MEMORY MANAGEMENT

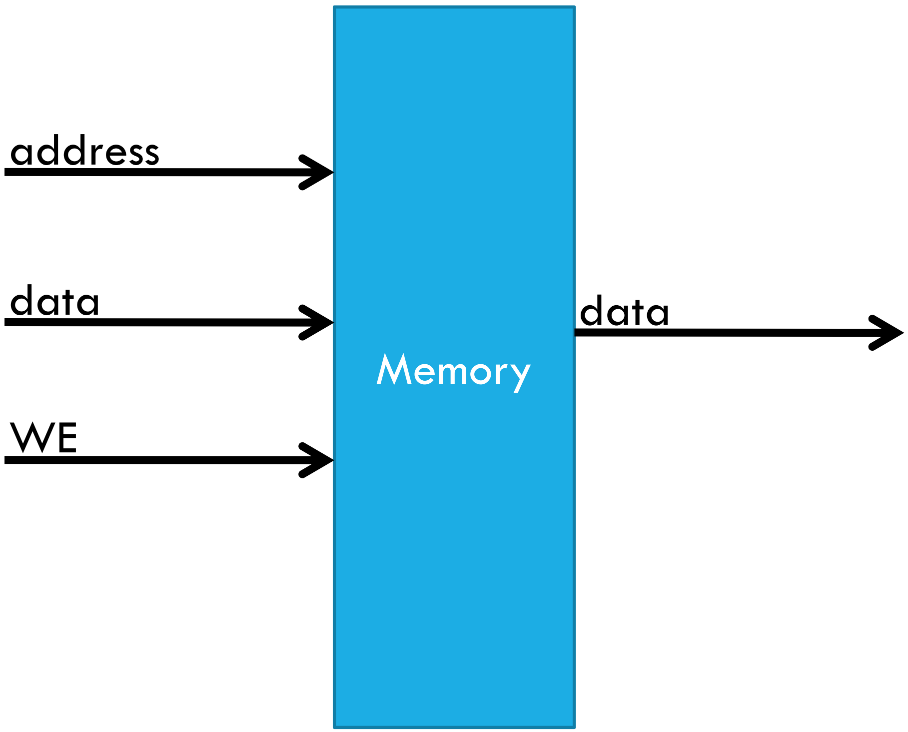
Instructor: William Zheng
Email:
william.zheng@digipen.edu
PHONE EXT: 1745

Physical Address Space

GOALS

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

MEMORY MODEL



I/O Device

Input

- Address
- Data
- WE (write-enable)

Output

- Data

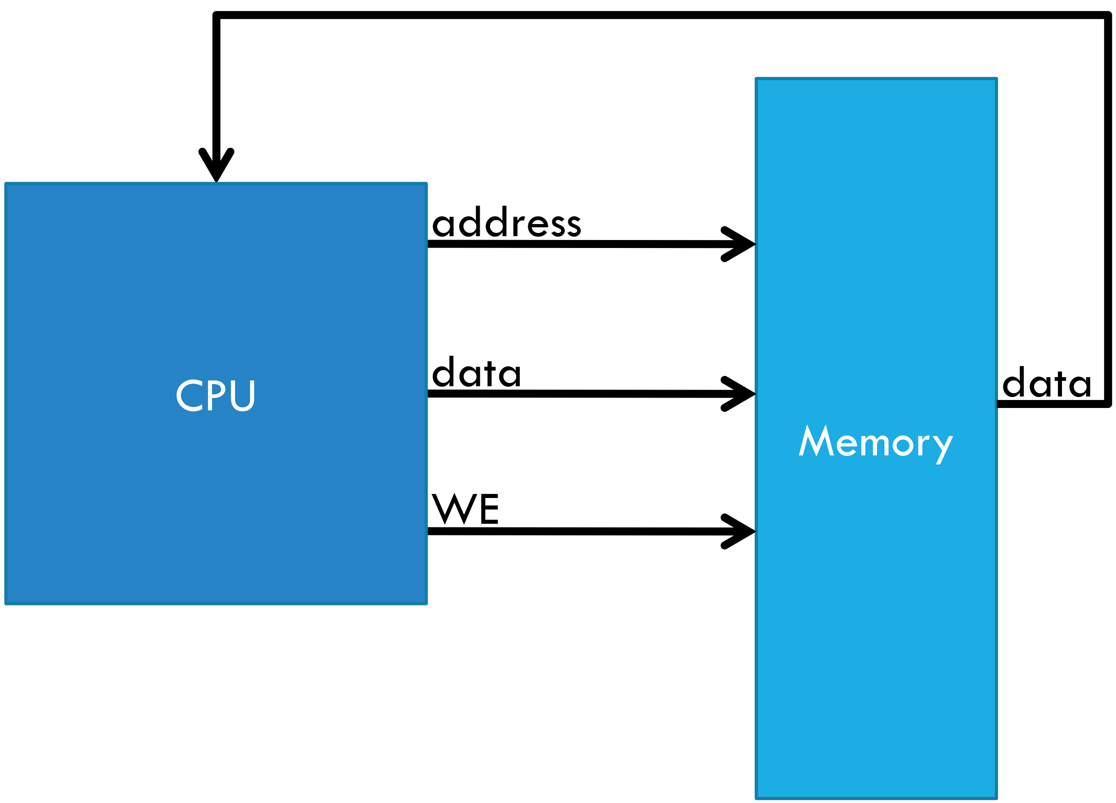
BYTE/ WORD ADDRESSABLE

___-addressable means a unique address associated with _____ amt of data

0	Byte
1	Byte
2	Byte
3	Byte
4	Byte
5	Byte
	⋮
0xFFFFFFFE	Byte
0xFFFFFFFF	Byte

0	Word
1	Word
2	Word
3	Word
4	Word
5	Word
	⋮
0xFFFFFFFE	Word
0xFFFFFFFF	Word

INTERFACE WITH THE CPU



Addresses generated by CPU

- When?

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

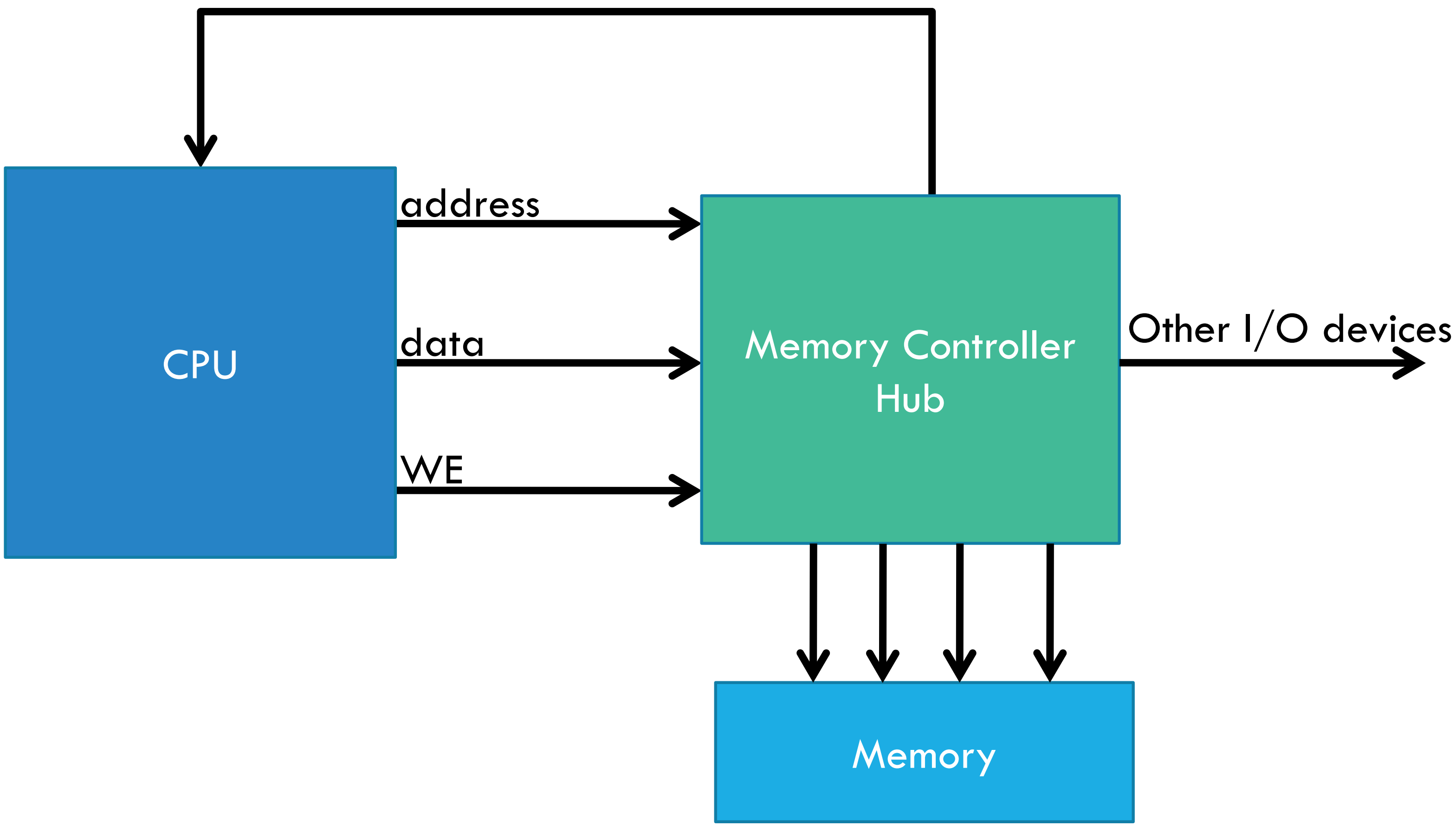
ADDRESS SPACES

A range of numbers (that's it?!)

Limited only by address width

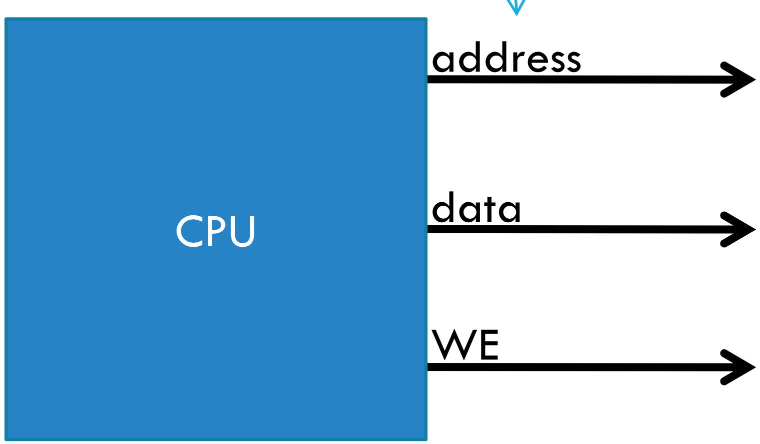
- 32 bits can generate addresses between 0 to 2^{32}-1
- If memory is byte-addressable, what is the largest memory size possible for 32 bits machine?

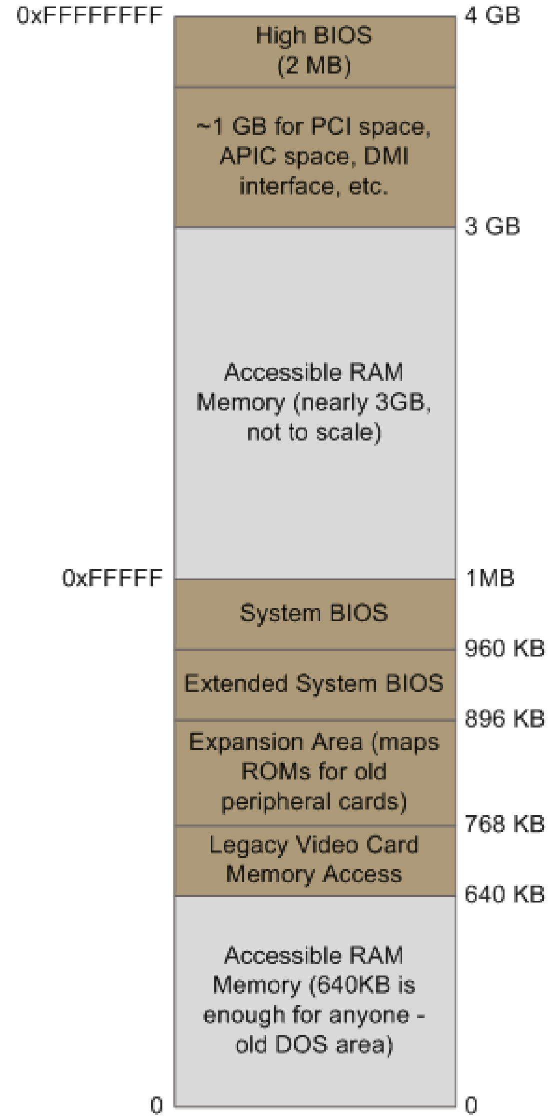
PHYSICAL SETUP



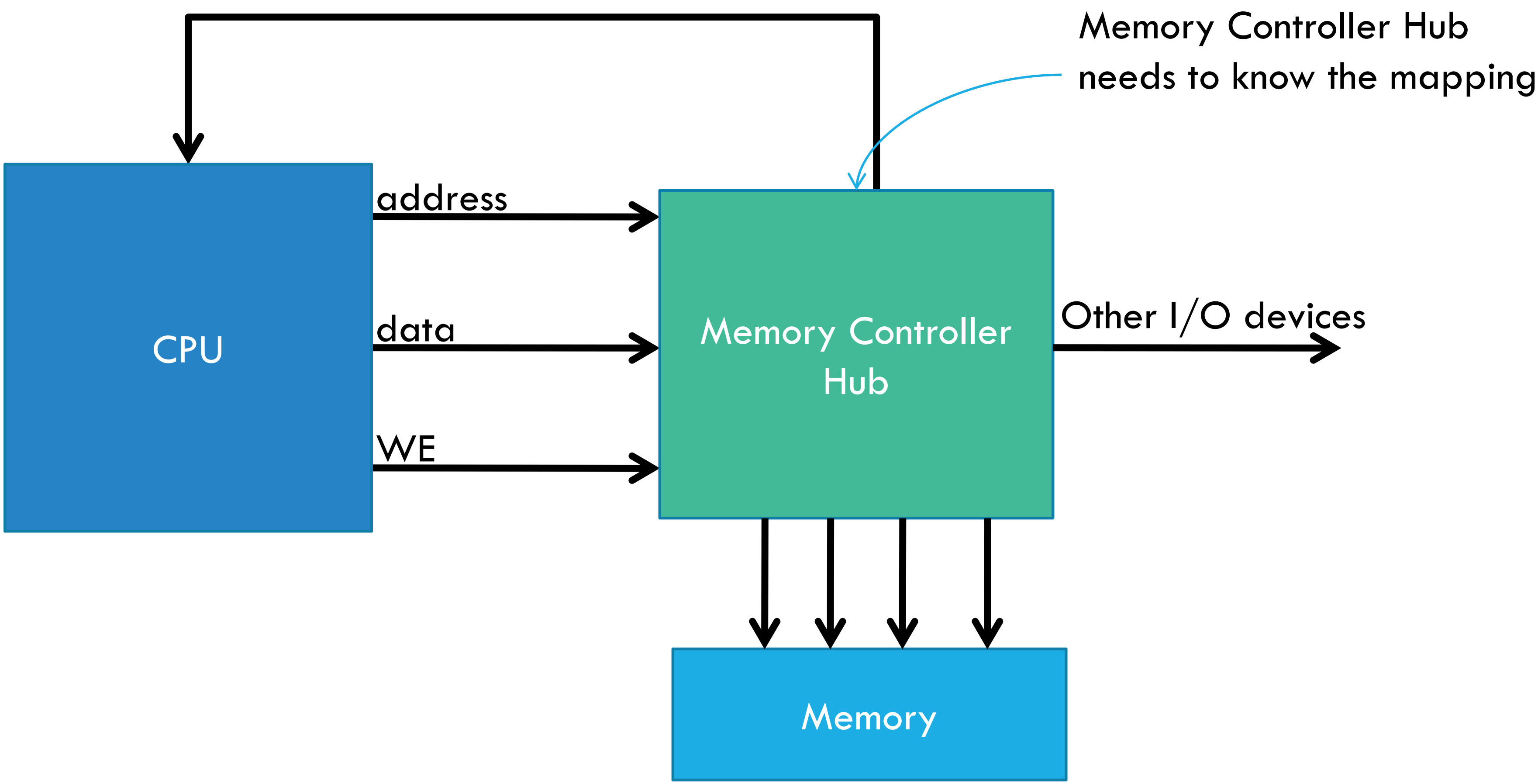
PHYSICAL ADDRESS SPACE (MAPPING)

All addresses issued by the CPU outwards are physical address!





PHYSICAL SETUP – II



Logical Address Spaces

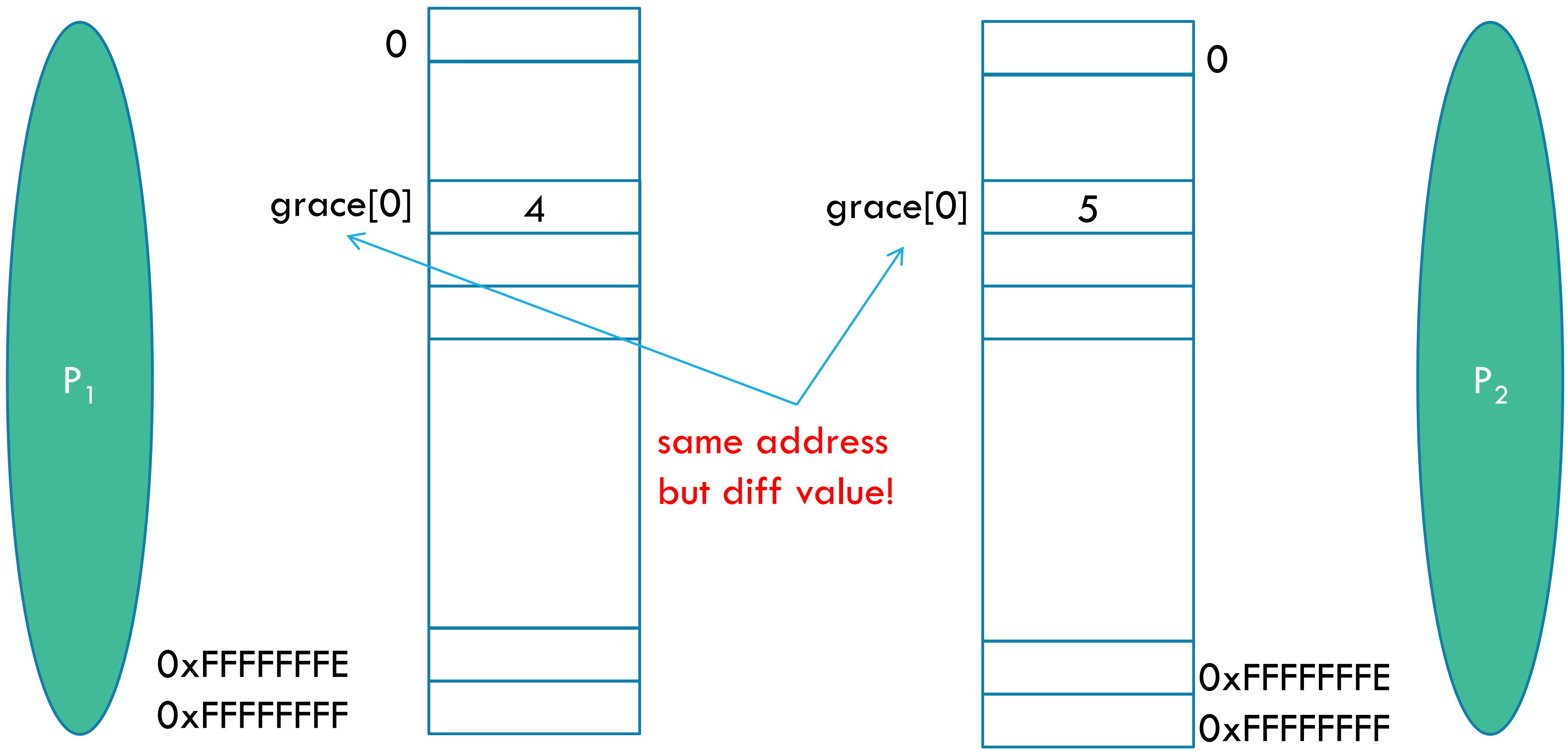
RUN TWO PROCESSES OF THIS CODE. WILL THE PRINTOUT BE THE SAME?

```
#include <stdio.h>
int grace[1000];
int main()
{
   char game[200];
   printf("address of global variable: %p\n", grace);
   printf("address of local variable: %p\n", game);
   printf("address of main function: %p\n", main);
}
```

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

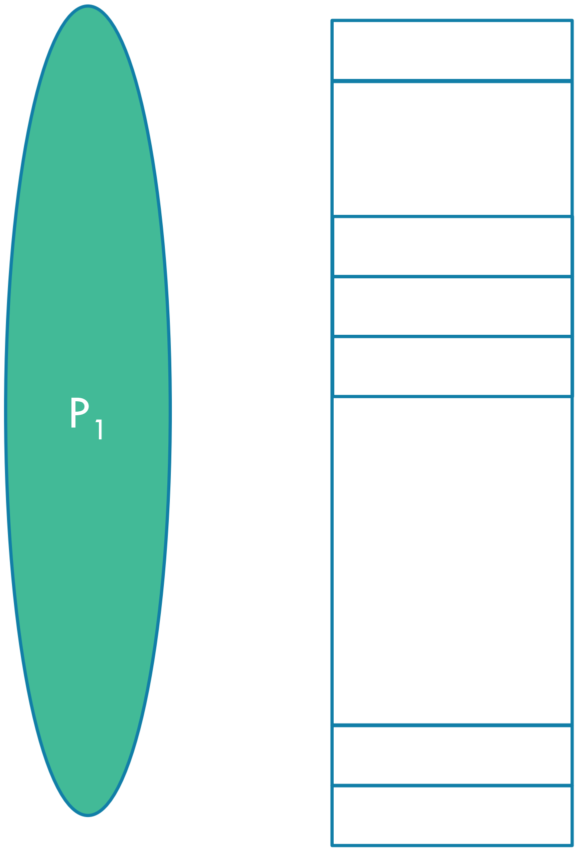
```
#include <stdio.h>
#include <stdlib.h>
int grace[1000];
int main(int argc, char**argv)
{
   char game[200];
   grace[0] = atoi(argv[1]);
   printf("%p, %d\n", &grace[0], grace[0]);
}
```

ADDRESS SPACES OF PROCESSES

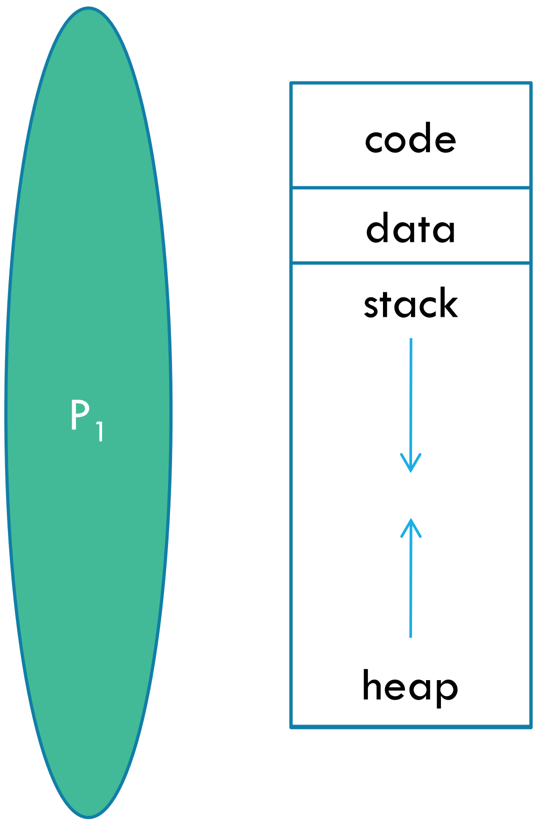


LOGICAL ADDRESS SPACES

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

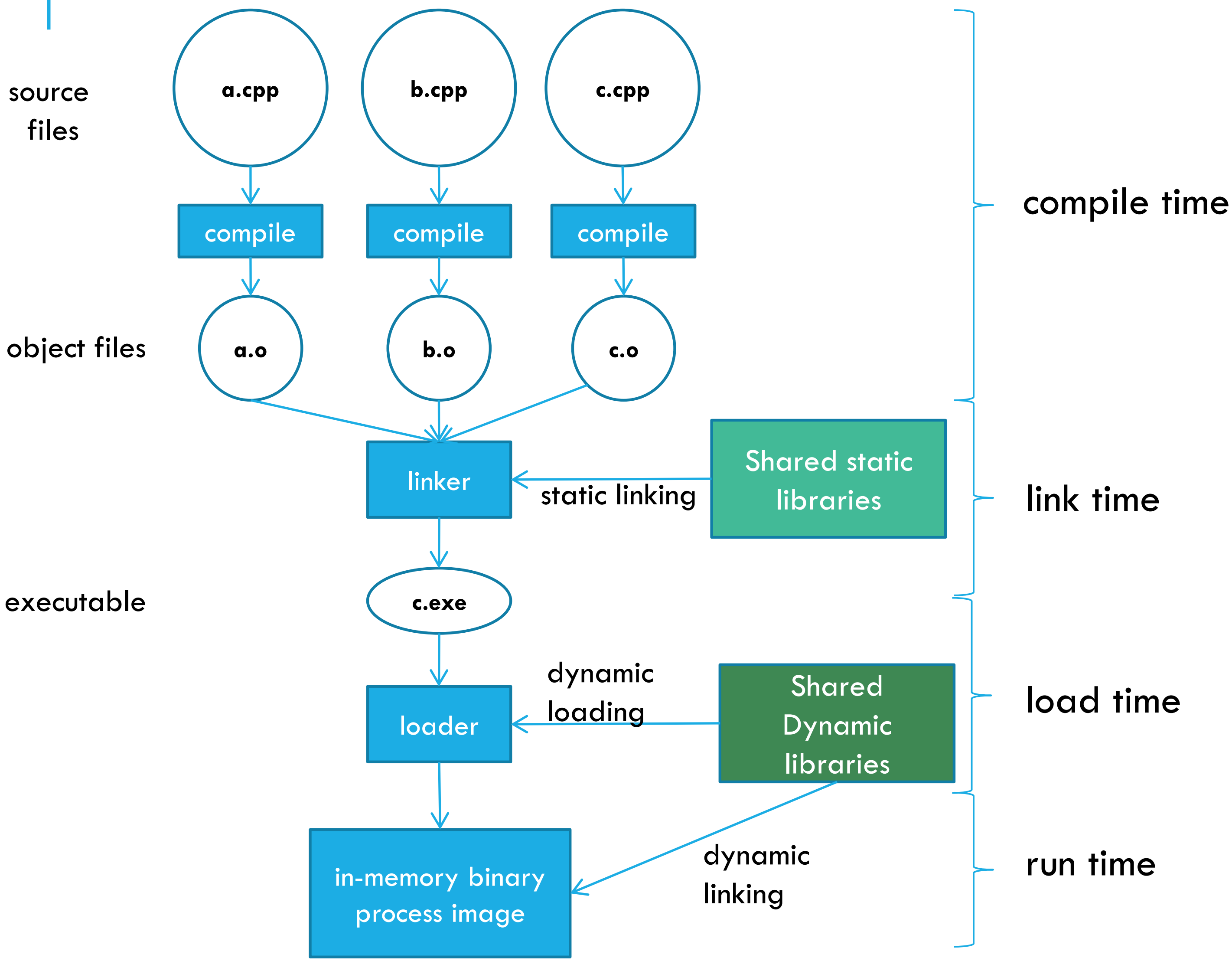


PROCESS'S LOGICAL ADDRESS SPACE



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

PROCESSING OF PROGRAM



Binding Logical Address Space to Physical Address Spaces

COMPILE/LINK-TIME BINDING

- Logical addresses=physical addresses
- Pros/Cons?

LOAD-TIME BINDING

- Addresses are relative to some base address in physical memory
  - (physical address) = (base) + (logical address)
- Programs can be loaded anywhere in physical memory
- Program can only be loaded if there is a contiguous block of free memory available large enough to hold the program and data

EXECUTION-TIME BINDING

- The physical address is computed in hardware at runtime by the *memory management unit* *(MMU)*
  - (physical address) <- (logical 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 contiguous physical memory
  - Used by most modern OS's

MODERN SOLUTION: SHARED LIBRARIES

Static libraries have the following disadvantages:

- Duplication in the stored executables (every function needs libc)
- Duplication in the running executables
- 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 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.

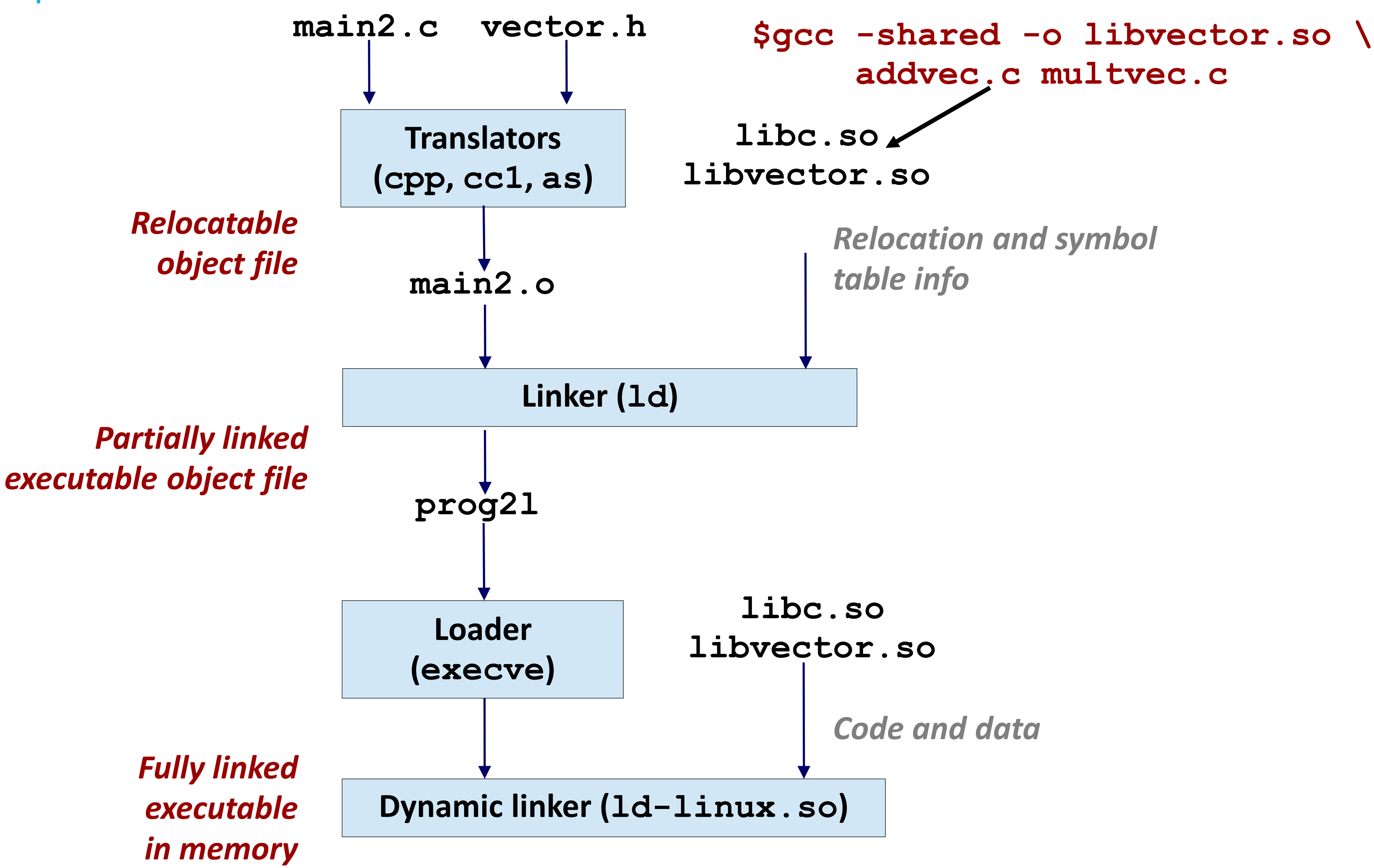
Dynamic linking can also occur after program has begun (run-time linking).

