Virtual memory

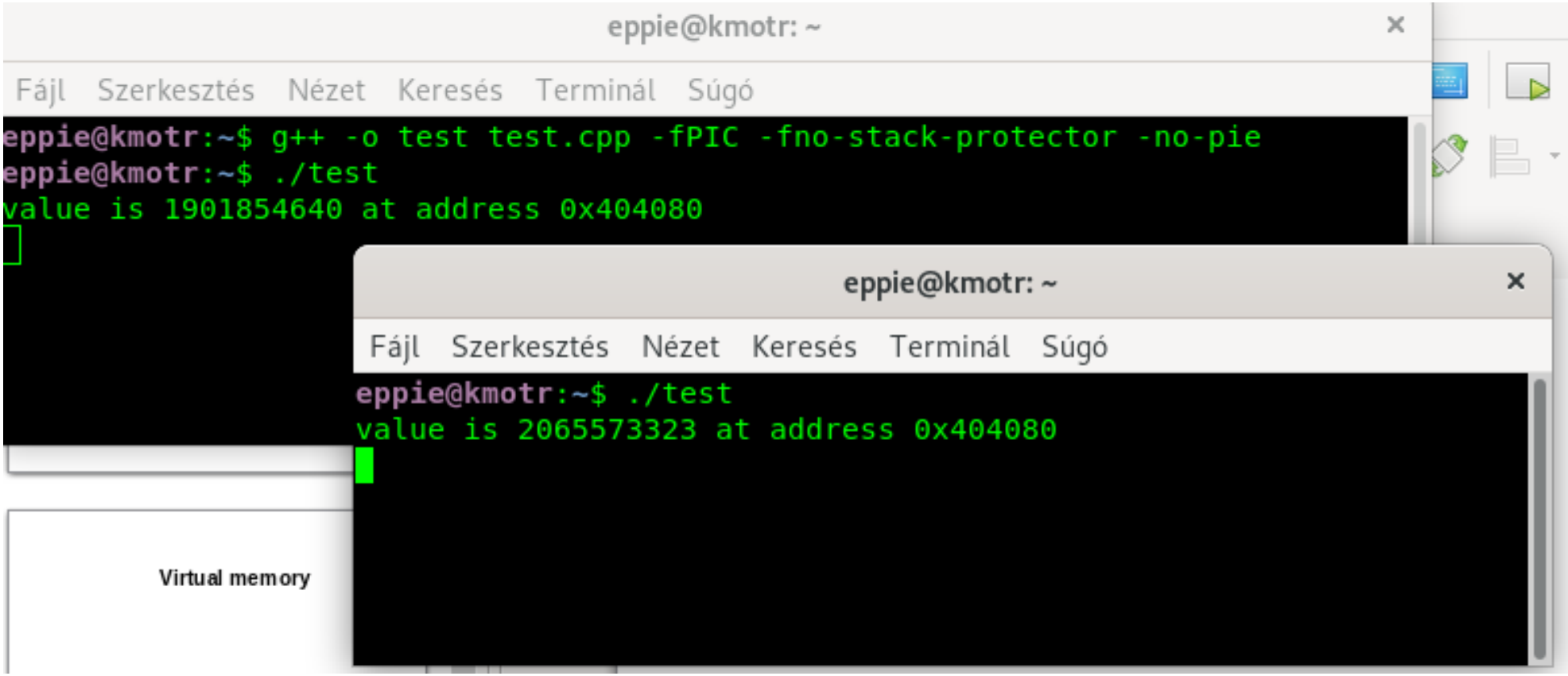
Questions

- What if you wanted to run a program that needs more memory than you have?
- What if you want to run 10 programs, which collectively need more memory than you have?
- What if multiple different programs want to store something at the same address?
- How do we protect one program's data from being read or written by another program, either intentionally or accidentally?
- How do we protect the operating system's data from being read or written by another program, either intentionally or accidentally?

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <unistd.h>

