CS 3502
Operating Systems

Memory Management

Kun Suo

Computer Science, Kennesaw State University

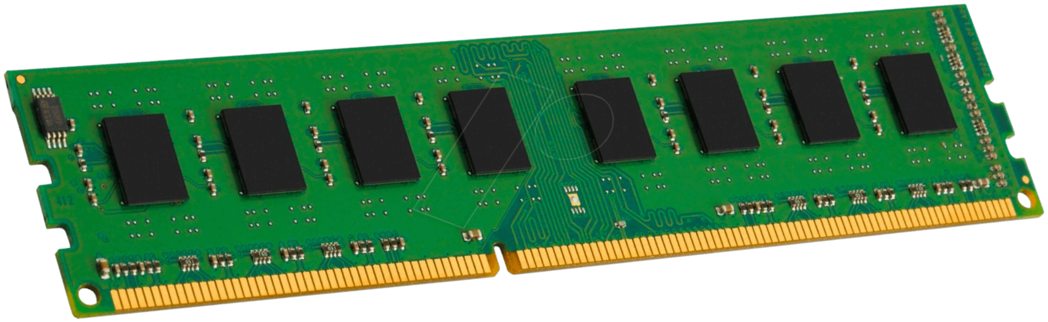
https://kevinsuo.github.io/

Outline

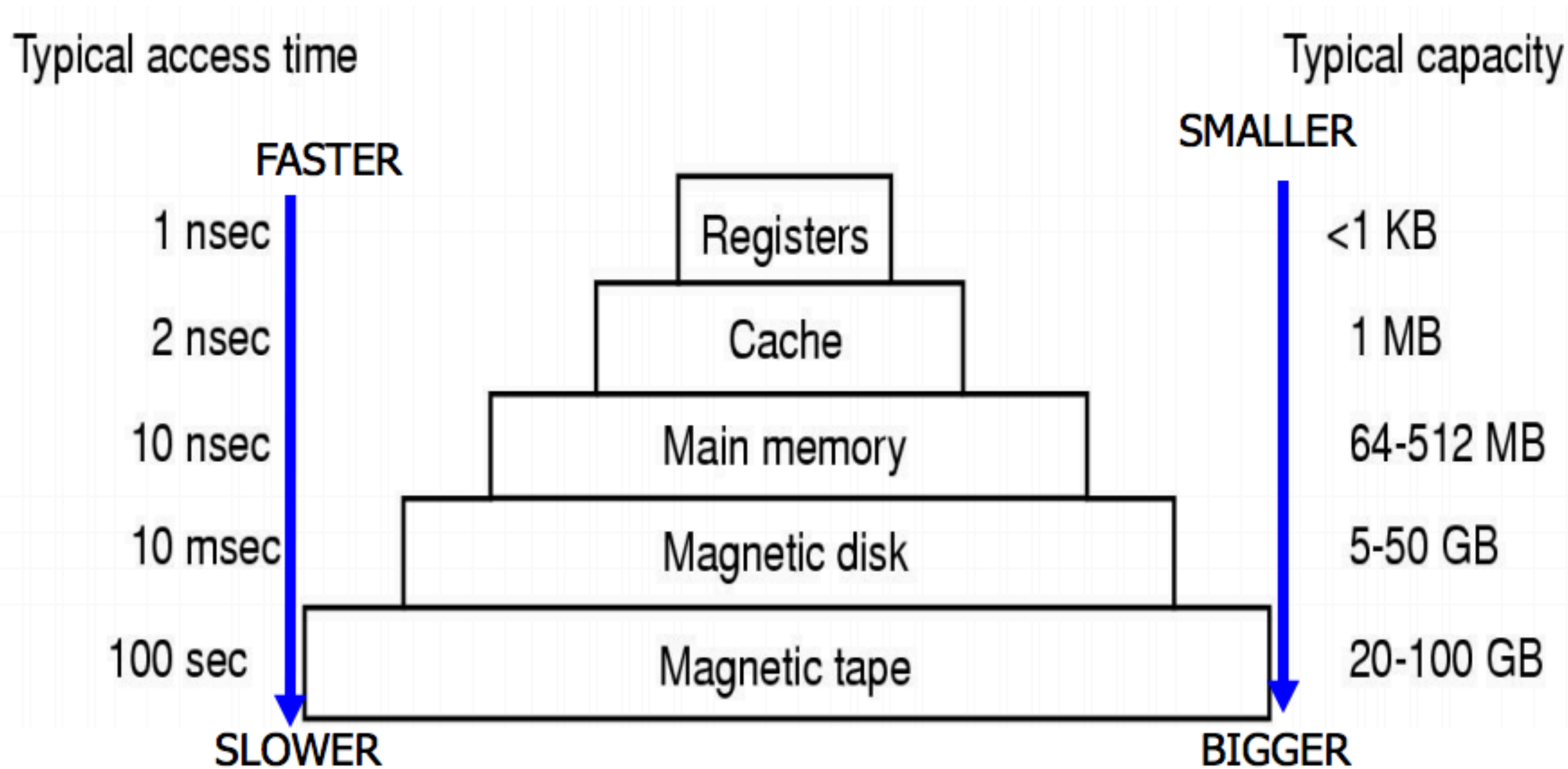
- Memory management overview
  - Memory abstraction and address spaces
  - Physical address and virtual address
  - Physical memory and virtual memory
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table

Why we need memory management

- Ideally user/programmers want memory that is
  - large --> 1GB, 2GB, 4GB, 16GB, …
  - fast --> DDR 2, DDR 3, DDR 4, …
  - non volatile --> not lose data when losing power



Typical Memory Hierarchy



Typical Memory Hierarchy

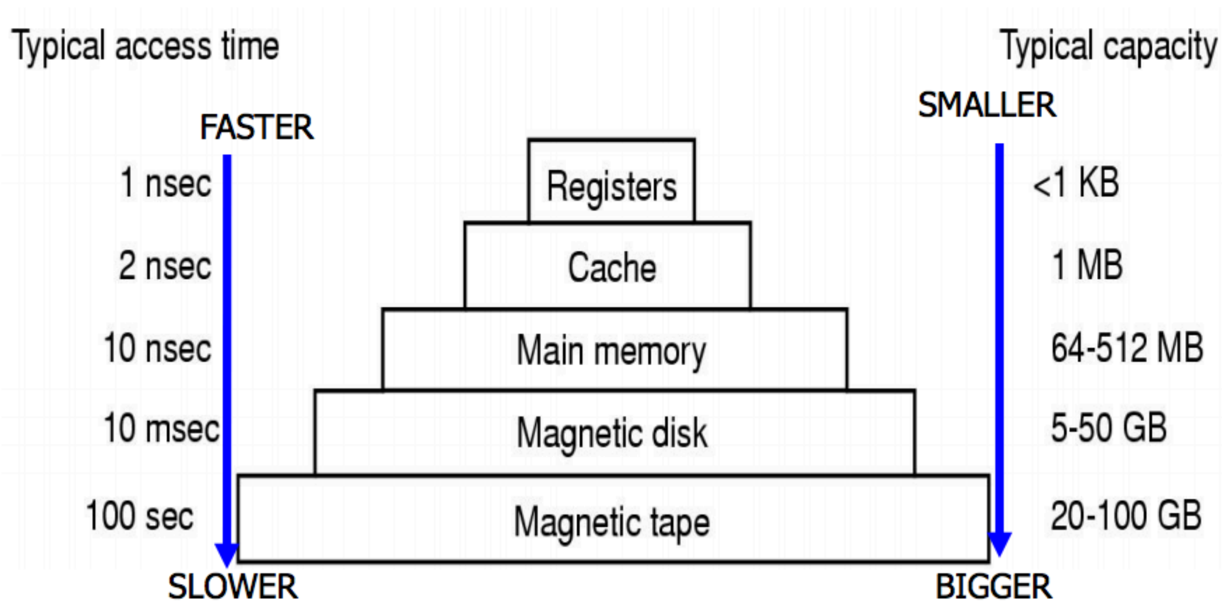
System Event	Actual Latency	Scaled Latency
One CPU cycle	0.4 ns	**1 s**
Level 1 cache access	0.9 ns	**2 s**
Level 2 cache access	2.8 ns	**7 s**
Level 3 cache access	28 ns	**1 min**
Main memory access (DDR DIMM)	~100 ns	**4 min**
Intel® Optane™ DC persistent memory access	~350 ns	**15 min**
Intel® Optane™ DC SSD I/O	<10 µs	**7 hrs**
NVMe SSD I/O	~25 µs	**17 hrs**
SSD I/O	50–150 µs	**1.5–4 days**
Rotational disk I/O	1–10 ms	**1–9 months**
Internet call: San Francisco to New York City	65 ms	**5 years**
Internet call: San Francisco to Hong Kong	141 ms	**11 years**

1000x than memory

20x than SSD

www.brendangregg.com/sysperfbook.html

Why we need memory hierarchy like this?



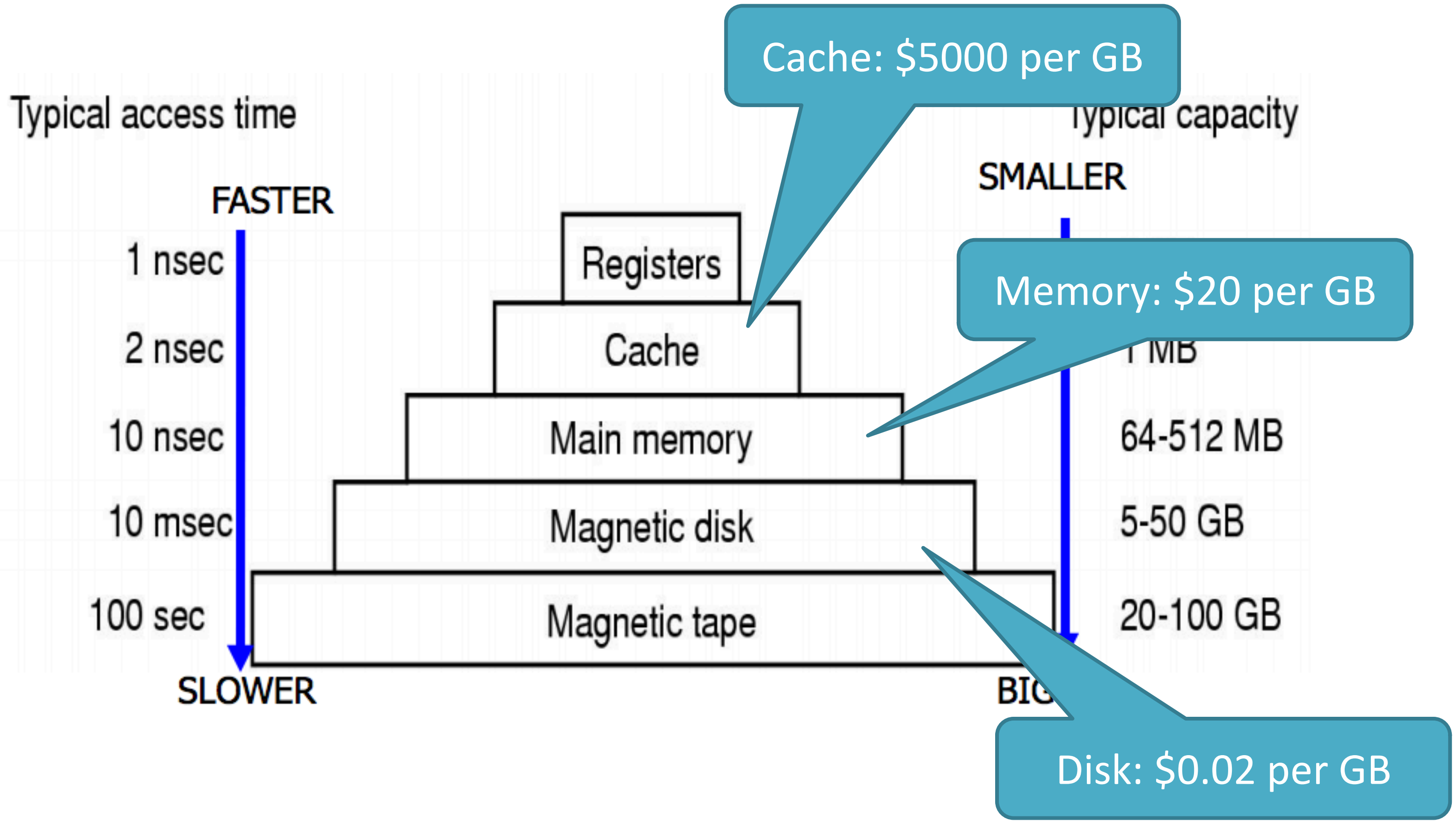
Single layer memory hierarchy?

Memory

Why we need memory hierarchy like this?

- Cost

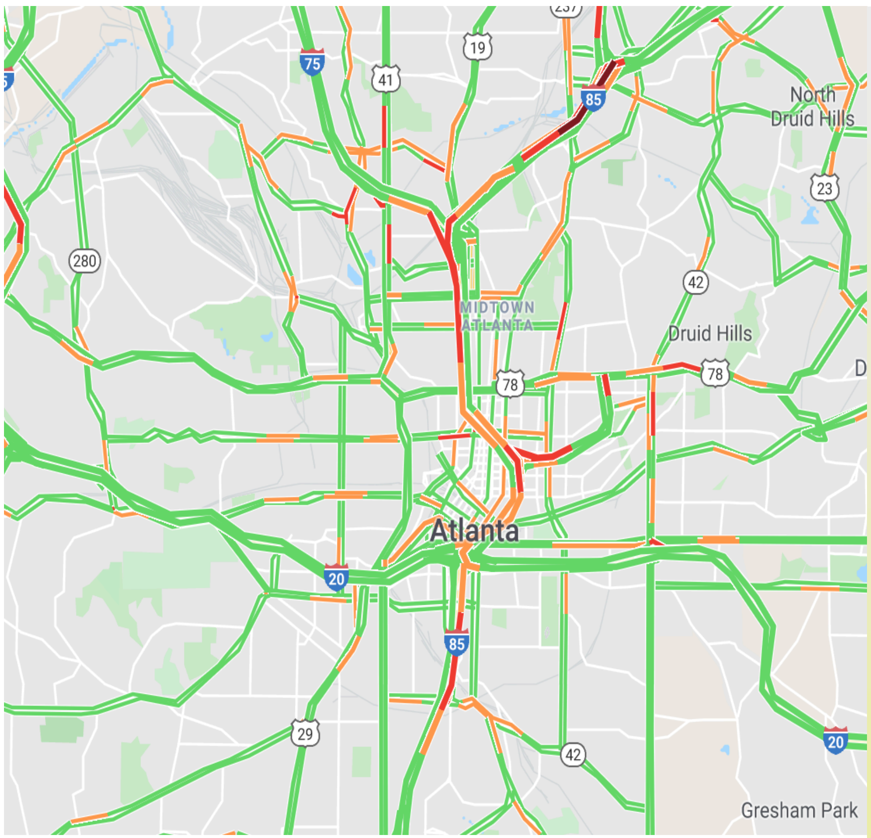
Year: 2010 data

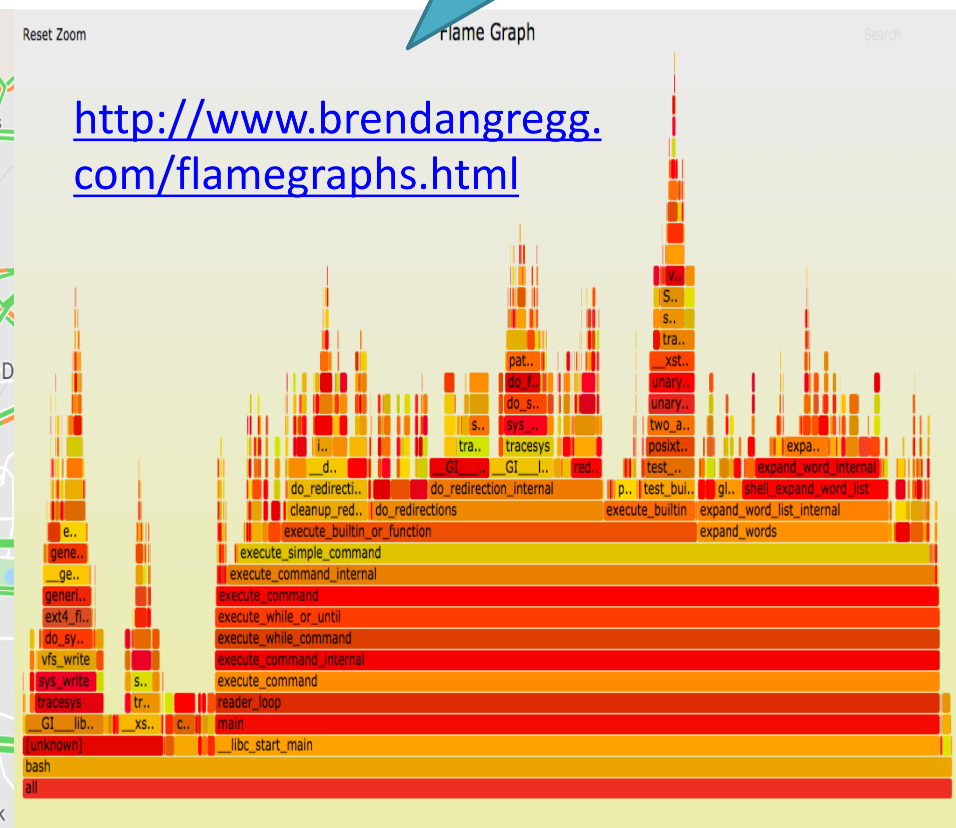


Why we need memory hierarchy like this?

