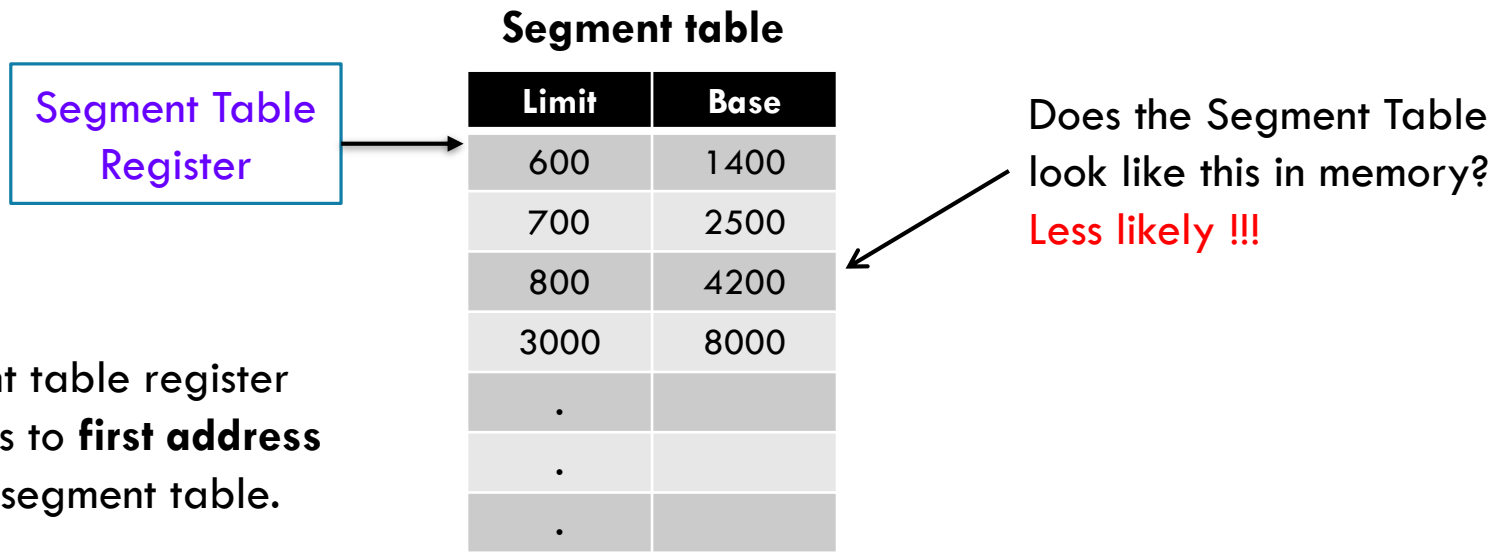
<Segment-number, offset>



REMAINING ISSUES

- Where are these segment tables stored?
  - RAM (Memory)
- How many segment tables per process?
  - Local Segment Table per process
  - Global Segment Table shared by all processes (ex:-shared library)
- Does the MMU read the segment table for translation?
  - Yes… but which segment table does it read?
  - **How does a hardware MMU know the address of the segment tables? !!!**

SEGMENT TABLE REGISTER

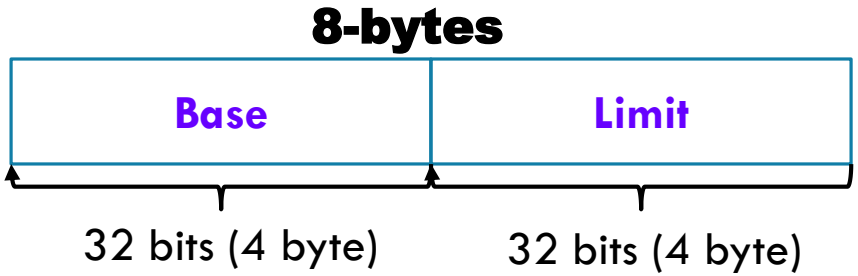


Segment table register

- Points to **first address** of a segment table.

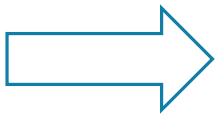
SEGMENT TABLE ENTRY

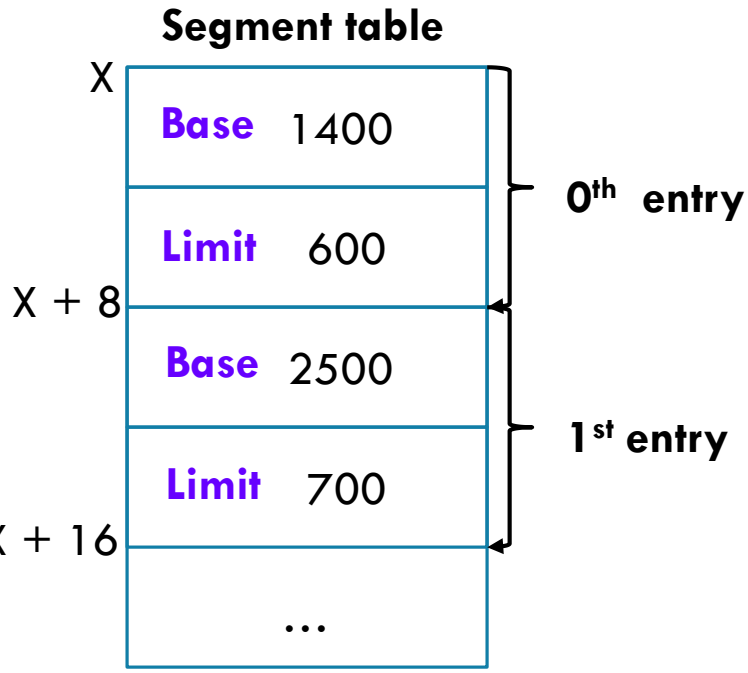
In our setup, a segment table entry has 2 values – **Base** and **Limit.**



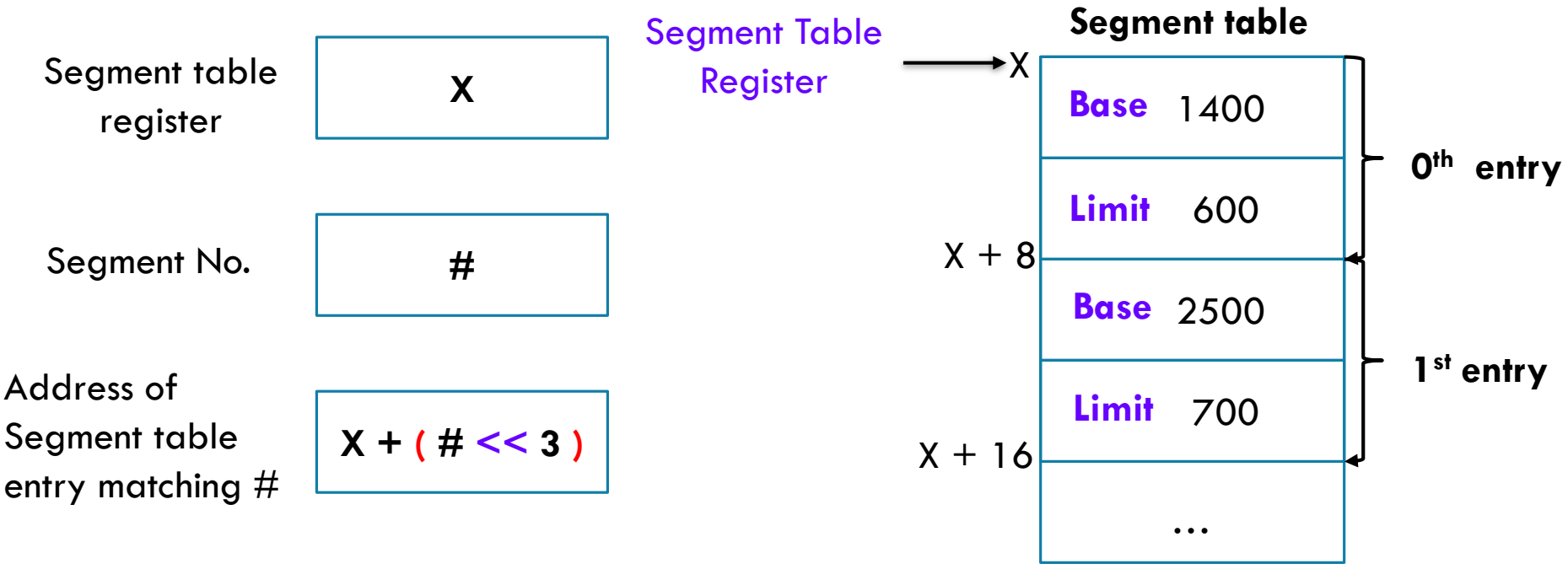
Seg. No.	Limit	Base
0	600	1400
1	700	2500
2	800	4200
3	3000	8000

Segment table

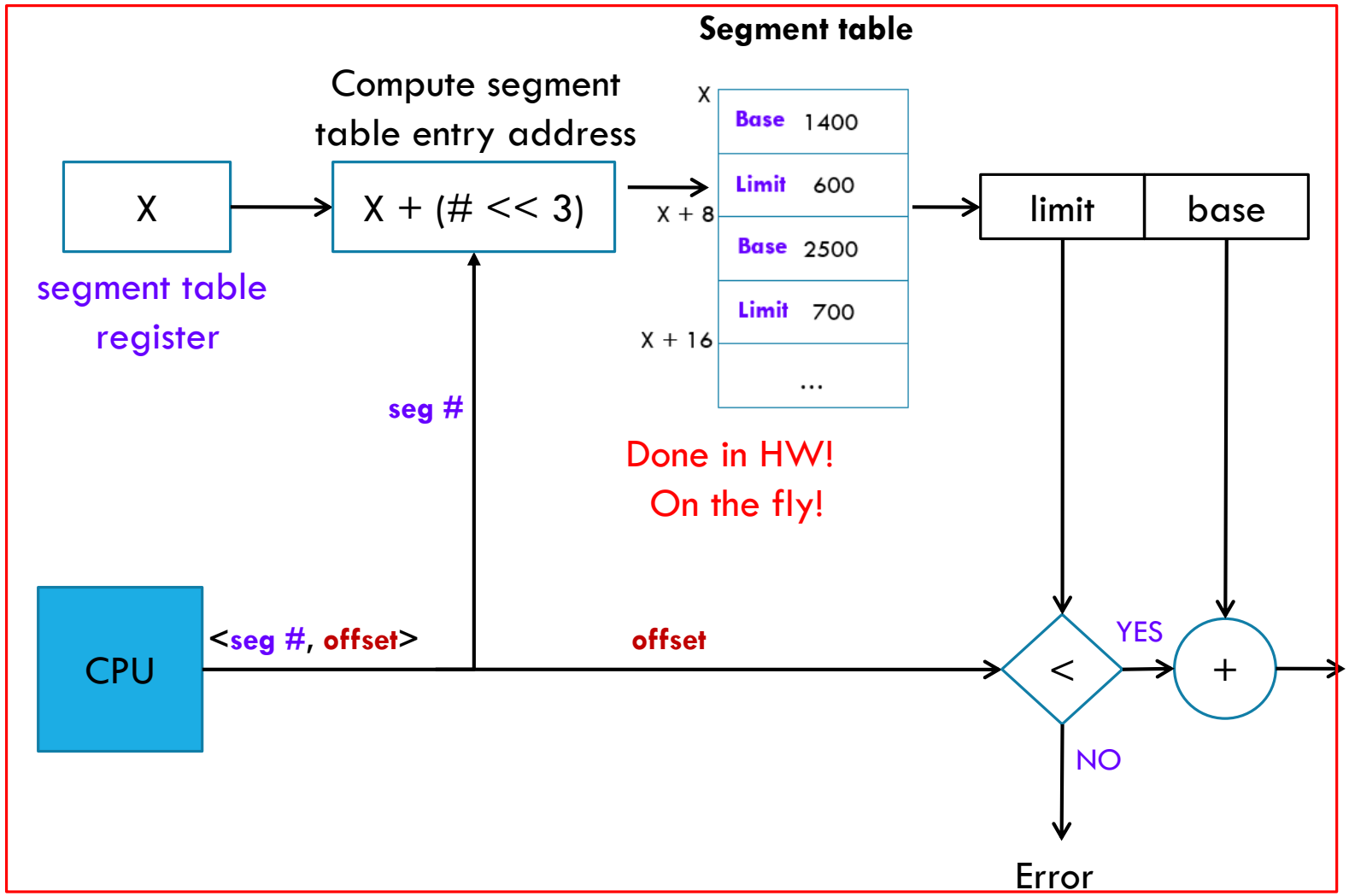




USING SEGMENT TABLE REGISTER + SEGMENT NUMBER TO ACCESS SEGMENT TABLE ENTRIES



TRANSLATING LOGICAL ADDRESS TO PHYSICAL ADDRESS



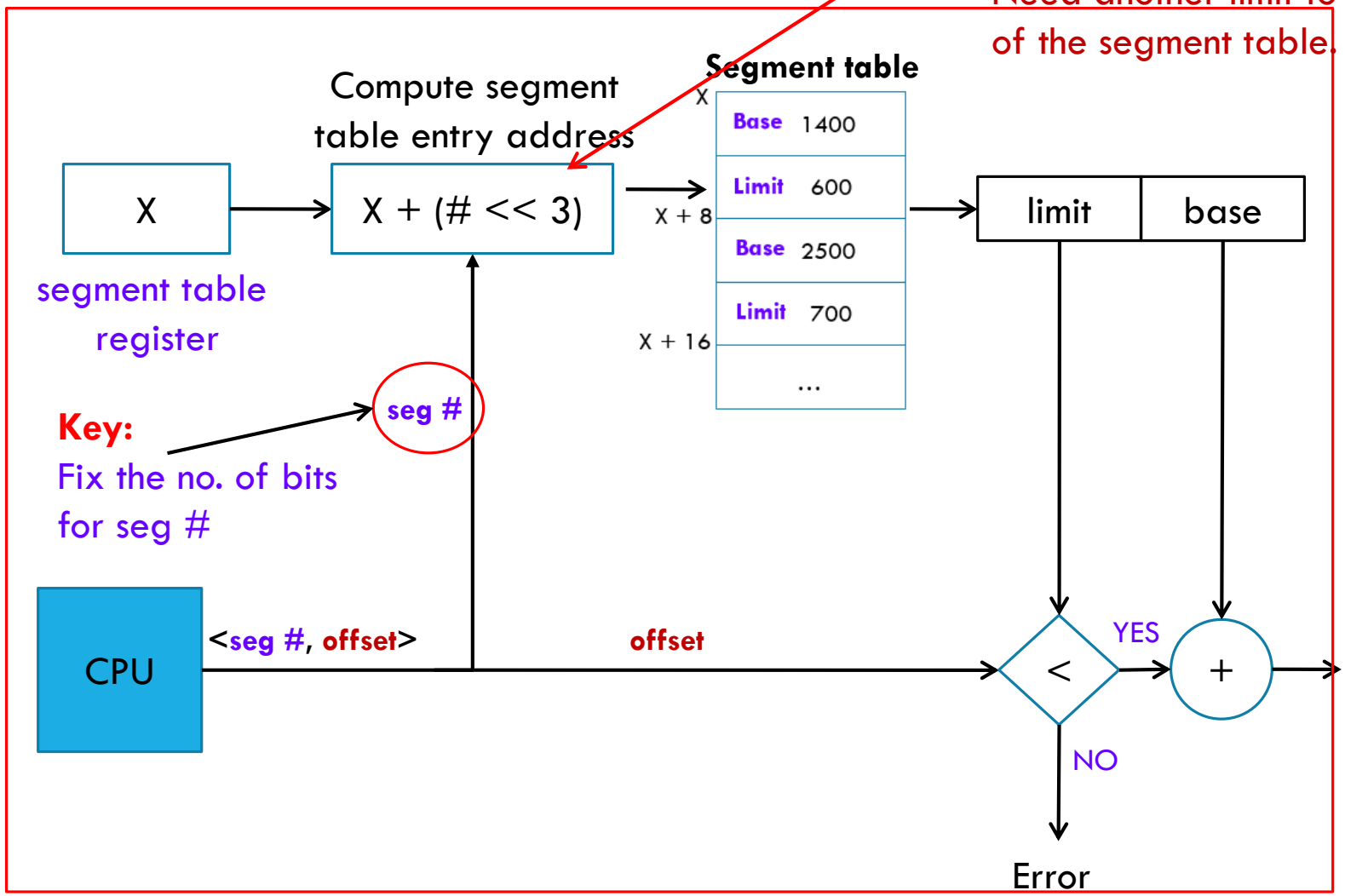
SEGMENT TABLE SIZE

- Should the size be infinite?
  - This is crazy of course.
- Then.. Should the size be variable or fixed?
  - What does variable size mean?
    - As many segments as required by the process
  - What does fixed size mean?
    - Well.. There's a limit to number of segments a process can have.

VARIABLE SEGMENT TABLE SIZE

- No way to tell whether this address **exceeds the segment table area.**
- Need another limit to limit the size of the segment table.

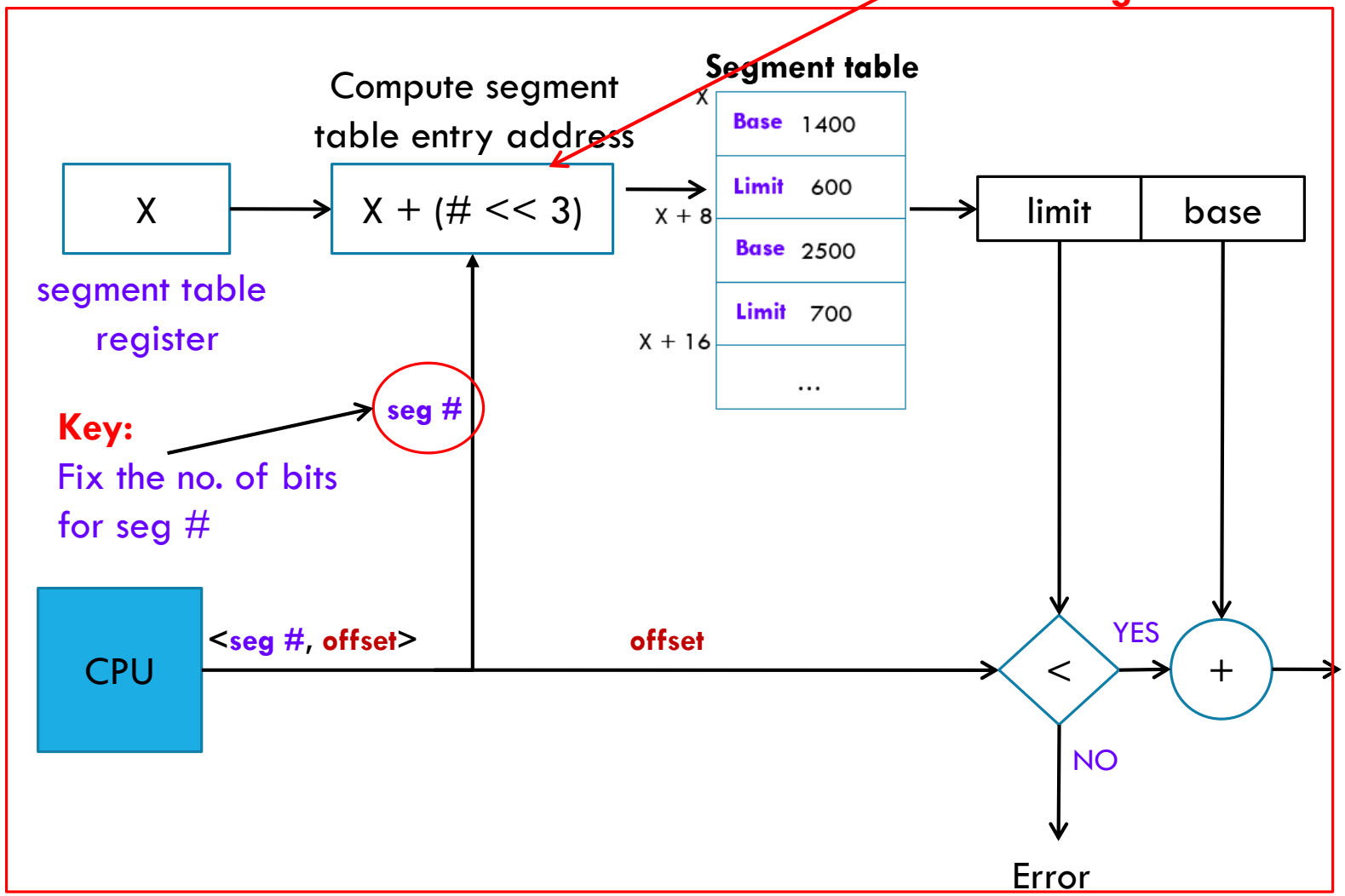
Does this still work then?



FIXED-SIZED SEGMENT TABLE

Does this still work then? … Basically, No.

As before, how does this ensure that we don't **exceed the segment table size?**



FIX NUMBER OF BITS?

Basic CS100 stuff.

If X is unsigned N bits.

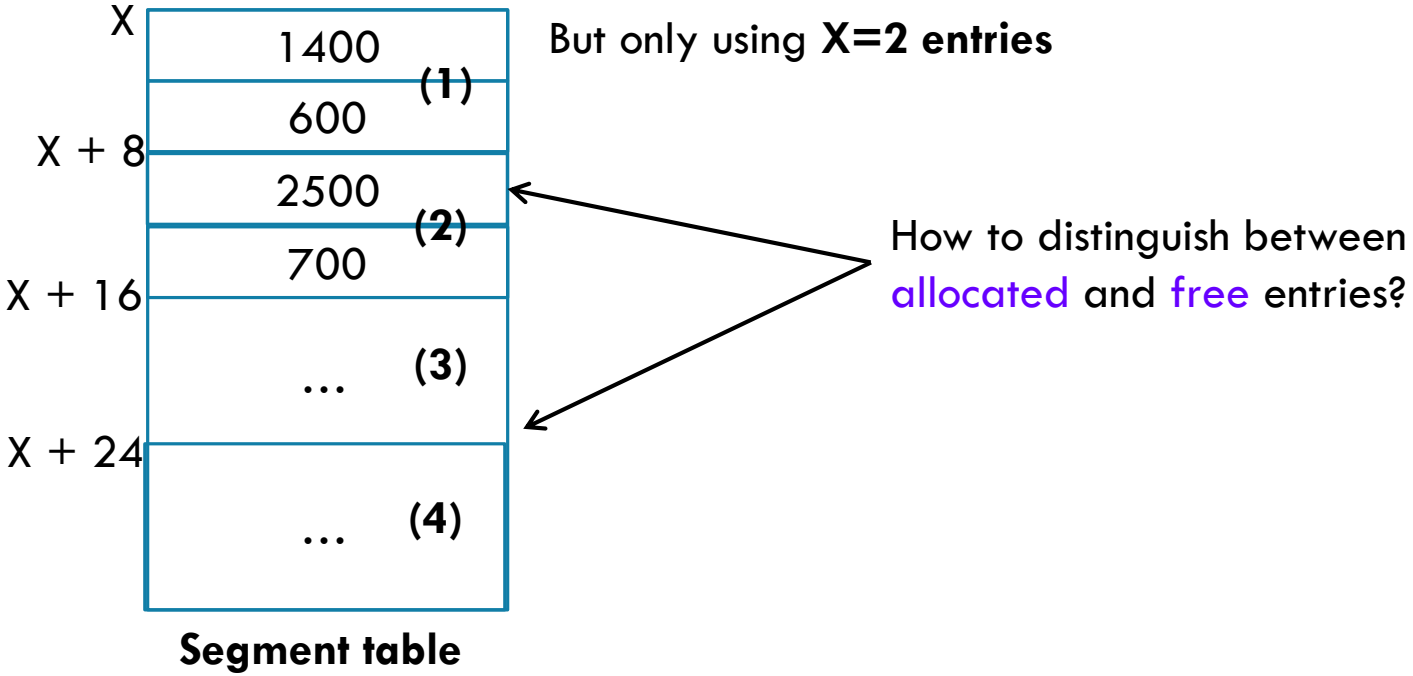
$0 <= X <= 2^N-1$

So, if you fix no. of bits,
You implicitly fix the Segment Table Size.

OK… BUT WHAT IF FIXED SIZE IS Y ENTRIES, I USE ONLY X < Y ENTRIES?

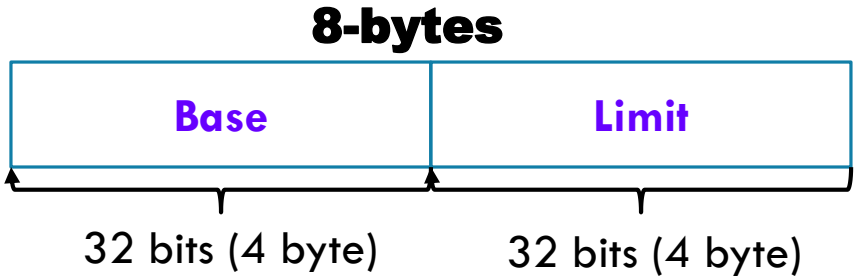
Recall that the segment table is just a series of numbers…

Say the size of the segment table is **Y=4 entries.** And you are using only **2 entries**?

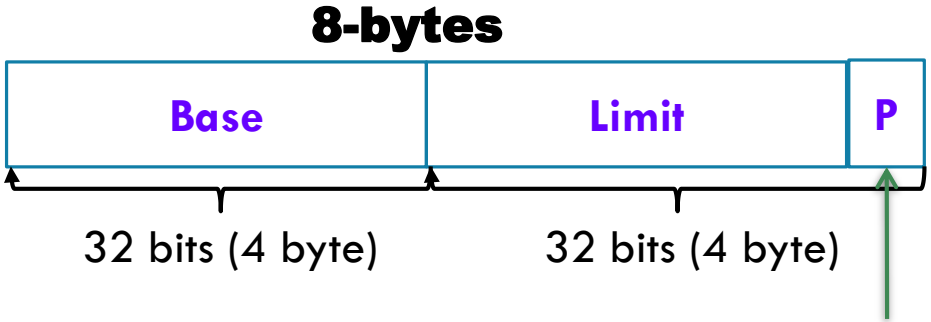


IMPROVED SEGMENT TABLE ENTRY

Recall: Old Segment Table Entry

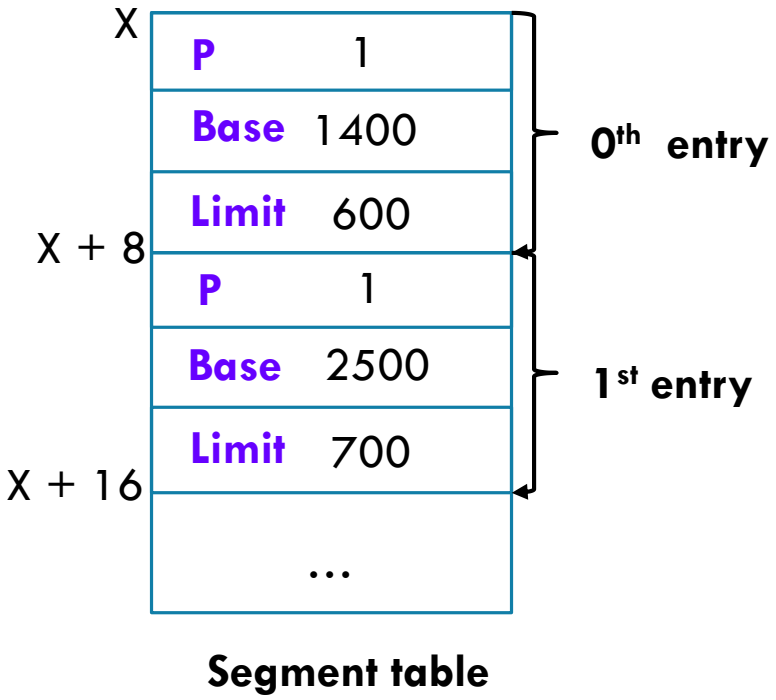


Improved Segment Table Entry

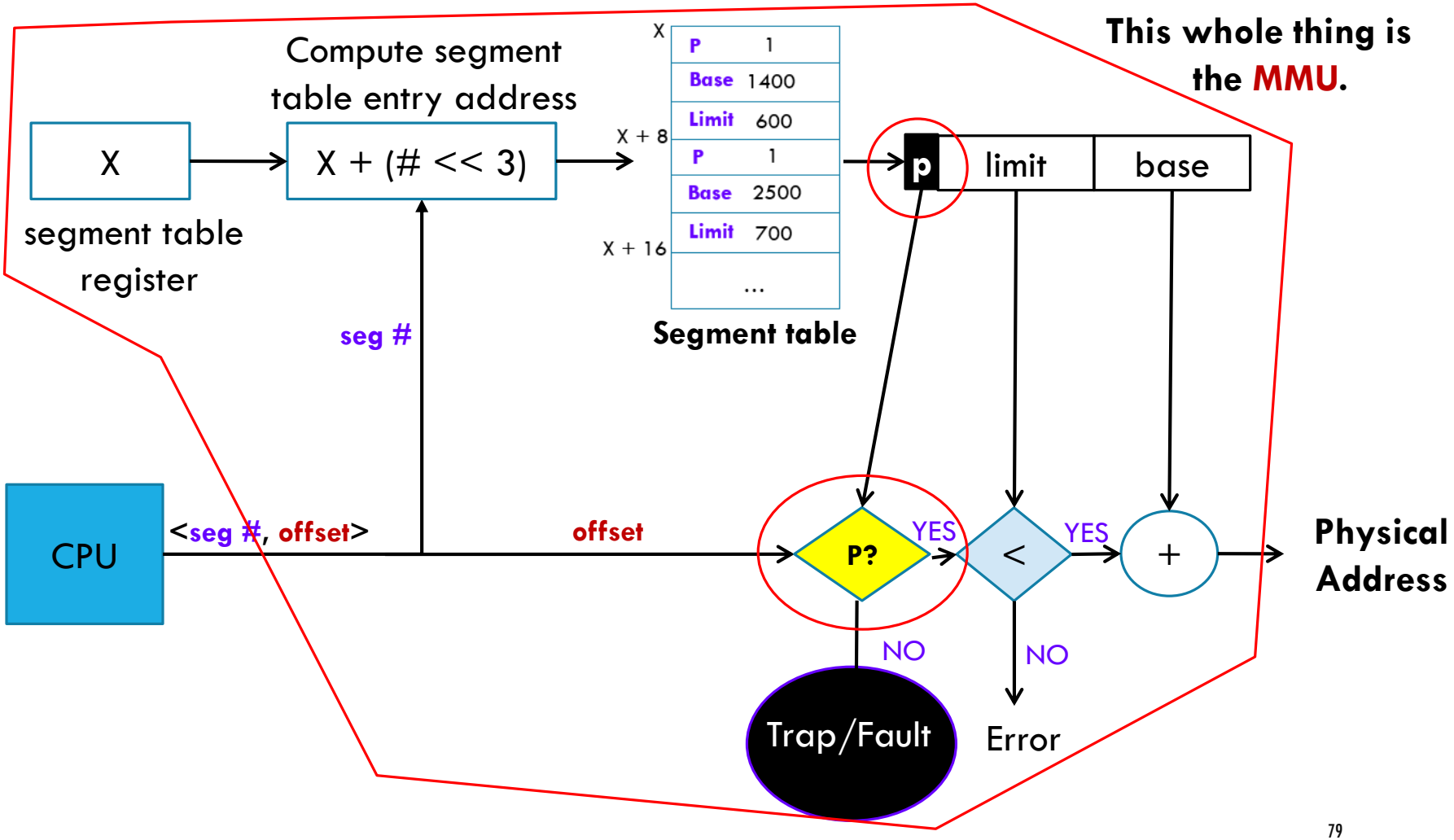


- **P** for present, 1 bit value
- **0** means segment not in memory
- **1** means segment in memory
- **0** is a **default** value.

