

# CS 3502

# Operating Systems

## Memory Management

Kun Suo

Computer Science, Kennesaw State University

<https://kevinsuo.github.io/>

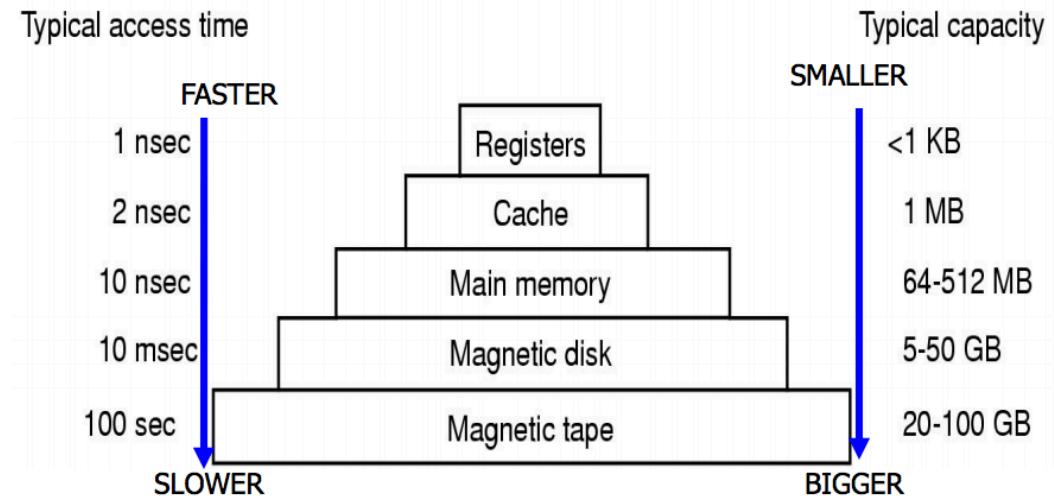
# Outline

---

- Memory management overview
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# Memory in the OS



# Outline

---

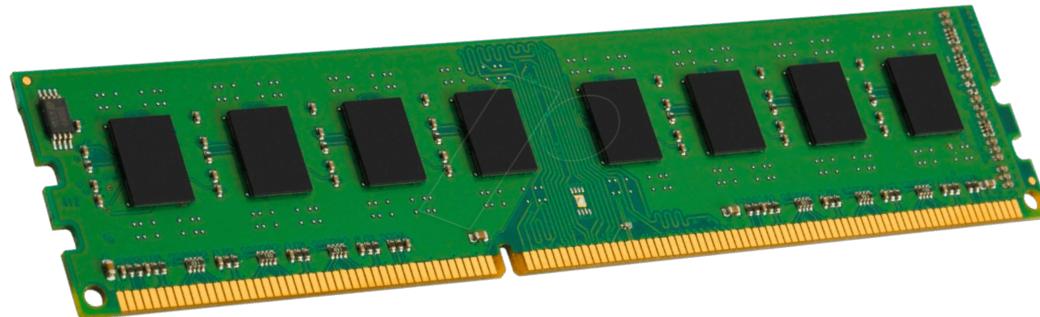
- Memory management overview
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# Why we need memory management

---

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



# Why we need memory management

Ideal: programmers/users want memory that is	Reality: Memory hierarchy
large	
fast	
Non-volatile	

**Memory management handles the semantic gap**



# 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 semantic gap**

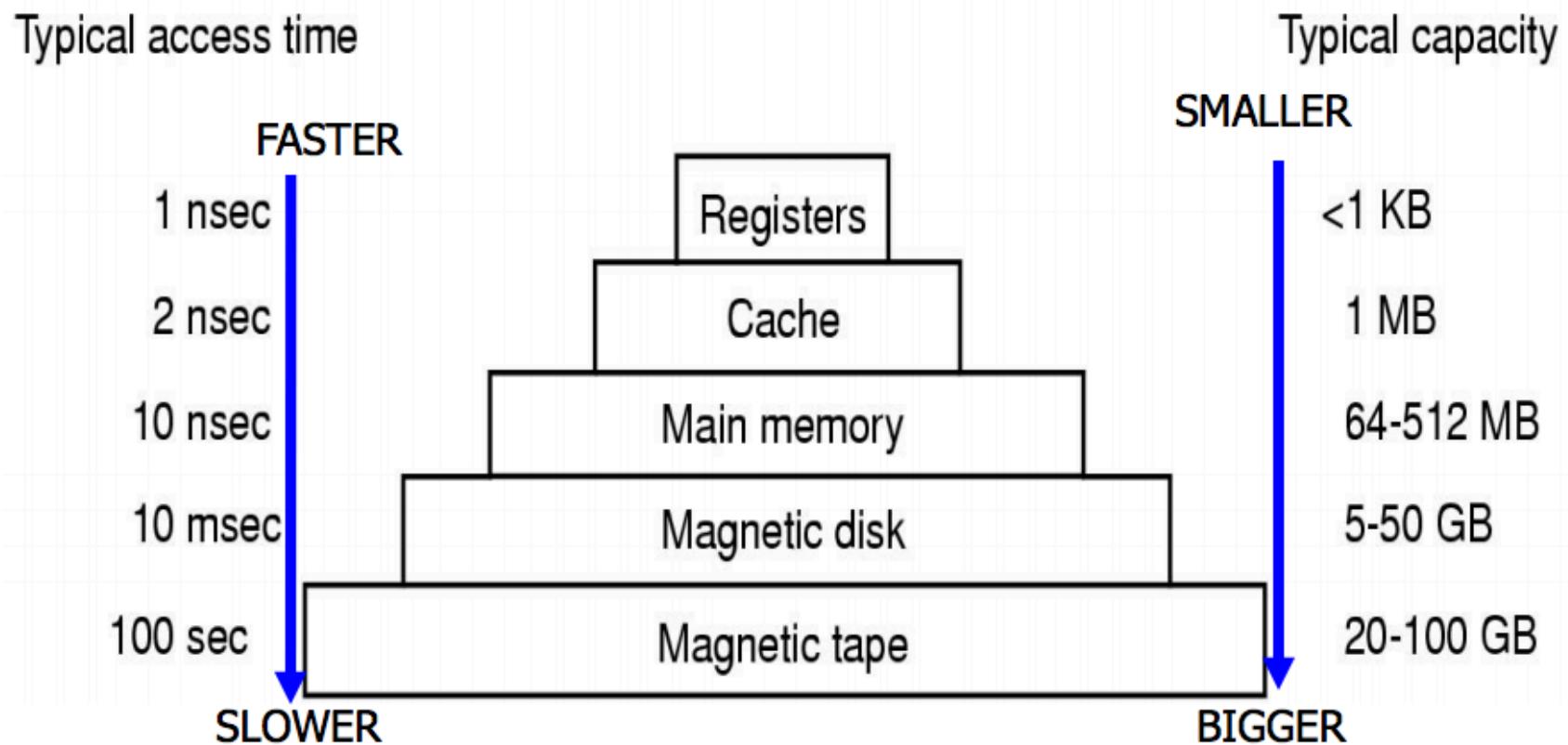
# Outline

---

- **Memory management overview**
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- **Memory management**
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# Typical Memory Hierarchy



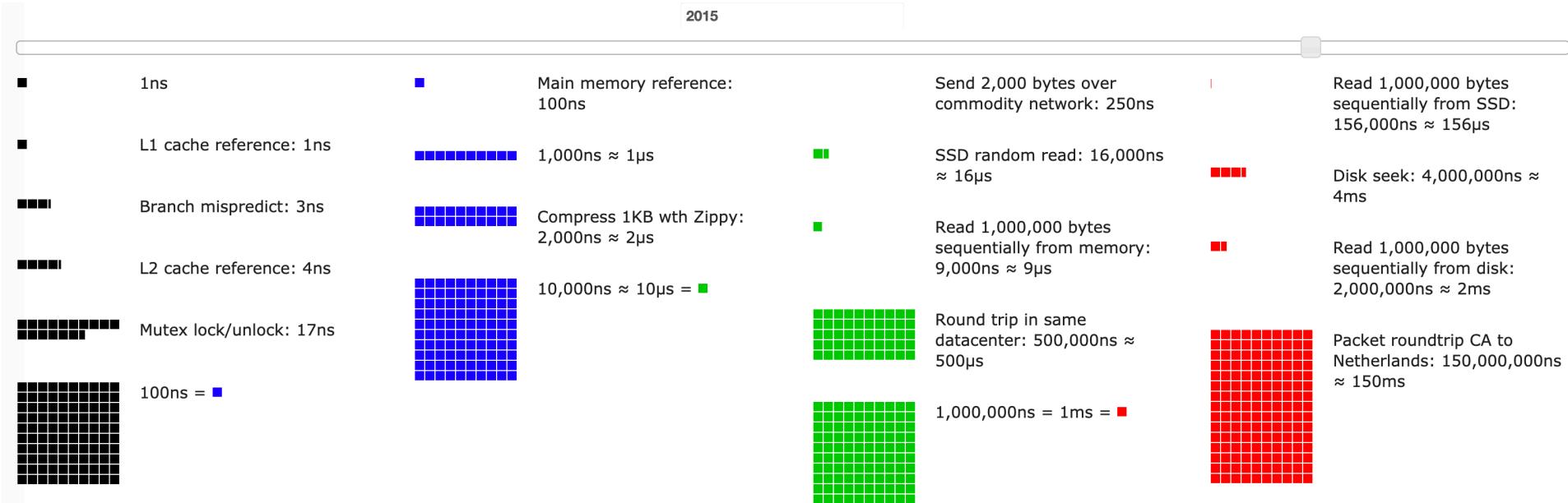
# Typical Memory Hierarchy

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

[www.brendangregg.com/sysperfbook.html](http://www.brendangregg.com/sysperfbook.html)



# Latency over the years

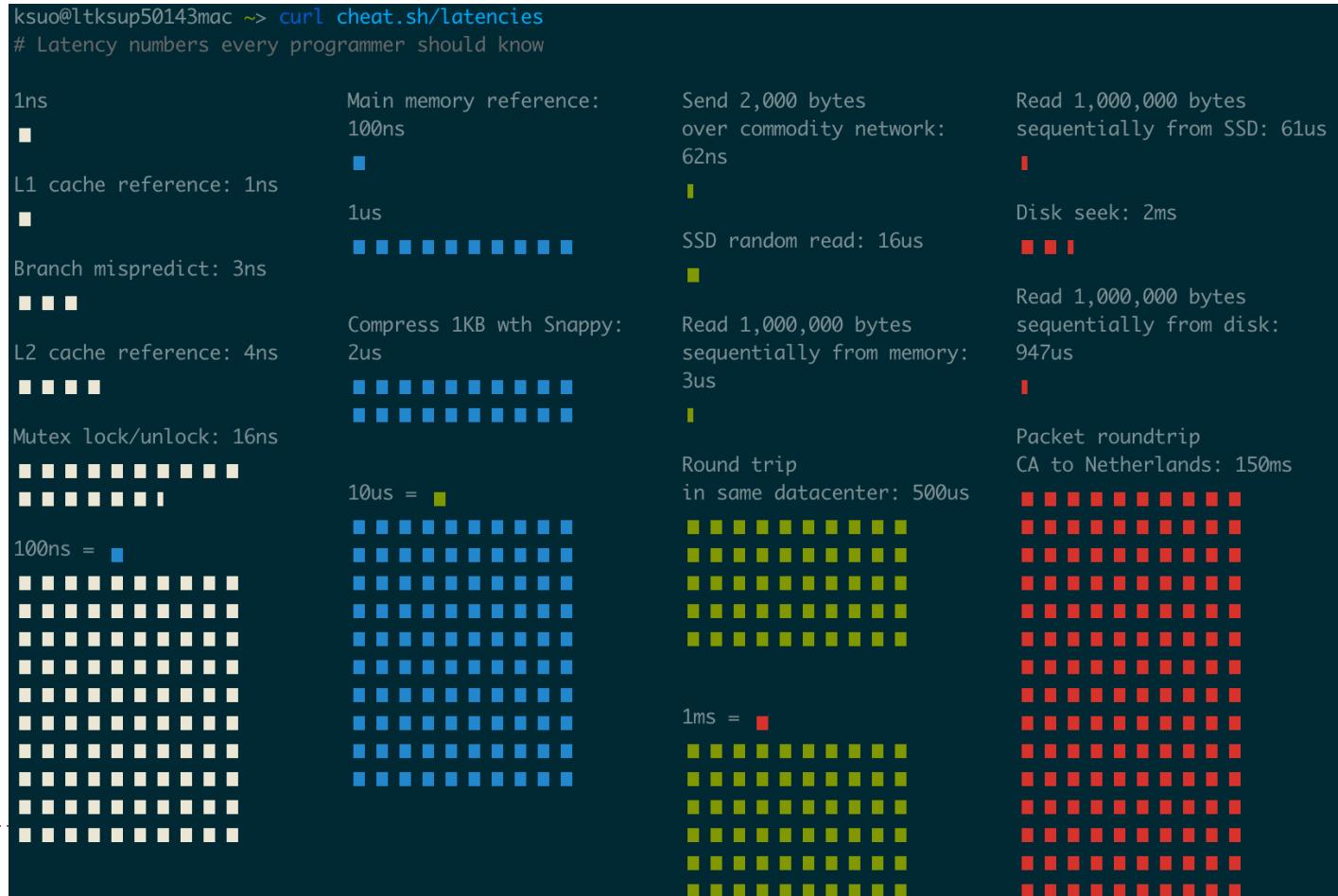


- [https://colin-scott.github.io/personal\\_website/research/interactive\\_latency.html](https://colin-scott.github.io/personal_website/research/interactive_latency.html)



# Latency numbers every programmer should know

- curl cheat.sh/latencies



# Test: what is the speed of your memory?

---

- STREAM Benchmark:

it can measure the performance of the memory system,  
including bandwidth and latency.

<https://www.cs.virginia.edu/stream/>

Function	Rate (MB/s)	Avg time	Min time	Max time
Copy:	16302.4084	0.0026	0.0020	0.0031
Scale:	13411.0440	0.0029	0.0024	0.0040
Add:	15529.6662	0.0041	0.0031	0.0050
Triad:	12889.0264	0.0042	0.0037	0.0047

# Test: what is the speed of your memory?

---

Copy

```
void tuned_STREAM_Copy()
```

```
{
```

```
    int j;
```

```
    for (j=0; j<N; j++)
```

```
        c[j] = a[j];
```

*Read from one memory cell  
Write to another memory cell*

```
}
```



# Test: what is the speed of your memory?

---

Scale

```
void tuned_STREAM_Scale(double scalar)
```

```
{
```

```
    int j;
```

```
    for (j=0; j<N; j++)
```

```
        b[j] = scalar*c[j];
```

```
}
```

*Read from one memory cell  
Make a multiply operation  
Write to another memory cell*



# Test: what is the speed of your memory?

---

Sum

```
void tuned_STREAM_Add()
```

```
{
```

```
    int j;
```

```
    for (j=0; j<N; j++)
```

```
        c[j] = a[j]+b[j];
```

```
}
```

*Read from two memory cell  
Make an add operation  
Write to another memory cell*



# Test: what is the speed of your memory?

---

Triad

```
void tuned_STREAM_Triad(double scalar)
```

```
{
```

```
    int j;
```

```
    for (j=0; j<N; j++)
```

```
        a[j] = b[j]+scalar*c[j];
```

```
}
```

*Read from two memory cell*

*Make an add operation and a multiply operation*

*Write to another memory cell*



# Test: what is the speed of your memory?

---

- 1, Download STREAM Benchmark

```
$ wget https://www.nersc.gov/assets/Trinity--NERSC-8-RFP/Benchmarks/Jan9/stream.tar
```

- 2, Compile & run

```
$ tar xvf stream.tar
```

```
$ gcc stream.c -o stream
```

```
$ ./stream
```



# Test: what is the speed of your memory?

---

- 1, install memory bandwidth app

```
$ sudo apt-get install mbw
```

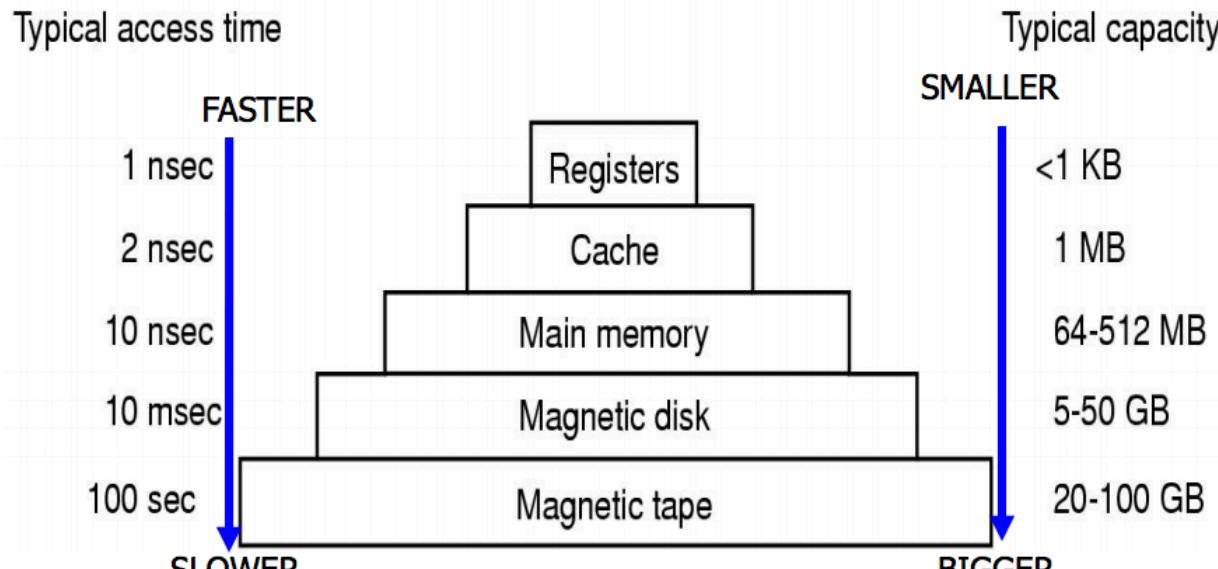
- 2, run

```
$ mbw -q -n 100 256
```

n: run 100 times

256: means test 256MB in memory

# Why we need memory hierarchy like this?



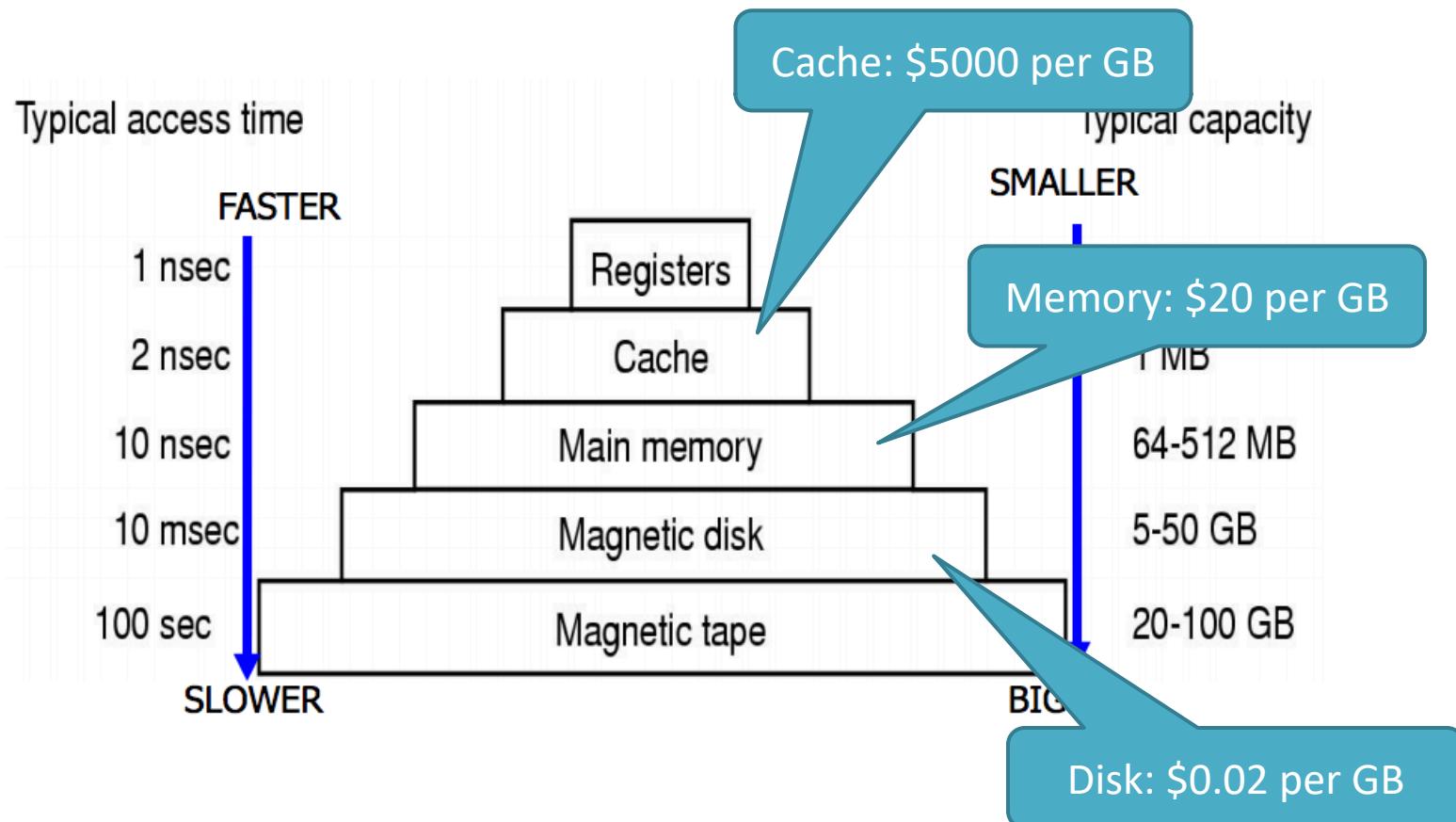
Single layer memory hierarchy?