// compile with: g++ -o test test.cpp -fPIC -fno-stack-protector -no-pie

using namespace std;
int main() {
    static int value;
    srand(time(NULL));
    value = rand();
    while(1) {
        cout << "value is " << value << " at address "
                             << &value << endl;
        sleep(2);
    }
    return 0;
}
```


eppie@kmotr: ~
Fájl Szerkesztés Nézet Keresés Terminál Súgó
eppie@kmotr:~$ g++ -o test test.cpp -fPIC -fno-stack-protector -no-pie
eppie@kmotr:~$ ./test
value is 1901854640 at address 0x404080
eppie@kmotr: ~
Fájl Szerkesztés Nézet Keresés Terminál Súgó
eppie@kmotr:~$ ./test
value is 2065573323 at address 0x404080
Virtual memory

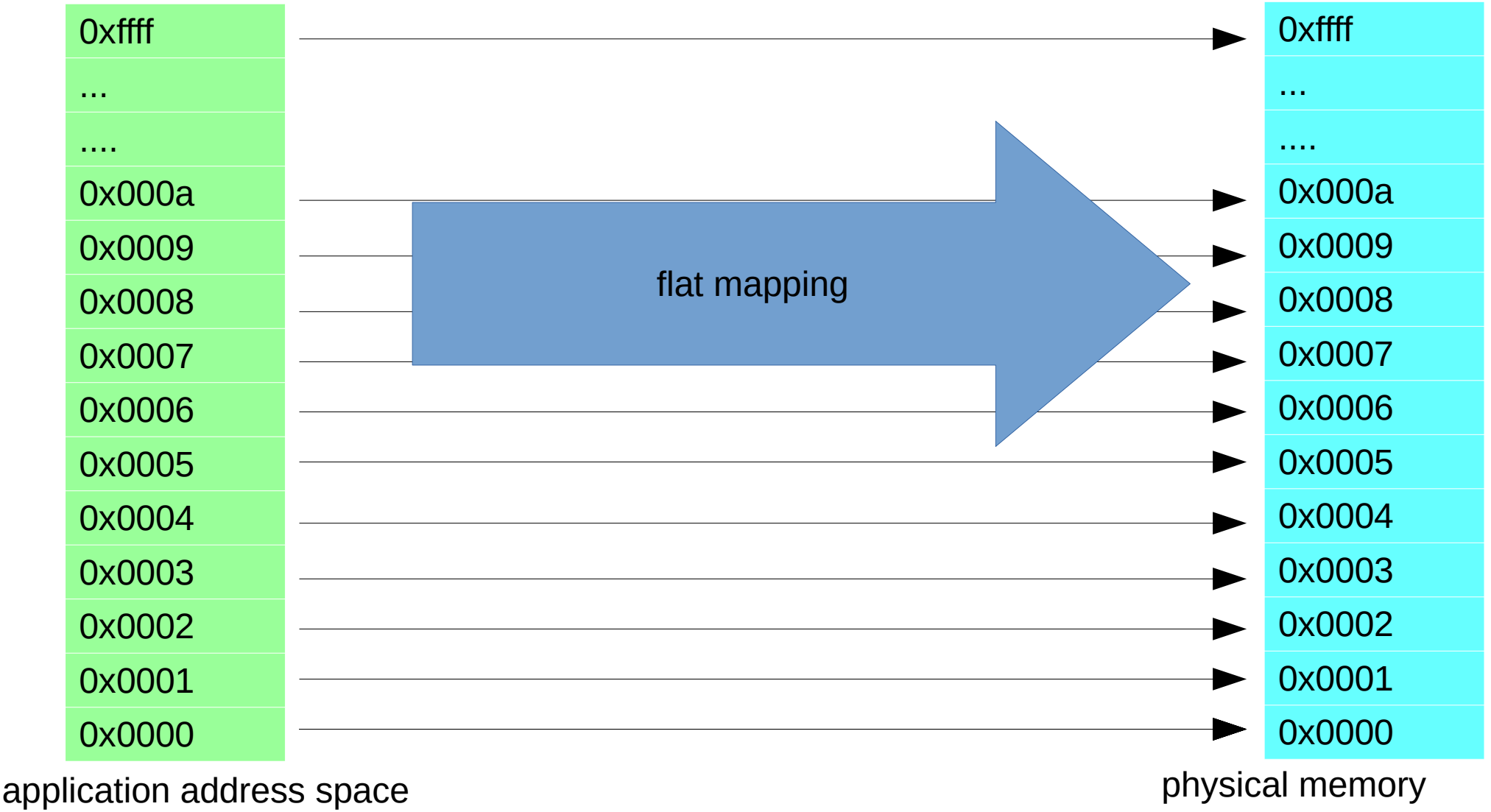
Address space

Address space is the set of memory addresses, and their corresponding memory cells, that are available to a program.

On E20, the address space is expressed as a 16-bit pointer value, so it has 2**16 elements, starting from zero.

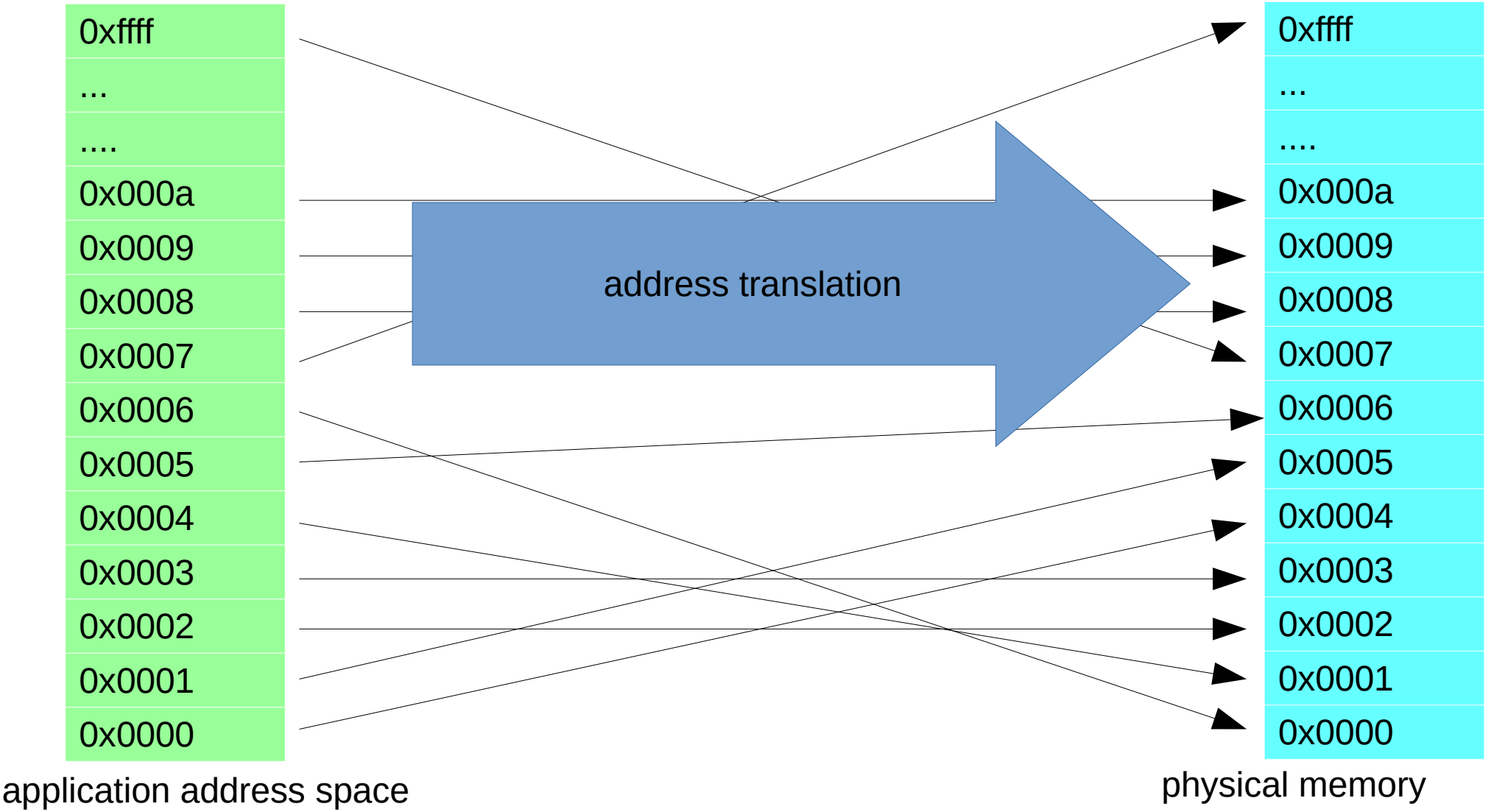
Flat address space mapping

lw $1, foo($0)

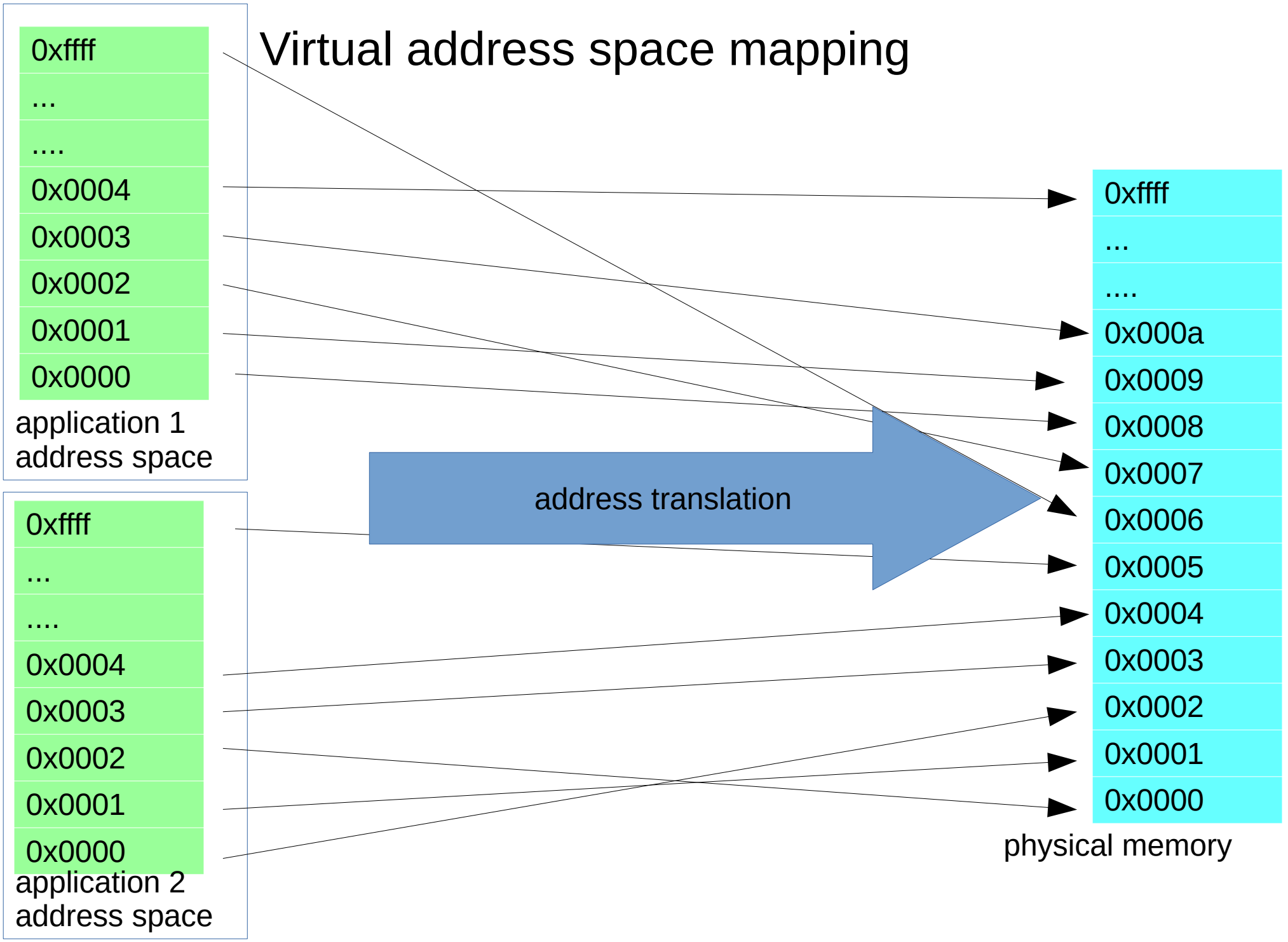


Virtual address space mapping

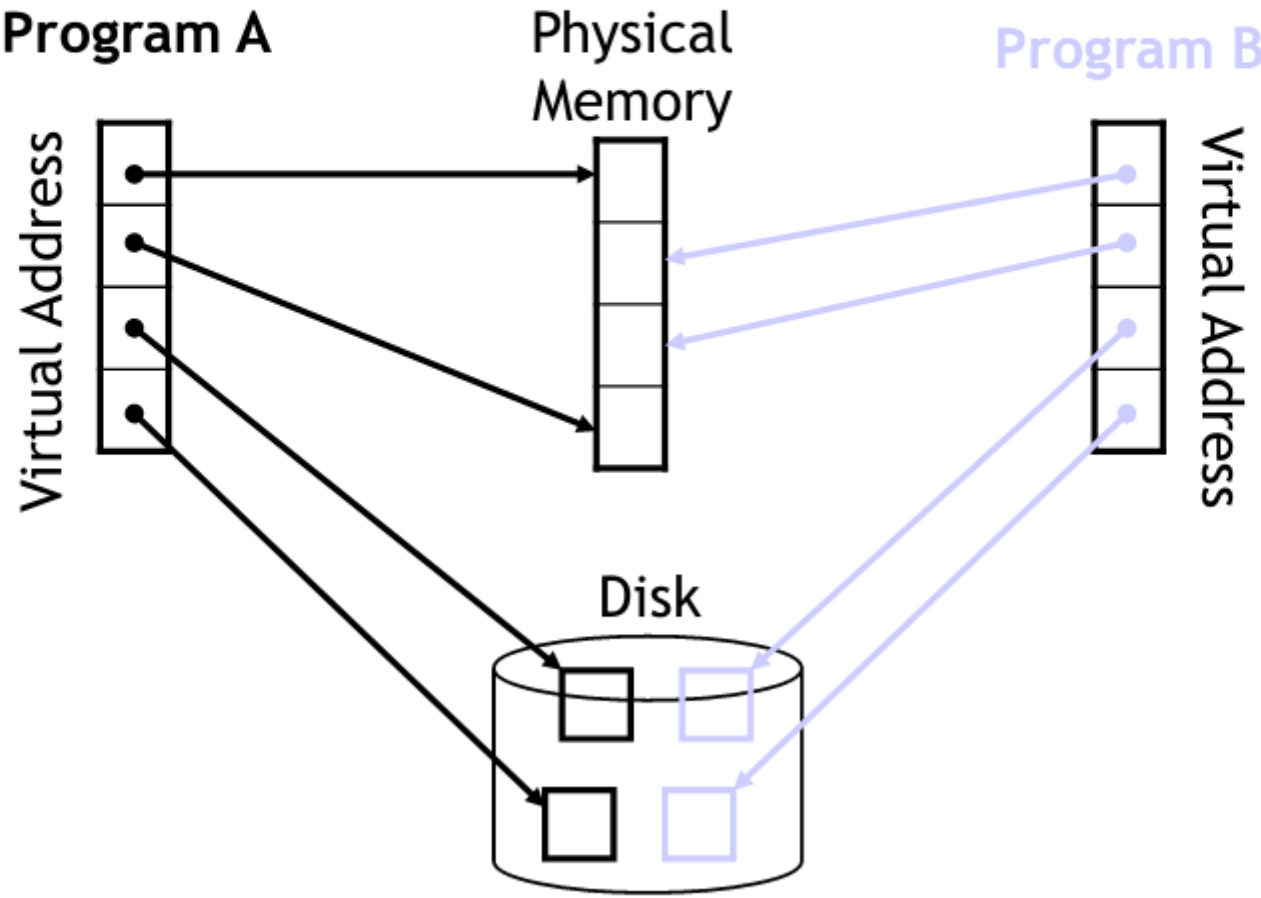
lw $1, foo($0)



Virtual address space mapping

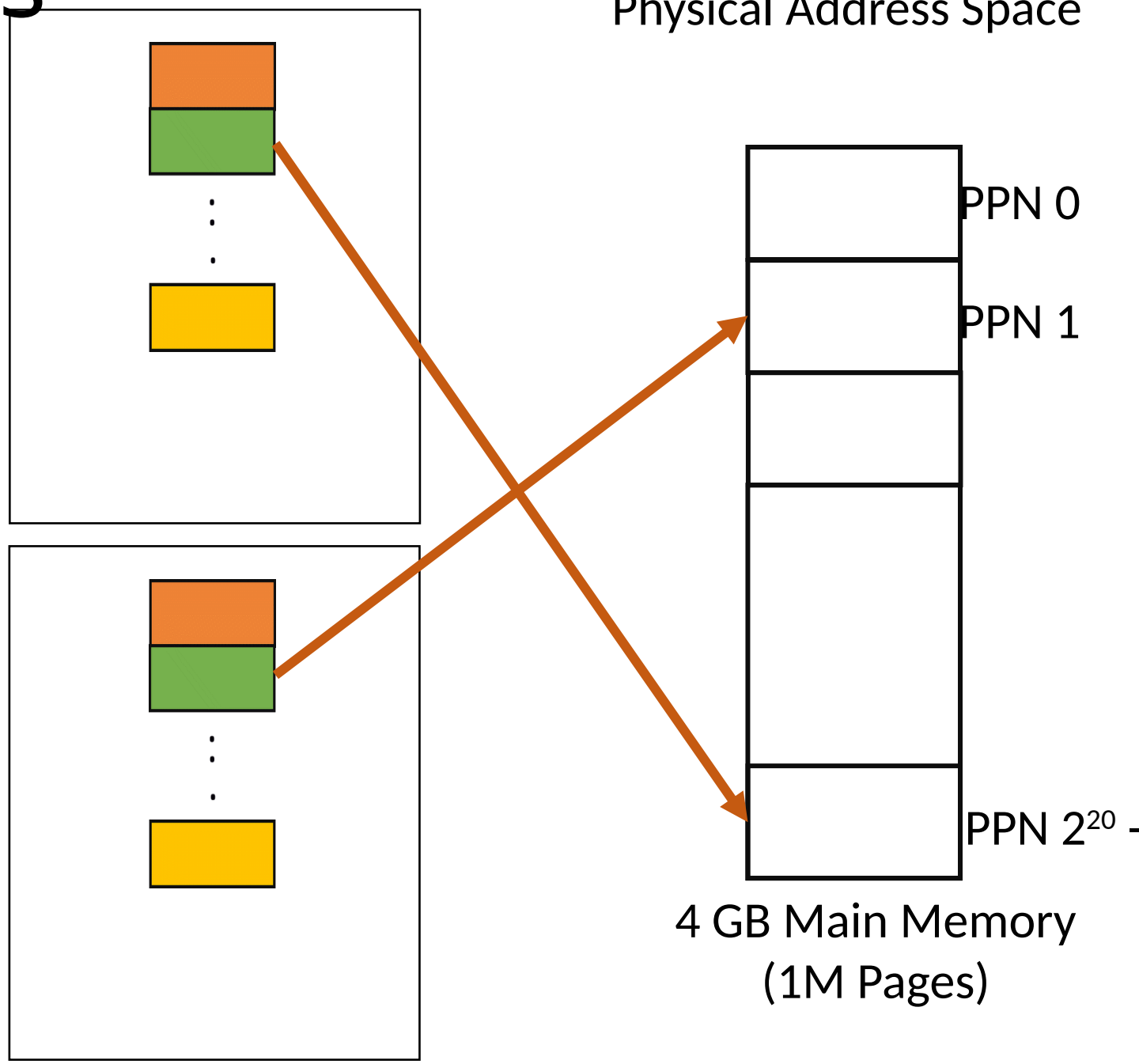