USING SEGMENT TABLE REGISTER + SEGMENT NUMBER TO ACCESS SEGMENT TABLE ENTRIES

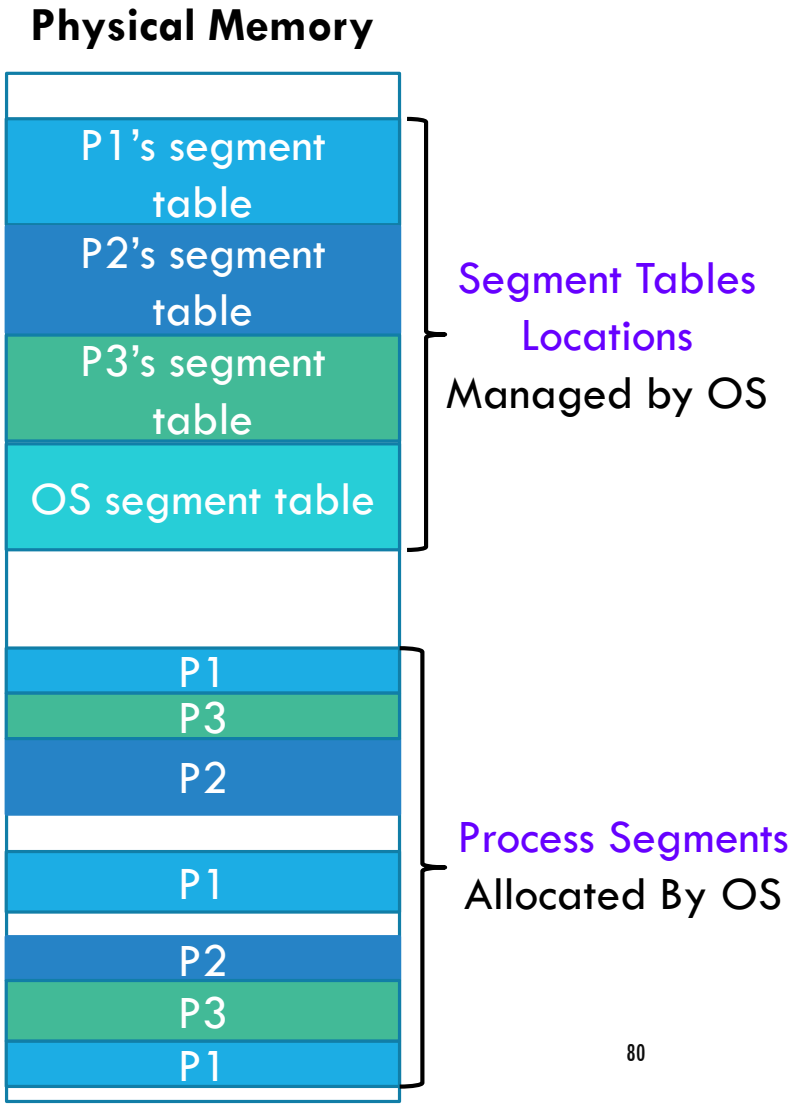


IMPROVED LOGICAL ADDRESS TRANSLATION



SEGMENTATION AND CONTEXT SWITCHING

1. Suppose **P1** is running now.
   Segment Table Register = P1's segment table starting address
2. Interrupt happens!
   A. **P1's context is saved.**
      - Save P1's segment table starting address to P1's PCB
   B. Set Segment Table Register = **OS's** segment table starting address
3. Scheduler decides to run **P2**
   A. **Restore P2's context**
      - Get P2's segment table address from P2's PCB.
      - Set segment table register = P2's segment table starting address



IN PRACTICE

Segment Logical Addressing is slightly more complex

- 1 more bit is needed to indicate whether accessing the local or global segment table

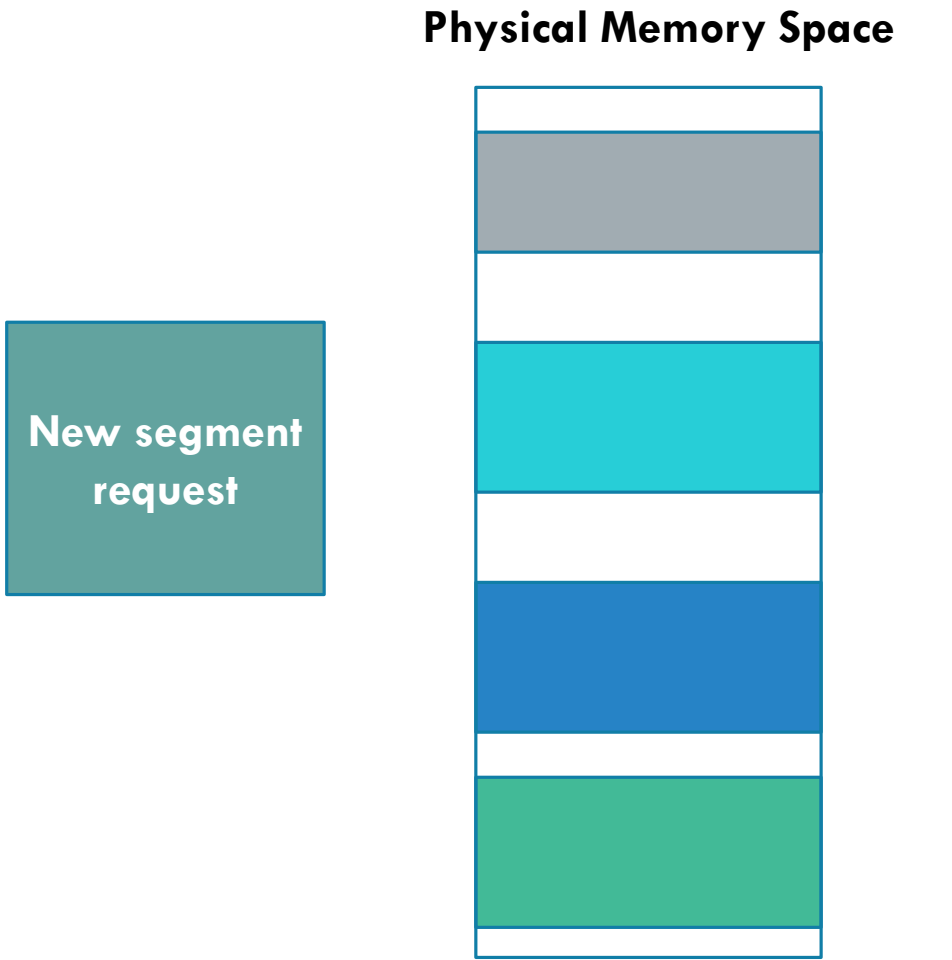
Segment Table Entry is more complex

- Include fields such as privilege, permissions, etc.

Caching segment table entries

- MMU can cache entries to save on memory lookups.

RUNNING OUT OF PHYSICAL MEMORY SPACE

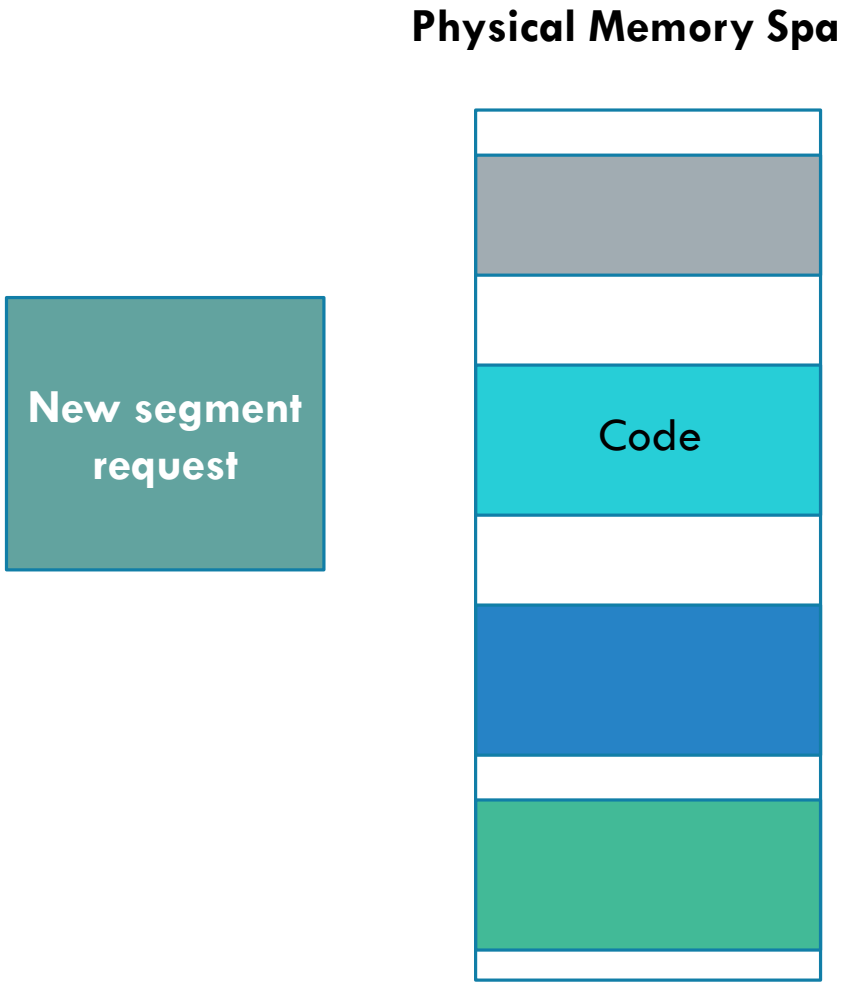


Cannot fit!! What to do?

Options:

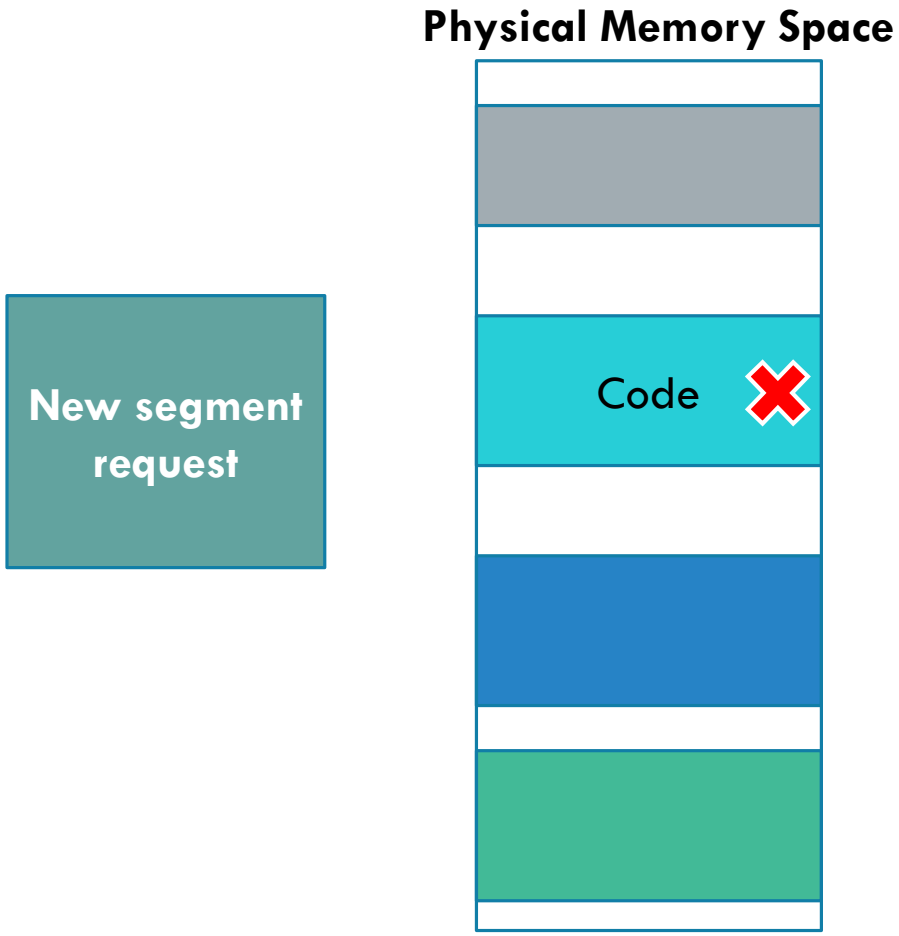
1. *Reject* the request
2. *Delete* some **read-only segments**
3. *Move* some segments into the hard disk.

DELETE SOME READ-ONLY SEGMENTS

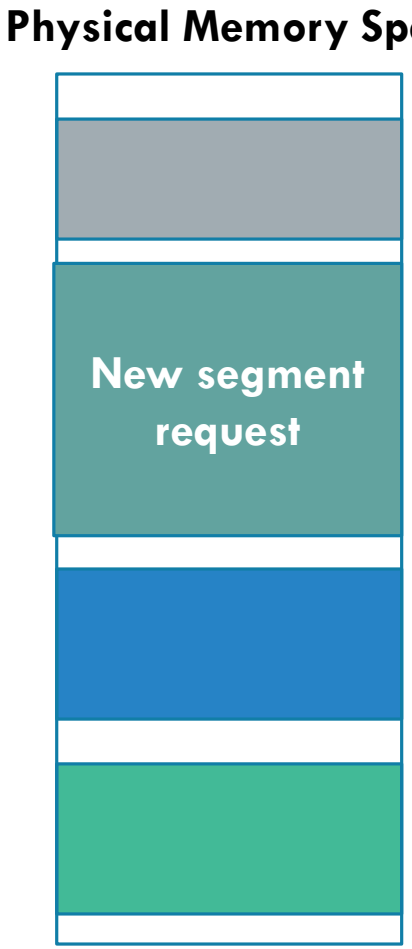


- Usually, this means removing some code segments
  - Whose code?
    - If the request is P1's, *usually* some non-P1's code.
  - Why code?
    - If P2's code is missing when it's needed, OS can recover P2's code from permanent storage.

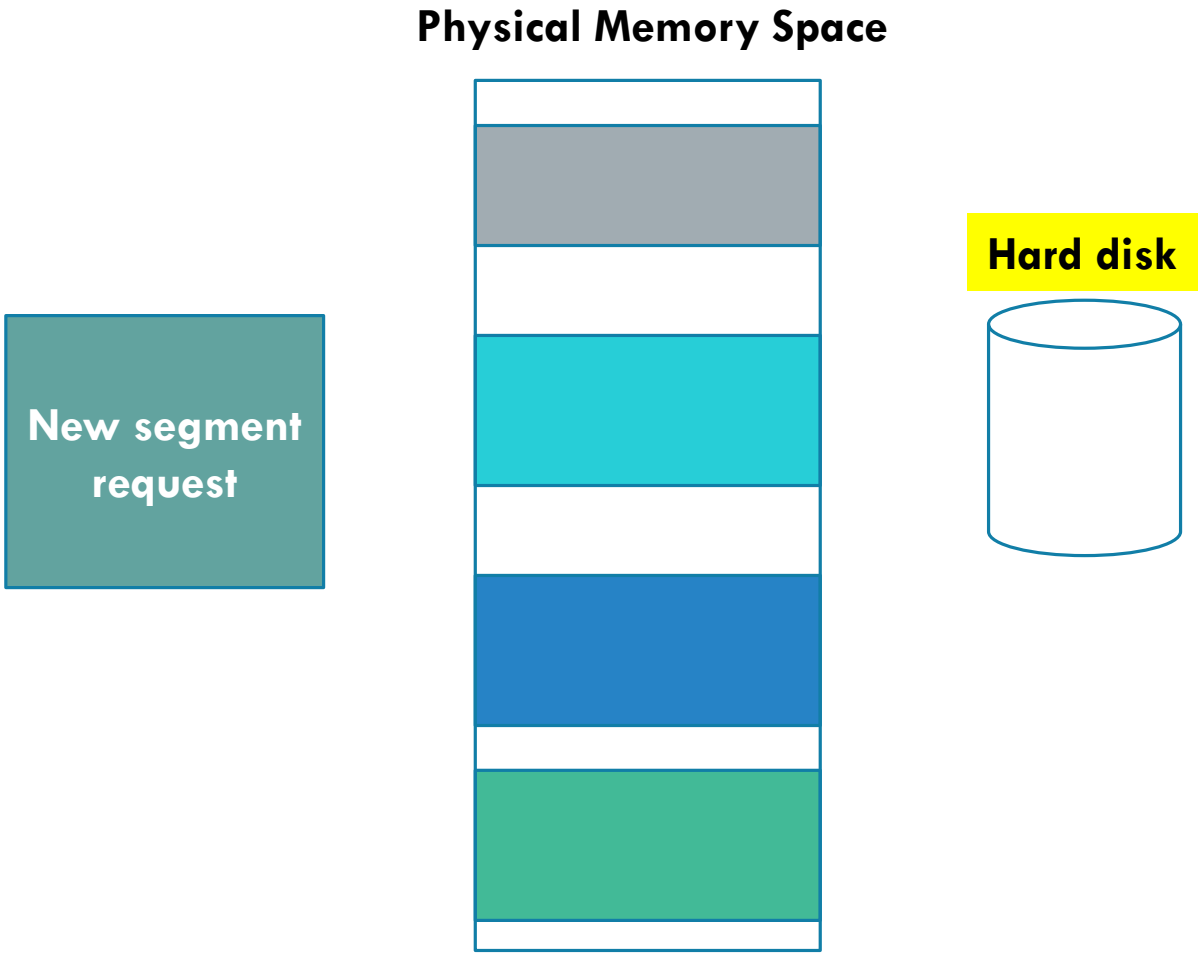
DELETE SOME READ-ONLY SEGMENTS



DELETE SOME READ-ONLY SEGMENTS



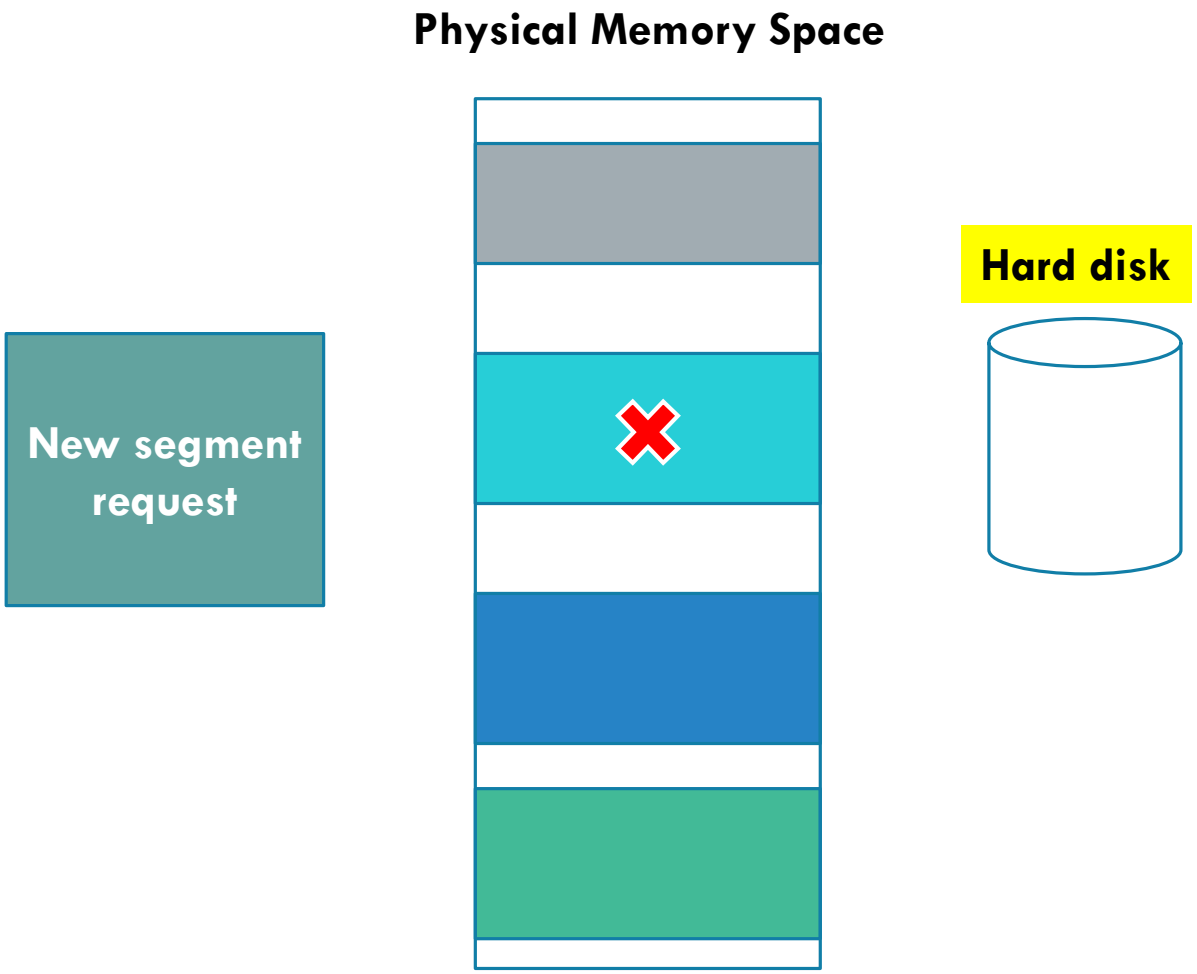
USING HARD DISK AS A BACKING STORE



OS needs to **reserve** a portion of the hard disk for storing segments when needed

(Usually some unformatted space)

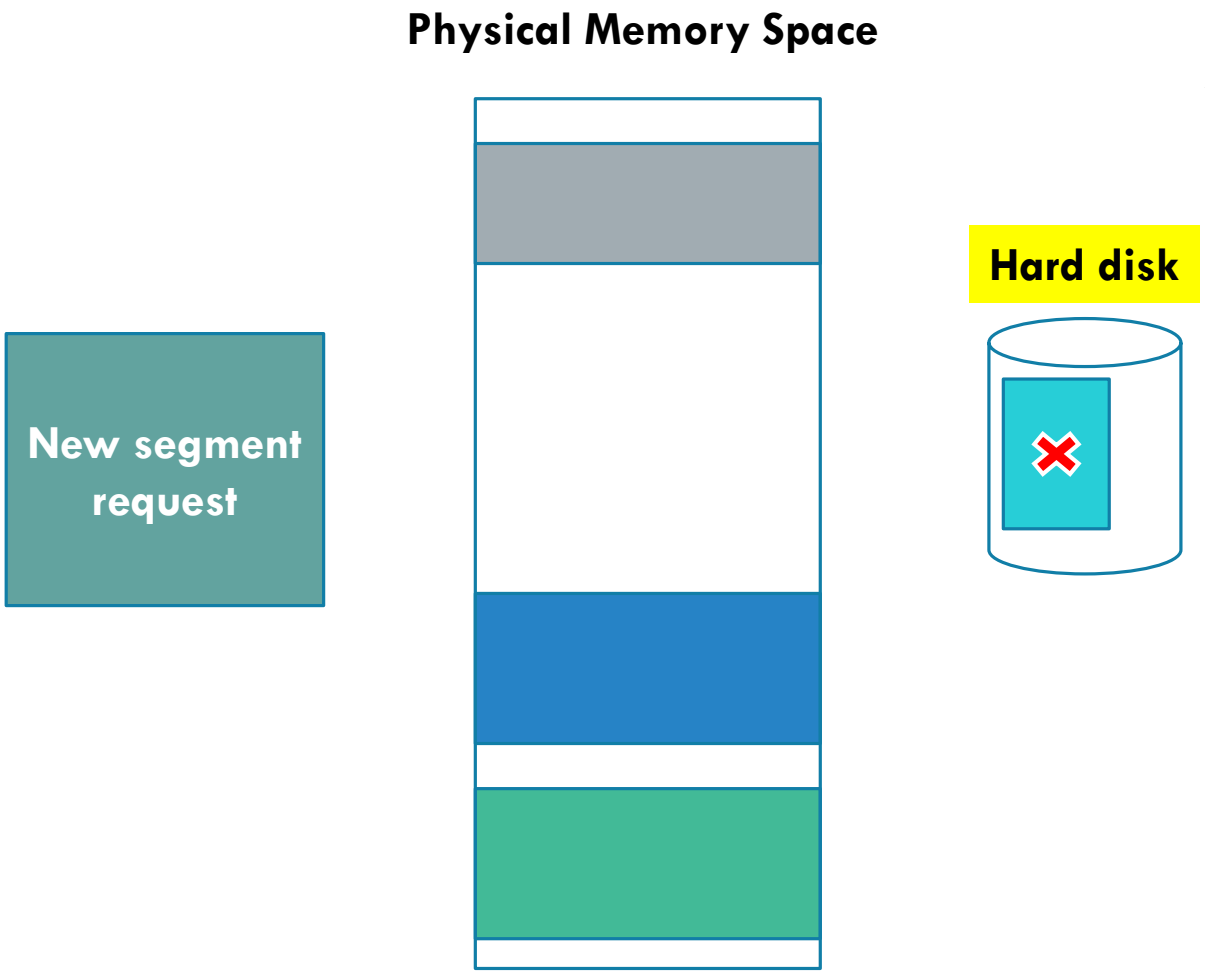
USING HARD DISK AS A BACKING STORE



What needs to be done?

1. Need to choose at least 1 segment to be put into the hard disk.

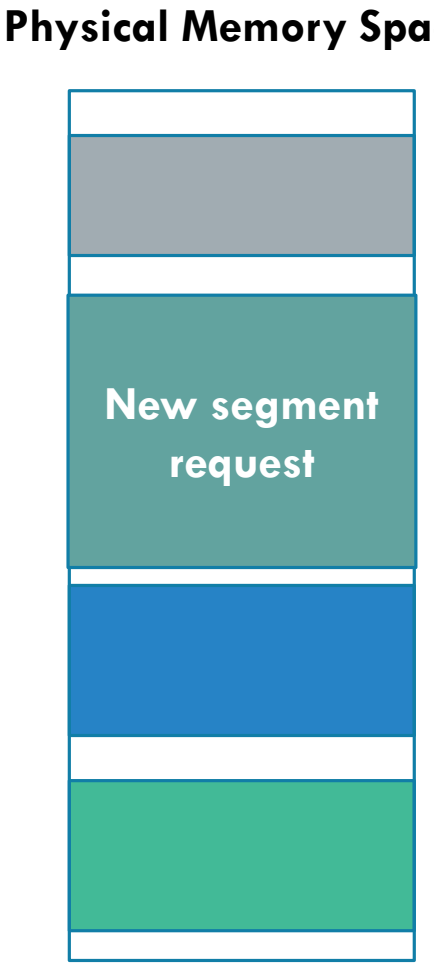
USING HARD DISK AS A BACKING STORE

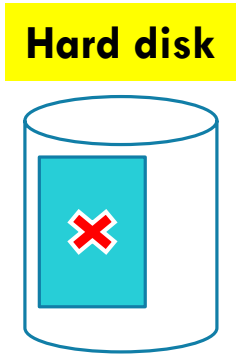


What needs to be done?

1. Need to choose at least 1 segment to be put into the hard disk.
2. For the chosen segment, the **present bit** of the corresponding segment table entry is set to **0**.
3. Move the segment into the hard disk.

USING HARD DISK AS A BACKING STORE





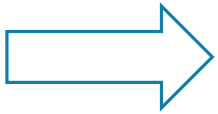
What needs to be done?

1. Need to choose at least 1 segment to be put into the hard disk.
2. For the chosen segment, the **present bit** of the corresponding segment table entry is set to **0**.
3. Move the segment into the hard disk.
4. **Create** a new segment table entry. Allocate a new segment.