Memory

# Why we need memory hierarchy like this?

- Cost

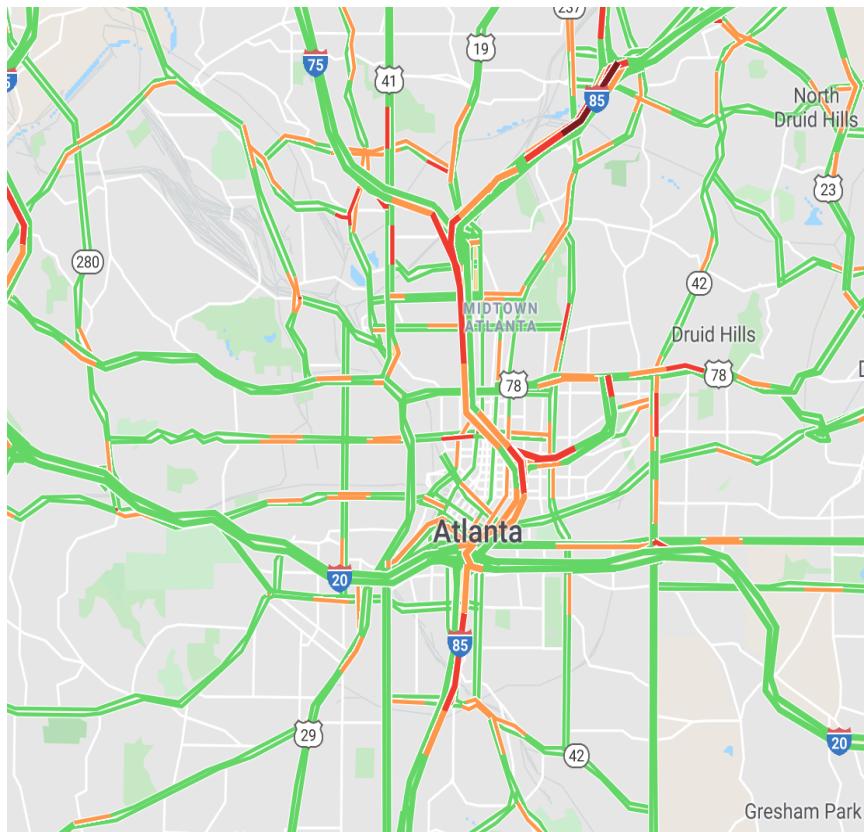
Year: 2010 data



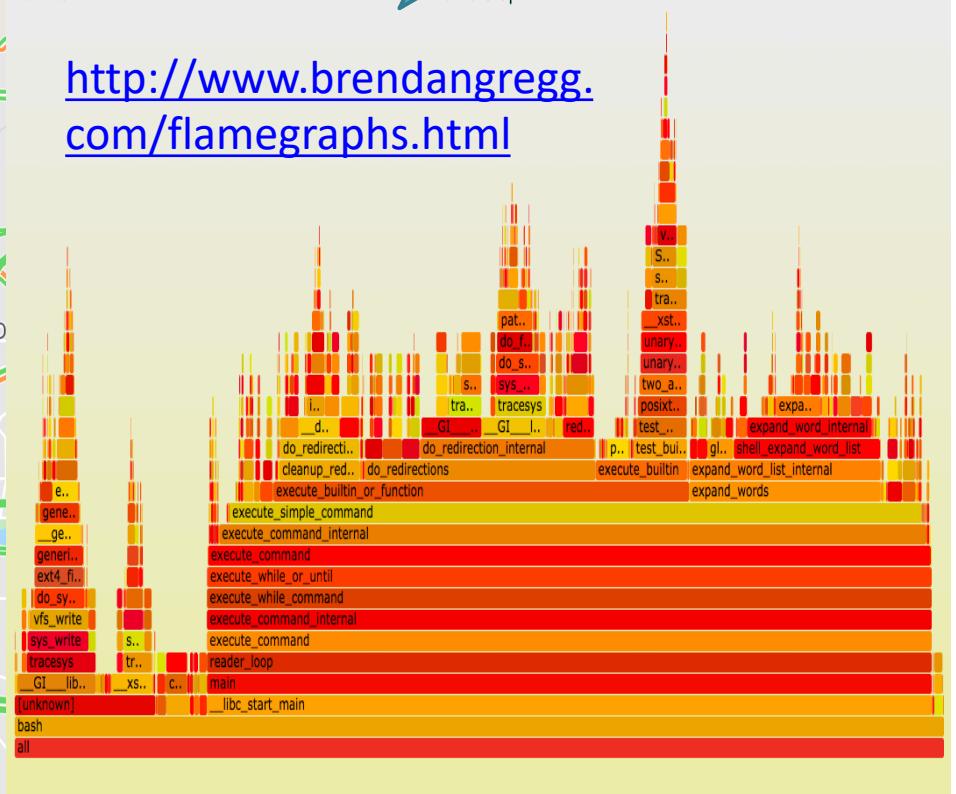
# Why we need memory hierarchy like this?

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

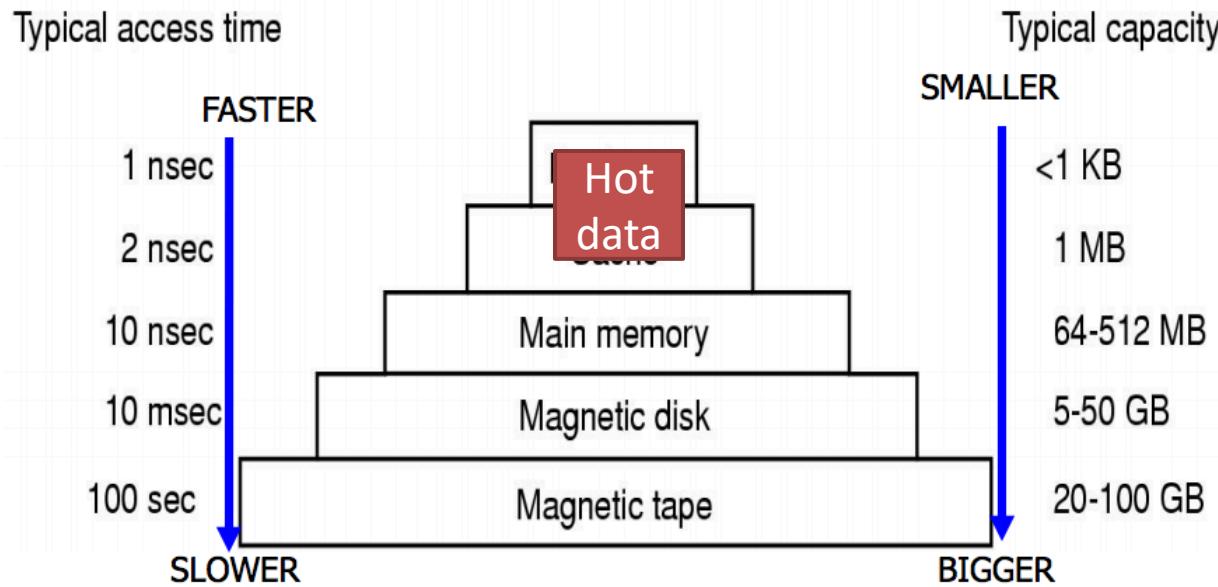
We only need to put the hot code inside the top hierarchy of memory



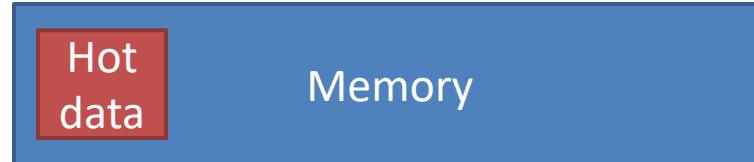
<http://www.brendangregg.com/flamegraphs.html>



# Why we need memory hierarchy like this?



Single layer memory hierarchy?



# Outline

---

- Memory management overview
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# 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)

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

int main()
{
    int n, i, *ptr, sum = 0;

    printf("Enter number of elements: ");
    scanf("%d", &n);

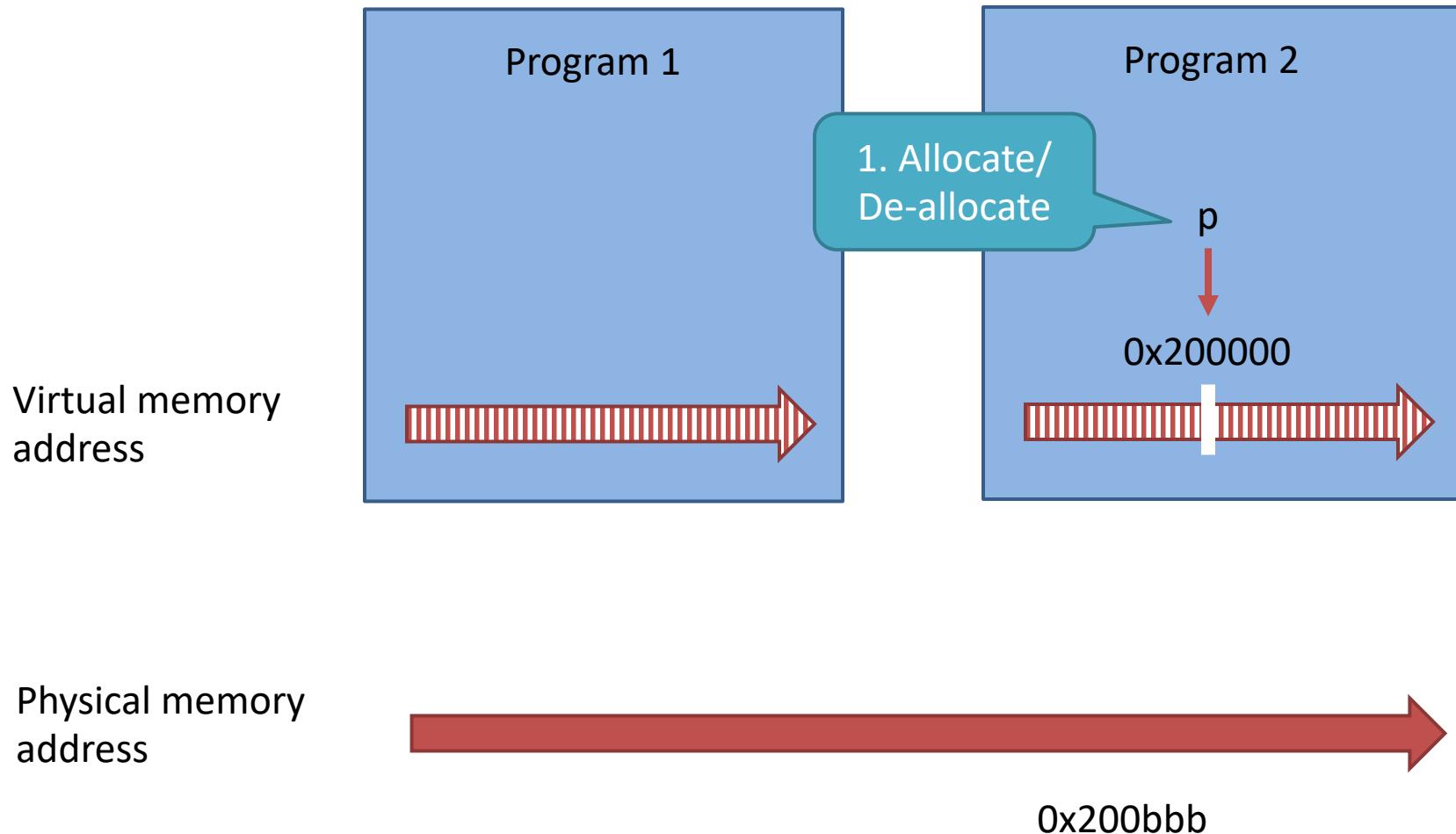
    ptr = (int*) malloc(n * sizeof(int));
```

```
    printf("Sum = %d", sum);

    // deallocated the memory
    free(ptr);