- Because different processes will have different mappings from virtual to physical addresses, two programs can freely use the same virtual address.
- By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each others data.



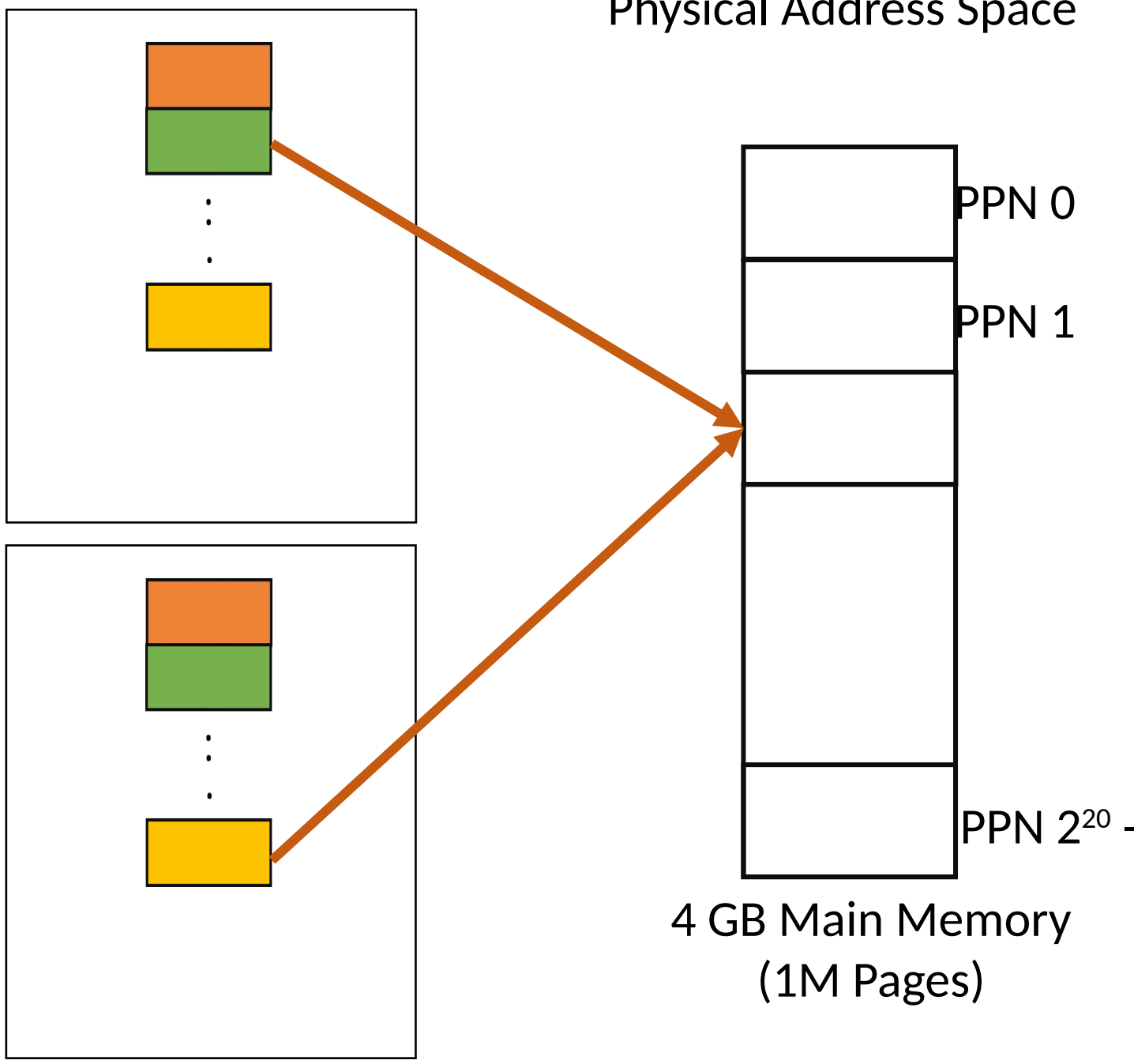
Homonyms

- The same VA (in different processes) can map to two different physical addresses

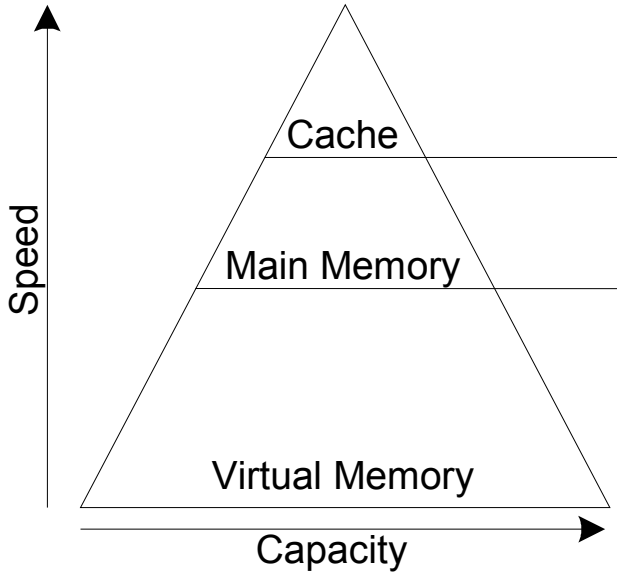


Synonyms

- Different VAs can map to the same physical address
- Shared code and data



Memory Hierarchy

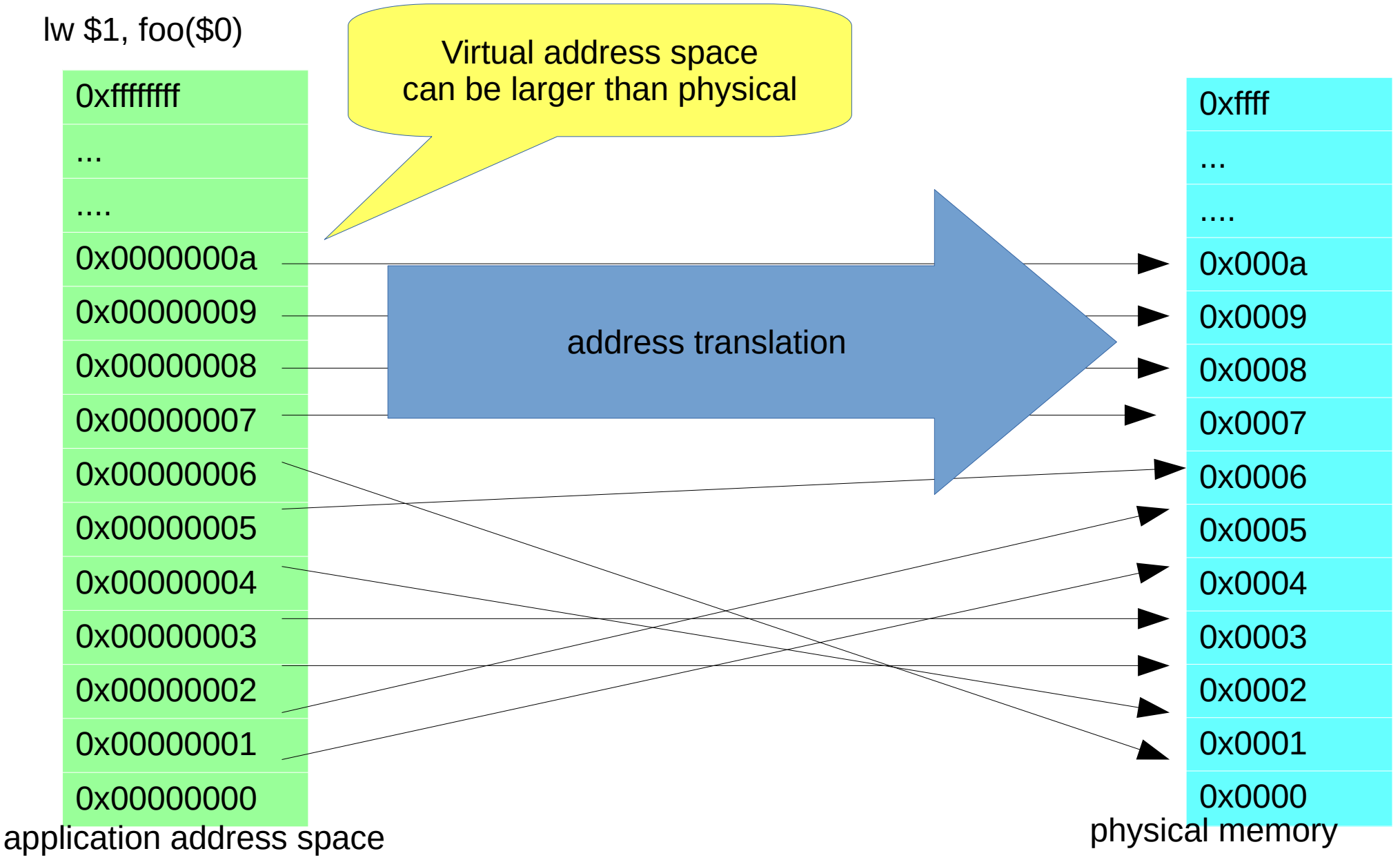


Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

SRAM, DRAM – volatile

SSD, HDD – non-volatile