ACCESSING A NON-PRESENT SEGMENT

C Code

```
…
x = y + z;
array[10] = x;
…
```

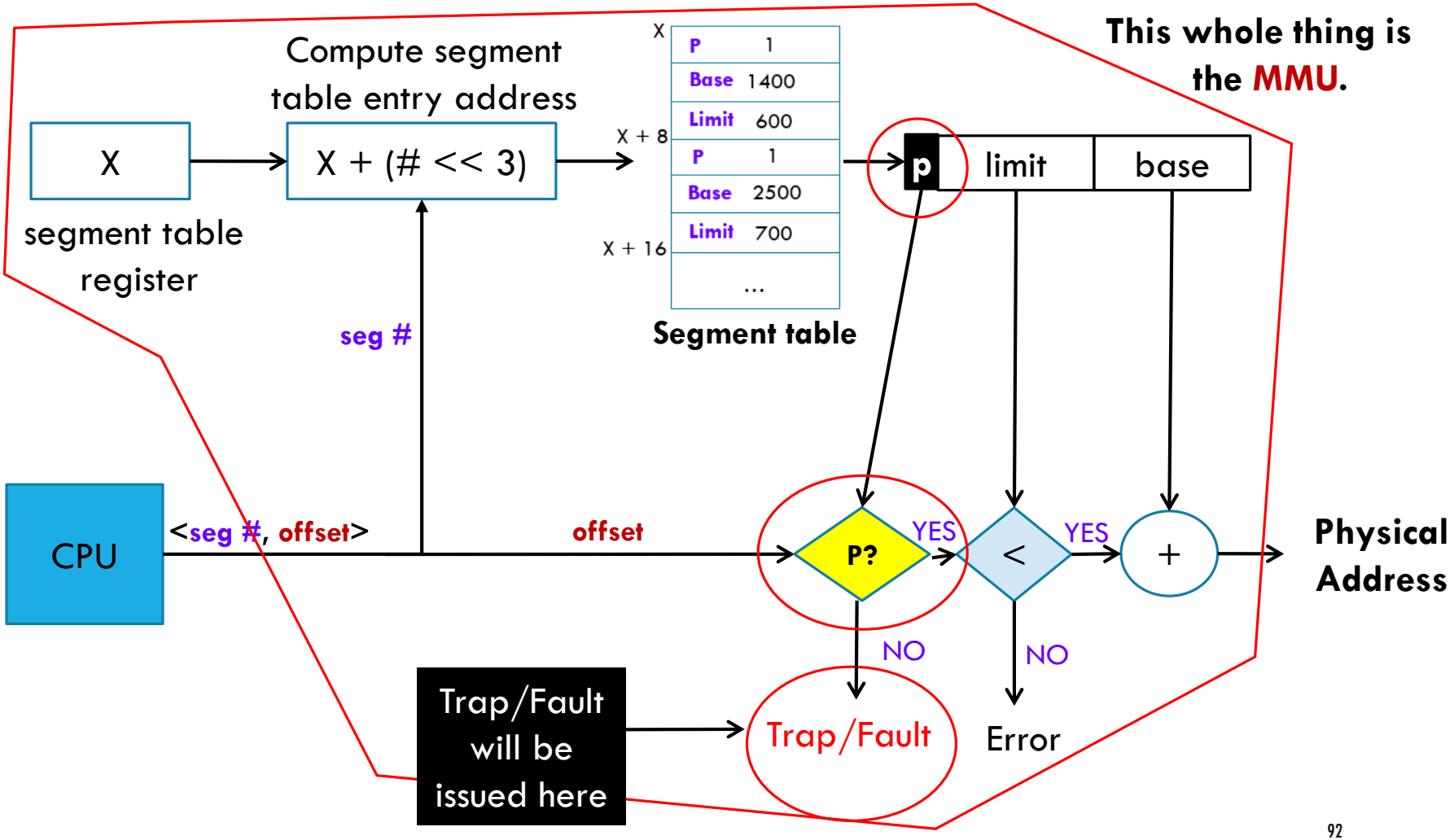


Assembly

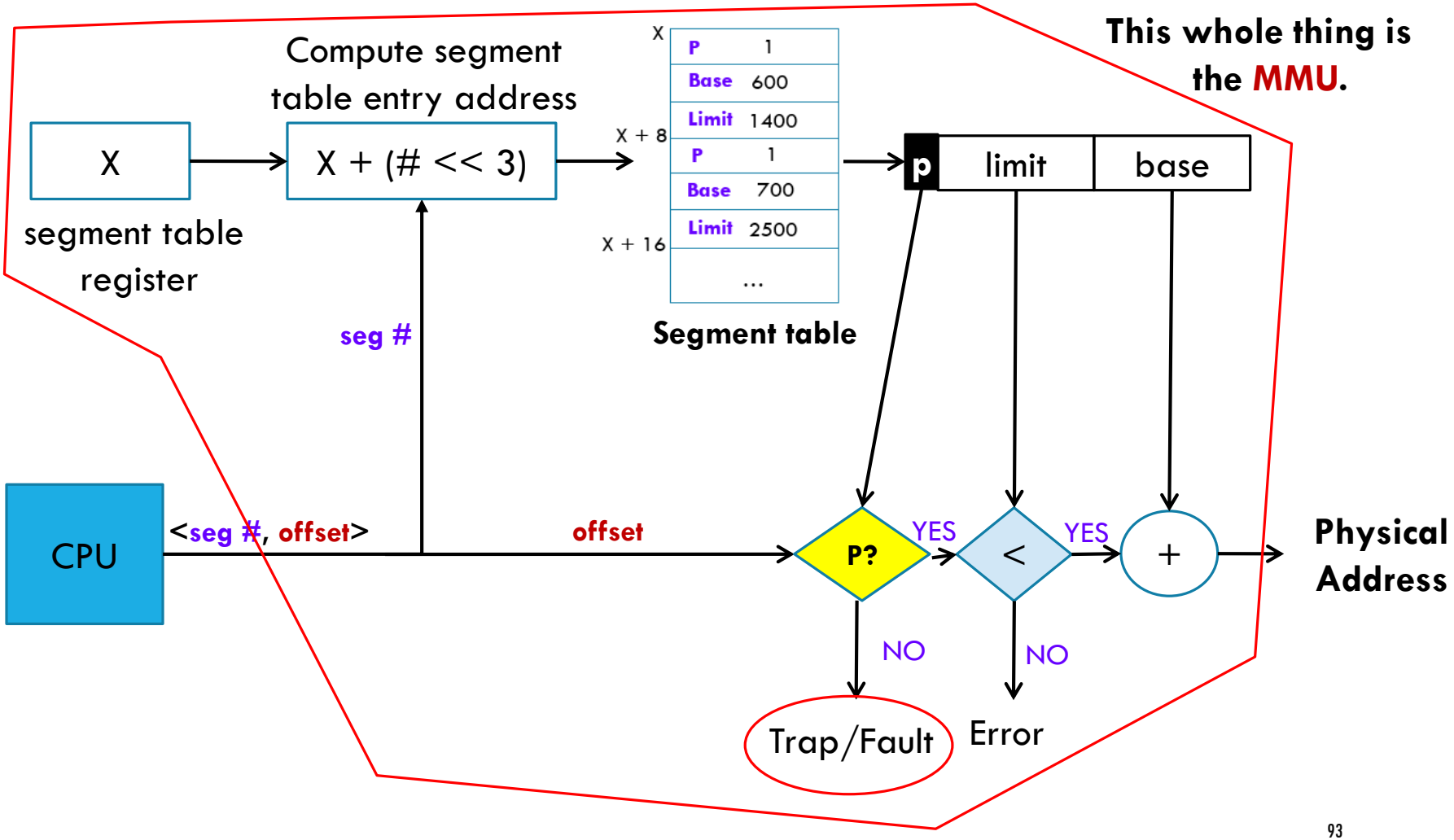
```
…
add blah, blah
store blah, blah
…
```

Problem:
What happens if this instruction is trying to access a non-present segment?

IMPROVED LOGICAL ADDRESS TRANSLATION



IMPROVED LOGICAL ADDRESS TRANSLATION

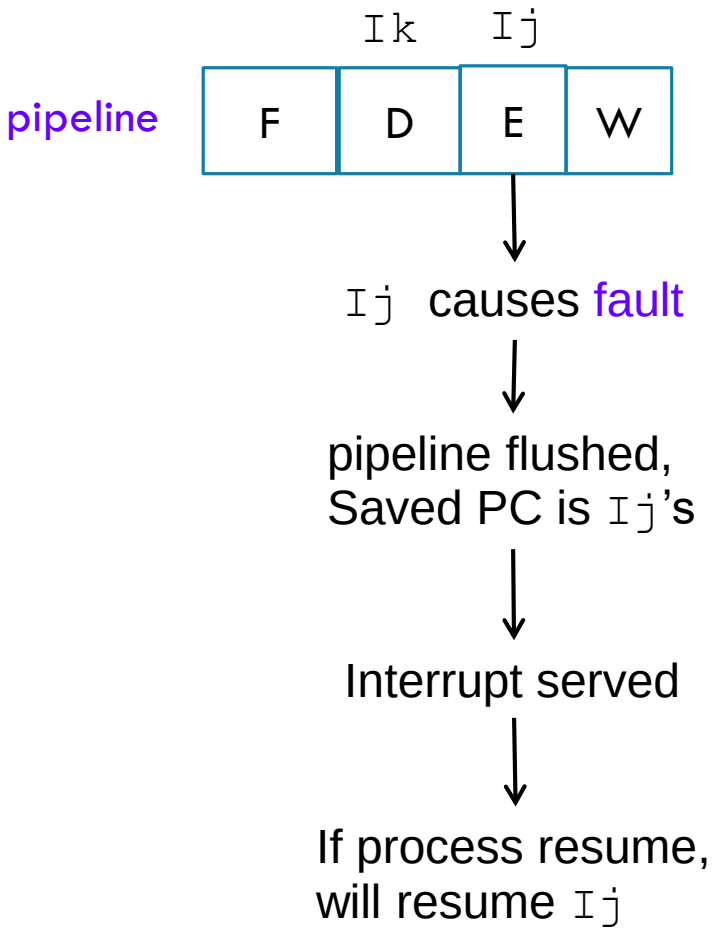


INTERRUPT HANDLING OVERVIEW

```
…
…
Ii: add blah, blah
Ij: store blah, blah
Ik: mul blah, blah
…
…
```

This line causes **fault** (interrupt)!

Recall that the interrupt handling will ensure the executing process will **restart from this instruction again.**



HANDLING MEMORY FAULTS

- What is a **fault**?
  - It's a kind of an interrupt.
- What code actually runs as a result of the interrupt?
  - A suitable interrupt service routine is called.
- What does an ISR serving memory fault do?
  - Find out which memory access causes the **segmentation fault**.
  - **If the segment is present in memory, 2 possibilities:**
    - Illegal access. (permissions problem). Terminate process
    - **Examples:** - Trying to write to a read-only memory region.
      - Attempting to execute a memory region that is not marked as executable.
      - Accessing a memory-mapped hardware register that only the operating system or privileged code can access.
    - Out-of-bounds access. Terminate process
    - **Examples:** - Accessing an element of an array beyond its declared size.
      - Dereferencing a pointer that points outside the allocated memory.
      - Buffer overflows, where data is written beyond the end of a buffer.
  - **If the segment is not present in memory, 2 possibilities:**
    - Segment was never allocated. Terminate process.
    - Segment is in secondary storage (e.g. HD/SSD). OS brings segment into memory. Updates segment table entry, process resumes.

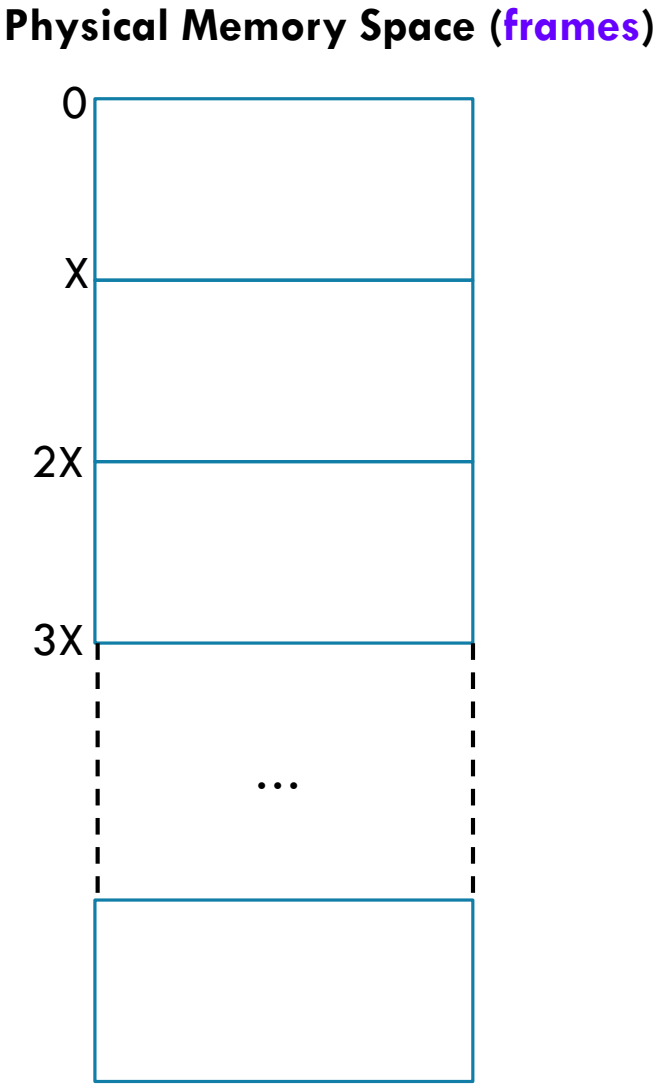
PAGING: NON-CONTIGUOUS MEMORY ALLOCATION

The basic idea of **paging** is to break the **physical memory** into **fixed-sized blocks** of size X called *frames* and to break **logical memory** into blocks of the same size X called *pages*.

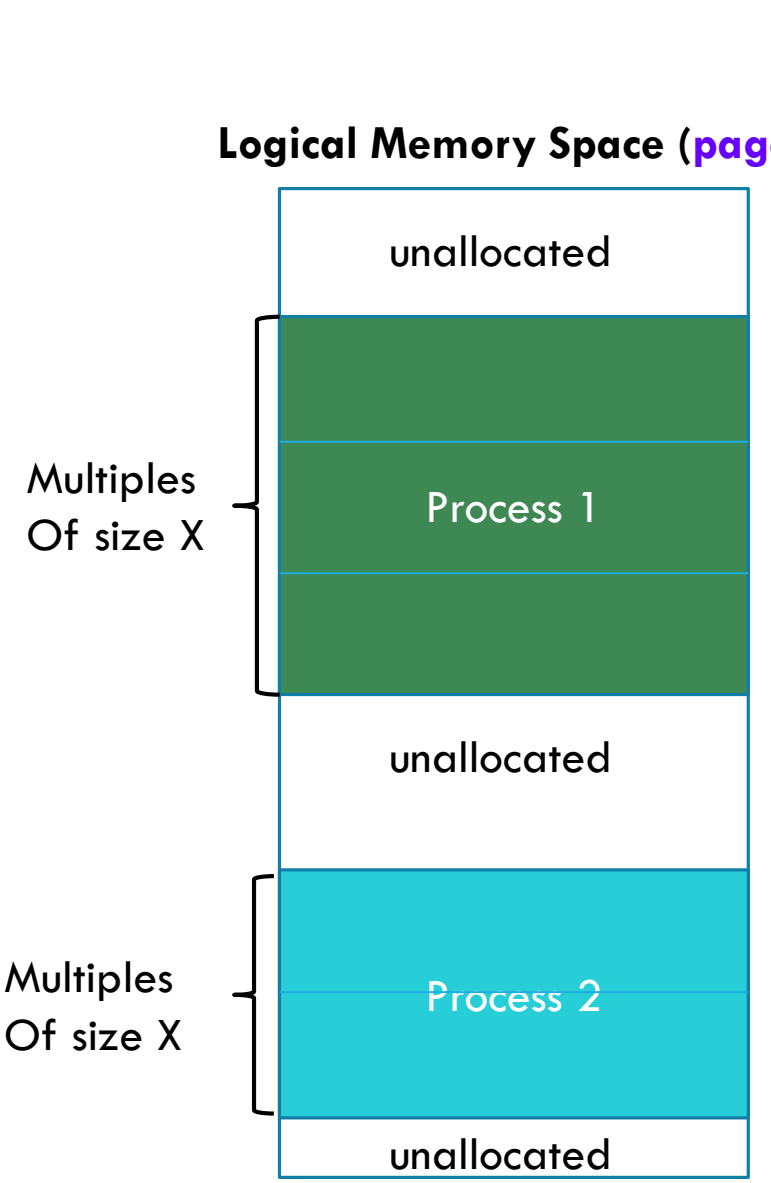
PAGING: NON-CONTIGUOUS MEMORY ALLOCATION

Divide physical memory into **"frames"**.

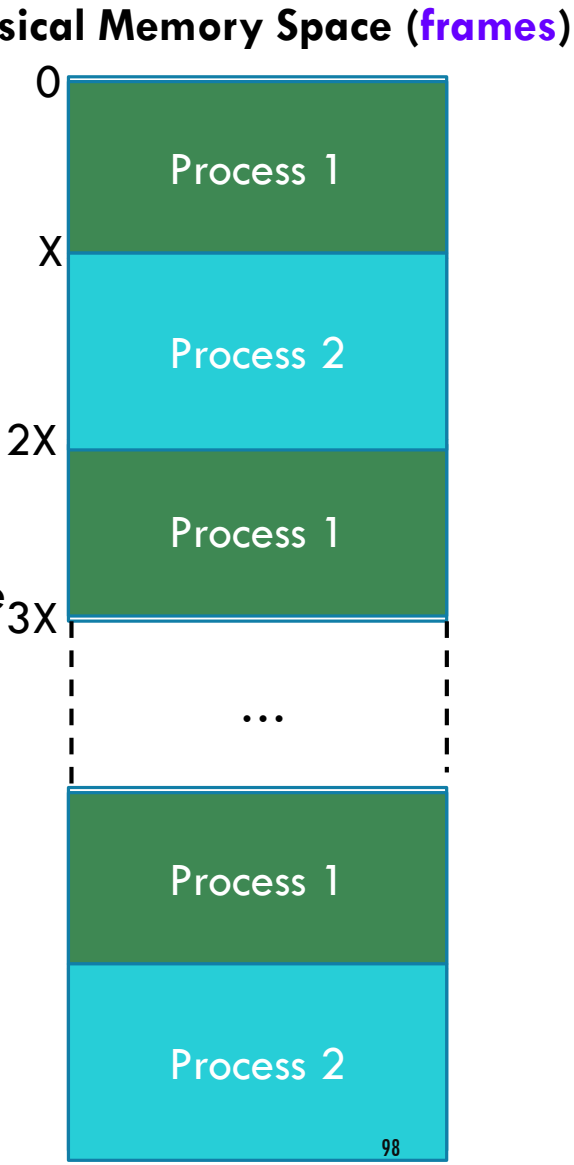
Each frame has the **same fixed size.**



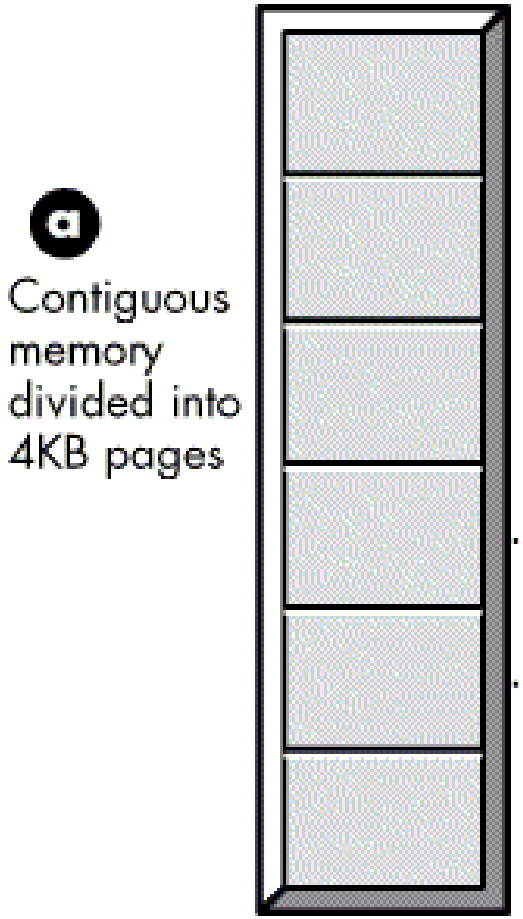
PAGING OVERVIEW



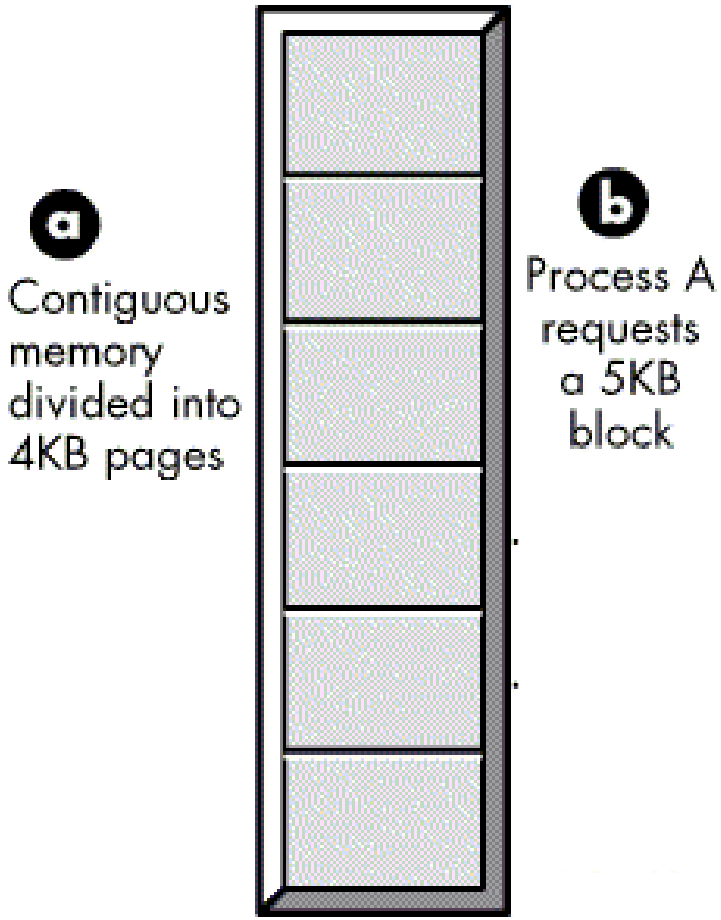
1. What appears to be contiguous to the process may not be in physical memory.
2. Process logical memory space is always allocated as a **multiple of size X.**
3. X is the frame size.
4. Paging may lead to **internal fragmentation.**



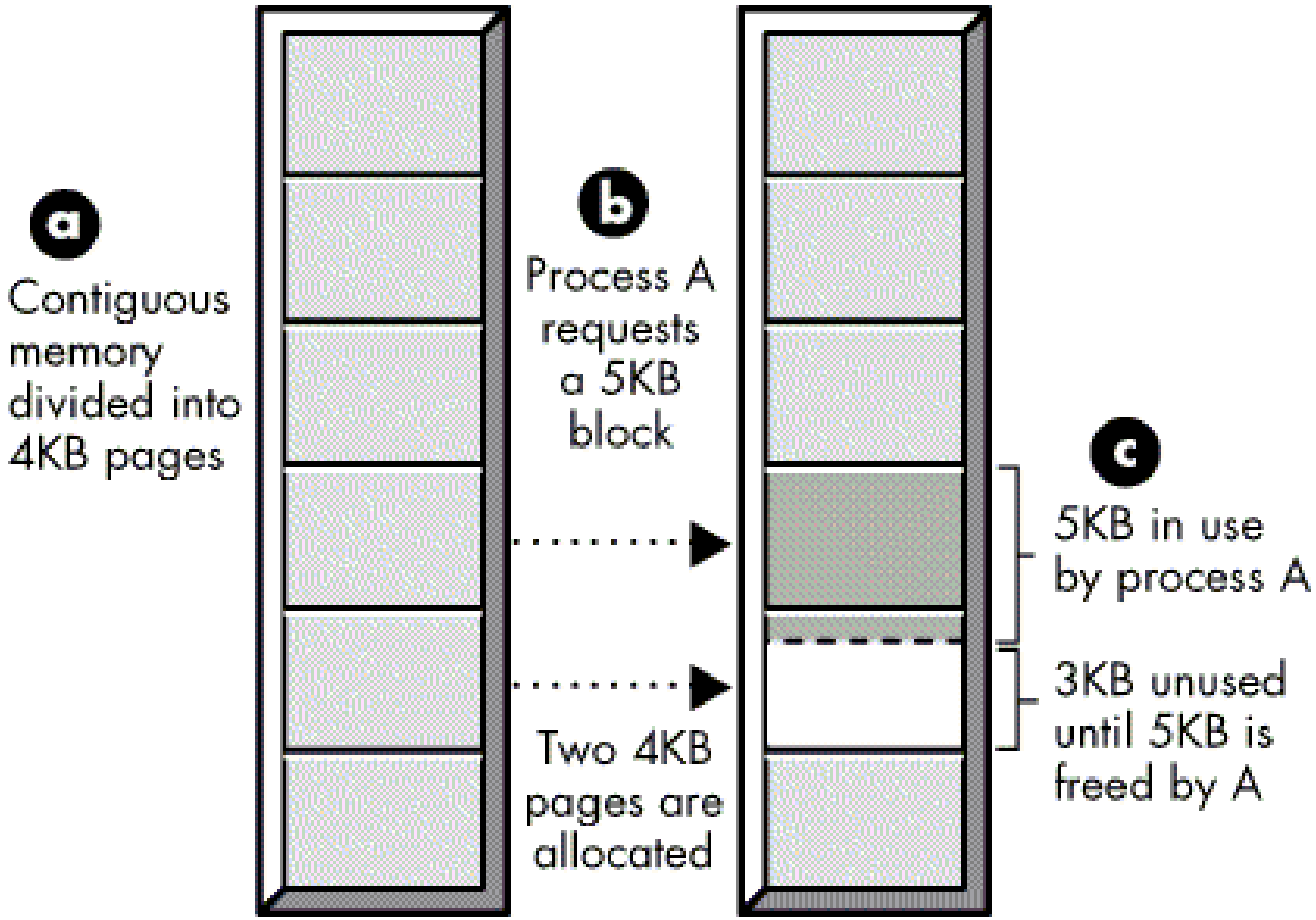
PAGING - INTERNAL FRAGMENTATION



PAGING - INTERNAL FRAGMENTATION



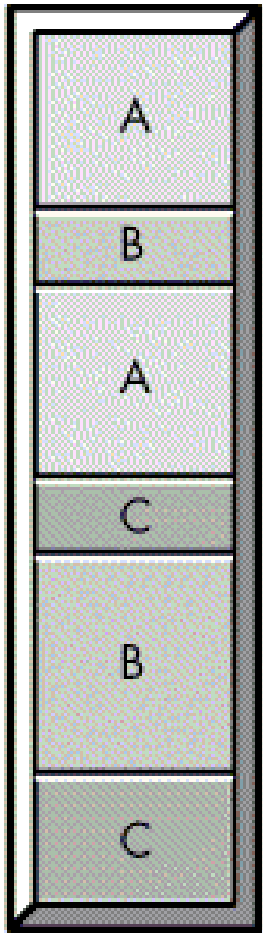
PAGING - INTERNAL FRAGMENTATION



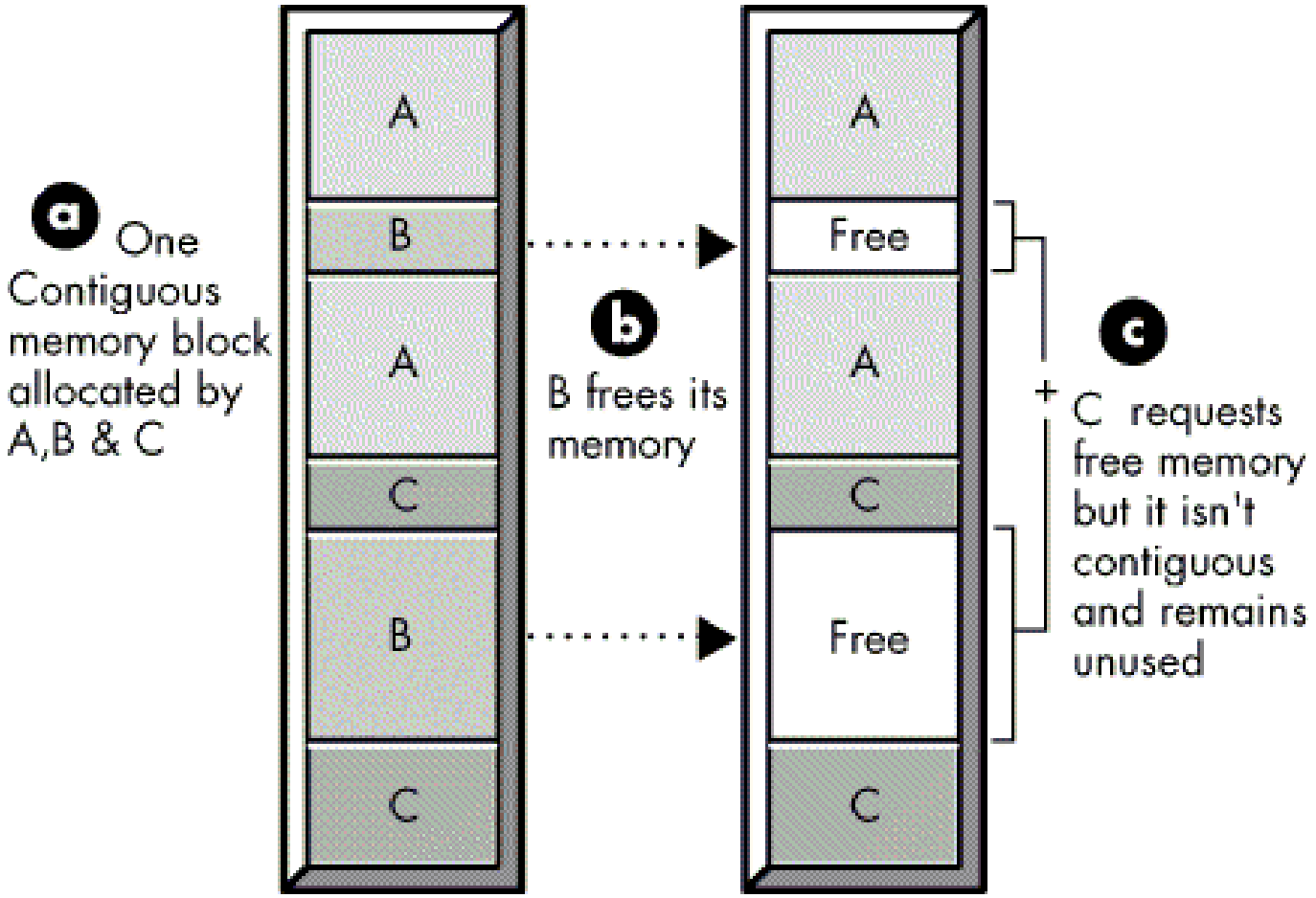
Internal fragmentation occurs when memory is divided into fixed-sized partitions