- 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



DYNAMIC LINKING AT RUN-TIME (1)

```
#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);
    }
```

dll.c

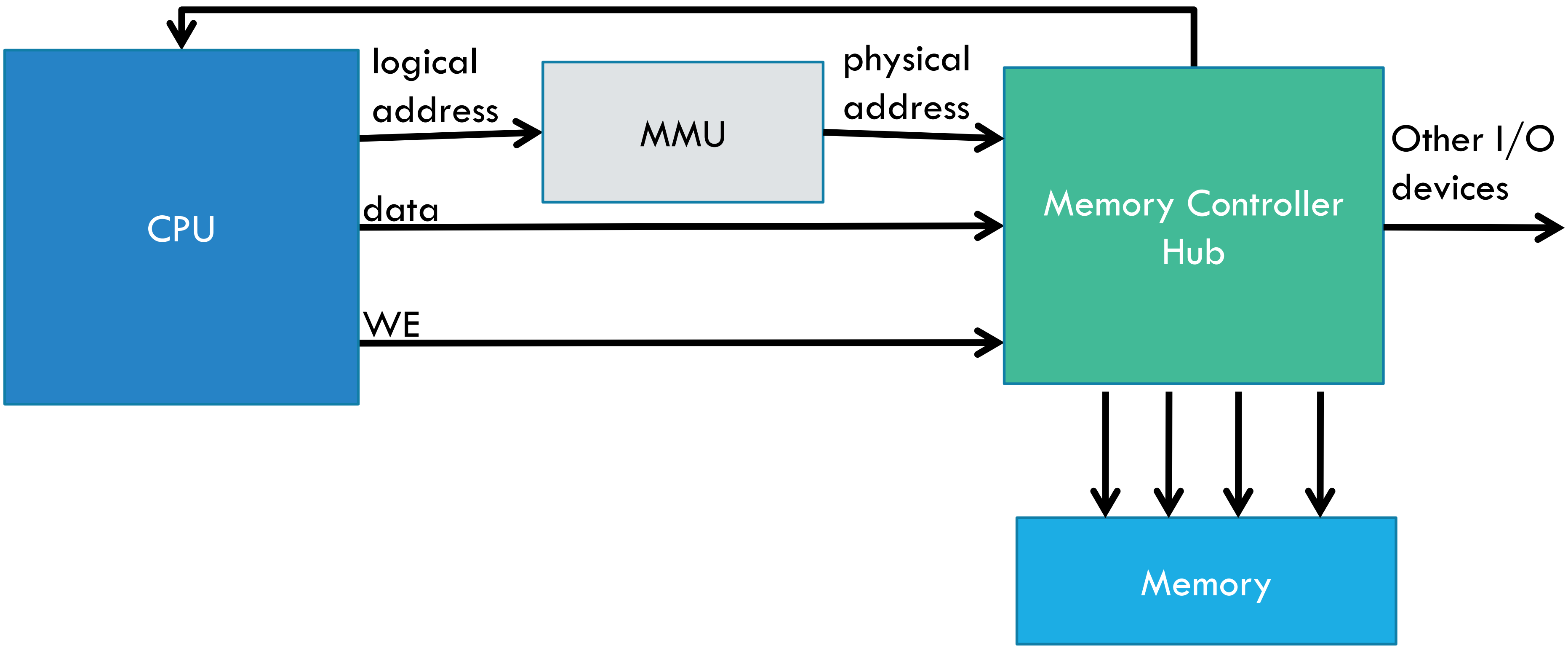
DYNAMIC LINKING AT RUN-TIME (2)

```
    ...

/* 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

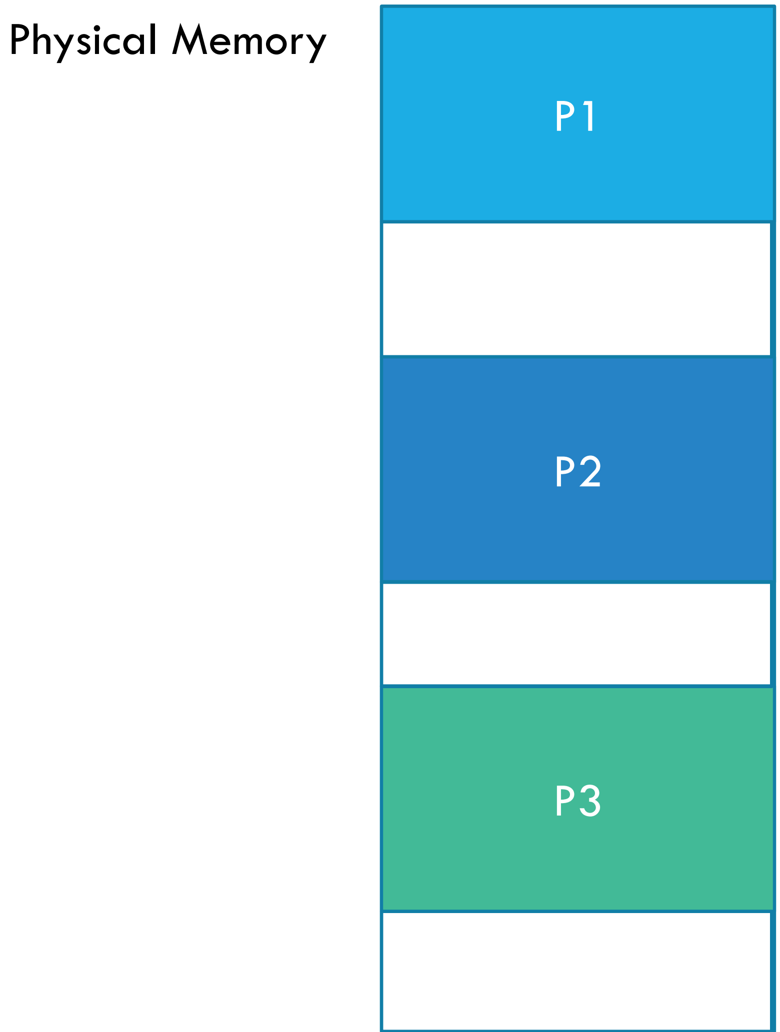
Binding Logical Address Space to Physical Address Spaces

MMU

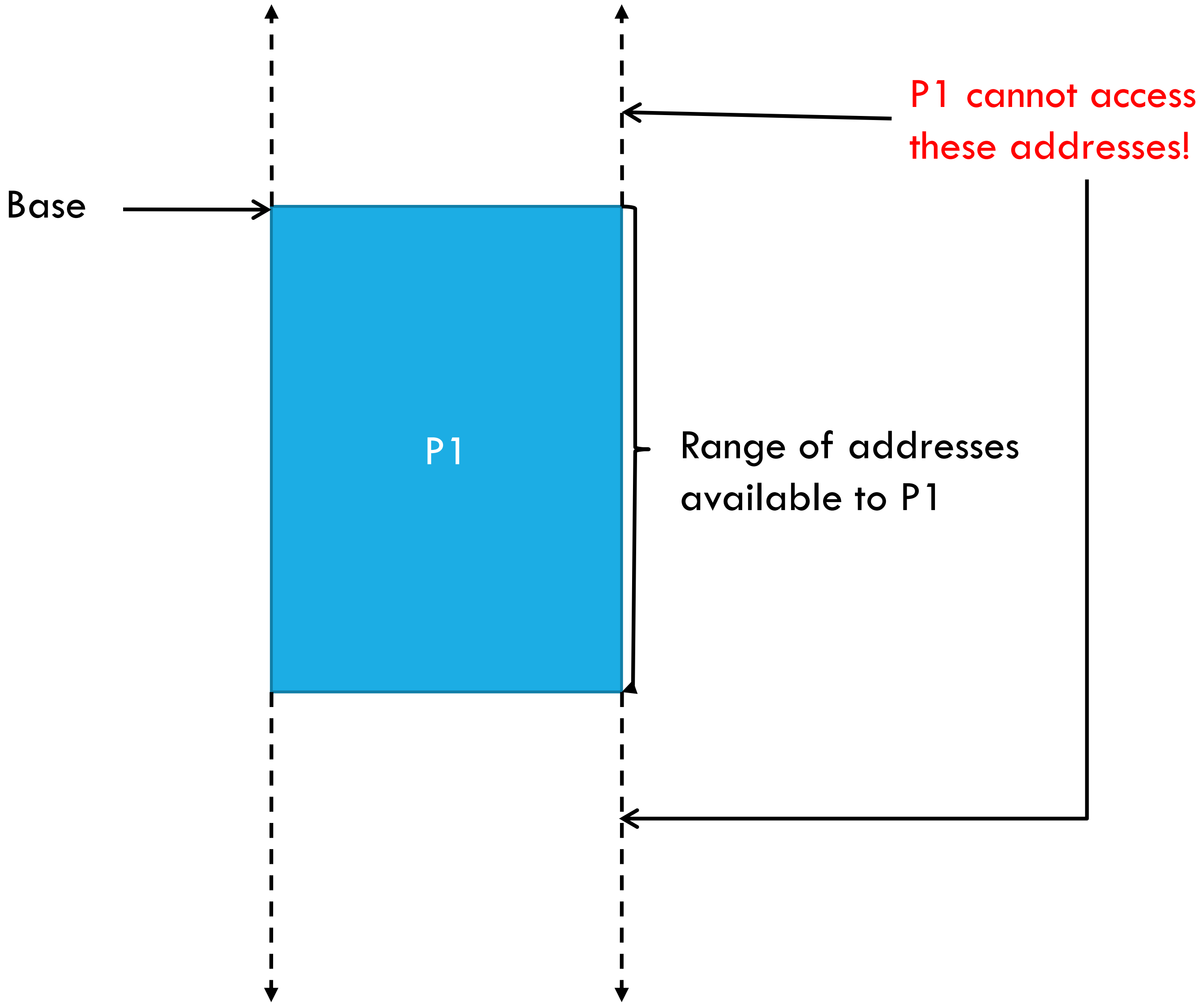


CPU
logical
address
MMU
physical
address
data
WE
Memory Controller
Hub
Other I/O
devices
Memory

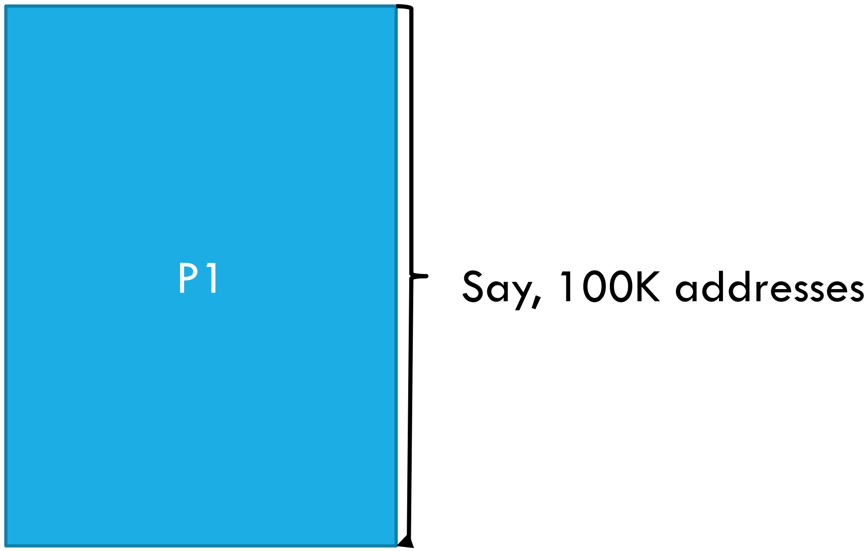
NAÏVE IDEA: EACH PROCESS GETS A PIECE OF PHYSICAL MEMORY



BASE AND LIMIT



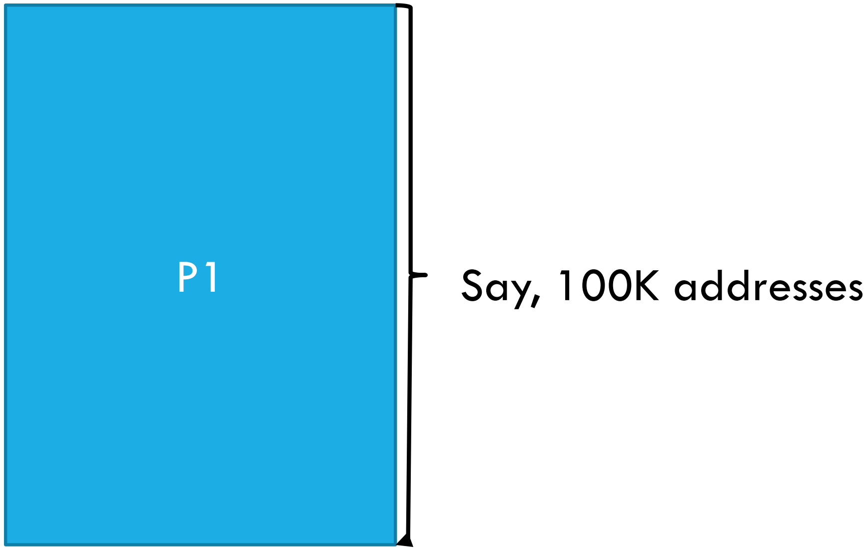
P1'S LOGICAL ADDRESS SPACE



P1's logical address space will range from 0 to (100K -1).

LOGICAL ADDRESS SPACE AND CODE