- 20/80 rule: only a small proportion of code/path is hot in programs

We only need to put the hot code inside the top hierarchy of memory





http://www.brendangregg.com/flamegraphs.html

Why we need memory management

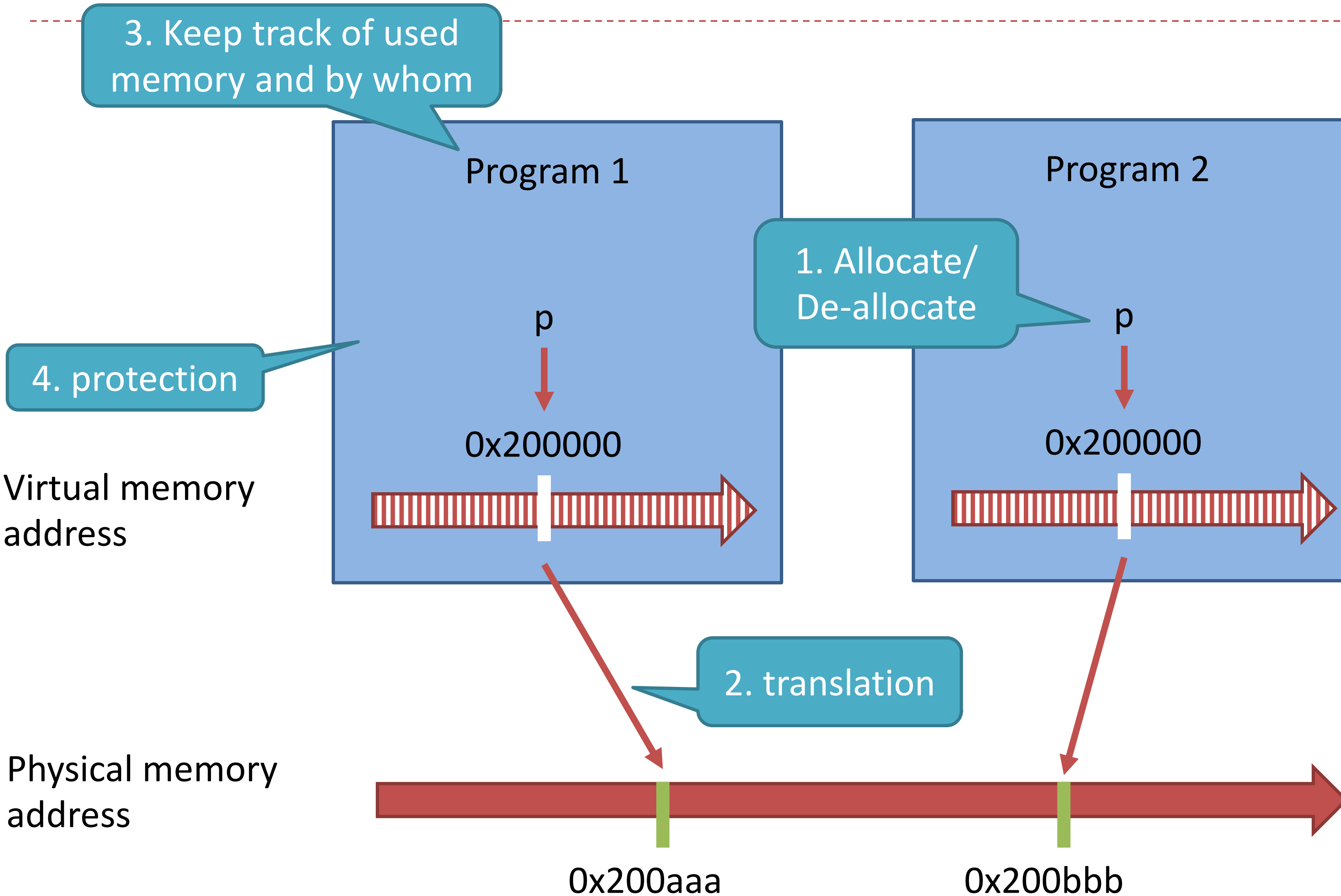
Ideal: programmers/users want memory that is	Reality: Memory hierarchy
large	small amount of fast, expensive memory e.g., register, cache
fast	some medium-speed, medium price main memory, lots of slow, cheap disk storage
Non-volatile	Volatile (e.g., top hierarchy)

Memory management handles the sematic gap

What will memory management do?

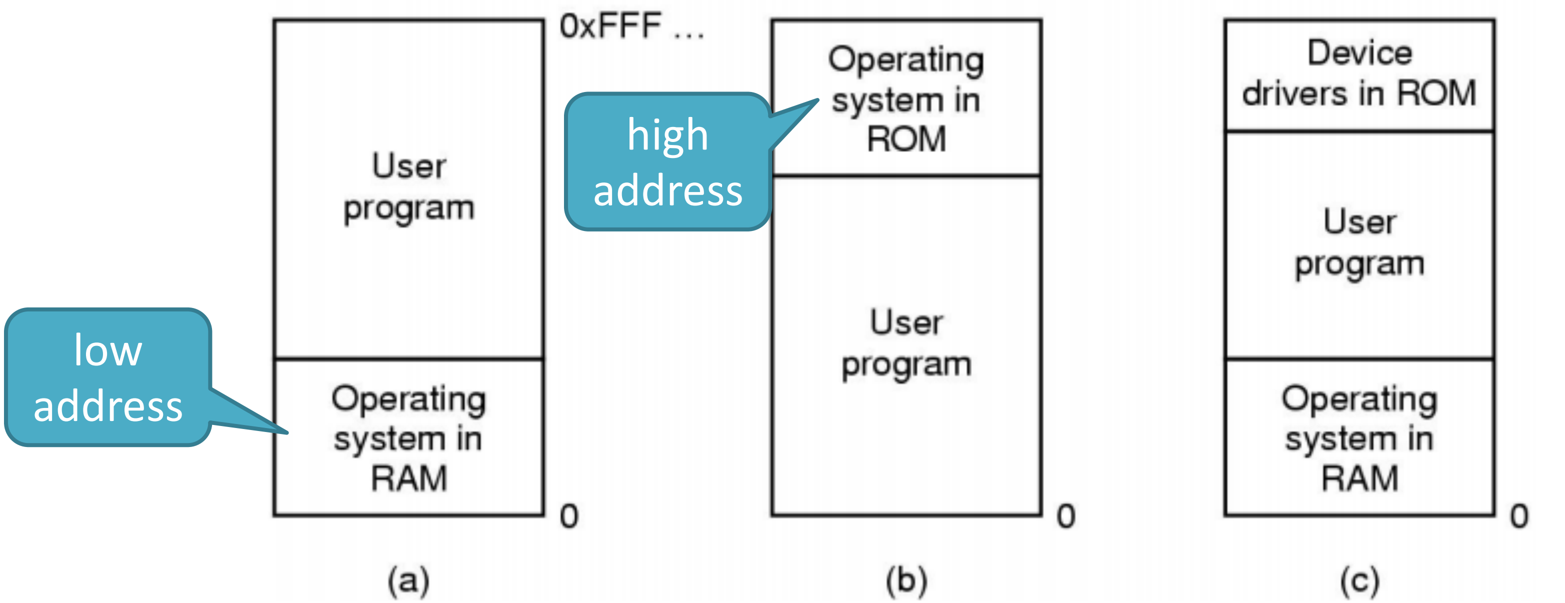
- Memory management key tasks:
  1. Allocate and de-allocate memory for processes (performed by OS and improve the programming efficiency)
  2. Address translation (e.g., between physical address and virtual address)
  3. Keep track of used memory size and used by whom
  4. Memory protection

These key tasks are realized by memory abstraction



Without memory abstraction (early era)

Memory management is simple. Put small piece for OS, the rest for apps.



Three simple ways of organizing memory:

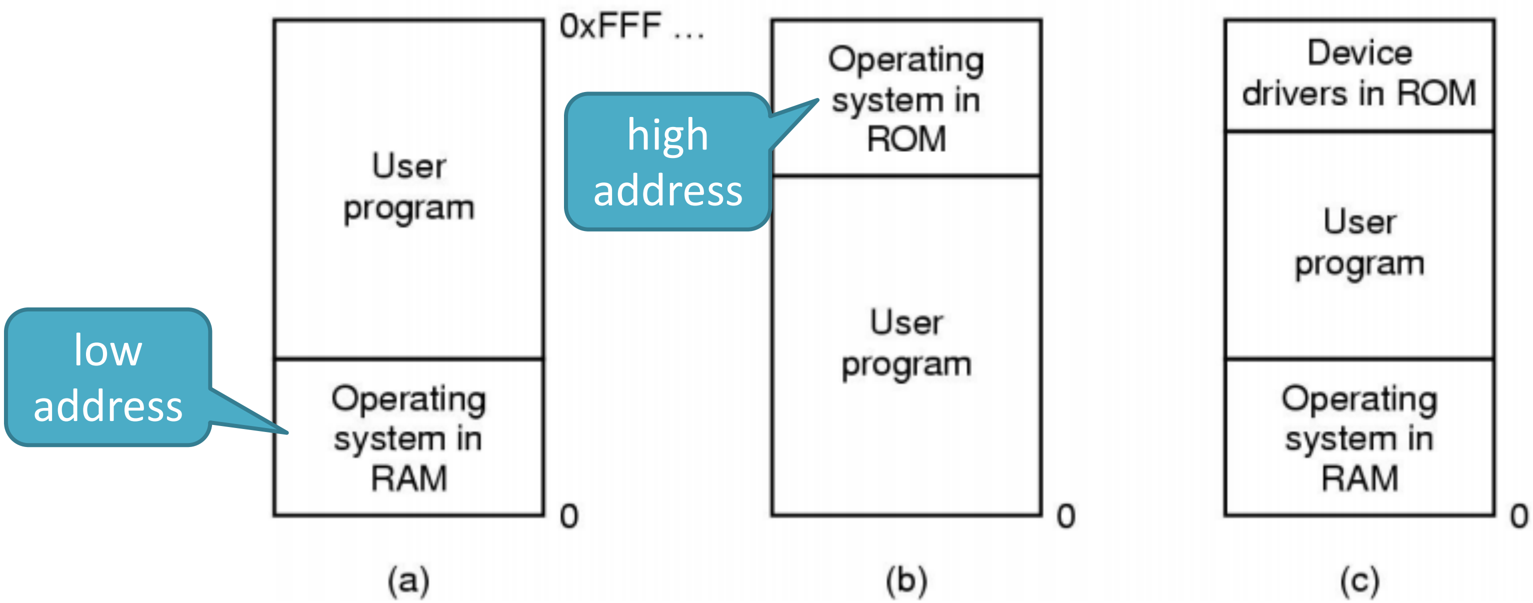
(a) early mainframes.

(b) Handheld and embedded systems.

(c) early PC.

Without memory abstraction (early era)

Memory management is simple. Put small piece for OS, the rest for apps.

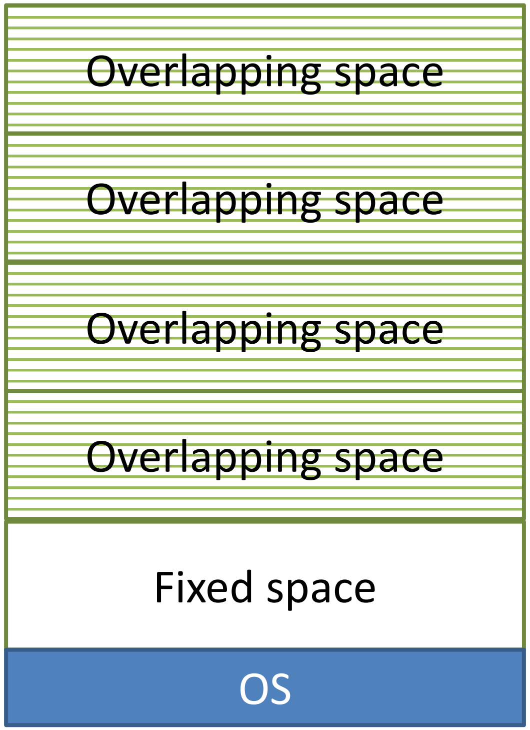


Problems:

Simple memory organization: Cannot support multi-programming and only works for devices, like washing machines, microwaves, etc.

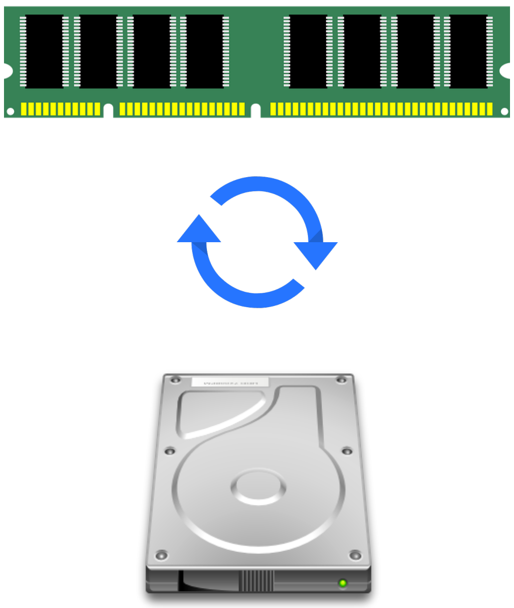
Without memory abstraction (What will happen when memory is not enough?)

- Overlapping
  - Memory is small and cannot hold all program data
  - User space is divided to one fixed space and several overlapping space
  - Fixed space: hold most frequent data
  - Overlapping space: the data will overlap those which never be used (after overlapping, data is gone)



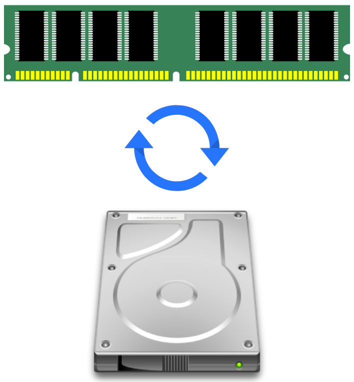
Without memory abstraction (What will happen when memory is not enough?)

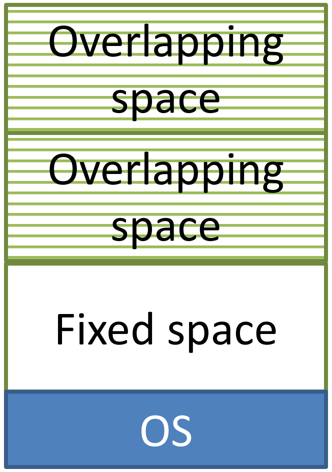
- Swapping
  - Leave memory and are swapped out to disk (data still there)
  - Re-enter memory by getting swapped in from disk



Without memory abstraction (What will happen when memory is not enough?)