SEGMENTATION - EXTERNAL FRAGMENTATION

(a) One Contiguous memory block allocated by A,B & C



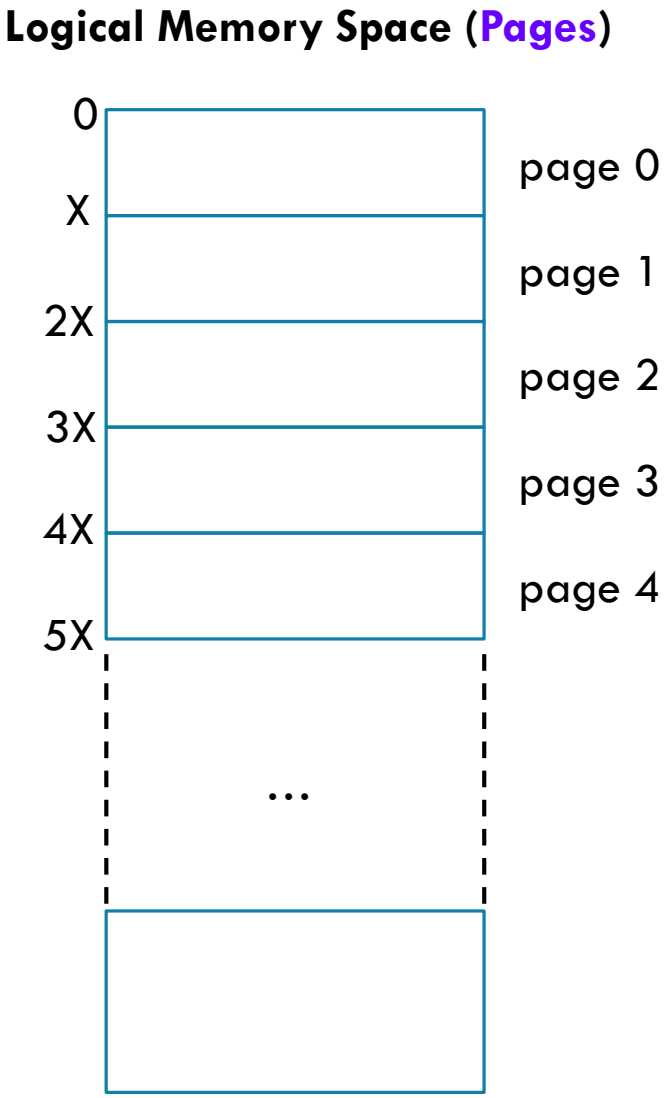
SEGMENTATION - EXTERNAL FRAGMENTATION



External fragmentation occurs when memory is divided into variable size partitions based on the size of processes

DEMO SHOWING **MEMORY ALLOCATION** FOR **PAGING**

LOGICAL ADDRESS SPACE



- ***Logical address space*** is organized in terms of *pages*.
- **X** is usually powers of 2 which simplifies memory addressing

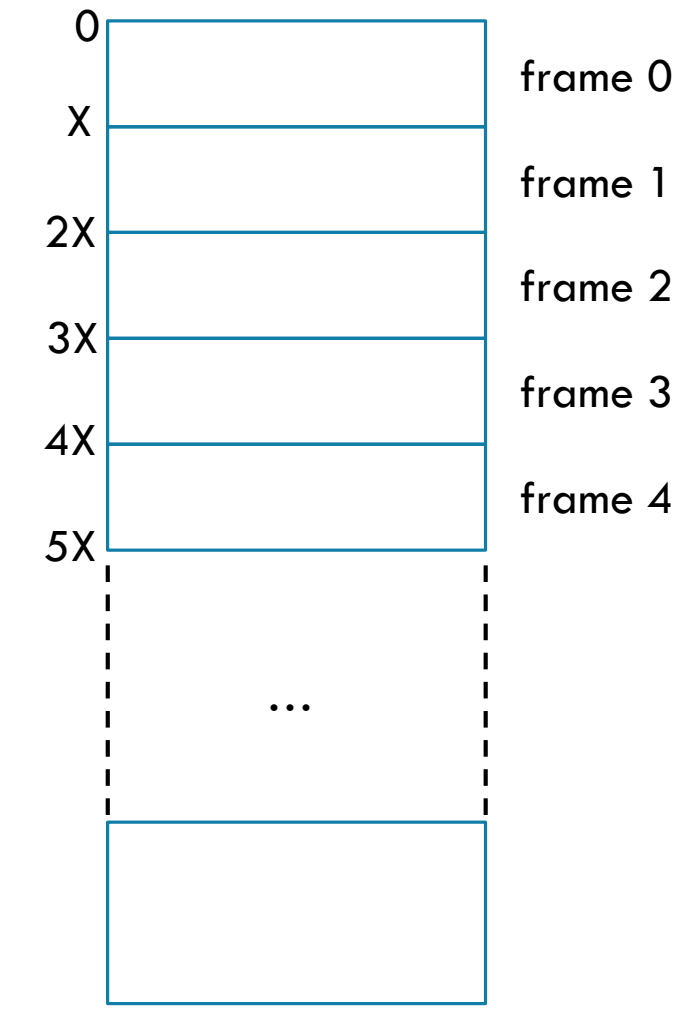
PAGING LOGICAL ADDRESS

The **logical address** is just a memory address

So.. What does offset = 0 mean?

- 1st address of a **Page**.

PHYSICAL ADDRESS SPACE



- **Physical address** space is organized in terms of *frames*.

PHYSICAL ADDRESS

The **physical address** is just a memory address

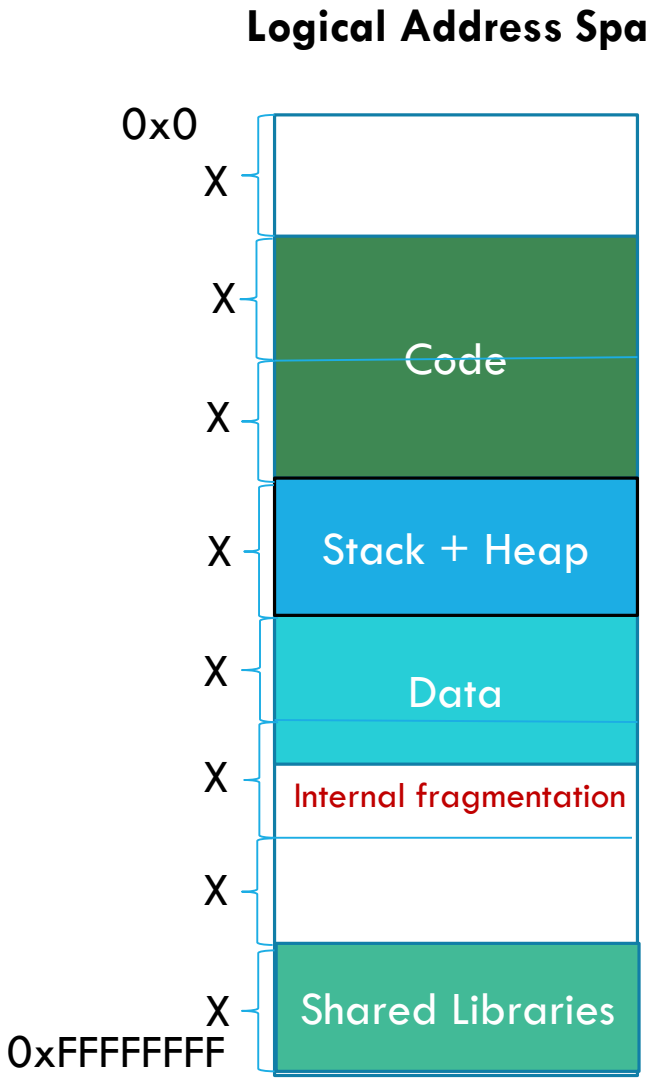
So.. What does offset = 0 mean?

- 1st address of a **Frame**.

WHAT? THE PREVIOUS FEW SLIDES LOOK SO SIMILAR!

1. What's the diff between pages and frames?
   a) Frames are going to contain pages.
   b) Pages can move but frames can't.
      - Pages in *virtual memory* can be moved around or swapped in and out of physical memory as needed.
      - If a page from a process is *mapped* to a particular frame (physical memory location), that page cannot be moved to another frame within physical memory. The frame's location in physical memory remains static.
2. Any other differences?
   a) Number of frames limited by amount of physical memory.
   b) Number of pages … limited by the number of address bits for logical address.

IMPLICATION? FREEDOM!



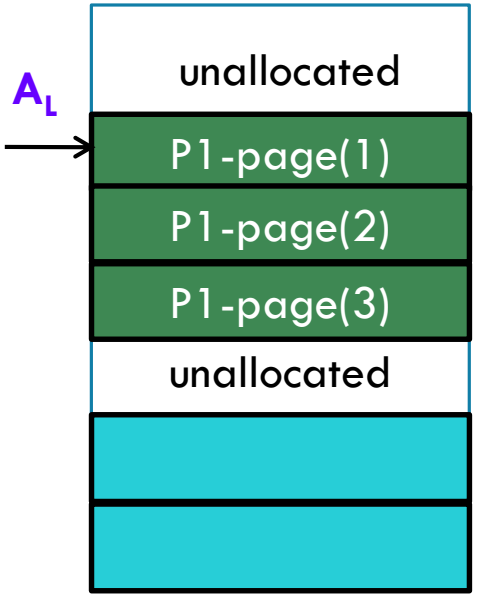
1. **Code** and **Data**'s logical address generated in compile/link time.
   - ❑ Freedom for compiler and linker to set the logical address of these regions.

2. **Stack + Heap** and **Shared Libraries** are allocated by loader/OS.
   - ❑ Freedom for OS to allocate logical addresses for these regions.

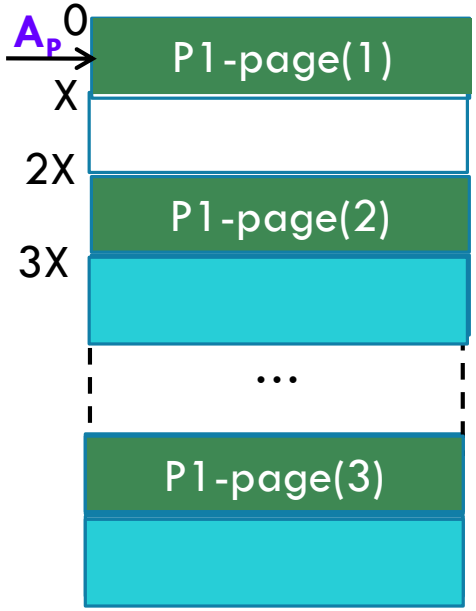
X is the frame size

TRANSLATING LOGICAL ADDRESS INTO PHYSICAL ADDRESS

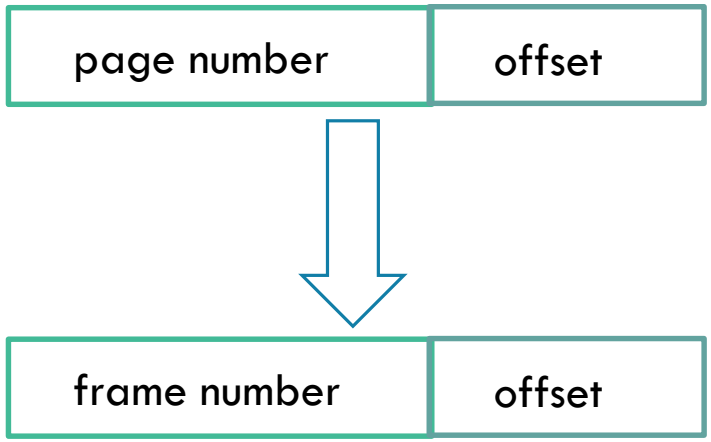
Logical Address Space
(what a process sees)



Physical Memory Space
(in reality)



Translating A_L into A_P



It is really about finding the matching frame number for a page number.

- How to match them?
  - ❑ **Page Table** !!!

A_L → logical address

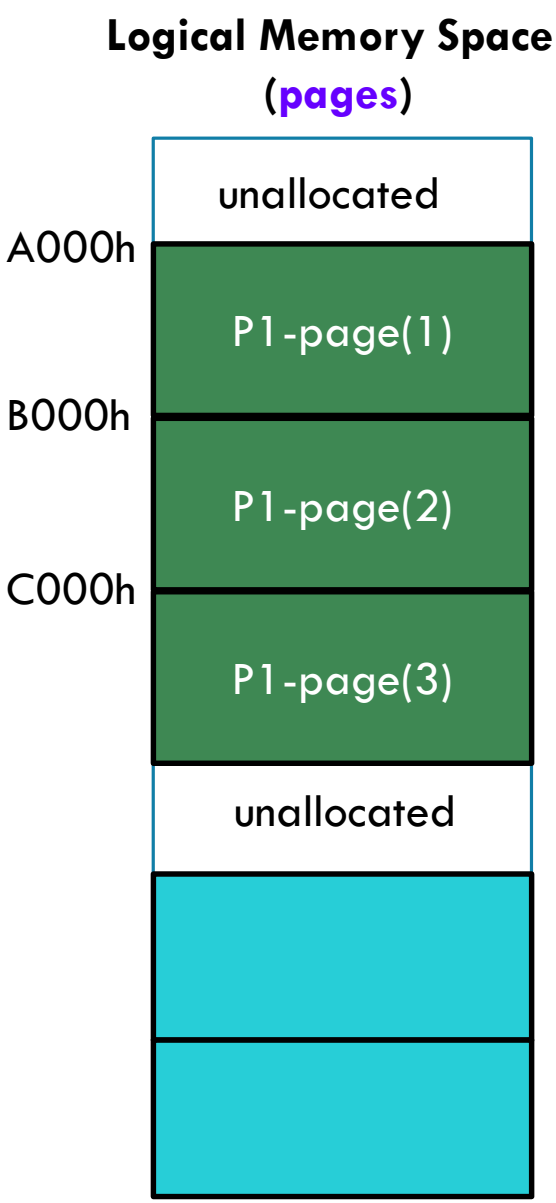
A_p → physical address

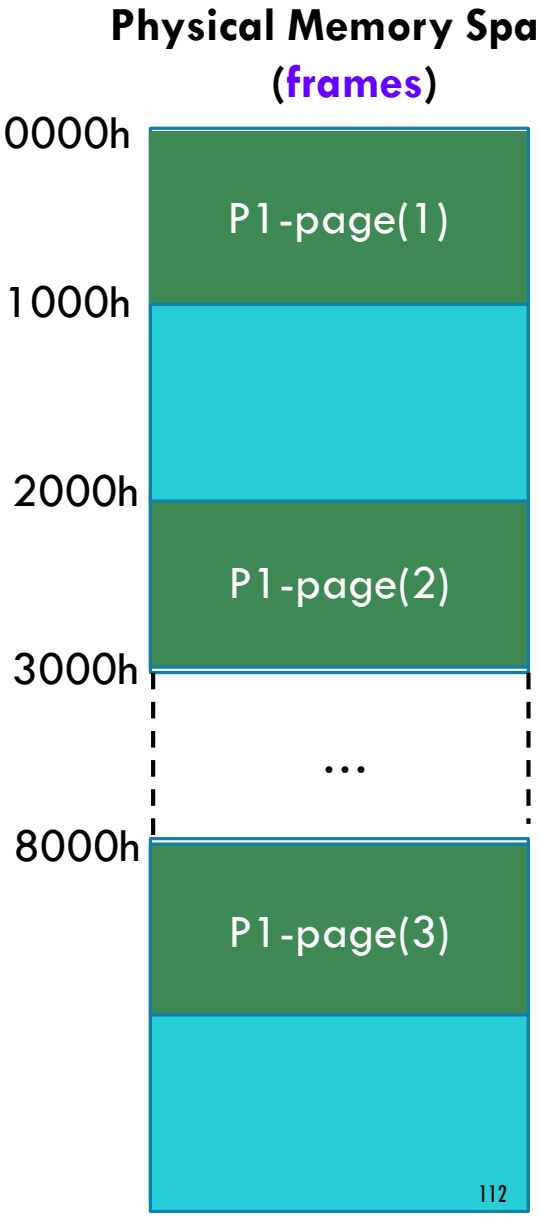
PAGE TABLE

Page table

	Frame Number
…	
page(1)	0
page(2)	2
page(3)	8
…	

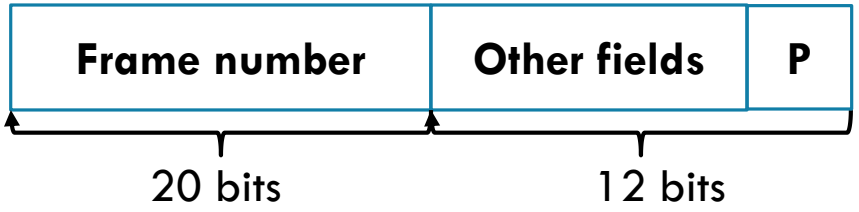
- **Per-process page table**
- **Page-to-Frame mapping**
- **Usually Page Size = 4K**

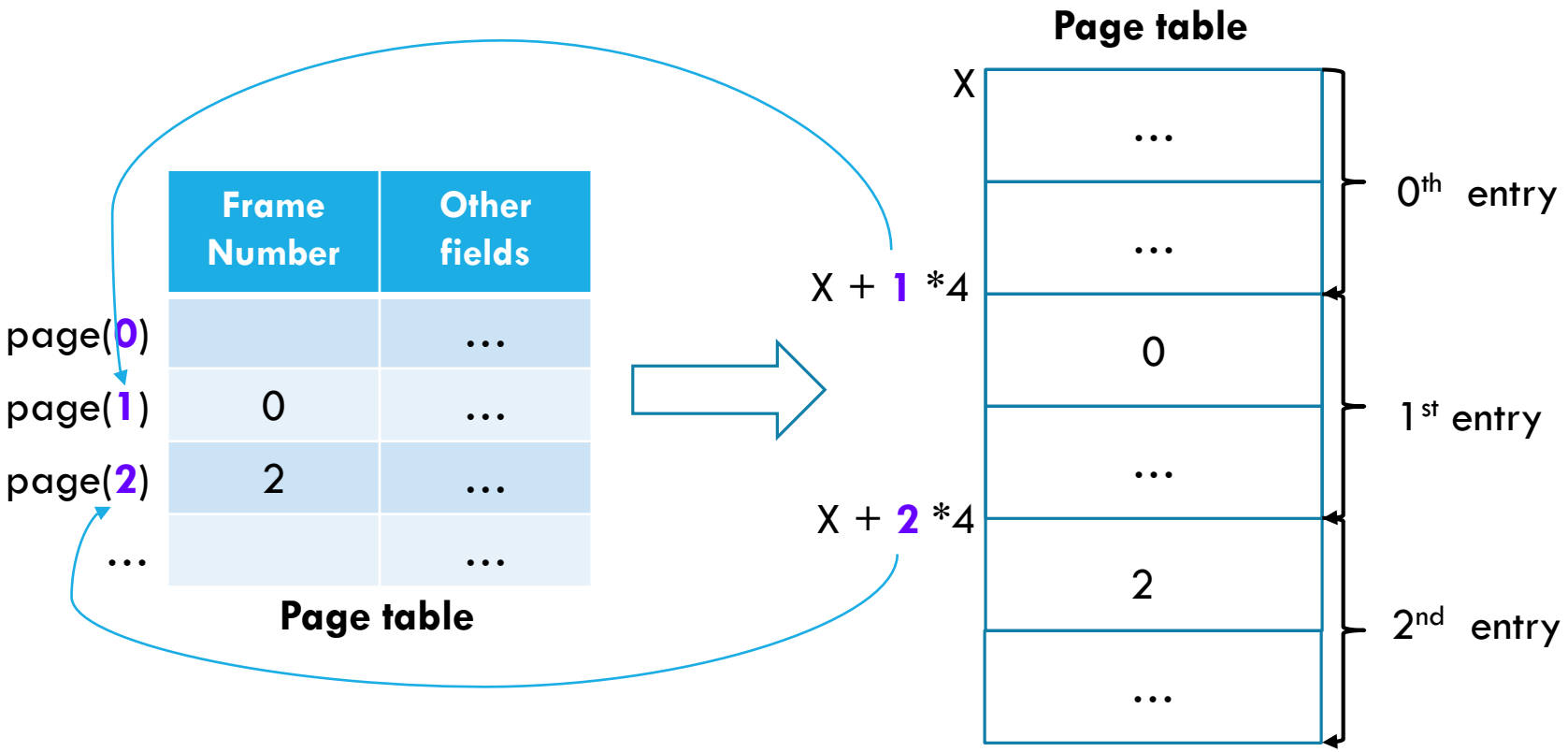




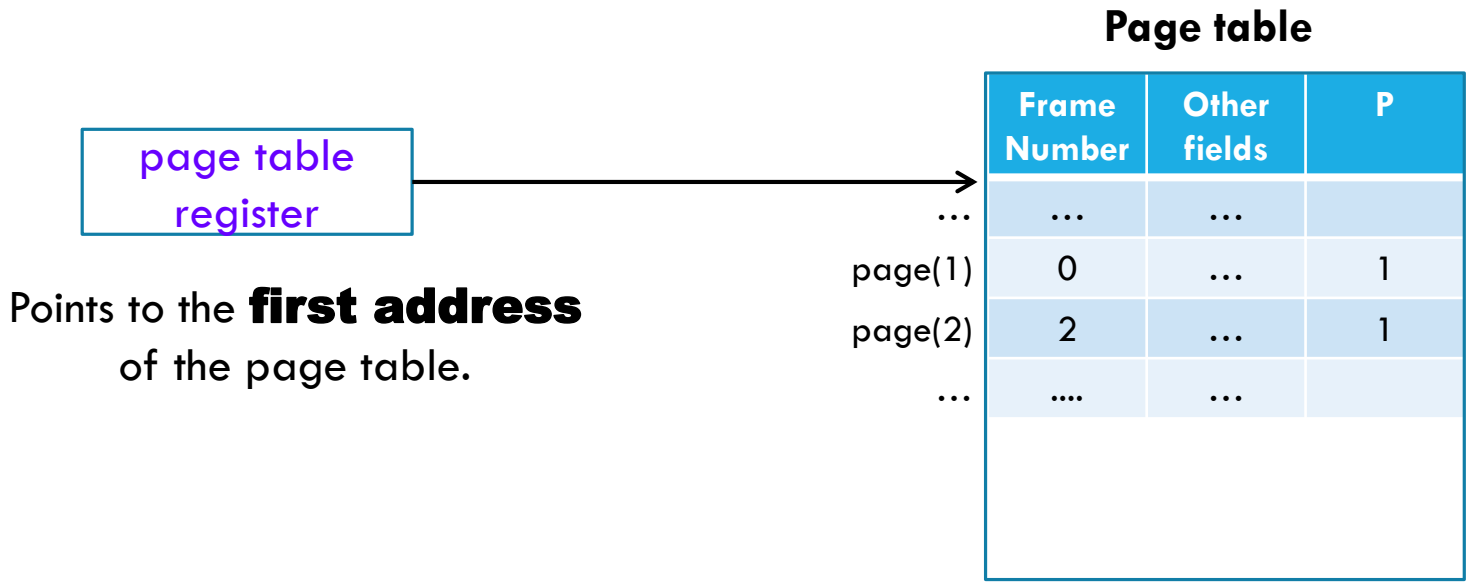
PAGE TABLE ENTRY

A realistic page table entry contains more than just a frame number



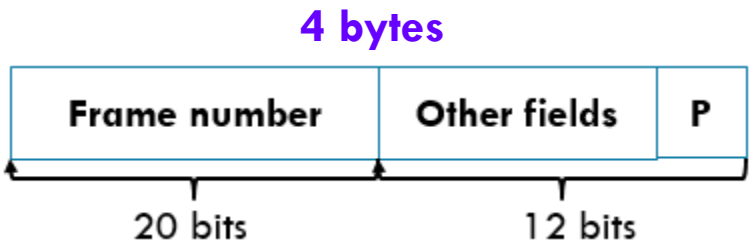


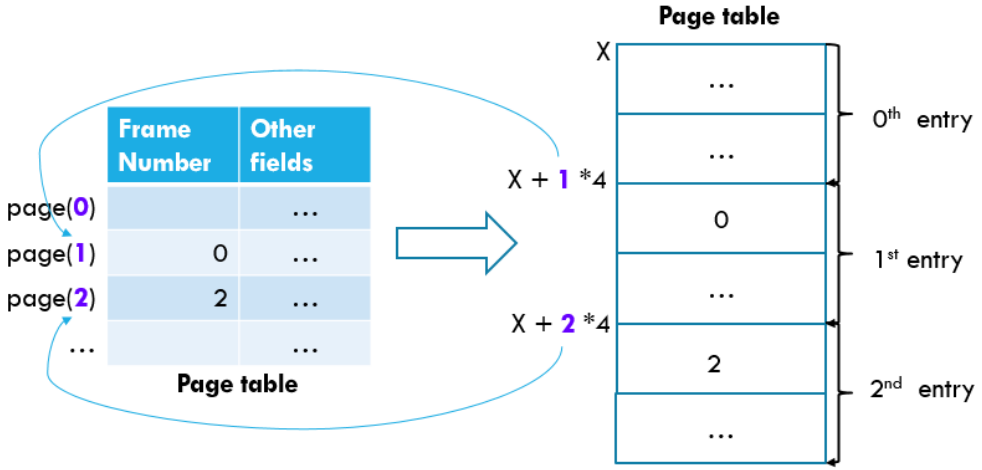
PAGE TABLE REGISTER



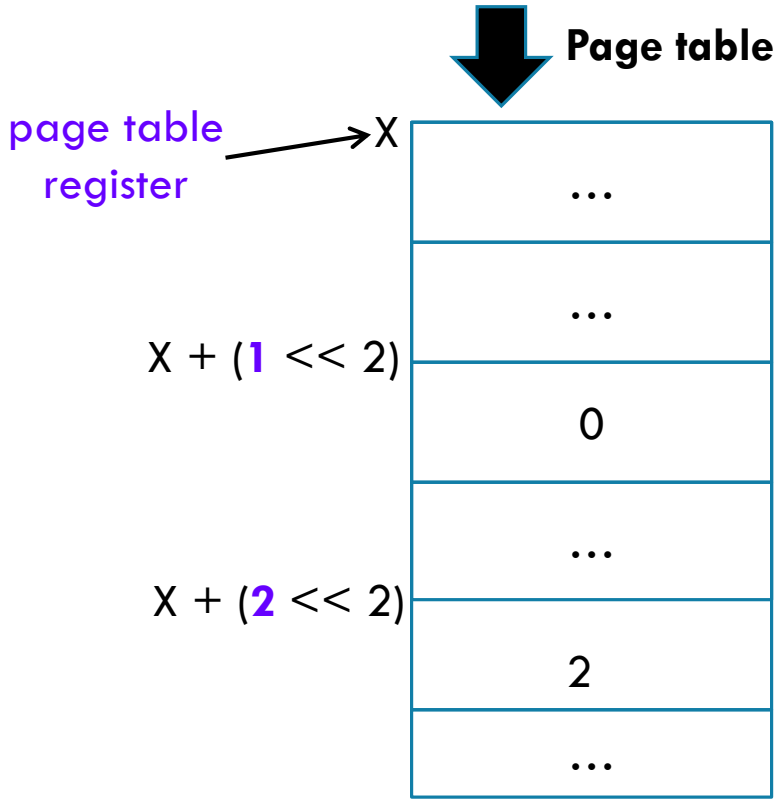
Points to the **first address** of the page table.

USING PAGE TABLE REGISTER + PAGE NO.