```
.
.
.
*(pointer) = 100;
.
.
.
.
```



Address contained by `pointer` SHOULD NOT exceed (100K-1)!

TRANSLATING LOGICAL ADDRESS TO PHYSICAL ADDRESS

`*(pointer) = 100;` C code

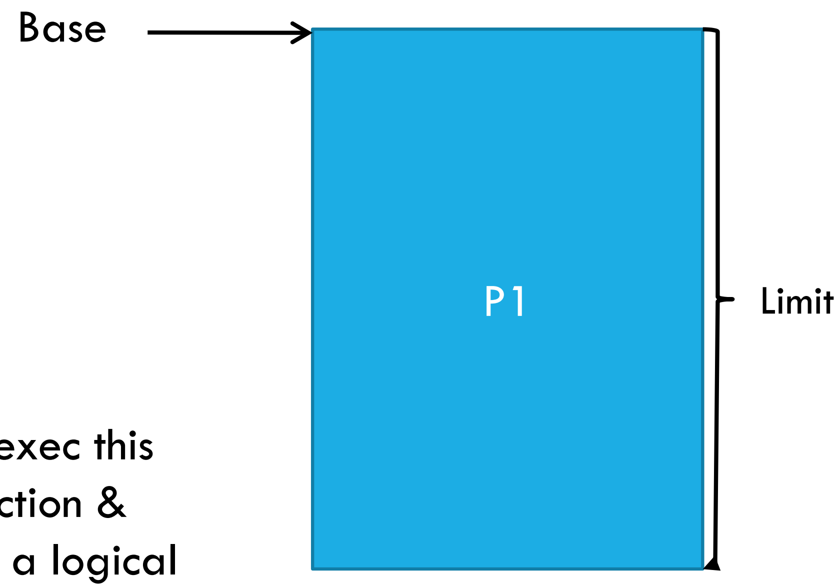
In assembly:

`mov  Address, 100`

During run time:

`mov` $\boxed{\texttt{500,}}$ `100`

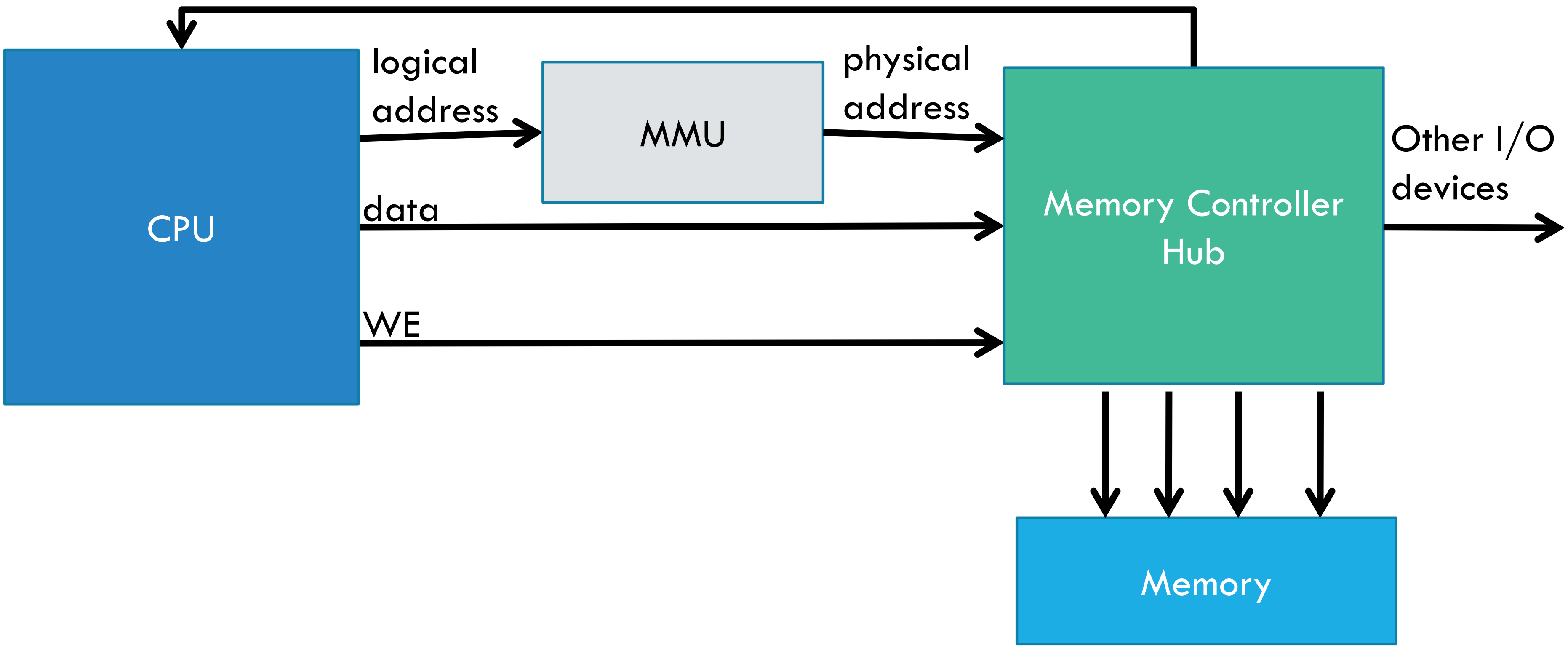
CPU exec this instruction & this is a logical address!



Translate by doing this in HW!

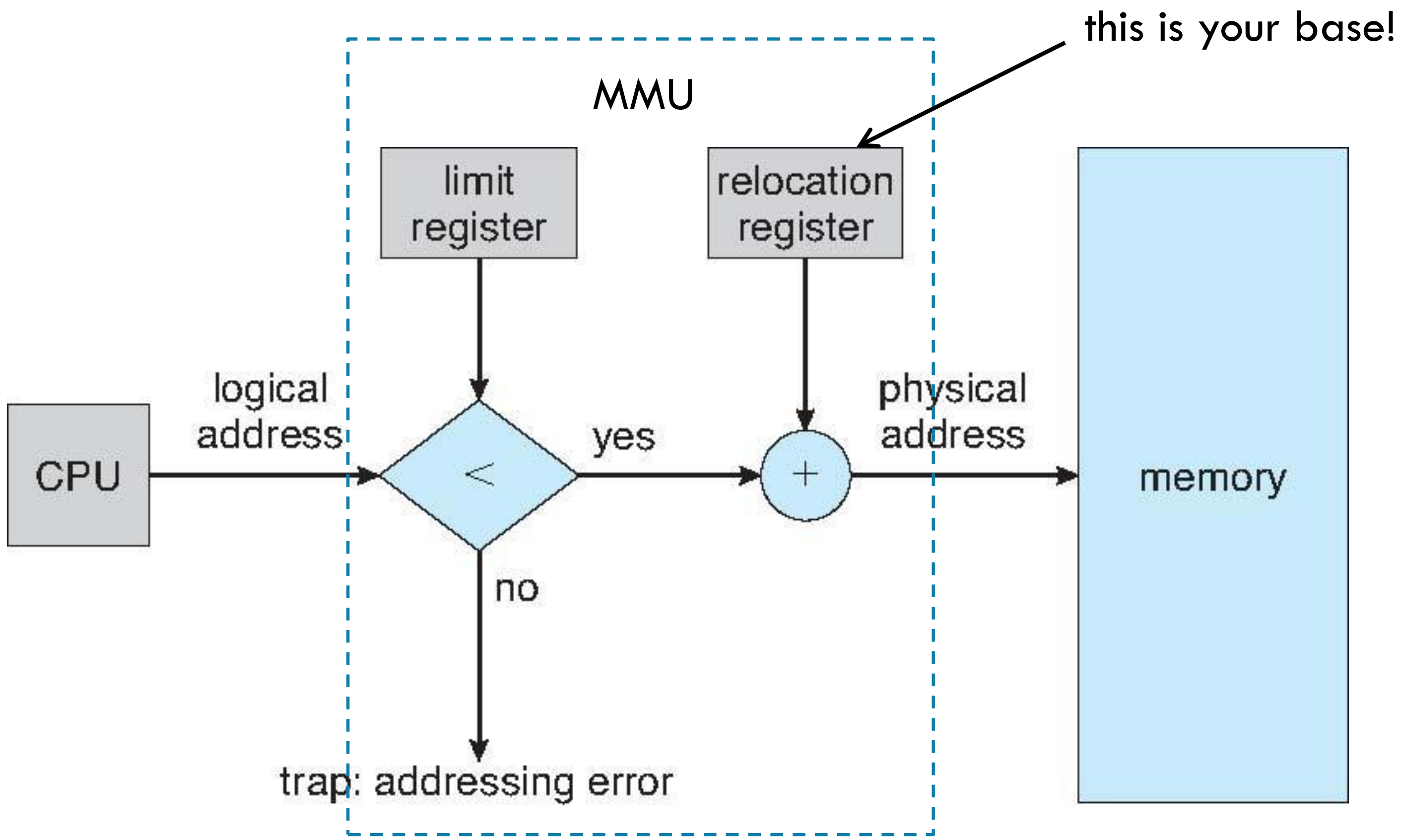
```
if (500 < limit)
    Physical Address = Base + 500
else
    cause a fault
```

MMU



CPU
logical
address
MMU
physical
address
data
WE
Memory Controller
Hub
Other I/O
devices
Memory

SIMPLE CONTIGUOUS MEMORY ALLOCATION



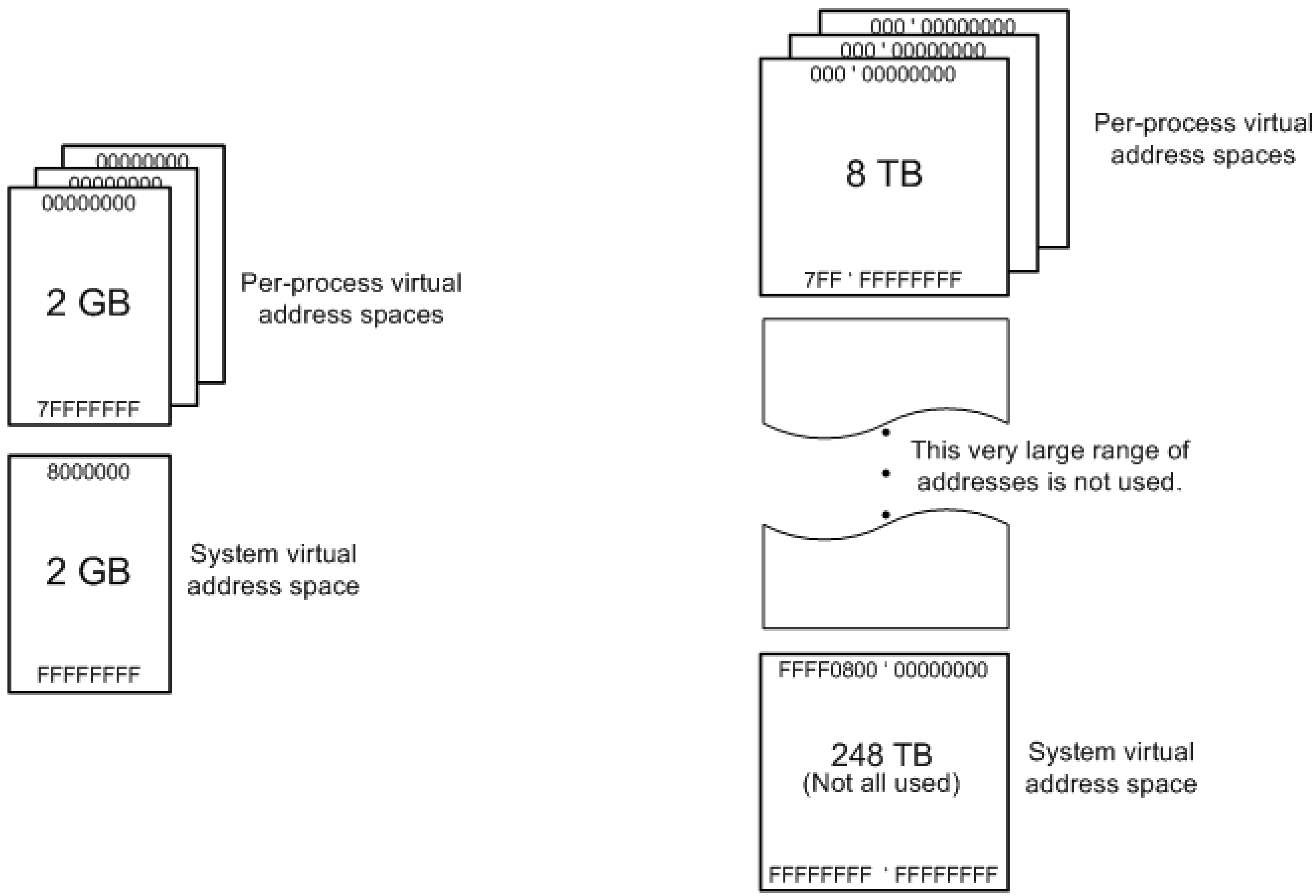
KERNEL ADDRESS SPACE

- 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

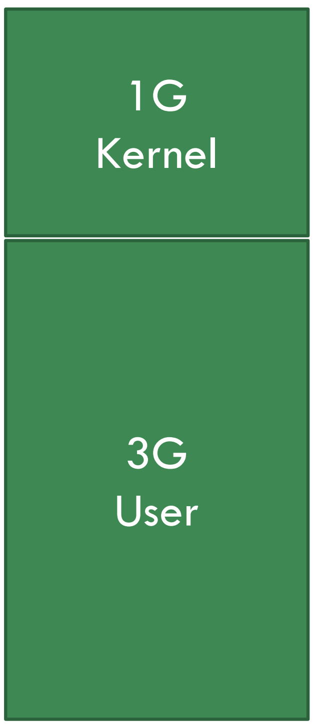
- Whose code is running in kernel mode?
  - OS
- Does the OS use logical address space?
  - OS “turns off” translation
- How does OS turn off the translation?
  - Setting Base to 0 and Limit to maximum memory size.
- How about processes?
  - They can only access logical address space in user mode.

USER AND KERNEL SPACE MODEL (WINDOWS)

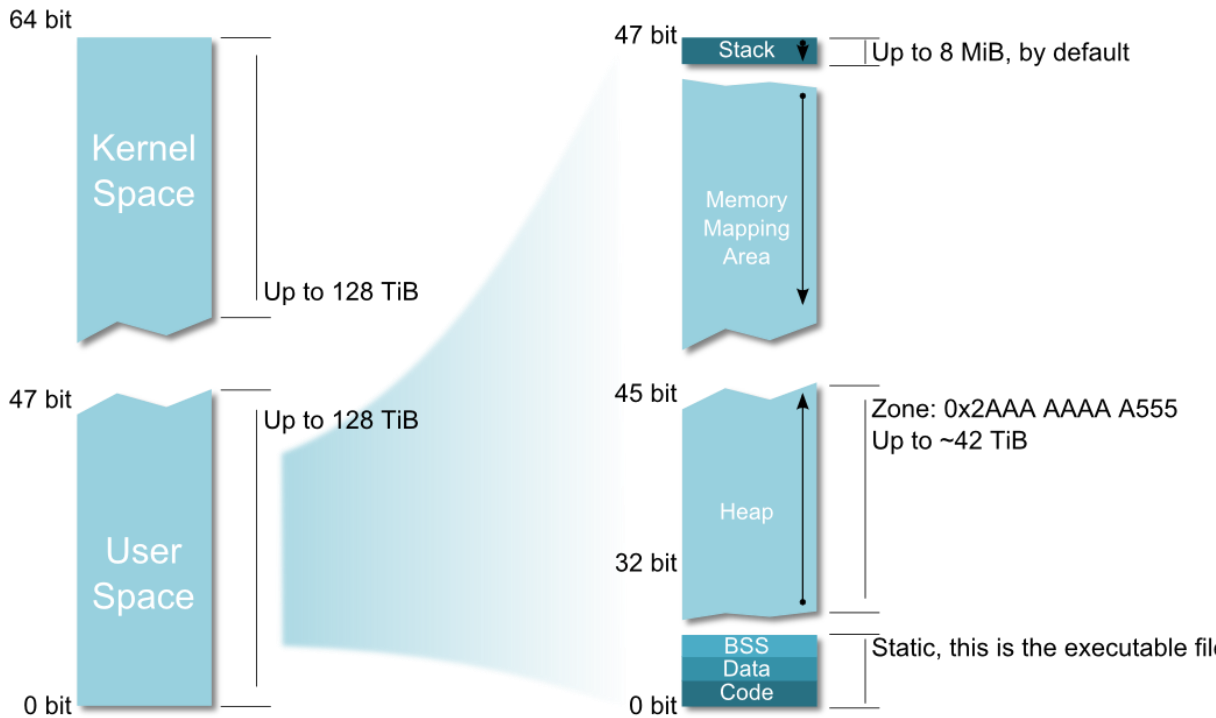


USER AND KERNEL SPACE MODEL (LINUX)

x86



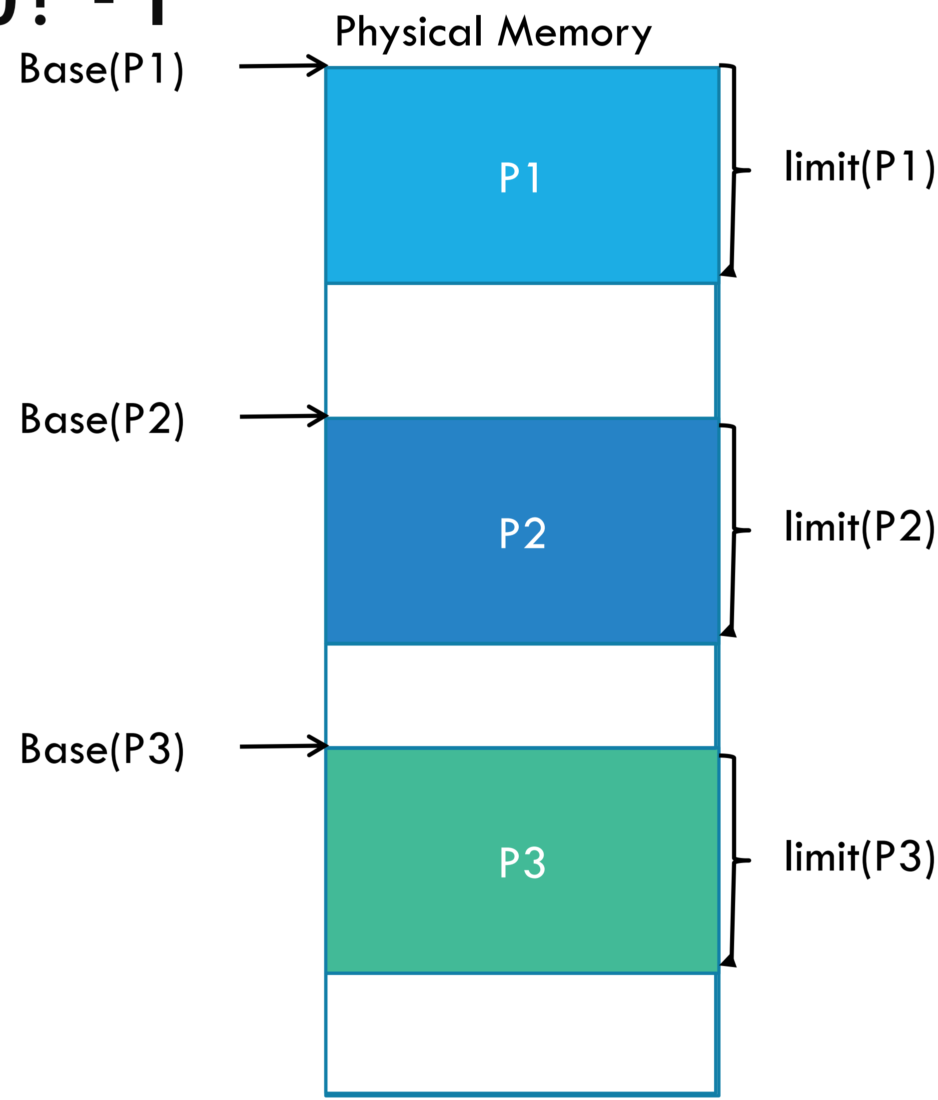
x64



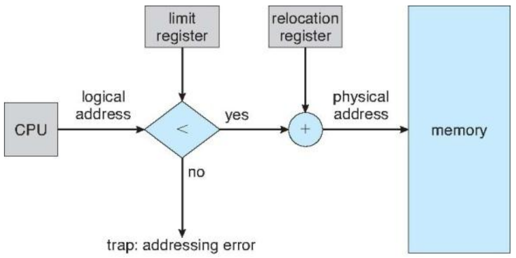
So Kernel + User Spaces add for 256 TiB which is a tiny part of the 16 777 216 TiB addressable over 64 bit!

WHAT HAPPENS IN CONTEXT SWITCHING? - I

1. Each process has a base and limit.
2. Stored in the PCB.



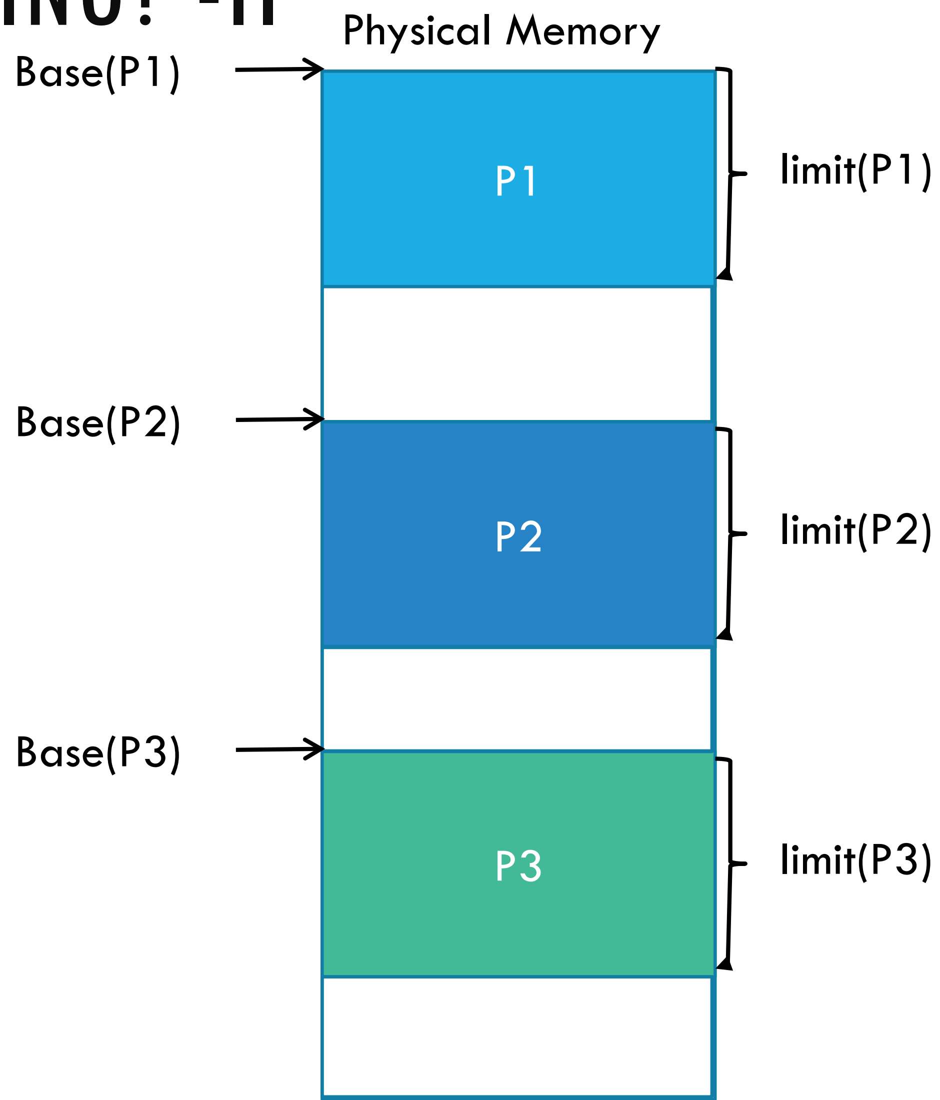
WHAT HAPPENS IN CONTEXT SWITCHING? -II



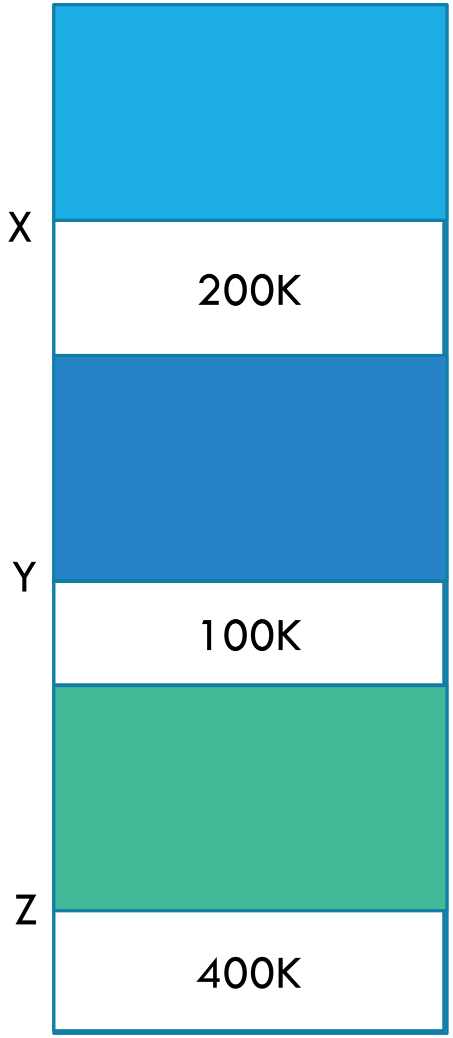
1. Suppose P1 is running now.
   Relocation Register = Base(P1)
   Limit Register = Limit(P1)

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

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

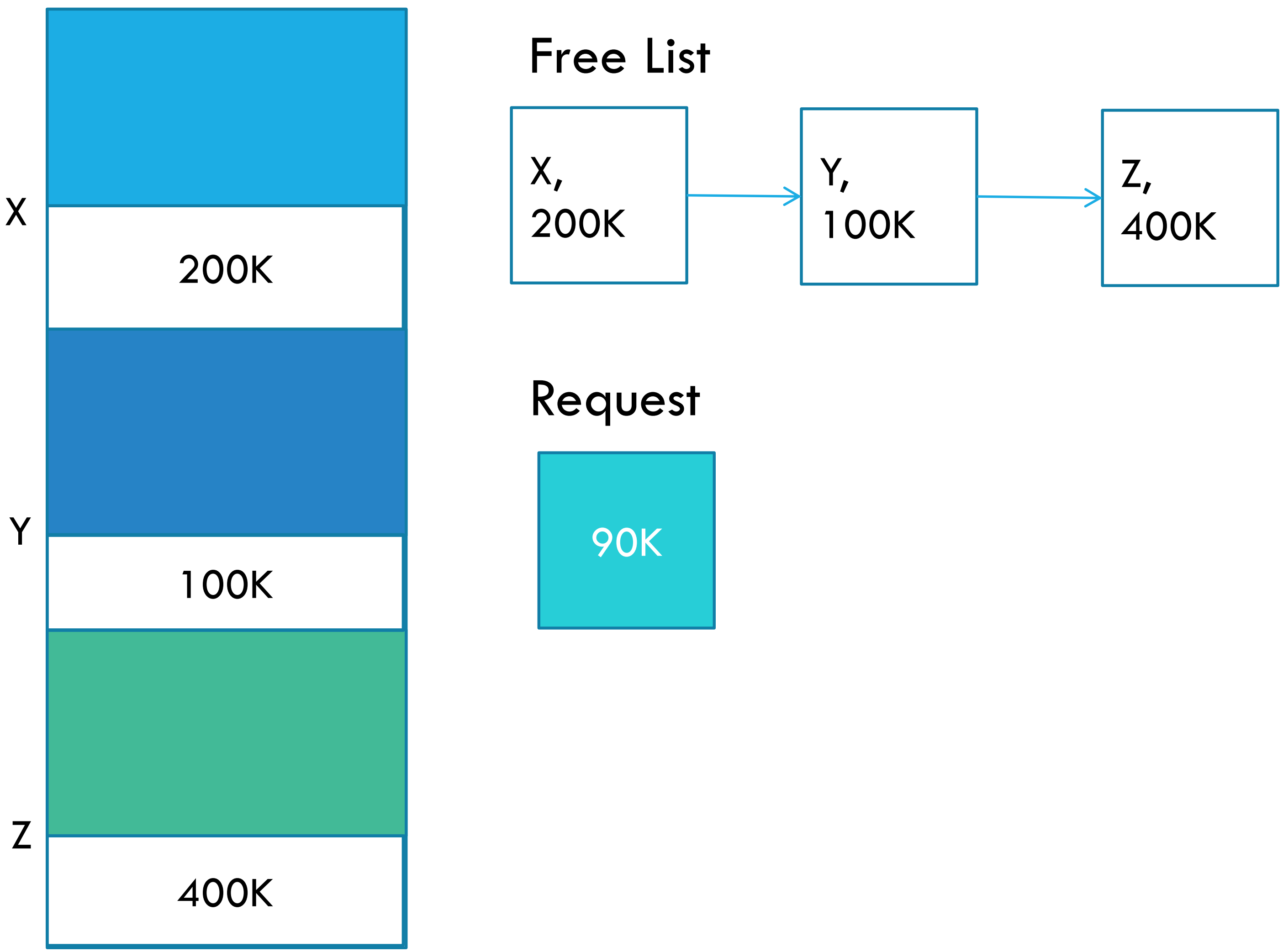


CONTIGUOUS MEMORY ALLOCATION

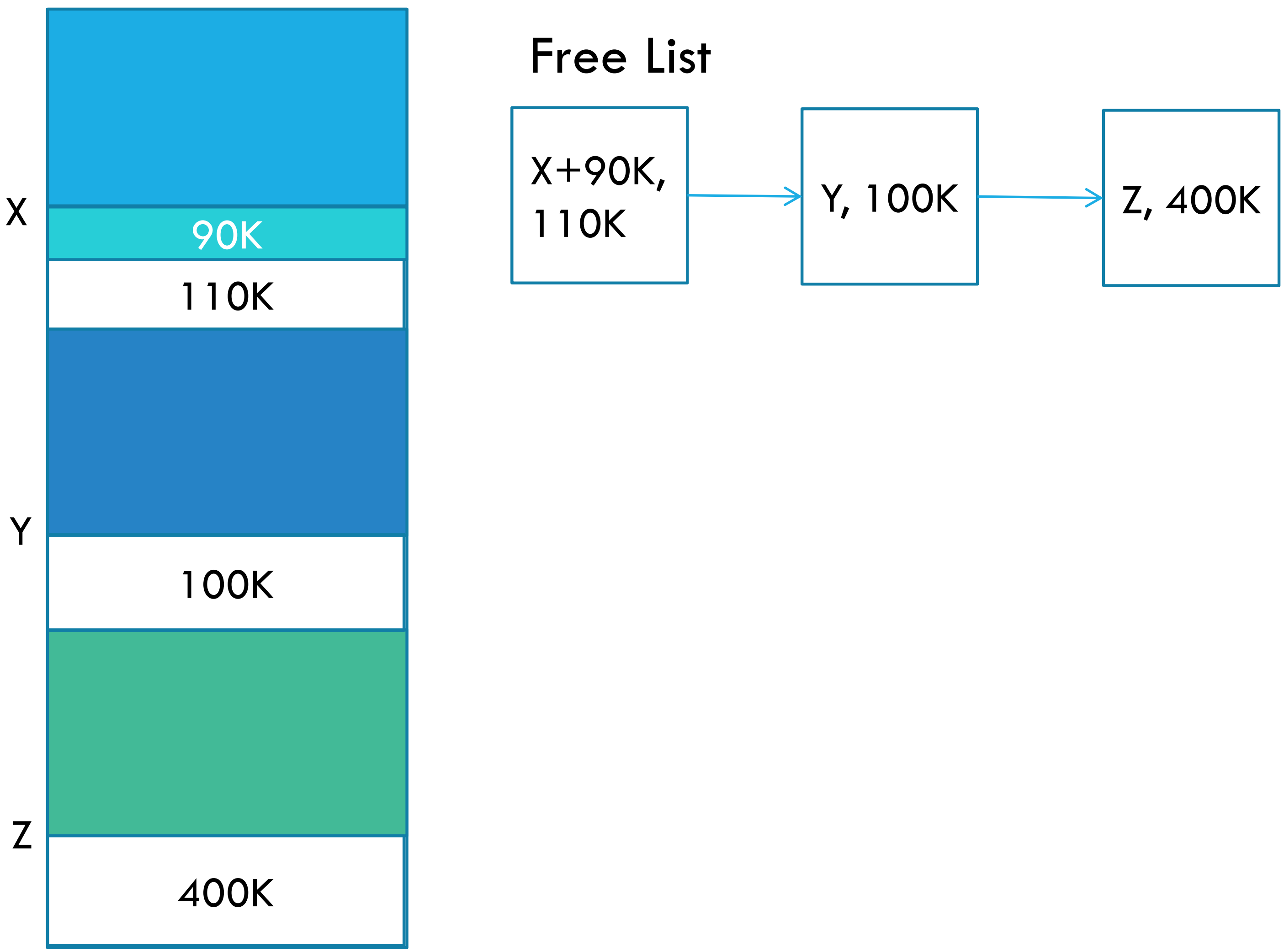


- Maintain a list of free blocks
- First fit, best fit, worst fit

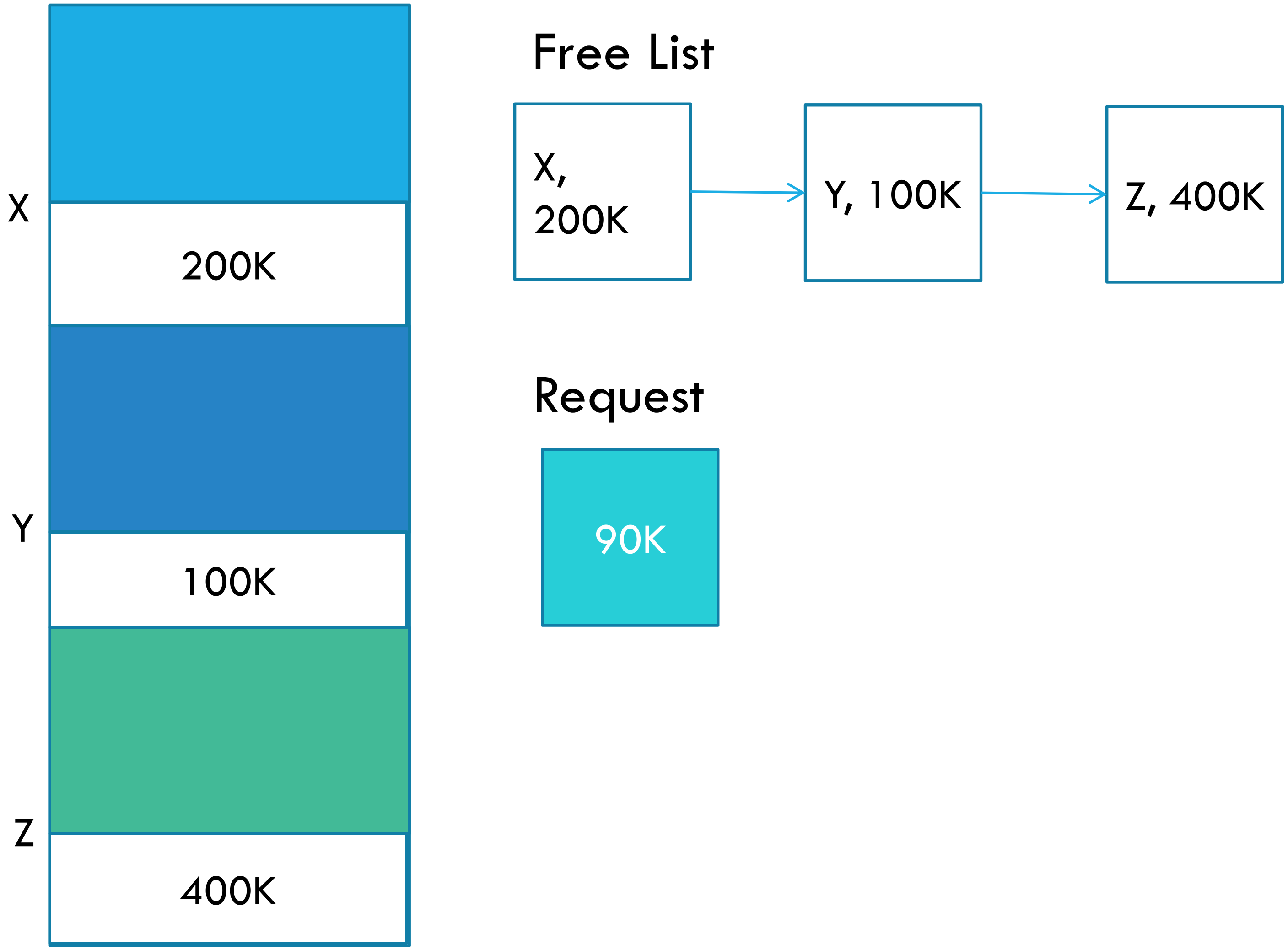
FIRST FIT



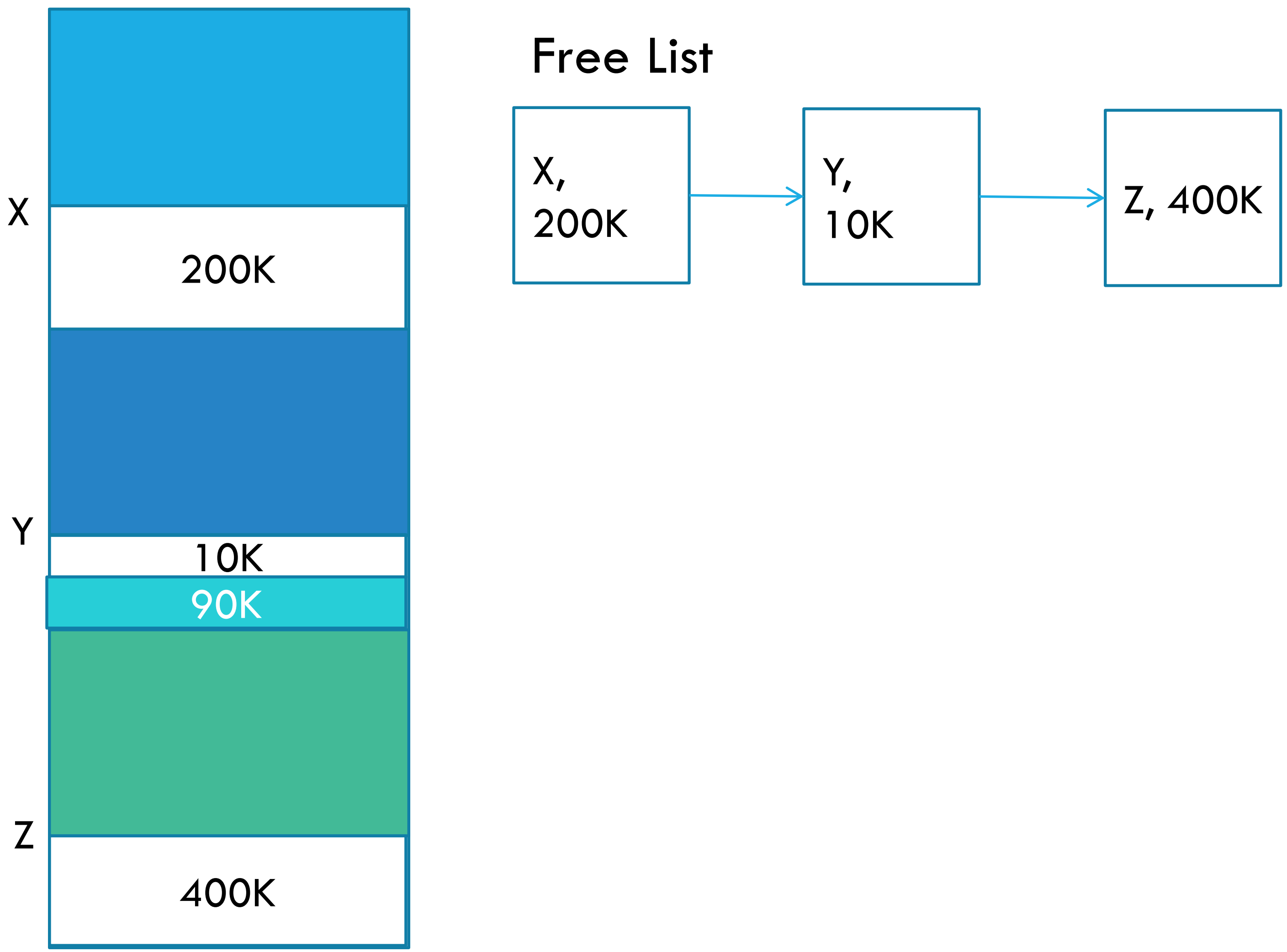
FIRST FIT



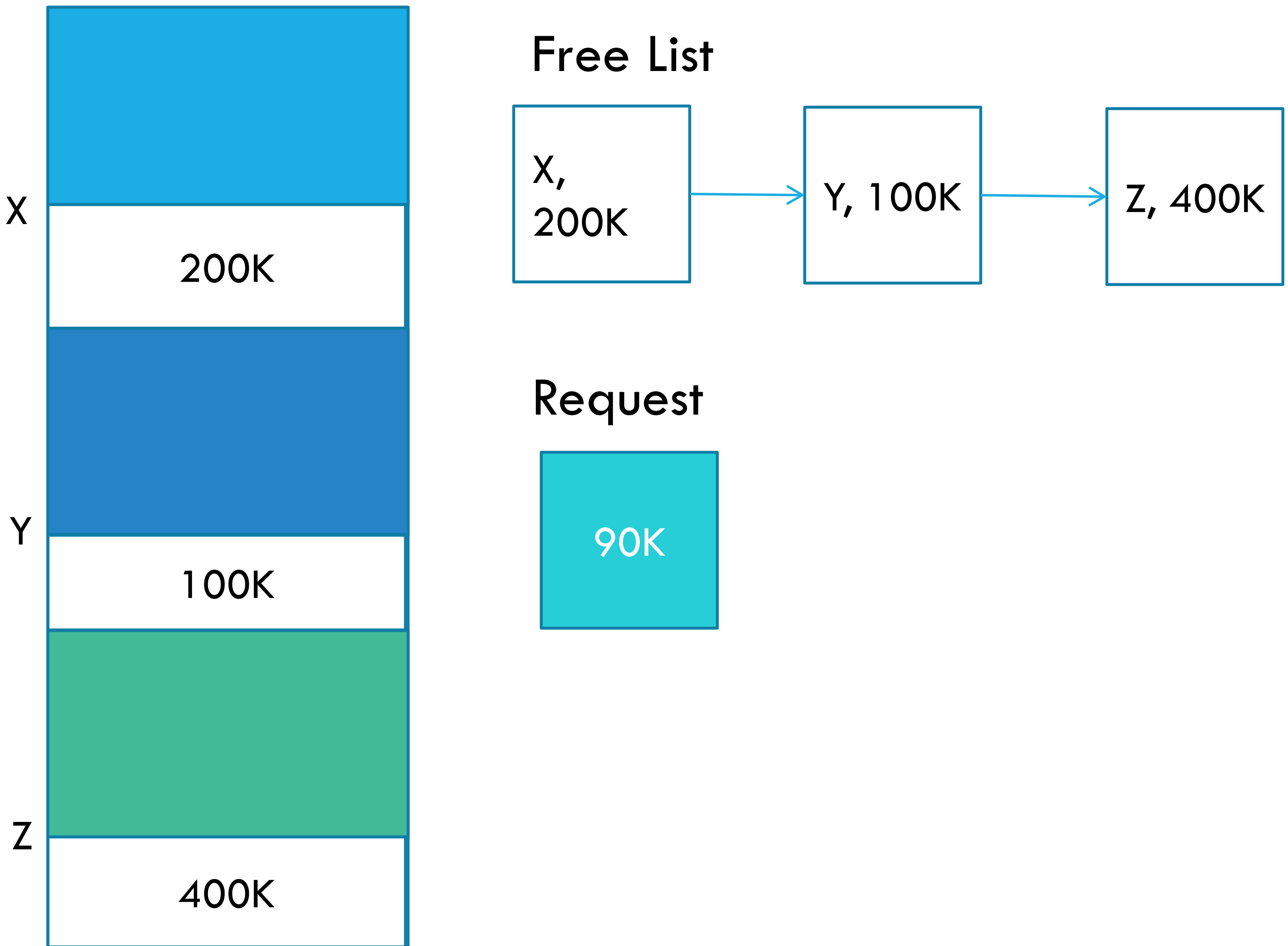
BEST FIT



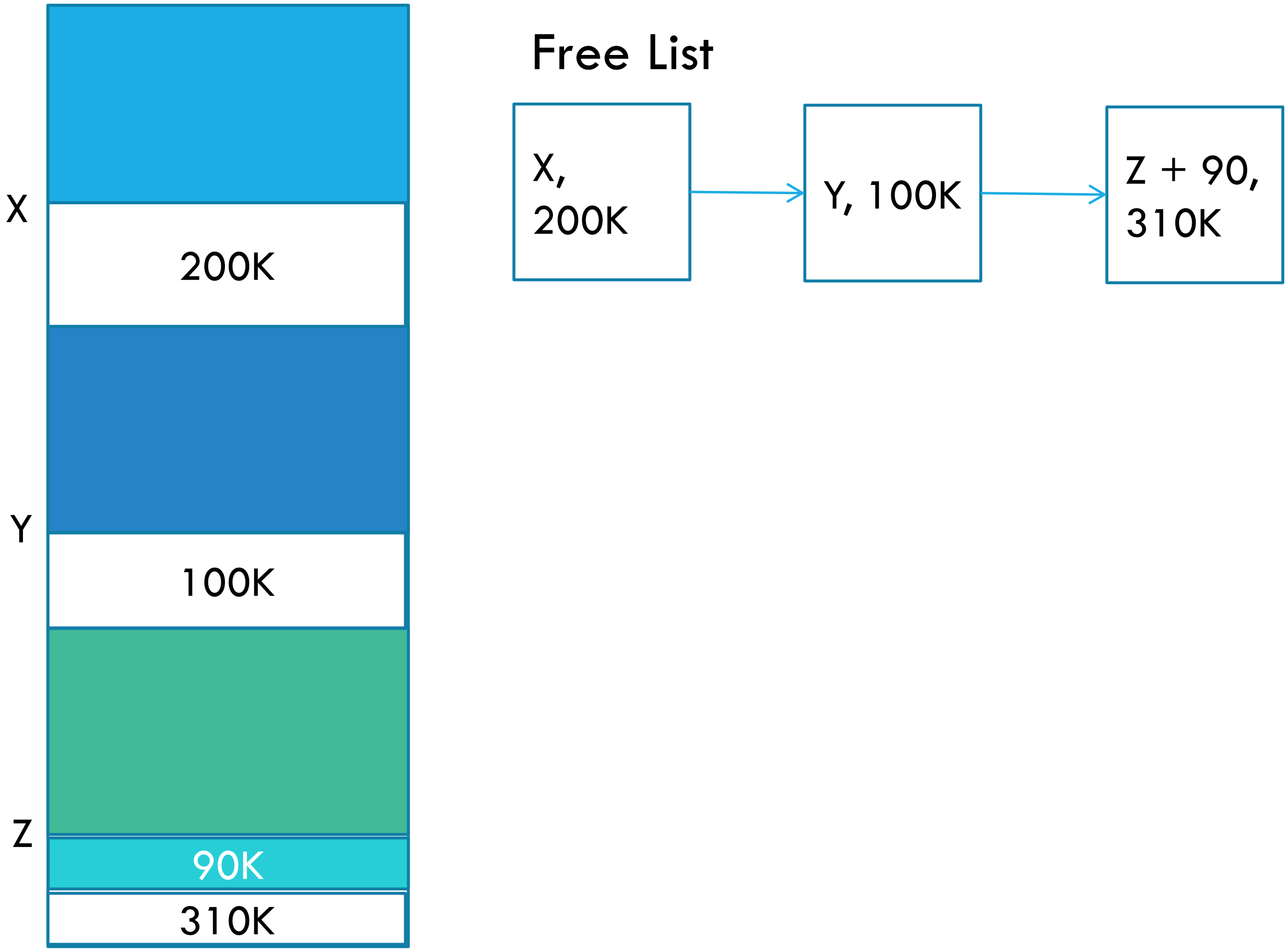
BEST FIT



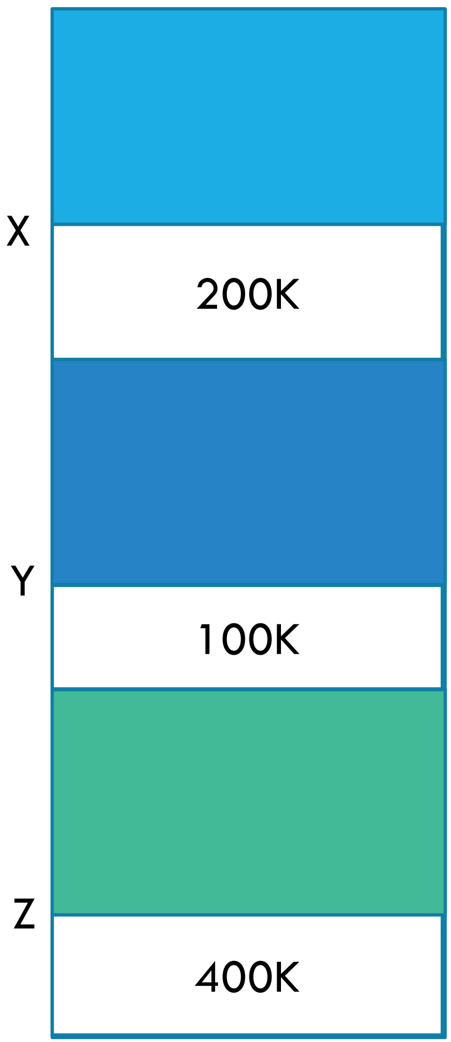
WORST FIT



WORST FIT



EXTERNAL FRAGMENTATION



Request

What to do?

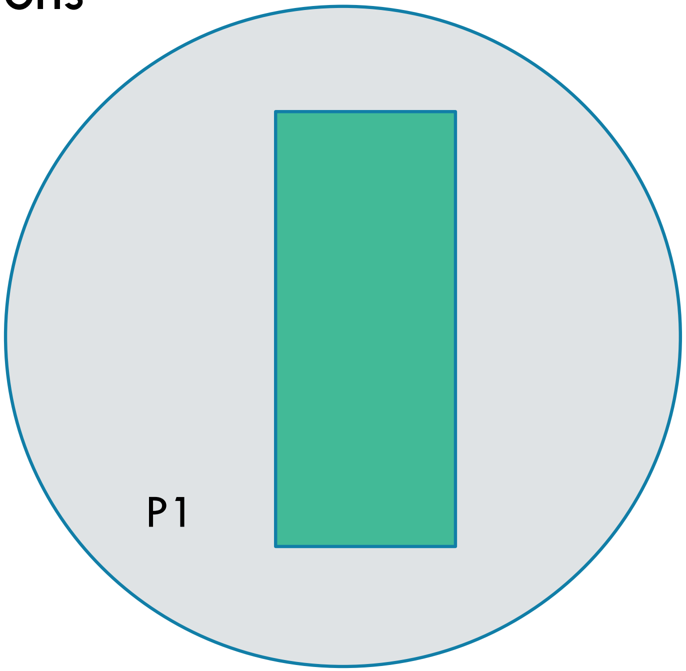
External Fragmentation:

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

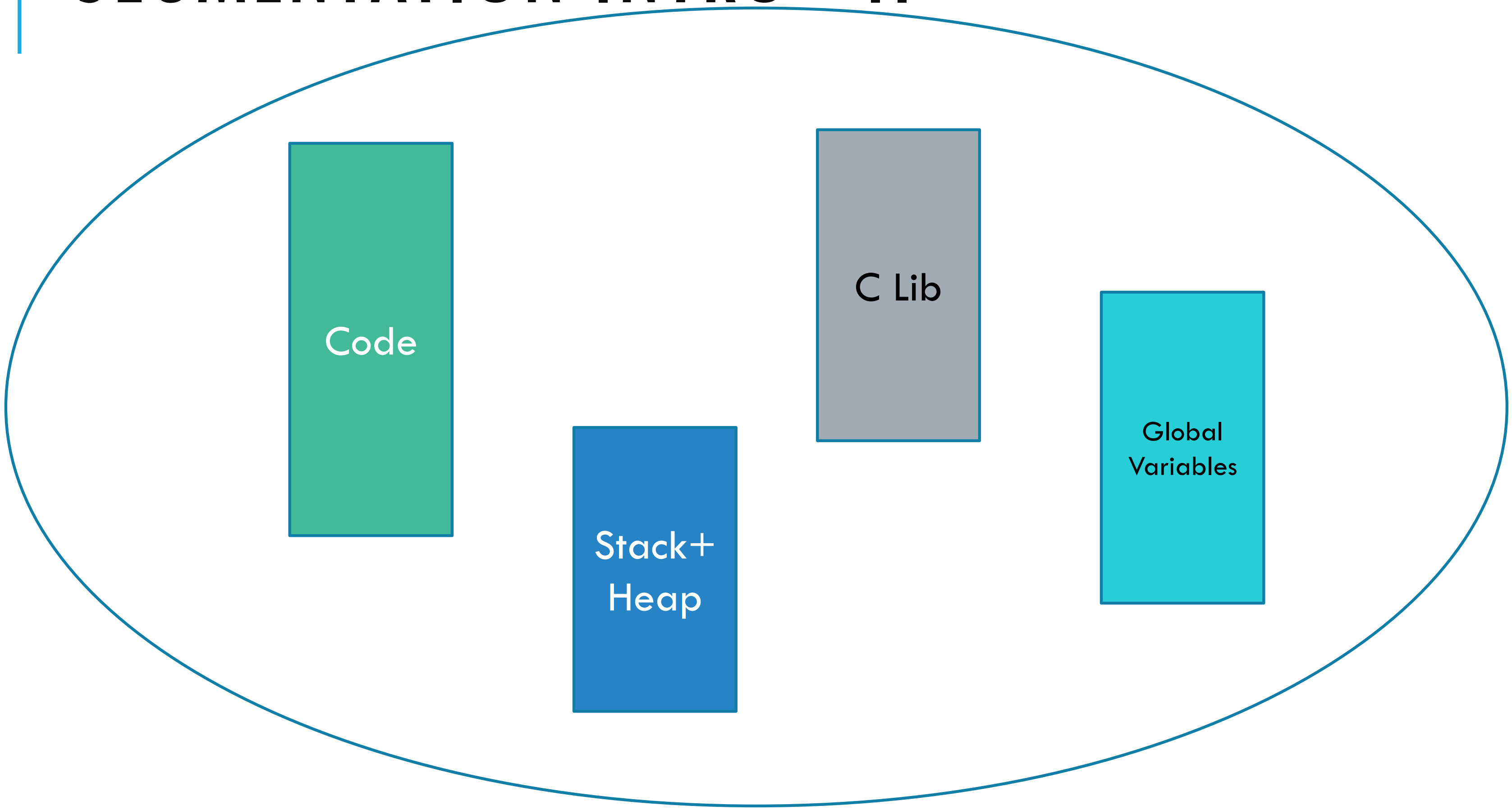
Proper definition:
External fragments of memory exists outside allocated regions that *may* be unallocatable for a specific request within the size of free memory.

SEGMENTATION INTRO - I

- Programmers think in terms of “regions”
  - Text
  - Data
  - Stack
  - Heap
  - Etc
- Not just 1 contiguous area

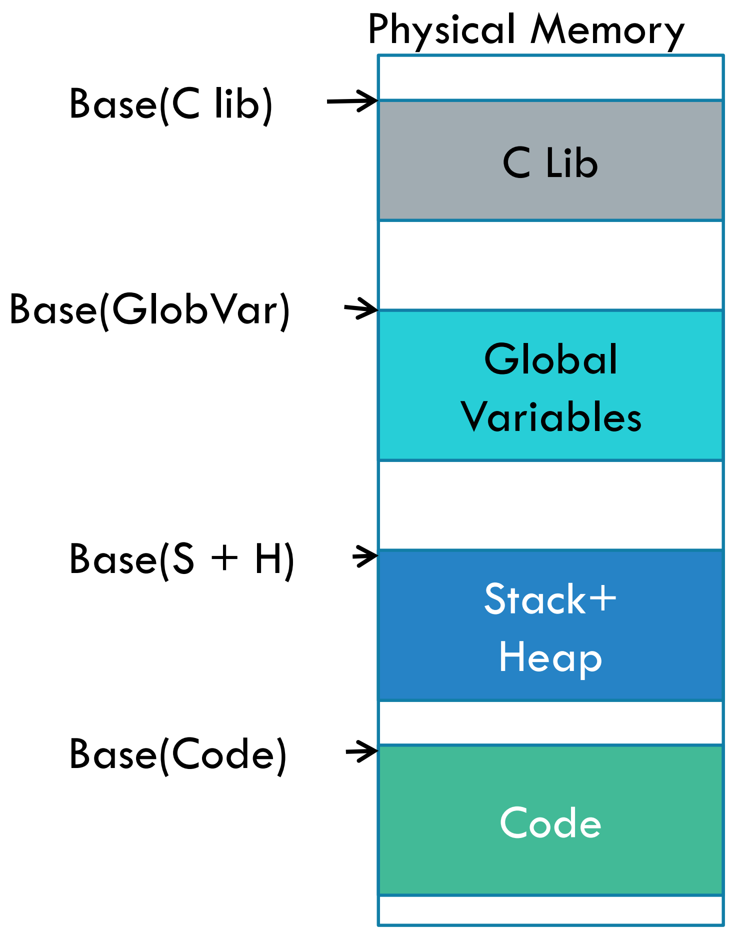


SEGMENTATION INTRO - II



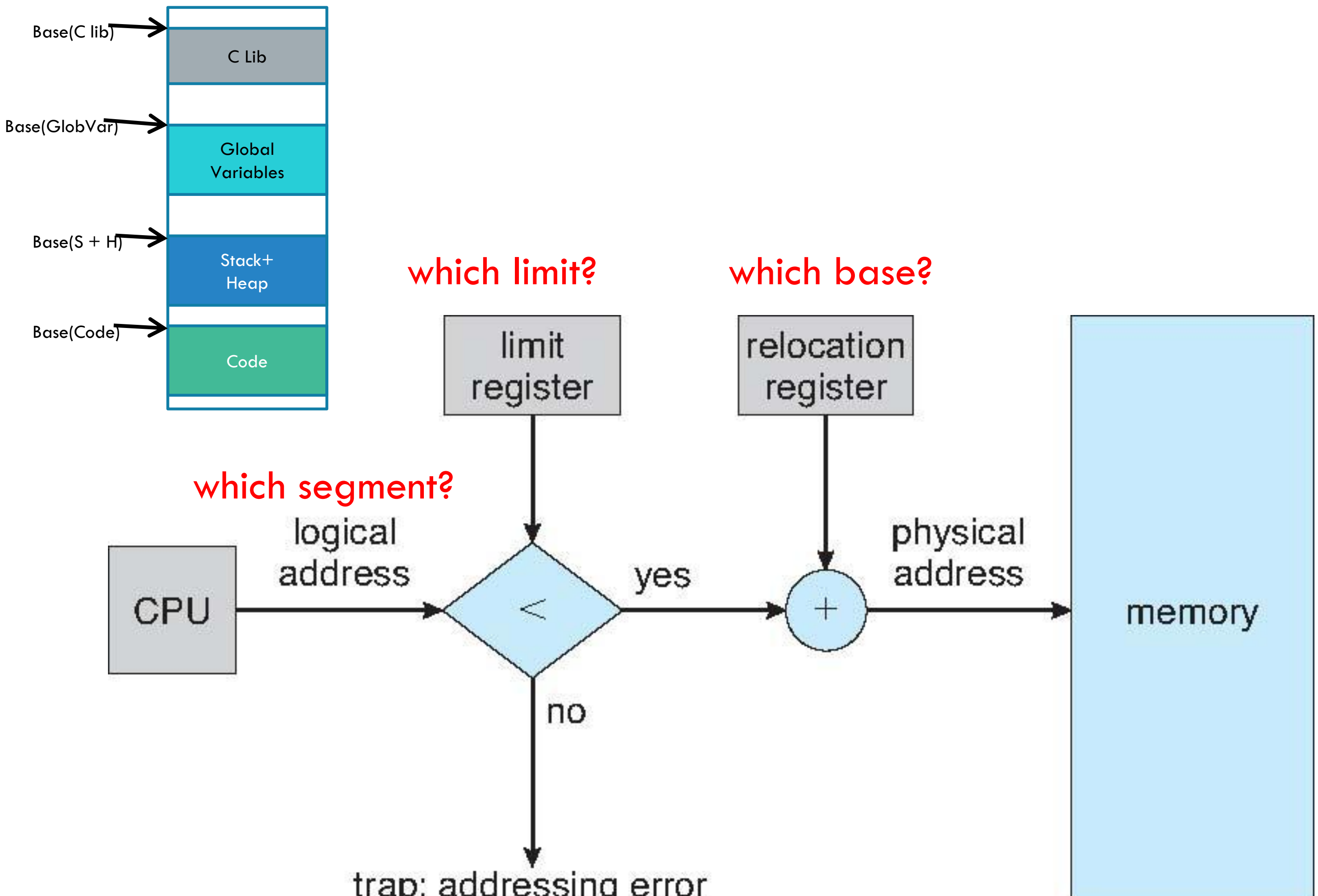
Idea: Why don’t we have multiple contiguous memory segments instead?

SEGMENTATION INTRO - III



Each process has multiple “segments”. Each segment has it’s own base and limit

DOES THIS SETUP STILL WORK?



SEGMENTATION LOGICAL ADDRESS

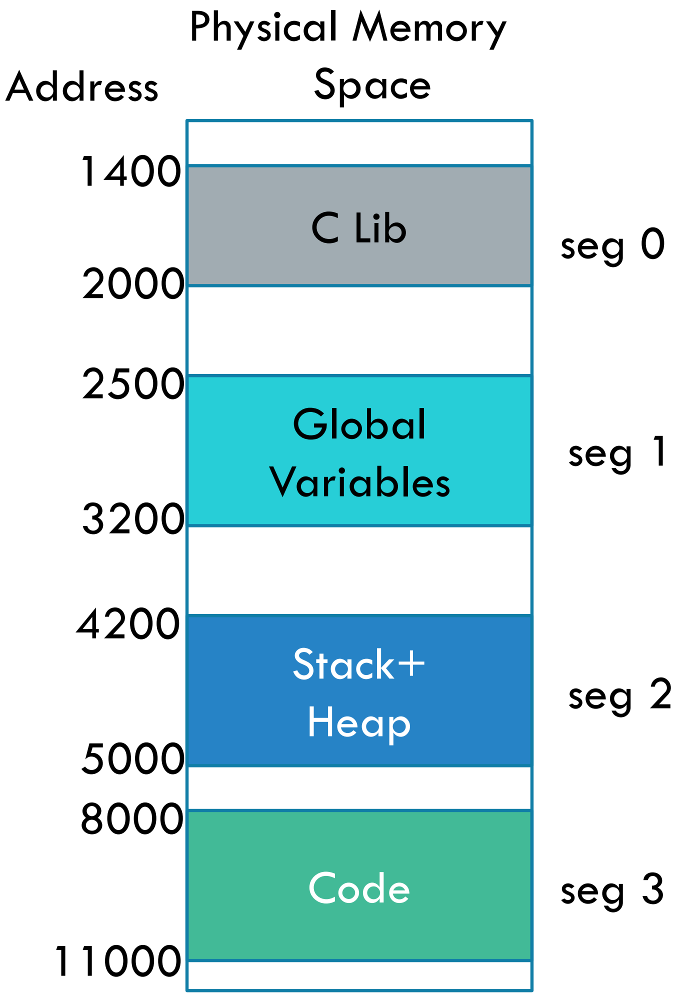
$<$Segment-number, offset$>$

1. Instead of a number, the 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

	Limit	Base
0	600	1400
1	700	2500
2	800	4200
3	3000	8000



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
- Does the MMU read the segment table for translation?
  - Yes… but which segment table does it read?
  - How in the world does a HW MMU know the address of the segment tables?!

SEGMENT TABLE REGISTER

segment table register

	Limit	Base
0	600	1400
1	700	2500
2	800	4200
3	3000	8000