Virtual address space mapping



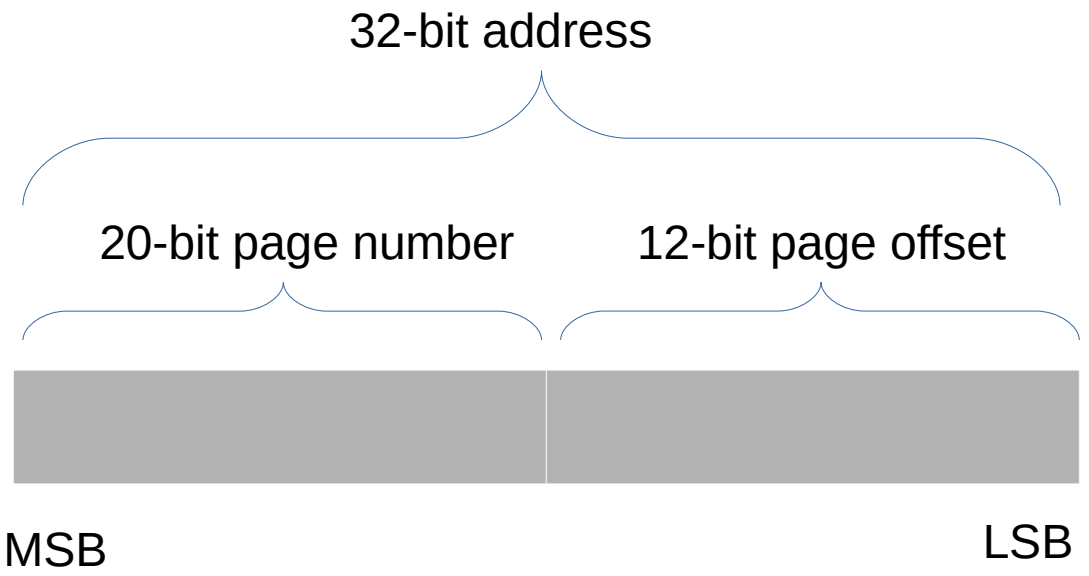
Virtual address	**Physical address**
Visible to applications	Not visible to applications; may be accessible to operating system
Used by `lw` and `sw` (or their equivalents)	
Used for fetching instructions from memory	

For every memory access, the virtual address must be translated to a physical address. This translation is done by special address translation hardware.

The translation happens automatically: the application program has no idea.

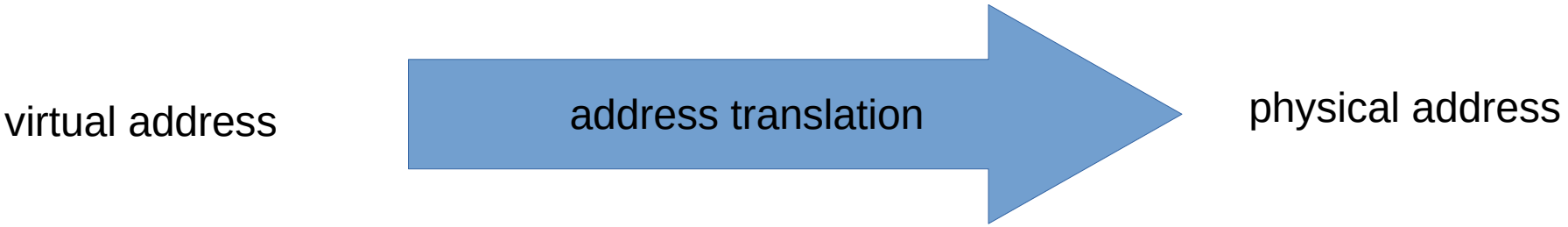
We divide address space (both virtual and physical) into *pages*. A typical size for pages is 4KB (4096, or 0x1000 bytes).