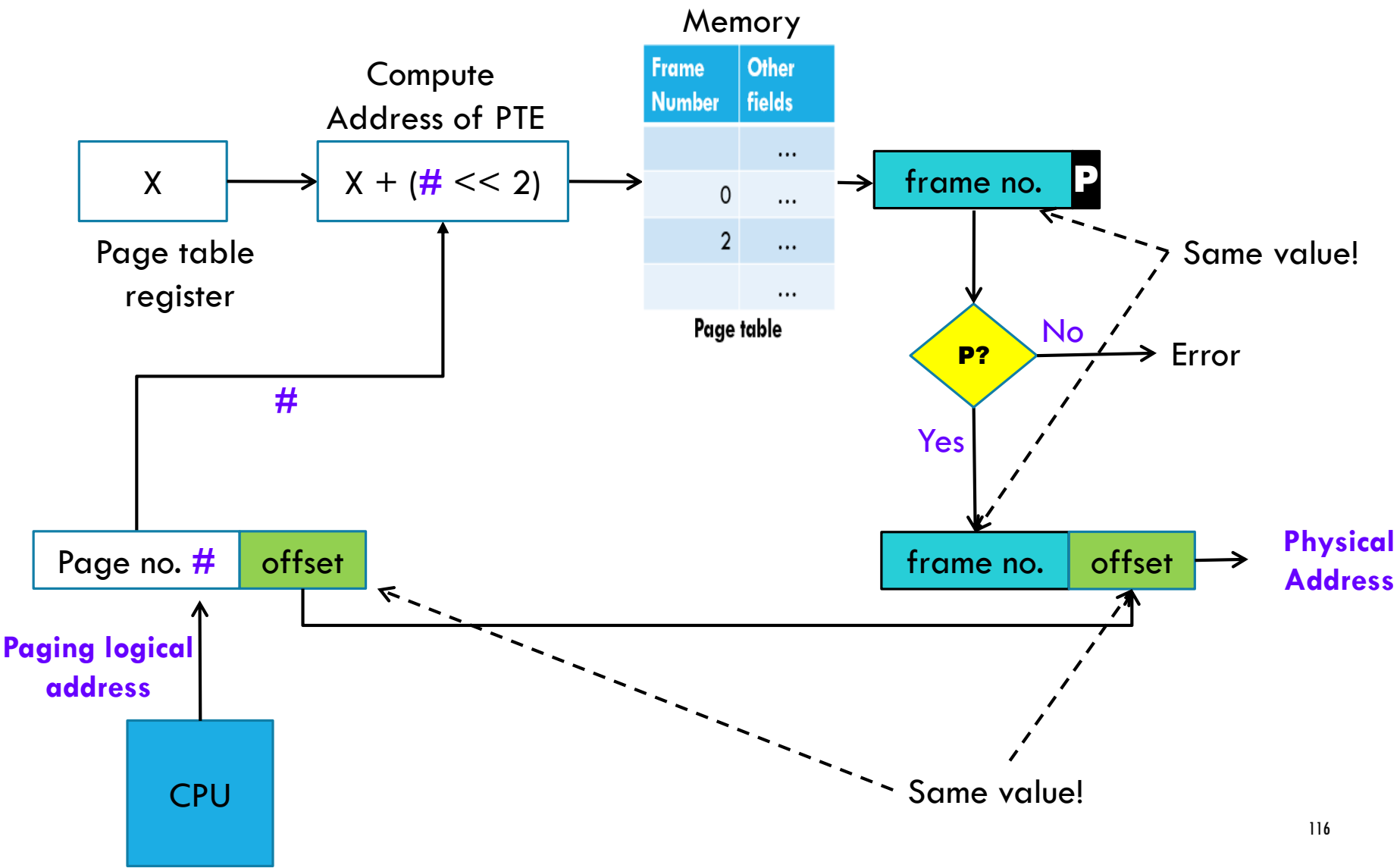




Page table register	X
Page No.	#
Address of Page table entry matching #	X + (# << 2)



PAGING LOGICAL ADDRESS TRANSLATION



PAGE TABLE SIZES

- Page size = 2^N bytes
- Number of page table entries = $2^{(\text{Address bits} - N)}$
- Page table size = Number of page table entries x sizeof(1 page table entry)

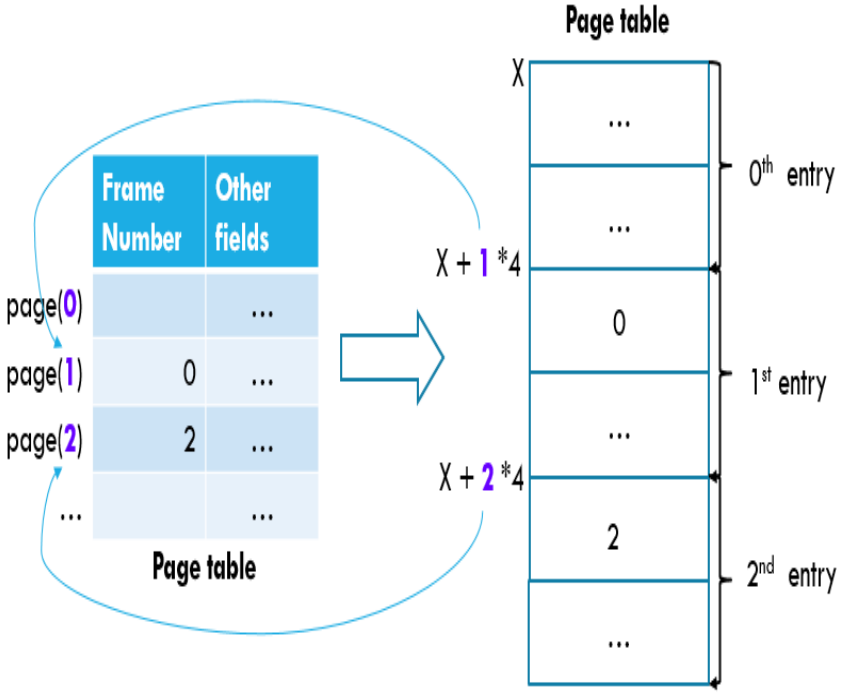
Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

Each process needs a page table size of **4MB** in physical memory!!! Can we do better?

REDUCE SIZE OF PAGE TABLES!

- **Problem:** Allocating page table contiguously in main memory for each process.
- **Solution:** Divide the page table into smaller pieces.
- **Idea:** Do so by employing 2-level paging. *Take a page table of the page table*.



Each process needs a page table size of **4MB** in physical memory!!!

2-LEVEL PAGING BASIC IDEA

Original Page Table (4MB)

Need **4MB** in Main Memory per process

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

Each process needs a page table size of **4MB** in physical memory!!! Can we do better?

We can break the Original Page Table down into multiple "smaller pages" !!!.

2-LEVEL PAGING BASIC IDEA

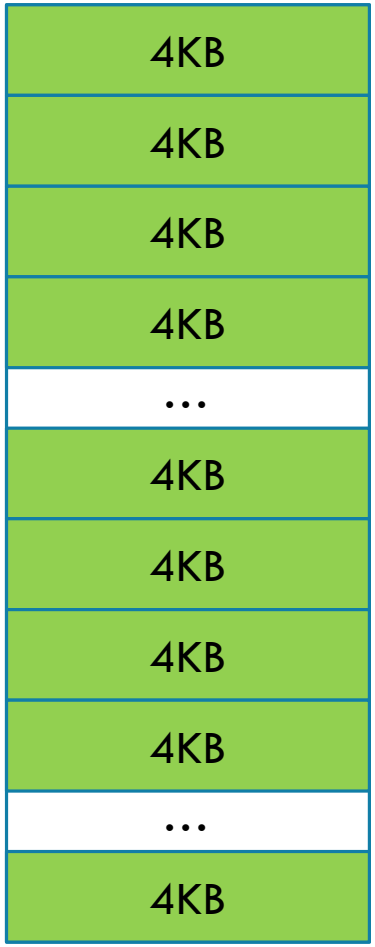
Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20}**x 4 bytes = 4MB**

Each process needs a page table size of 4MB in physical memory!!! Can we do better?

We can break the Original Page Table down into multiple "smaller pages", each of 4KB size !!!.

Original Page Table (4MB)



Let's think about how each "smaller page" (**size of 4KB**) of original page table entries now matches the logical address space.

- **4MB** consists of **1K** of **4KB** pages i.e., 4 MB consists of 1K pages.

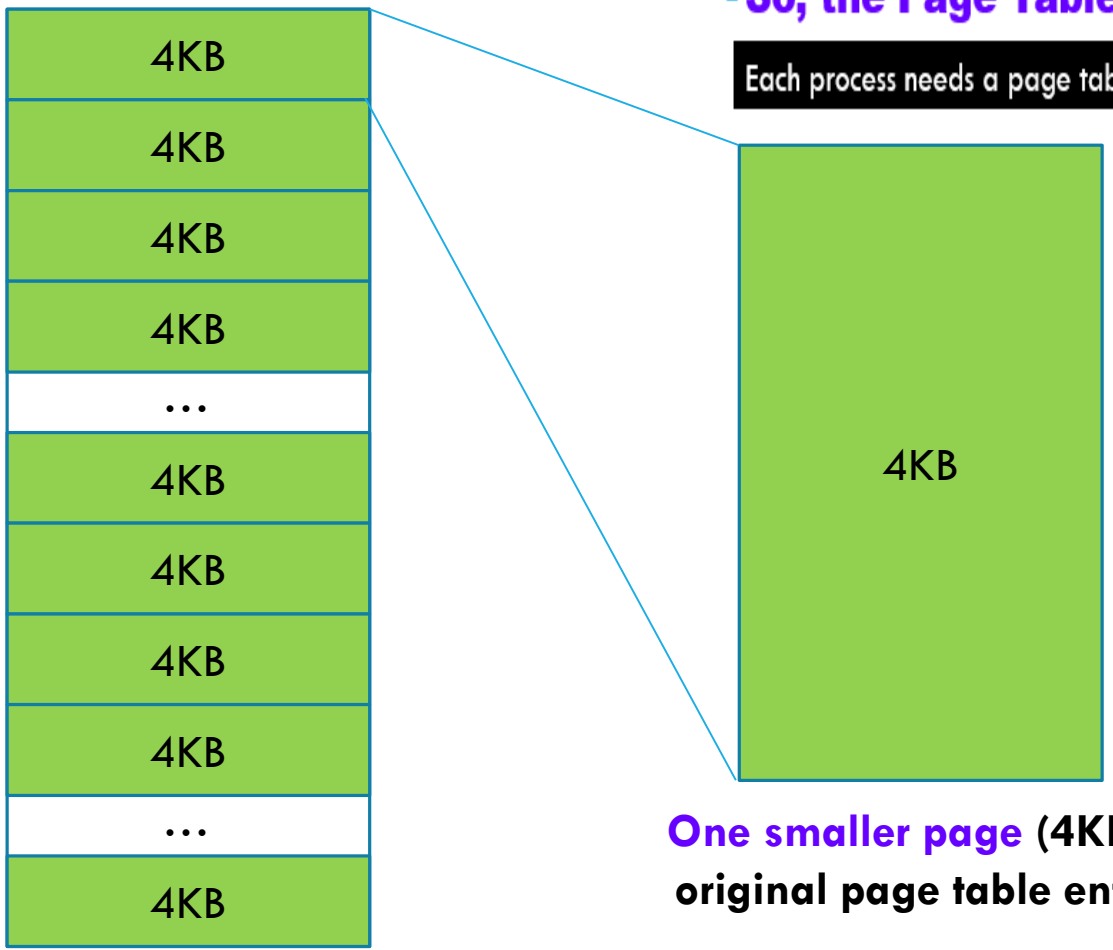
2-LEVEL PAGING BASIC IDEA

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB = 2^{12} bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size = 2^{20}x 4 bytes = 4MB**

Each process needs a page table size of **4MB** in physical memory!!! Can we do better?

Original Page Table (4MB)



One smaller page (4KB) of original page table entries

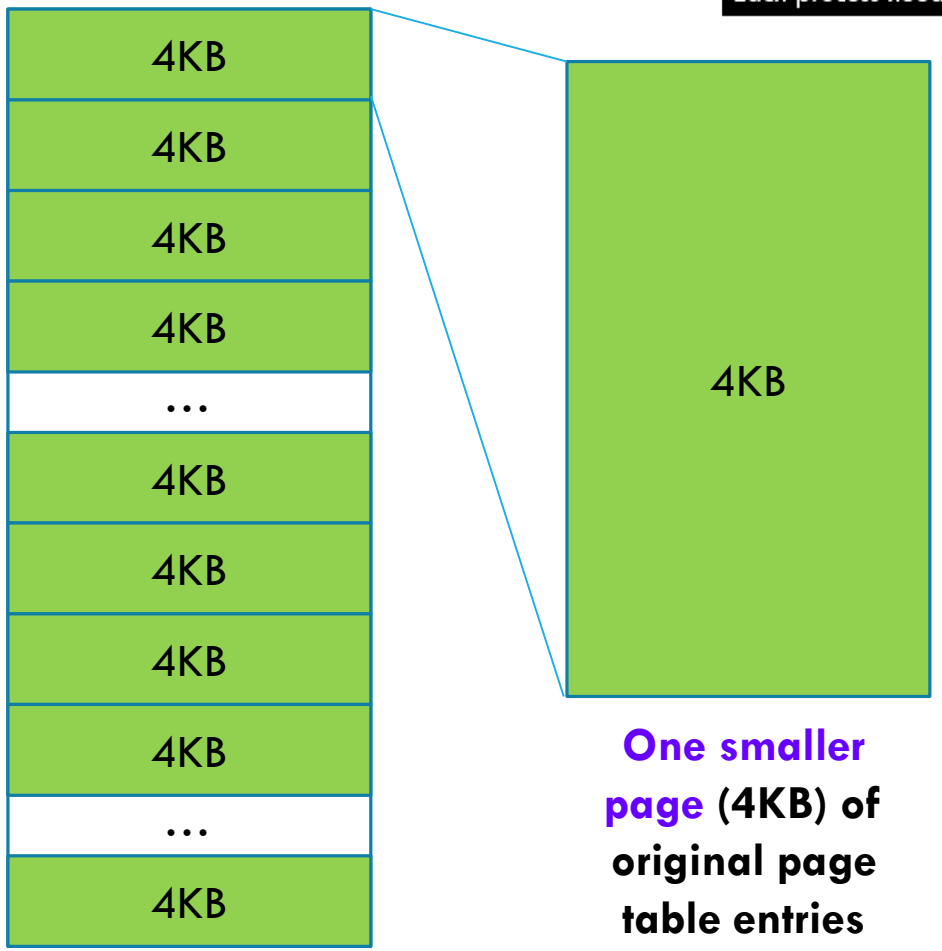
2-LEVEL PAGING BASIC IDEA

Example:

- Let the number of Address bits be **32** (virtual address bits)
- Let the Page size be **4KB** = 2^{12} **bytes.**
- Then, number of Page Table Entries (PTE) = $2^{(\text{Address bits} - N)} = 2^{32-12} = 2^{20}$
- Let one page table entry size = **4 bytes**
- **So, the Page Table Size** = 2^{20} **x 4 bytes = 4MB**

Each process needs a page table size of **4MB** in physical memory!!! Can we do better?

Original Page Table (4MB)



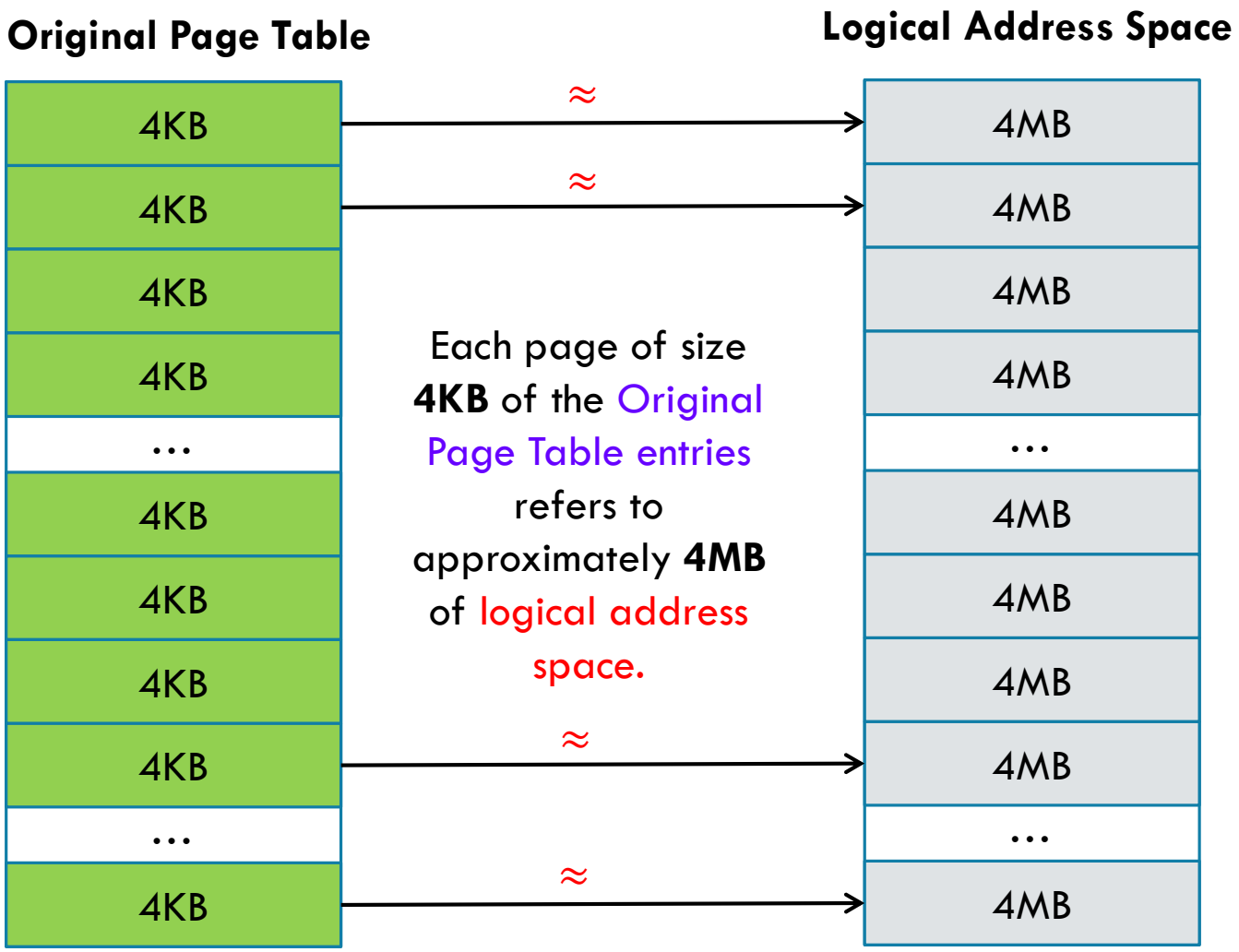
One smaller page (4KB) of original page table entries

One smaller page 4KB of original page table entries

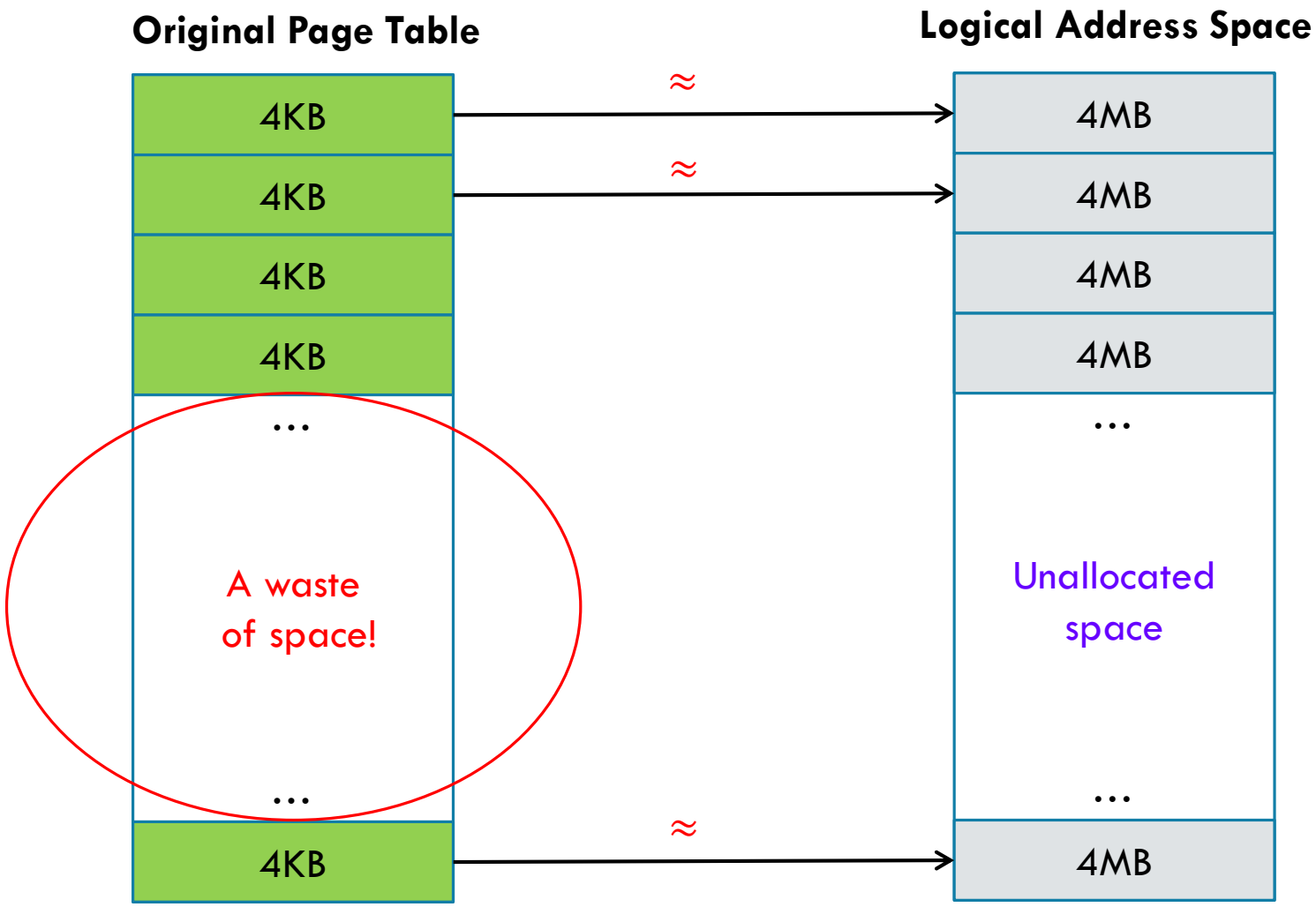
- **One smaller page (4KB)** of Original Page Table entries is 2^{10} = **1024,** 32-bit Page Table Entries **(PTE).**
- Recall that each Original Page Table Entry refers to a page of size **4KB.**

So **1024 Original Page Table entries** *each referring to a page size* of **4KB** refers to **4MB** of logical memory !!!

RELATIONSHIP BETWEEN THE PAGE TABLE AND THE LOGICAL ADDRESS SPACE

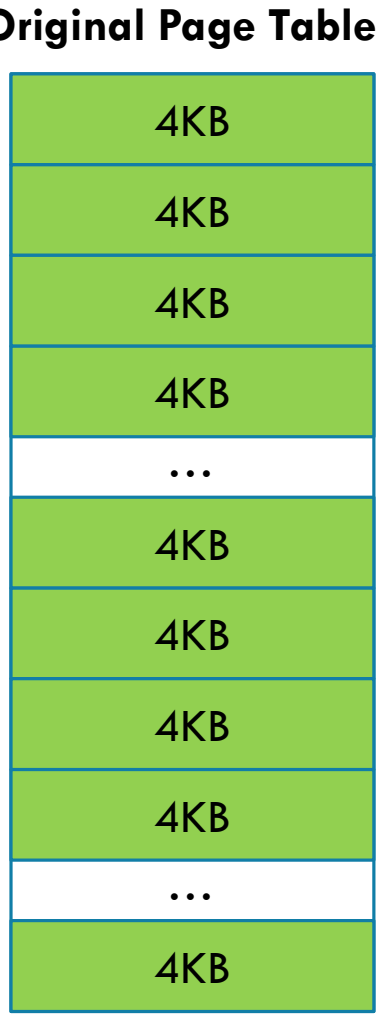


PROBLEM: NOT ALL LOGICAL ADDRESS SPACES ARE USED/ALLOCATED!



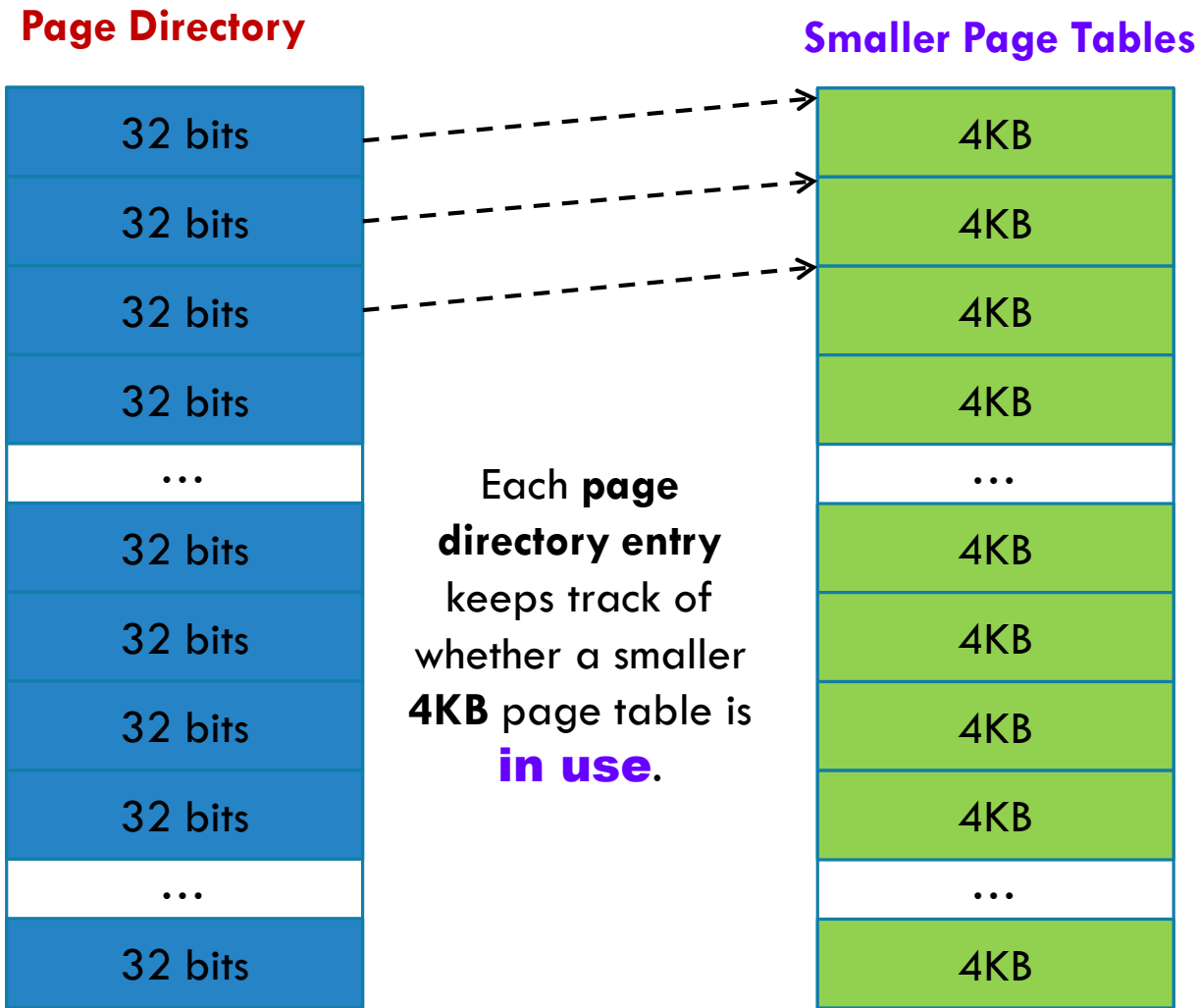
SOLUTION: 2–LEVEL PAGING

1. Break the **Original Page Table** into smaller page tables of **4KB** size each.
2. Original Page Table size = **4MB** = **1K** smaller page tables of **4KB** each.