Does the table look like this in memory?

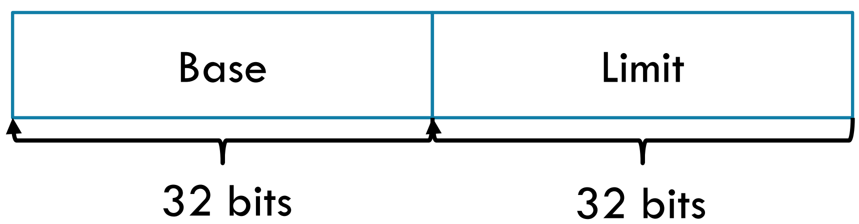
Fat Chance!

Segment table register

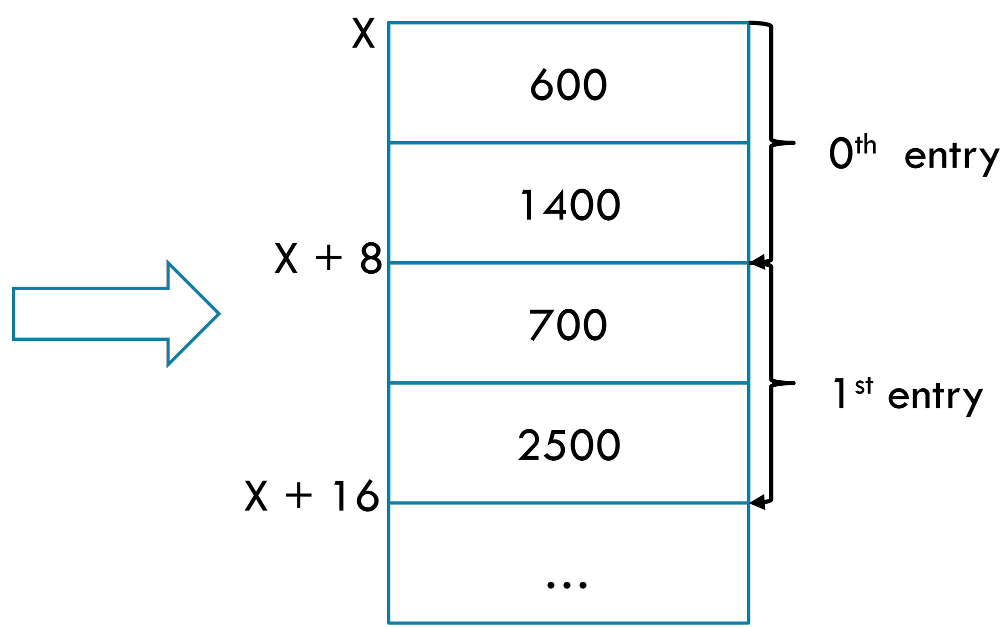
- Points to first address of segment table.

SEGMENT TABLE ENTRY

In our setup, a segment has 2 values – base and limit.

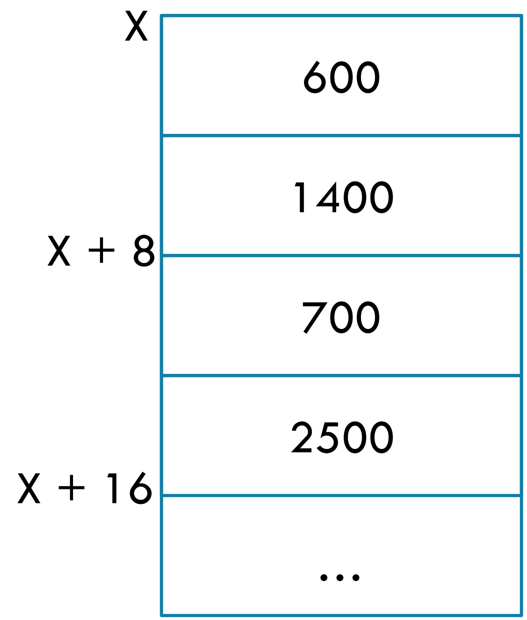


	Limit	Base
0	600	1400
1	700	2500
2	800	4200
3	3000	8000

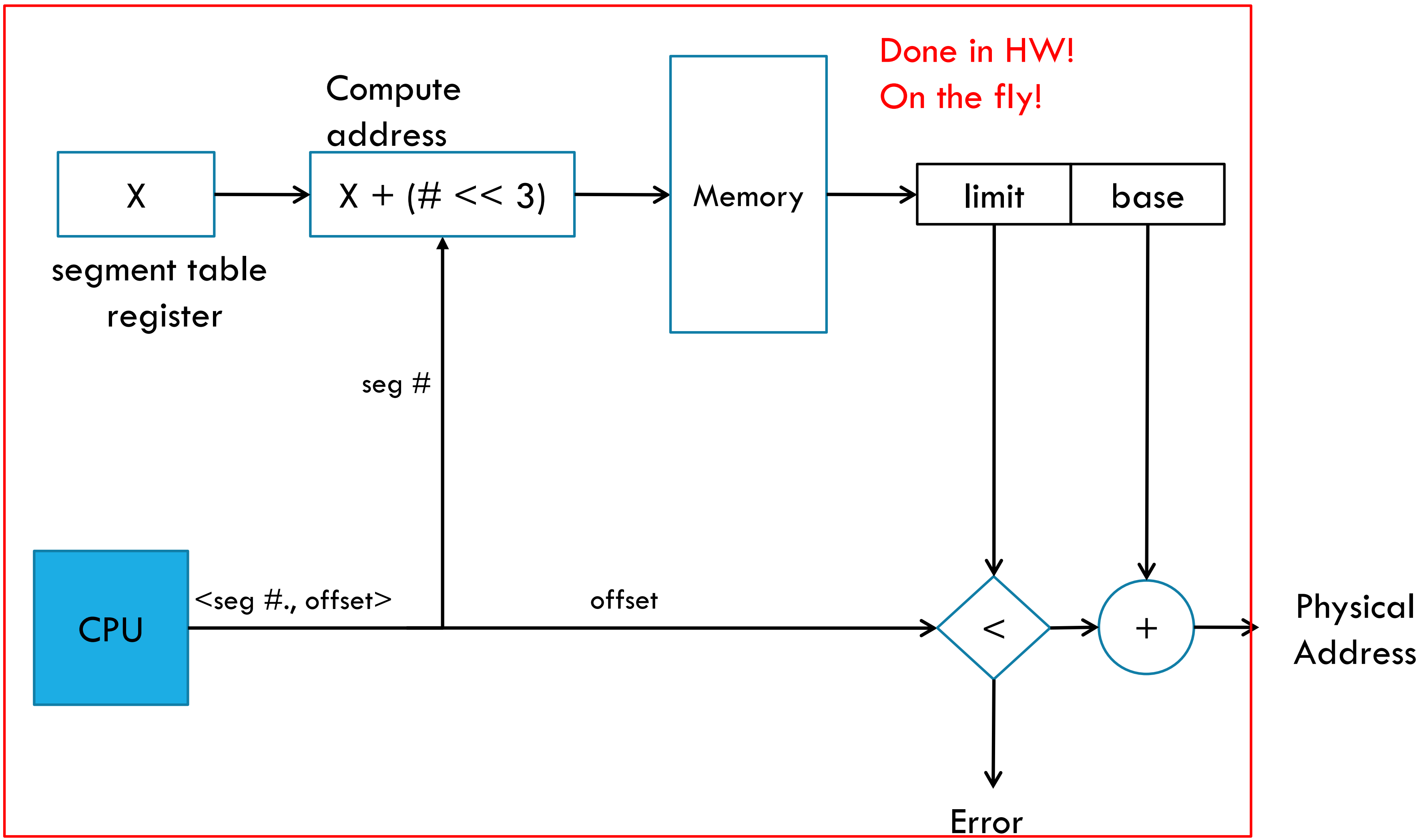


USING SEGMENT TABLE REGISTER + SEGMENT NO.

Segment table register	X
Segment No.	#
Address of Segment table entry matching #	X + (# << 3)



TRANSLATING LOGICAL ADDRESS TO PHYSICAL ADDRESS



SEGMENT TABLE SIZE

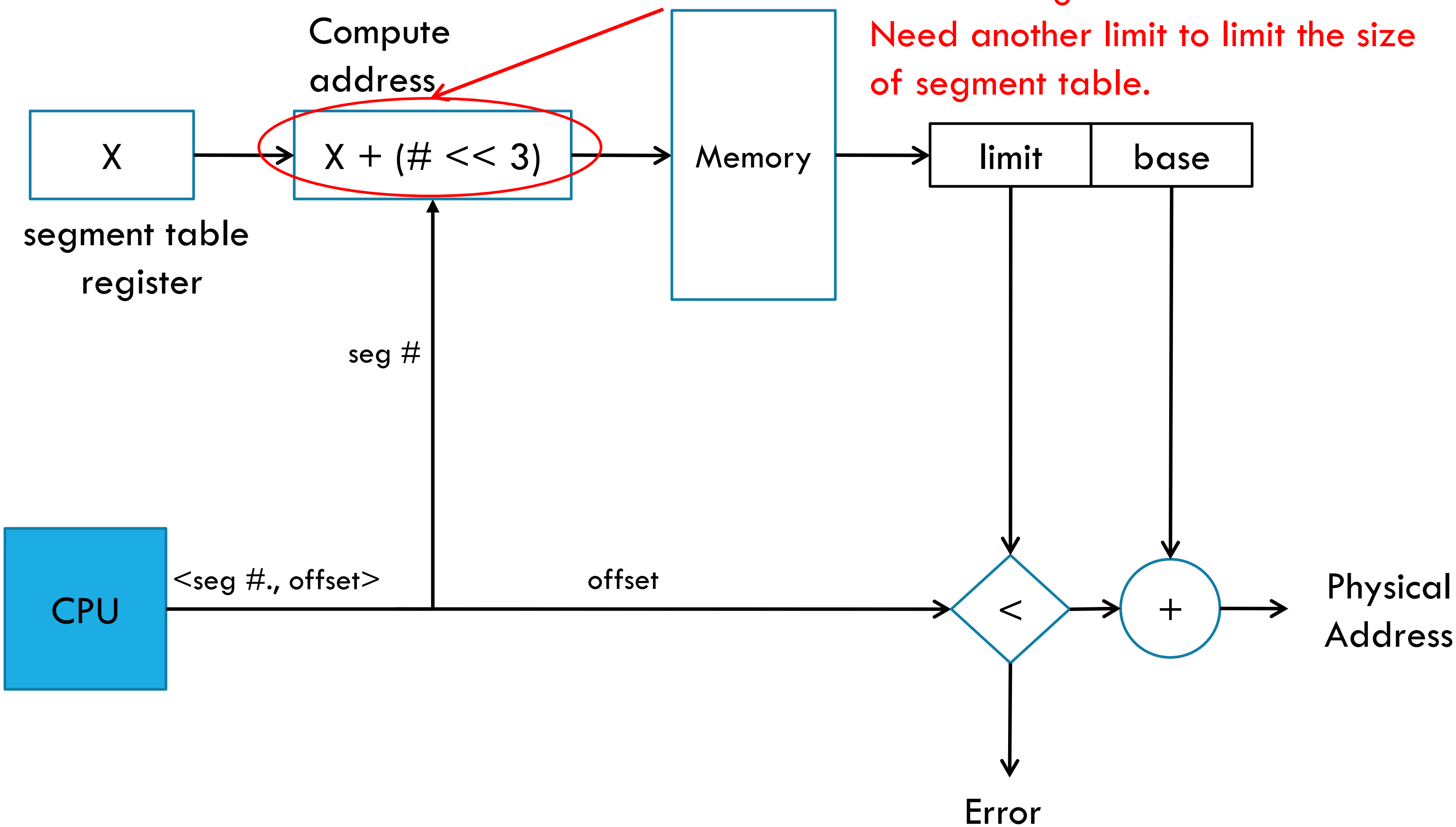
- Should the size be infinite?
  - This is crazy of course.
- Then.. Should the size be variable or fixed?
  - What does variable size mean?
    - As many segments as required by the process
  - What does fixed size mean?
    - Well.. There's a limit to number of segments a process can have.

VARIABLE SEGMENT TABLE SIZE

Does this still work then?

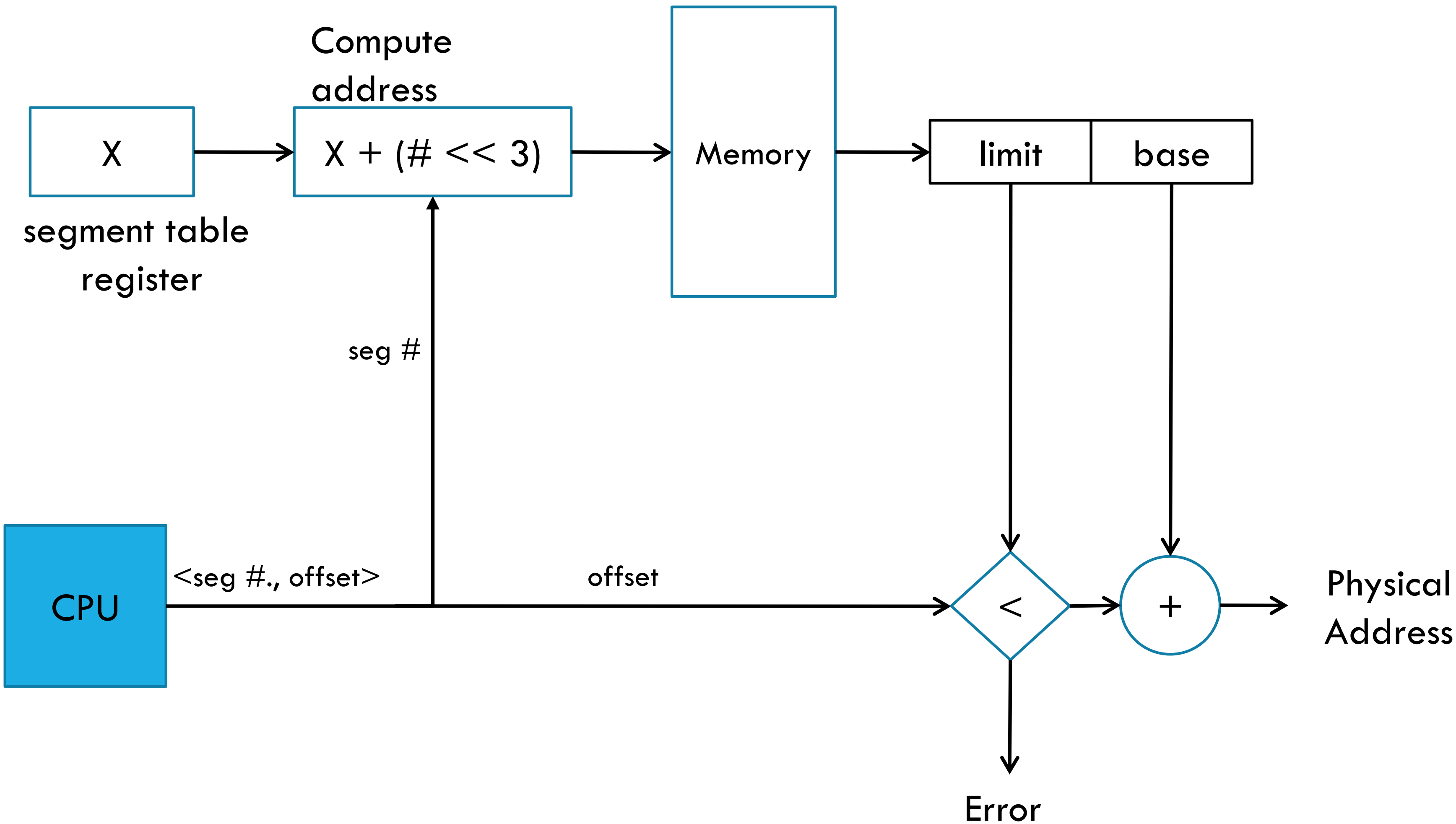
No way to tell whether this address exceeds the segment table area.

Need another limit to limit the size of segment table.

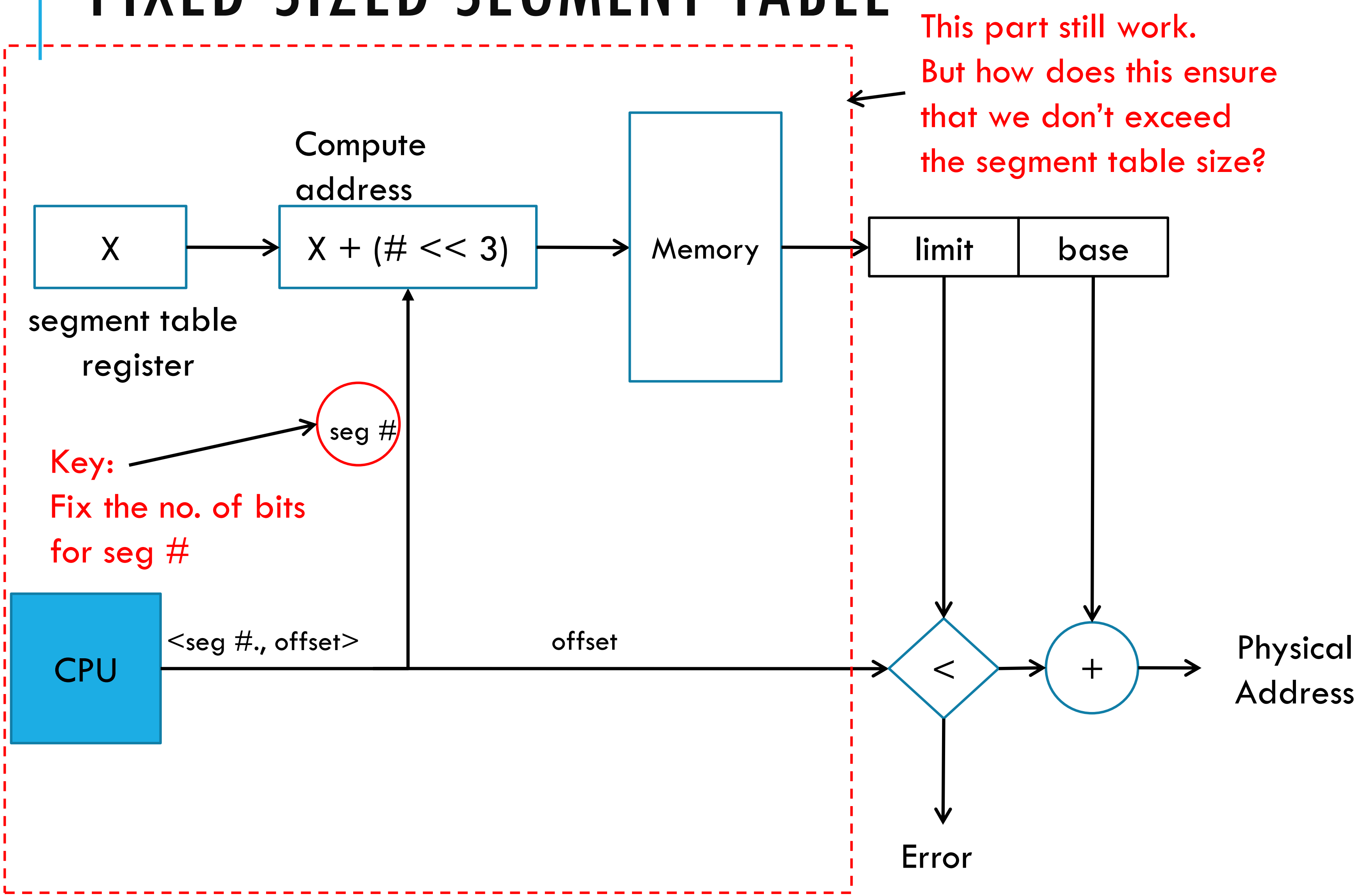


FIXED-SIZED SEGMENT TABLE

Does this still work then? … Yes and No. Basically, No.



FIXED-SIZED SEGMENT TABLE



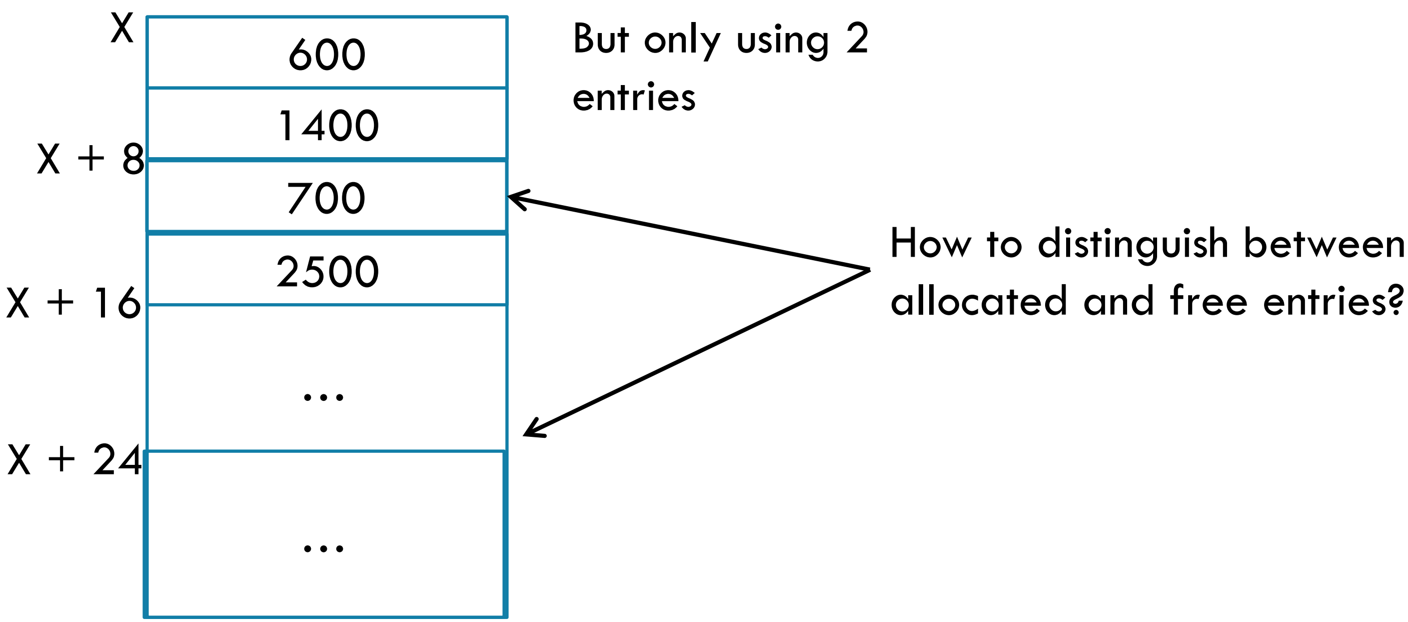
FIX NO. OF BITS?

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 size.

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

Recall that segment table
is but a series of numbers…
Say size of segment table
is 4 entries.



IMPROVED SEGMENT TABLE ENTRY

Base	Offset	P

P for present

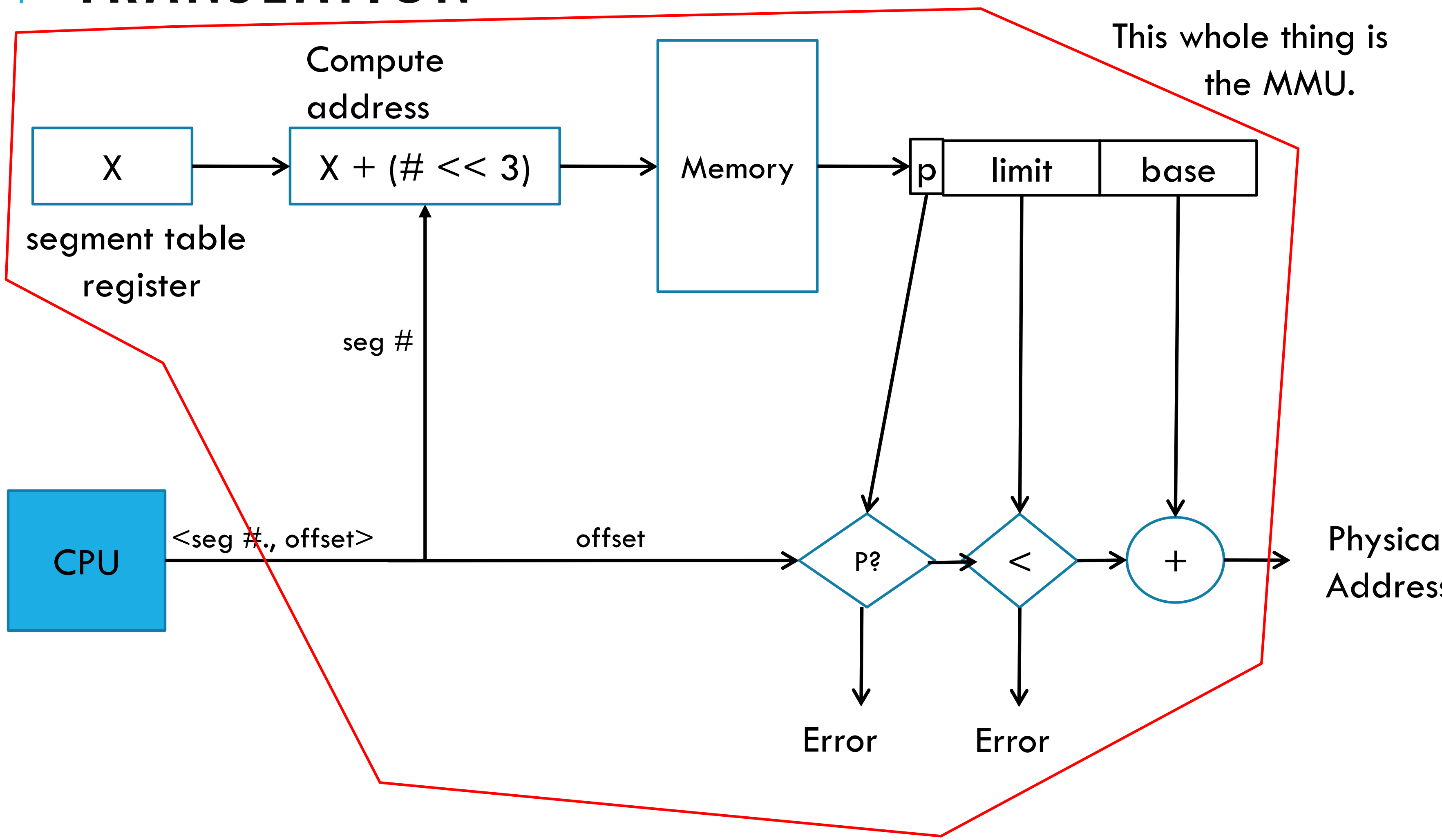
1 bit value

0 means segment not in memory

1 means segment in memory

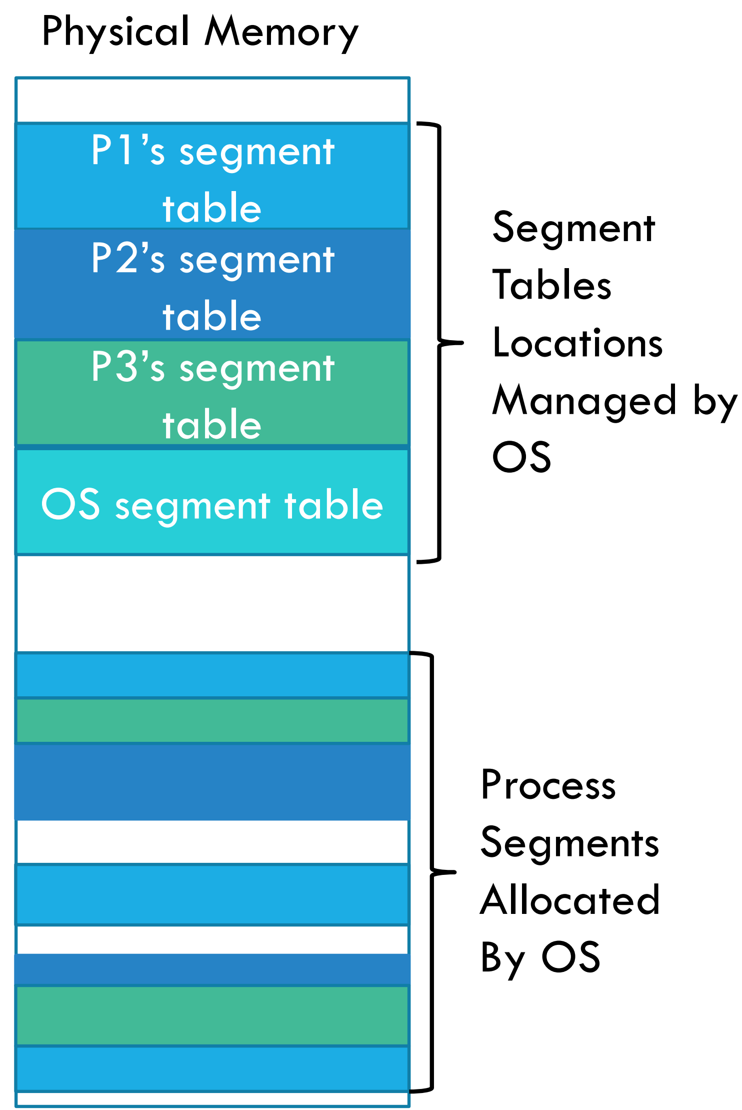
0 is a default value.

IMPROVED LOGICAL ADDRESS TRANSLATION



SEGMENTATION AND CONTEXT SWITCHING

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



IN PRACTICE

Segment Logical Address slightly more complex

- 1 more bit indicating whether accessing local or global segment table

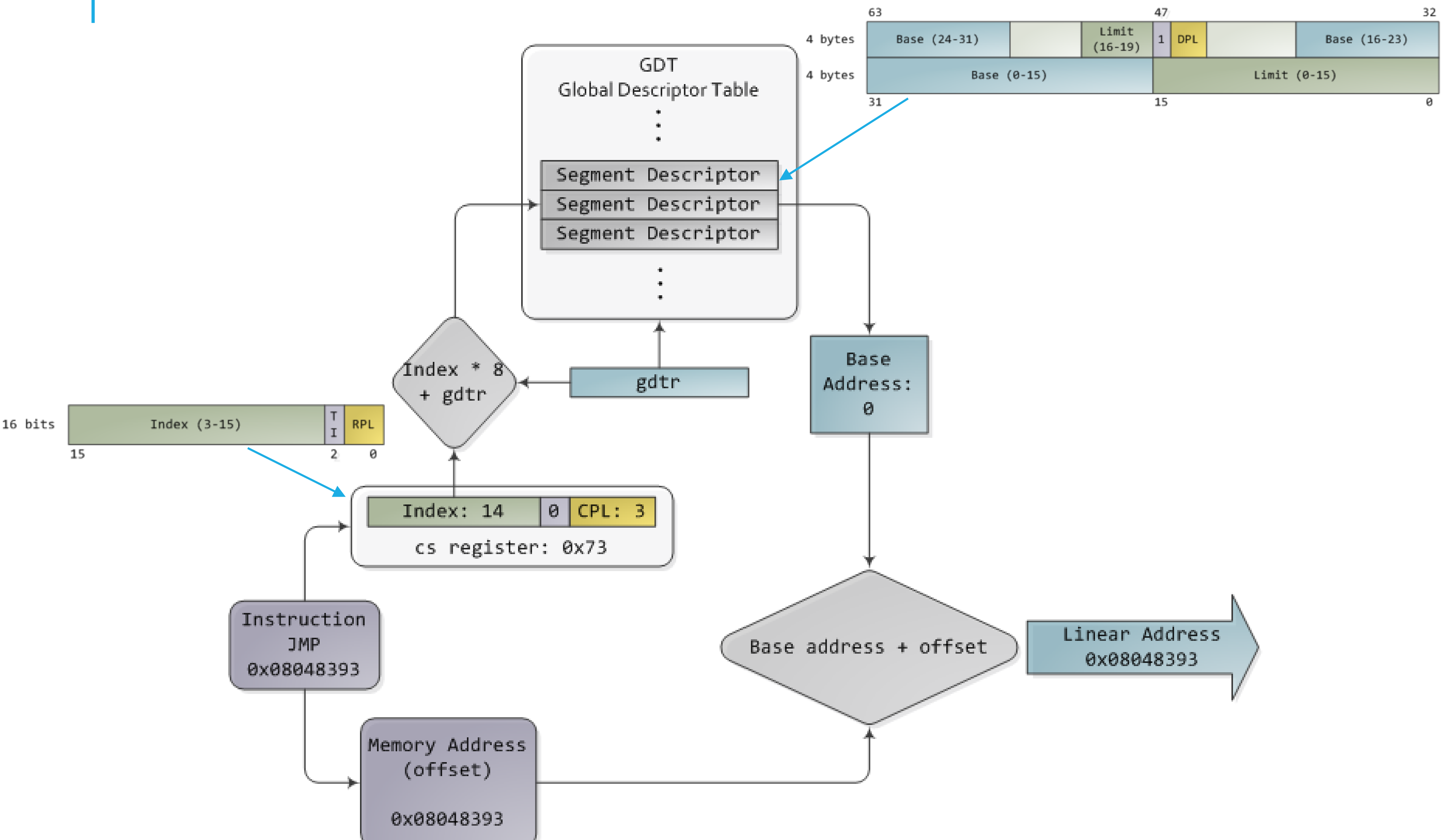
Segment Table Entry more complex

- Include fields such as privilege, permissions etc.

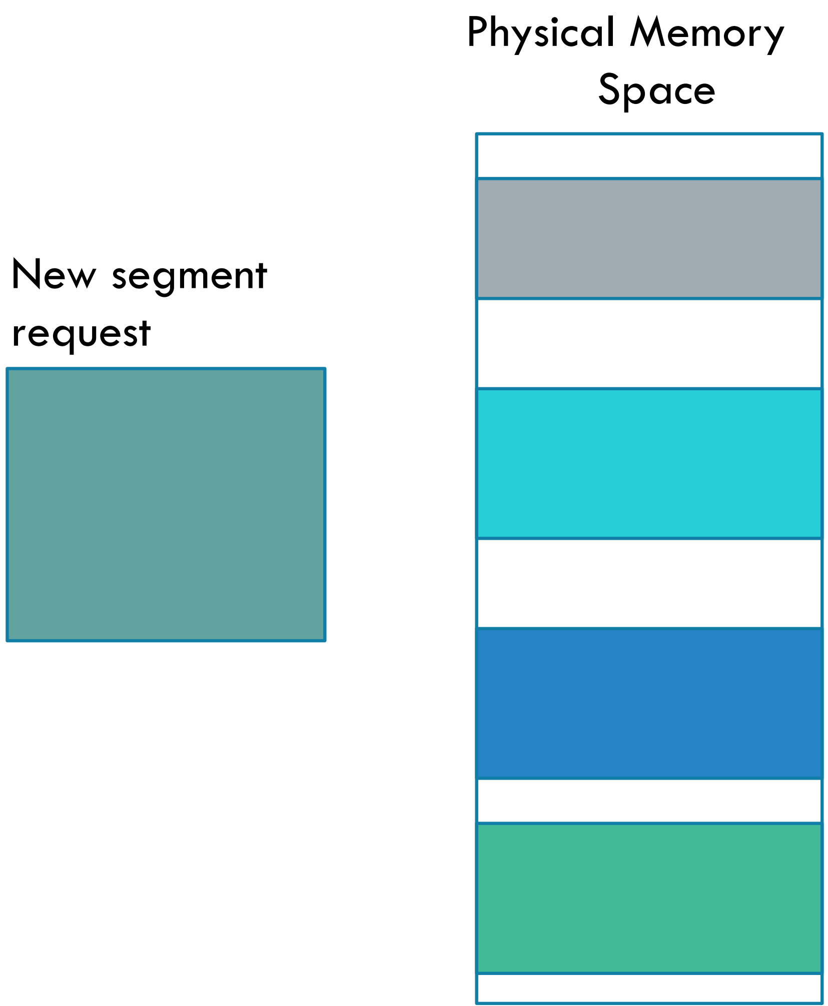
Caching segment table entries

- MMU can cache entries to save on memory lookups.

LINUX 32-BIT PROTECTED MODE



RUNNING OUT OF PHYSICAL MEMORY SPACE

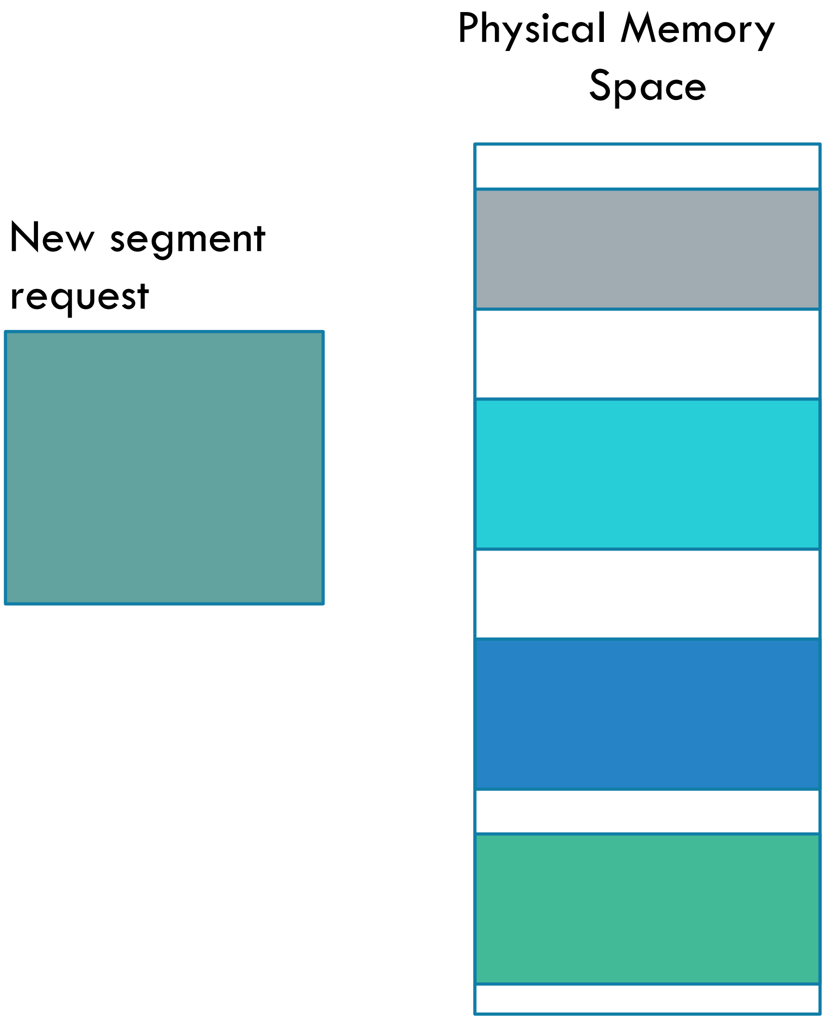


Cannot fit! What to do?

Options:

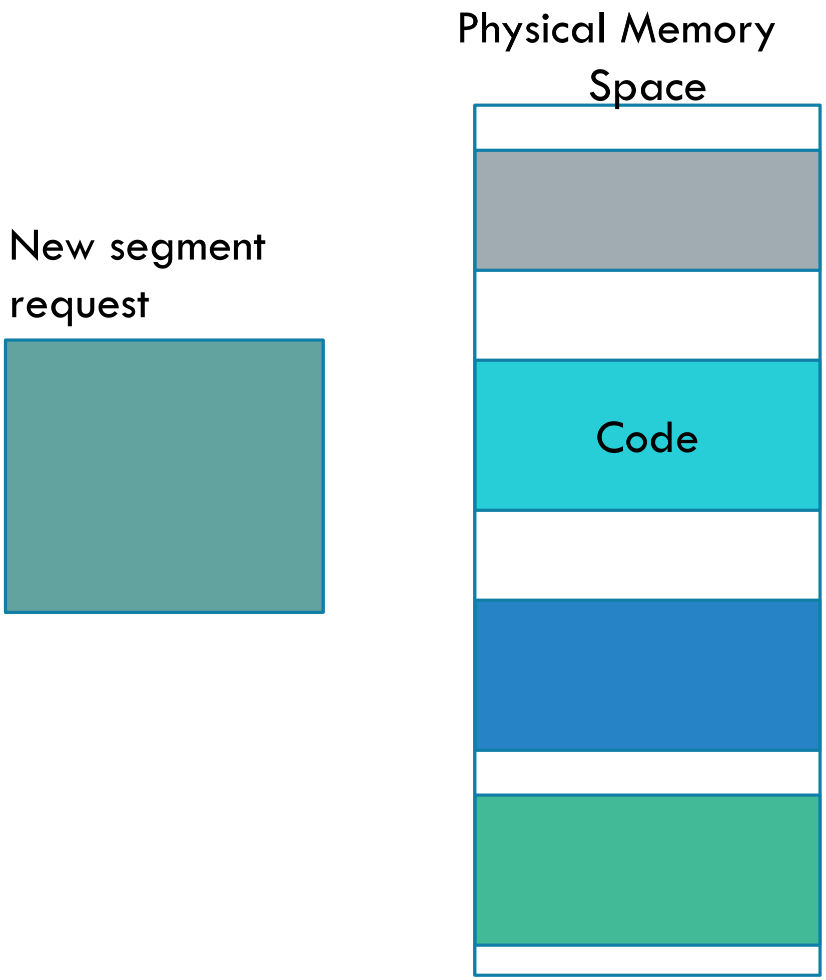
1. Reject the request
2. Delete some read-only segments
3. Move some segments into hard disk.

DELETE SOME READ-ONLY SEGMENTS

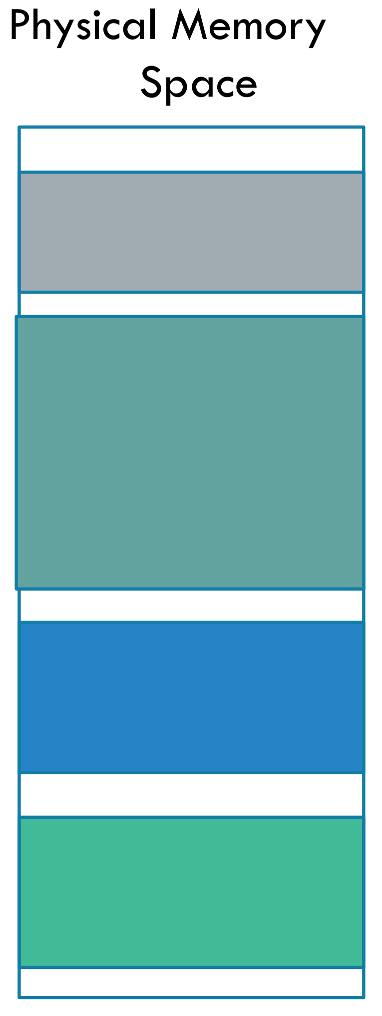


- Usually this means removing some code segments
  - Whose code?
    - If Request is P1's, *usually* some non-P1's code.
  - Why code?
    - If P2's code is missing when it's needed, OS can recover P2's code from permanent storage.

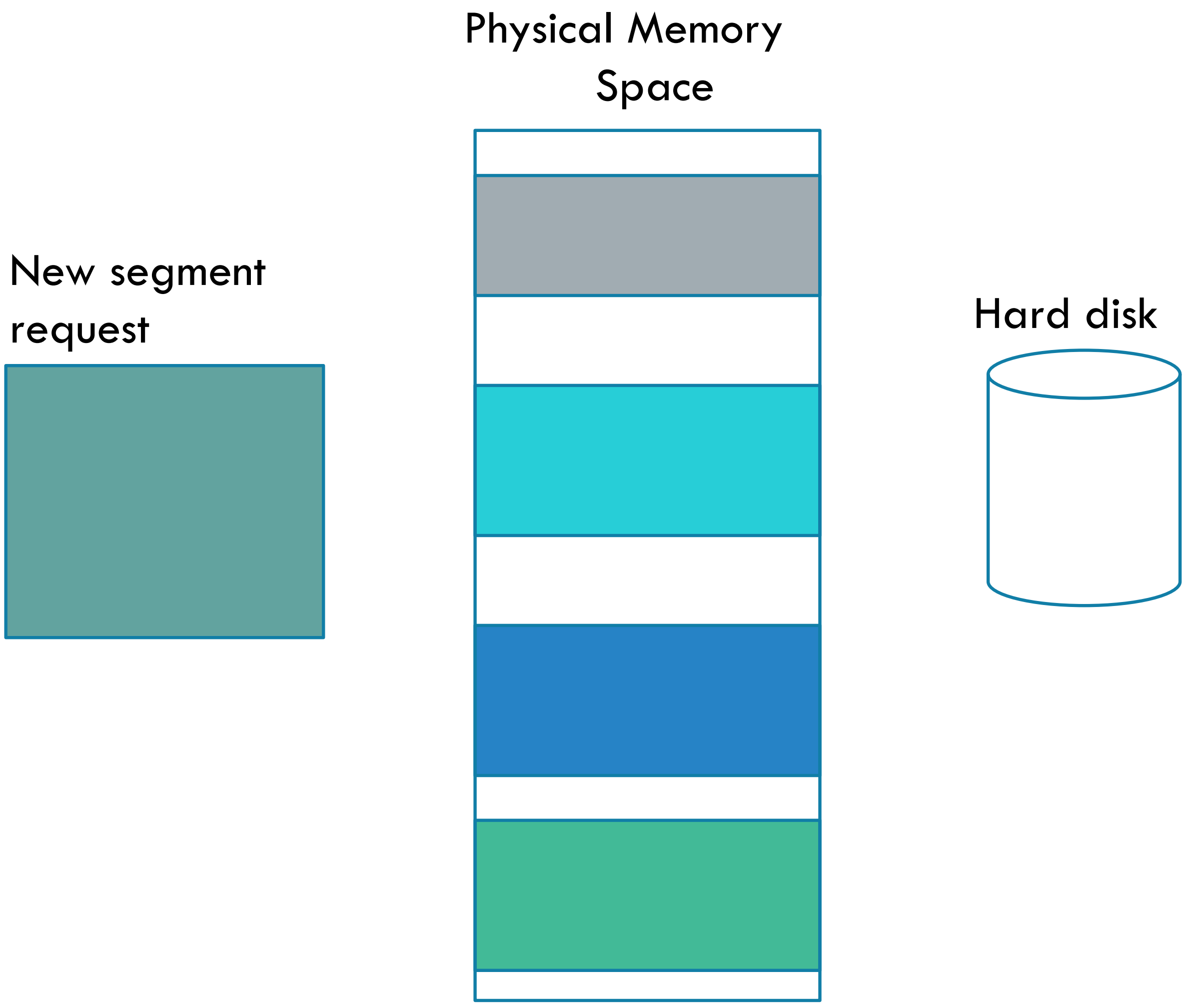
DELETE SOME READ-ONLY SEGMENTS



DELETE SOME READ-ONLY SEGMENTS



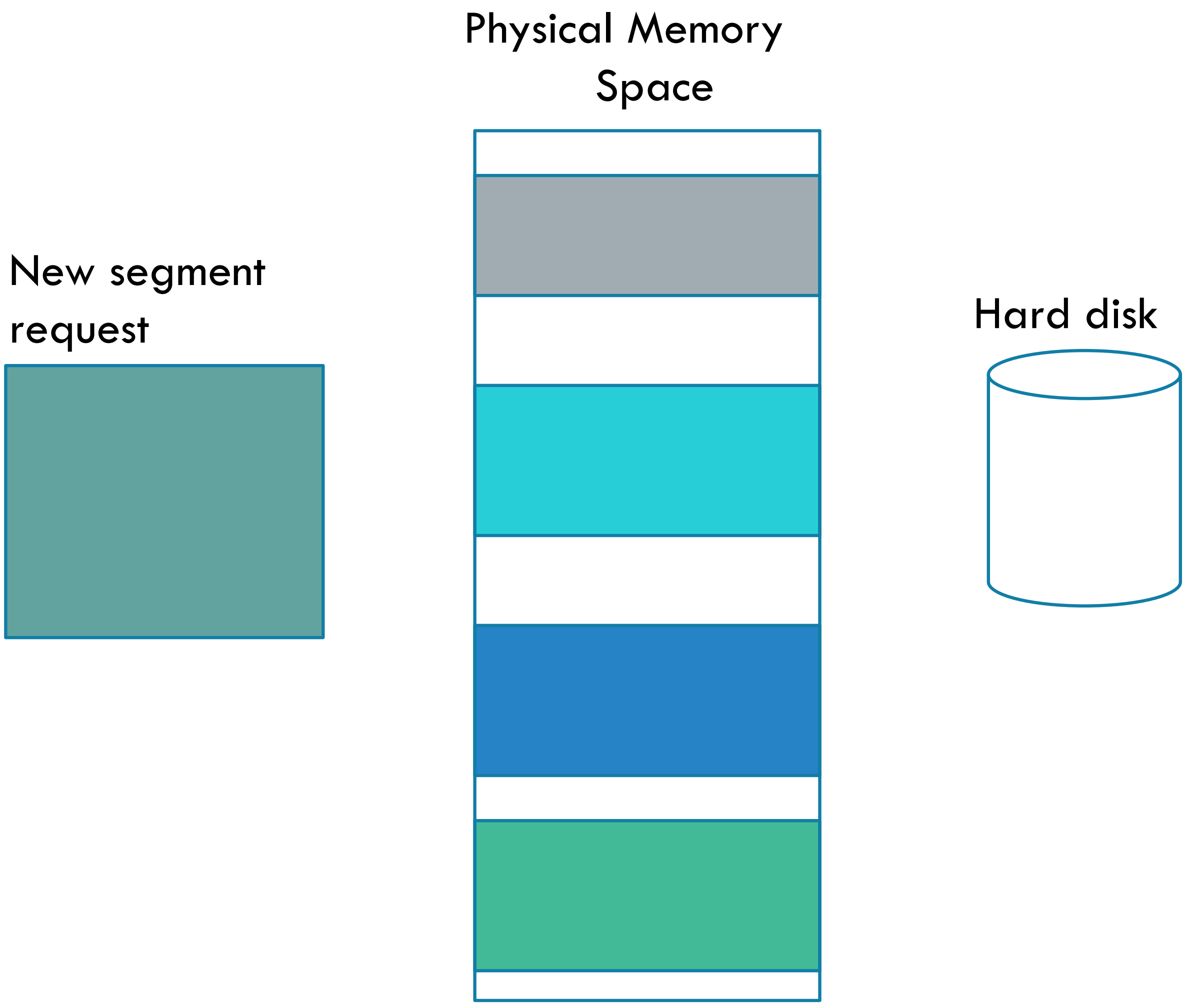
USING HARD DISK AS A BACKING STORE



OS need to reserve a portion of the hard disk for storing segments when needed.

(Usually some unformatted space)

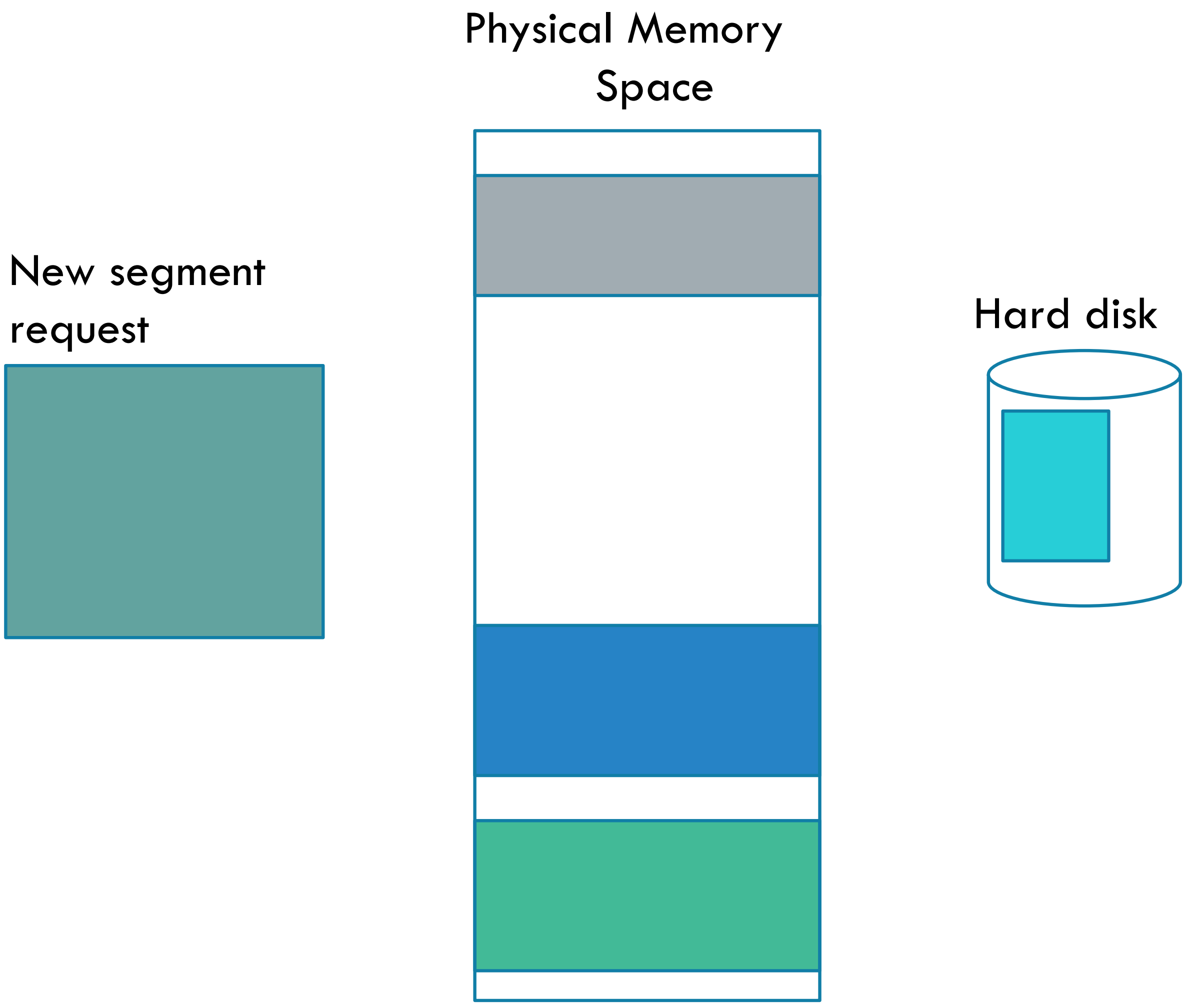
USING HARD DISK AS A BACKING STORE



What needs to be done?

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

USING HARD DISK AS A BACKING STORE

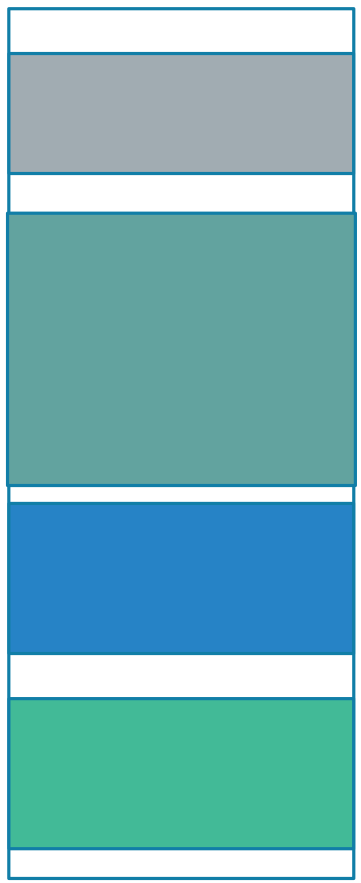


What needs to be done?

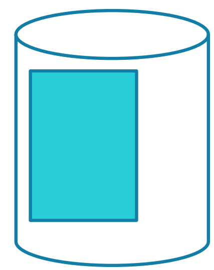
1. Need to choose at least 1 segment to be put into hard disk.
2. For the chosen segment, the present bit of the corresponding segment entry is set to 0.
3. Move the segment into hard disk

USING HARD DISK AS A BACKING STORE