    return 0;
}
```

# What will memory management do?



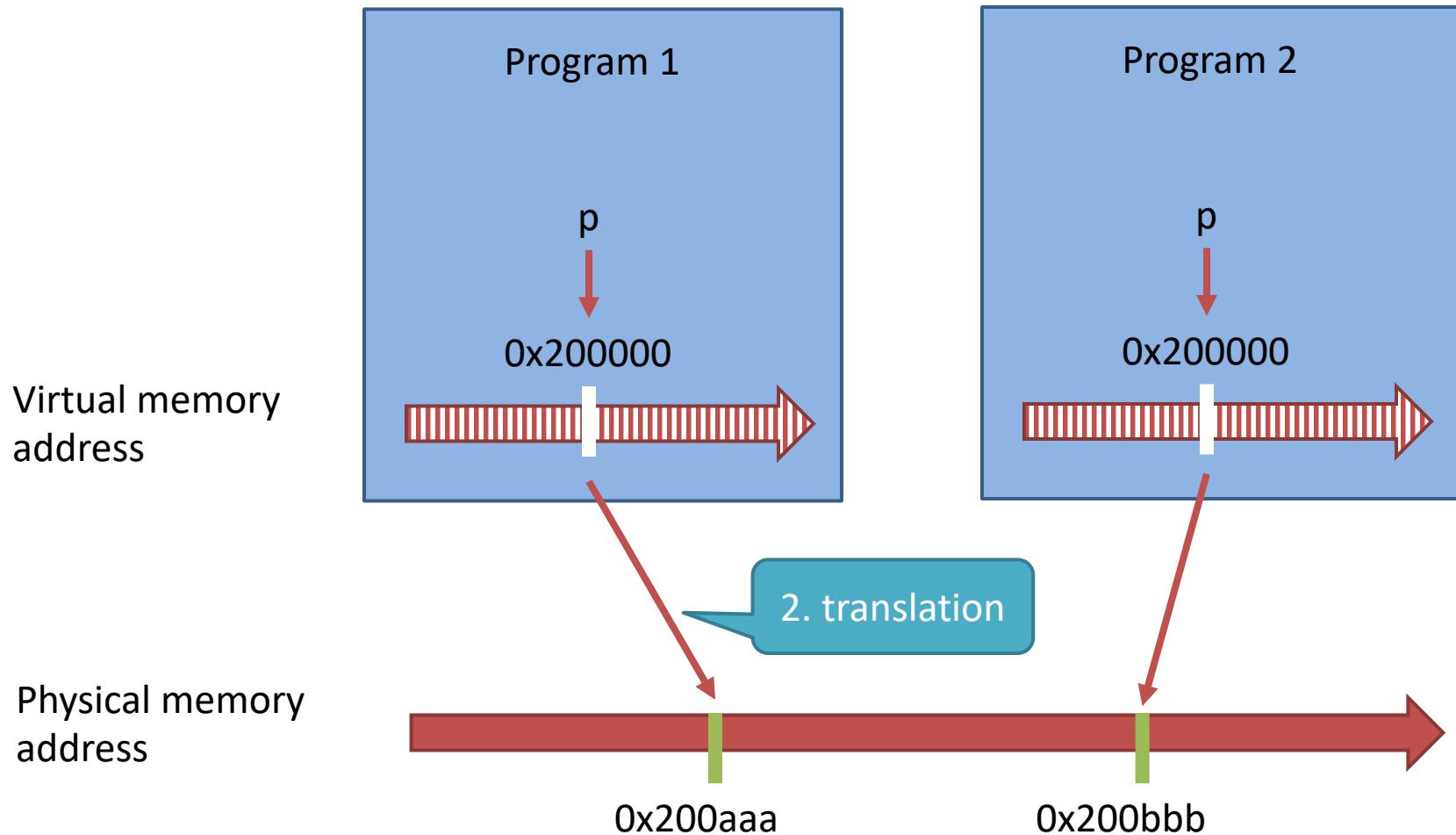
# What will memory management do?

---

- 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)



# What will memory management do?



# What will memory management do?

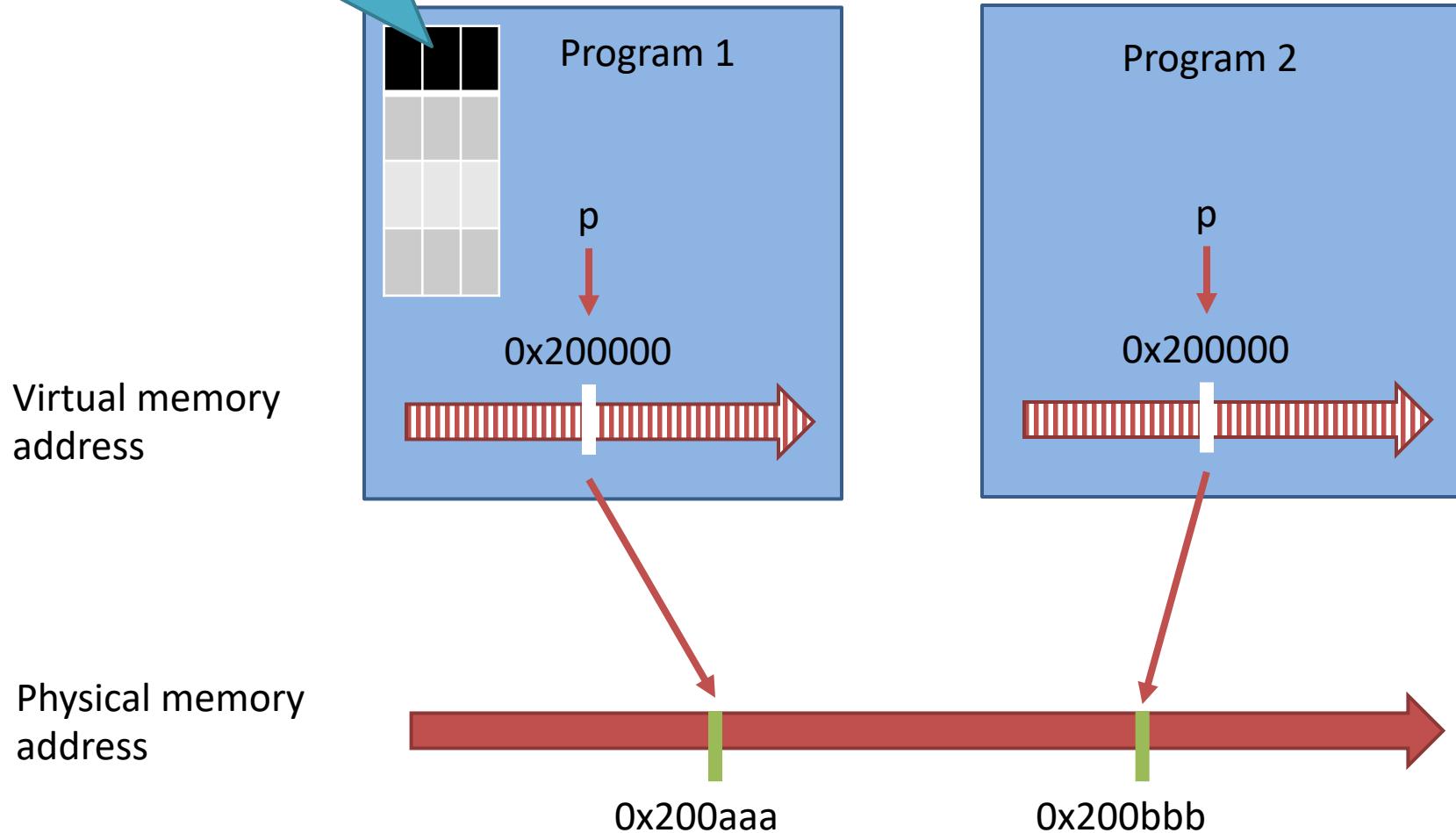
---

- 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



3. Keep track of used memory and by whom  
(try: nmon, then press "t")

# memory management do?



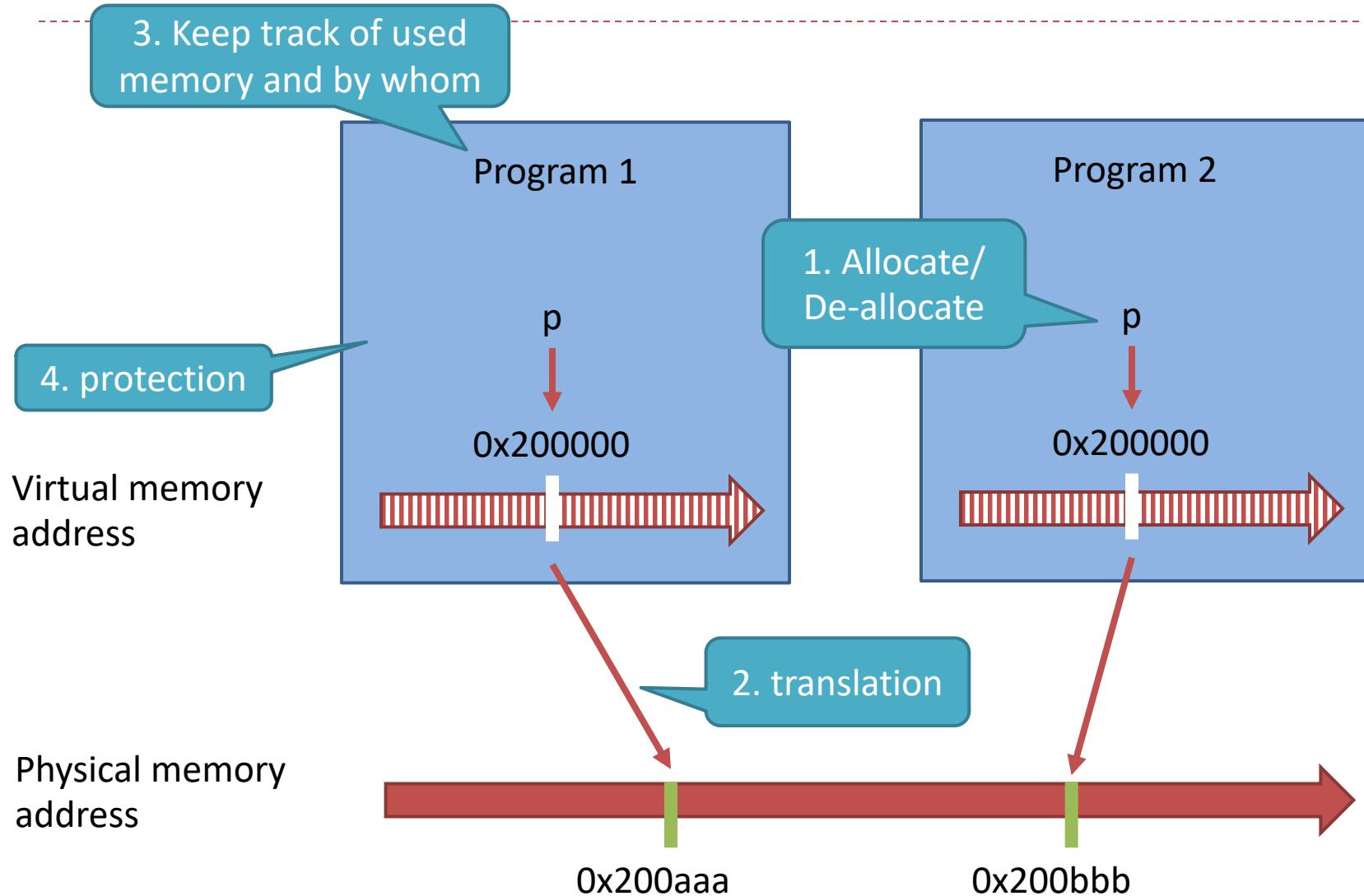
# What will memory management do?

---

- 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



# What will memory management do?



# Outline

---

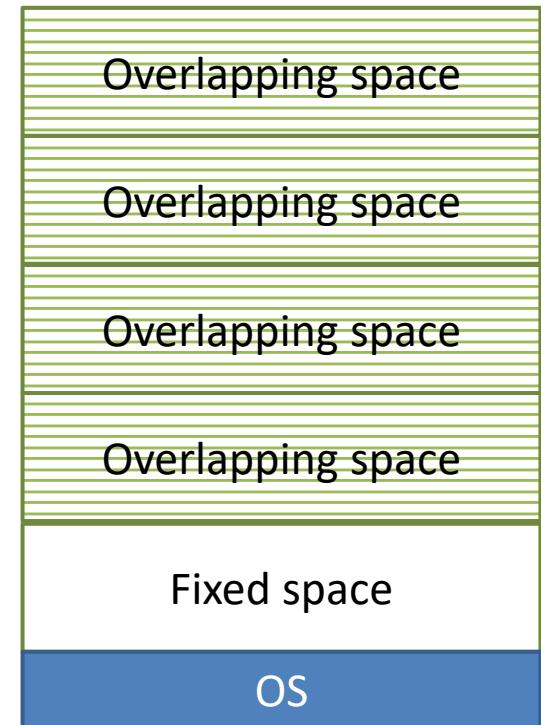
- Memory management overview
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# What will happen when memory is not enough?

- Overlapping

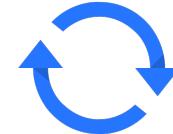
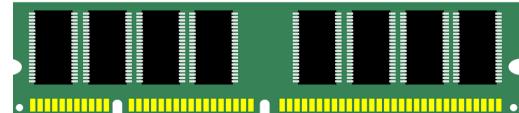
- Memory is small and cannot hold all program data
- User space is divided to one fixed space and several overlapping space
- Fixed space: hold most frequent data
- Overlapping space: the data will overlap those which never be used (after overlapping, **data is gone**)



# 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



# What will happen when memory is not enough?

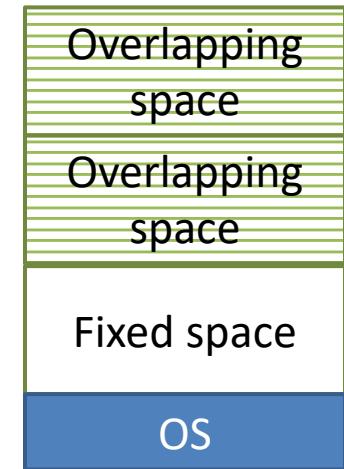
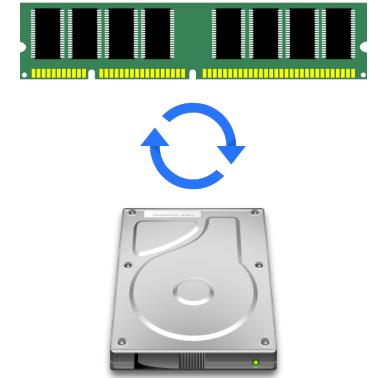
The screenshot shows a macOS desktop environment with three windows:

- Terminal Window 1:** Shows a root shell session on a host named "LinuxKernel2". It runs a stress test with 2 virtual cores, 16G bytes, and a 60-second timeout. The session ends with an exit command.
- Terminal Window 2:** Shows a root shell session on the same host. It runs a docker container with the same stress test configuration.
- nmon Monitor Window:** Monitors system performance. The CPU Utilisation tab shows four cores at 100% usage. The Memory and Swap tab shows low free memory (14677.7 MB total, 4096.0 MB swap). The Global-CPU-Stats tab shows high CPU usage (0.89 user load average) and context switches (78.4).

# What will happen when memory is not enough?

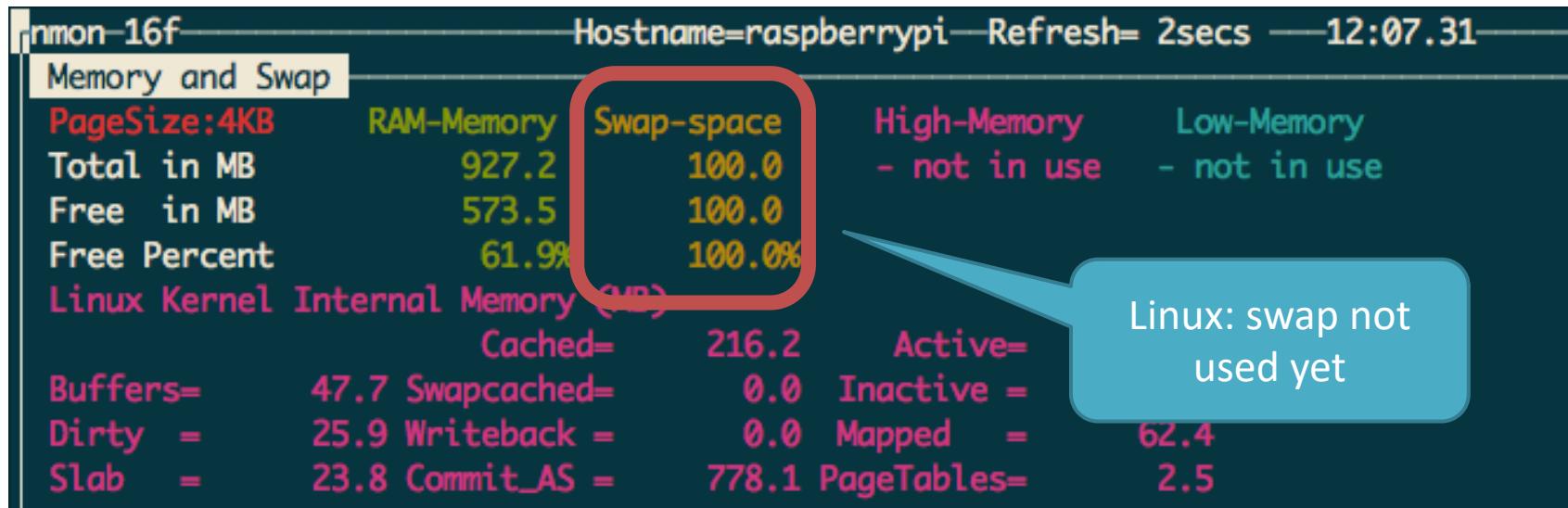
- Difference

- 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)

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	60,1 MB
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	
Google Chrome	103,2 MB	57,8 MB	10	120	611	baunov	

MEMORY PRESSURE

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

Available Memory: 2,01 GB  
Wired Memory: 3,70 GB  
Compressed: 1,70 GB

Mac: 3GB swap used

# Outline

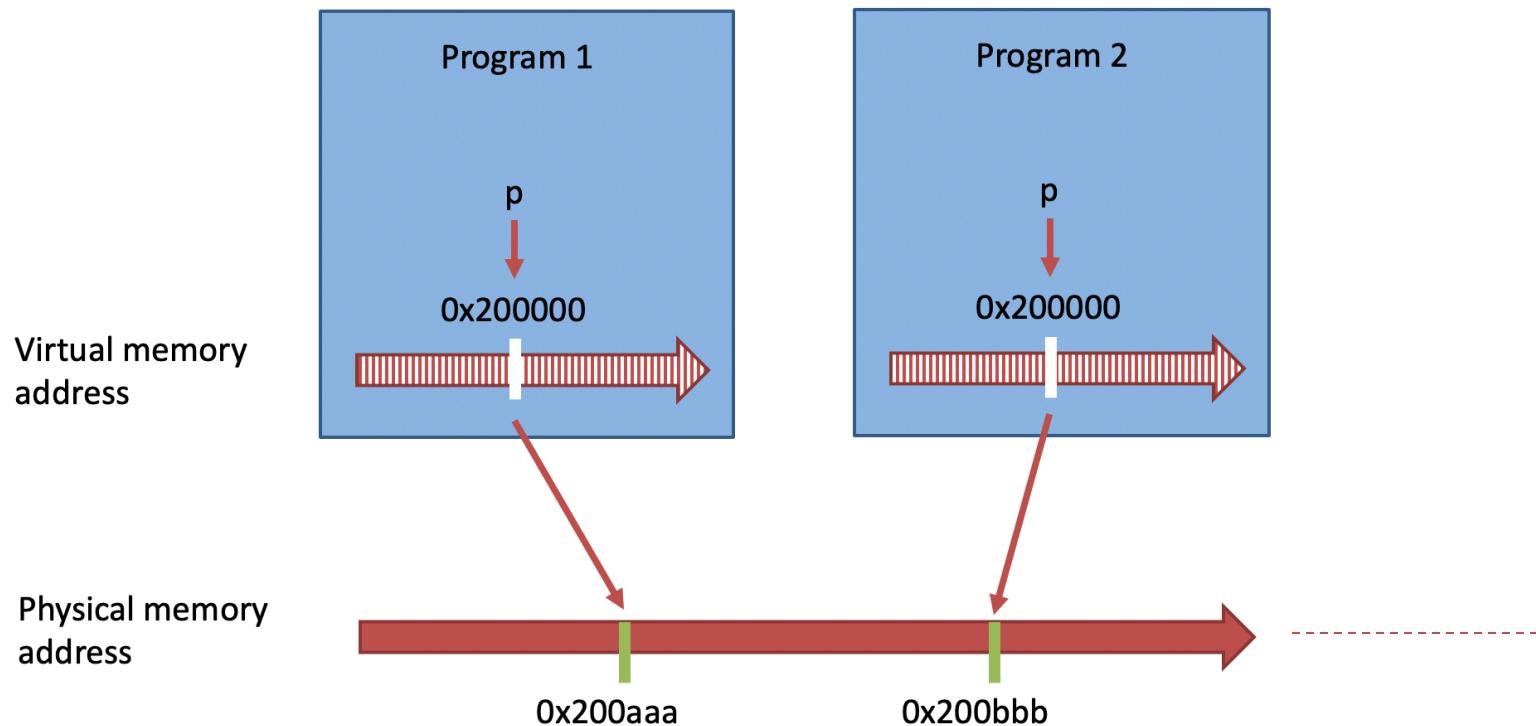
---

- **Memory management overview**
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- **Memory management**
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# Memory Abstraction

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



# 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**)

```
#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);