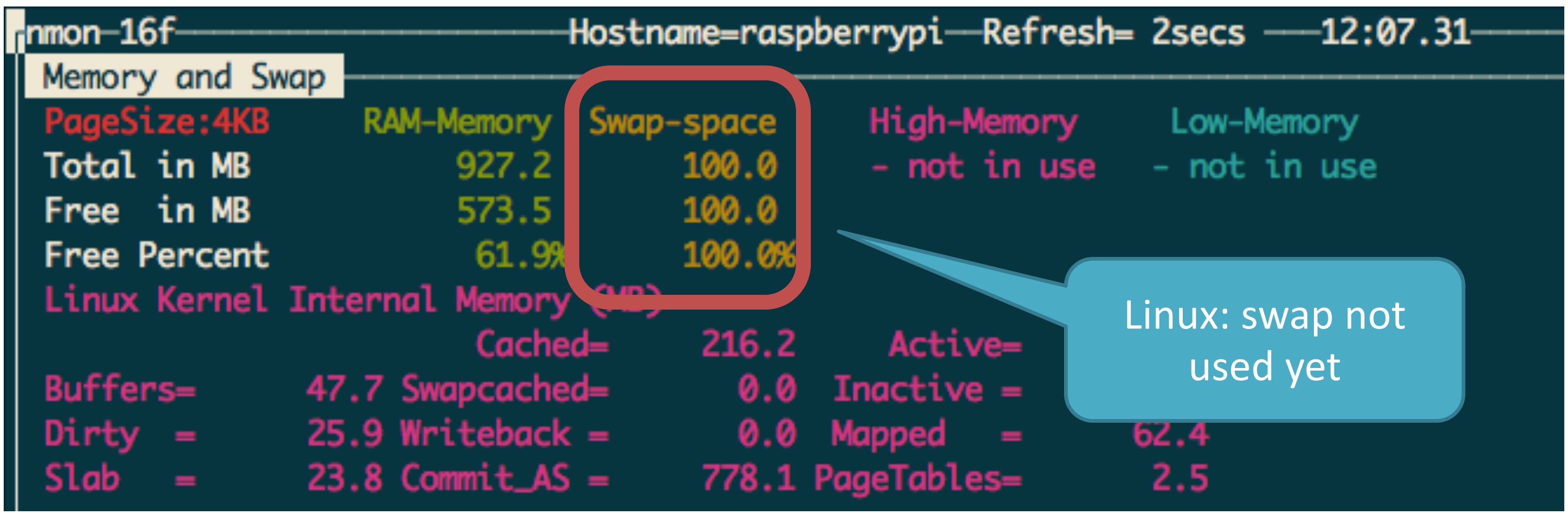
- Difference
  - Swapping is used for different processes while overlapping is used for one program/process (for security, otherwise causing app crash)
  - Swapping is still widely used today while overlapping is not



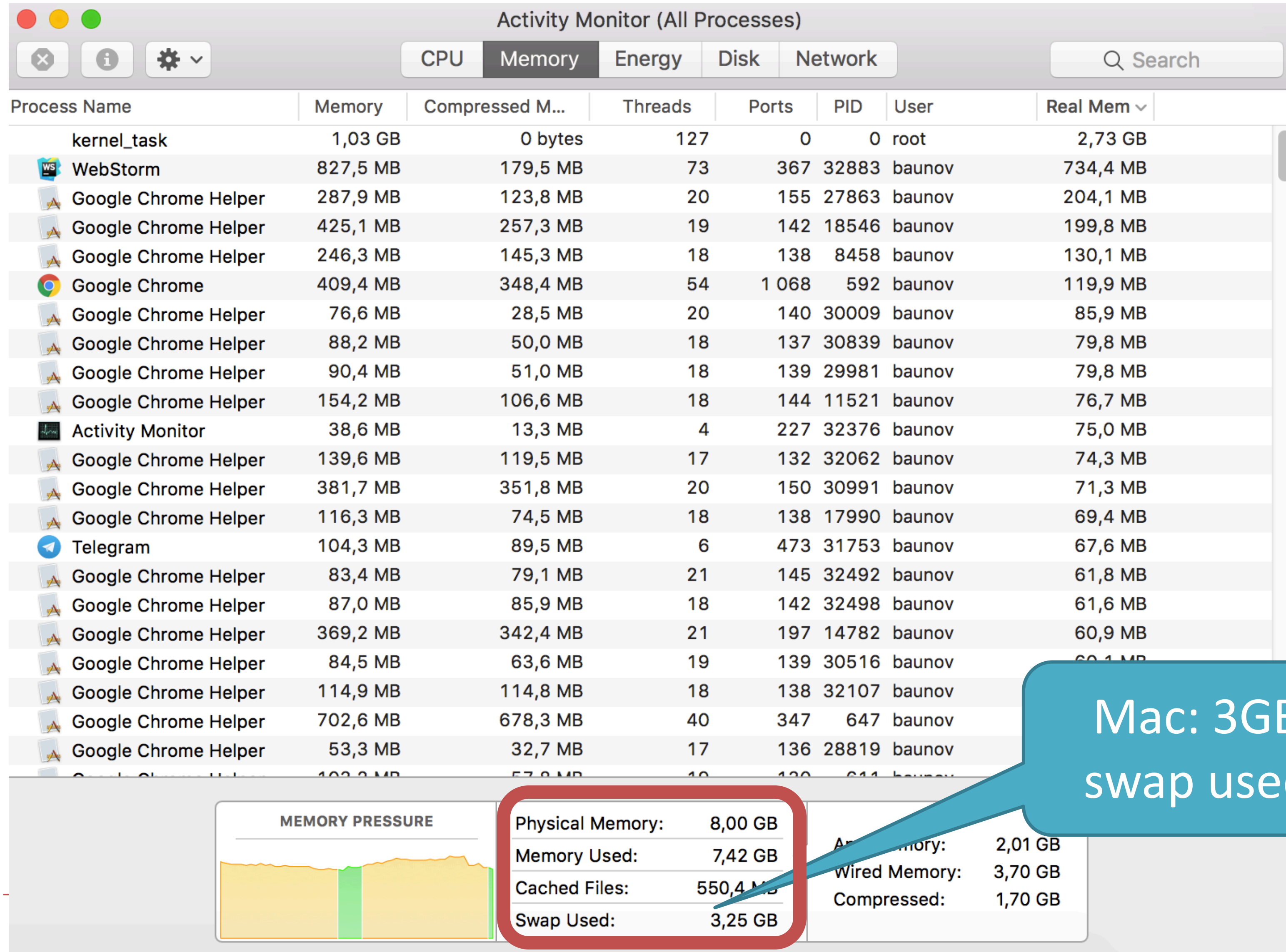


Swap example in Linux

- $ nmon, then press ‘m’ to check statistics from memory



Swap example in Mac



Activity Monitor (All Processes)

CPU | Memory | Energy | Disk | Network

Process Name	Memory	Compressed M...	Threads	Ports	PID	User	Real Mem
kernel_task	1,03 GB	0 bytes	127	0	0	root	2,73 GB
WebStorm	827,5 MB	179,5 MB	73	367	32883	baunov	734,4 MB
Google Chrome Helper	287,9 MB	123,8 MB	20	155	27863	baunov	204,1 MB
Google Chrome Helper	425,1 MB	257,3 MB	19	142	18546	baunov	199,8 MB
Google Chrome Helper	246,3 MB	145,3 MB	18	138	8458	baunov	130,1 MB
Google Chrome	409,4 MB	348,4 MB	54	1 068	592	baunov	119,9 MB
Google Chrome Helper	76,6 MB	28,5 MB	20	140	30009	baunov	85,9 MB
Google Chrome Helper	88,2 MB	50,0 MB	18	137	30839	baunov	79,8 MB
Google Chrome Helper	90,4 MB	51,0 MB	18	139	29981	baunov	79,8 MB
Google Chrome Helper	154,2 MB	106,6 MB	18	144	11521	baunov	76,7 MB
Activity Monitor	38,6 MB	13,3 MB	4	227	32376	baunov	75,0 MB
Google Chrome Helper	139,6 MB	119,5 MB	17	132	32062	baunov	74,3 MB
Google Chrome Helper	381,7 MB	351,8 MB	20	150	30991	baunov	71,3 MB
Google Chrome Helper	116,3 MB	74,5 MB	18	138	17990	baunov	69,4 MB
Telegram	104,3 MB	89,5 MB	6	473	31753	baunov	67,6 MB
Google Chrome Helper	83,4 MB	79,1 MB	21	145	32492	baunov	61,8 MB
Google Chrome Helper	87,0 MB	85,9 MB	18	142	32498	baunov	61,6 MB
Google Chrome Helper	369,2 MB	342,4 MB	21	197	14782	baunov	60,9 MB
Google Chrome Helper	84,5 MB	63,6 MB	19	139	30516	baunov	
Google Chrome Helper	114,9 MB	114,8 MB	18	138	32107	baunov	
Google Chrome Helper	702,6 MB	678,3 MB	40	347	647	baunov	
Google Chrome Helper	53,3 MB	32,7 MB	17	136	28819	baunov	

MEMORY PRESSURE

Physical Memory:	8,00 GB
Memory Used:	7,42 GB
Cached Files:	550,4
Swap Used:	3,25 GB

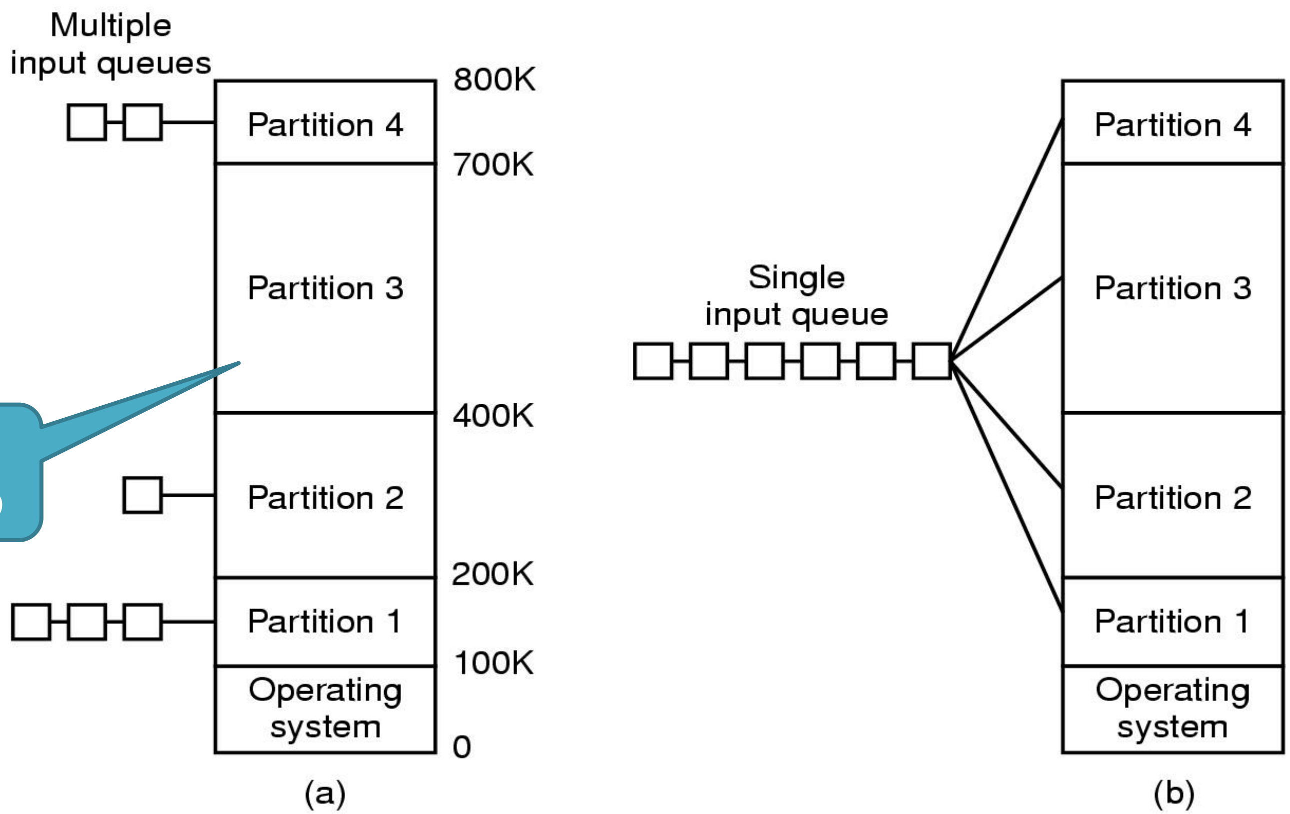
…mory:	2,01 GB
Wired Memory:	3,70 GB
Compressed:	1,70 GB

Mac: 3GB swap used

Without memory abstraction (How to support multiprogramming)

with Fixed Partitions

Each partition is used for one app

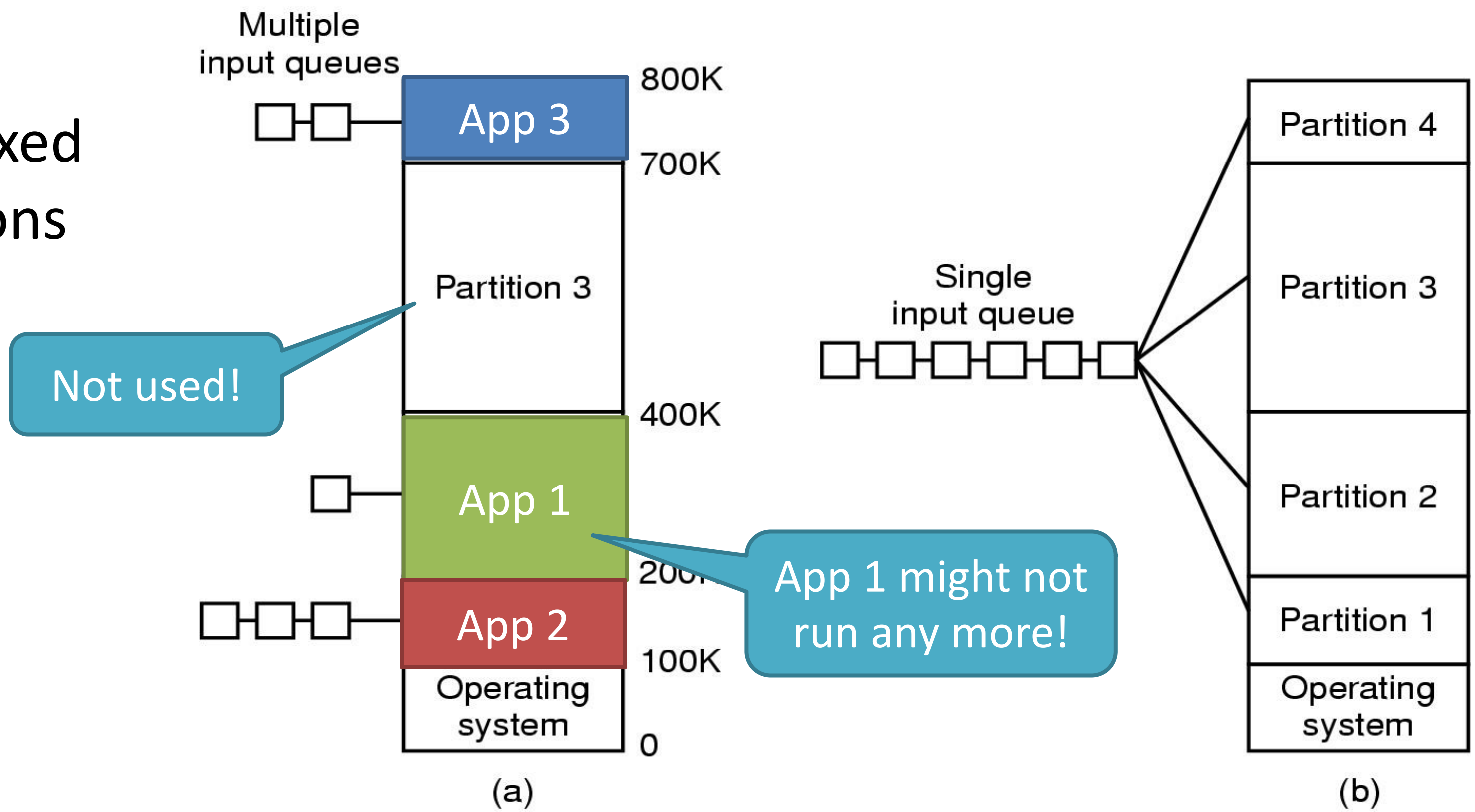


(a) separate input queues of processes for each partition

(b) single input queue

Without memory abstraction (How to support multiprogramming)

with Fixed Partitions

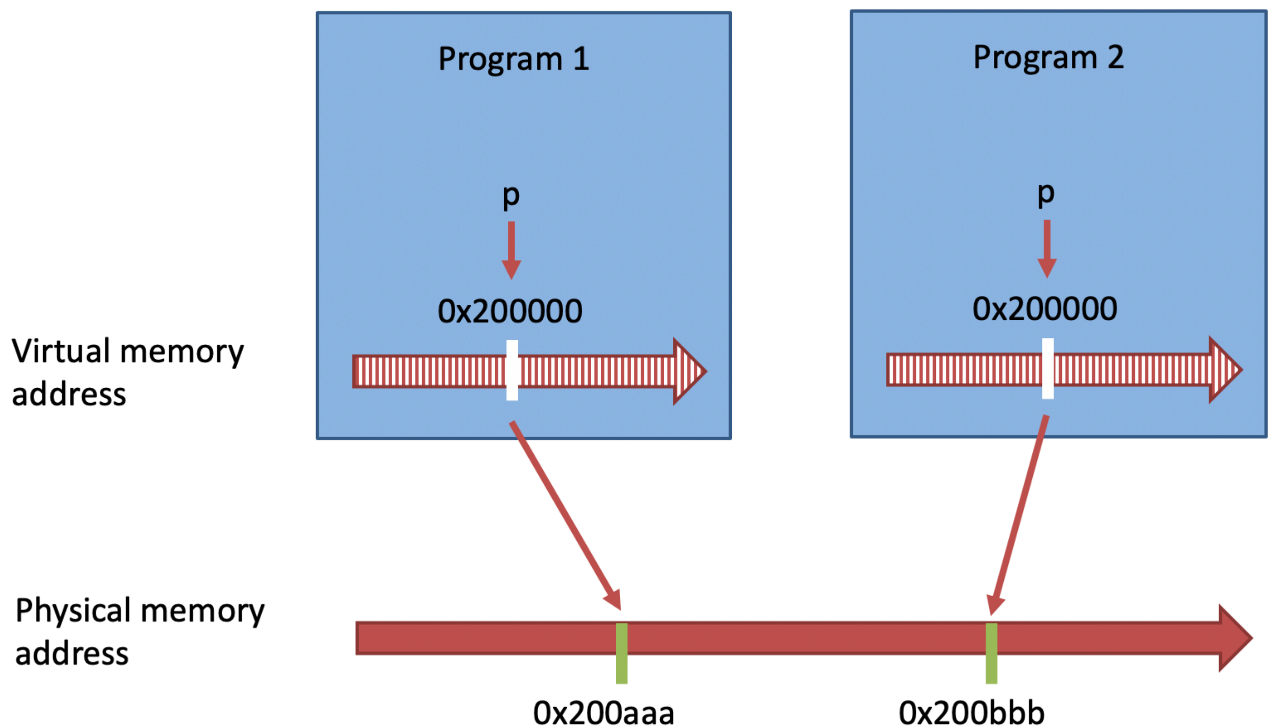


Problems:

(a) fragmentation
(b) low efficiency

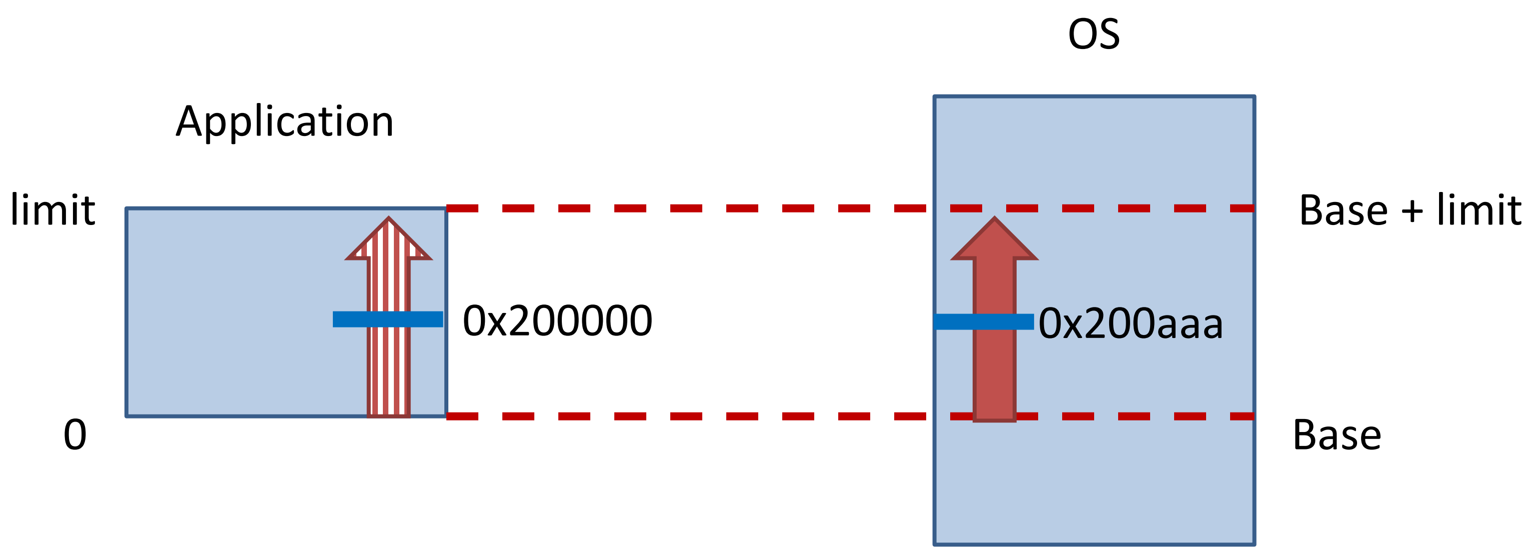
Memory Abstraction: Address Spaces

- Program should have their own views of memory
  - The address space – logical address
  - Non-overlapping address spaces – protection
  - Move a program by mapping its addresses to a different place - relocation

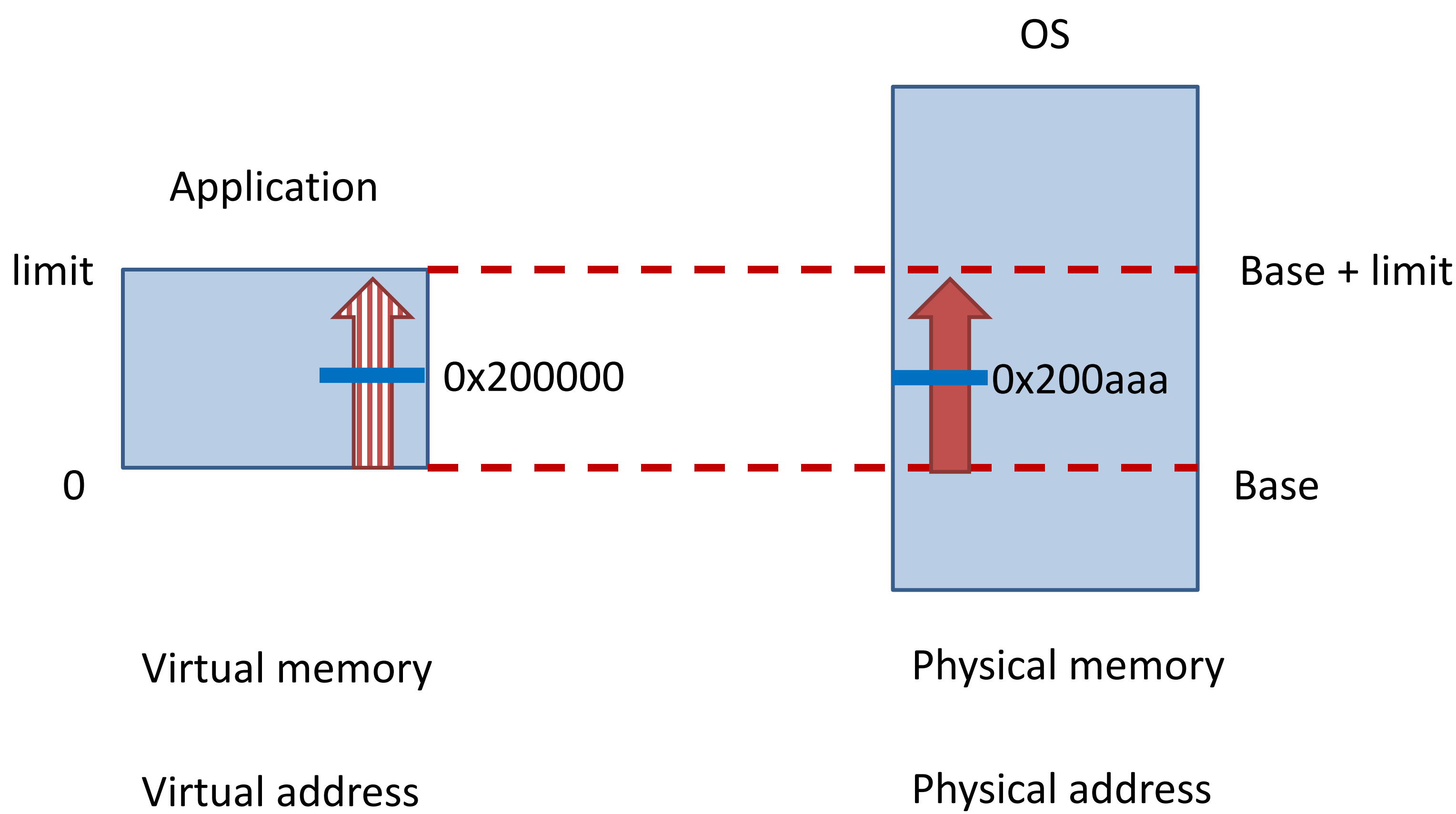


Memory Abstraction: Address Spaces

- OS dynamically relocates programs in memory
  - Two hardware registers: base and limit
  - **Base**: start address of a process
  - **Limit**: the upper bound address of a process
  - Hardware adds relocation register (base) to virtual address to get a physical address



Memory Abstraction: Address Spaces



Physical address vs. Virtual address

- Virtual address: A memory address that an operating system allows a process to use **(Program view)**

- Physical address: A unit address for memory chip level that corresponds to the address bus to which the processor and CPU are connected **(OS view)**

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

int main(int argc, char *argv[])
{
	int x = 3;

	printf("location of code : %p\n", (void *) main);
	printf("location of heap : %p\n", (void *) malloc(1));
	printf("location of stack : %p\n", (void *) &x);

	return 0;
}
```

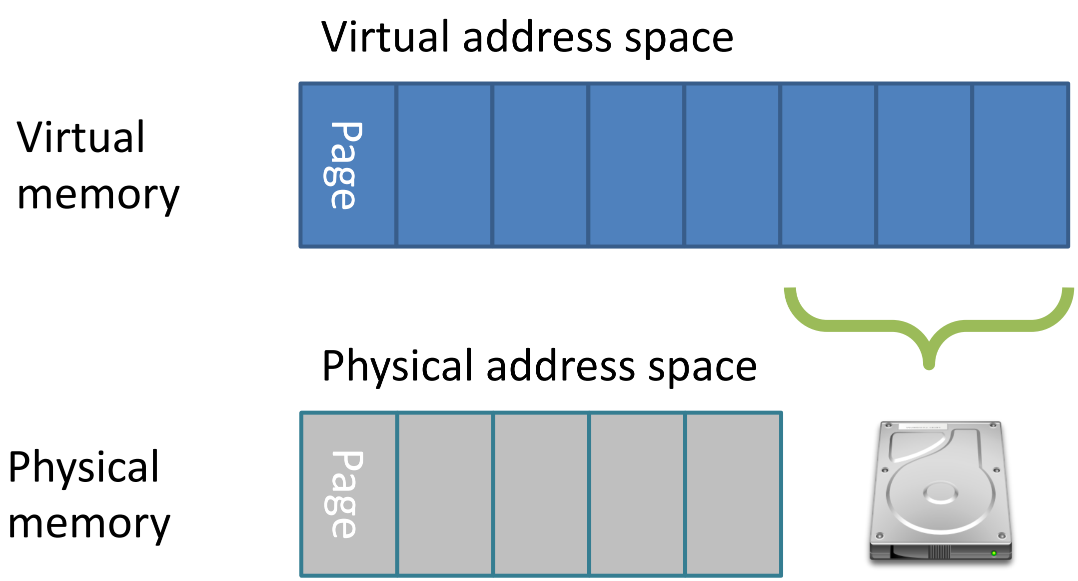
All virtual addresses (program view)