Address range	Page
0x0b000..0x0bfff	11
0x0a000..0x0afff	10
0x09000..0x09fff	9
0x08000..0x08fff	8
0x07000..0x07fff	7
0x06000..0x06fff	6
0x05000..0x05fff	5
0x04000..0x04fff	4
0x03000..0x03fff	3
0x02000..0x02fff	2
0x01000..0x01fff	1
0x00000..0x00fff	0



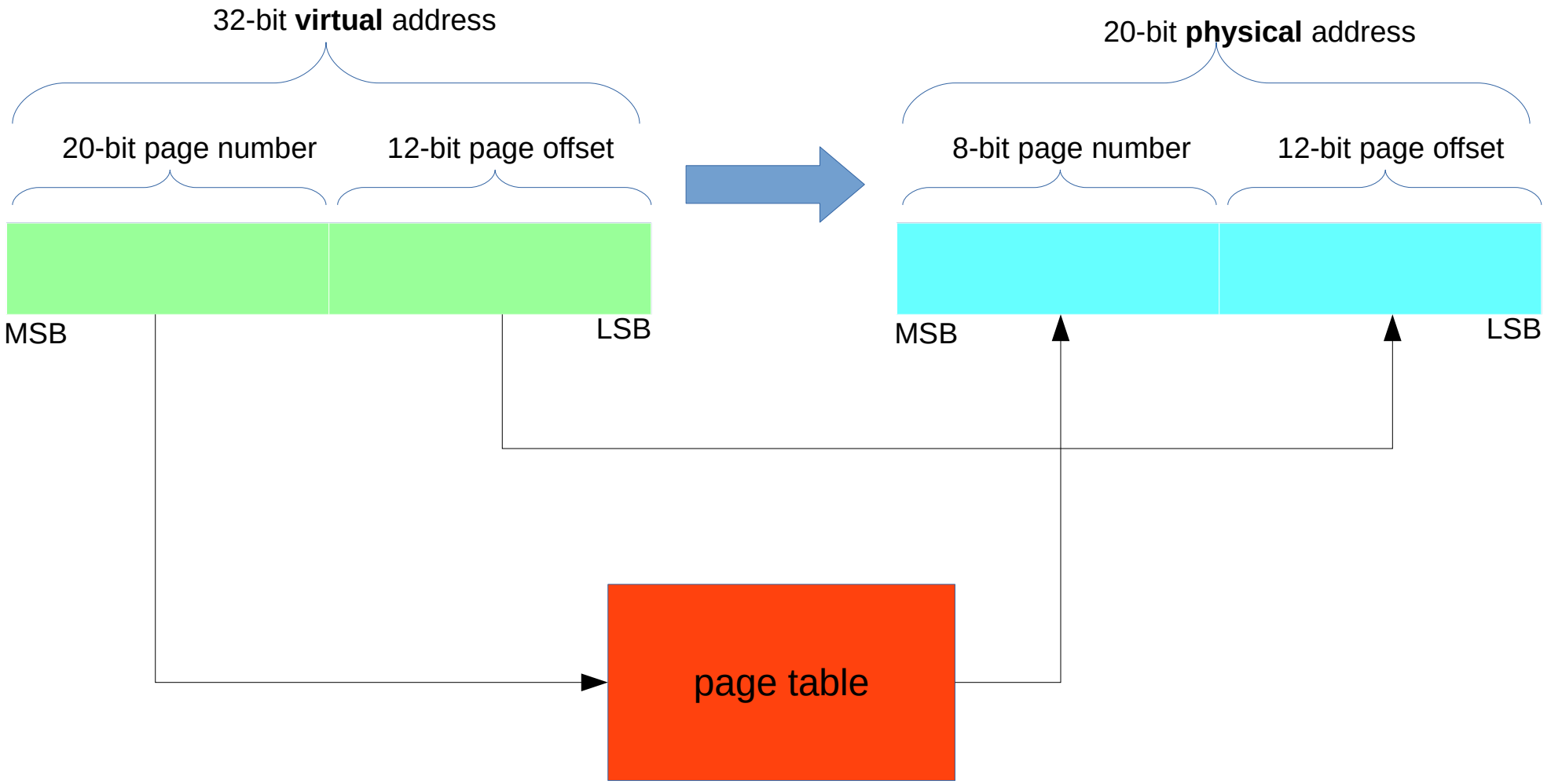
In other words:

```
page_num = addr // page_size
page_offset = addr % page_size
```

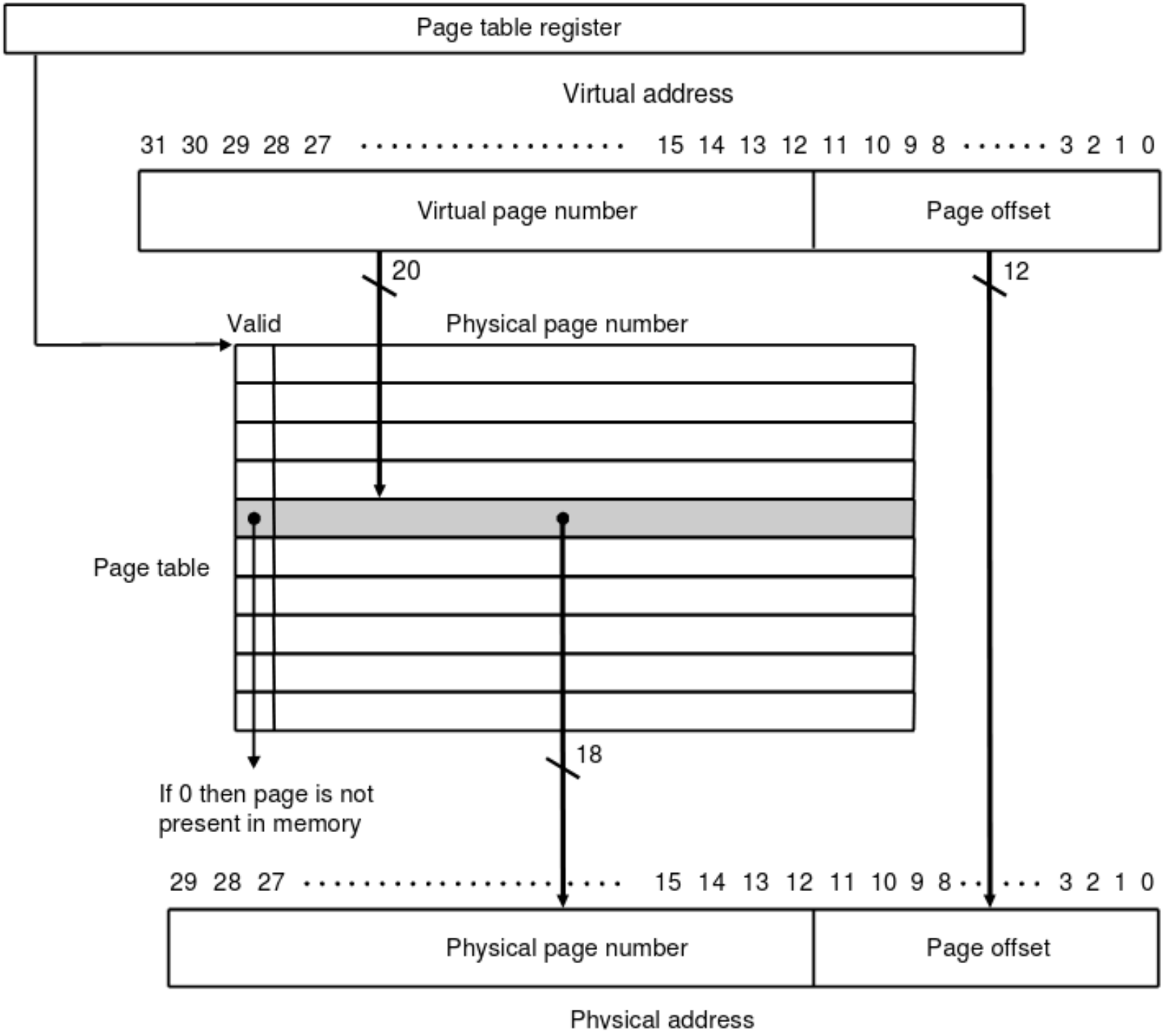

virtual address
address translation
physical address

Address translation

When translating a virtual address to a physical address, we are really translating a *virtual page* to *physical page*. The offset stays the same.



The *page table* is a data structure mapping virtual page numbers to physical page numbers.


Page table register
Virtual address
31 30 29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Virtual page number
Page offset
20
12
Valid
Physical page number
Page table
18
If 0 then page is not present in memory
29 28 27 ..... 15 14 13 12 11 10 9 8 ..... 3 2 1 0
Physical page number
Page offset
Physical address

A page table

A page table is just an array. The index is the virtual page. Each virtual matches to a physical page.

Virtual Page	Physical Page
0	34
1	12
2	0
3	50
4	13
5	14
....	...