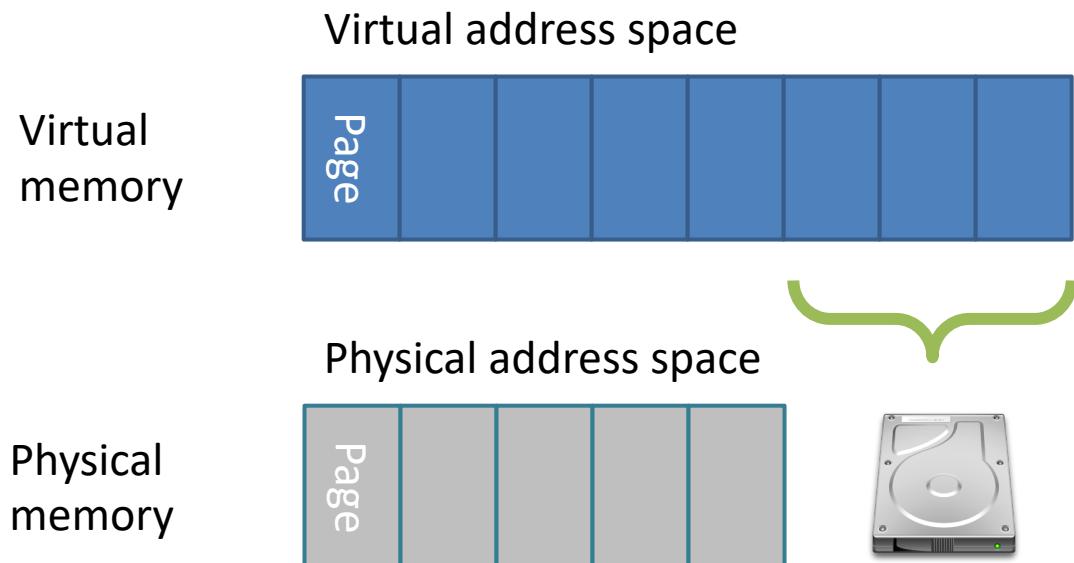
    return 0;
}
```

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**.



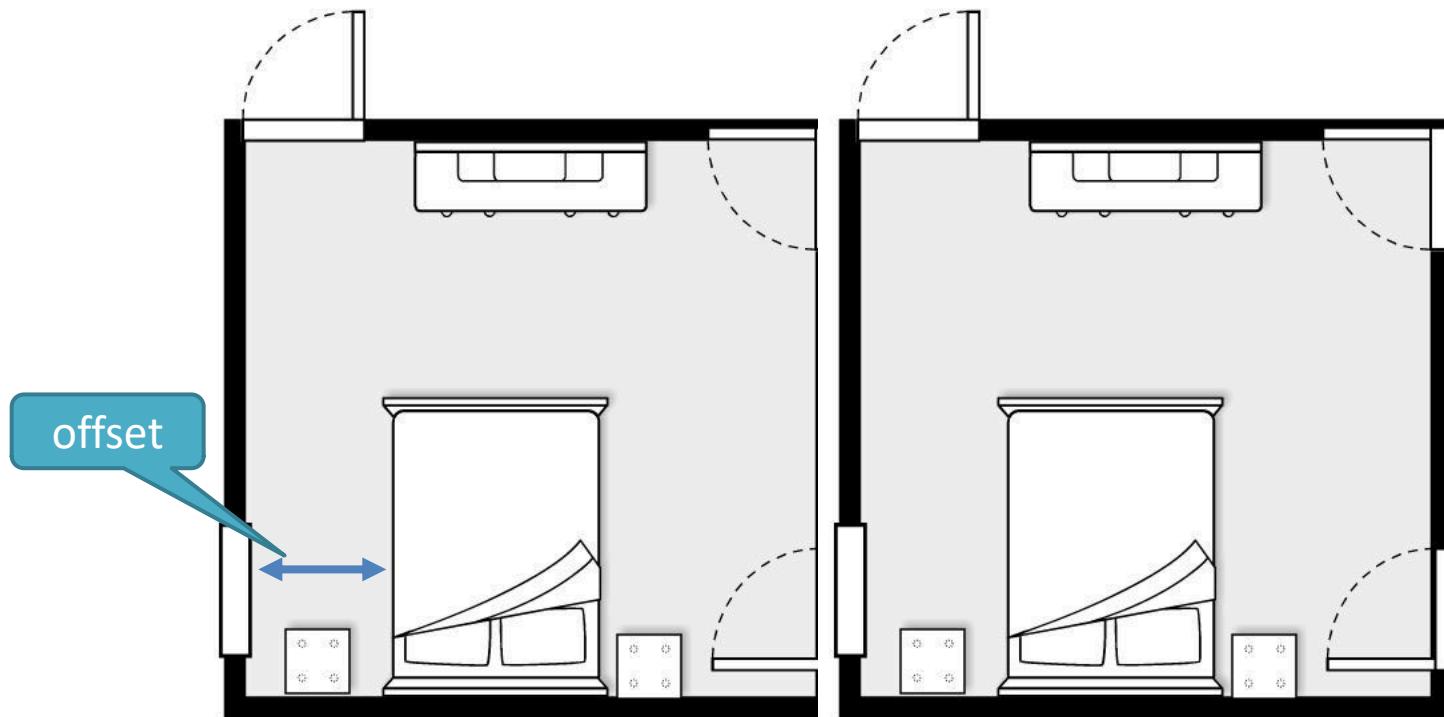
# How to organize memory? Page number and offset



	JR PROPERTIES INC
JR Property Co.	SUITE 101
Dawson Pools, LLC	SUITE 102
The Saytman Group	SUITE 103
Carghill	SUITE 201
Laymen & Associates	SUITE 202
BrandyWood Medical	SUITE 203
FoundAbility.com	THIRD FLOOR

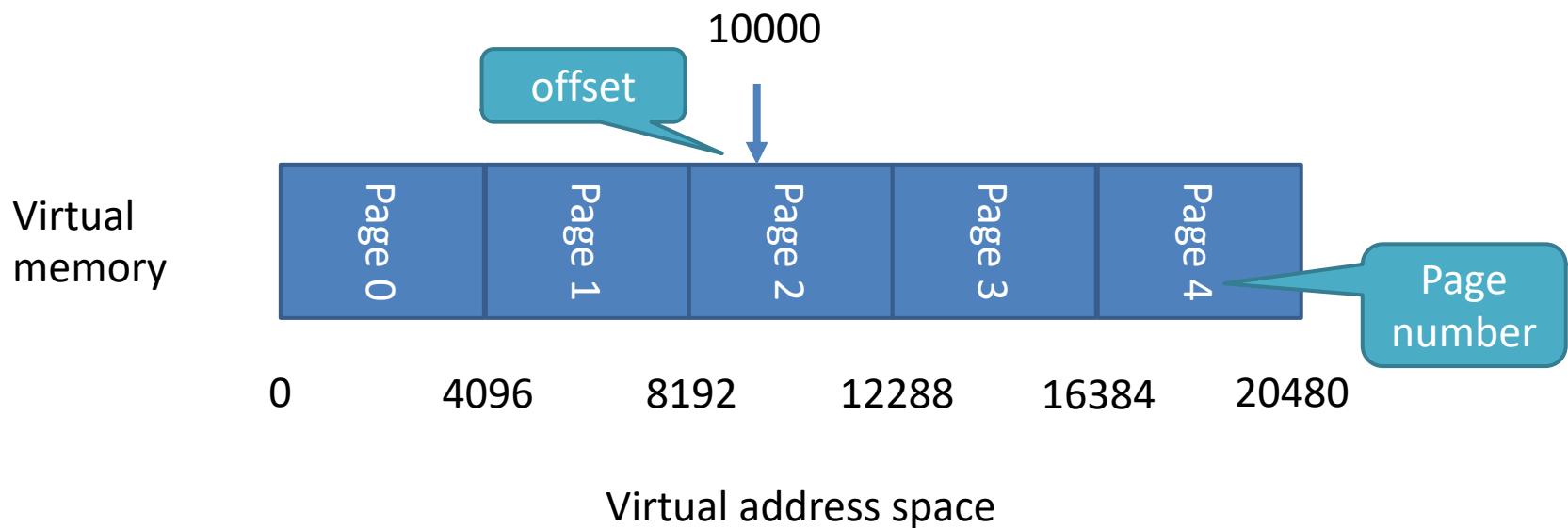


# How to organize memory? Page number and offset



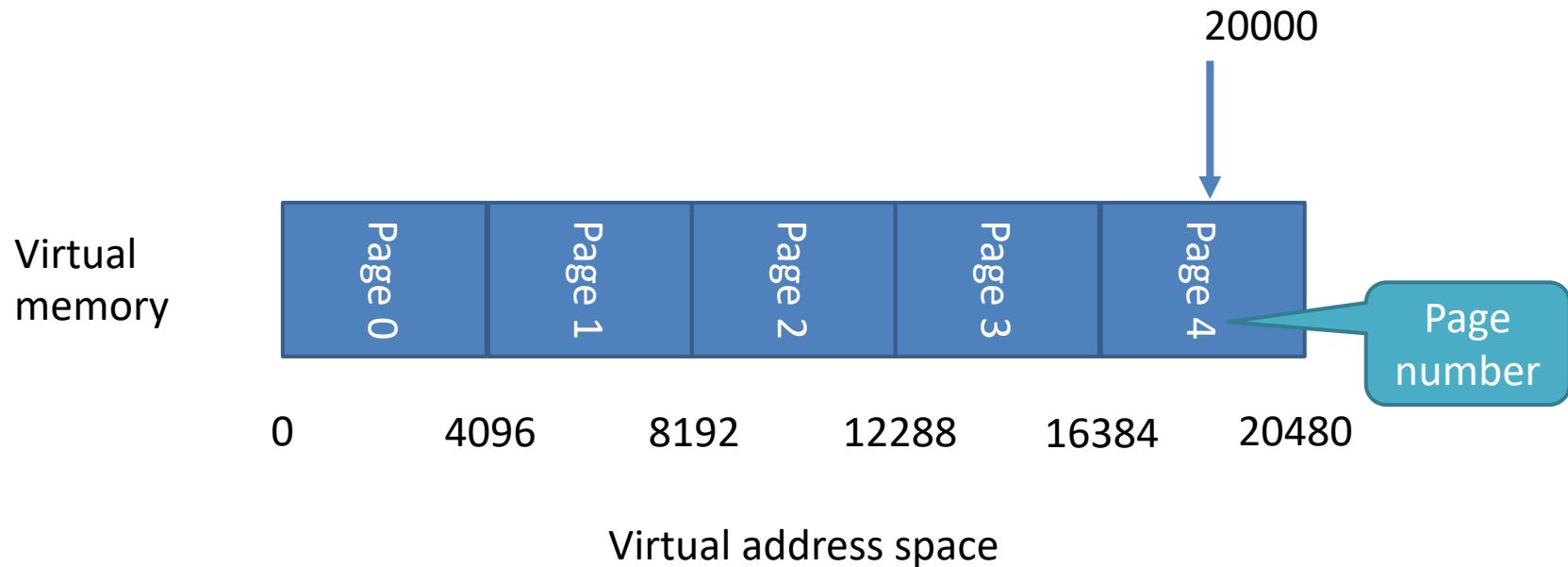
# How to organize memory? Page number and offset

- Virtual address
  - *Page number: which page the address is in*
  - *Page offset: the distance between this address and the start of that page*



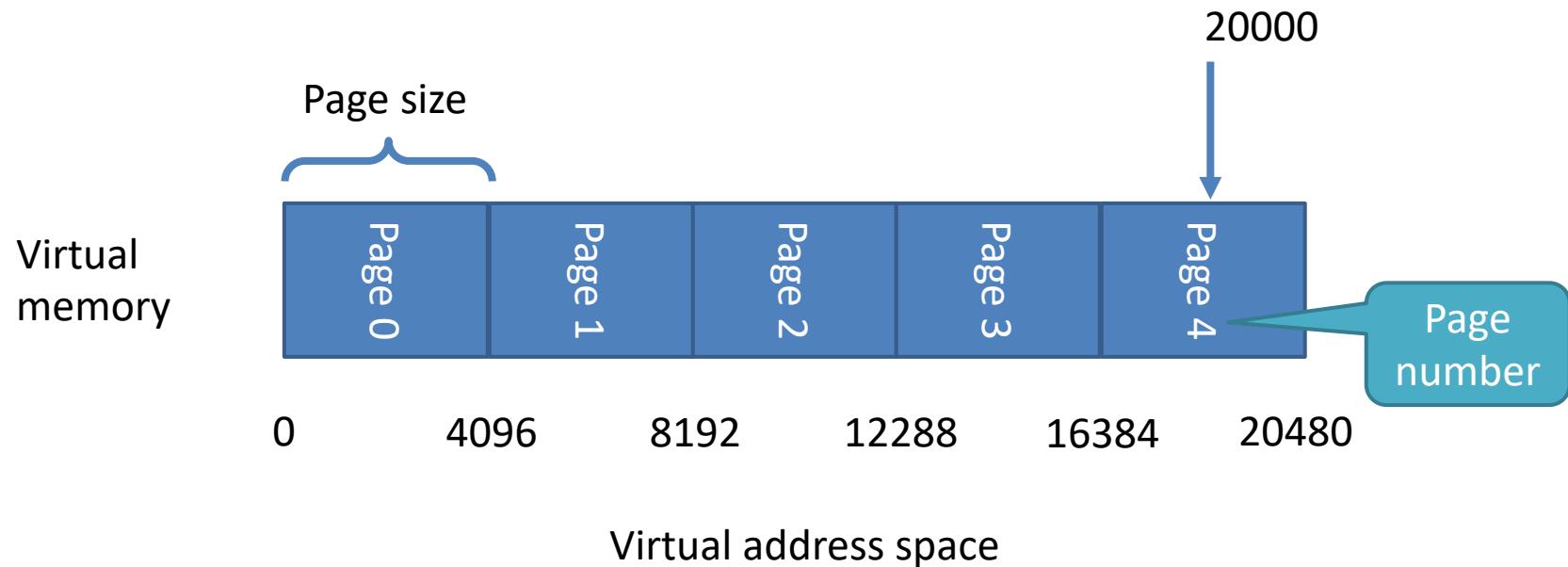
# How to organize memory? Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - The virtual page number is  $20000/4096 = 4$



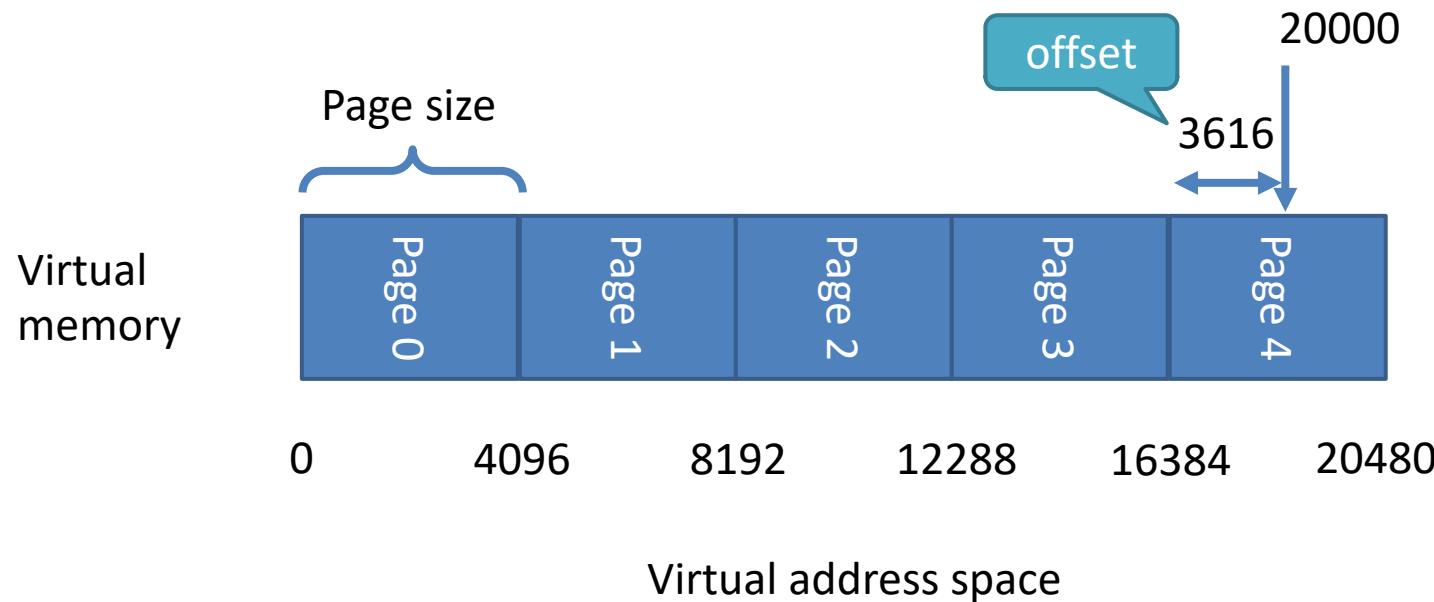
# How to organize memory? Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - page number = address / page size



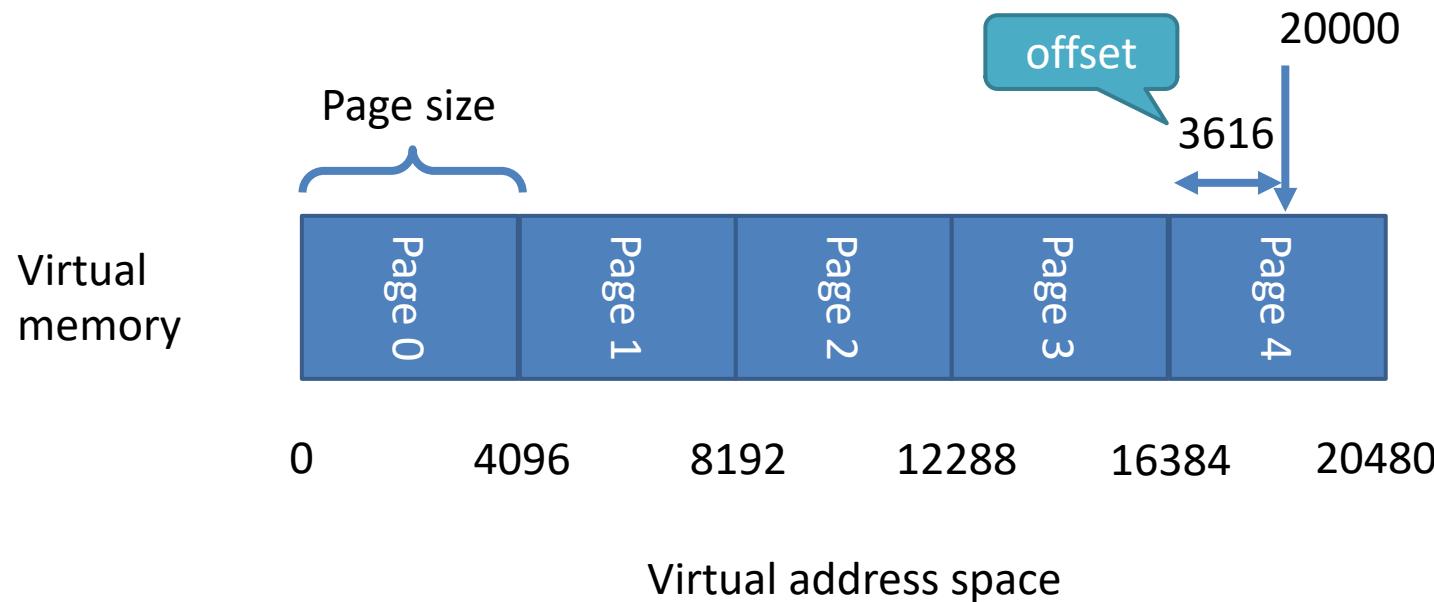
# How to organize memory? Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - The offset is  $20000 \% 4096 = 3616$



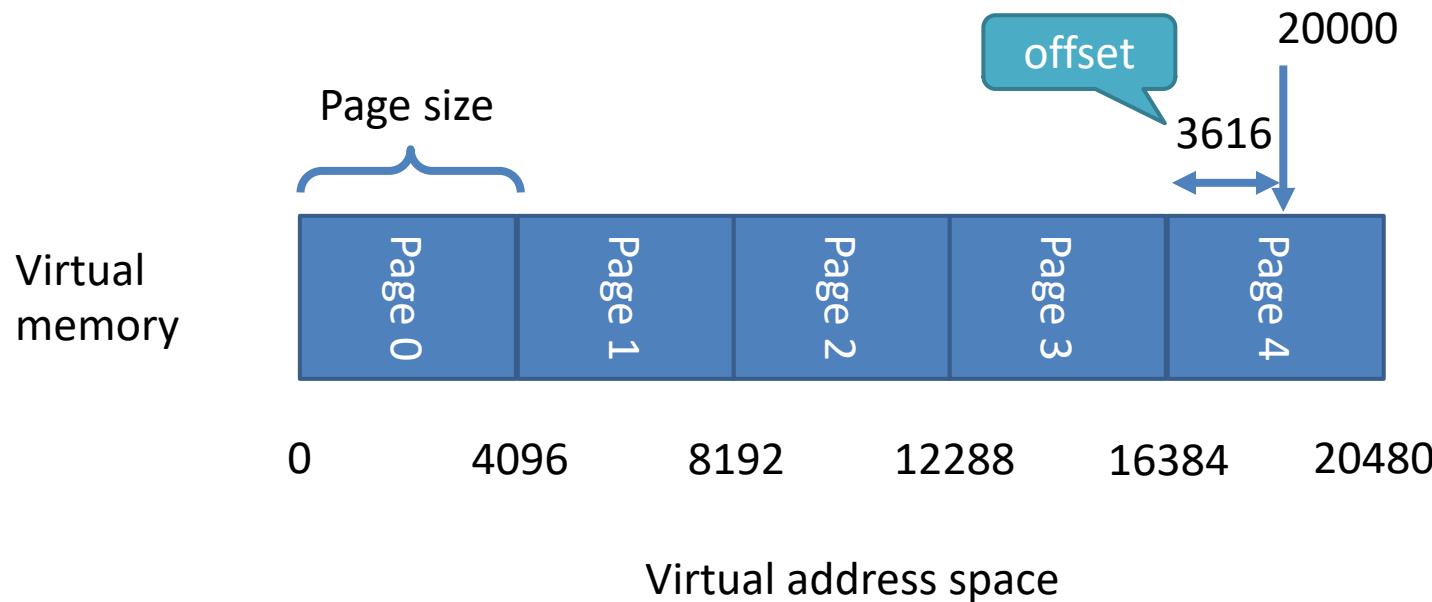