Physical Memory Space



Hard disk



What needs to be done?

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

ACCESSING A NON-PRESENT SEGMENT

C Code

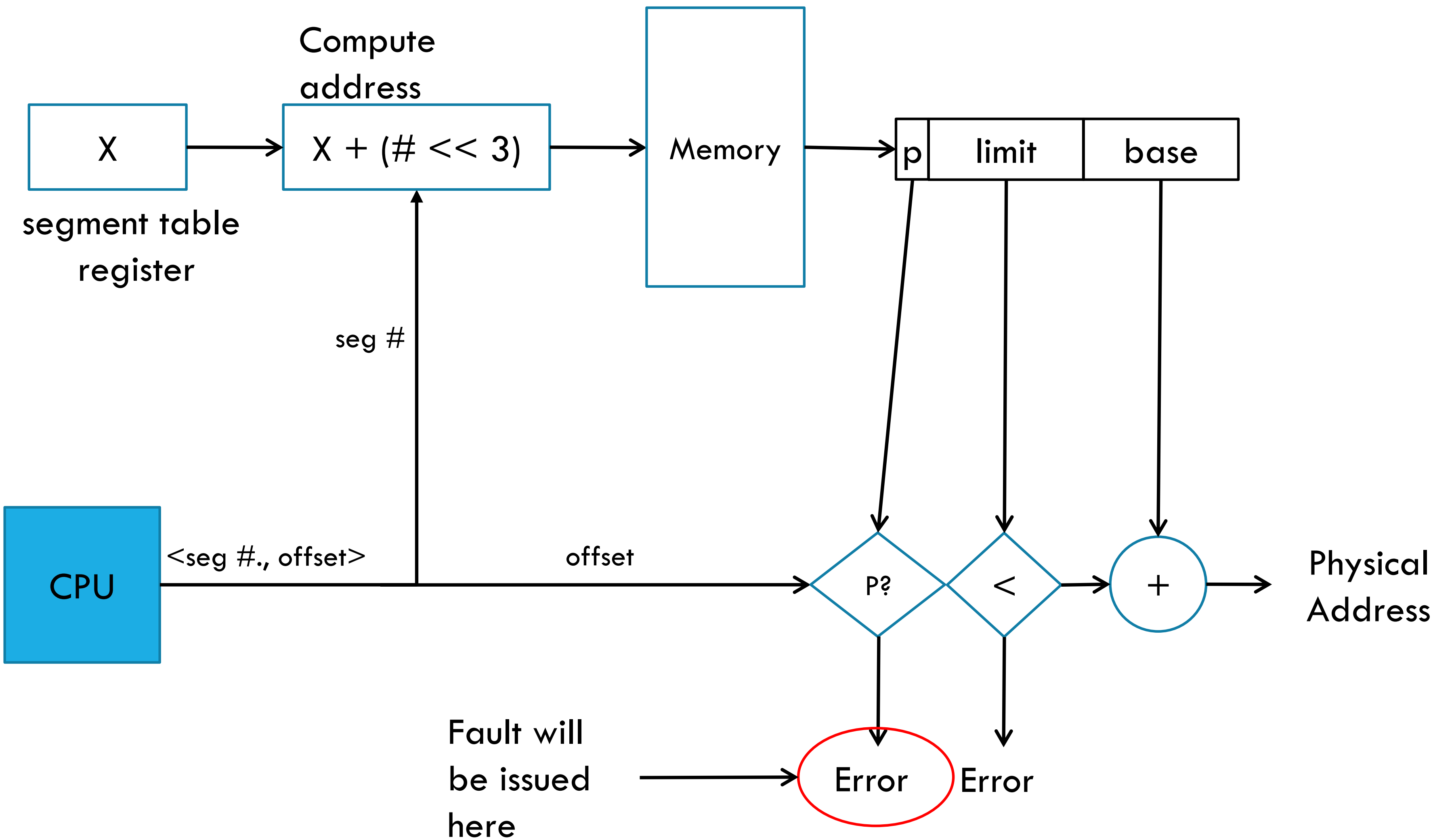
```
…
x = y + z;
array[10] = x;
…
```

Assembly

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

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

RECALL...

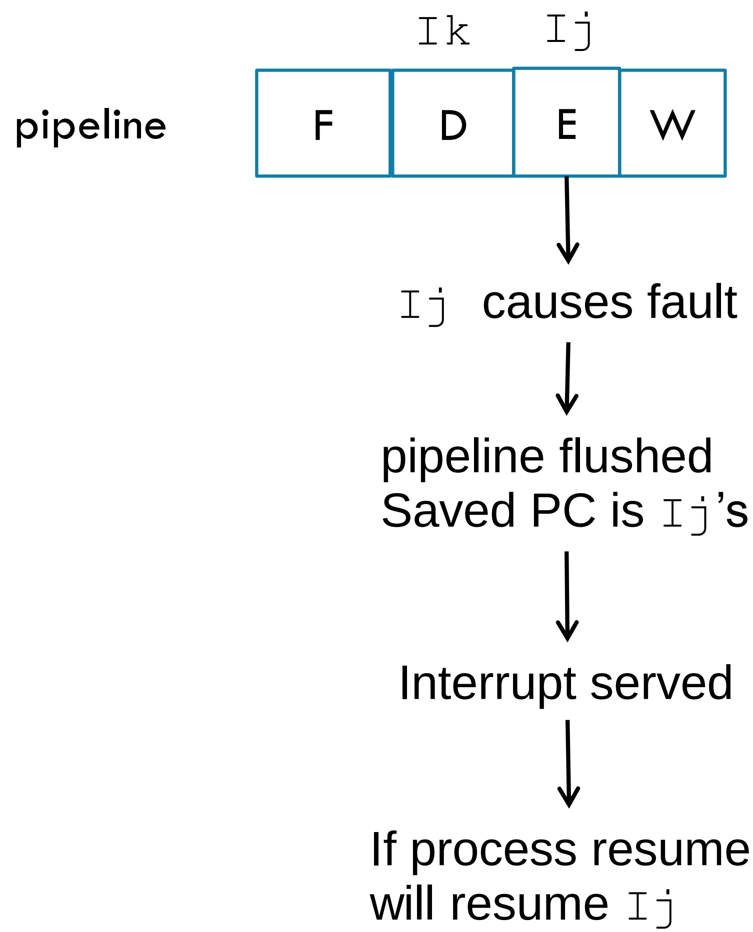

Compute
address
X
X + (# << 3)
Memory
p
limit
base
segment table
register
seg #
CPU
<seg #., offset>
offset
P?
<
+
Physical
Address
Fault will
be issued
here
Error
Error

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.



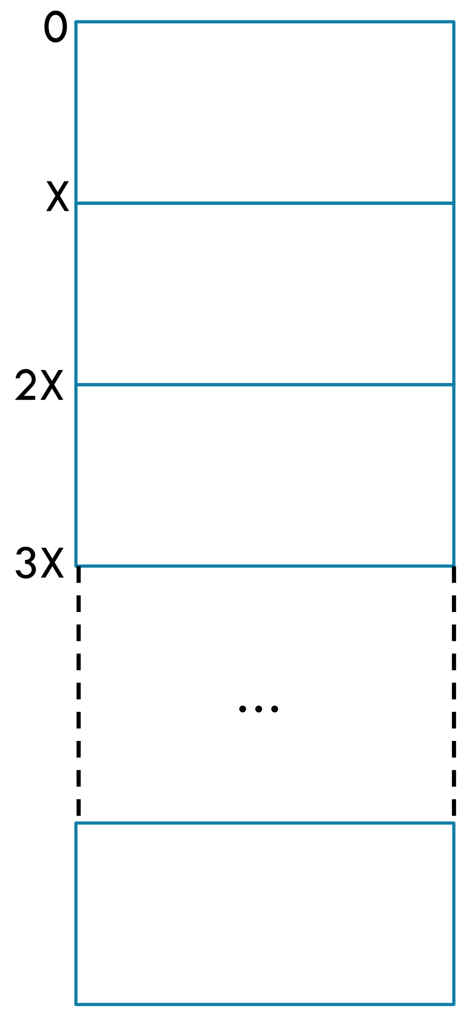
HANDLING MEMORY FAULTS

- 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 segment present, 2 possibilities
    - Illegal access. (permissions problem). Terminate process
    - Out of bounds access. Terminate
  - If segment is not present, 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.

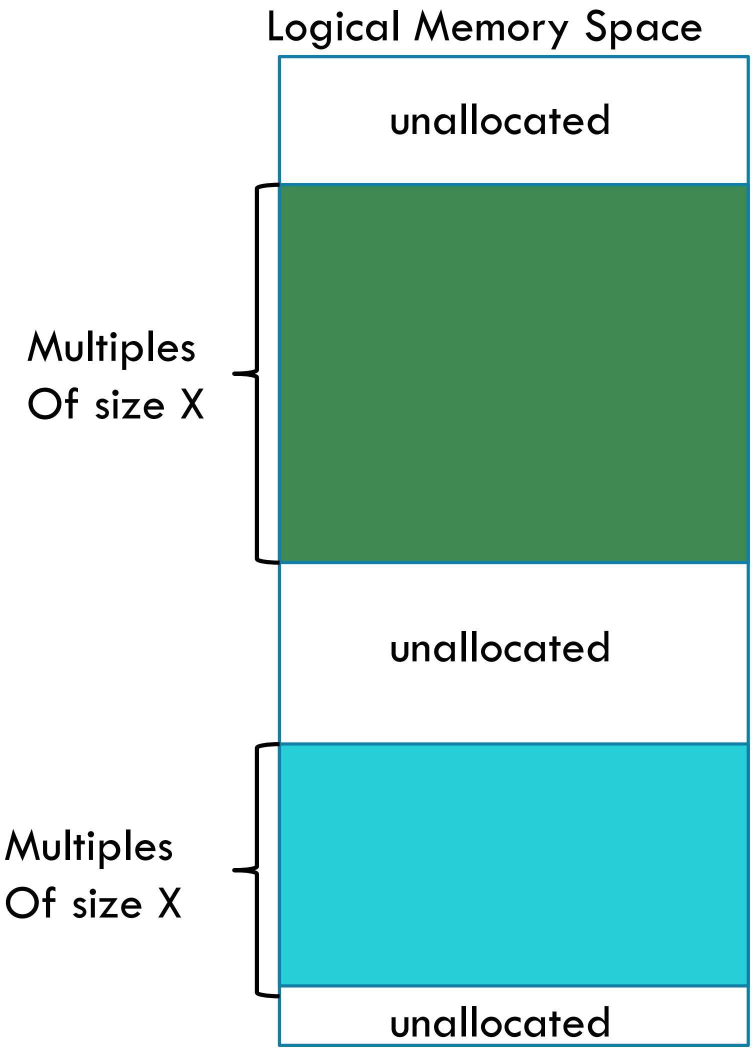
PAGING (NON-CONTIGUOUS MEMORY ALLOCATION)

Divide up physical memory into “page frames”.

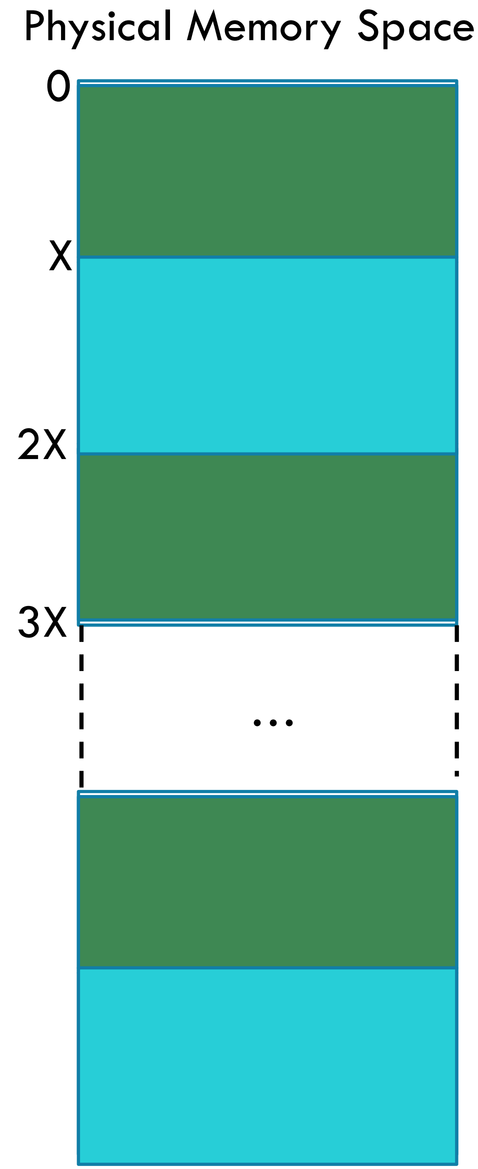
Each page frame has the same fixed size.



PAGING OVERVIEW

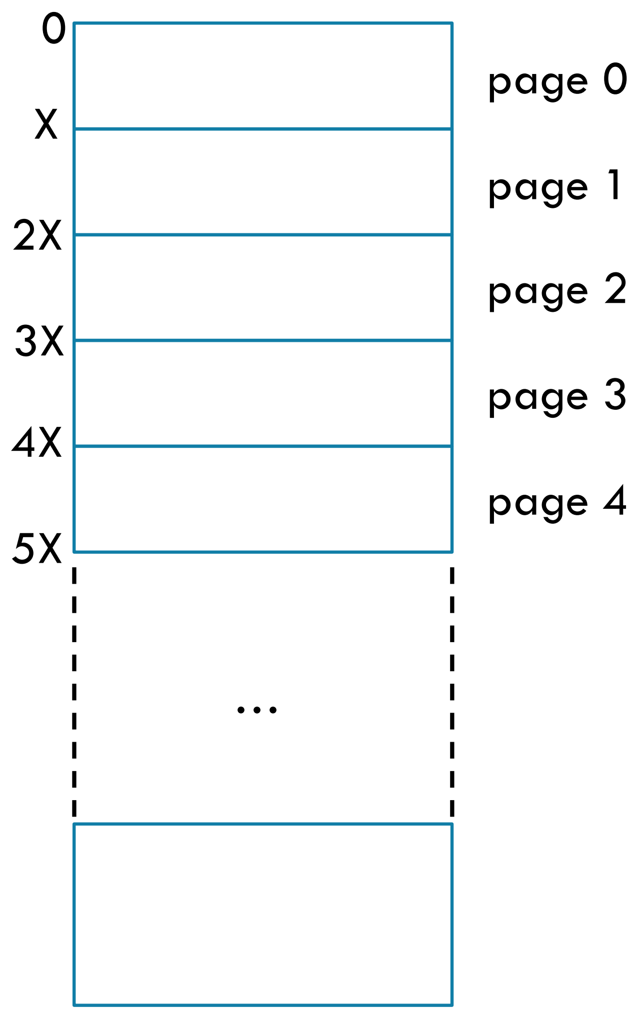


1. What appears to be contiguous to process may not be in physical memory.
2. Allocated memory space is always a multiple of size X.
3. X is the page size.



DEMO SHOWING MEMORY ALLOCATION FOR PAGING.

LOGICAL ADDRESS SPACE



Logical address space is organized in terms of **pages.**

X is usually powers of 2.

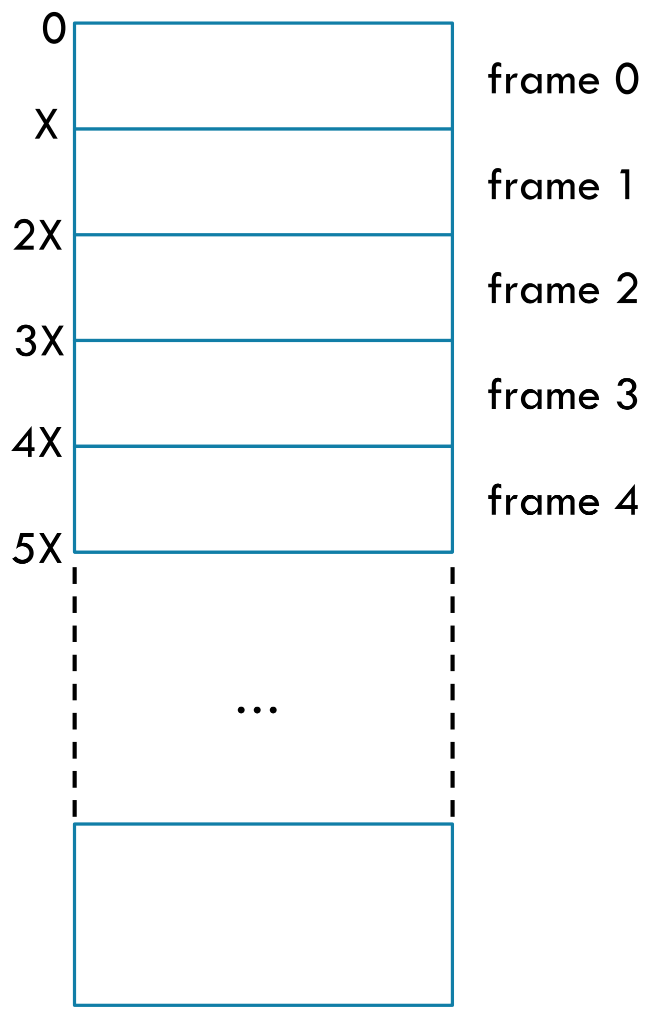
PAGING LOGICAL ADDRESS

page number	offset
1010 0101 1010 0101 1010	0101 1010 0101

Just a number!
But….

So.. what does offset = 0 mean?
1st address of pages.

PHYSICAL ADDRESS SPACE



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

PHYSICAL ADDRESS

frame number	offset
1010 0101 1110 0111 1010	0101 1010 0101

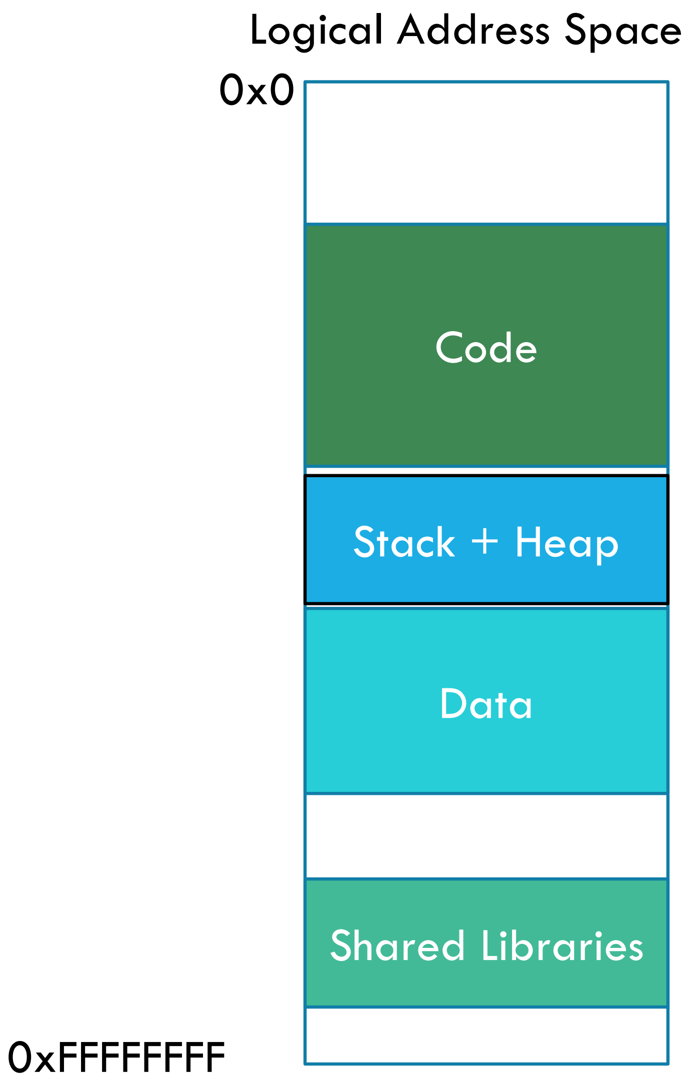
Just a number!
But….

So.. what does offset = 0 mean?
1^{st} address of frames.

WHAT? THE PREVIOUS FEW SLIDES LOOK SO SIMILAR!

1. What's the diff btw page and frames?
   a) Frames are going to contain pages.
   b) Pages can move but frames can't.
2. Any other differences?
   a) No. of frames limited by amt of physical memory.
   b) No. of pages … limited by number of address bits for logical address.

IMPLICATION? FREEDOM!

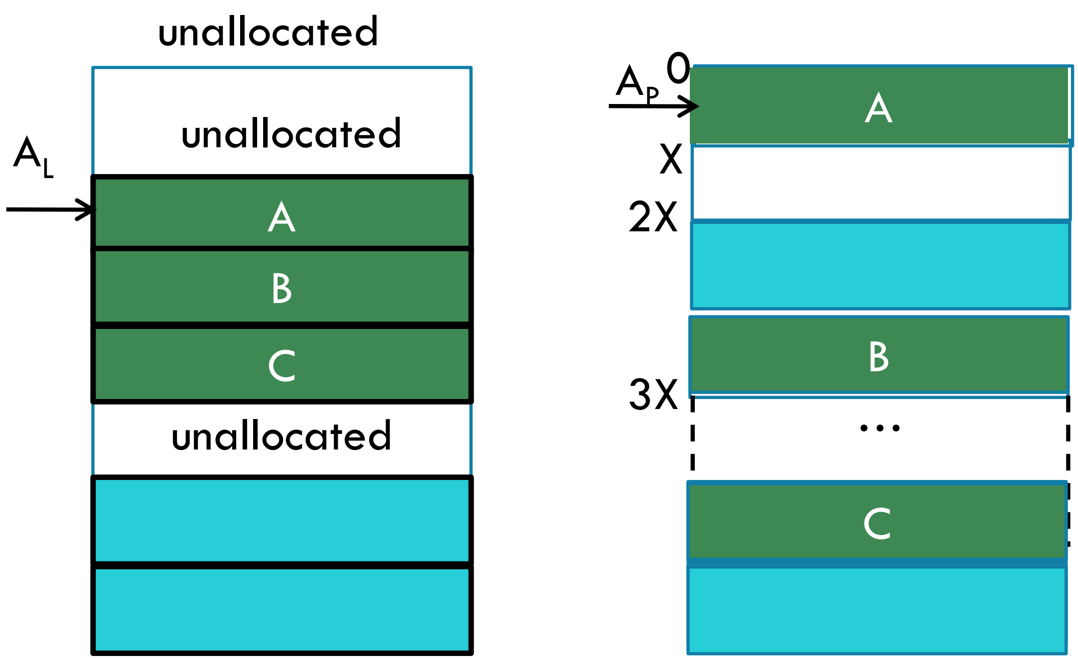


1. Code and Data's logical address generated in compile-link time. Freedom for compiler and linker to set address of code and data.
2. Stack + Heap and Shared Libraries are allocated by loader/OS. Freedom for OS to allocate logical addresses for these regions.

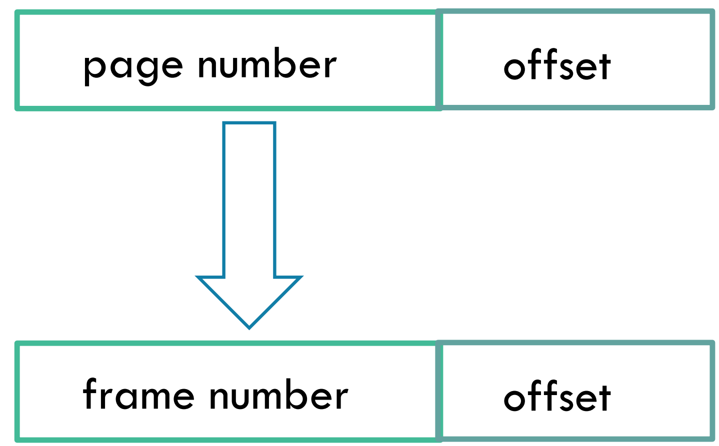
TRANSLATING LOGICAL ADDRESS INTO PHYSICAL ADDRESS

Logical Address Space
(what a process sees)

Physical Memory Space
(but really)



So really, translating A_L into A_P.

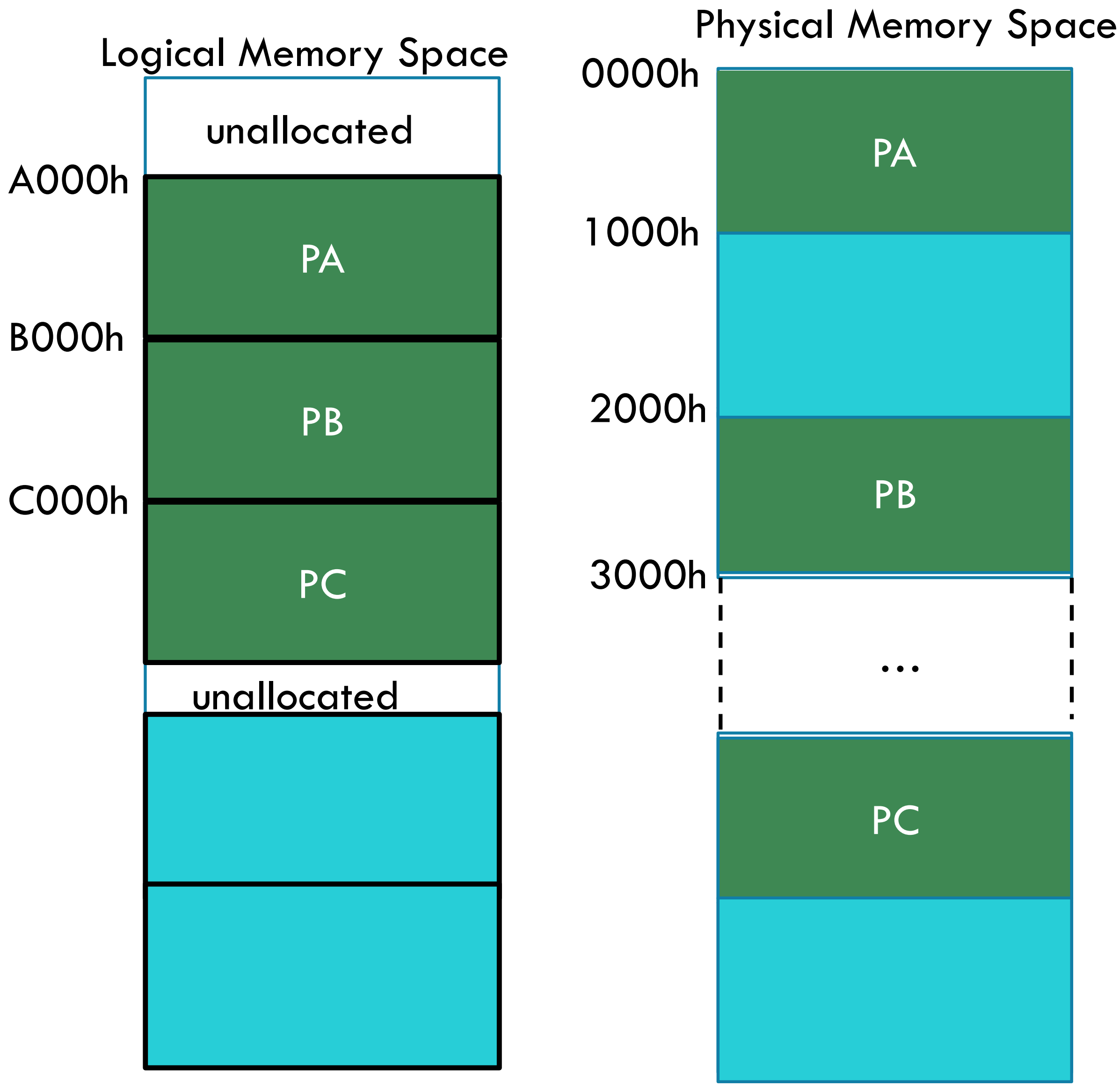


Is really about finding the matching frame number for a page number.
How to match them?

PAGE TABLE

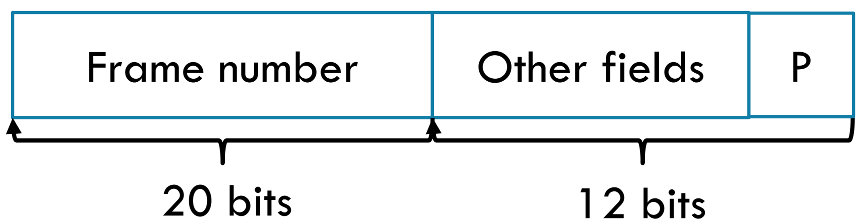
Page Size = 4K

	Frame Number
…	
A	0
B	2
…	

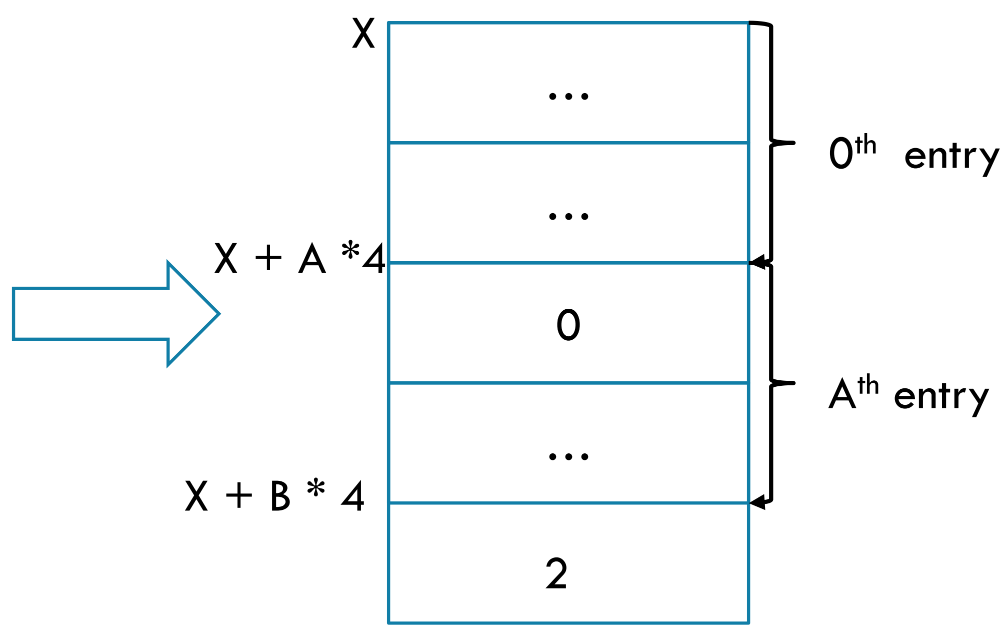


PAGE TABLE ENTRY

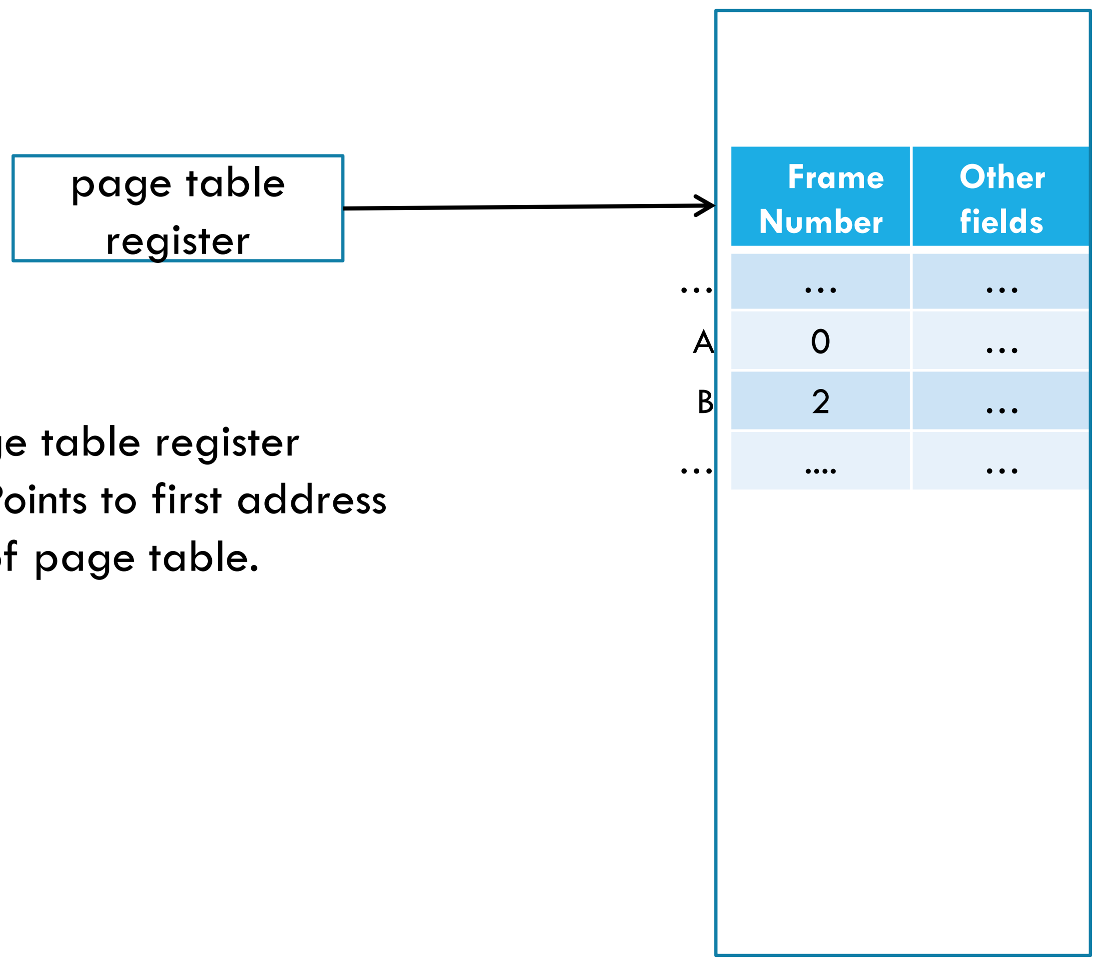
A realistic page table entry contains more than just frame number



	Frame Number	Other fields
…		…
A	0	…
B	2	…
…		…



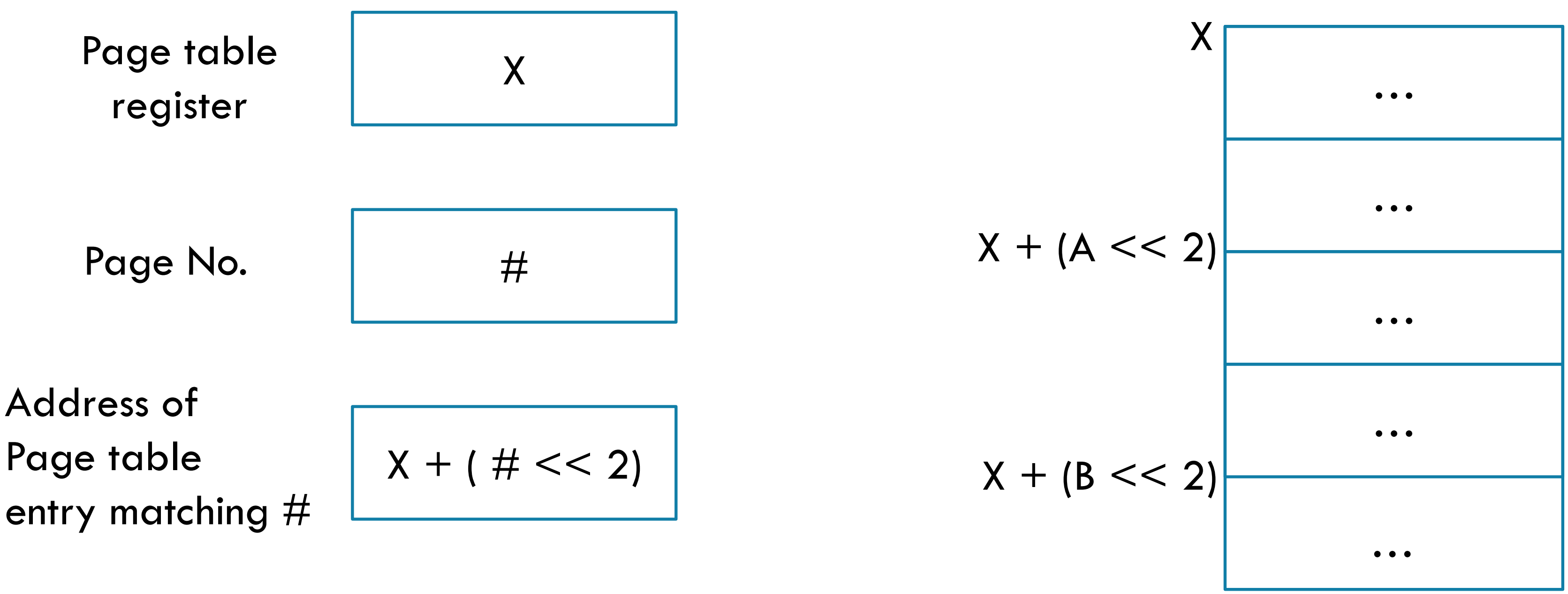
PAGE TABLE REGISTER



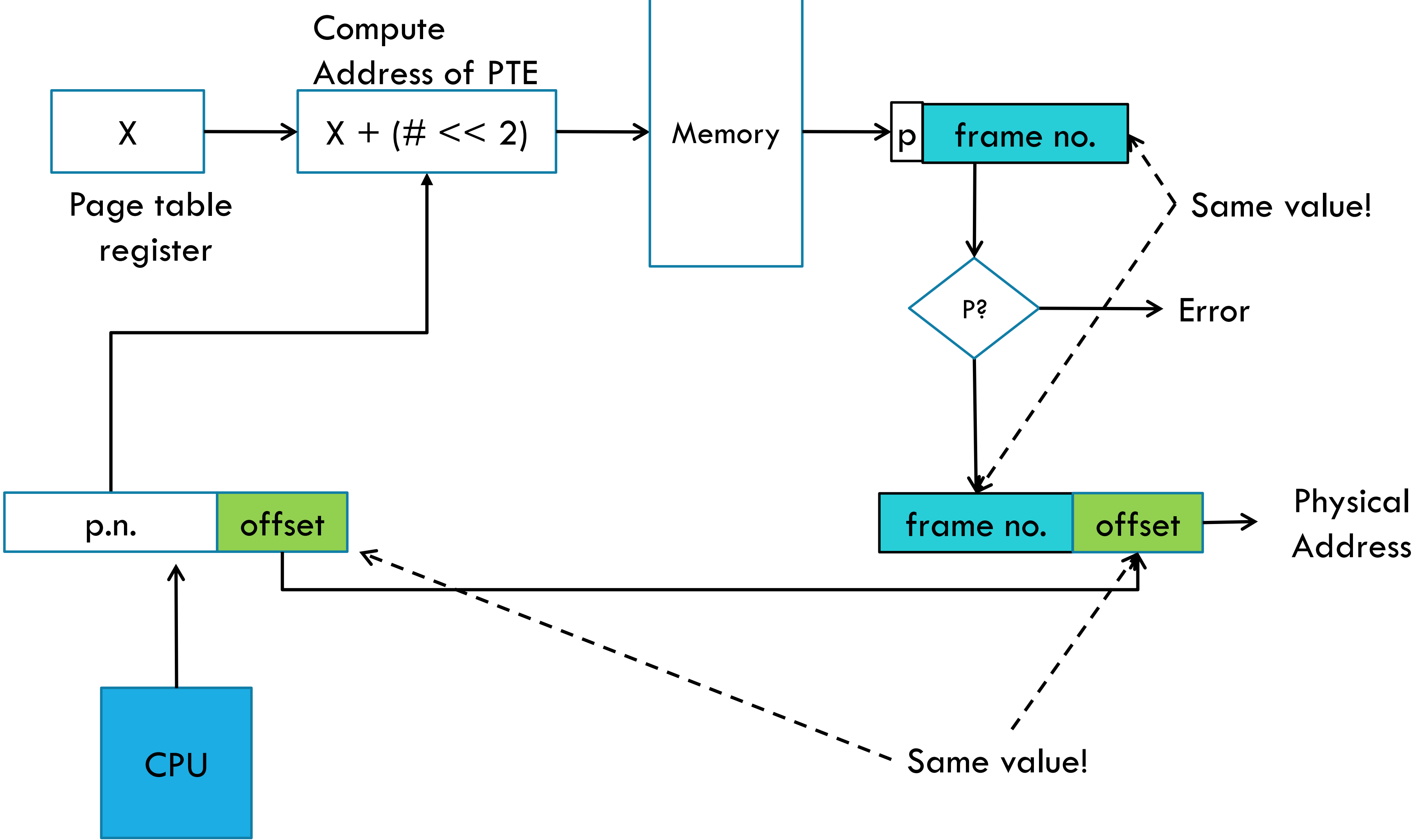
page table register

- ❑ Points to first address of page table.

USING PAGE TABLE REGISTER + PAGE NO.



PAGING LOGICAL ADDRESS TRANSLATION

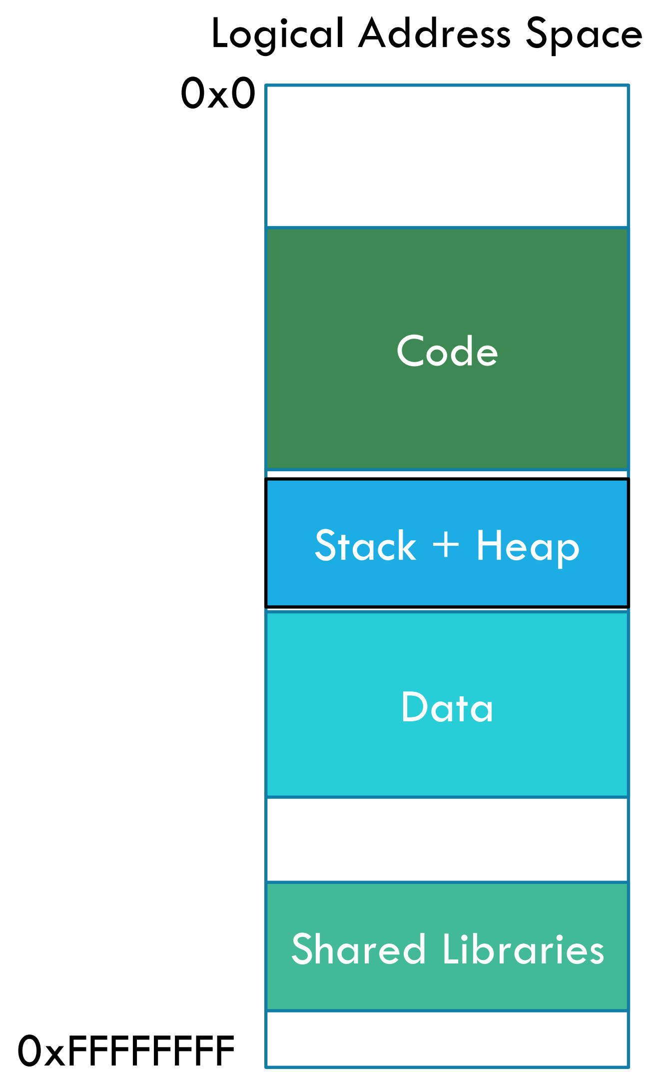


PAGE TABLE SIZES

- Page size = 2^N bytes
- Number of page table entries = $2^{(\text{Address bits} - N)}$
- Page table size = $2^{(\text{Address bits} - N)}$ x sizeof(1 page table entry)