```
pi@raspberrypi ~> ./test.o
location of code : 0x10470
location of heap : 0x156a410
location of stack : 0x7ef381c4
```

Physical memory vs. Virtual memory

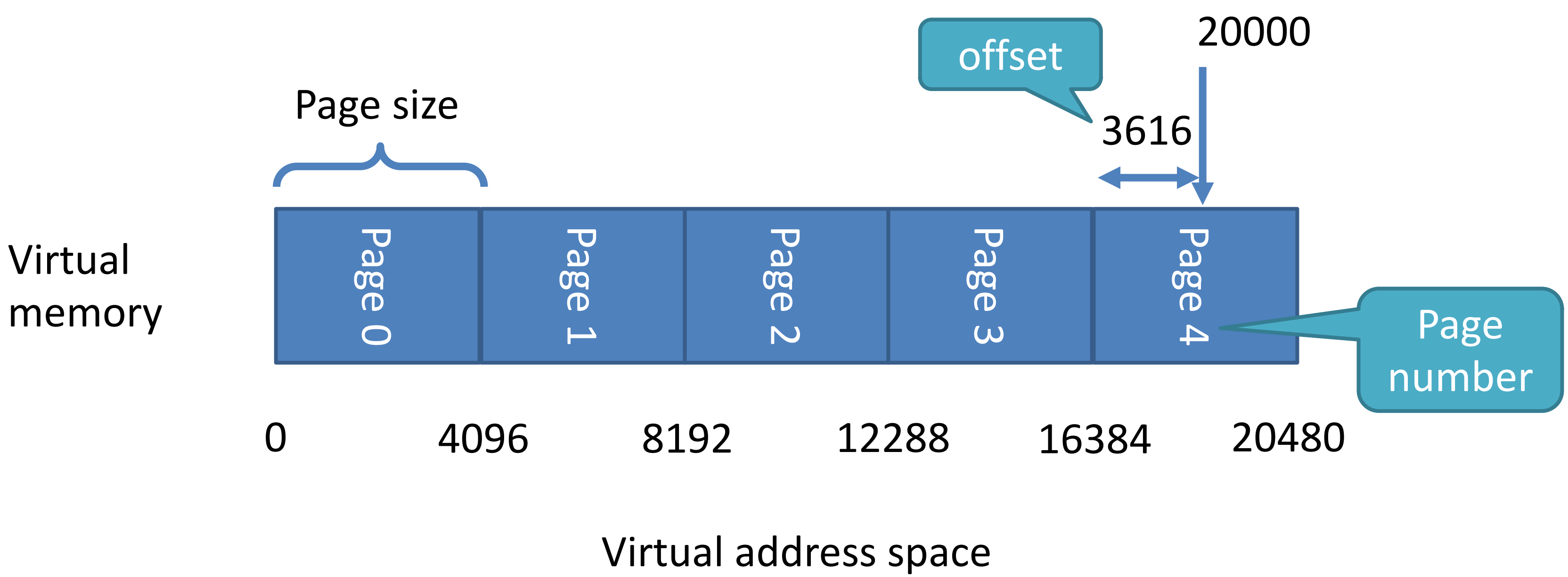
- Virtual memory: the combined size of the program, data, and stack **may exceed** the amount of physical memory available (due to swapping).

- Memory is divided into pages and OS decides which pages stay in **memory** and which get moved to **disk**.



Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - The virtual page number is *20000/4096 = 4*
  - The offset is *20000%4096 = 3616*



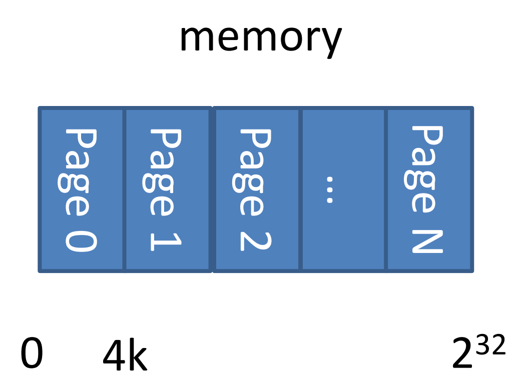
Question

- Consider a machine that has a 32-bit virtual address space and 4K Byte page size.

1. What is the memory size?

2^{32} = 4GB

//2^{10} = 1KB, 2^{20} = 1MB, 2^{30} = 1GB



Question

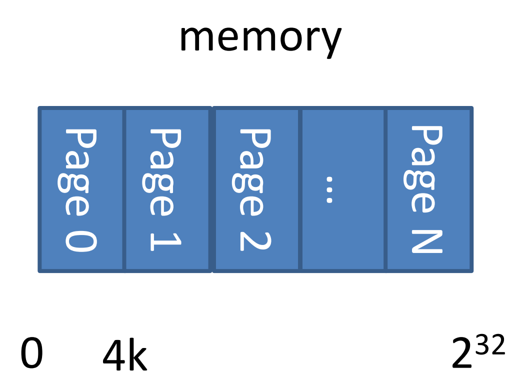
- Consider a machine that has a 32-bit virtual address space and 4K Byte page size.

2. How many virtual pages could a process have?

The total memory is 4GB

The single page size is 4KB

So the page number = 4GB/4KB = 2^{20}

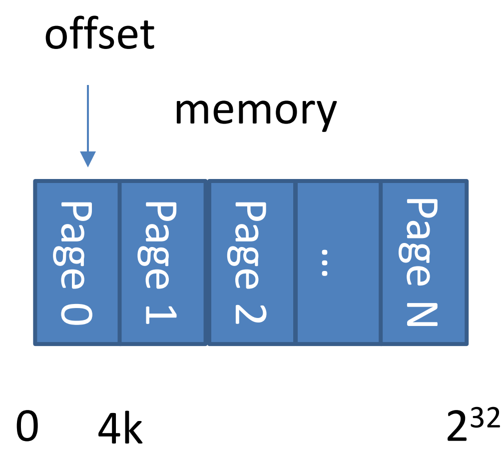


Question

- Consider a machine that has a 32-bit virtual address space and 4K Byte page size.

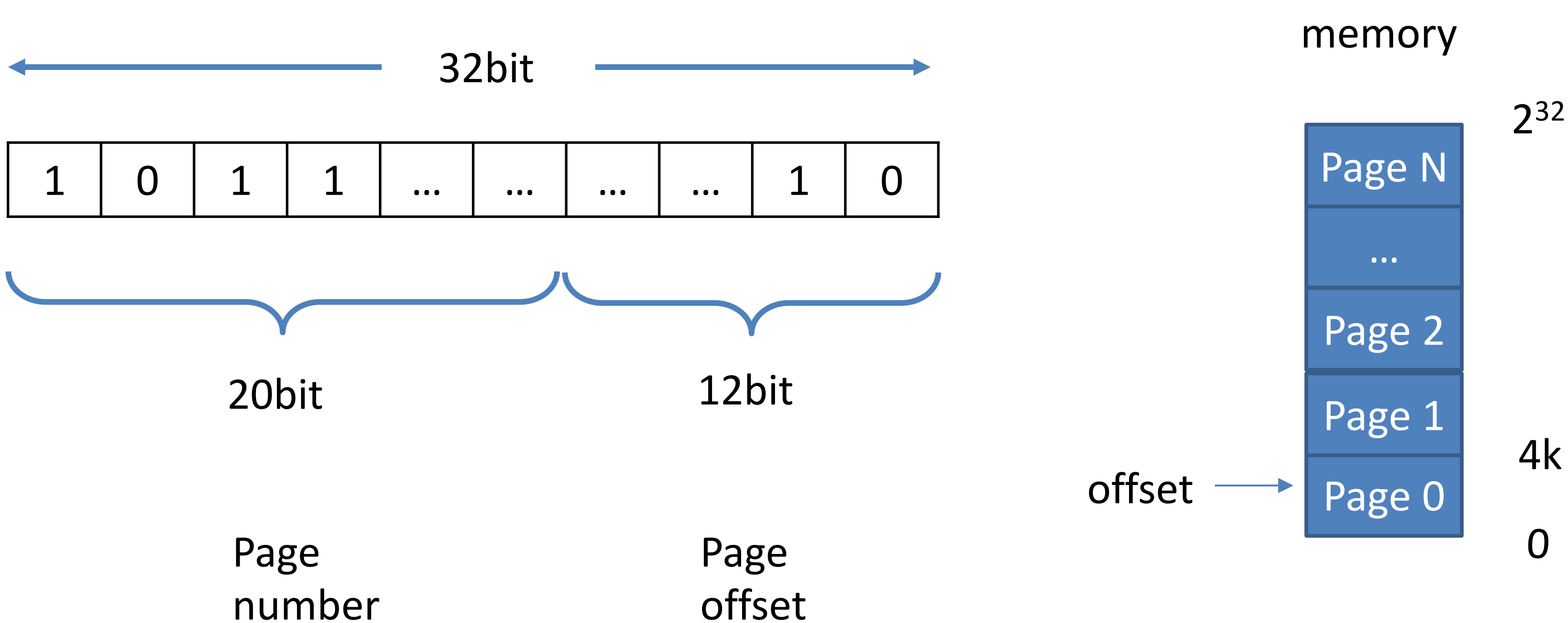
3. Given a 4KB page, how many address bits do we need for the offset ?

$\text{Log}_2(4\text{KB}) = \text{Log}_2(2^{12}) = 12$



Question

- Consider a machine that has a 32-bit virtual address space and 4K Byte page size.



Question

- Consider a machine that has a 32-bit virtual address space and 8KByte page size.

1. What is the total size (in bytes) of the virtual address space for each process?

Total size (in bytes) of the virtual address space for each process
$= 2^{32} = 4 * 1024 * 1024 * 1024$ bytes
= 4 GB

Question

- Consider a machine that has a 32-bit virtual address space and 8KByte page size.

2. How many bits in a 32-bit address are needed to determine the page number of the address?