# How to organize memory? Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - offset = address % page size



# How to organize memory? Page number and offset

- Suppose the page size is 4KB, the address is 20000
  - page number = address / page size
  - offset = address % page size



# Question

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

## 1. What is the memory size?

(32-bit can represent how many different numbers?)

$$2^{32} = 4\text{GB}$$



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

$$0 \quad 4k \quad 2^{32}$$

# Question

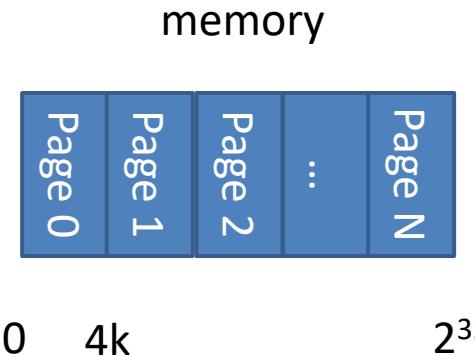
- Consider a machine that has a 32-bit virtual address space and 4K Byte page size.

2. How many virtual pages could a process have?

The total memory is 4GB

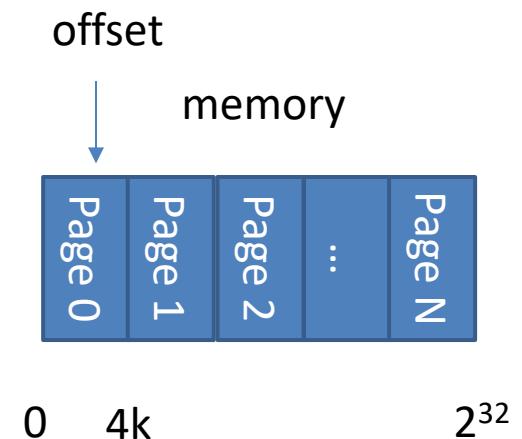
The single page size is 4KB

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



# 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 ?

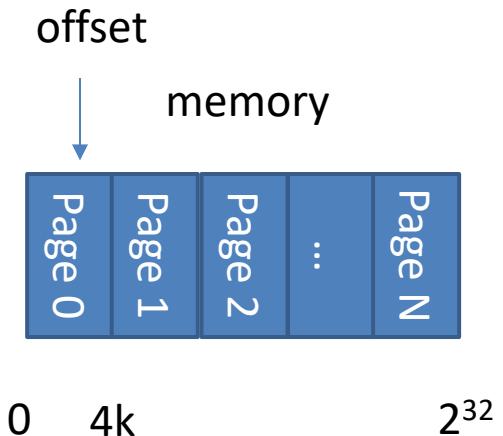


# Question

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

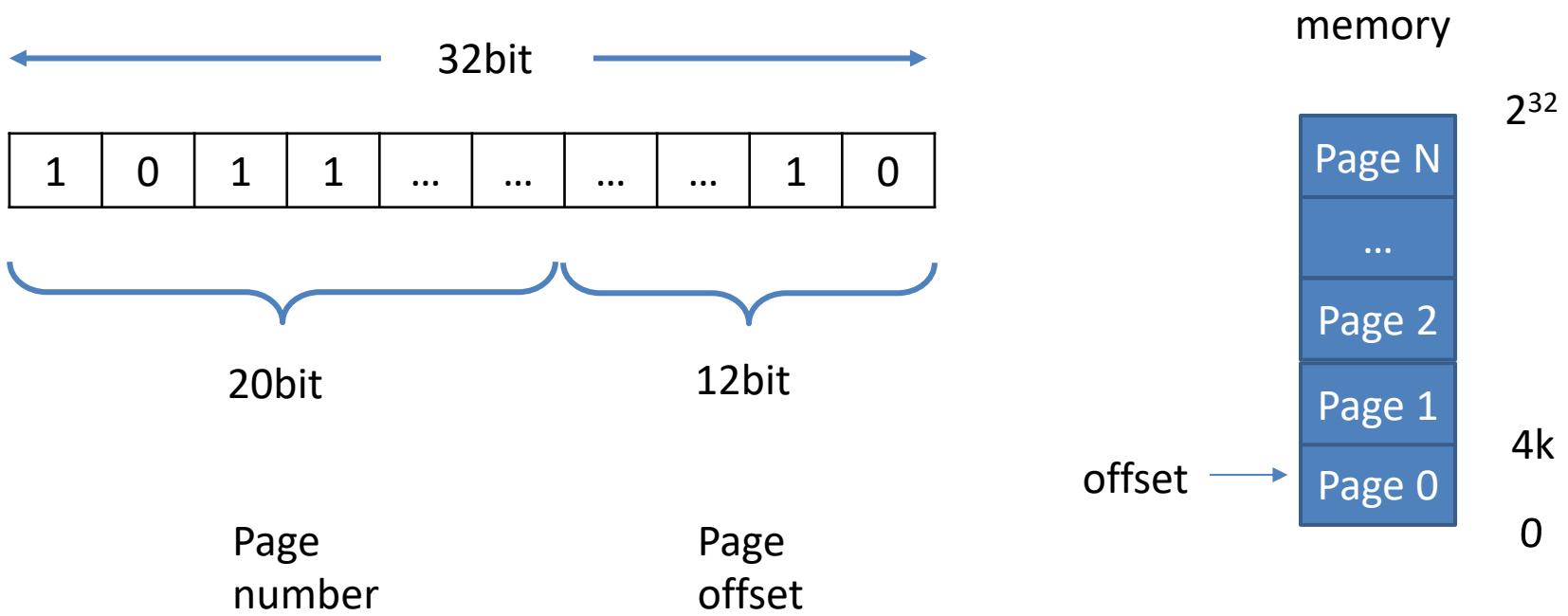
3. Given a 4KB page, how many address bits do we need for the offset ?

$$\log_2(4\text{KB}) = \log_2(2^{12}) = 12$$



# 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.
  - What is the total size of the memory (virtual address) space?

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