- So, say if
  - Address bits is 32
  - Page size is 4K = 2^{12} bytes.
  - Page Table Size = 2^{20}x 4 = 4MB
  - Each process needs a page table size of 4MB! Can we do better?

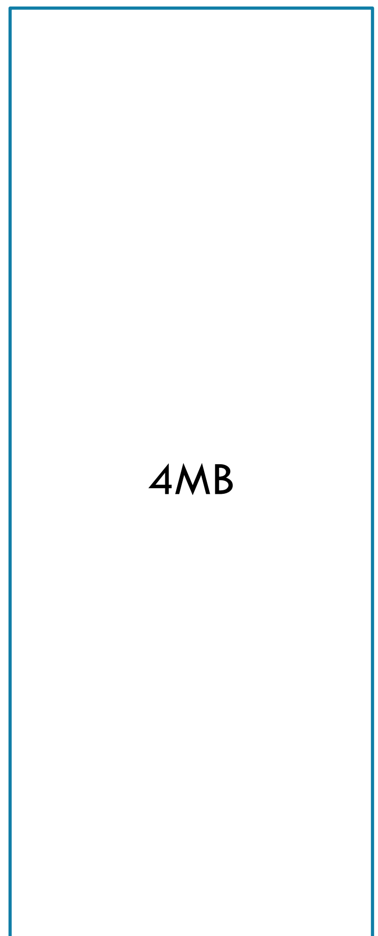
REDUCE SIZE OF PAGE TABLES!

- Idea: Waste of space to maintain Page Table Entries for all the non-allocated space.
- Do so by employing 2-level paging.
- Idea: Make a page table of the page table.



2-LEVEL PAGING BASIC IDEA

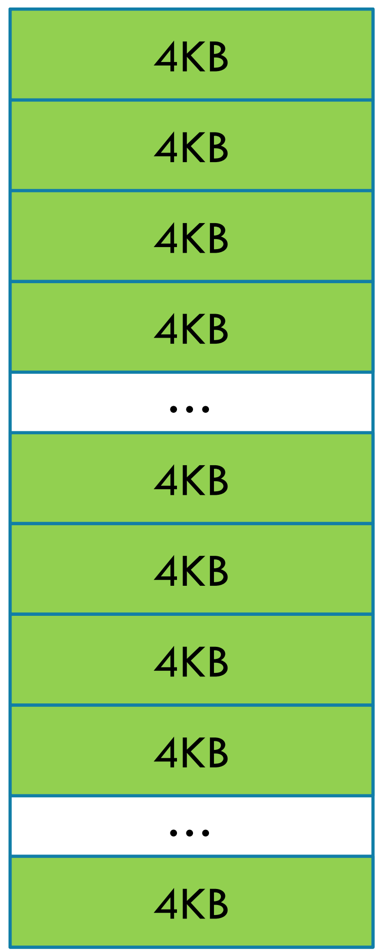
Original Page Table



We can break this down into “pages”.

2-LEVEL PAGING BASIC IDEA

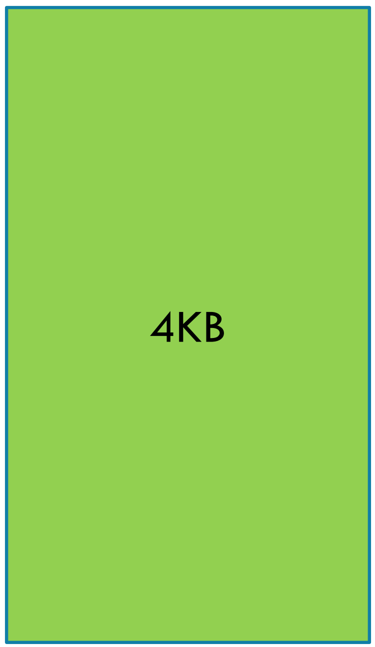
Original Page Table



Let's think about how each "page" of page table entries matches the logical address space.

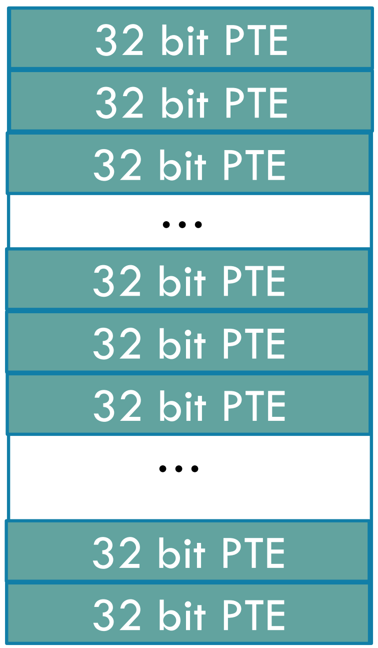
4MB will consists of 1K of 4KB. i.e., 4 MB consists of 1K pages.

2-LEVEL PAGING BASIC IDEA



4KB of page table entries

2-LEVEL PAGING BASIC IDEA

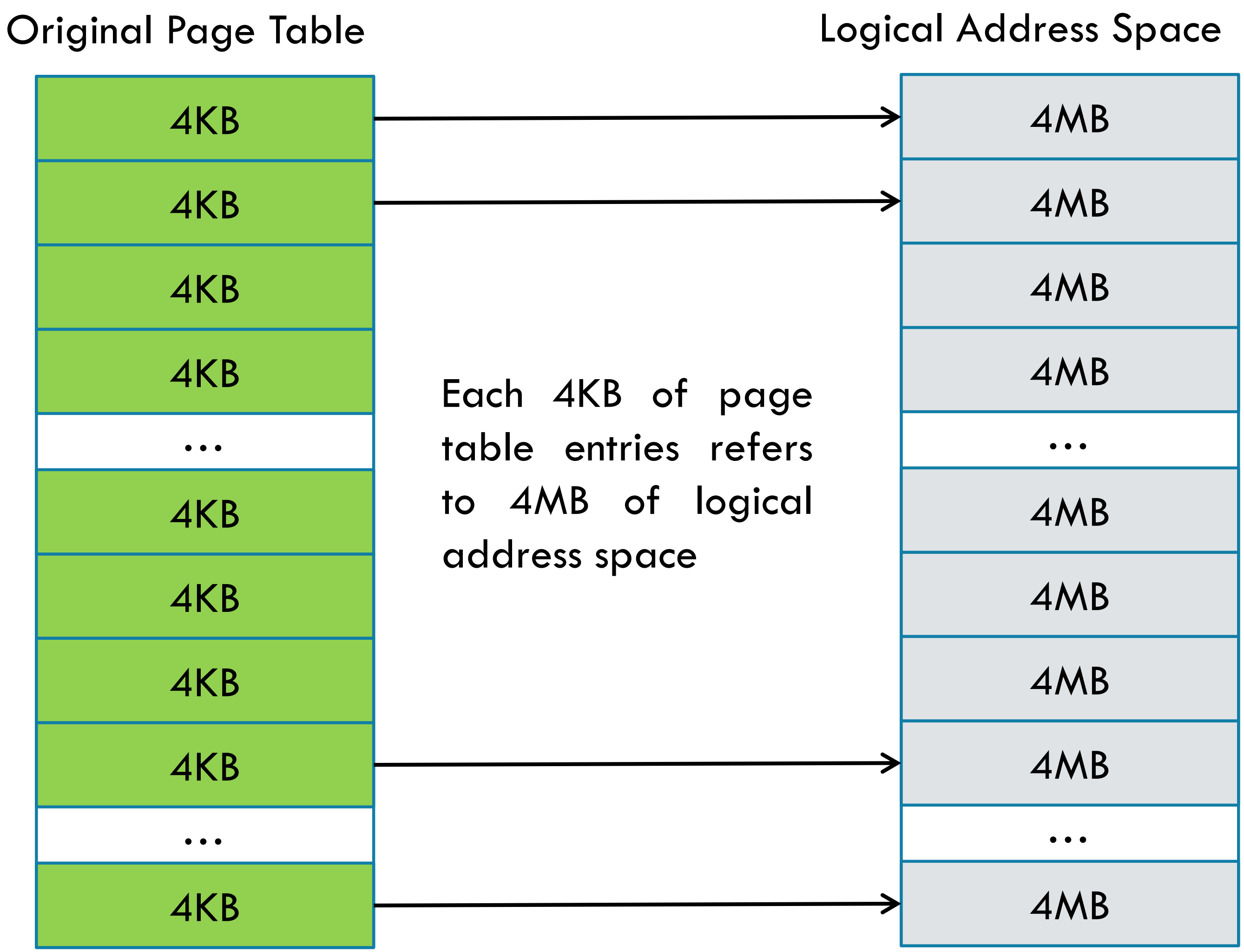


4KB of page table entries is basically $2^{10} = 1024$ page table entries.

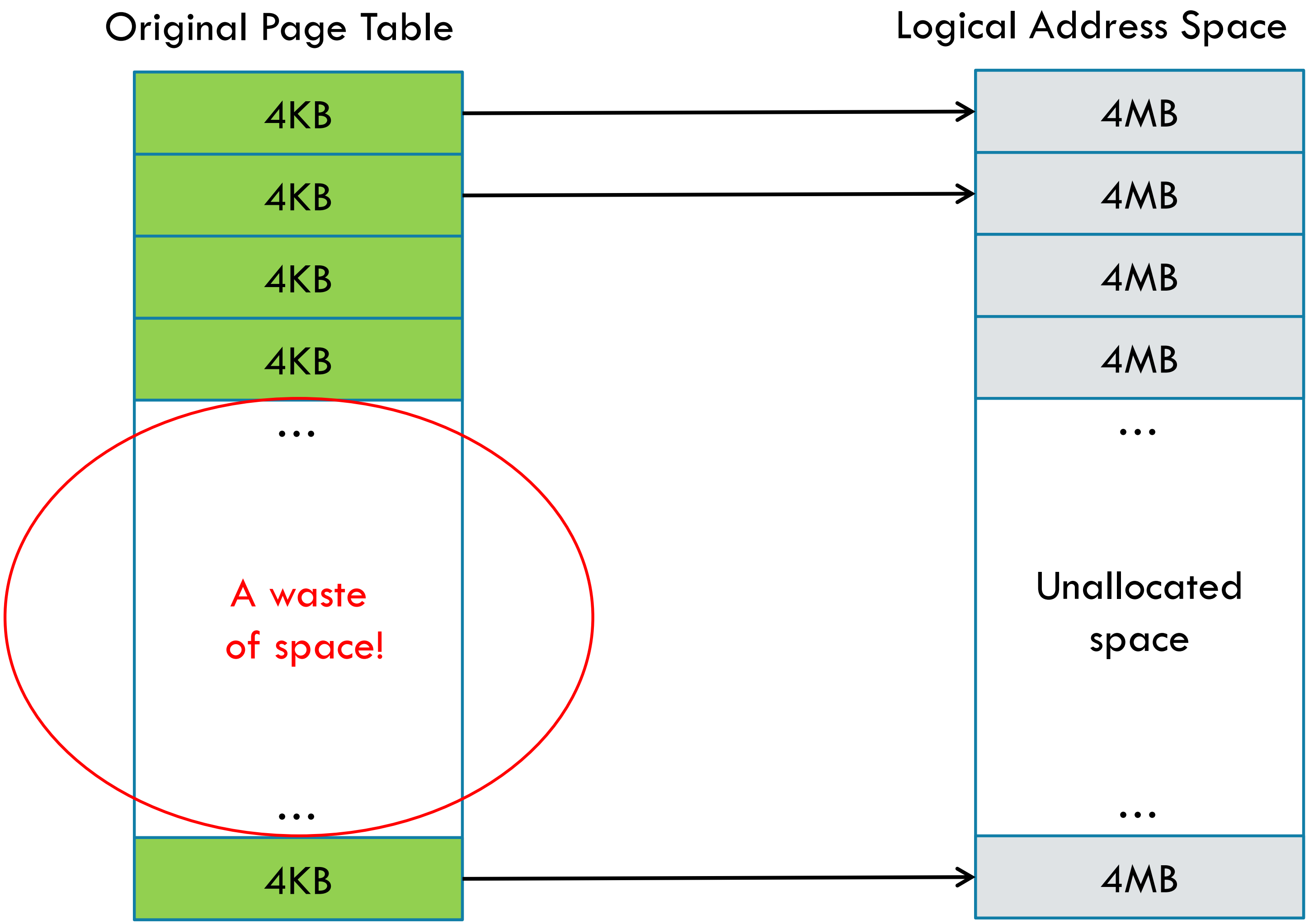
Recall that each table entry refers to a page size of 4K.

So 1024 page table entries refers to 4MB of logical memory.

RELATIONSHIP BETWEEN THE PAGE TABLE AND THE LOGICAL ADDRESS SPACE



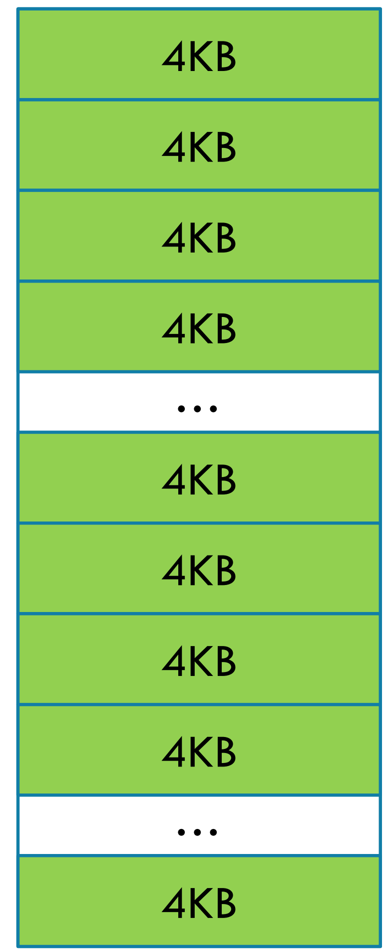
PROBLEM: NOT ALL LOGICAL ADDRESS SPACES ARE USED/ALLOCATED!



SOLUTION: 2 – LEVEL PAGING

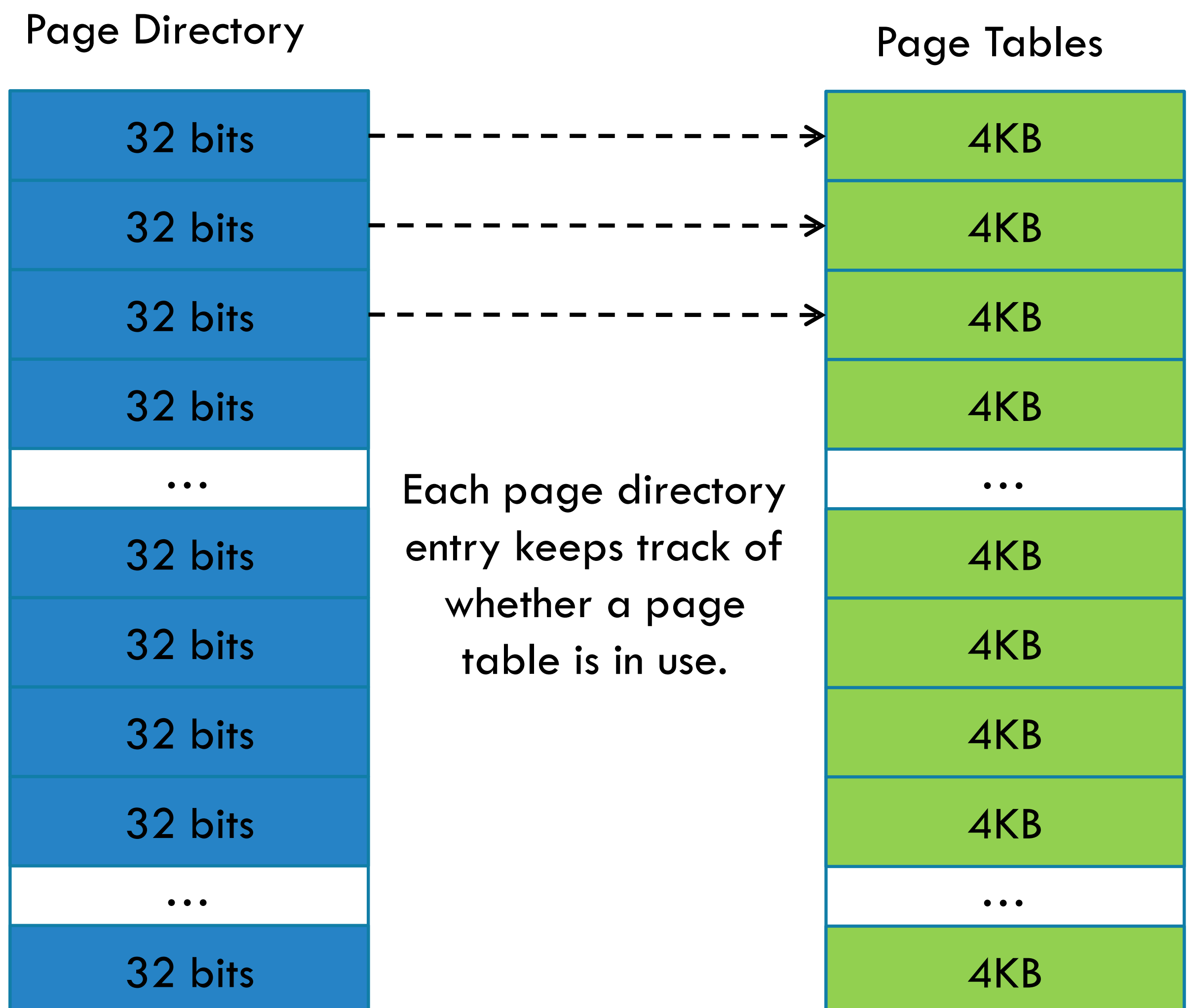
1. Break the page table into smaller tables of 4KB size each.
2. Original page table size = 4MB = 1K smaller page tables of 4KB each.

Original Page Table



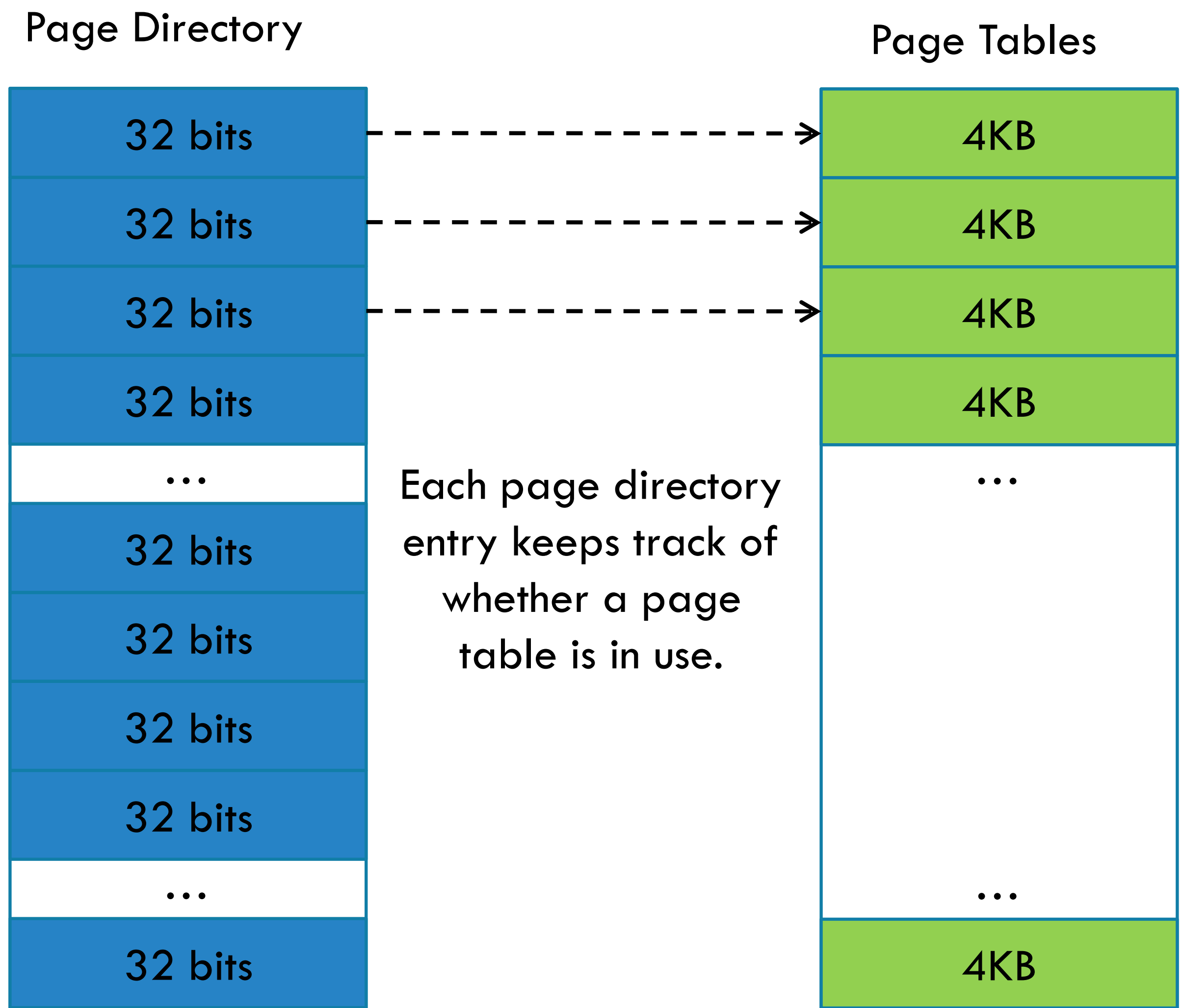
SOLUTION: 2 – LEVEL PAGING (VIRTUALIZE THE PAGE TABLE)

1. Break the page table into smaller tables of 4K size each.
2. Page table size = 4MB = 1K pages.
3. Use a page directory to keep track of page tables.

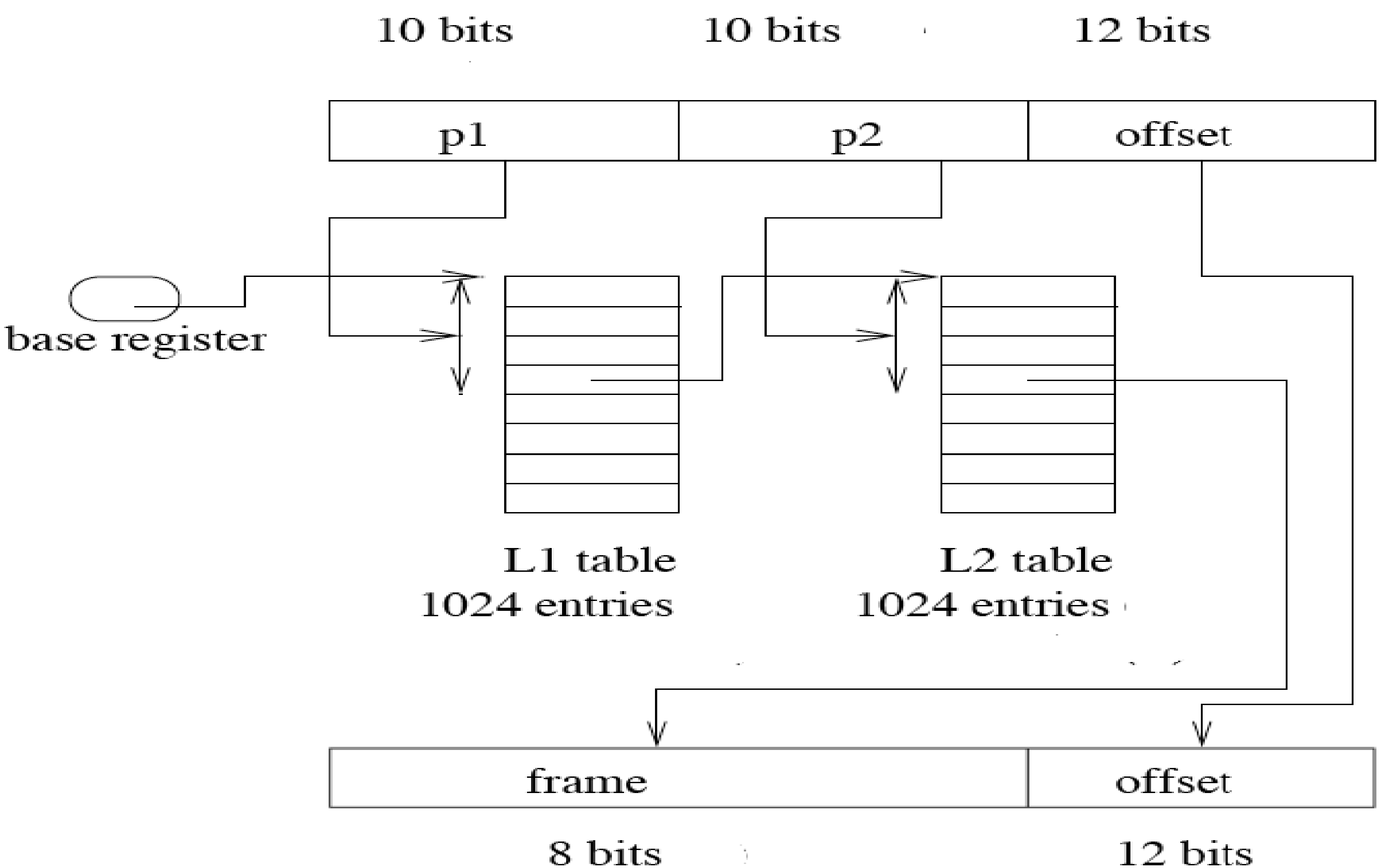


SOLUTION: 2 – LEVEL PAGING (VIRTUALIZE THE PAGE TABLE)

1. Break the page table into smaller tables of 4K size each.
2. Page table size = 4MB = 1K pages.
3. Use a page directory to keep track of page tables.

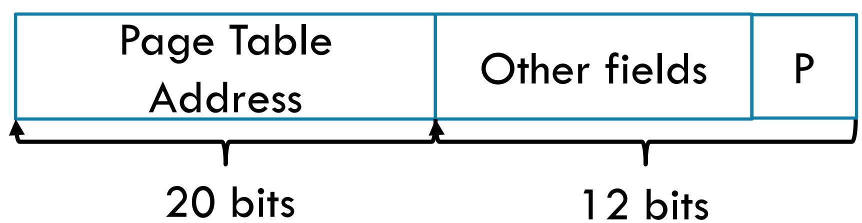


2-LEVEL PAGING VIEW: EXAMPLE



PAGE DIRECTORY ENTRY

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

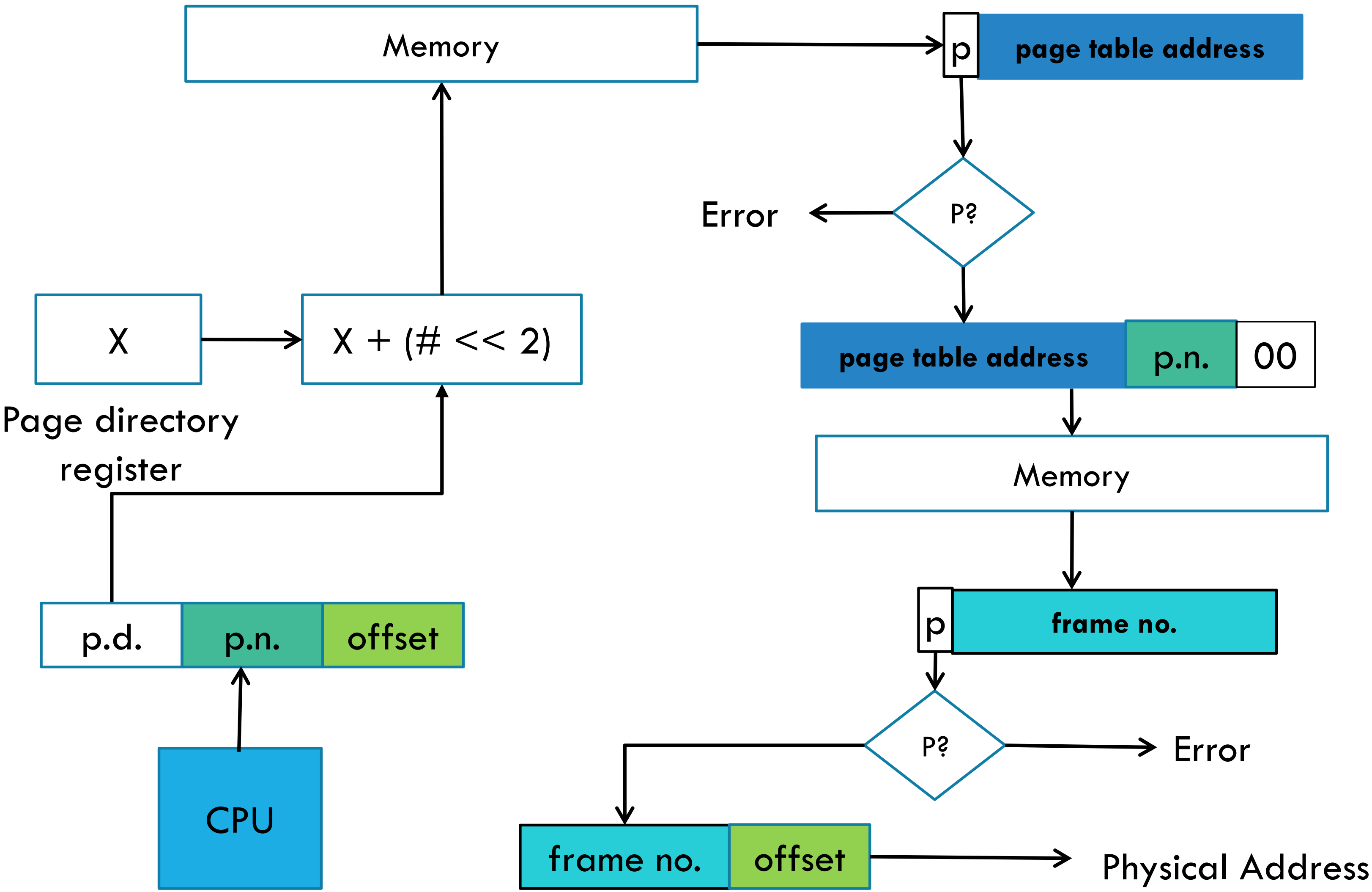


2-LEVEL PAGING LOGICAL ADDRESS

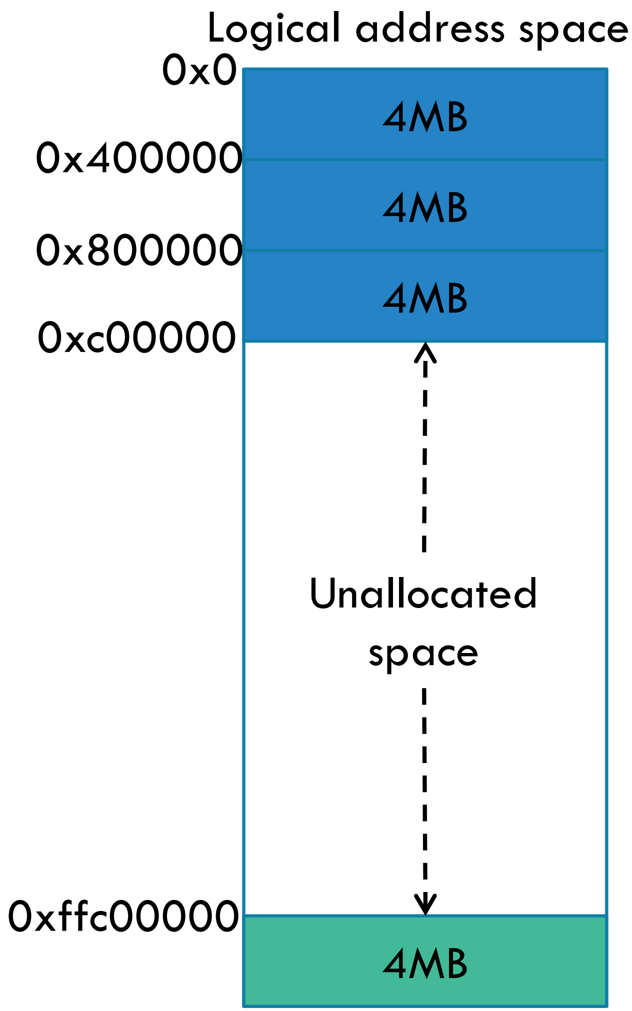
page directory number	page number	offset
1010 0101 10	10 0101 1010	0101 1010 0101

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

2-LEVEL PAGING LOGICAL ADDRESS TRANSLATION



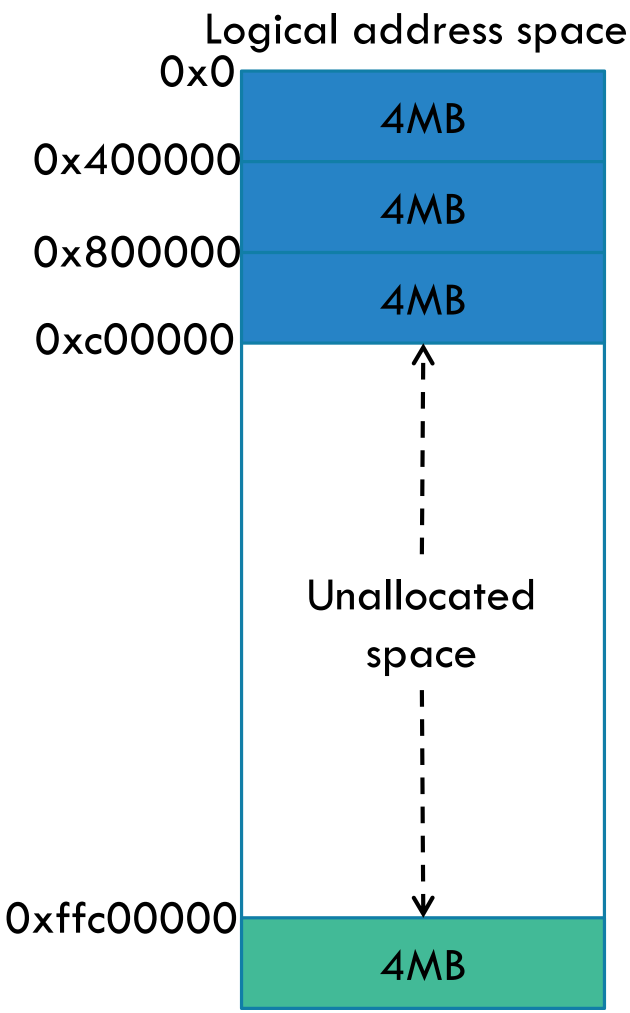
AN EXAMPLE FOR 2-LEVEL PAGING TO SHOW THE SIZE OF PAGE TABLES IN RAM



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 are un-allocated.

WHAT IF WE HAVE 1-LEVEL PAGING? - I



Consider the page numbers for these pages.

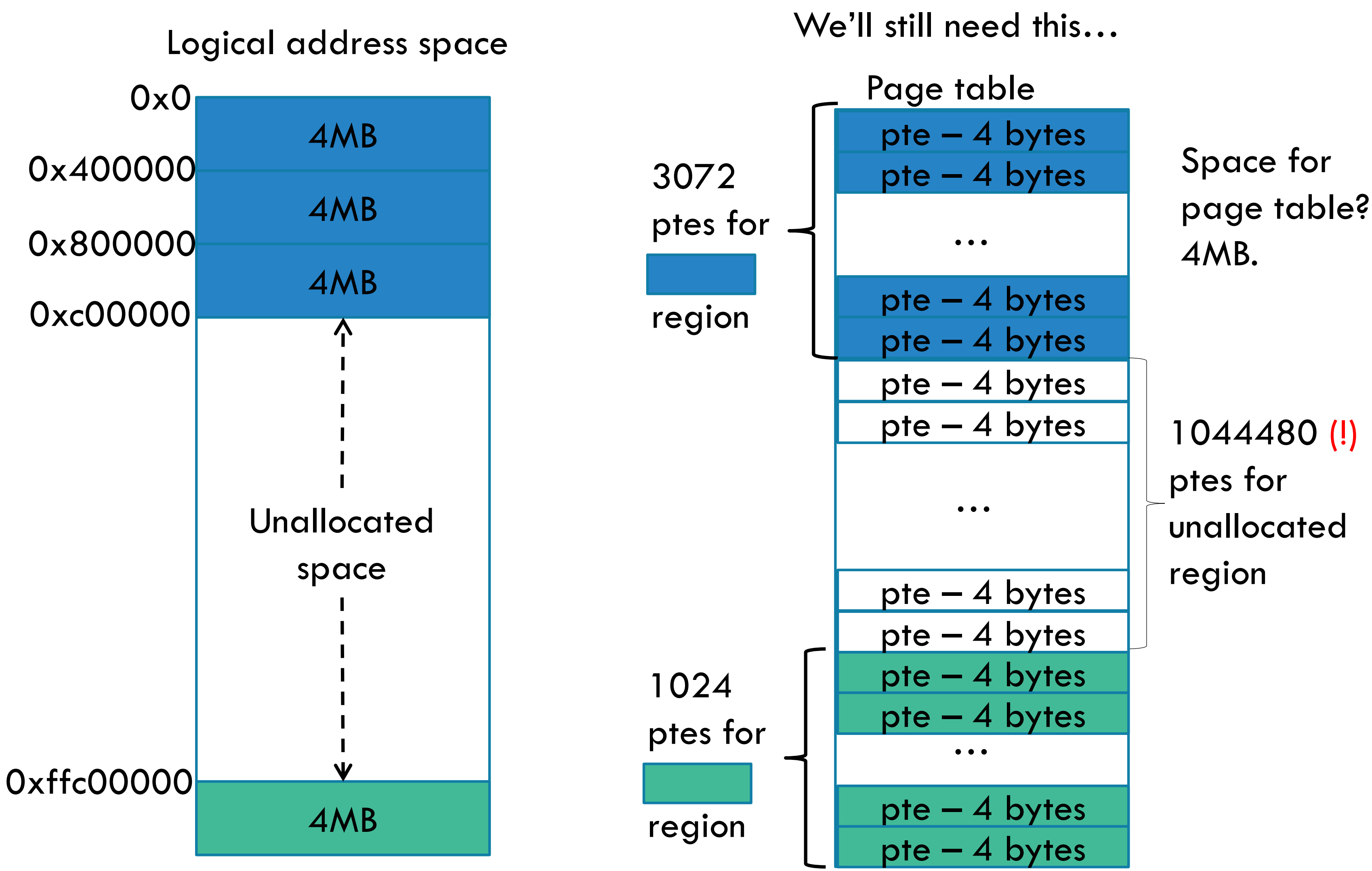
The region has addresses ranging from 0x0 to 0xbfffff.

Page numbers for these addresses range from 0x0 to 0xbff.

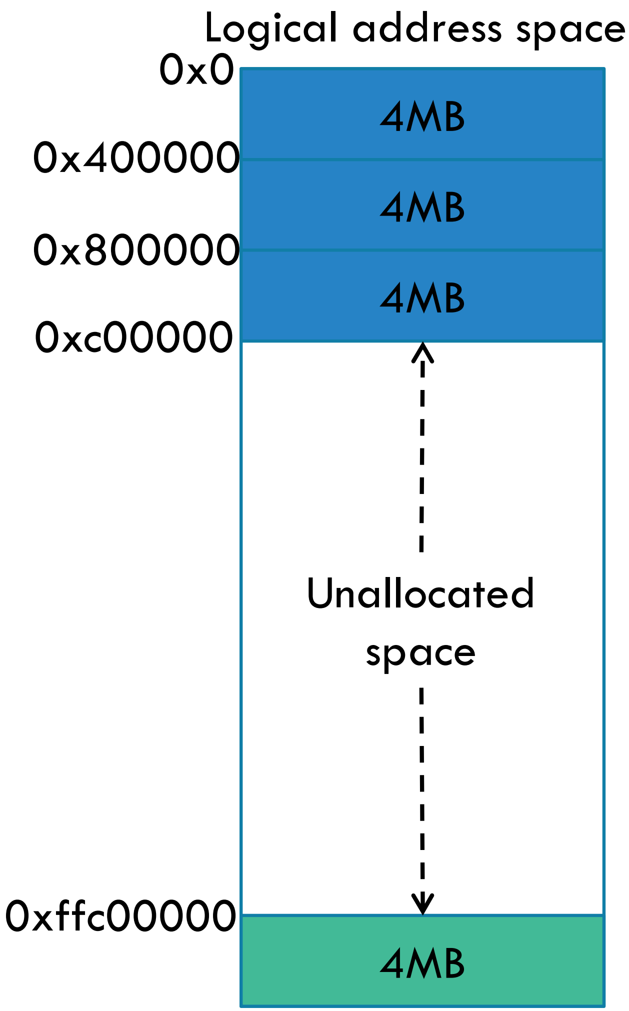
The region has addresses ranging from 0xffc00000 to 0xffffffff.

Page numbers for these addresses range from 0xffc00 to 0xfffff.

WHAT IF WE HAVE 1-LEVEL PAGING? - II



WHAT IF WE HAVE 2-LEVEL PAGING? - I



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

The region has addresses ranging from 0x00000000 to 0x00bfffff.

Page Directory numbers for these addresses range from 0x0 to 0x2.

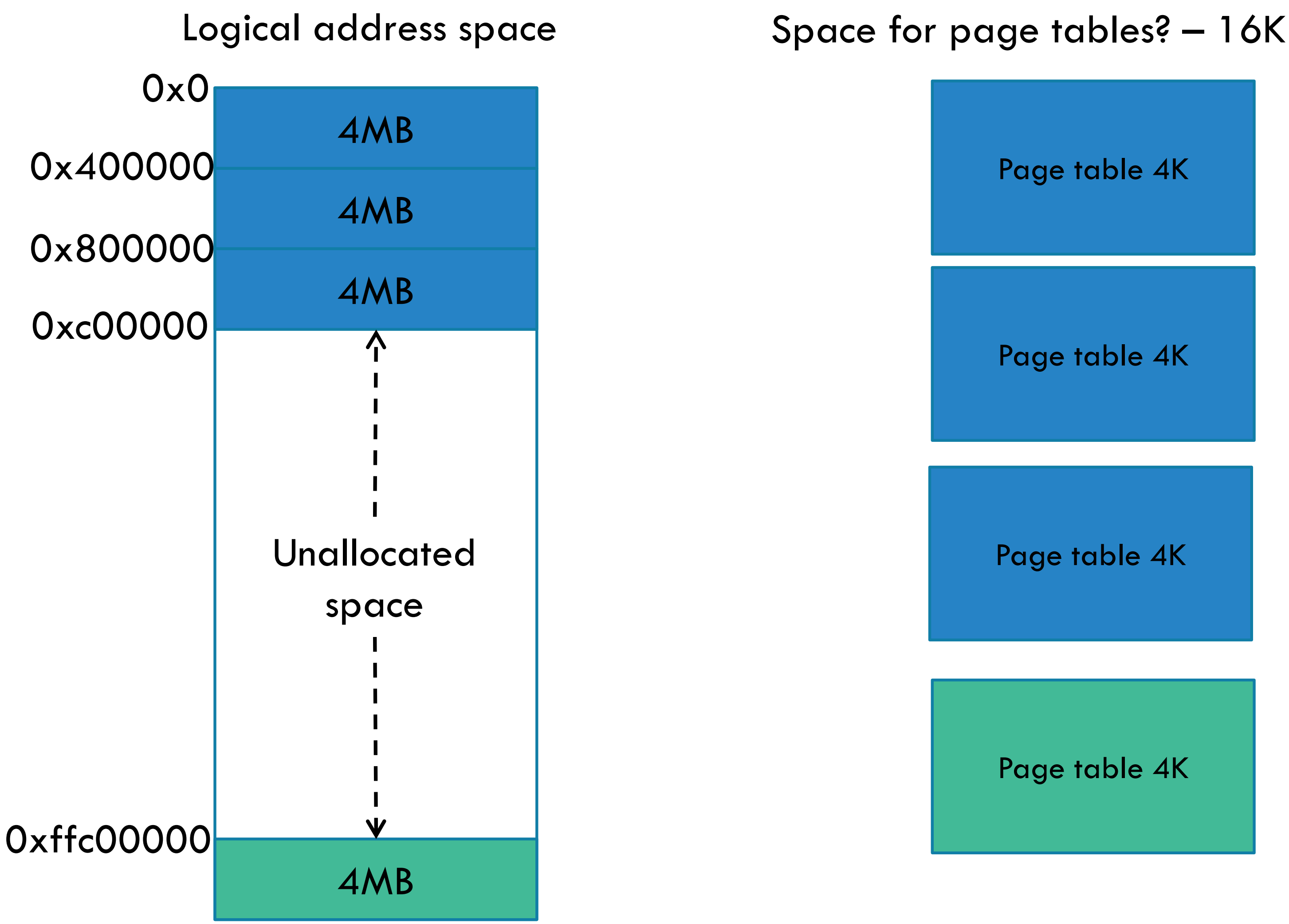
Page numbers for these addresses range from 0 to 0x3FF.

The region has addresses ranging from 0xffc00000 to 0xffffffff.

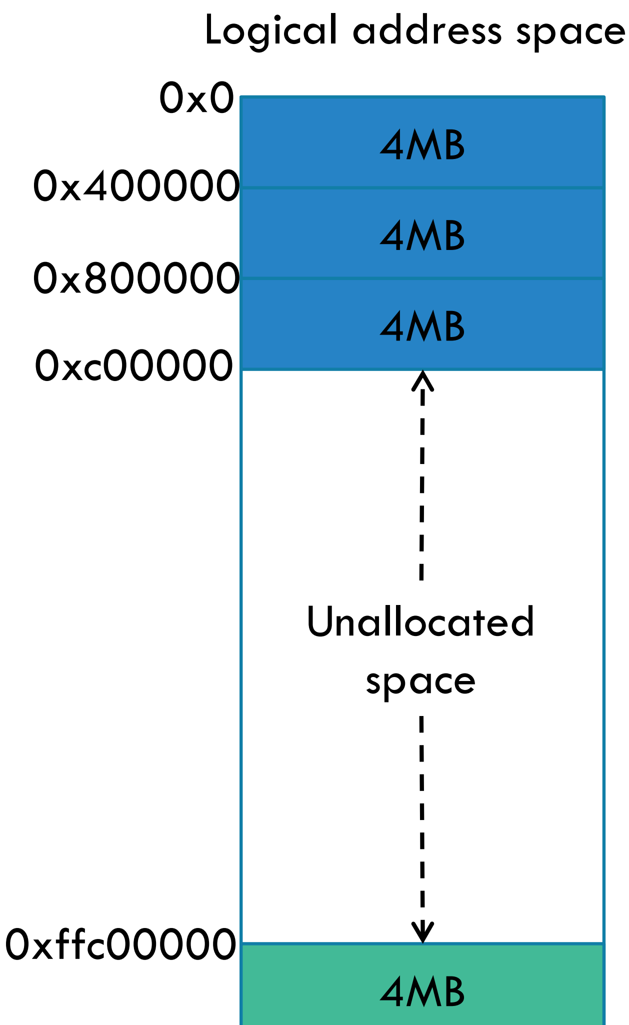
Page Directory numbers for these addresses range from 0x3FF to 0x3FF.

Page numbers for these addresses range from 0 to 0x3FF.

HOW ABOUT 2-LEVEL PAGING? – III