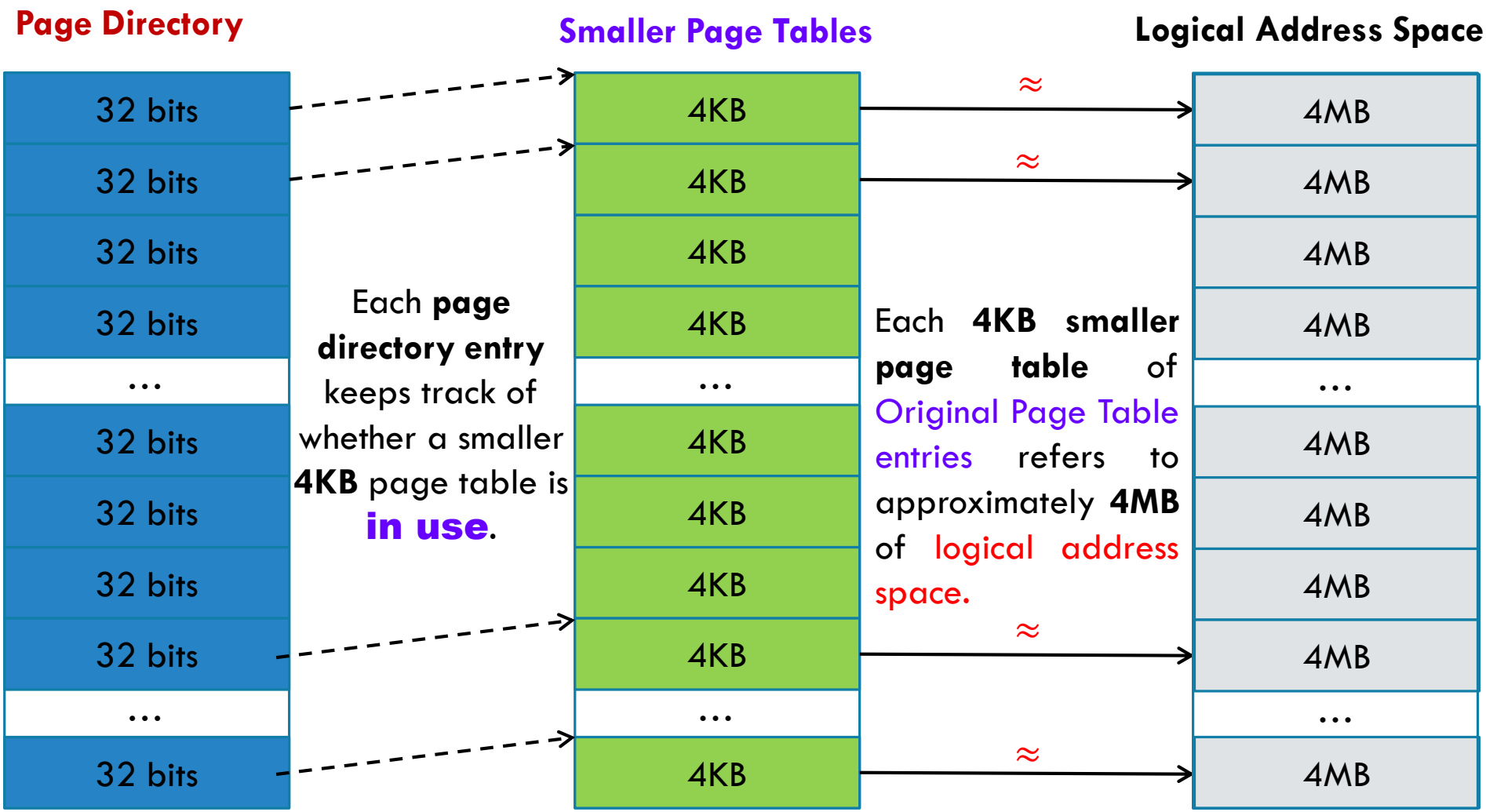


SOLUTION: 2–LEVEL PAGING (VIRTUALIZE THE PAGE TABLE)

1. Break the **Original Page Table** into smaller page tables of **4KB** size each.
2. Original Page Table size = **4MB** = **1K** smaller page tables of **4KB** each.
3. Use a **page directory** of size **1K** to keep track of the **starting address** of each smaller page table.

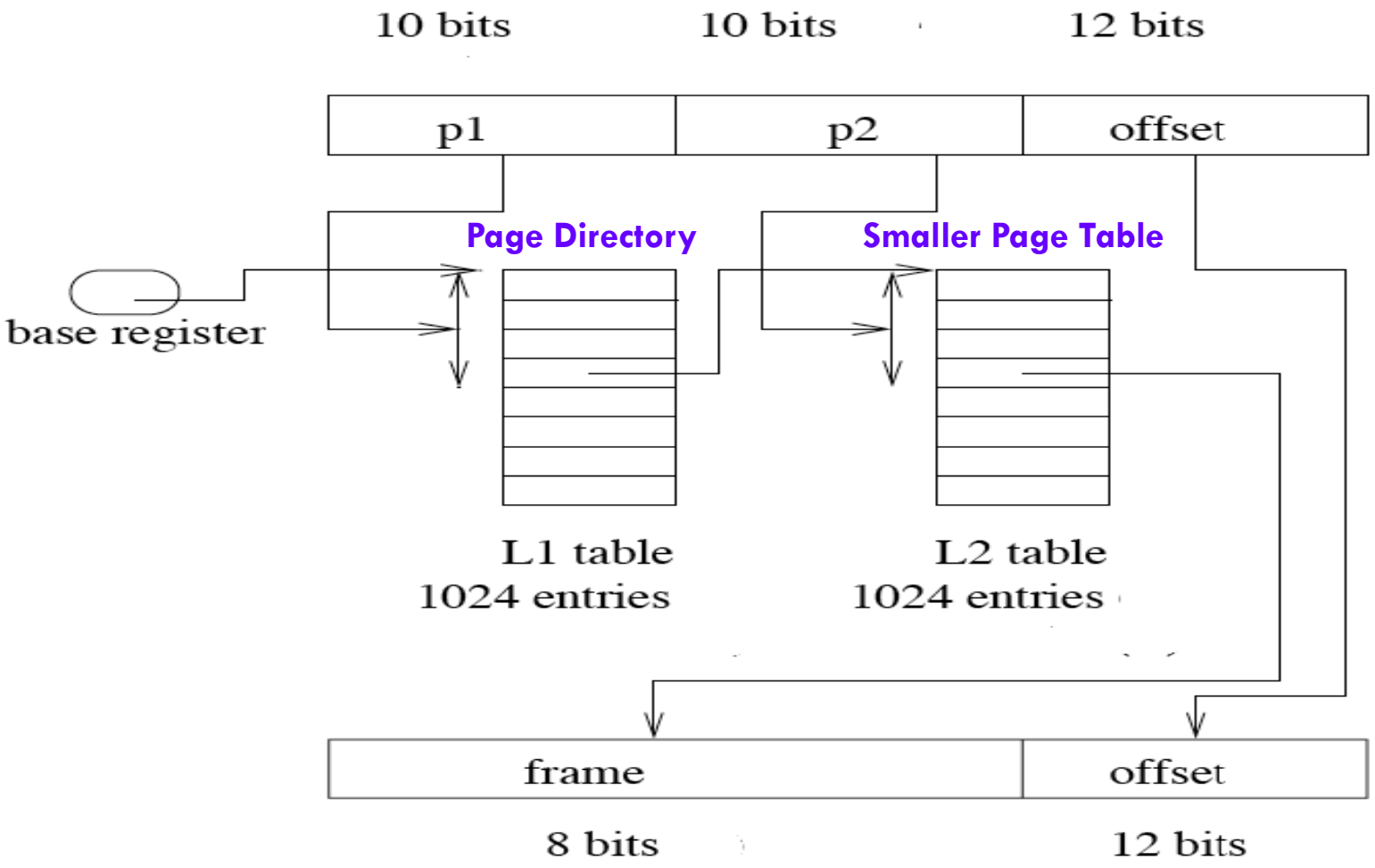


2-LEVEL PAGING: OVERVIEW



2-LEVEL PAGING VIEW: EXAMPLE

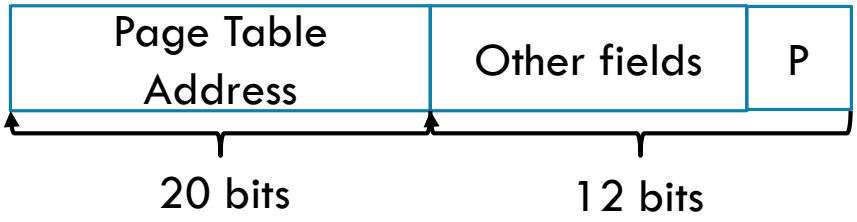
* Page directory → **outer page table**
* Smaller Page Table →**inner page table**



PAGE DIRECTORY ENTRY

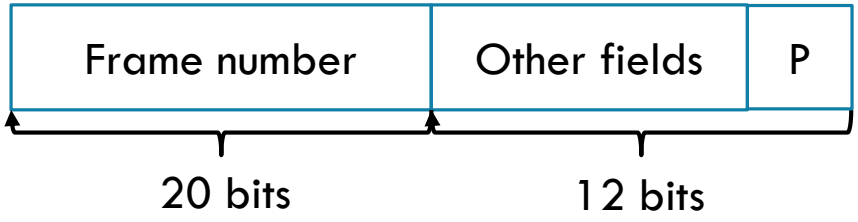
Page Directory Entry looks the same as a **page table entry**. Instead of the frame number, we have the starting address of the page table.

Page Directory Entry



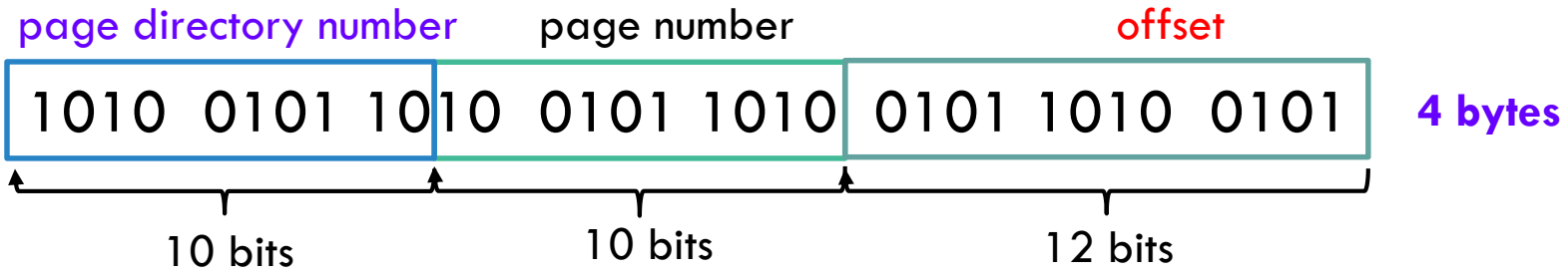
4 bytes

Page Table Entry

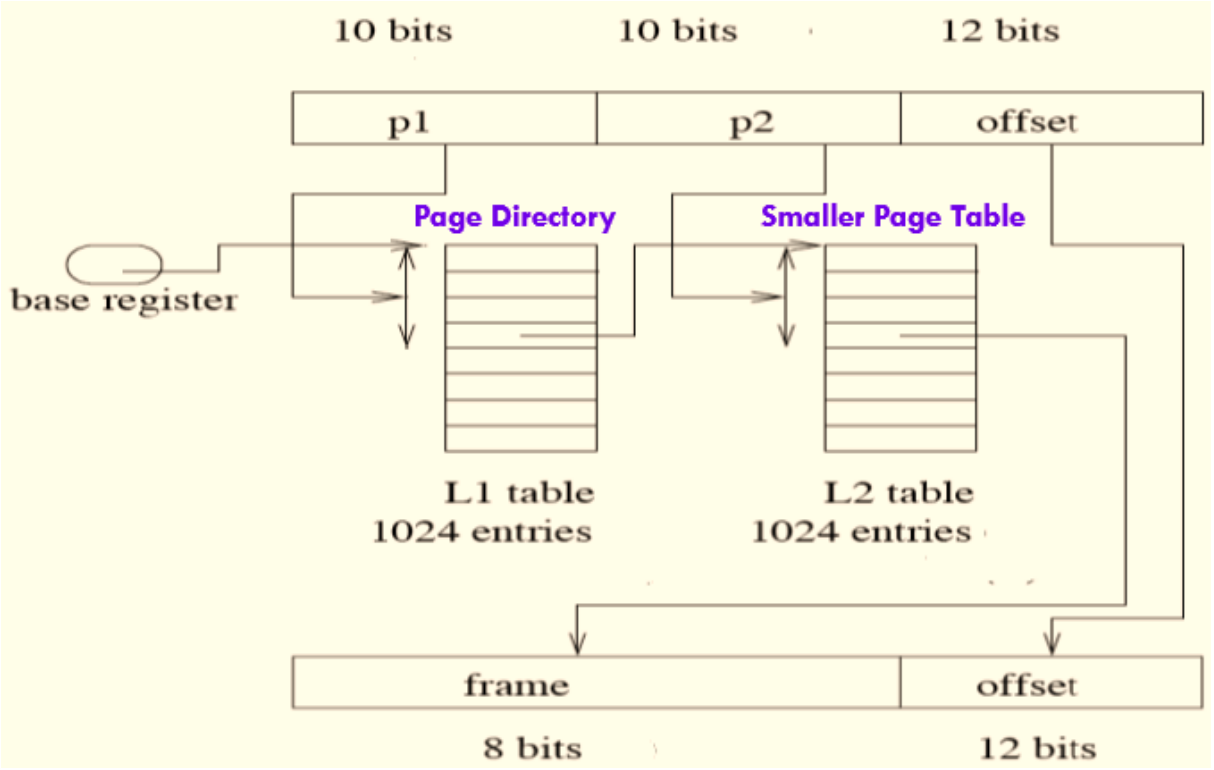


4 bytes

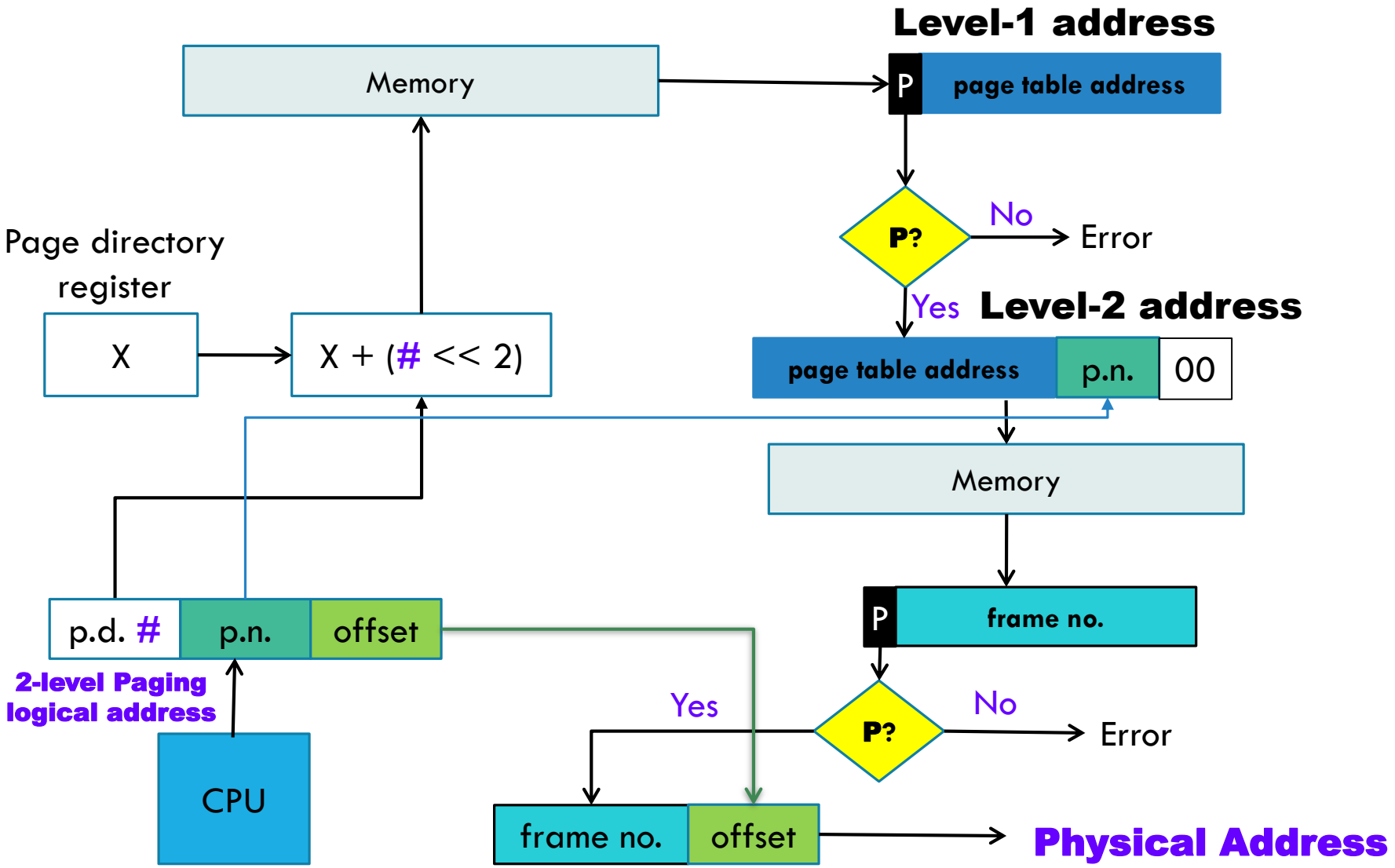
2-LEVEL PAGING LOGICAL ADDRESS



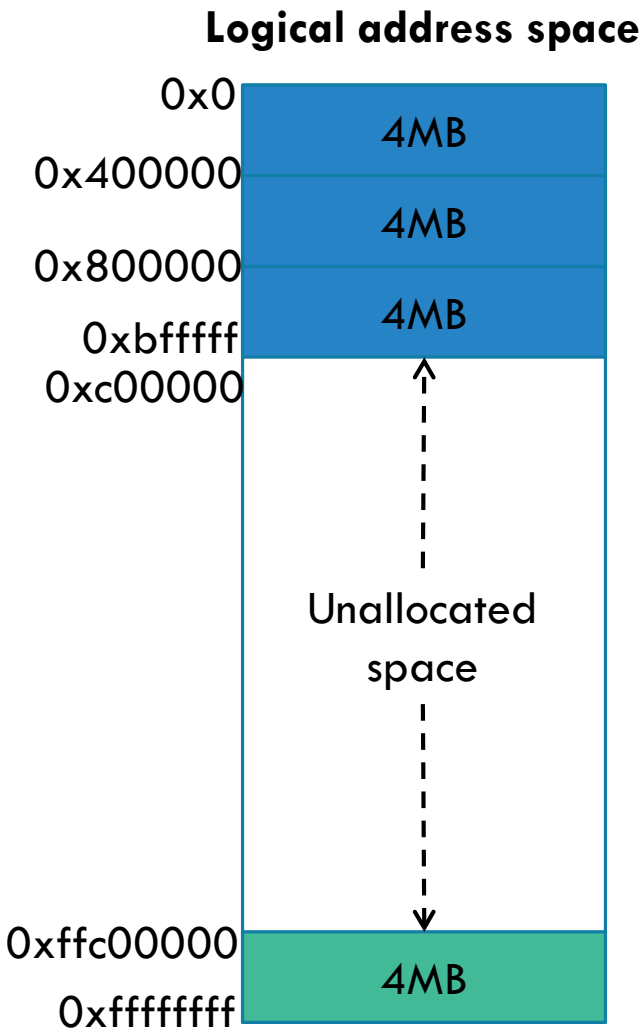
- Now instead of just **2 parts**. The logical address consists of **3 parts**.



2-LEVEL PAGING LOGICAL ADDRESS TRANSLATION



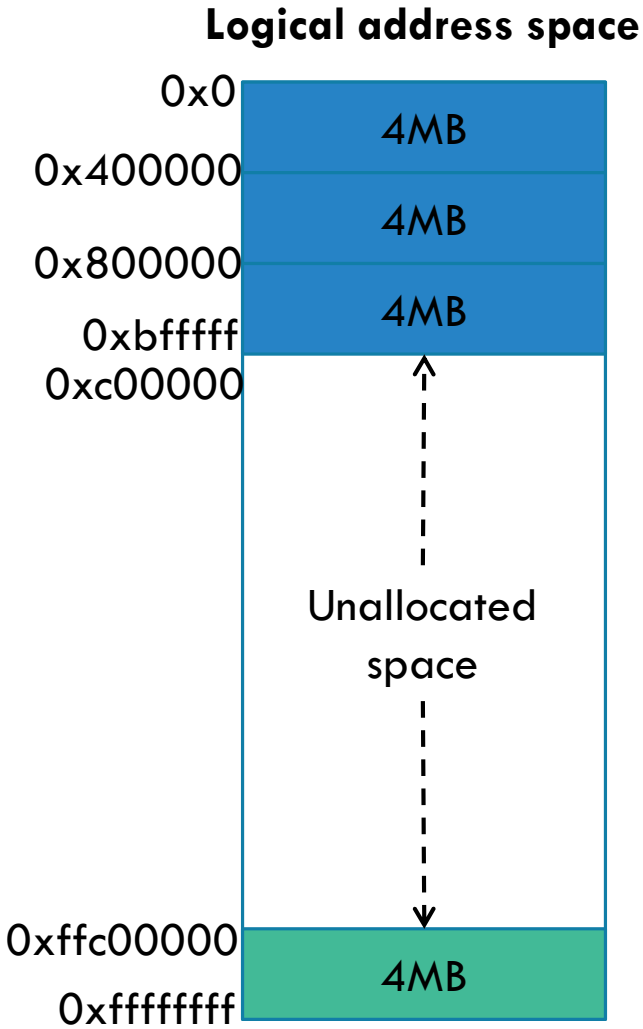
AN EXAMPLE FOR 2-LEVEL PAGING TO SHOW THE SIZE OF PAGE TABLES IN RAM



Suppose that this is a the logical address space of a process currently running.

The first **12MB** are **allocated**. The last **4MB** is allocated. The rest of the space is **un-allocated**.

WHAT IF WE HAVE 1-LEVEL PAGING?



Consider the page numbers for these pages.

The region has three 4MB blocks whose addresses ranging from **0x0 to 0xbfffff** ≈ **12MB**

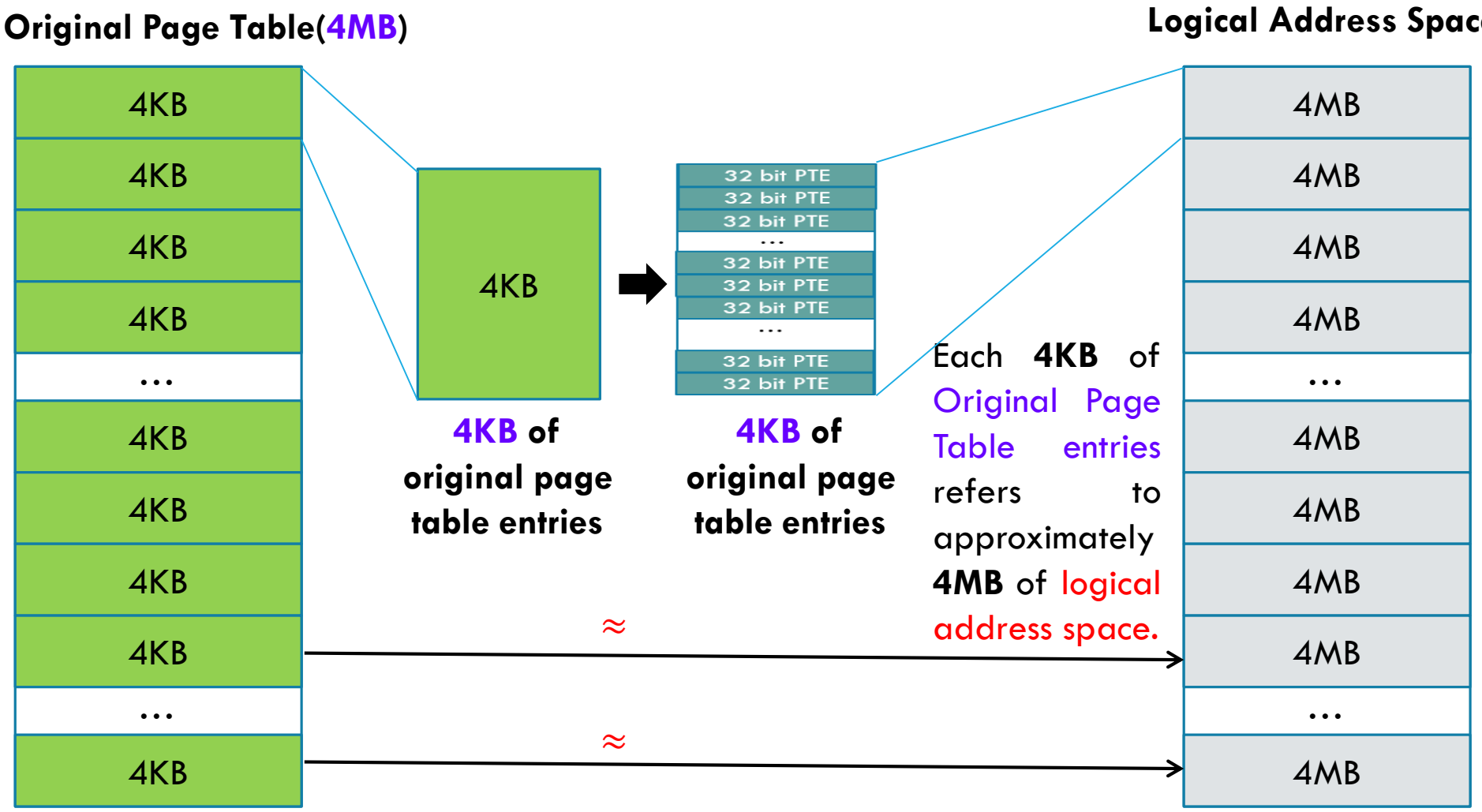
- Page numbers for these addresses (three 4MB block) range from **0x0 to 0xbff** = **3072**, which is 1024 PTE × 3

The region has a single 4MB block whose addresses ranging from **0xffc00000 to 0xffffffff** ≈ **4MB**

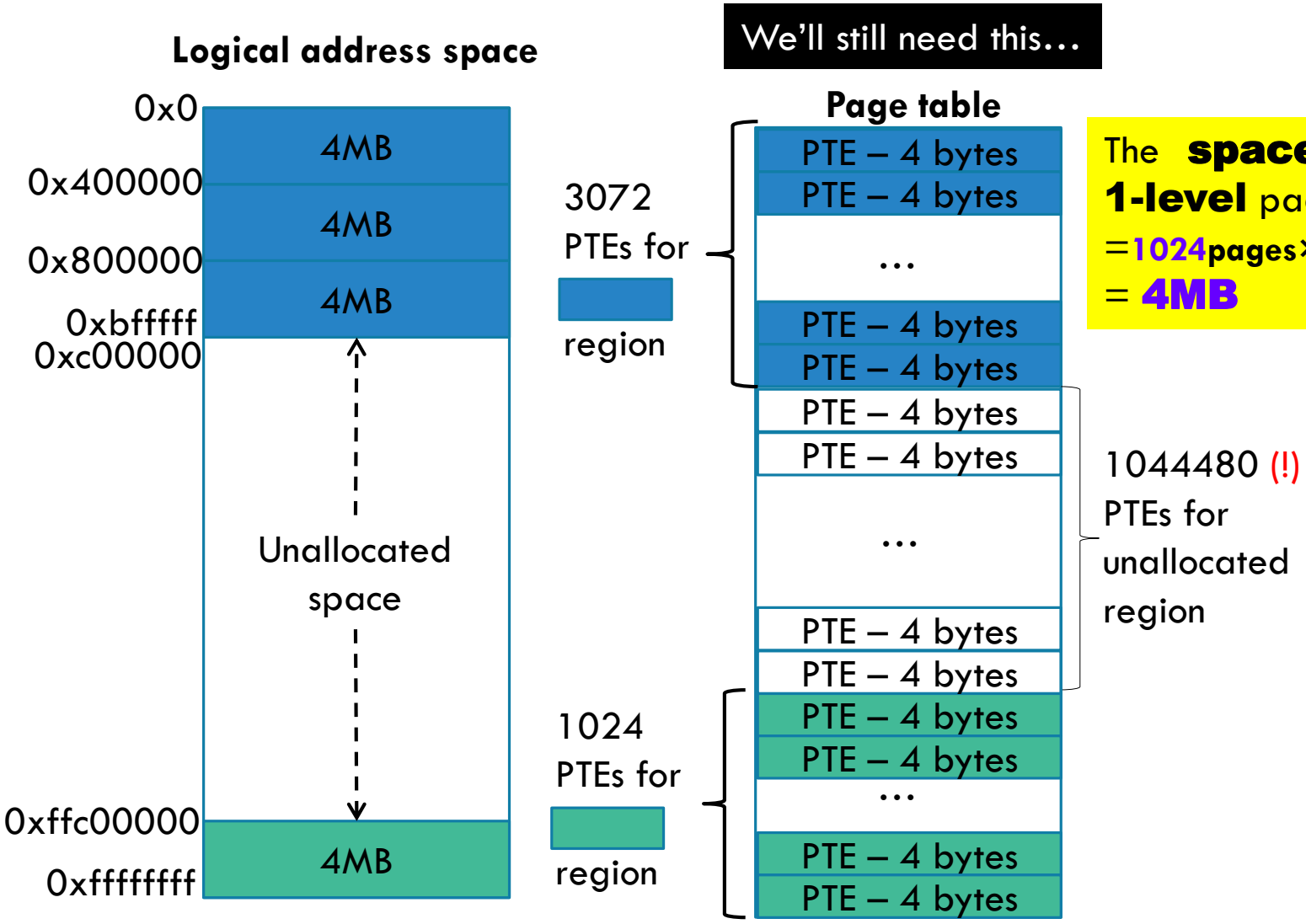
- Page numbers for these addresses (single 4MB block) range from 0xffc00 to 0xfffff = **1024**

WHAT IF WE HAVE 1-LEVEL PAGING?