Number of pages in virtual address space = 4GB/8KB = 512*1024 = $2^9 * 2^{10} = 2^{19}$

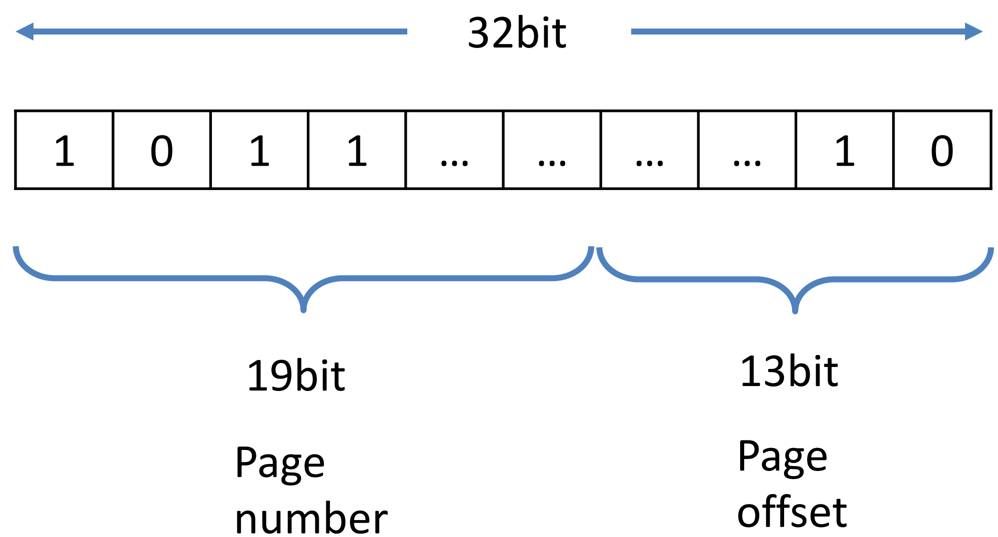
So the number of bits in a 32-bit address are needed to determine the page number of the address is 19 bits

Question

- Consider a machine that has a 32-bit virtual address space and 8KByte page size.

3. How many bits in a 32-bit address represent the byte offset into a page?

32 – 19 = 13 bits



Question

- Consider a machine that has a 32-bit virtual address space and 8KByte page size.

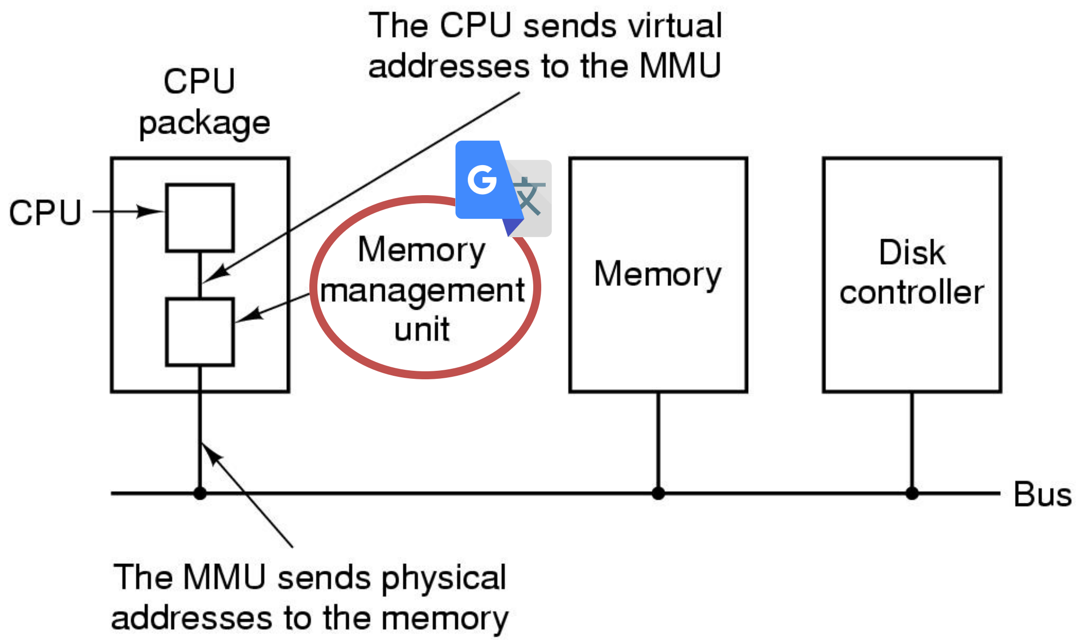
4. How many page-table entries are present in the page table?

Number of PTEs
= Number of pages in virtual address
= 4GB/8KB
= 2^{19} pages

Outline

- Memory management overview
  - Memory abstraction and address spaces
  - Physical address and virtual address
  - Physical memory and virtual memory
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table

Physical and virtual address translation

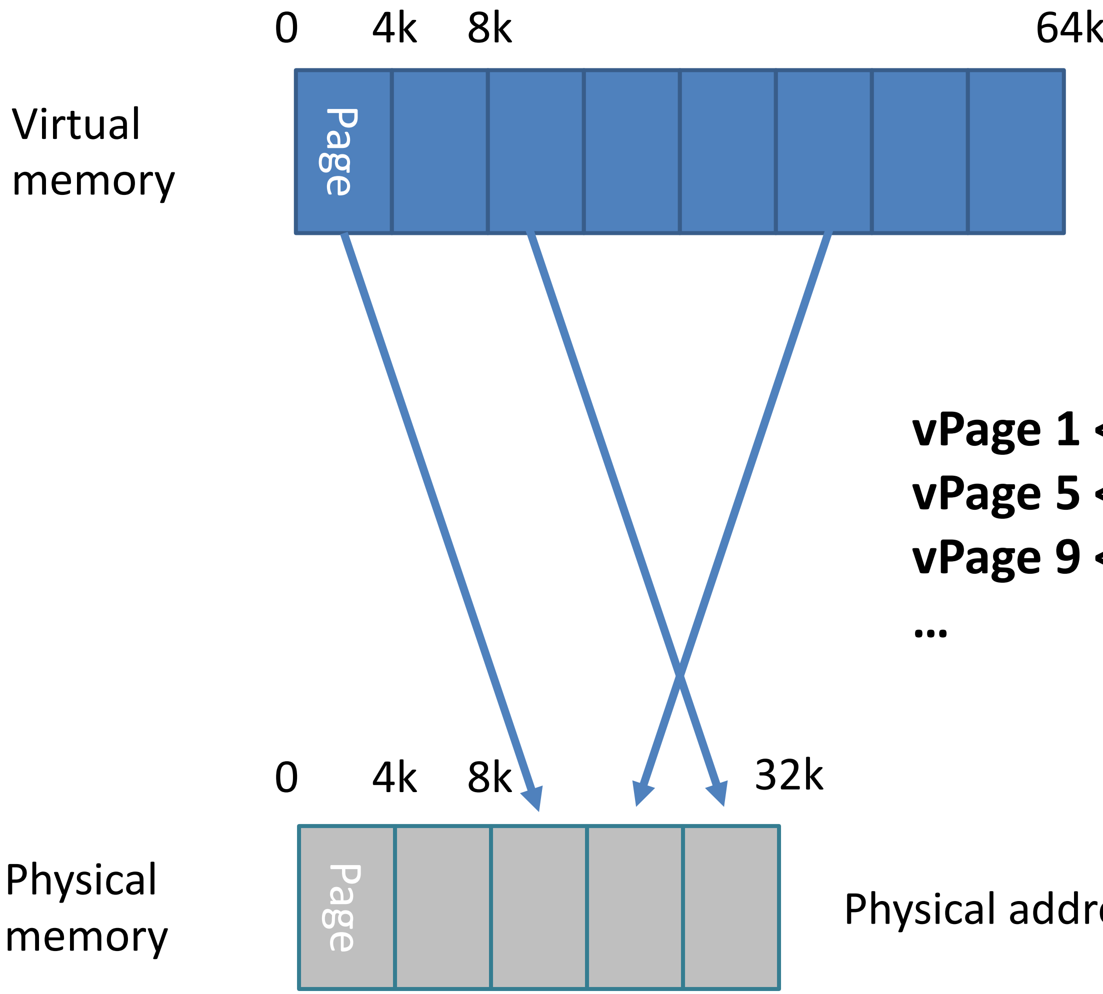


Logical program works in its contiguous virtual address space

Address translation done by MMU

Actual locations of the data in physical memory

Page and page table



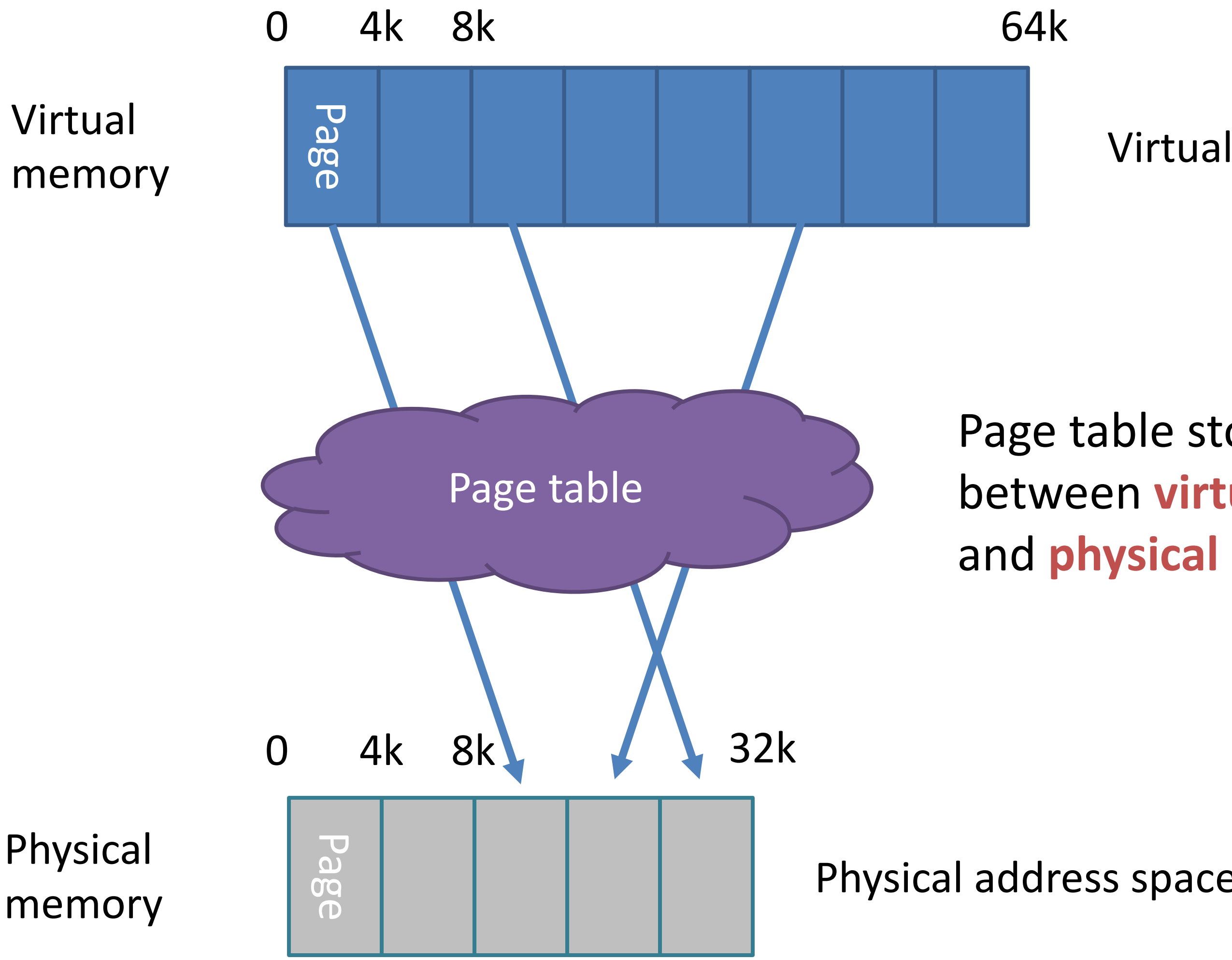
Virtual address space

Physical address space

vPage 1 <---> pPage 3
vPage 5 <---> pPage 8
vPage 9 <---> pPage 2
…

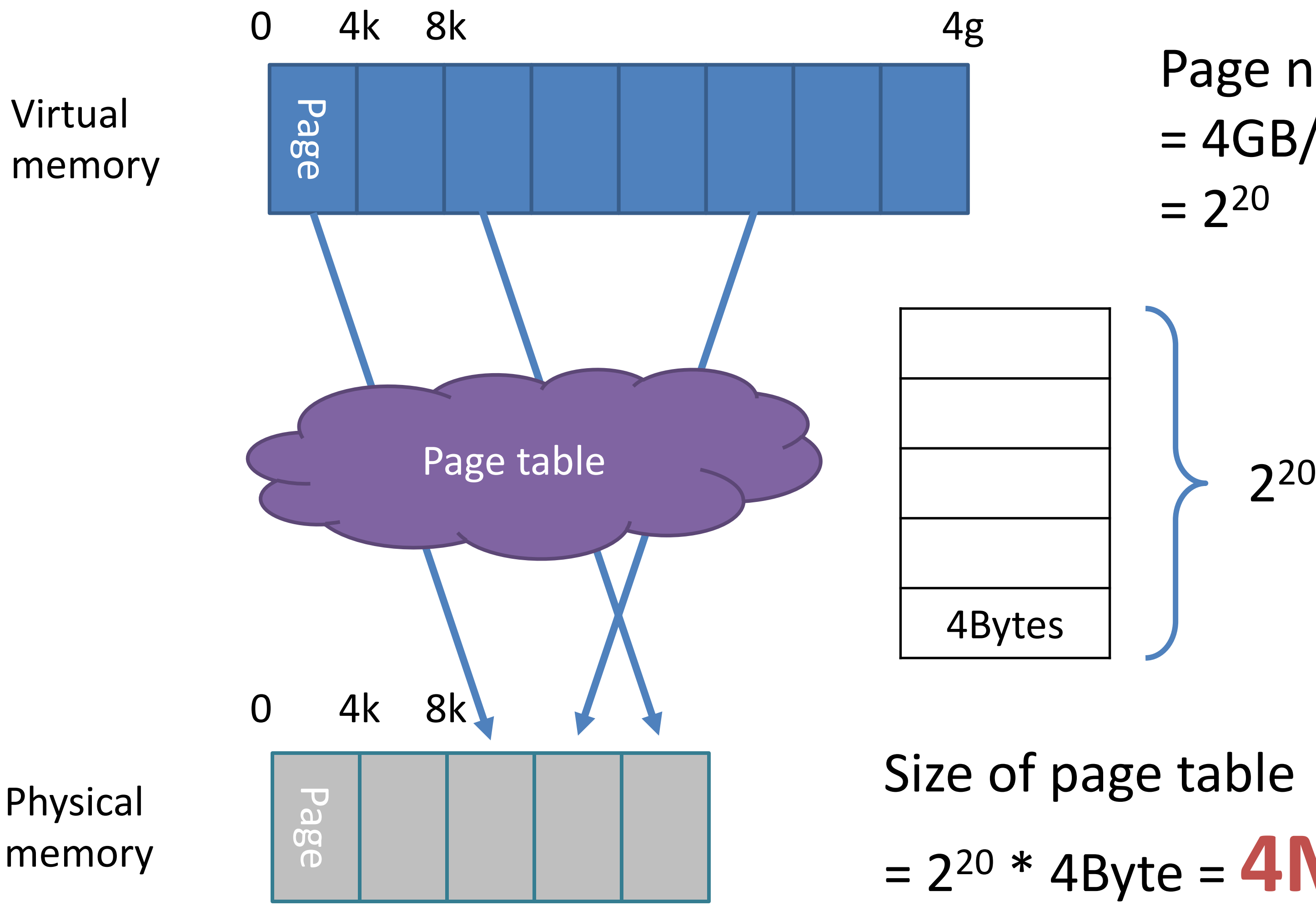
Where to put these info?

Page and page table



Page table stores the relation between **virtual addresses** and **physical addresses**

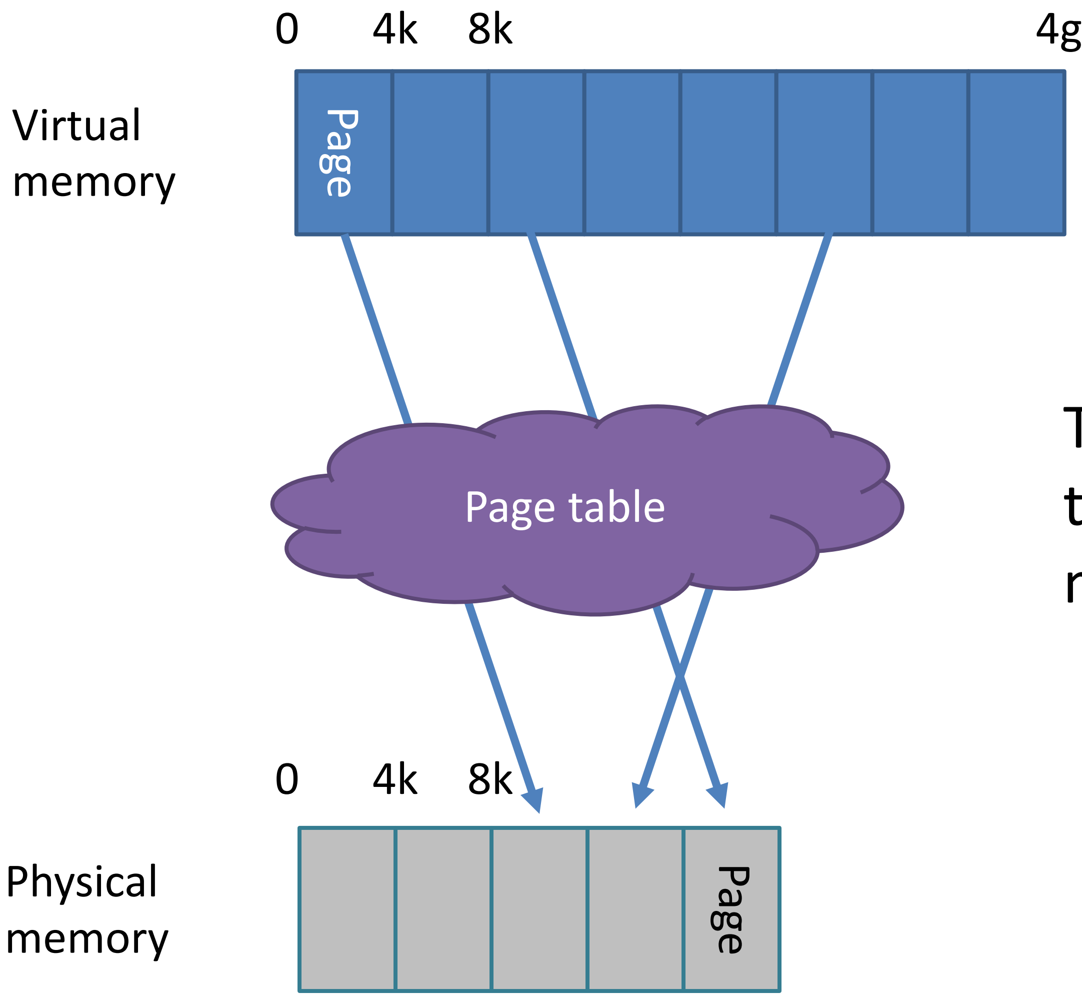
Suppose virtual address is 4GB and page size is 4KB. The page table item is 4 Byte. What would be the size of page table?



Page number
$= 4GB/4KB$
$= 2^{20}$

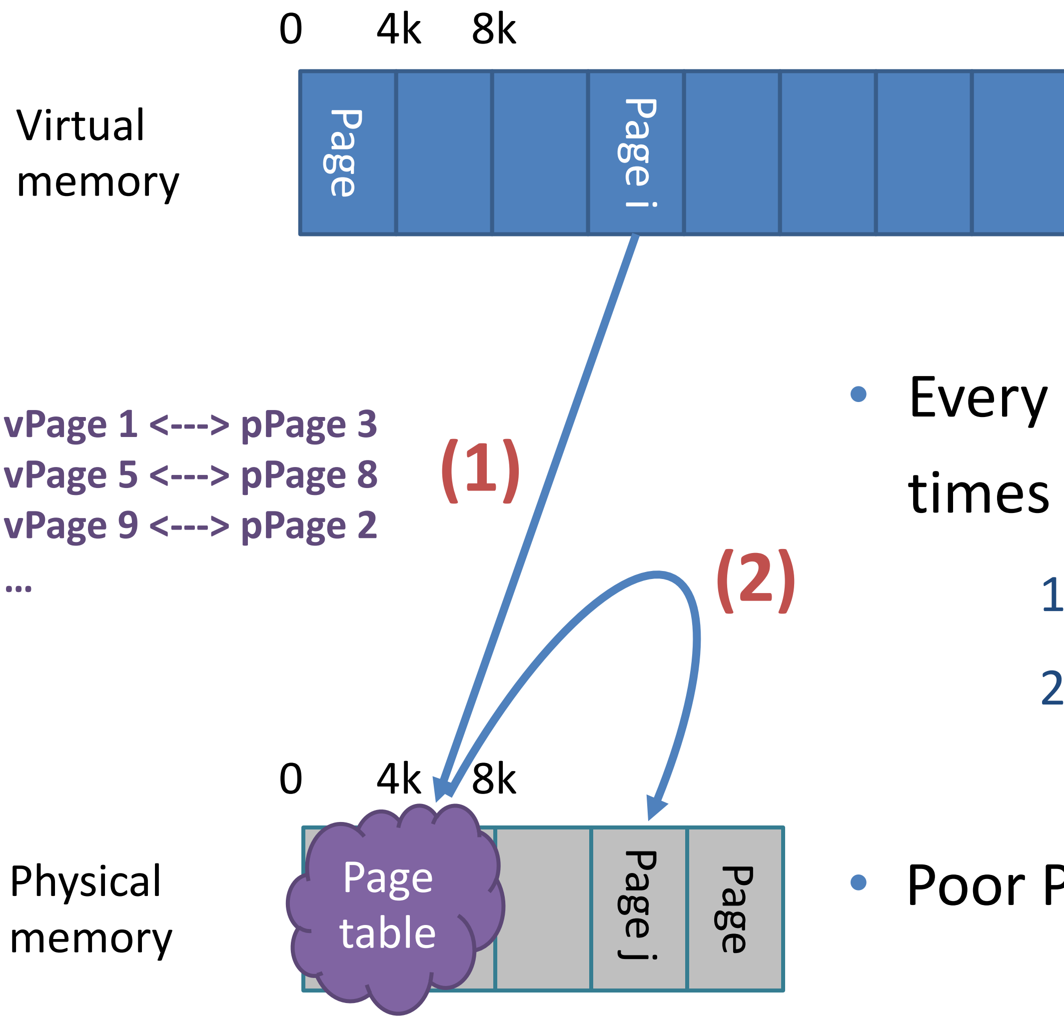
Size of page table
$= 2^{20} * 4Byte =$ **4MB**

Where does the page table exist?



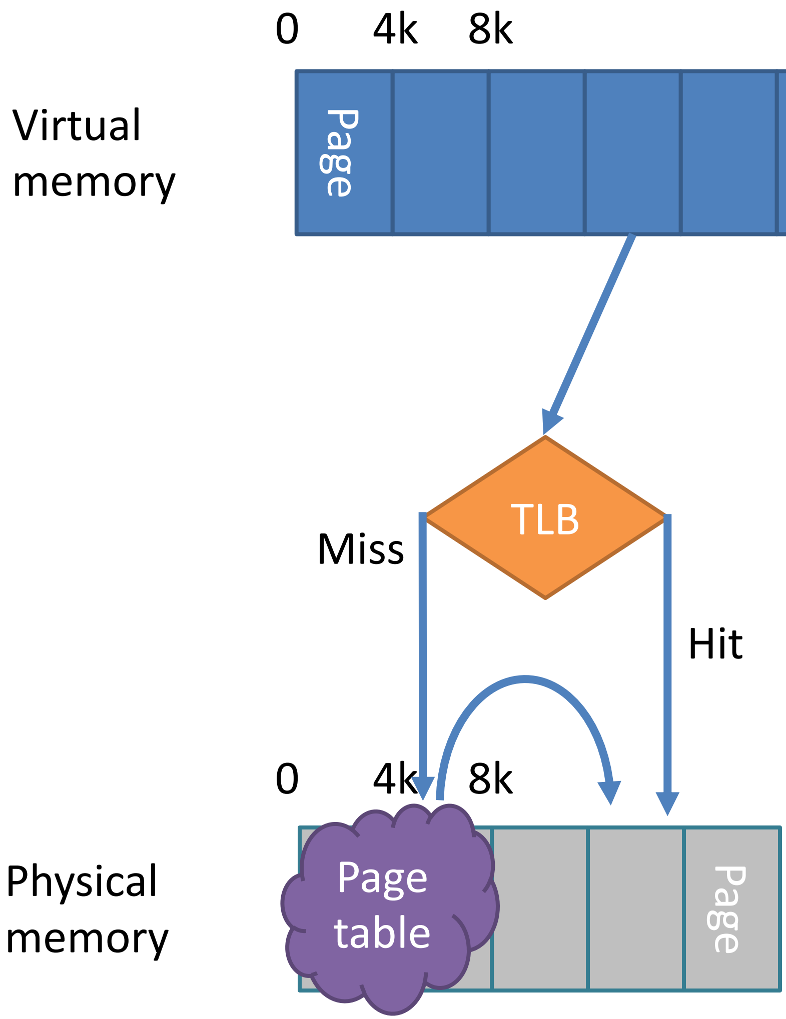
The table is located inside the **physical memory** and maintained by the **OS**

Page table lookup