# 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 =  $4\text{GB}/8\text{KB} = 512 * 1024 = 2^9 * 2^{10} = 2^{19}$

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

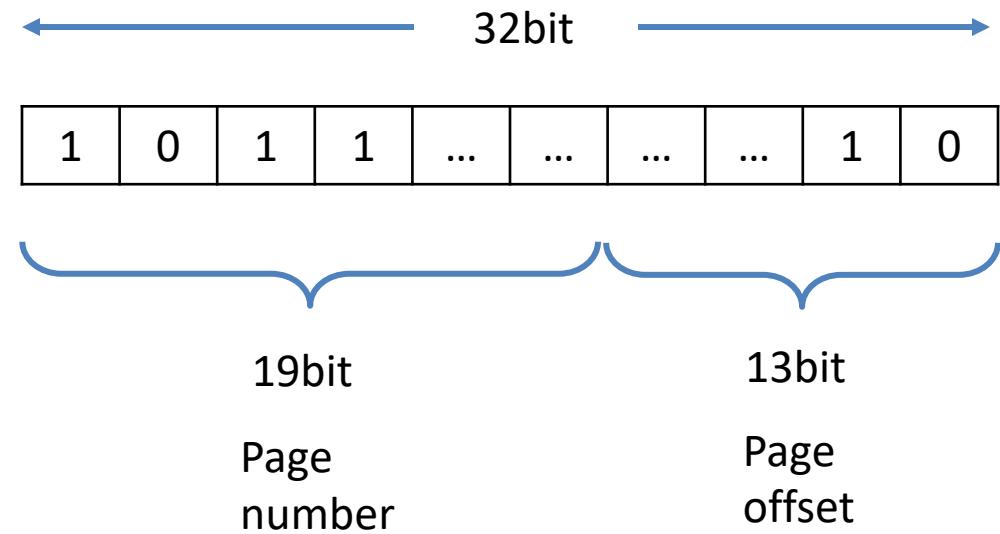


# Question

- 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?

$$32 - 19 = 13 \text{ bits}$$



# Question

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

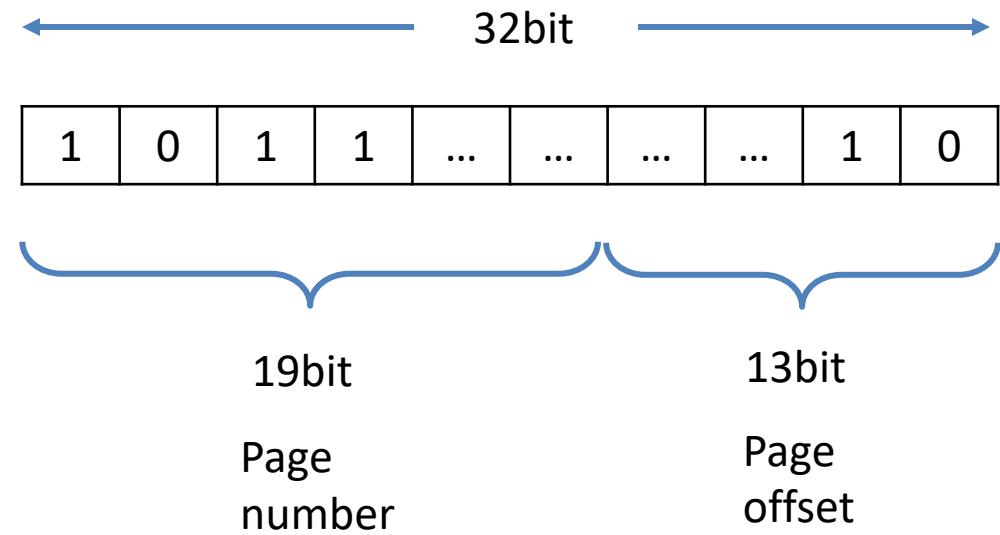
4. How many page-table entries or how many pages we have?

Number of PTEs

= Number of pages in virtual address

=  $4\text{GB}/8\text{KB}$

=  $2^{19}$  pages



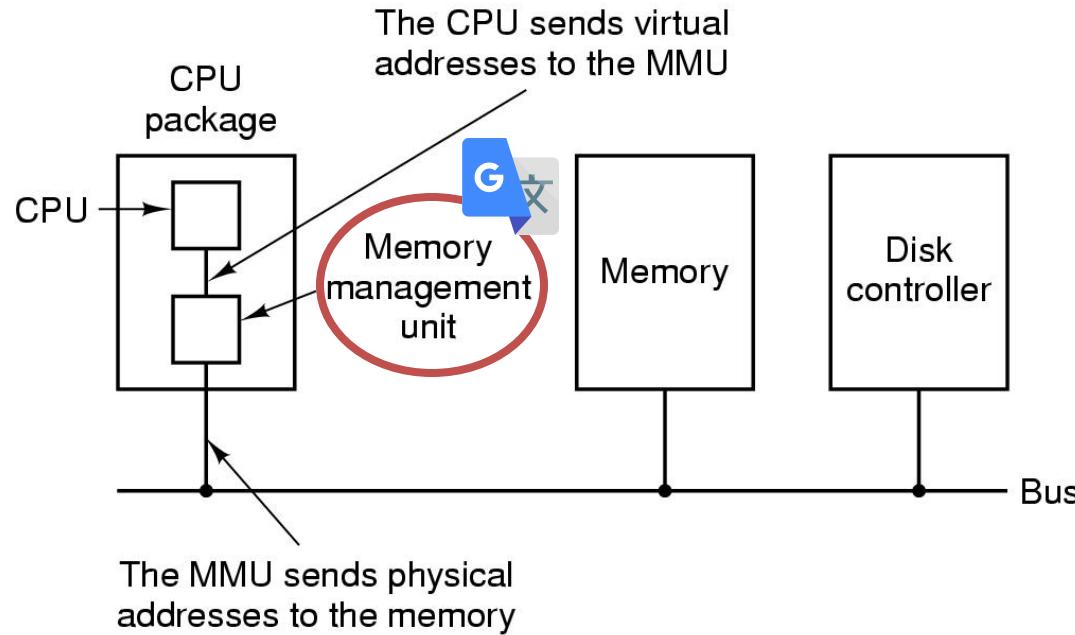
# Outline

---

- Memory management overview
  - Why we need memory management?
  - Memory hierarchy?
  - What will memory management do?
  - What will happen when memory is not enough?
  - How to organize the memory?
- Memory management
  - Translation look-aside buffer
  - Page table
  - Multi-level page table



# Physical and virtual address translation

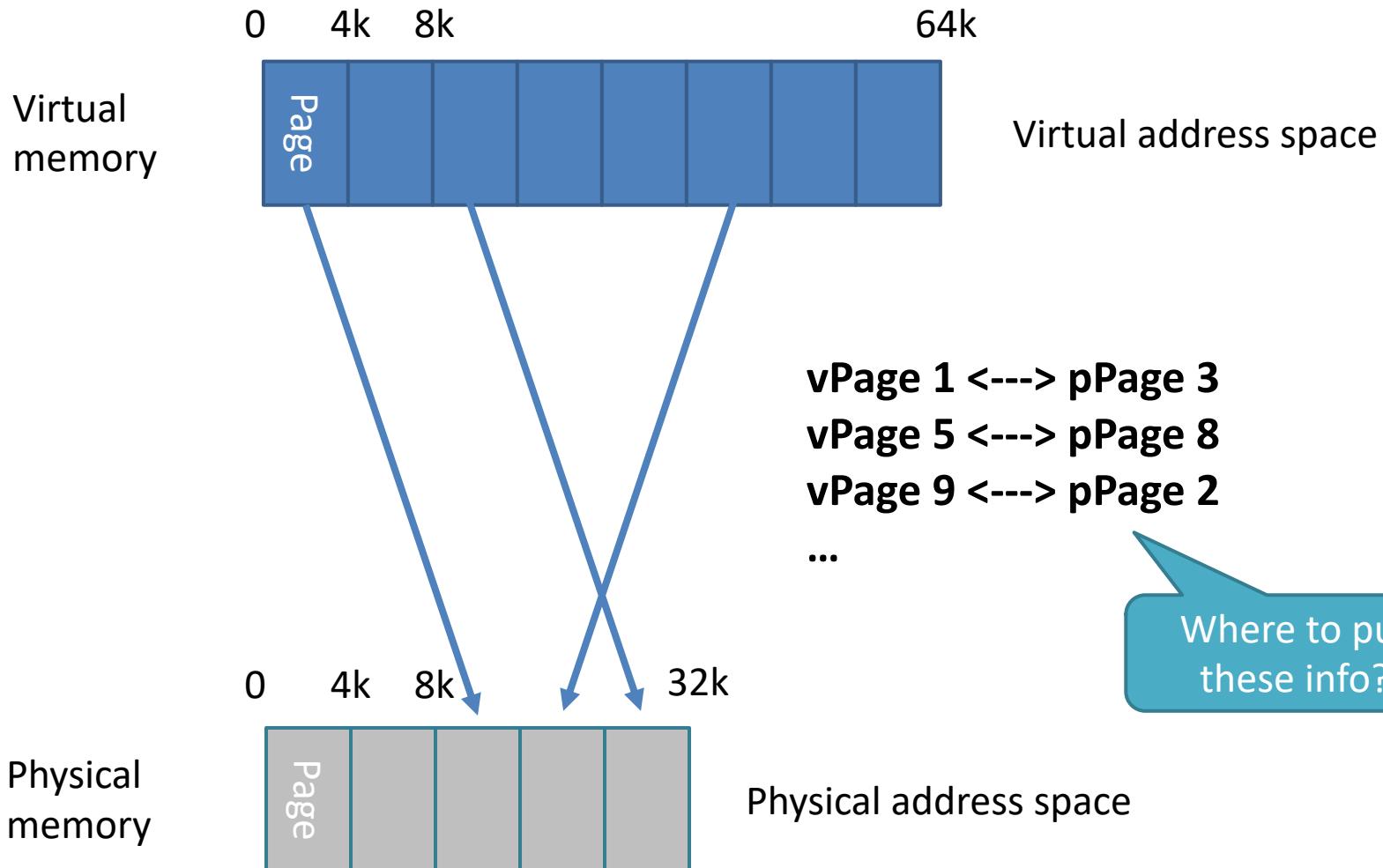


Logical program works in its contiguous virtual address space

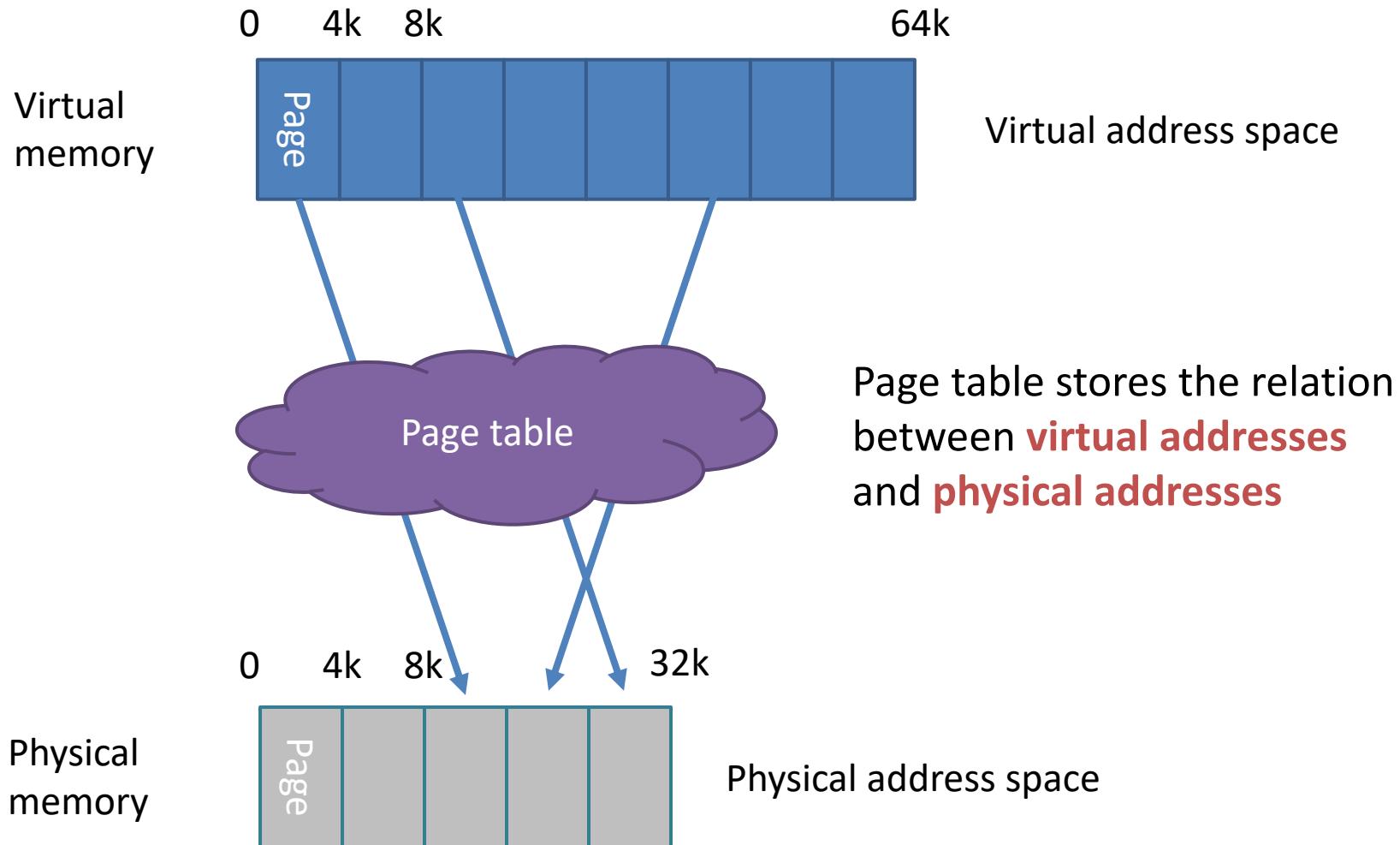
Address translation  
done by MMU

Actual locations of the data in physical memory

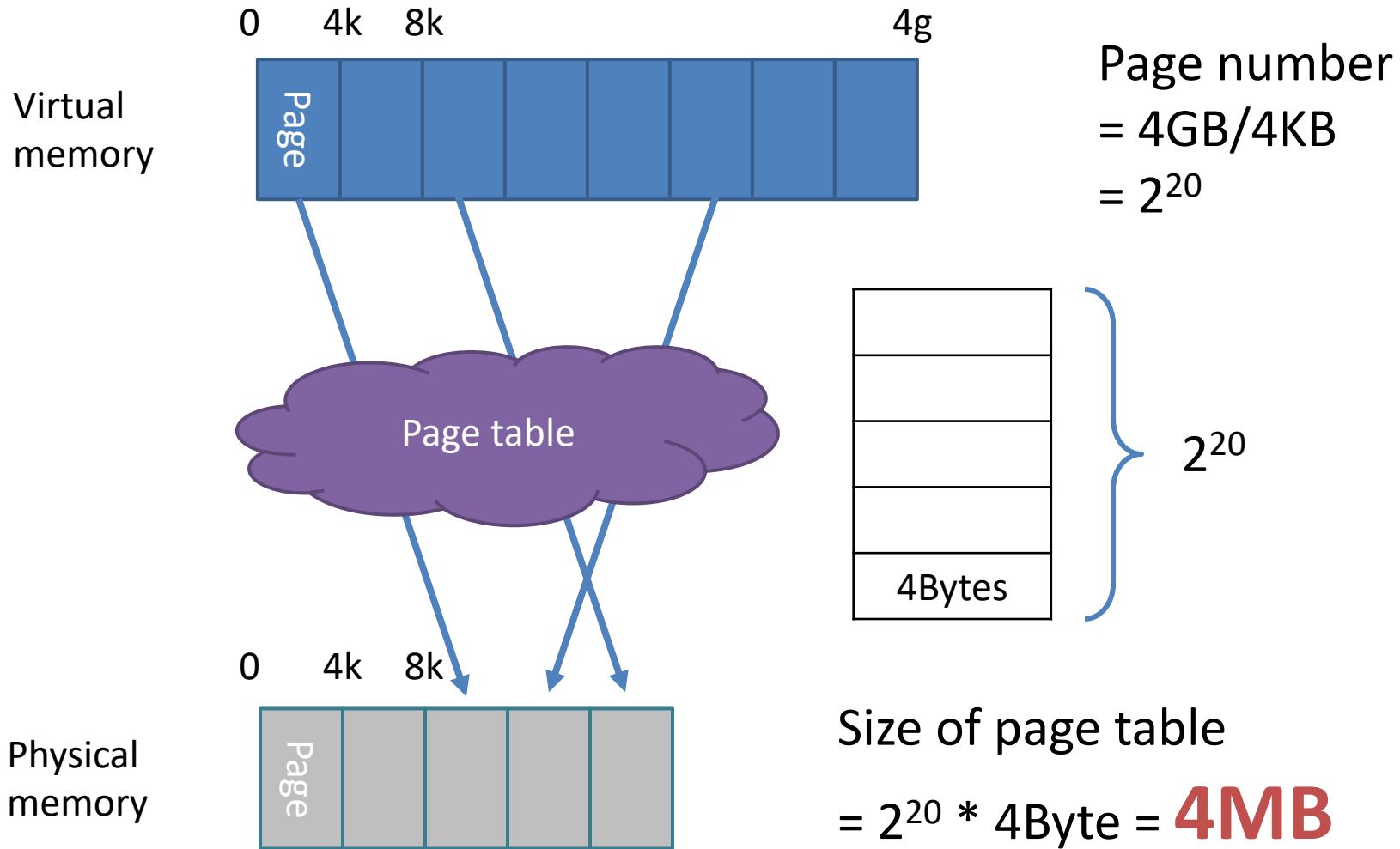
# Page and page table



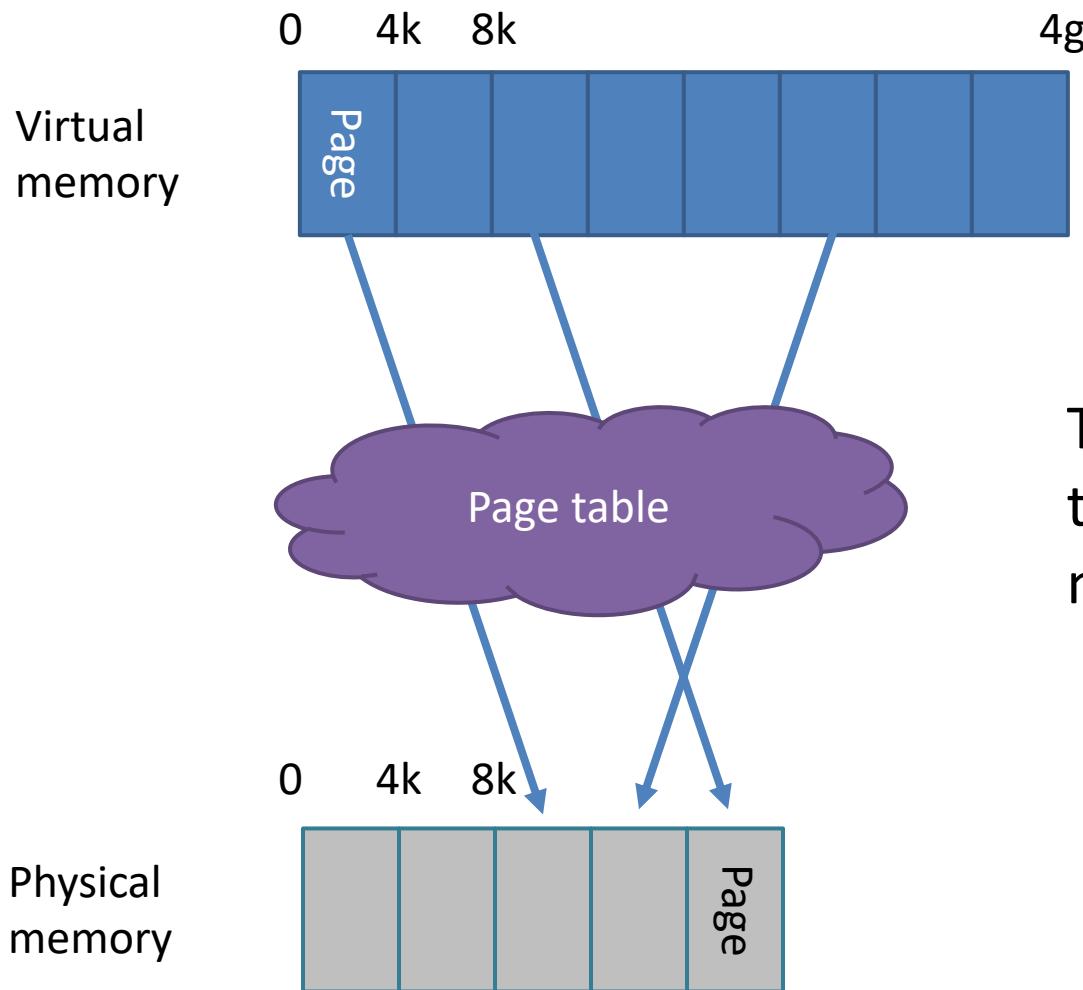