- There is a **one-to-one mapping** between a PTE and a Page. **The size of the page table is proportional to the number of virtual pages.**

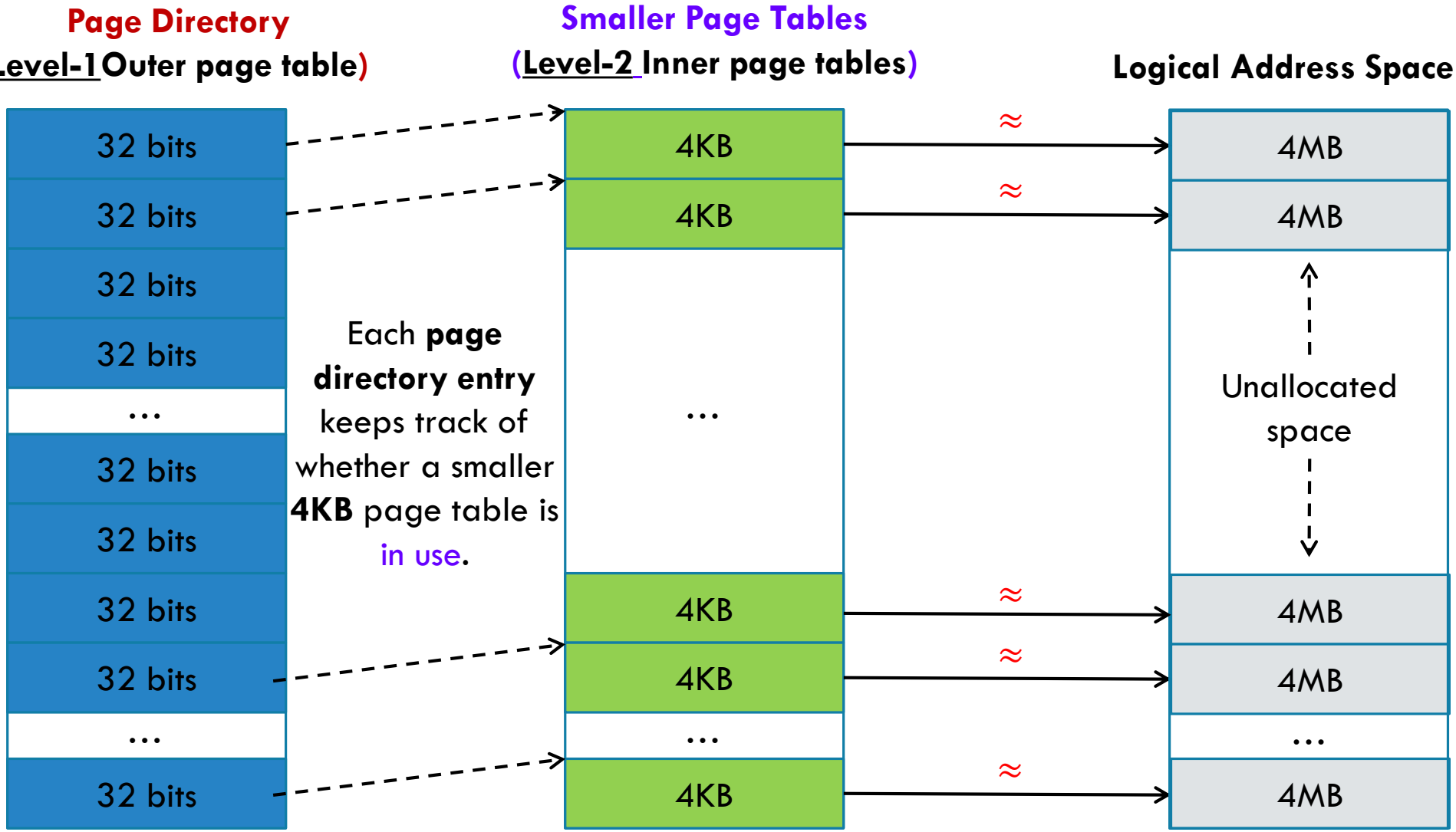


WHAT IF WE HAVE 1-LEVEL PAGING?

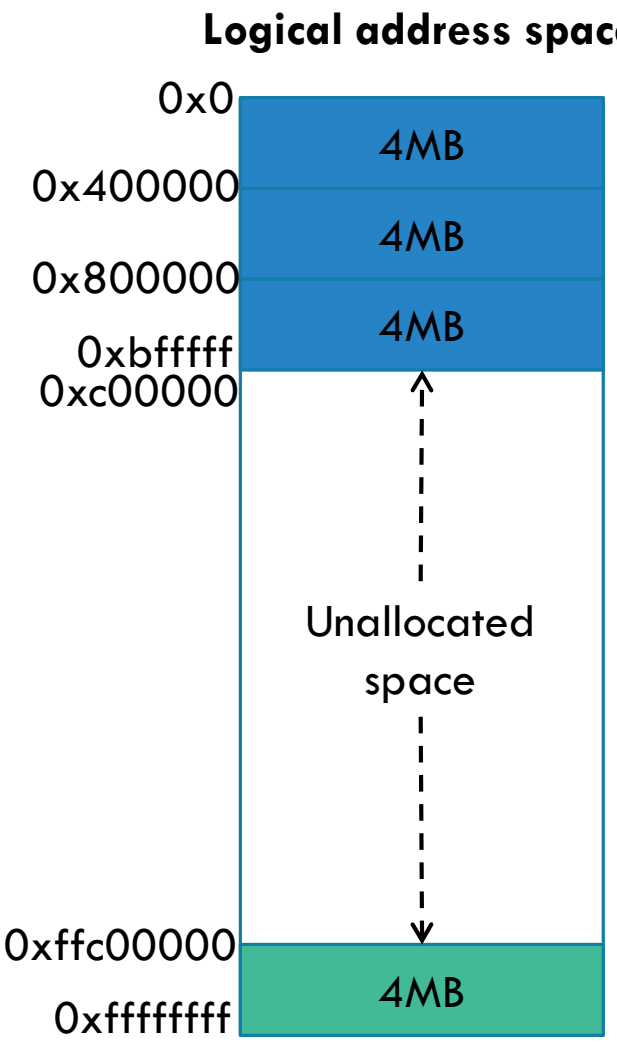


WHAT IF WE HAVE 2-LEVEL PAGING?

- **The inner page table is only created for the portions of the address space that are actually in use.**



WHAT IF WE HAVE 2-LEVEL PAGING?



Consider the range of frame and page numbers for these pages:

The region has three 4MB blocks whose addresses range from **0x00000000** to **0xbfffff.**

Page Directory numbers for these addresses (each 4MB block) range from **0x0** to **0x2.**

Page numbers for these addresses (**each** 4MB block) range from **0** to **0x3FF = 1024.**

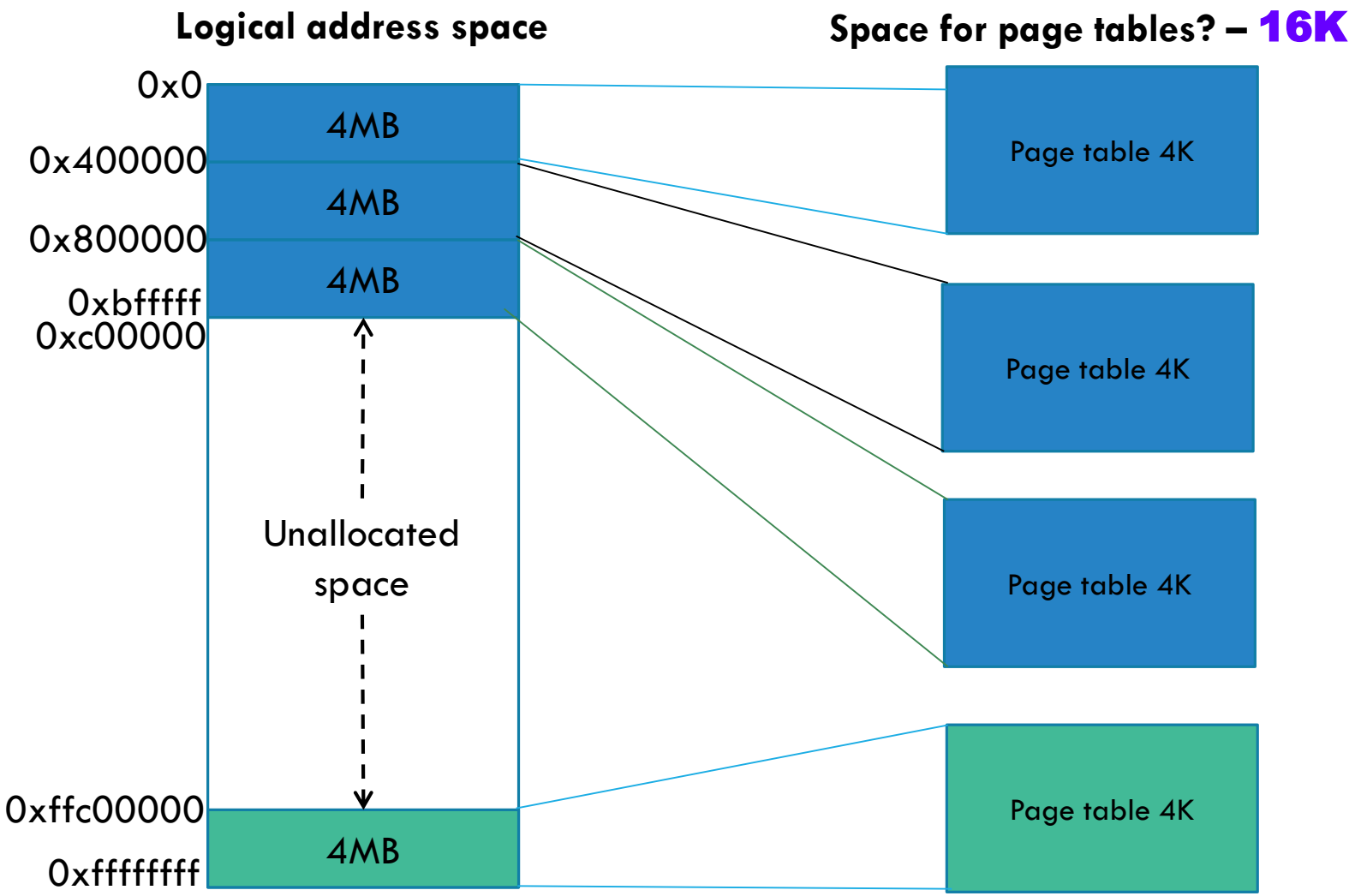
The region has a single 4MB block whose addresses ranging from **0xffc00000** to **0xffffffff.**

Page Directory numbers for these addresses (single 4MB block) range from **0x3FF** to **0x3FF.**

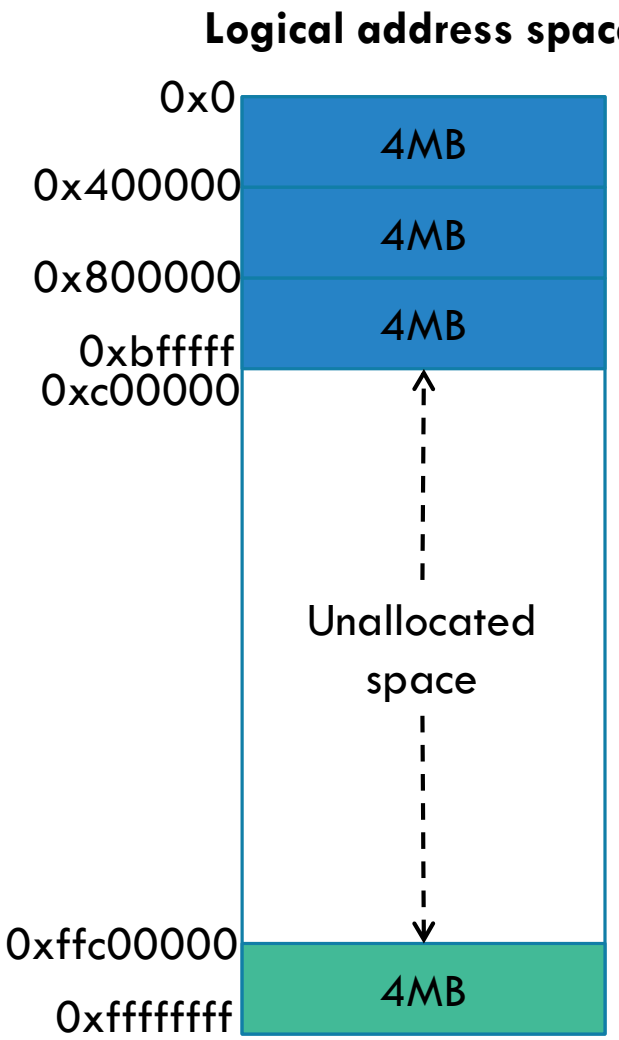
Page numbers for these addresses (**single** 4MB block) range from **0** to **0x3FF = 1024.**

WHAT IF WE HAVE 2-LEVEL PAGING?

The inner page table is only created for the portions of the address space that are actually in use.



COMPARING 1-LEVEL WITH 2-LEVEL PAGING



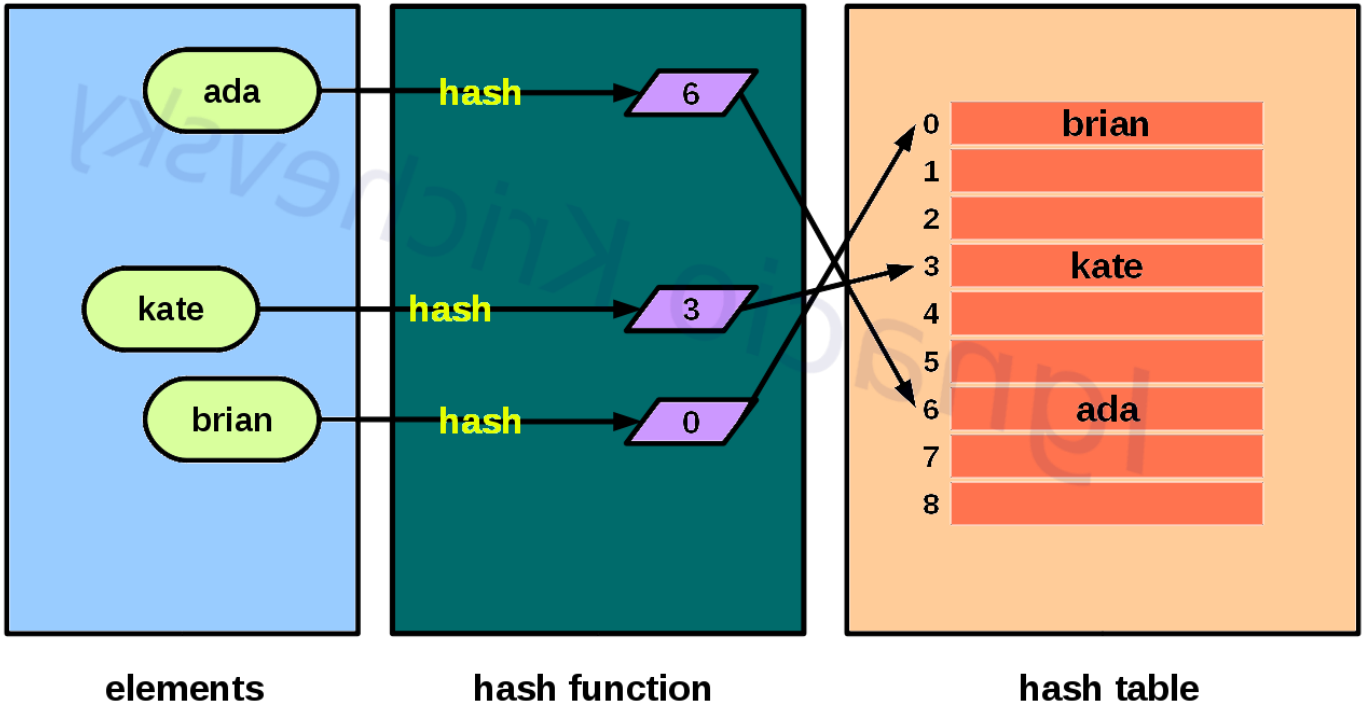
	1-level paging	2-level paging
Space usage	Same for all processes	Scale with allocated memory for process
Scalability	Size doesn't scale well.	Better scaling in terms of size of page tables.
Speed	1 extra memory lookup	2 extra memory lookup

COMPARING SEGMENTATION WITH PAGING

Segmentation	Paging (1-level)
Any size allocation	Allocation in units of pages
External Fragmentation	Internal Fragmentation
Each process has multiple contiguous logical address space.	Each process has only 1 contiguous logical address space.
Logical address – a tuple: (segment #, offset)	Logical address – a tuple: (page #, offset)
When need to make space for new segments in memory, entire segment needs to be swap out.	When need to make space for new page in memory, only 1 page needs to be swap out.

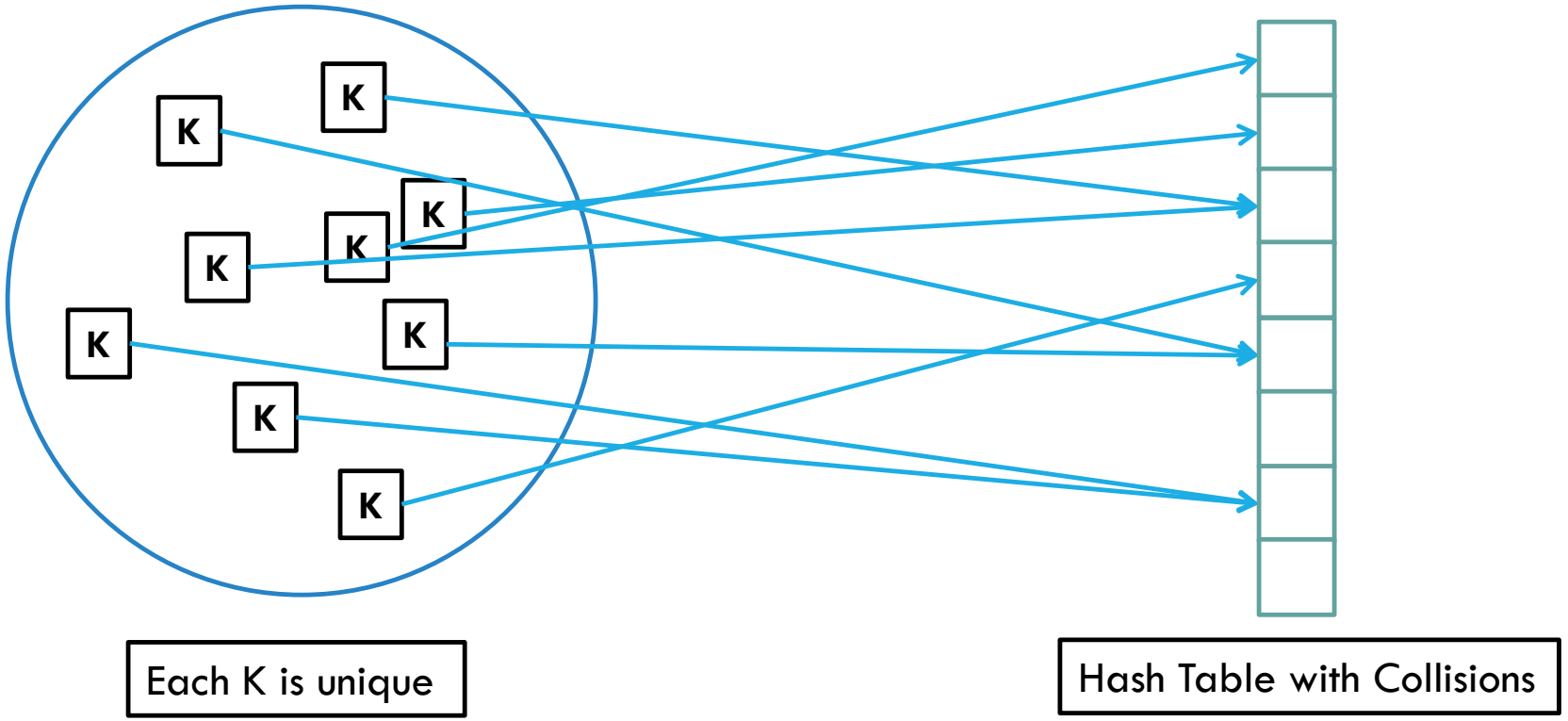
WHAT IS HASHING?

- A **hash function** maps a key (of any type) into an index (an integer) by performing certain arithmetic operations on the key.
- A hash function is a many-to-one mapping. **Different inputs can have the same output – Collision.**
- The resulting hashed integer indexes into a (hash) table.



WHAT IS COLLISION IN HASHING?

- Whenever two keys hash to the same index, then we have a **collision** in hash table.

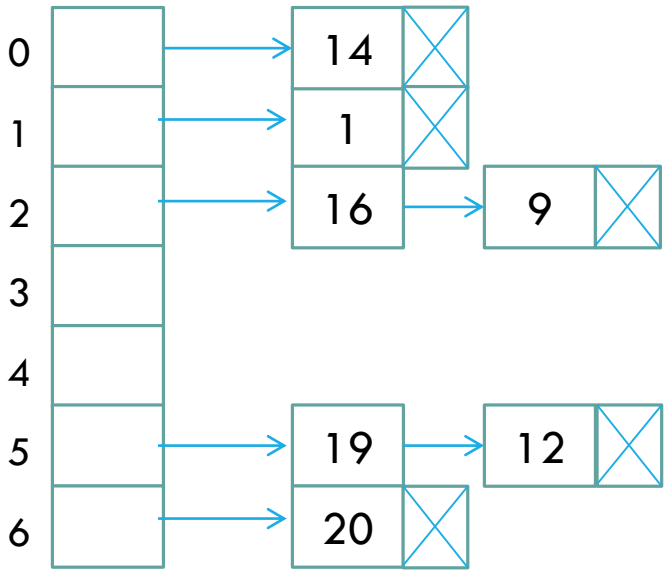


RESOLVING COLLISION IN HASHING-CHAINING

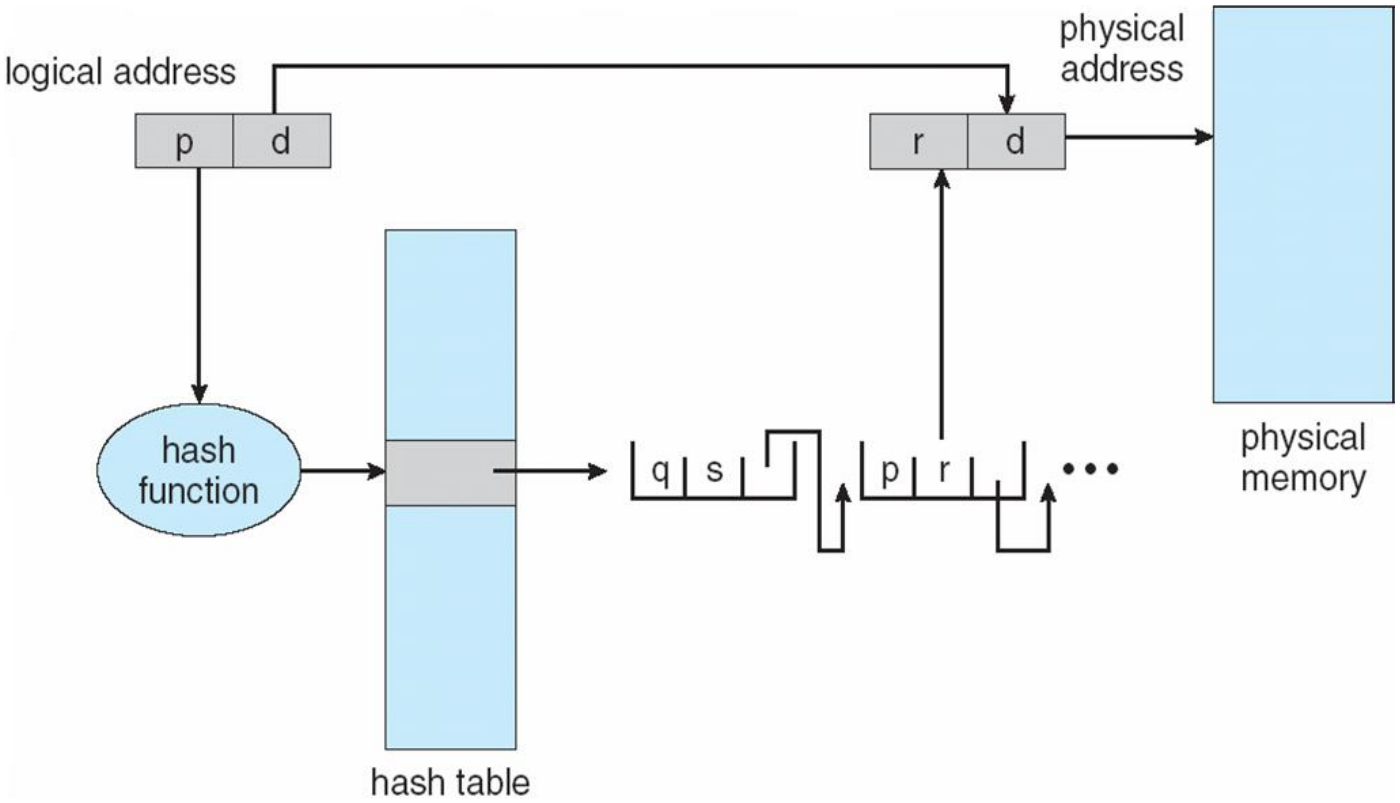
- One of the methods used for Collision Resolution is **Chaining**.
- The **chaining** technique uses **linked-lists** to store the data items.
- The hash table simply contains **pointers to the first item** in each linked list.

Example: Insert the following keys into the hash table: 19, 16, 9, 14, 1, 12, 20

```
h(x) = x % size
```



HASHED PAGE TABLE



HASHED PAGE TABLES

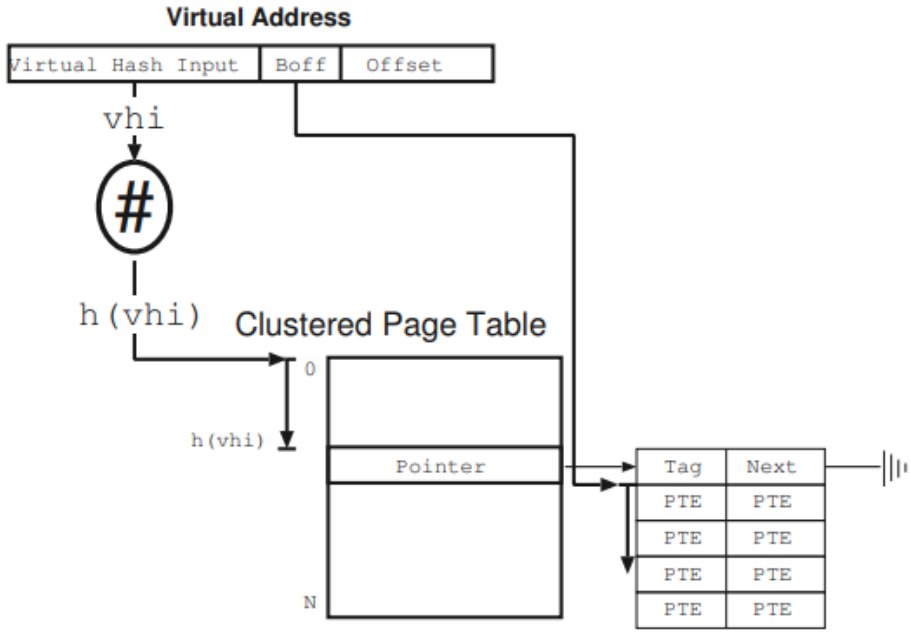
- This approach is common in handling **address spaces > 32 bits**
- The virtual page number is hashed into a page table
  - This page table contains a chain of elements hashing to the same location
- Each element contains
  - the virtual page number
  - the value of the mapped page frame
  - a pointer to the next element
- Virtual page numbers are compared in this chain searching for a match
  - *If a match is found*, the corresponding physical frame is extracted
  - *If a match is not found* a **page fault** exception is executed and the new page is hashed

Variation for 64-bit addresses is **clustered page tables**

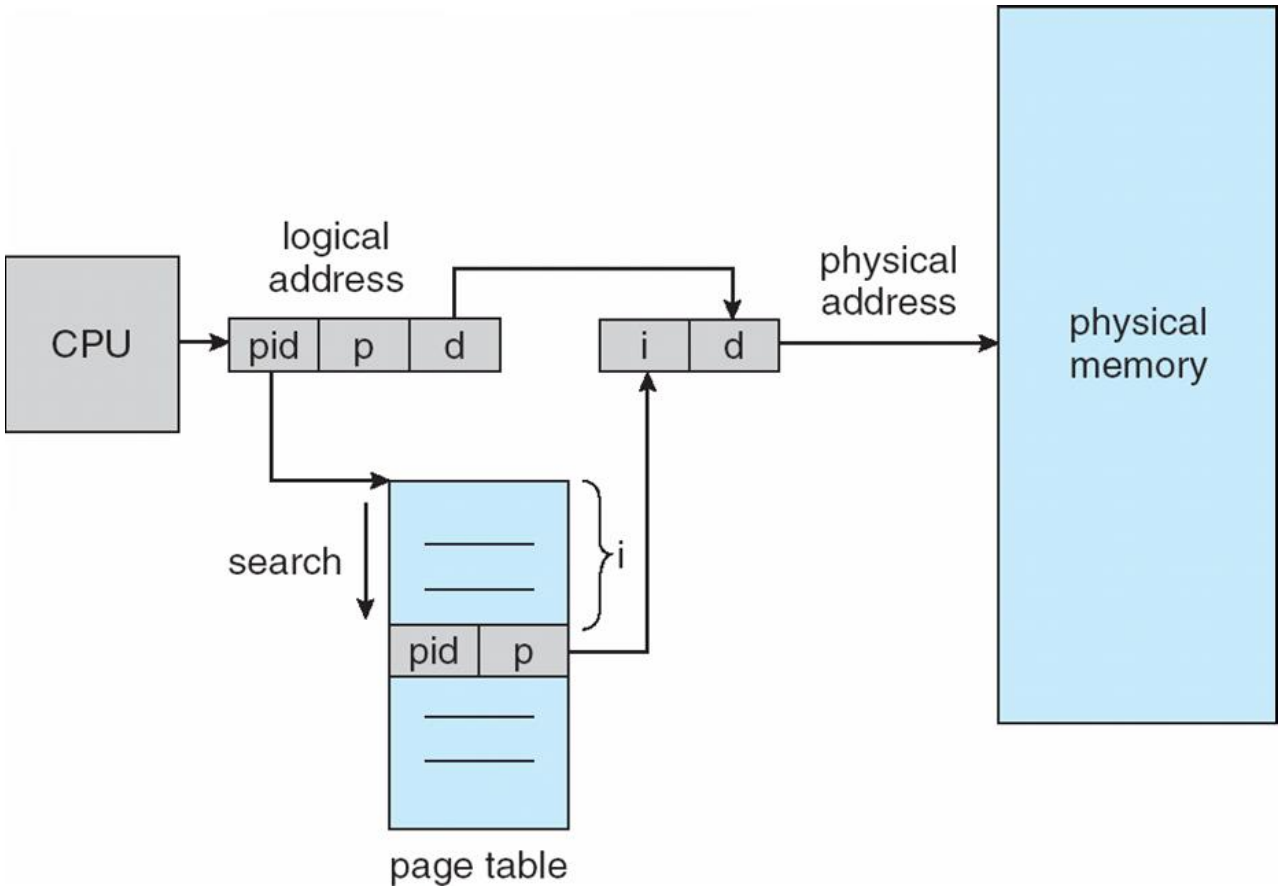
- Similar to hashed but each entry refers to several pages (such as 16) rather than 1
- Especially useful for **sparse** address spaces (where memory references are non-contiguous and scattered)

CLUSTERED PAGE TABLE

- Each entry in the table contains a cluster of PTEs — a set of PTEs for contiguous pages in the virtual address space.
- The virtual page number is split into two parts: the virtual hash input (VHI), and the block offset (BOFF).
  - The first part is hashed and used as an index into the CPT. Chained out from this are a number of clustered page table entries (CPTEs), each of which holds a single tag (which should match the VHI) and a set of n PTEs, where n = |BOFF|. CPTEs also contain pointer fields to allow the chaining of entries whose VHI hashes to the same value.