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
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 single number
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.

HASHED PAGE TABLES

Common in 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 (1) the virtual page number (2) the value of the mapped page frame (3) a pointer to the next element

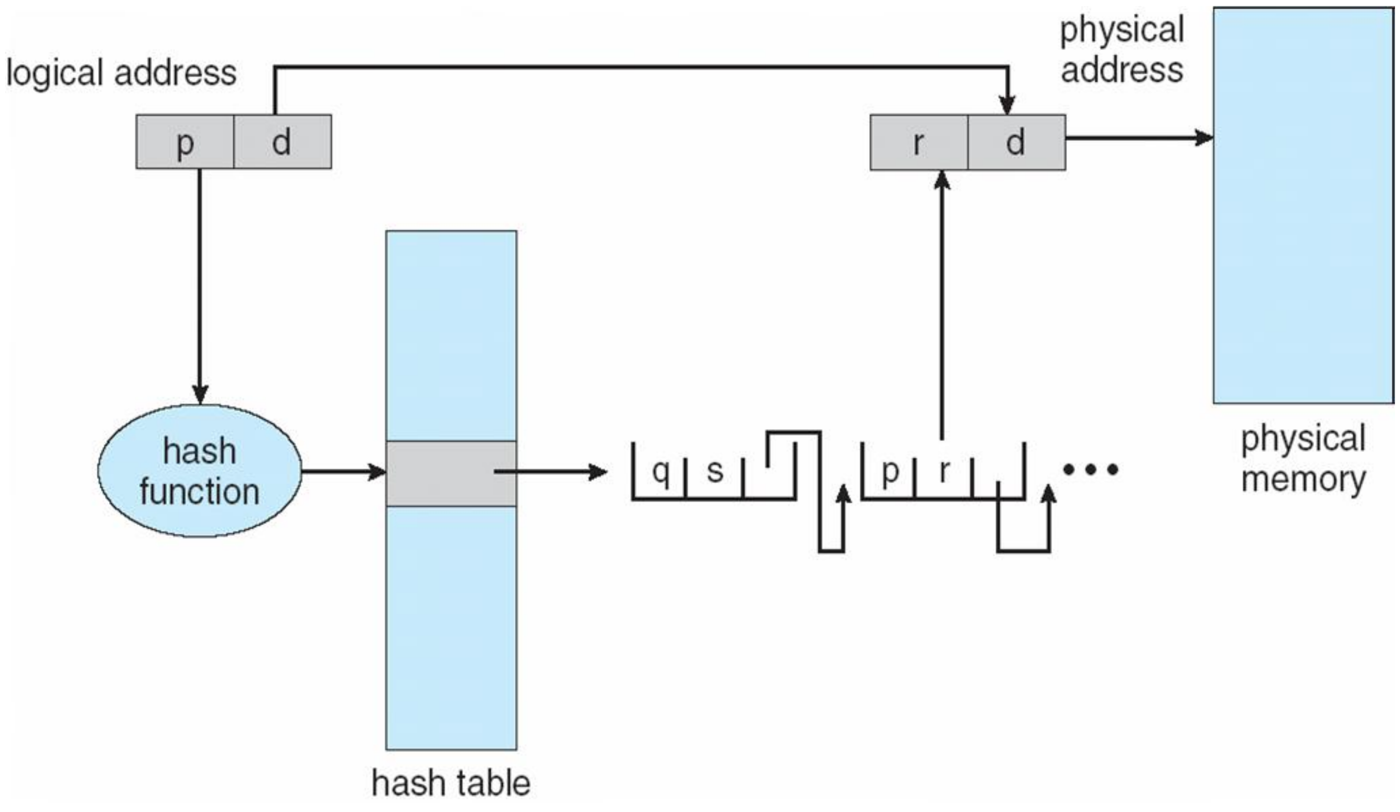
Virtual page numbers are compared in this chain searching for a match

- If a match is found, the corresponding physical frame is extracted

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)

HASHED PAGE TABLE



INVERTED PAGE TABLE

Rather than each process having a page table and keeping track of all possible logical pages, track all physical pages

One entry for each real page of memory

Entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns that page

Decreases memory needed to store each page table, but increases time needed to search the table when a page reference occurs

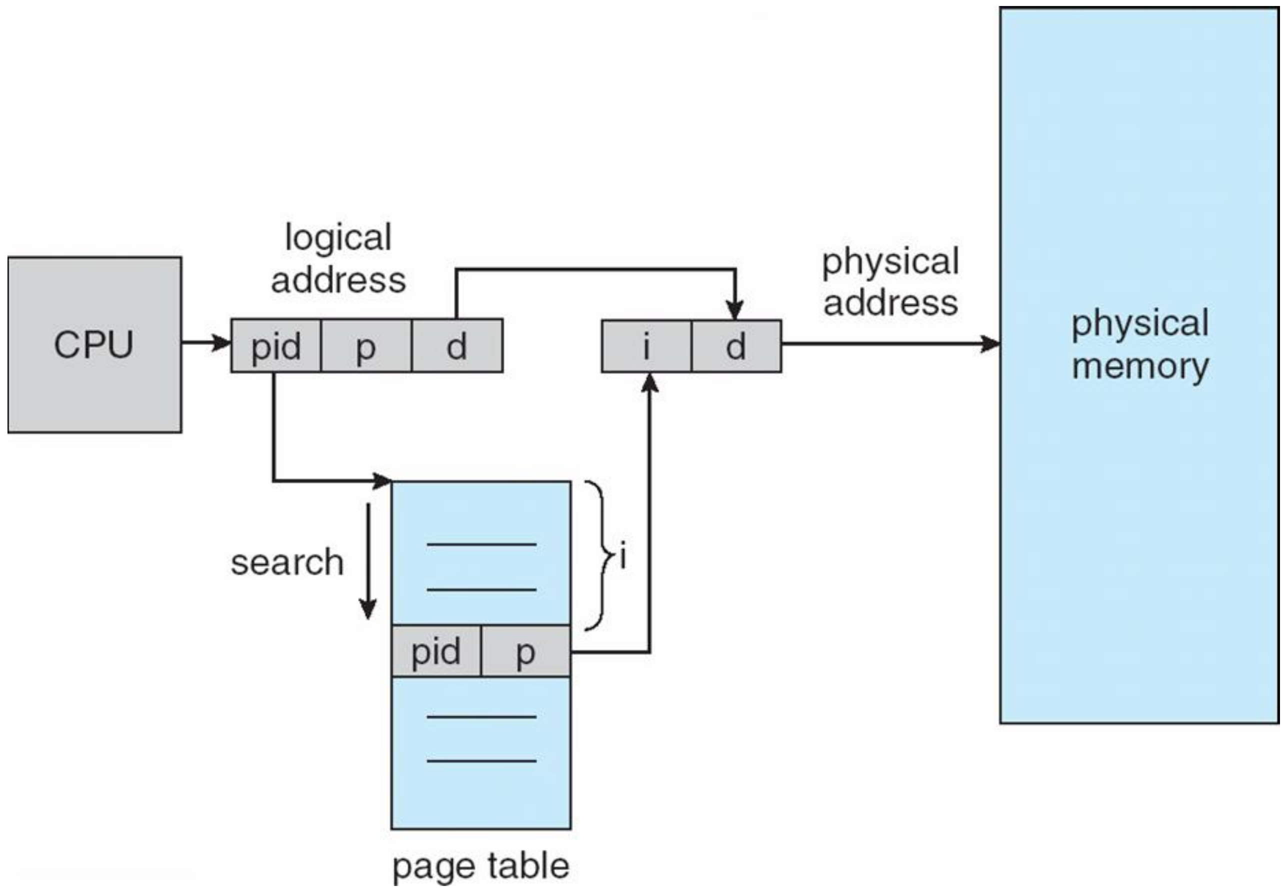
Use hash table to limit the search to one — or at most a few — page-table entries

- TLB can accelerate access

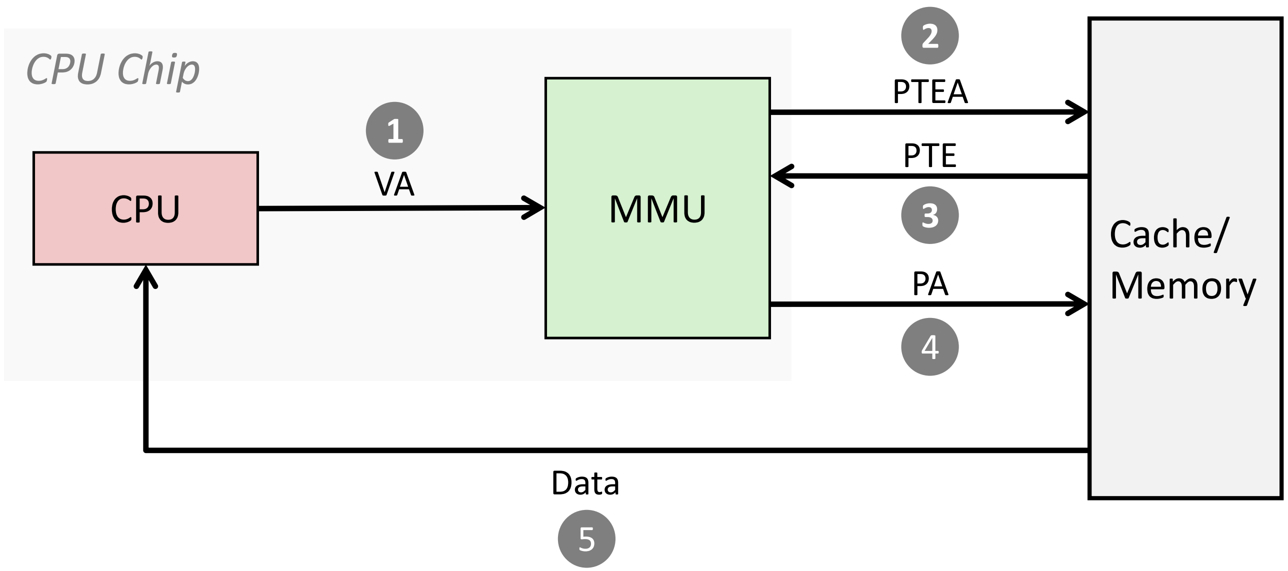
But how to implement shared memory?

- One mapping of a virtual address to the shared physical address

INVERTED PAGE TABLE ARCHITECTURE



ADDRESS TRANSLATION: PAGE HIT



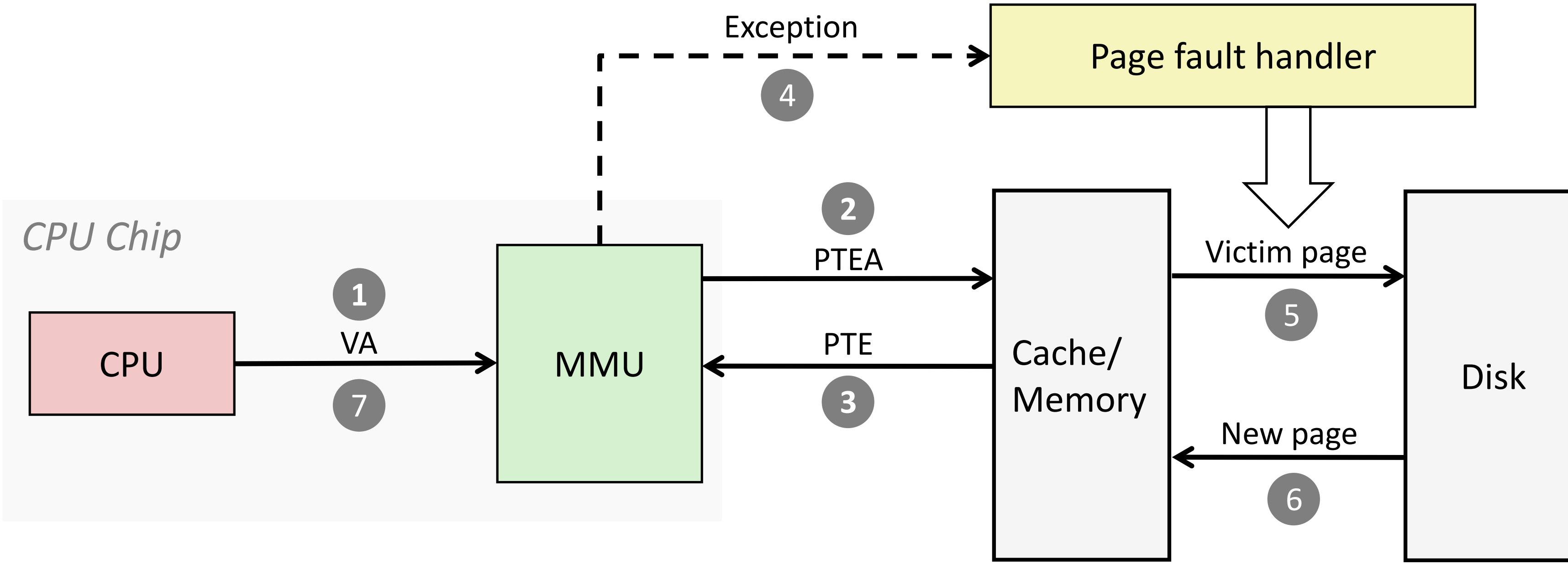
1) Processor sends virtual address to MMU

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

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

ADDRESS TRANSLATION: PAGE FAULT



1) Processor sends virtual address to MMU

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

4) Valid bit is zero, so MMU triggers page fault exception

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

6) Handler pages in new page and updates PTE in memory

7) Handler returns to original process, restarting faulting instruction

TRANSLATION LOOKASIDE BUFFER

- Cache of page numbers to physical addresses
- Speeds up the translation process
- However, a miss will still incur extra memory lookups to find the physical address.
- The TLB resides within the MMU.
- Flushed with every context switch.

VIRTUAL MEMORY SPACE

- We allow the logical memory size to exceed physical memory size
- Pages and frames are added to the page table as needed
- When physical memory has been exceeded, frames are written (temporarily) to the hard disk to free up physical memory (wrong! Not all the time)
- *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 pages of physical memory
- **Paging –** a page of physical memory is written to disk
  - Frees up a single page of physical memory
  - Several contiguous pages 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 memory within that page is accessed
- Each entry 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

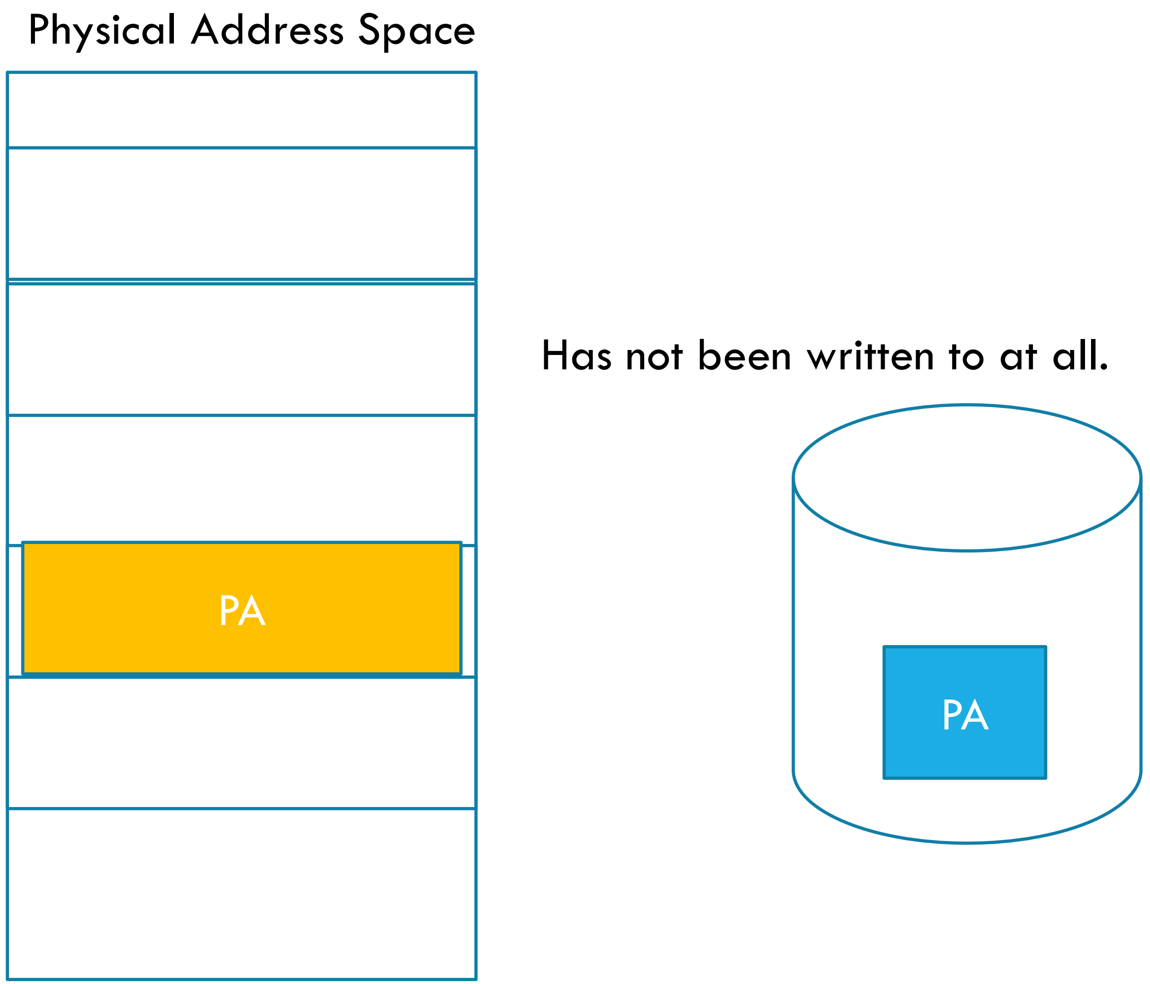
DEMAND PAGING (2)

- When a location in logical memory is accessed, the valid/invalid bit of corresponding page 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 read in
- 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 needs to be changed
- Dirty bit indicates whether modified

OPTIMIZATION OF PAGE SWAPPING



ANALYZING PAGE REPLACEMENT ALGORITHMS

- Page reference sequence
  - A series of page numbers.
  - 1, 0, 7, 1, 0, 2, 1, 2, 3, 0, 3, 2, 4, 0, 3, 0, 2, 1
  - This is a series of page numbers being requested.
- For this particular page ref seq,
  - What is the number of page faults?
  - Aim of any page replacement algo 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 as the victim frame
- Easy to implement using a queue
- Often does not yield optimal performance: more than the minimum number of page faults will occur