# Page and page table



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?

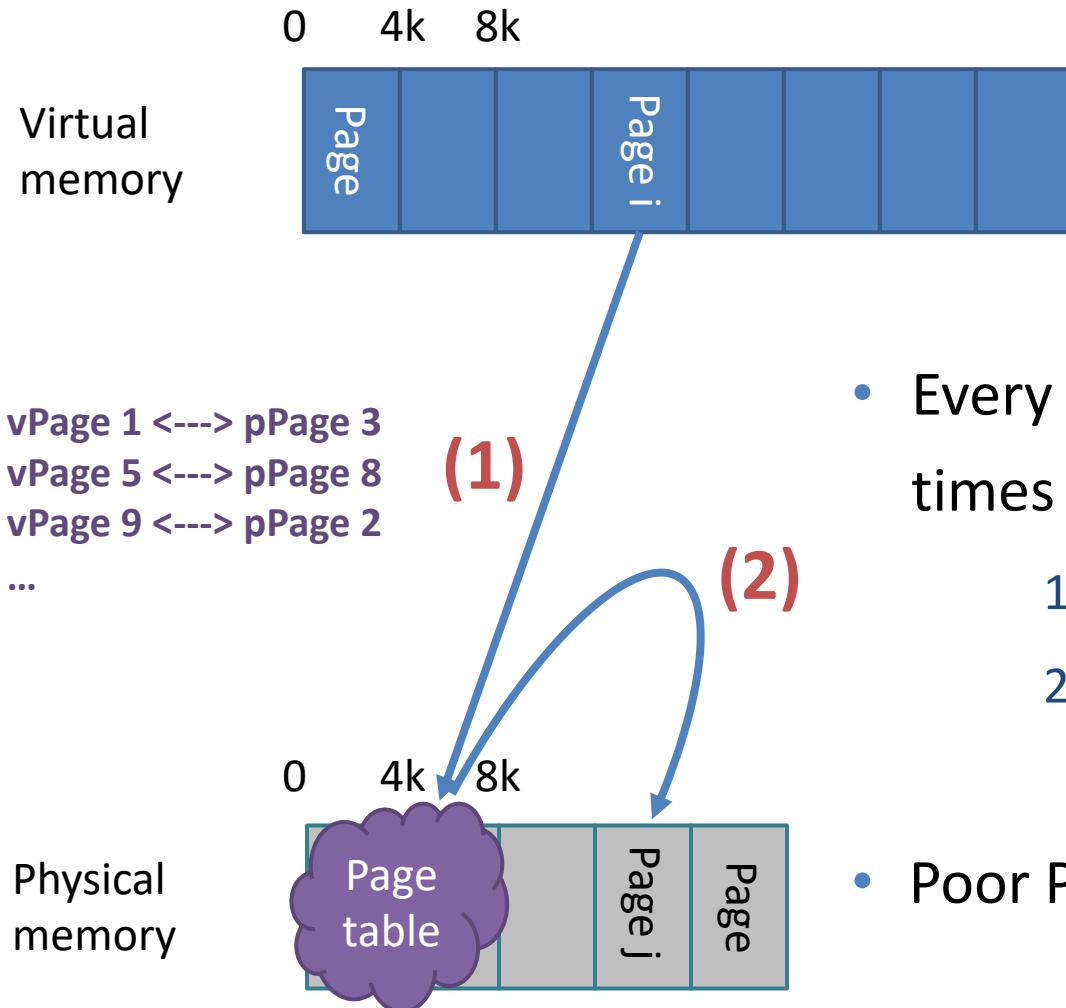


# Where does the page table exist?



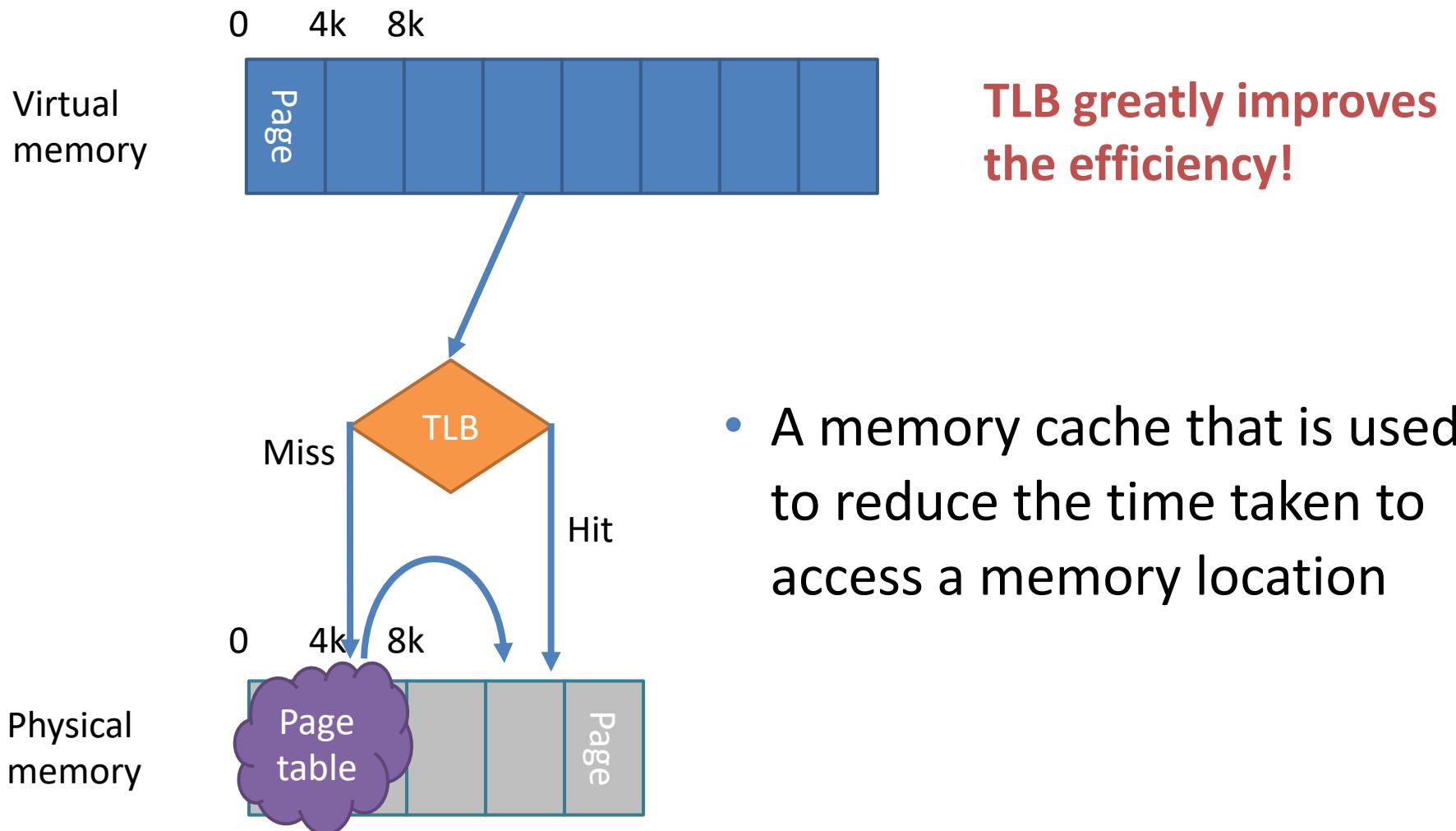
The table is located inside the **physical memory** and maintained by the **OS**

# Page table lookup

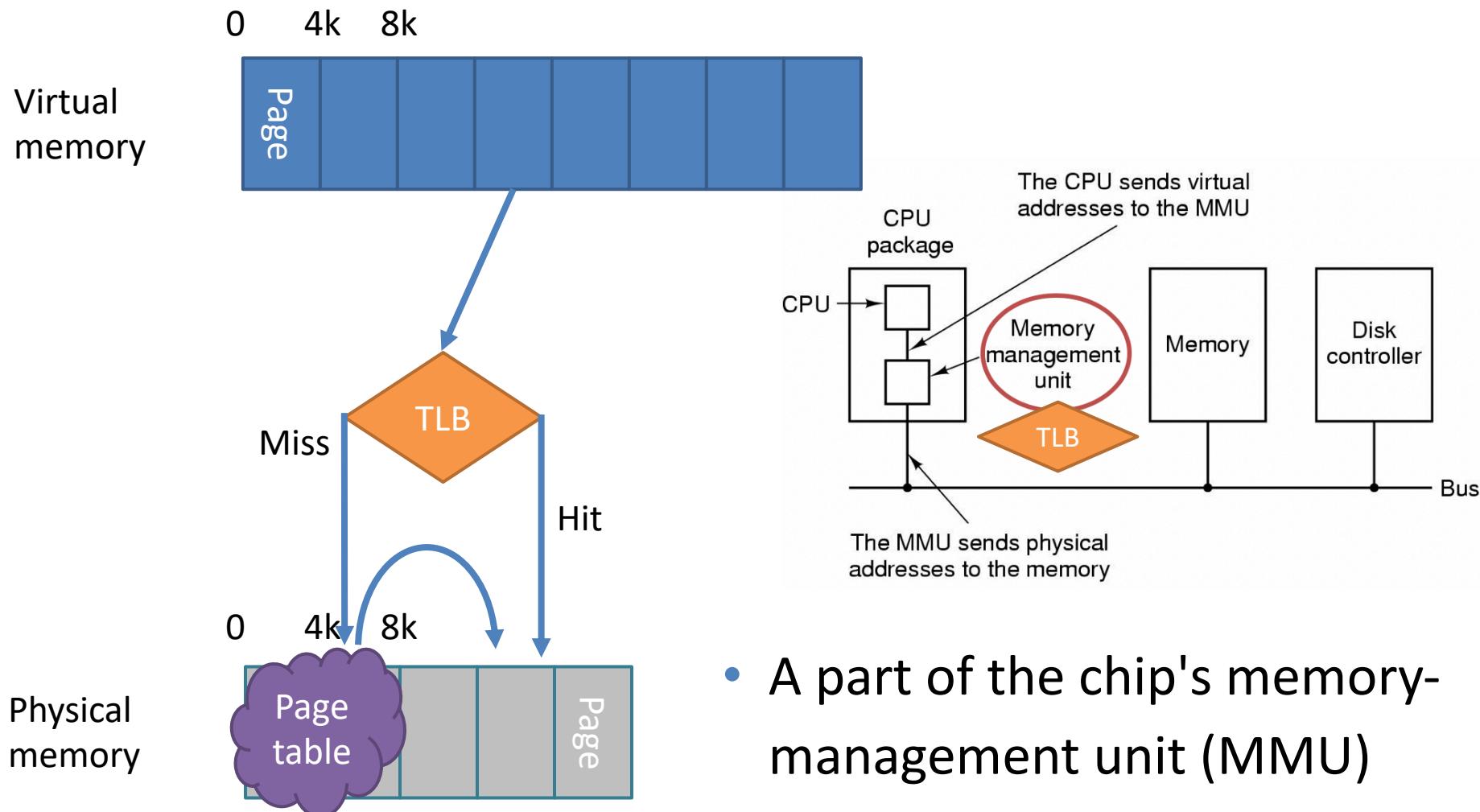


- Every visit to memory requires two times of physical memory access:
  - 1, access to page table
  - 2, visit the destination address
- Poor Performance

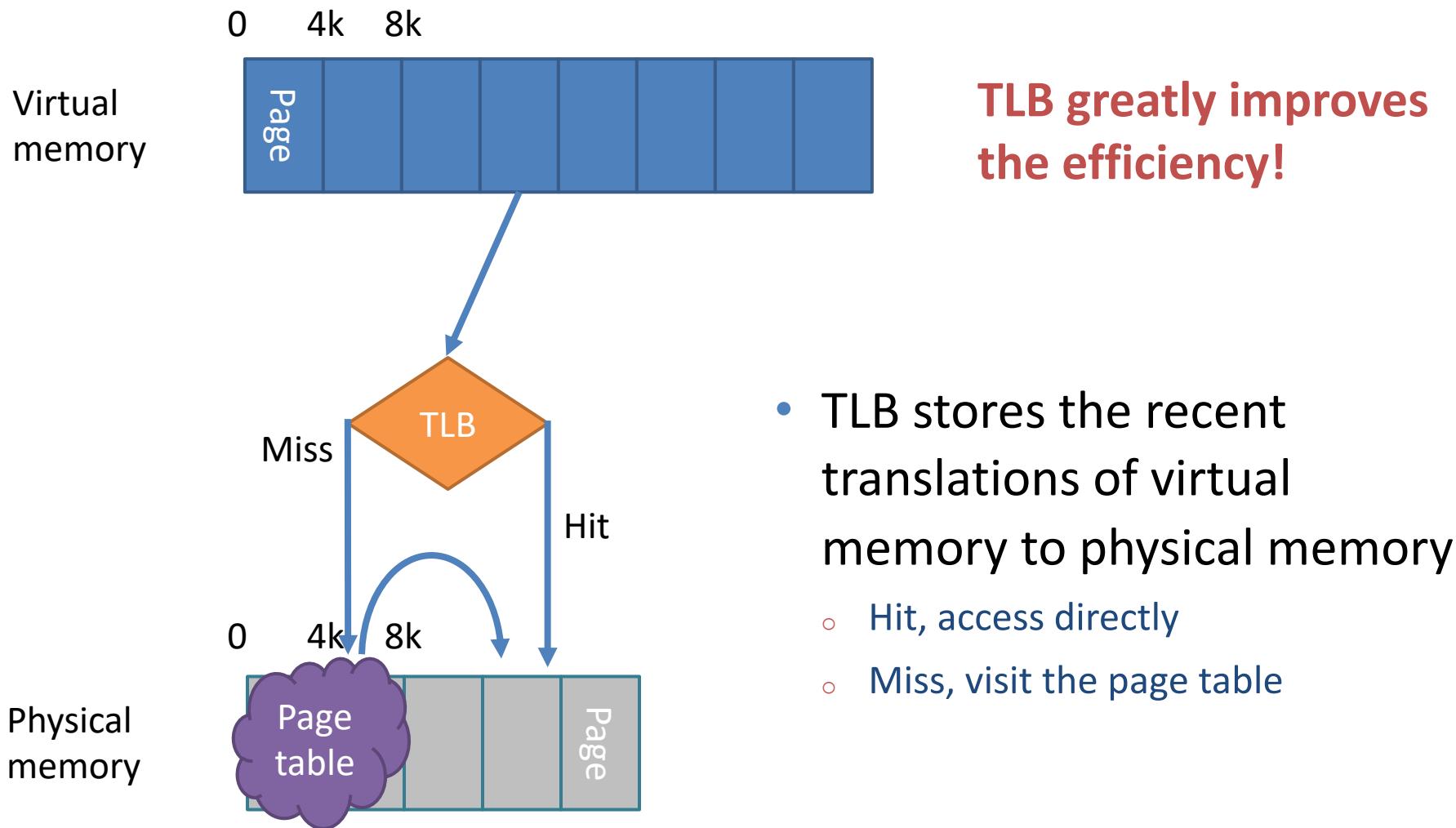
# Translation Look-aside Buffers (TLB)



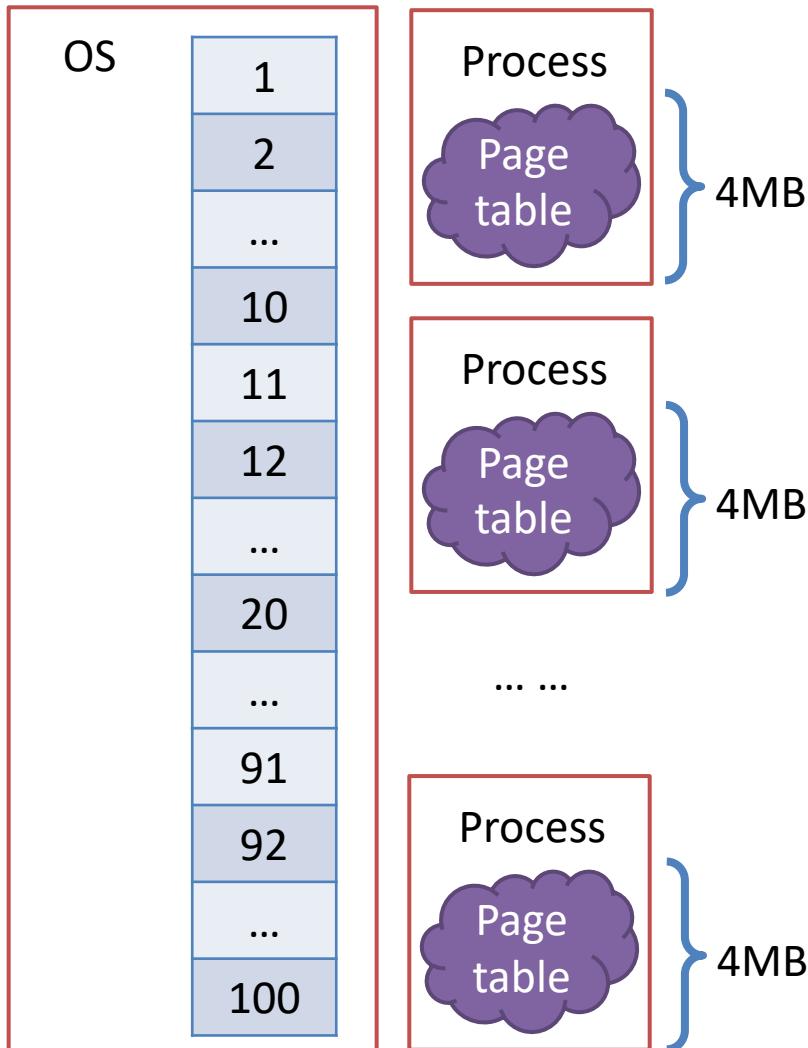
# Translation Look-aside Buffers (TLB)



# Translation Look-aside Buffers (TLB)

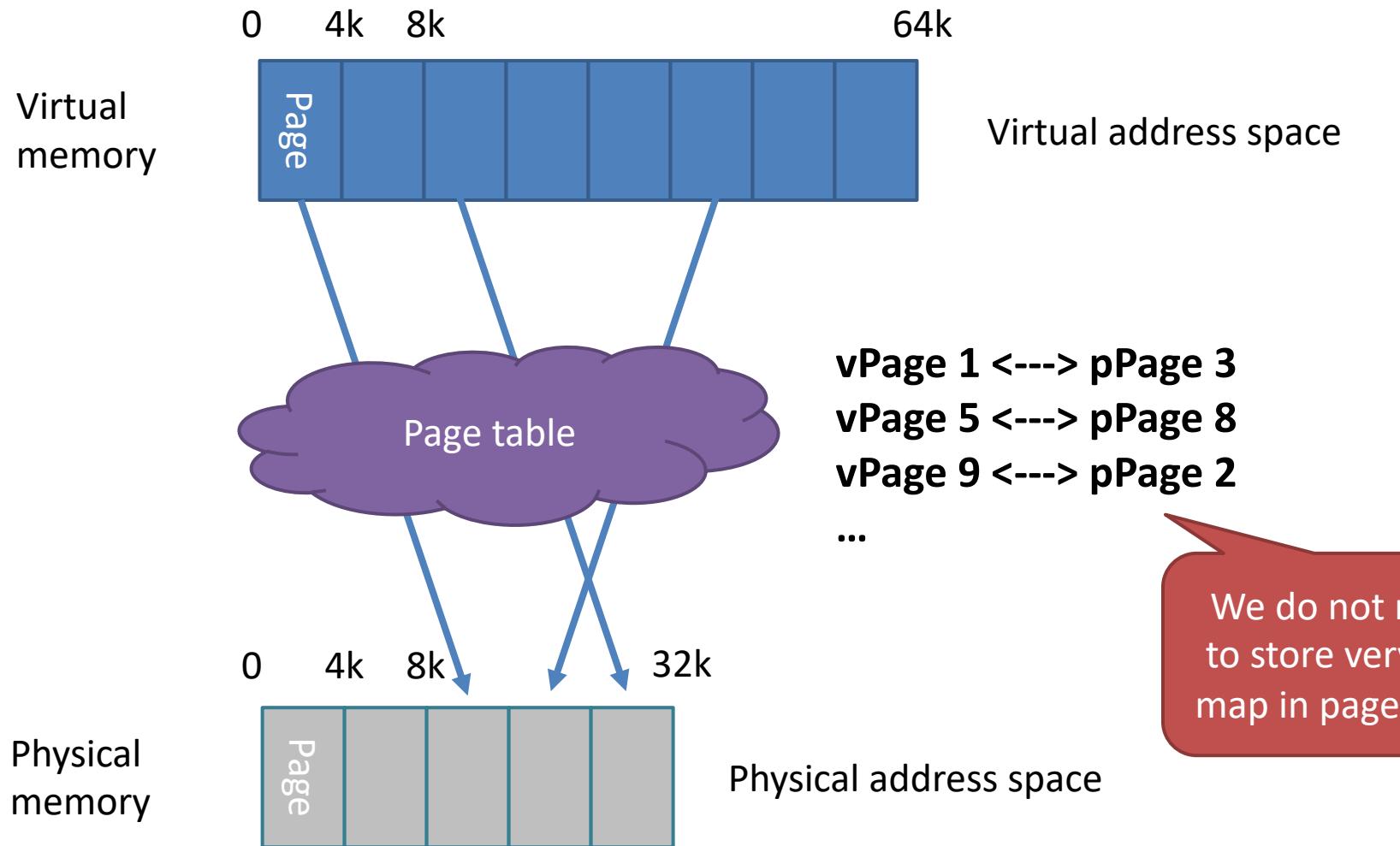


# Page table problems



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

# Page table size optimization

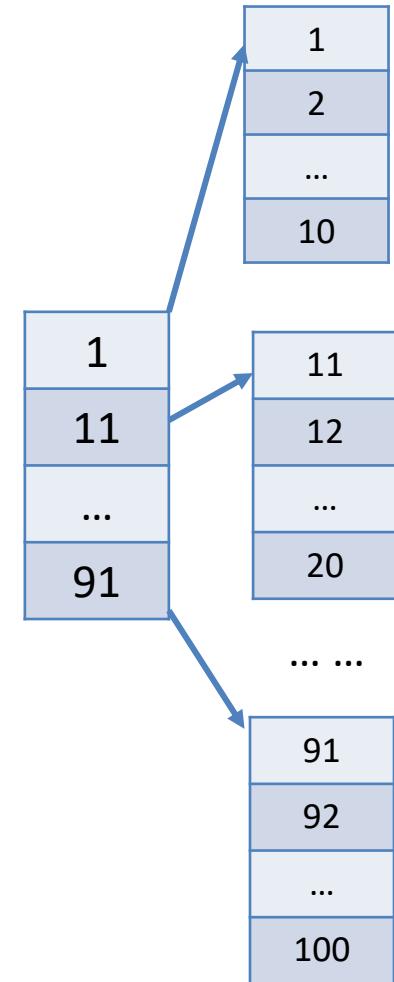


# Multi-level Page Tables

Single-level  
page tables

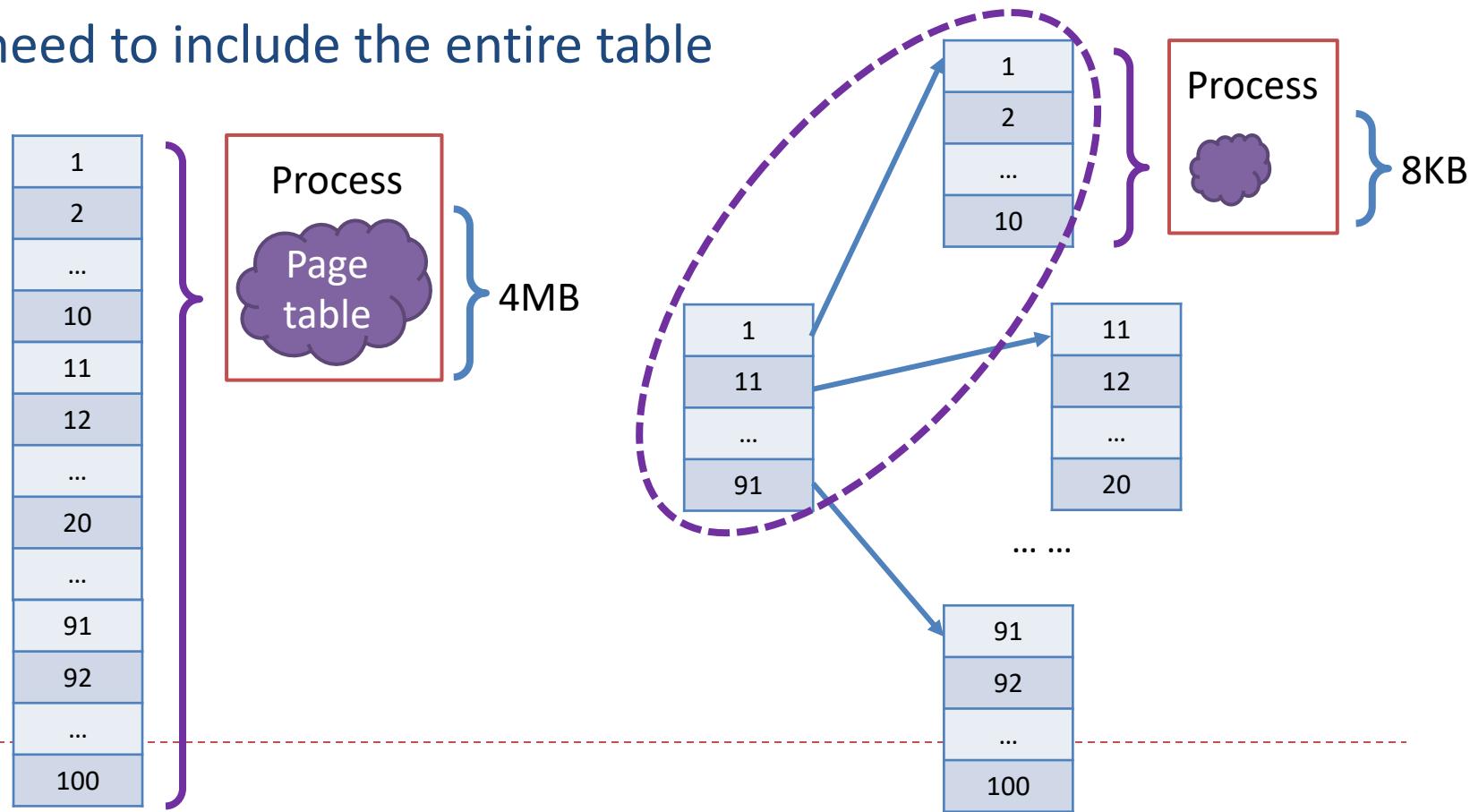
Multi-level  
page tables

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



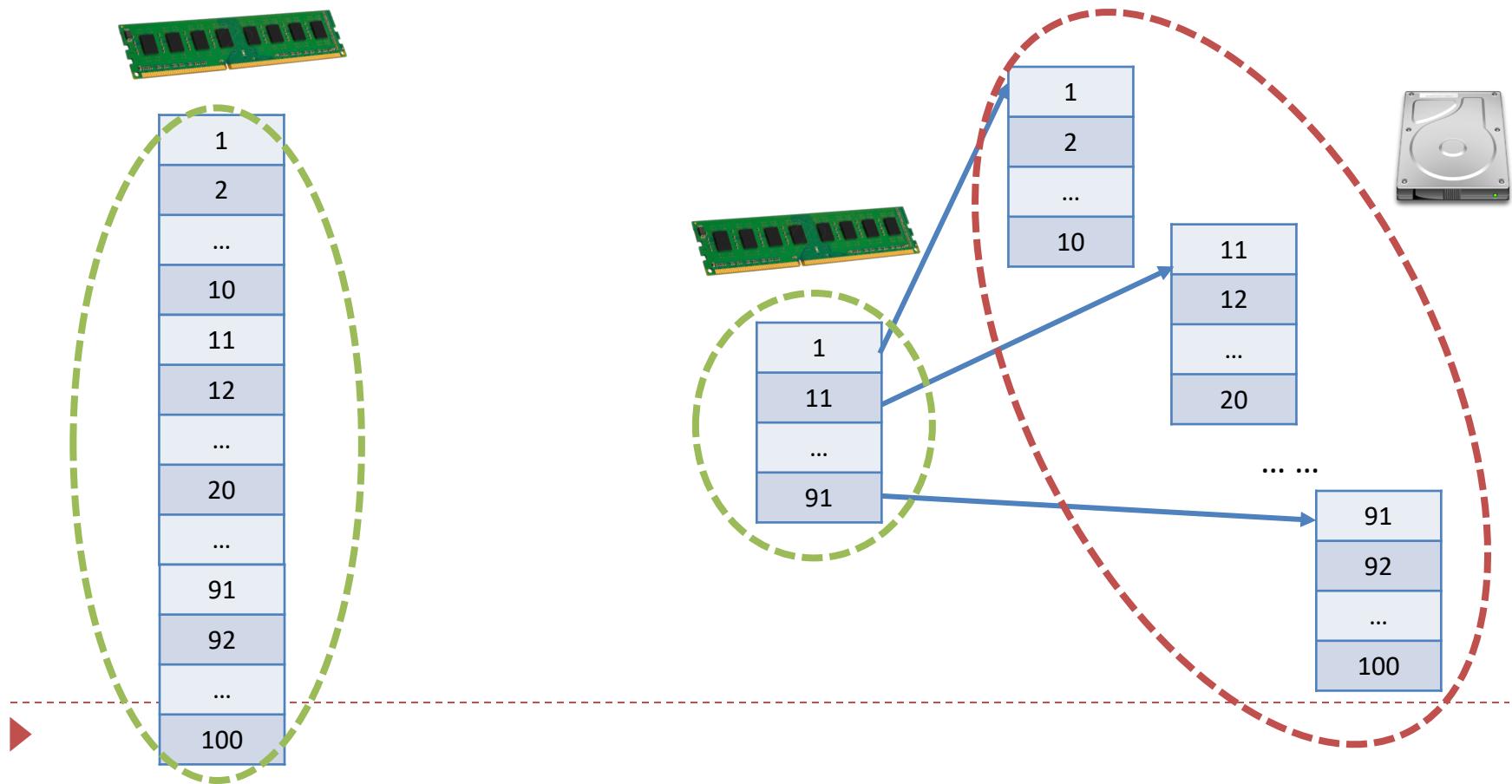
# Multi-level Page Tables benefits

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



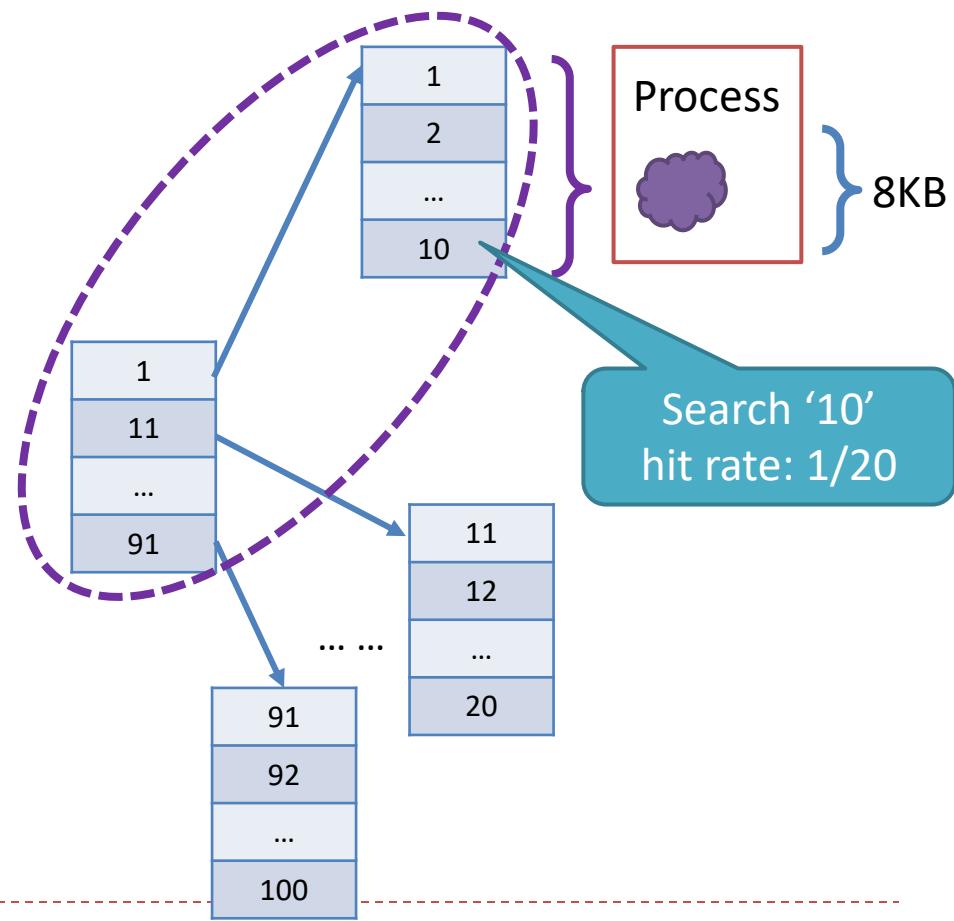
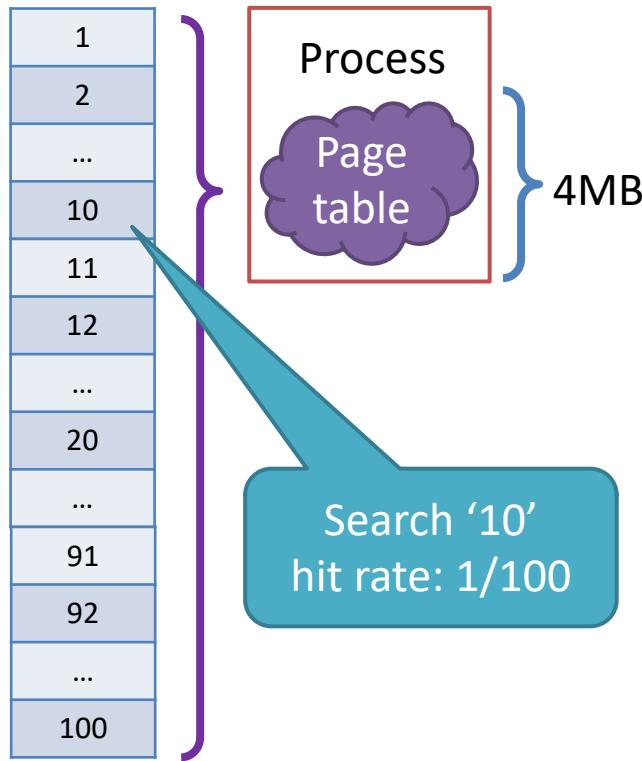