- Every visit to memory requires two times of physical memory access:
  - 1, access to page table
  - 2, visit the destination address
- Poor Performance

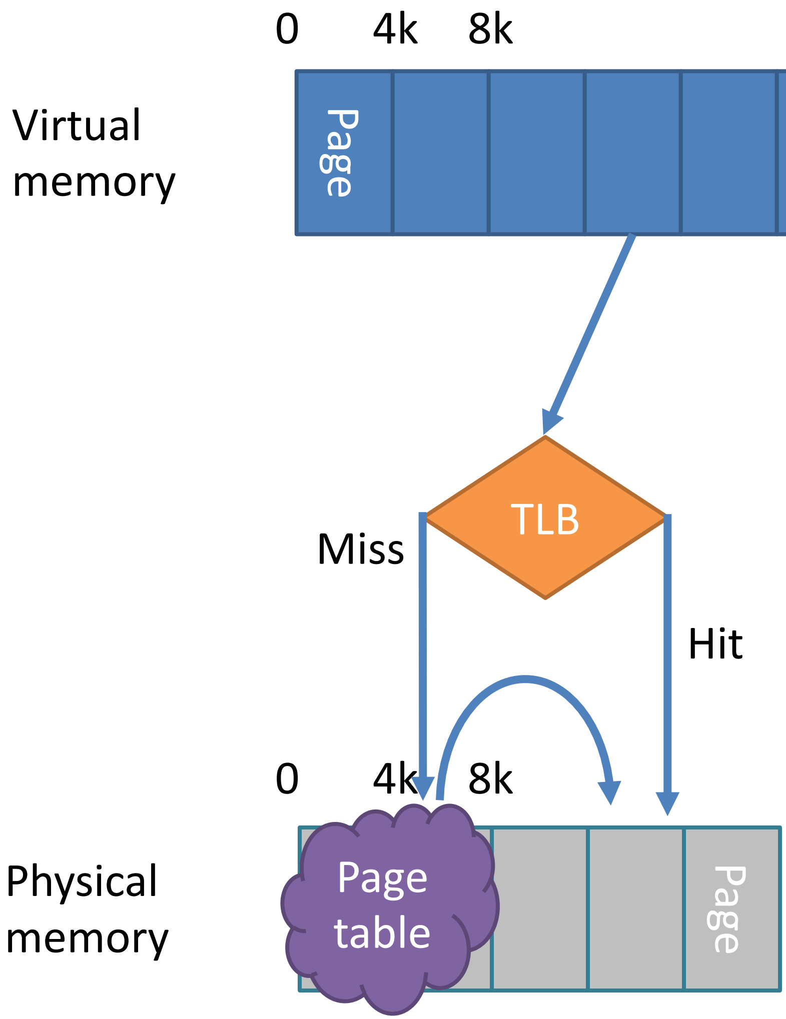
Translation Look-aside Buffers (TLB)

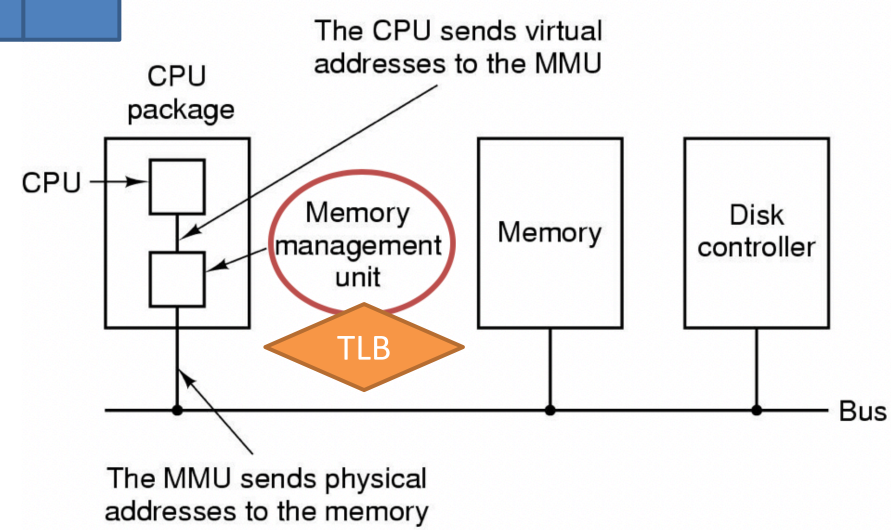


TLB greatly improves the efficiency!

- A memory cache that is used to reduce the time taken to access a memory location

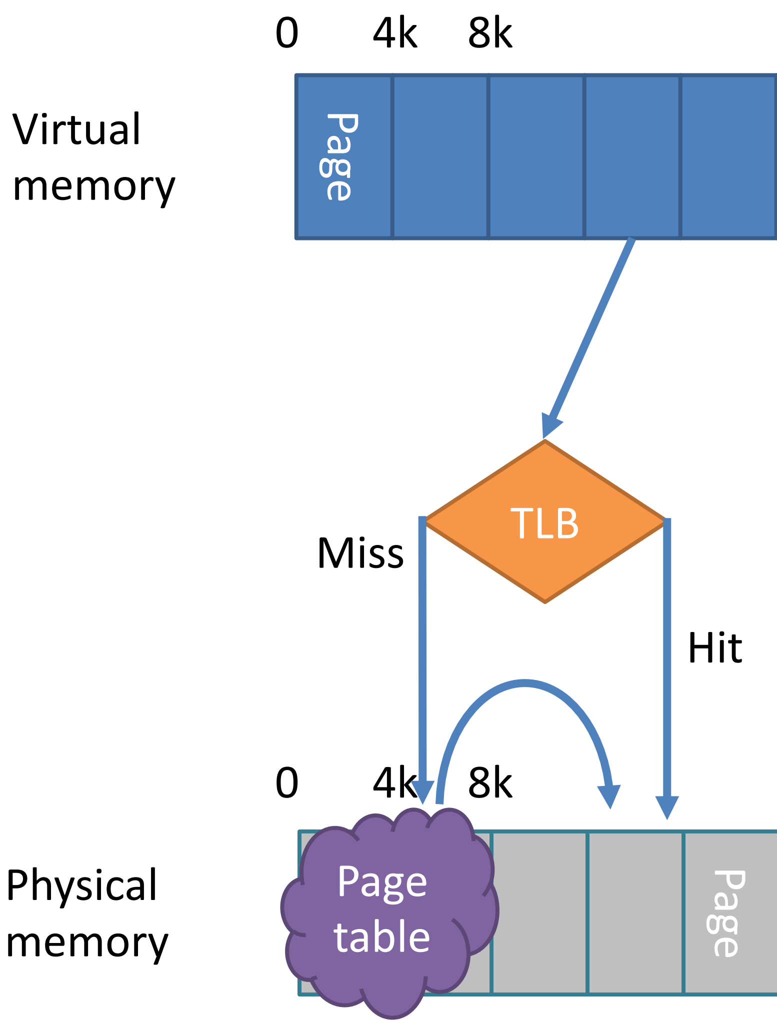
Translation Look-aside Buffers (TLB)





- A part of the chip's memory-management unit (MMU)

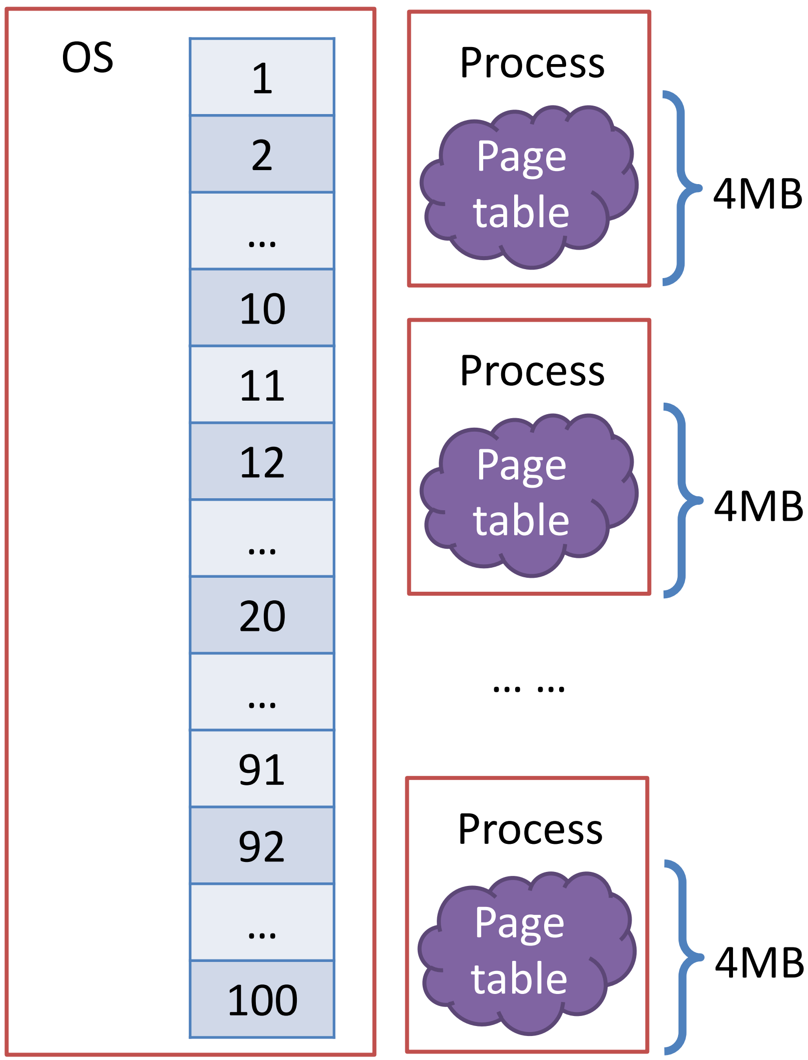
Translation Look-aside Buffers (TLB)



TLB greatly improves the efficiency!

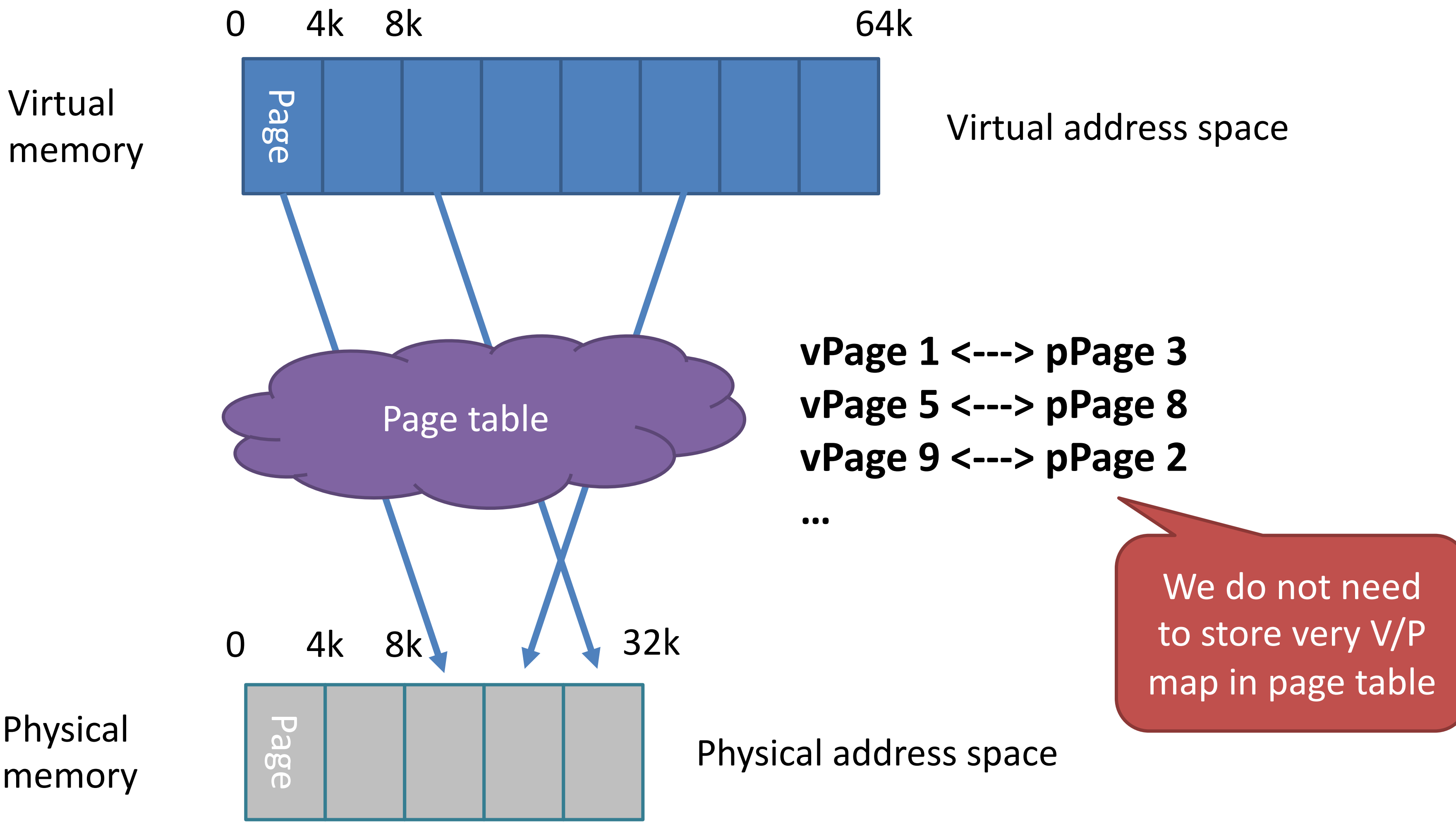
- TLB stores the recent translations of virtual memory to physical memory
  - Hit, access directly
  - Miss, visit the page table

Page table problems



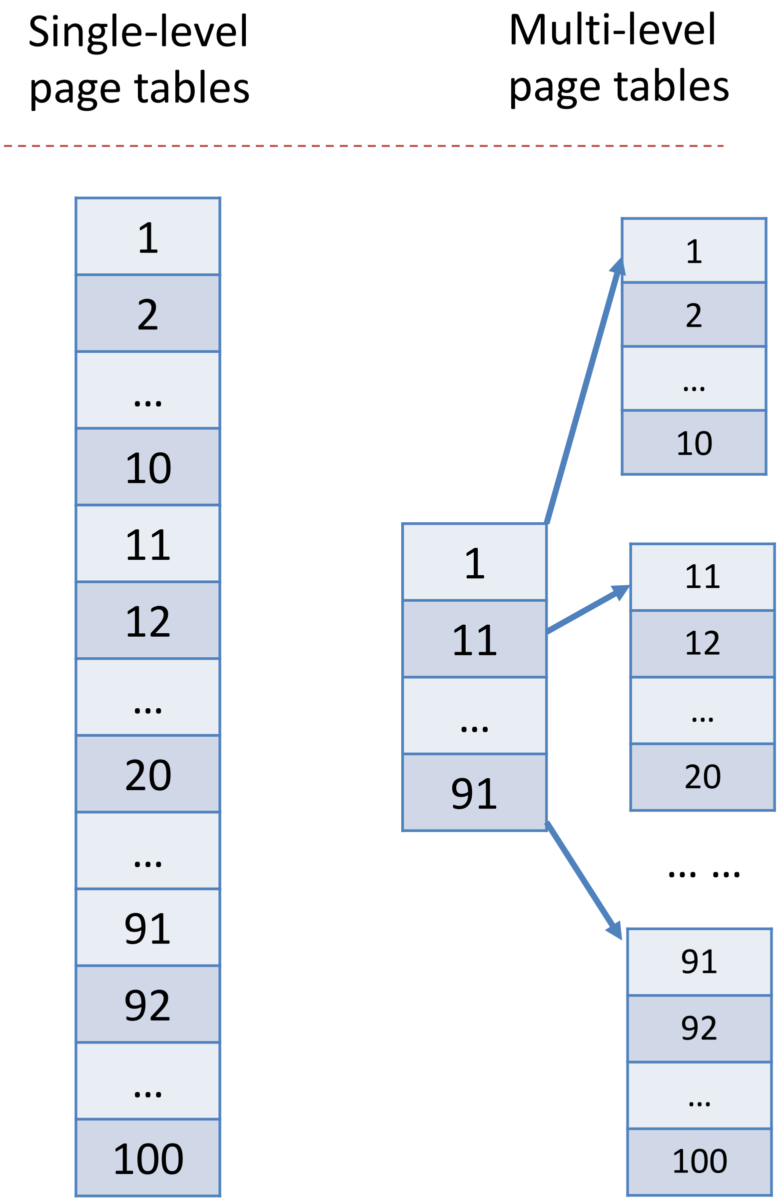
- Page table is a per-process data structure
- Large amount of memory could be used to store page table

Page table size optimization



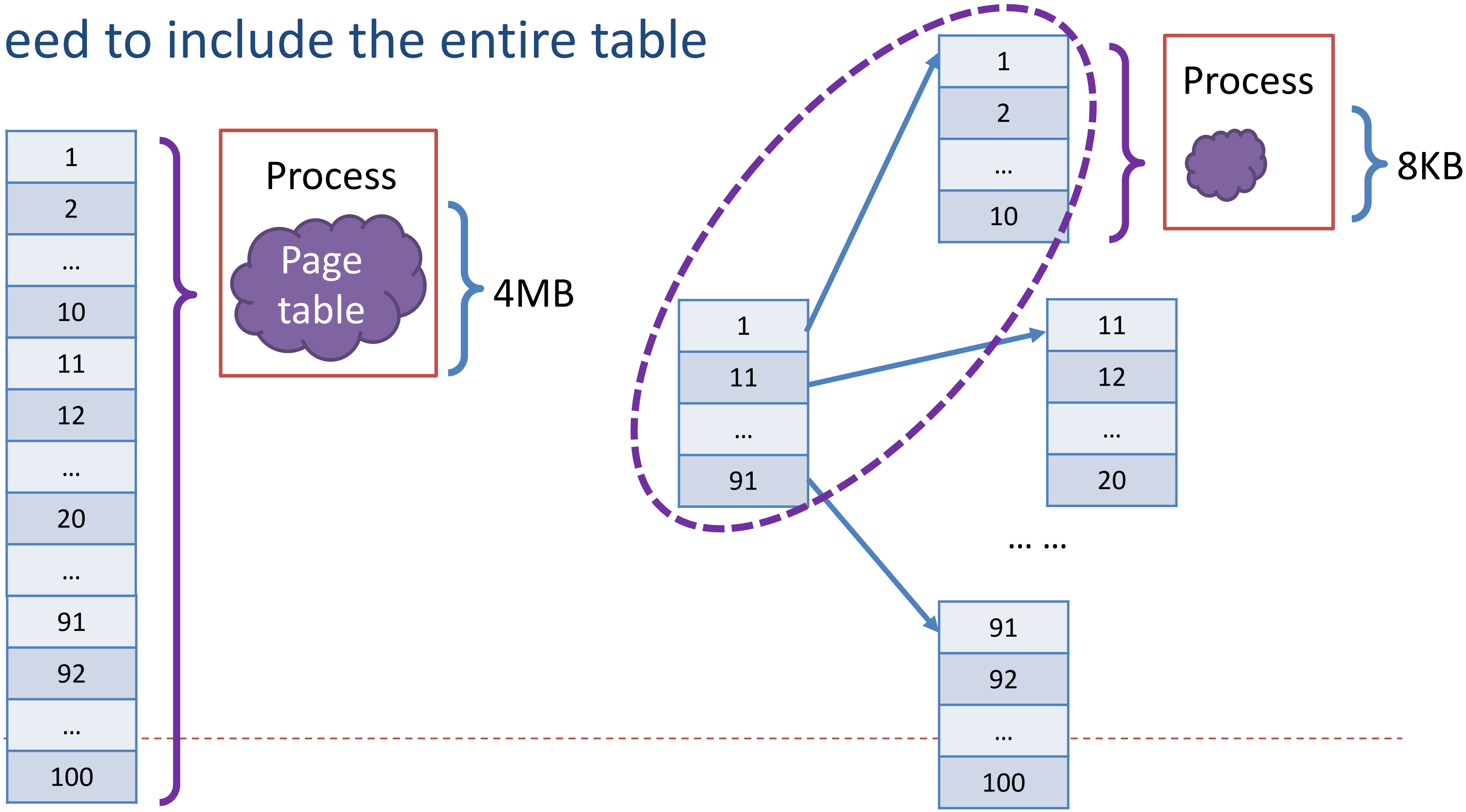
Multi-level Page Tables

- Multi-level page tables can reduce the memory occupied by page tables and improve the memory efficiency



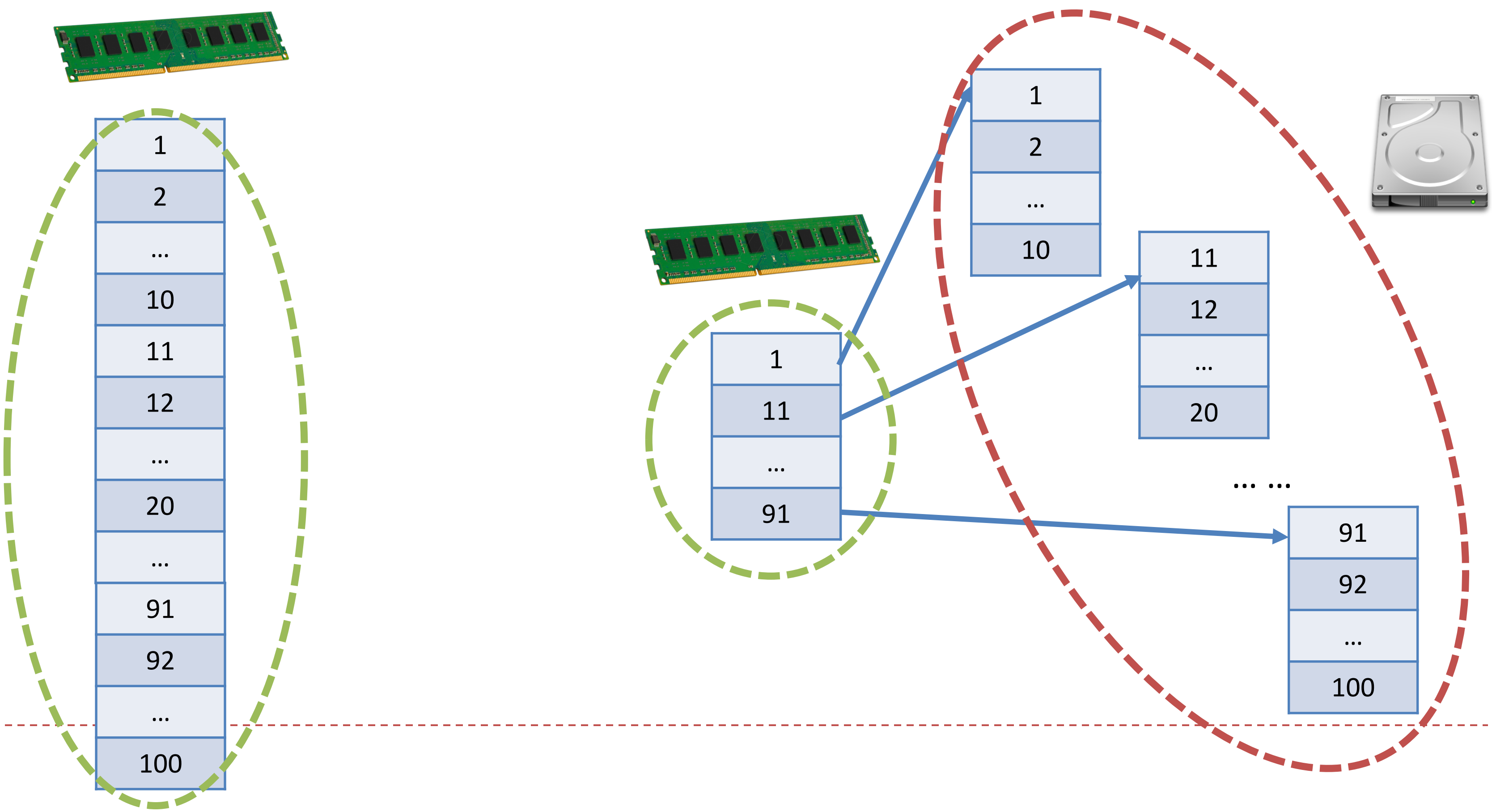
Multi-level Page Tables benefits

- Save memory space
  - 1) Far less page table to individual process (locality), not need to include the entire table



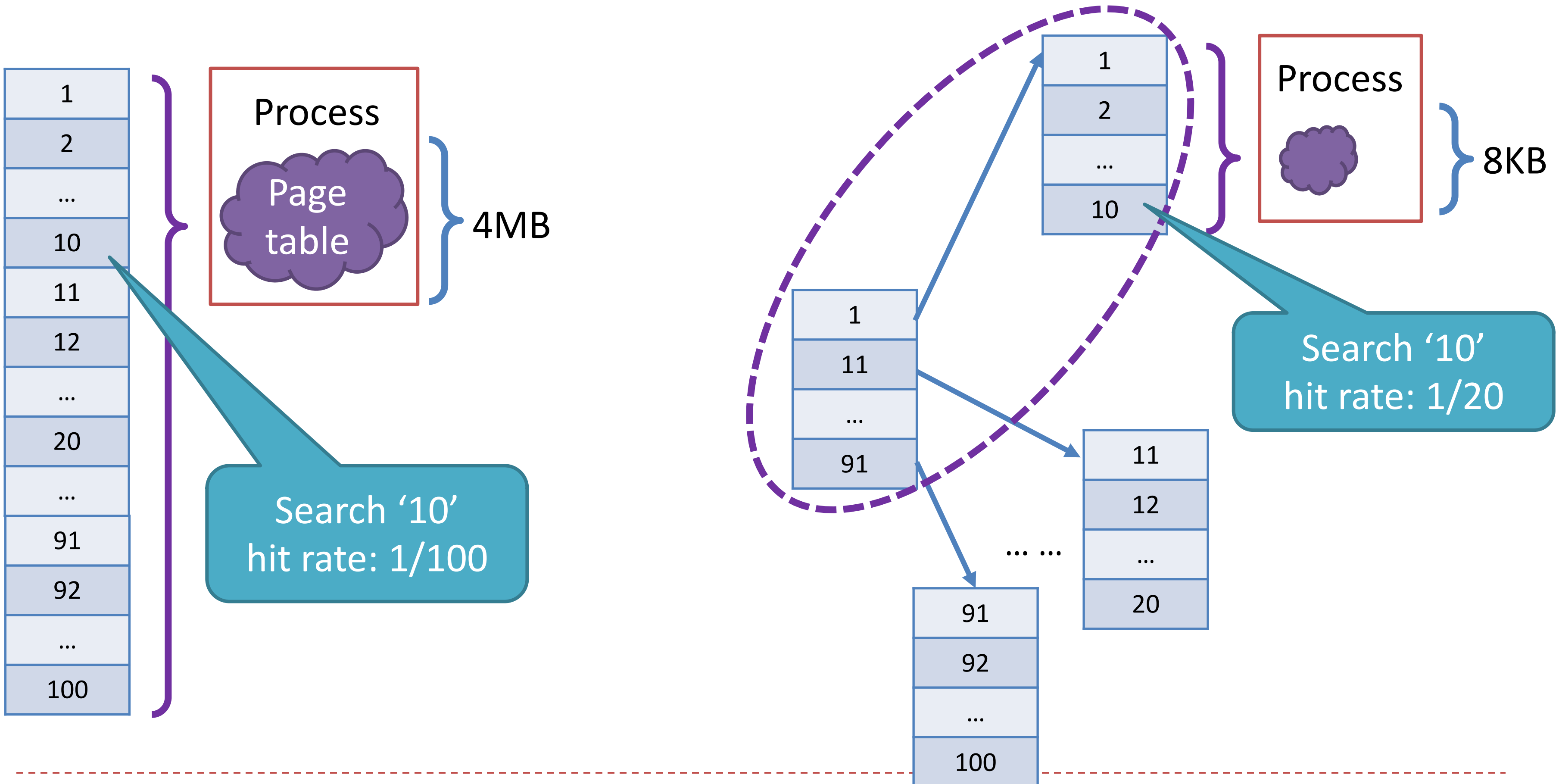
Multi-level Page Tables benefits

- Save memory space
  - 2) Second level page might not exist in memory, could be on the disk



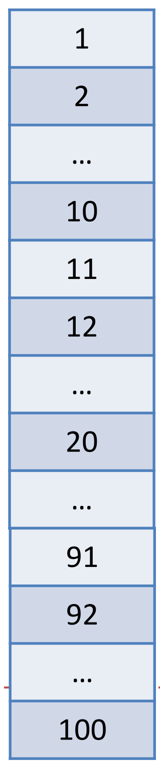
Multi-level Page Tables benefits

- High access efficiency due to table index

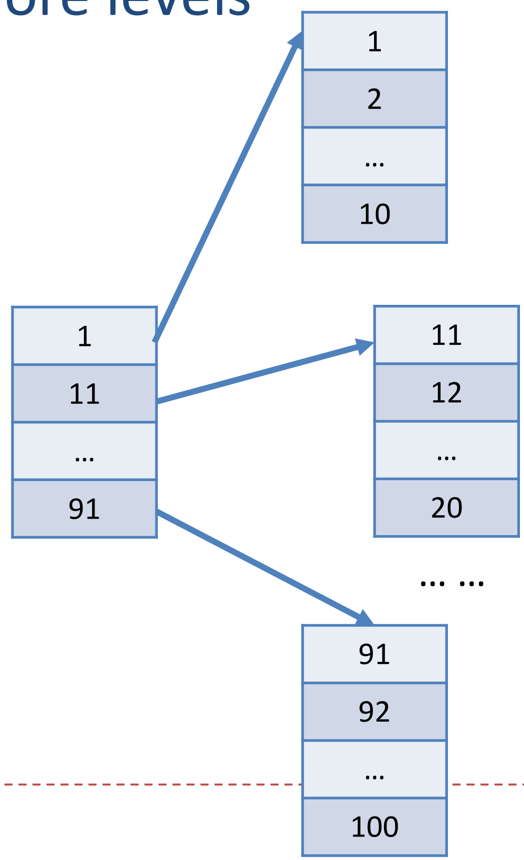


Multi-level Page Tables drawbacks

- Drawbacks
  - Index space
  - Higher complexity with more levels



Total amount space for single level page table is 100



Total amount space for single level page table is 110

Multi-level Page Tables

- Benefits:
  - Save memory space