FIFO ANALYSIS (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								
Frame 2		0	0	0	0	0	1	1	1	1								
Frame 3			7	7	7	7	7	7	3	3								
Page Fault?	Y	Y	Y	N	N	Y	Y	N	Y	Y								

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 number of frames?
  - Expect Less page faults with more frames.
- Belady's anomaly
  - For a particular replacement algo and page reference sequence, we have more page faults when we have more frames.

FIFO DISPLAYS BELADY'S ANOMALY

3 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

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

FIFO PAGE REPLACEMENT (2)

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

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

- The queue states are

fault, [7], fault, [7,0], fault, [7,0,1], fault, [0,1,2], fault, [1,2,3], fault, [2,3,0], fault,

[3,0,4], fault, [0,4,2]

- Total of 8 page faults

SECOND CHANCE ALGORITHM (1)

- Modification of the FIFO algorithm
- Each page in memory has a **reference bit** associated to 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 set to 0, and the next item in the queue is selected *in circular queue fashion*

SECOND CHANCE ALGORITHM (2)

- 3 frames of physical memory (as before)
- These pages are accessed (as before):

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

- The queue states are (+ = 1, - = 0):

  fault, [7+], fault, [7+,0+], fault,

  [7+,0+,1+], fault, [0-,1-,2+], [0+,1-,2+],

  fault, [2+,0-,3+], [2+,0+,3+], fault,

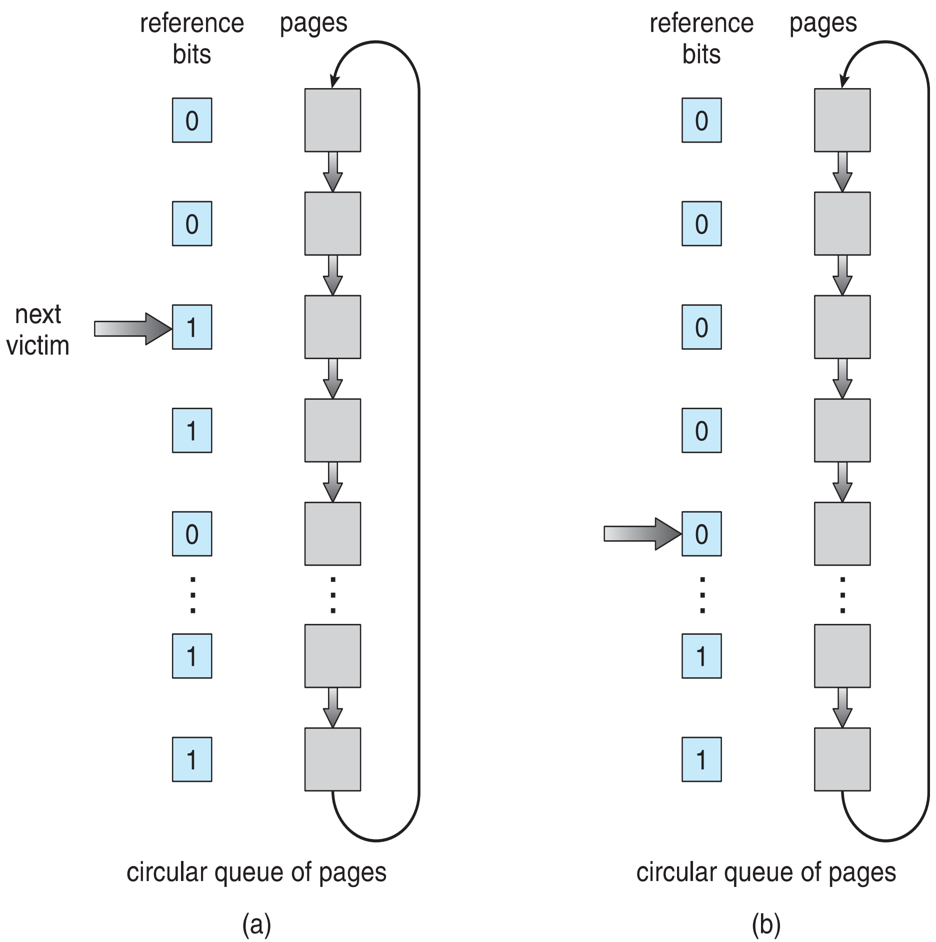
  [0-,3-,4+], [0+,3-,4+], fault, [4+,0-,2+]

- Total of 7 page faults

SECOND CHANCE ANALYSIS (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 +								
Frame 2		0 +	+ 0	+ 0	+ 0	0 -	1 +	1 +	1 +	1 -								
Frame 3			7 +	7 +	7 +	7-	7-	7 -	3 +	3 -								
Page Fault?																		

SECOND-CHANCE (CLOCK) PAGE-REPLACEMENT ALGORITHM



LRU ALGORITHM (1)

- **Least recently used algorithm** – the page that has not been used for the longest period of time is selected as victim frame
- Implemented in one of two ways
  - Time stamp is used - whenever a page is referenced, it is marked with the time
    - The victim frame is the page with the smallest time stamp 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 ANALYSIS (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									
Frame 2		0	0	0	0	0	0	0	3									
Frame 3			7	7	7	2	2	2	2									
Page Fault?	Y	Y	Y			Y			Y									

LRU ALGORITHM (2)

- 3 frames of physical memory (as before)
- These pages are accessed (as before):

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

- Queues states ((#) = time stamp):

fault, [7(0)], fault, [7(0),0(1)], fault,

[7(0),0(1),1(2)], fault, [2(3),0(1),1(2)],

[2(3),0(4),1(2)], fault, [2(3),0(4),3(5)],

[2(3),0(6),3(5)], [2(7),0(6),3(5)], [2(7),0(8),3(5)], fault [2(7),0(8),4(9)],

[2(7),0(10),4(9)], [2(11),0(10),4(9)]

- Total of 6 page faults

OPTIMAL PAGE REPLACEMENT

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		
Frame 2		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		
Page Fault?	Y	Y	Y	N	N	Y	N	N	Y	N	N	N	Y	N	N	N		

Not used in the future!

ENHANCED SECOND-CHANCE ALGORITHM

Improve algorithm by using reference bit and modify bit (if available) in concert

Take ordered pair (reference, modify)

1. (0, 0) neither recently used not 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 called for, use the clock scheme but use the four classes replace page in lowest non-empty class

- Might need to search circular queue several times

THRASHING

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

AVOIDING PAGE FAULTS (1)

- 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

AVOIDING PAGE FAULTS (2)

- When doing two-dimensional array computations, process the array elements by rows
- If you have 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