INVERTED PAGE TABLE (IPT) ARCHITECTURE



- There is only one page table present in the system. **NO per-process page tables !!!**
- This page table has only one entry for a physical frame.

There is only one virtual page entry for every physical frame is IPT !!!

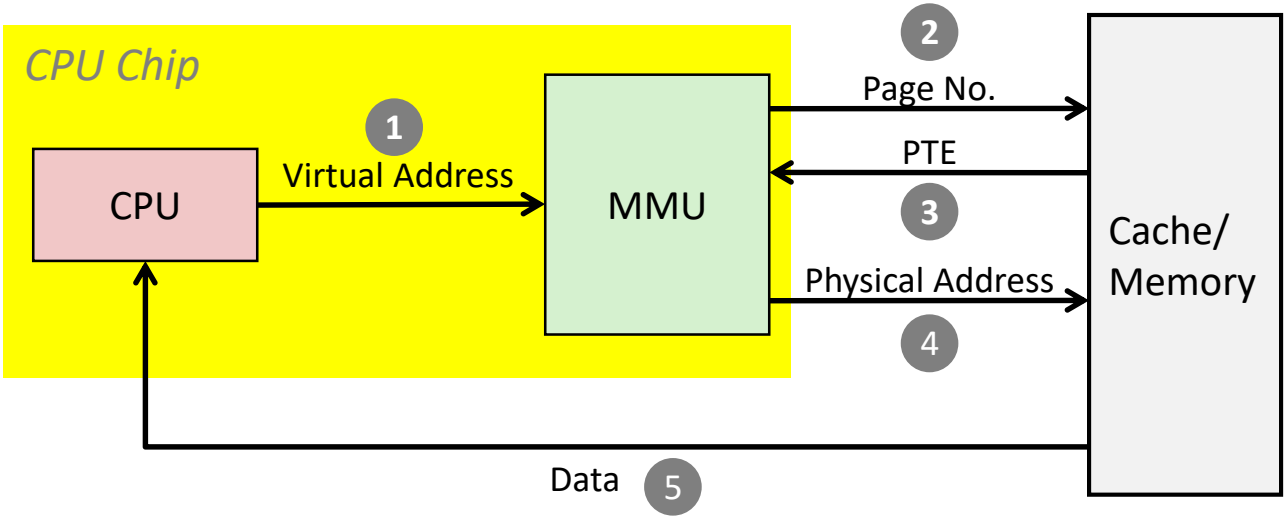
INVERTED PAGE TABLE (IPT)

- Rather than each process having a page table and keeping track of all possible logical pages, it is efficient to track all physical pages (frames).
- Inverted page table has **one entry for each real page (frame) of memory.**
- Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page.

<process-id, page-number, offset>

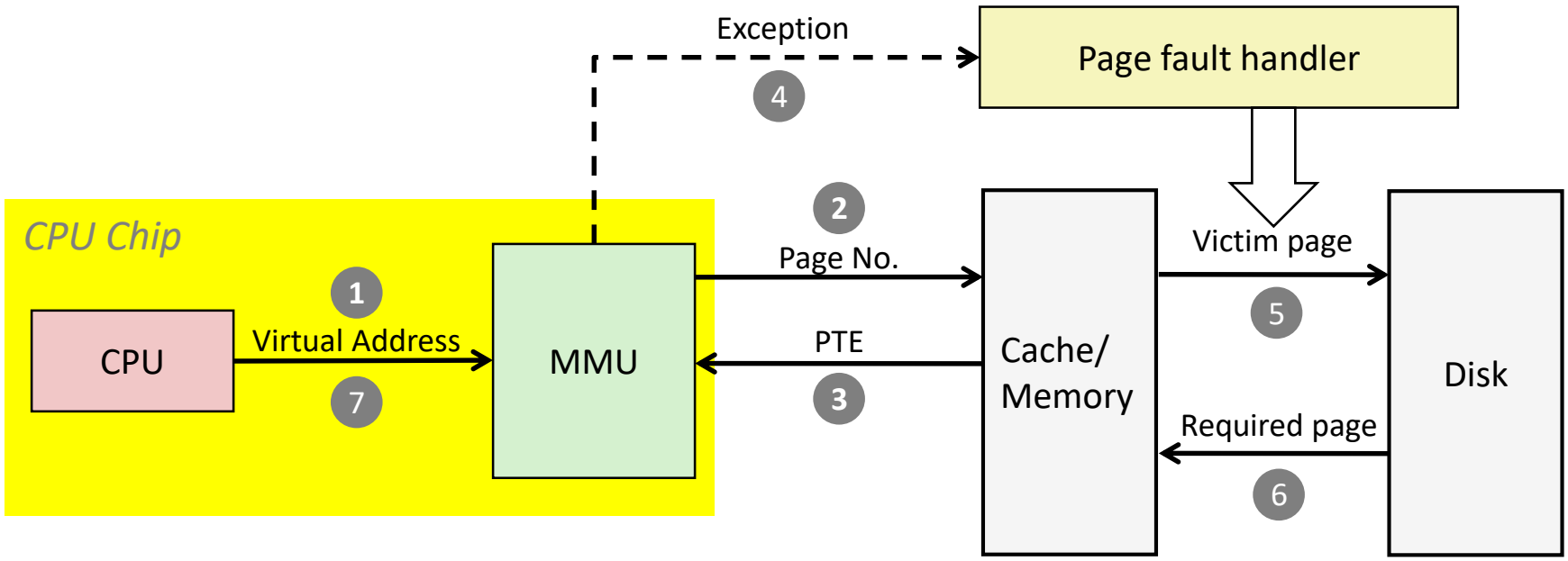
- **Decreases memory needed to store each page table, but increases the time needed to search the table when a page reference occurs:**
  - Use hash table to limit the search to one — or at most a few — page-table entries
  - TLB can accelerate access
- But how to implement **shared memory**?
  - **There is only one virtual page entry for every physical frame is IPT !!!**
    - Only one mapping of a virtual address to the shared physical address may occur at a time
    - A reference by another process sharing the memory will result in a page fault and replace the mapping with a different virtual address

ADDRESS TRANSLATION: PAGE HIT



1. The processor sends the virtual address to MMU
2. & 3. MMU fetches the Page Table Entry (PTE) from the page table in memory
4. If the valid bit is ONE. MMU computes and sends the physical address to cache/memory
5. Cache/memory sends data to the processor

ADDRESS TRANSLATION: PAGE FAULT



(1) The processor sends the virtual address to MMU

(2-3) MMU fetches the page table entry (PTE) from the page table in memory

(4) If the valid bit is ZERO, so MMU triggers page fault exception

(5) Interupt handler identifies victim (and, if **dirty**, pages it out to disk) - **Dirty: modified**

(6) Interupt handler pages in the required page and updates PTE in memory

(7) The handler returns to the original process, restarting the faulting instruction

TRANSLATION LOOK-ASIDE BUFFER (TLB)

- It's an associative, high-speed memory.
  - Each entry consists of **<Key, Value>** pair
- It's a **Cache** of **page numbers to physical address mapping.**
  - Speeds up the address translation process.
- However, a miss will still incur a **page fault** and initiate a page fault handling procedure (look at the last slide).
- **The TLB resides within the MMU.**
- **It is flushed with every context switch.**

VIRTUAL MEMORY SPACE

- We allow the logical memory size to exceed the physical memory size. **But How?**
  - The process pages are added to the page table as needed.
  - When physical memory has been exceeded, ***victim frames*** are written (temporarily) to the **hard disk** to free up physical memory.
  - ***Swap space*** is used for writing frames: a contiguous block of disk space set aside for this purpose.

FREEING PHYSICAL MEMORY

- **Swapping** – an **entire process** (including data, stack, and code segments) is removed from physical memory and written to disk
  - Frees up several frames of physical memory
  - Very costly
- **Paging** – a **physical page**(frame) of physical memory is written to disk
  - Frees up a single frame of physical memory
  - Several contiguous frames may be written to disk at one time
- Swapping and paging are expensive (in terms of time), so **should be minimized** !!!

DEMAND PAGING (1)

- Also called *lazy swapping*
- **A page in virtual memory is assigned a physical frame only when the memory within that page is accessed.**
- Each page table entry (PTE) in the page table contains a bit (flag) to mark the page:
  - **Valid** – the logical page has been assigned a physical frame
  - **Invalid** – the logical page has not been assigned a physical frame

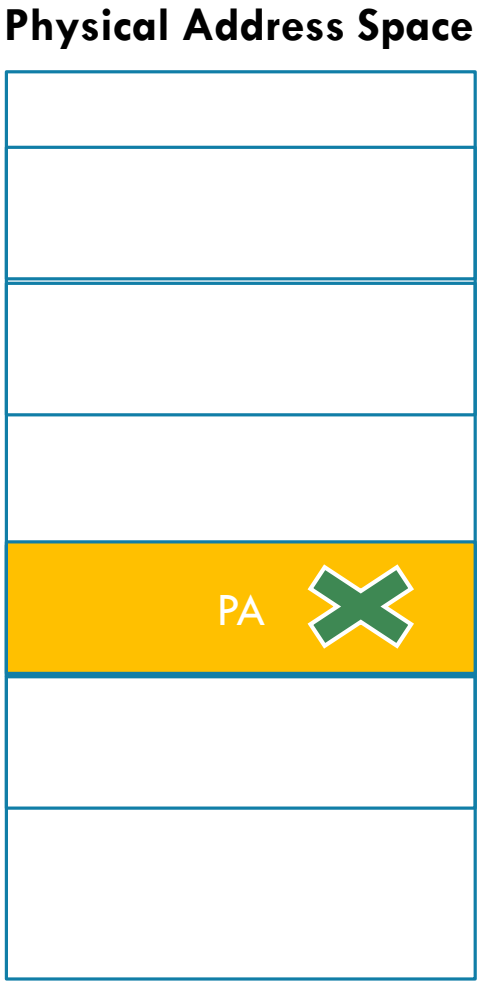
DEMAND PAGING (2)

- When a location in logical memory is accessed, the valid/invalid bit of the corresponding PTE in the page table is examined:
  - If the page is valid, process execution continues as normal
  - If the page is invalid, a **page fault** (interrupt, or trap) occurs
    - The OS finds a free frame in physical memory
    - The desired page is read into the new frame
    - The page table is updated with the new page information, and valid/invalid bit is set to valid
    - Process execution is resumed

PAGE REPLACEMENT

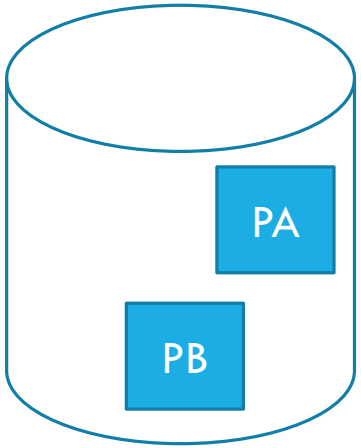
- When a **page fault occurs**, but there are no free **physical memory frames**:
  - A **victim frame** must be chosen – this frame will be paged out (written to disk)
  - The page that caused the fault will be assigned to the victim frame and paged in (brought into the memory)
- **Optimization:** a page that has not been modified since it was first read in (such as a code segment) need not be saved to disk (it is already there) – only the valid/invalid bit in the corresponding PTE in the page table needs to be changed.
  - A **dirty bit** indicates whether the page is modified after it was brought in

OPTIMIZATION OF PAGE SWAPPING



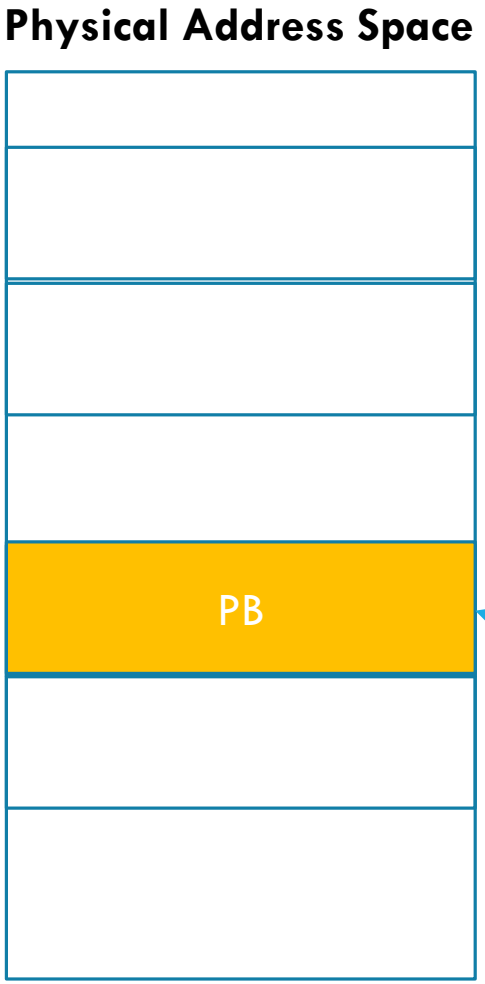
The page **PA** has not been written to at all…

Delete **PA.** It **need not** be written to disk (**it's not dirty)**



Need to page-in a page PB…

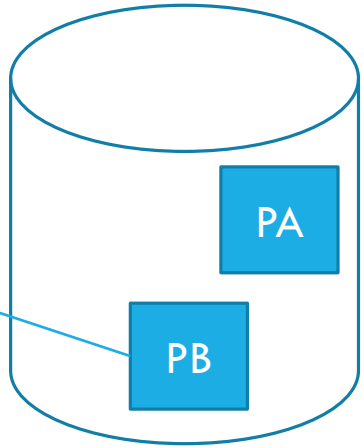
OPTIMIZATION OF PAGE SWAPPING



The page **PA** has not been written to at all…

Delete **PA.** It **need not** be written to disk (**it's not dirty)**

Page-in **PB** in place of **PA**



Need to page-in a page PB…

ANALYZING PAGE REPLACEMENT ALGORITHMS

- Page reference sequence
  - This is a series of page numbers being requested.

    1, 0, 7, 1, 0, 2, 1, 2, 3, 0, 3, 2, 4, 0, 3, 0, 2, 1

- For this particular page reference sequence
  - What is the number of **page faults**?
  - The aim of any page replacement algorithm is to minimize the number of **page faults**.

FIFO PAGE REPLACEMENT (1)

- This page replacement algorithm chooses the page that has been in physical memory for the **longest period of time** as the **victim page.**

- Easy to implement using a **queue**.

- Often does not yield optimal performance: more than the minimum number of page faults will occur

FIFO PAGE REPLACEMENT ANALYSIS(1)

- Suppose there are 3 frames of physical memory
- Suppose we access the following sequence of pages in logical memory:

p: **7, 0, 1, 2, 0, 3, 0, 2, 0, 4, 0, 2**

3 Frames

The queue states are:

Page reference string	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**2**	**0**	**4**	**0**	**2**
Frame 1	7	7	7	2	2	2	2	2	2	4	4	4
Frame 2		0	0	0	0	3	3	3	3	3	3	2
Frame 3			1	1	1	1	0	0	0	0	0	0
Page Fault?	✓	✓	✓	✓		✓	✓			✓		✓

remove oldest

PF = 8

FIFO PAGE REPLACEMENT ANALYSIS(2)

3 Frames

Page reference string	**1**	**0**	**7**	**1**	**0**	**2**	**1**	**2**	**3**	**0**	**3**	**2**	**4**	**0**	**3**	**0**	**2**	**1**
Frame 1	**1**	1	1	1	1	**2**	2	2	2	**0**	0	0	0	0	**3**	3	3	**1**
Frame 2		**0**	0	0	0	0	**1**	1	1	1	1	**2**	2	2	2	**0**	0	0
Frame 3			**7**	7	7	7	7	7	**3**	3	3	3	**4**	4	4	4	**2**	2
Page Fault?	**Y**	**Y**	**Y**			**Y**	**Y**		**Y**	**Y**		**Y**	**Y**		**Y**	**Y**	**Y**	**Y**

The total number of page faults : 13

BELADY'S ANOMALY

- Intuitively, what does it mean when we have **more frames**?
  - More physical memory, more RAM.
- How would you expect page faults to scale with a number of frames?
  - Expect Fewer page faults with more frames.
- **Belady's anomaly**
  - **For a particular page replacement algorithm and a page reference sequence, we have more page faults when we have more frames.**

FIFO DISPLAYS BELADY'S ANOMALY

3 Frames 9 Frames

Page reference string	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4
Page Fault?	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**	**Y**			**Y**	**Y**	

4 Frames 10 Frames

Page reference string	**1**	**2**	**3**	**4**	**1**	**2**	**5**	**1**	**2**	**3**	**4**	**5**
Frame 1	1	1	1	1	1	1	5	5	5	5	4	4
Frame 2		2	2	2	2	2	2	1	1	1	1	5
Frame 3			3	3	3	3	3	3	2	2	2	2
Frame 4				4	4	4	4	4	4	3	3	3
Page Fault?	**Y**	**Y**	**Y**	**Y**			**Y**	**Y**	**Y**	**Y**	**Y**	**Y**

SECOND CHANCE ALGORITHM

- Modification of the **FIFO** algorithm.

- Each page in memory has a **reference bit** associated with it.
  - Whenever a page is referenced (a memory location within the page is accessed), the reference bit is set to **1**
  - If a page has been selected for possible replacement, the value of the reference bit is examined:
    - If it is **0**, the page is replaced
    - If it is **1**, the bit is set to **0**, and the next item in the queue is selected in a ***circular queue fashion***

SECOND CHANCE ANALYSIS (1)

3 Frames

- Total of **7 page** faults

Page reference string	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**2**	**0**	**4**	**0**	**2**
Frame 1	**7*** **+**	**7*** +	**7*** +	**2** **+**	2 +	**2*** +	**2*** +	**2*** **+**	**2*** +	**4** **+**	4 +	**4*** +
Frame 2		**0** **+**	0 +	**0*** **-**	**0*** **+**	0 -	**0** **+**	0 +	**0** **+**	**0*** **-**	**0*** **+**	0 -
Frame 3			**1** **+**	1 -	1 -	**3** **+**	3 +	3 +	3 +	3 -	**3** **-**	**2** **+**
Page Fault?	**Y**	**Y**	**Y**	**Y**		**Y**				**Y**		**Y**

* indicates which page is to be replaced next

+ indicates the reference bit for this page is set 1

- indicates the reference bit for this page is not set 0

SECOND CHANCE ANALYSIS (2)

3 Frames

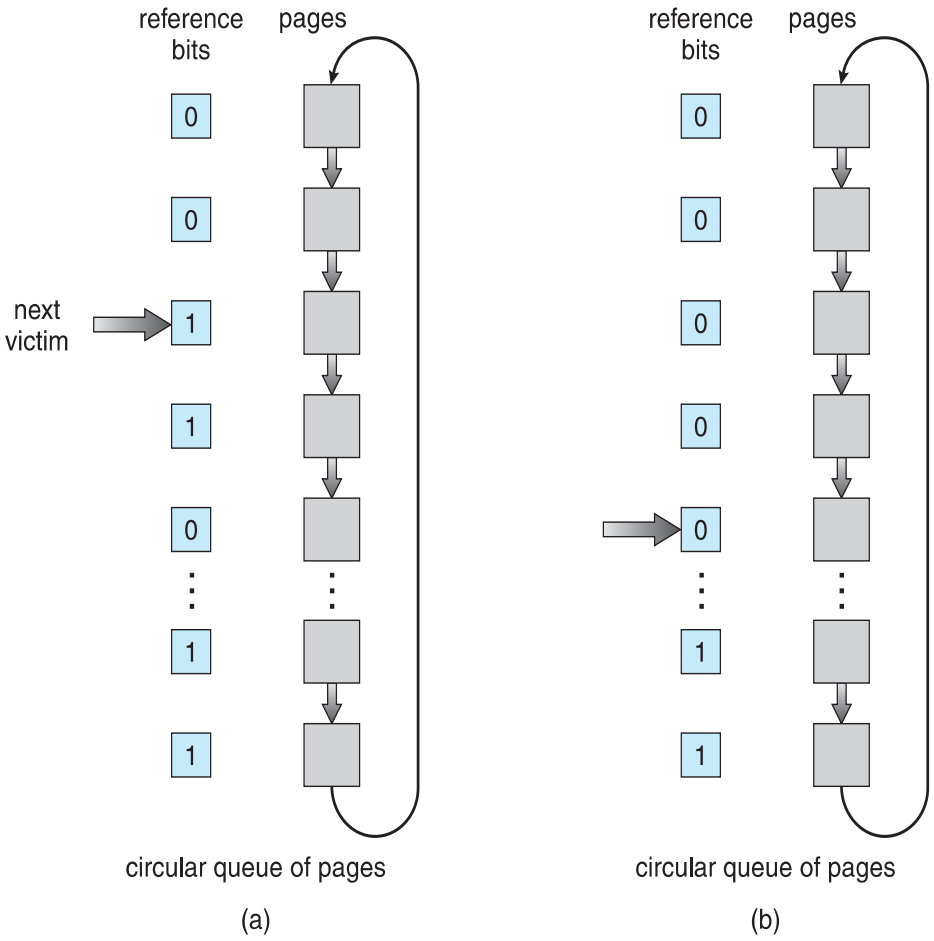
Page reference string	**1**	0	**7**	**1**	**0**	**2**	**1**	**2**	**3**	**0**	**3**	**2**	**4**	**0**	**3**	**0**	**2**	**1**
Frame 1	1* +	1* +	1* +	1* +	1* +	2 +	2 +	2 +	2* +	0 +	0 +	0 +	0* -	0* +	0 -	0 +	0 -	0* -
Frame 2		0 +	0 +	0 +	0 +	0* -	1 +	1 +	1 +	1* -	1* -	2 +	2 -	2 -	3* +	3* +	2 -	2 -
Frame 3			7 +	7 +	7 +	7 -	7* -	7* -	3 +	3 -	3 +	3* +	4 +	4 +	4 +	4 +	4* -	1 +
Page Fault?	**Y**	**Y**	**Y**			**Y**	**Y**		**Y**	**Y**		**Y**	**Y**		**Y**		**Y**	**Y**

* indicates which page is to be replaced next

+ indicates the reference bit for this page is set

- indicates the reference bit for this page is not set

SECOND-CHANCE (CLOCK) PAGE-REPLACEMENT ALGORITHM



ENHANCED SECOND-CHANCE ALGORITHM

Improve the algorithm by using reference-bit and modify-bit (if available) as an ordered pair:

(reference-bit, modify-bit)

1. **(0, 0)** neither recently used nor modified – best page to replace
2. **(0, 1)** not recently used but modified – not quite as good, must write out before replacement
3. **(1, 0)** recently used but clean – probably will be used again soon
4. **(1, 1)** recently used and modified – probably will be used again soon and need to write out before replacement

When page replacement is called for, use the **clock scheme** but replace the first page encountered in the lowest of the 4 classes listed above

- **Might need to search the circular queue several times**

OPTIMAL PAGE REPLACEMENT (1)

when a **page** needs to be swapped in, the operating system swaps out the **page** whose next use will occur farthest in the future.

Need check ahead who is being use and who is not being use in the future

Not used until …

Page reference string	**1**	**0**	**7**	**1**	**0**	**2**	**1**	**2**	**3**	**0**	**3**	**2**	**4**	**0**	**3**	**0**	**4**	**1**
Frame 1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3
Frame 2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Frame 3			7	7	7	2	2	2	2	2	2	2	4	4	4	4	4	4
Page Fault?	Y	Y	Y			Y			Y				Y					

Not used in the future!

OPTIMAL PAGE REPLACEMENT (2)

when a **page** needs to be swapped in, the operating system swaps out the **page** whose next use will occur farthest in the future.

Not used until …

Page reference string	**1**	**0**	**7**	**1**	**0**	**2**	**1**	**2**	**3**	**0**	**3**	**2**	**4**	**0**	**3**	**0**	**2**	**1**
Frame 1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	2	1
Frame 2		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Frame 3			7	7	7	2	2	2	2	2	2	2	4	4	4	4	4	4
Page Fault?	Y	Y	Y			Y			Y				Y				Y	Y

Not used in the future!

OPTIMAL PAGE REPLACEMENT (3)

3 Frames

Page reference string	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**2**	**0**	**4**	**0**	**2**
Frame 1	**7**	7	7	**2**	2	2	2	**2**	2	2	2	**2**
Frame 2		**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0
Frame 3			**1**	1	1	**3**	3	3	3	**4**	4	4
Page Fault?	**Y**	**Y**	**Y**	**Y**		**Y**				**Y**		

- Total of **6 page** faults

OPTIMAL PAGE REPLACEMENT

- The optimal page replacement algorithm is **difficult/expensive** to implement.
  - ❖ **It requires future knowledge of the Page Reference sequence !!!**
- It is mainly used for comparison studies as a Benchmark.
- **LRU page replacement** algorithm is an approximation of the Optimal Page Replacement algorithm !!!

LRU ALGORITHM

- **Least recently used algorithm** – the page that has not been used for the longest period of time is selected as the **victim frame.**

- Implemented in one of two ways:
  - **Timestamp** - whenever a page is referenced, it is marked with the **current time**
    - The **victim frame** is the page with the **smallest timestamp** value
  - **Stack is used** – if a page is referenced, it is removed from the stack and placed on top
    - The **victim frame** is the page at the **bottom of the stack**

LRU ALGORITHM ANALYSIS(1)

7 LRU 1 LRU 3 LRU

3 Frames

Page reference string	**7**	**0**	**1**	**2**	**0**	**3**	**0**	**2**	**0**	**4**	**0**	**2**
Frame 1	**7**	7	7	**2**	2	2	2	**2**	2	2	2	**2**
Frame 2		**0**	0	0	**0**	0	**0**	0	**0**	0	**0**	0
Frame 3			**1**	1	1	**3**	3	3	3	**4**	4	4
Page Fault?	**Y**	**Y**	**Y**	**Y**		**Y**				**Y**		

- Total of **6 page** faults

LRU ALGORITHM ANALYSIS(2)

3 Frames

Page reference string	**1**	**0**	**7**	**1**	**0**	**2**	**1**	**2**	**3**	**0**	**3**	**2**	**4**	**0**	**3**	**0**	**2**	**1**
Frame 1	**1**	1	1	**1**	1	1	**1**	1	1	**0**	0	0	**4**	4	4	4	**2**	2
Frame 2		**0**	0	0	**0**	0	0	0	**3**	3	**3**	3	3	**0**	0	**0**	0	0
Frame 3			**7**	7	7	**2**	2	**2**	2	2	2	**2**	2	2	**3**	3	3	**1**
Page Fault?	**Y**	**Y**	**Y**			**Y**			**Y**	**Y**			**Y**	**Y**	**Y**		**Y**	**Y**

11 PF

THRASHING

- Paging is expensive
  - Page fault (interrupt) handling: save PCB
  - Search for a free frame - apply page replacement algorithm if none free
  - Copy to and from the disk
  - I/O wait
  - Restore PCB and restart process
- **Thrashing** occurs when the system spends **more time** in paging than executing processes.
- This may happen if there are many processes running (CPU over-utilization).

AVOIDING PAGE FAULTS

As a programmer, there are some things you can do to decrease the amount of paging that you program will undergo when executing – thereby reducing the execution time:

- **Localize variable access** when possible – when accessing a set of variables repeatedly, try to keep the variables **near each other** in memory.
- When doing two-dimensional array computations, **process the array elements by rows.**
- If you have to use a linked list with a large number of entries, you can either use a **cursor-based linked list,** or allocate nodes from a **contiguous block of memory.**

WORKING SET MODEL

- This model is used to prevent **Thrashing.**
- It works on the assumption of locality.
- A parameter Δ defines the working set window.
- The working set is the set of pages in the most recent Δ page references the process is currently using for its computation.
  - If a page is in active use, it will be in the working set.
  - If it is not, it will be dropped from the working set time Δ units after its last reference.