# Multi-level Page Tables benefits

- 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



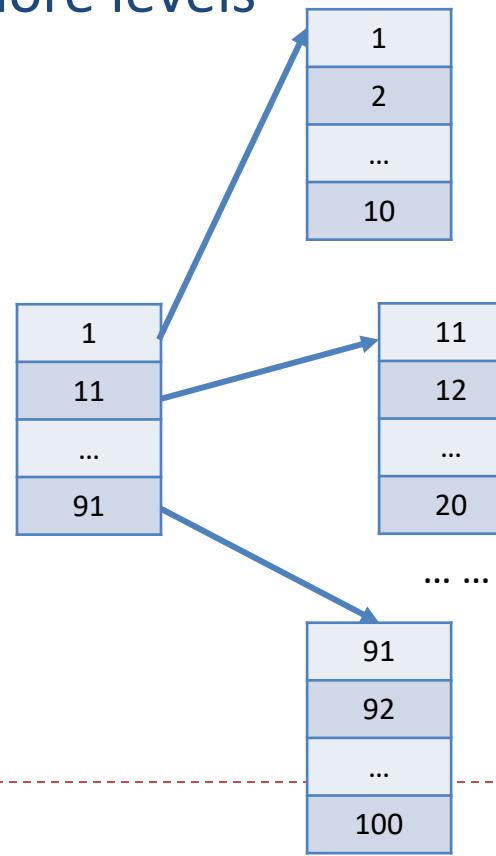
# Multi-level Page Tables drawbacks

- Drawbacks

- Index space
- Higher complexity with more levels

1
2
...
10
11
12
...
20
...
91
92
...
100

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

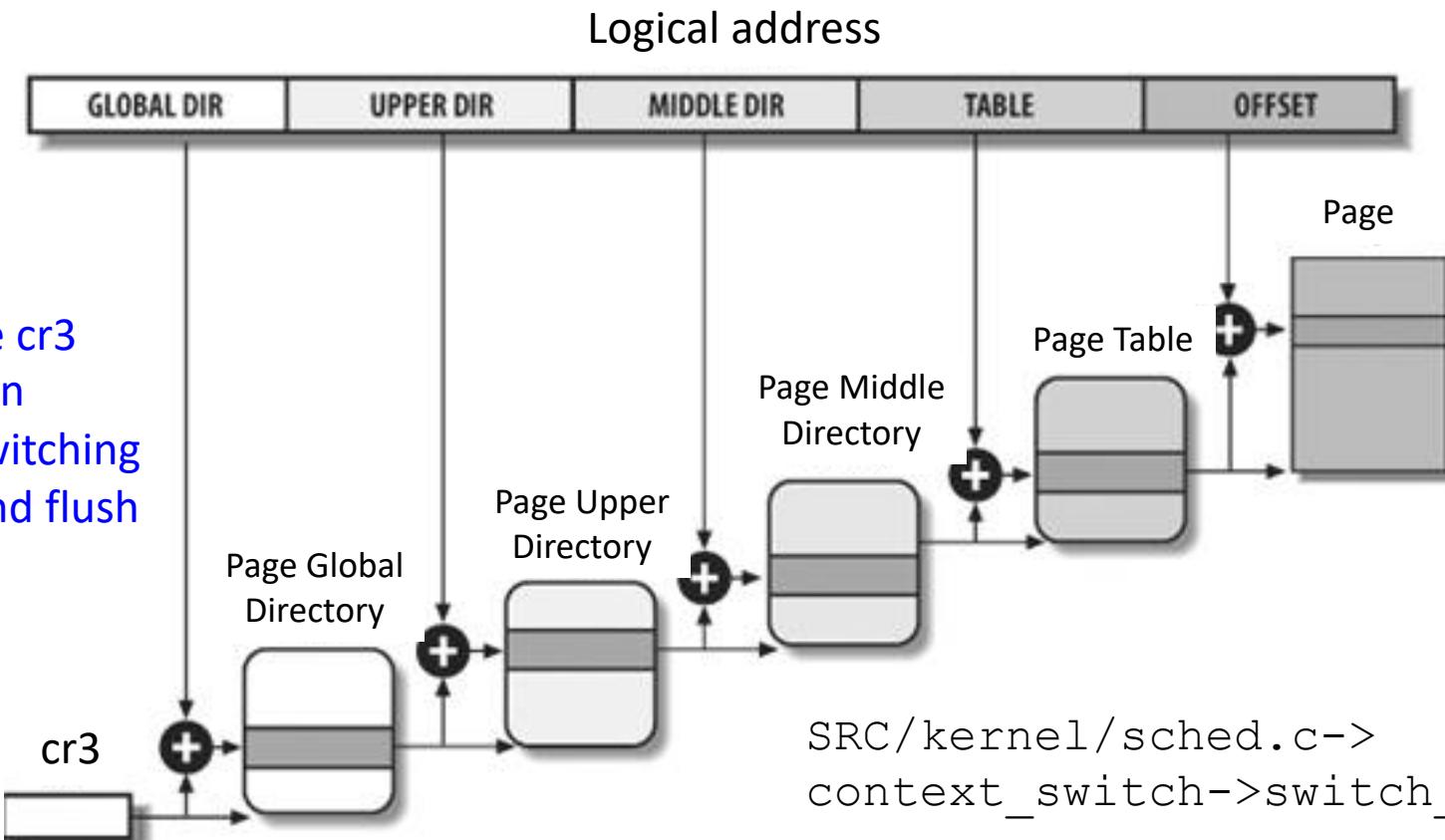


# Multi-level Page Tables in Linux

[https://elixir.bootlin.com/linux/latest/ident/pgdval\\_t](https://elixir.bootlin.com/linux/latest/ident/pgdval_t)

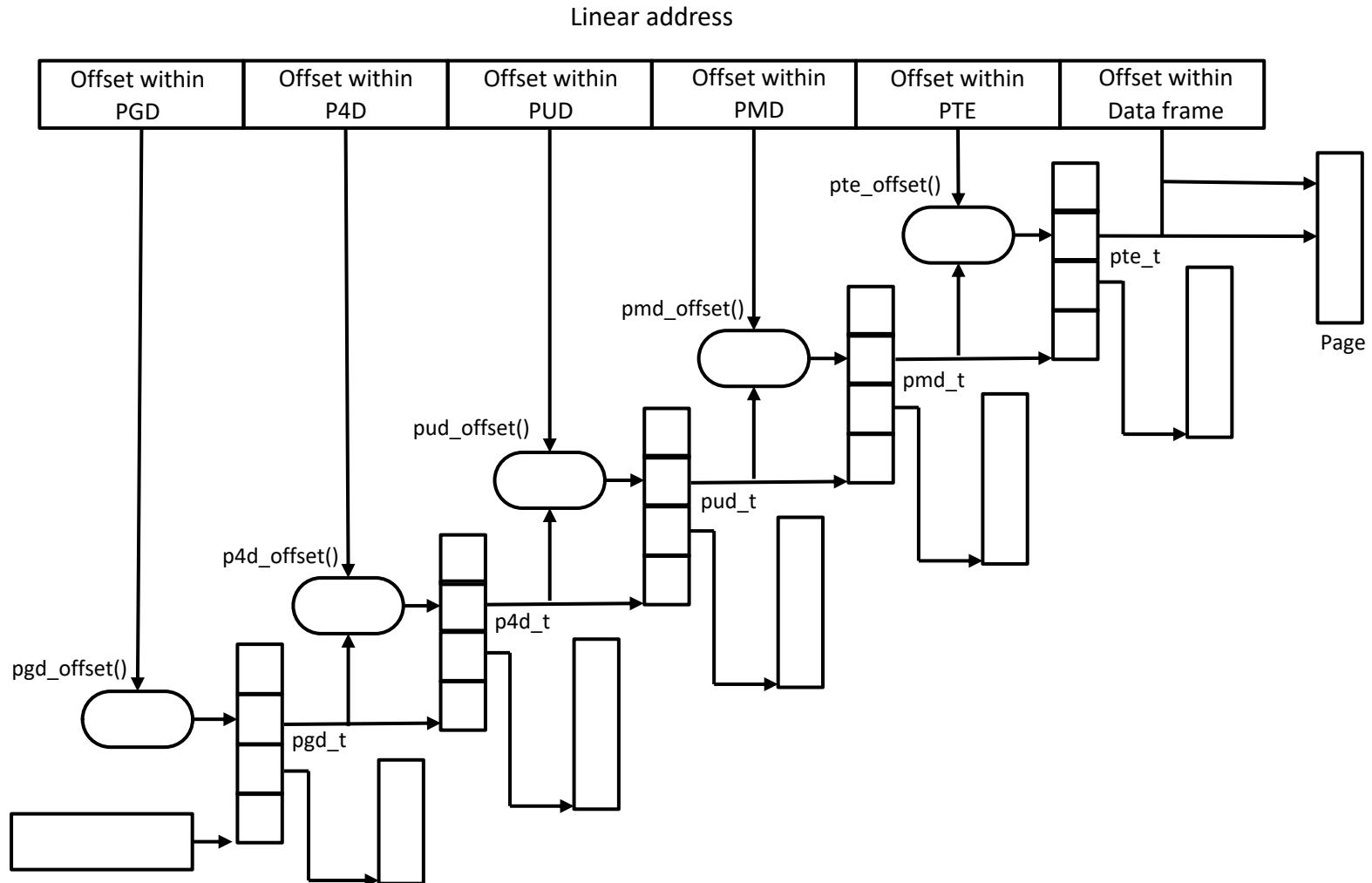
A common model for 32-bit (two-level, 4B pte) and 64-bit (four-level, 8B pte)

SRC/include/linux/sched.h->task\_struct->mm->pgd



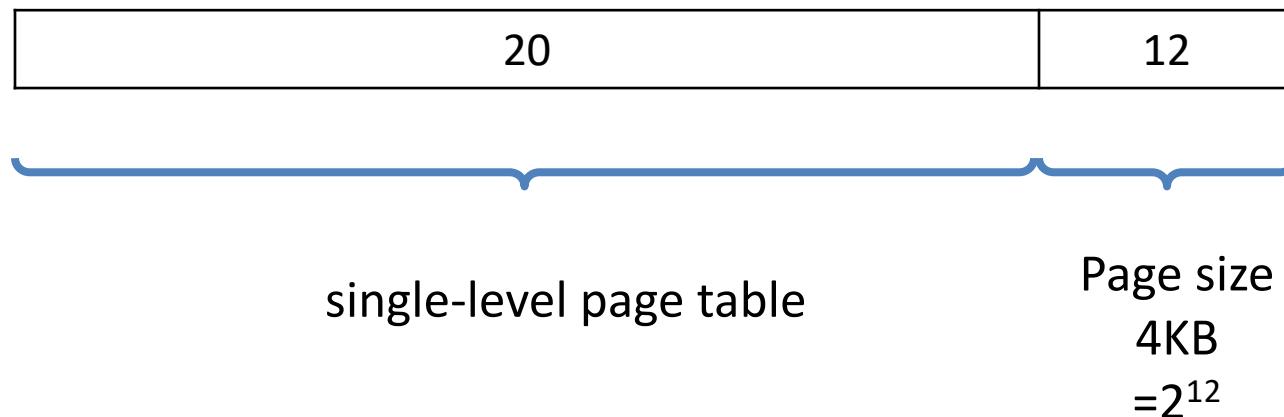
# Multi-level Page Tables in Linux

[https://elixir.bootlin.com/linux/latest/ident/pgdval\\_t](https://elixir.bootlin.com/linux/latest/ident/pgdval_t)



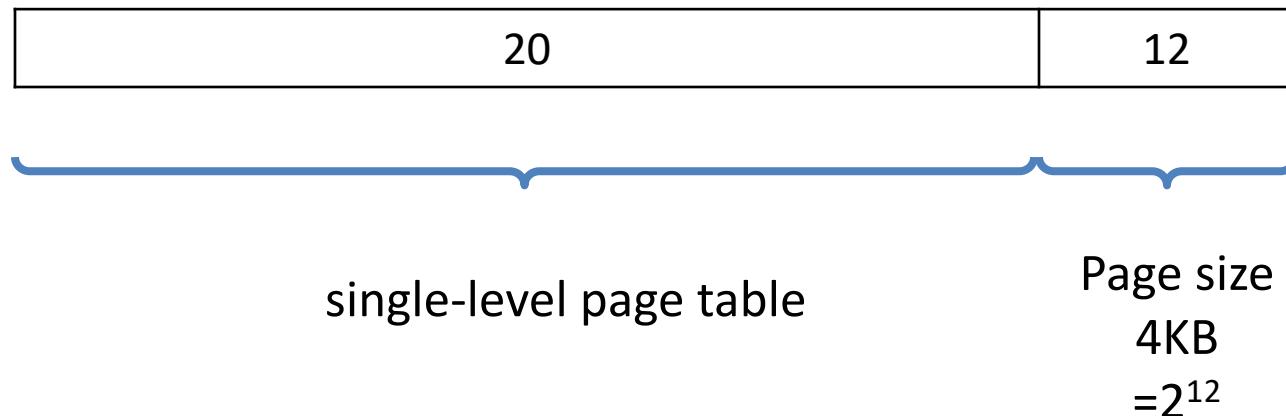
# Multi-level Page Tables Example

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



# Multi-level Page Tables Example

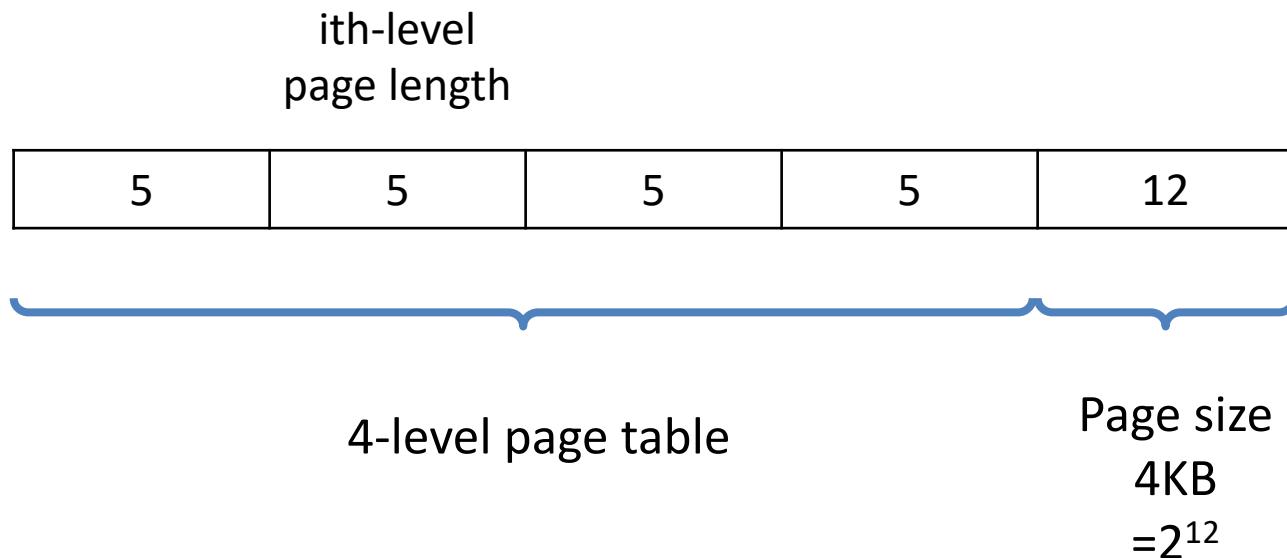
- In a 32-bit system, a single-level page table with 4KB pages. Suppose PT1=1, offset=10, what is its virtual address?



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



# 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