    - 1) Far less page table to individual process (locality), not need to include the entire table
    - 2) Second level page might not exist in memory, could be on the disk
  - High access efficiency due to table index

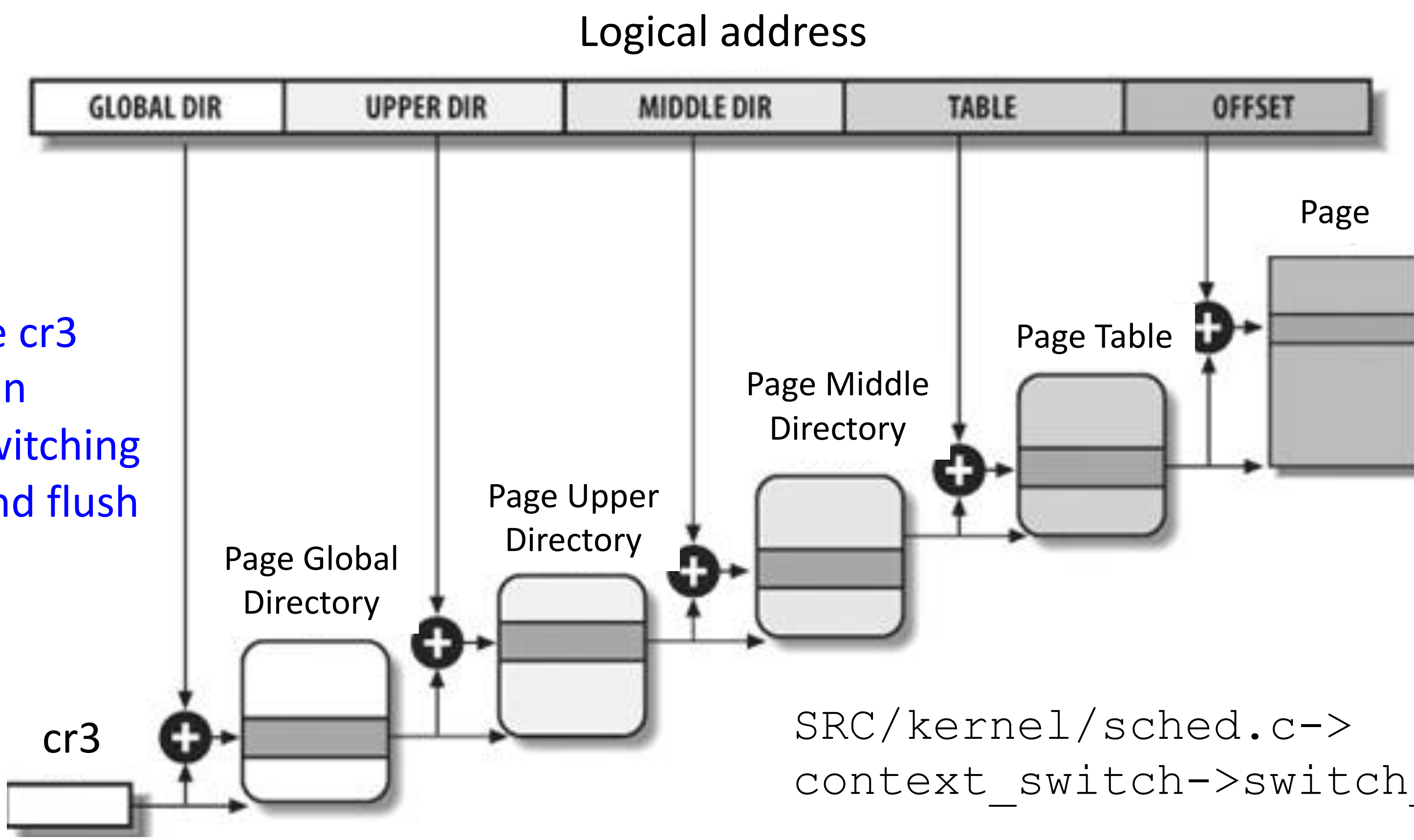
- Drawbacks:
  - Index space
  - Higher complexity with more levels

Multi-level Page Tables in Linux

https://elixir.bootlin.com/linux/latest/ident/pgdval_t

A common model for 32-bit (two-level, 4B pte) and 64-bit (four-level, 8B pte)

`SRC/include/linux/sched.h->task_struct->mm->pgd`

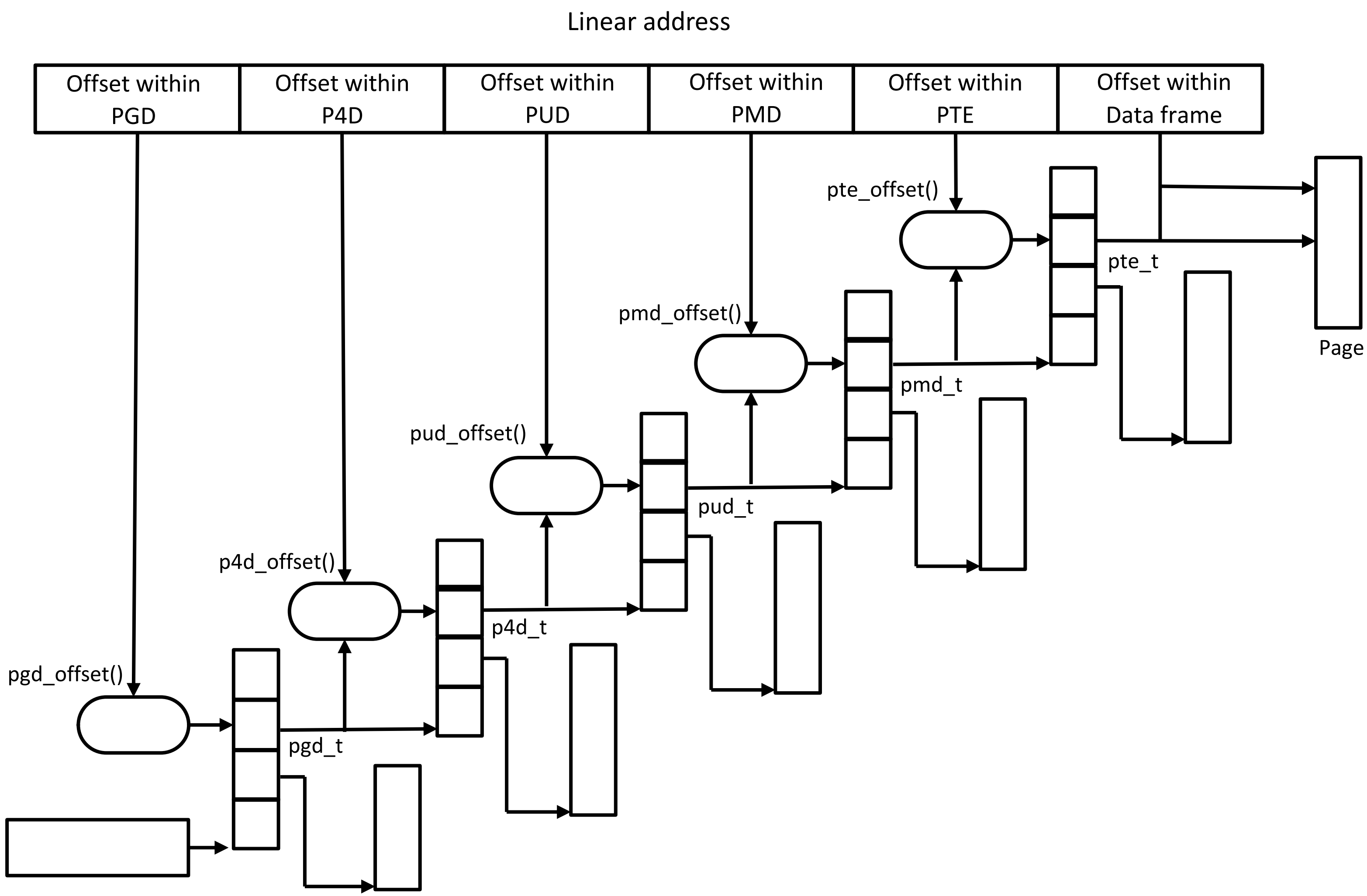


Switch the cr3 value when context switching threads and flush the TLB

`SRC/kernel/sched.c->context_switch->switch_mm`

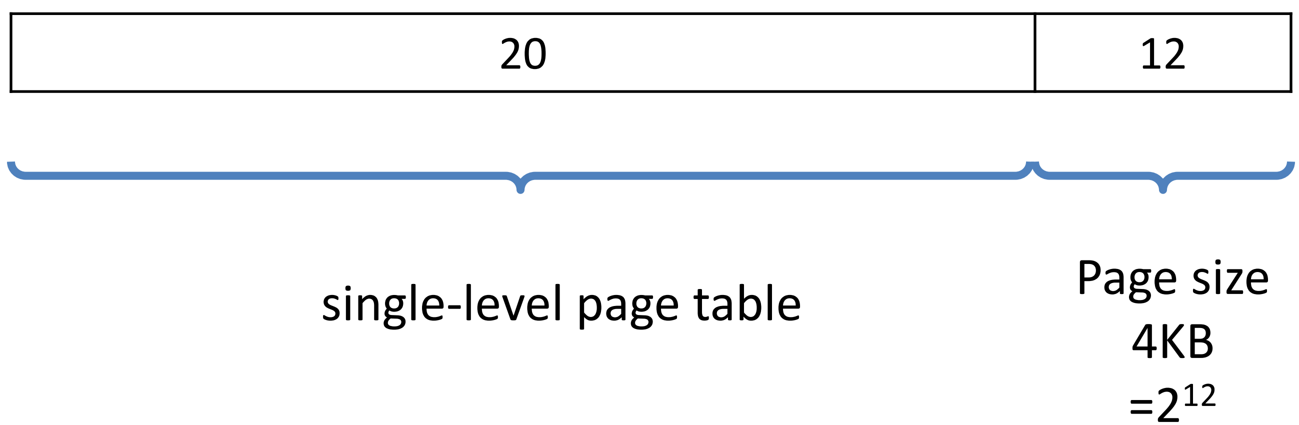
Multi-level Page Tables in Linux

https://elixir.bootlin.com/linux/latest/ident/pgdval_t



Multi-level Page Tables Example

- In a 32-bit system, a single-level page table with 4KB pages.



Multi-level Page Tables Example

- In a 32-bit system, a single-level page table with 4KB pages. Suppose PT1=1, offset=10, what is its virtual address?

20	12

$$\underbrace{20}_{\text{single-level page table}} \quad \underbrace{12}_{\text{Page size 4KB} = 2^{12}}$$

The virtual address = $1 * 2^{20} + 10 = 1048586$

Multi-level Page Tables Example

- In a 32-bit system, a four-level page table with 4KB pages. Each level is equal

ith-level
page length

5	5	5	5	12

4-level page table

Page size
4KB
$=2^{12}$

Multi-level Page Tables Example

- In a 32-bit system, a four-level page table with 4KB pages. Each level is equal. Suppose PT1=1, PT2=2, PT3=3, PT4=4, offset=10, what is the virtual address?

ith-level
page length

5	5	5	5	12

The virtual address = $1 * 2^{27} + 2 * 2^{22} + 3 * 2^{17} + 4 * 2^{12} + 10 = 143015946$

Conclusion

- Memory management overview
  - Memory abstraction and address spaces
  - Physical address and virtual address
  - Physical memory and virtual memory
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table