The page table is created and managed by the operating system.

Typically, each running program (process) will have its own page table, allowing each process to have its own address space.

There's usually a *special register* that holds the physical address of the page table. The CPU performs a page table lookup as part of its address translation on every memory access.

```
page_offset = virtual_addr % page_size
virtual_page = virtual_addr // page_size
physical_page = page_table[virtual_page]
physical_addr = physical_page * page_size + page_offset
```

A page table

Caveat: not all virtual pages have a corresponding physical page. If an application accesses such a page, the hardware signals to the OS that the memory access cannot proceed: this is a *page fault*.

Thus we had a "valid" bit, which is checked before each lookup.

Virtual Page	V	Physical Page
0	1	34
1	0	
2	1	0
3	1	50
4	1	13
5	1	14
....		...

```
page_offset = virtual_addr % page_size
virtual_page = virtual_addr // page_size
physical_page = page_table[virtual_page]["physical_page"] if page_table[virtual_page]["v"]
physical_addr = physical_page * page_size + page_offset
```

What is the total size (in bits) of a page table for this system?

Assume a **32-bit virtual address**, consisting of a 20-bit virtual page number and a 12-bit offset.

Assume a **20-bit physical address**, consisting of an 8-bit physical page number and a 12-bit offset.

Assume the page table contains a physical page and valid bit for each entry.

What is the total size (in bits) of a page table for this system?

Assume a **32-bit virtual address**, consisting of a 20-bit virtual page number and a 12-bit offset.

Assume a **20-bit physical address**, consisting of an 8-bit physical page number and a 12-bit offset.

Assume the page table contains a physical page and valid bit for each entry.

How many entries are in the page table?

2**20 = 1048576

What is the total size (in bits) of a page table for this system?

Assume a **32-bit virtual address**, consisting of a 20-bit virtual page number and a 12-bit offset.

Assume a **20-bit physical address**, consisting of an 8-bit physical page number and a 12-bit offset.

Assume the page table contains a physical page and valid bit for each entry.

How many entries are in the page table?

2**20 = 1048576

How many bits is each entry?

1 valid bit + 8-bit physical page number

What is the total size (in bits) of a page table for this system?

Assume a **32-bit virtual address**, consisting of a 20-bit virtual page number and a 12-bit offset.

Assume a **20-bit physical address**, consisting of an 8-bit physical page number and a 12-bit offset.

Assume the page table contains a physical page and valid bit for each entry.

How many entries are in the page table?

2**20 = 1048576

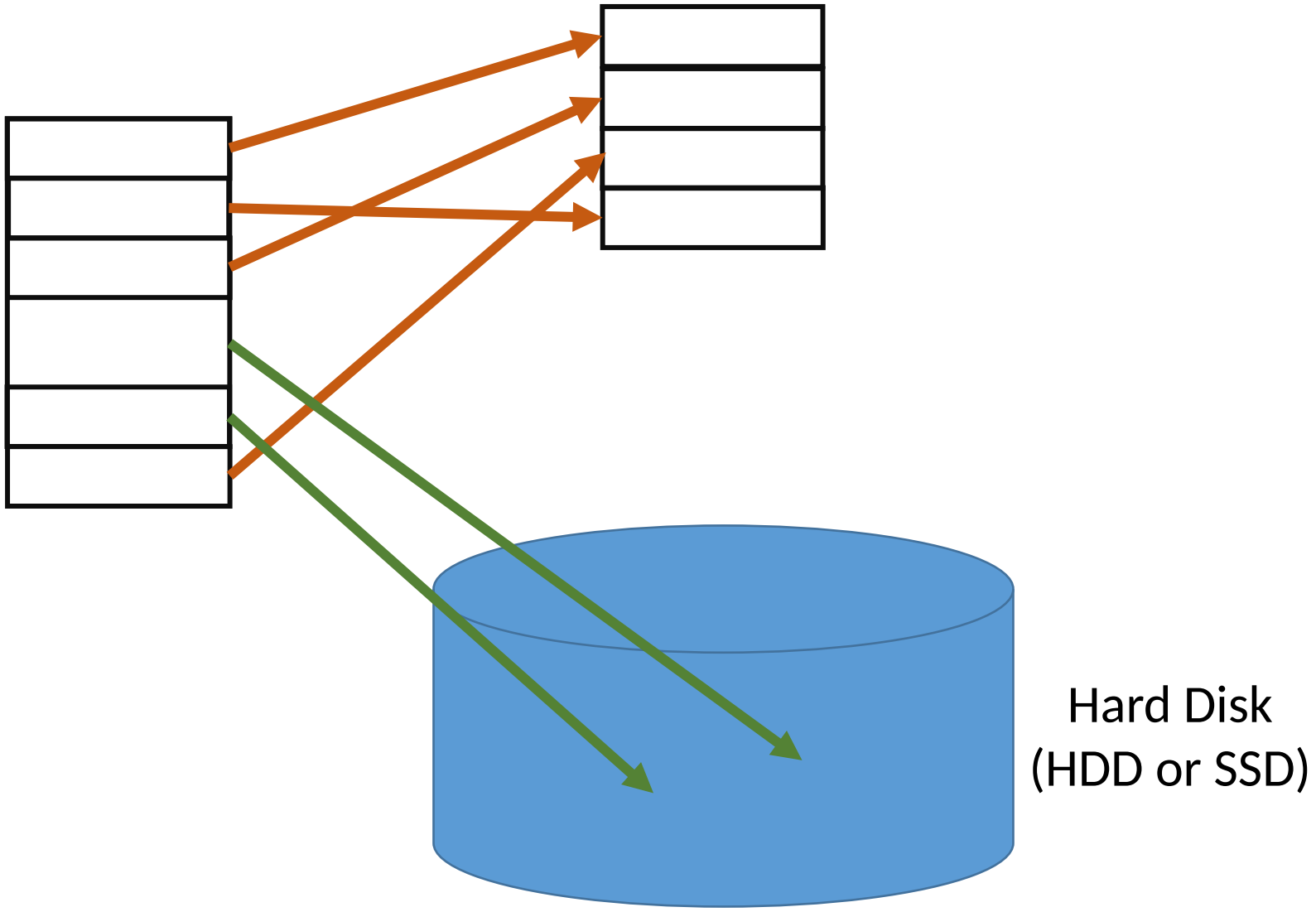
How many bits is each entry?

1 valid bit + 8-bit physical page number

What is the total size of the page table?

2**20 * (1 + 8) = 9437184 bits

Virtual Memory



Anything does not fit in main memory is stored on disk

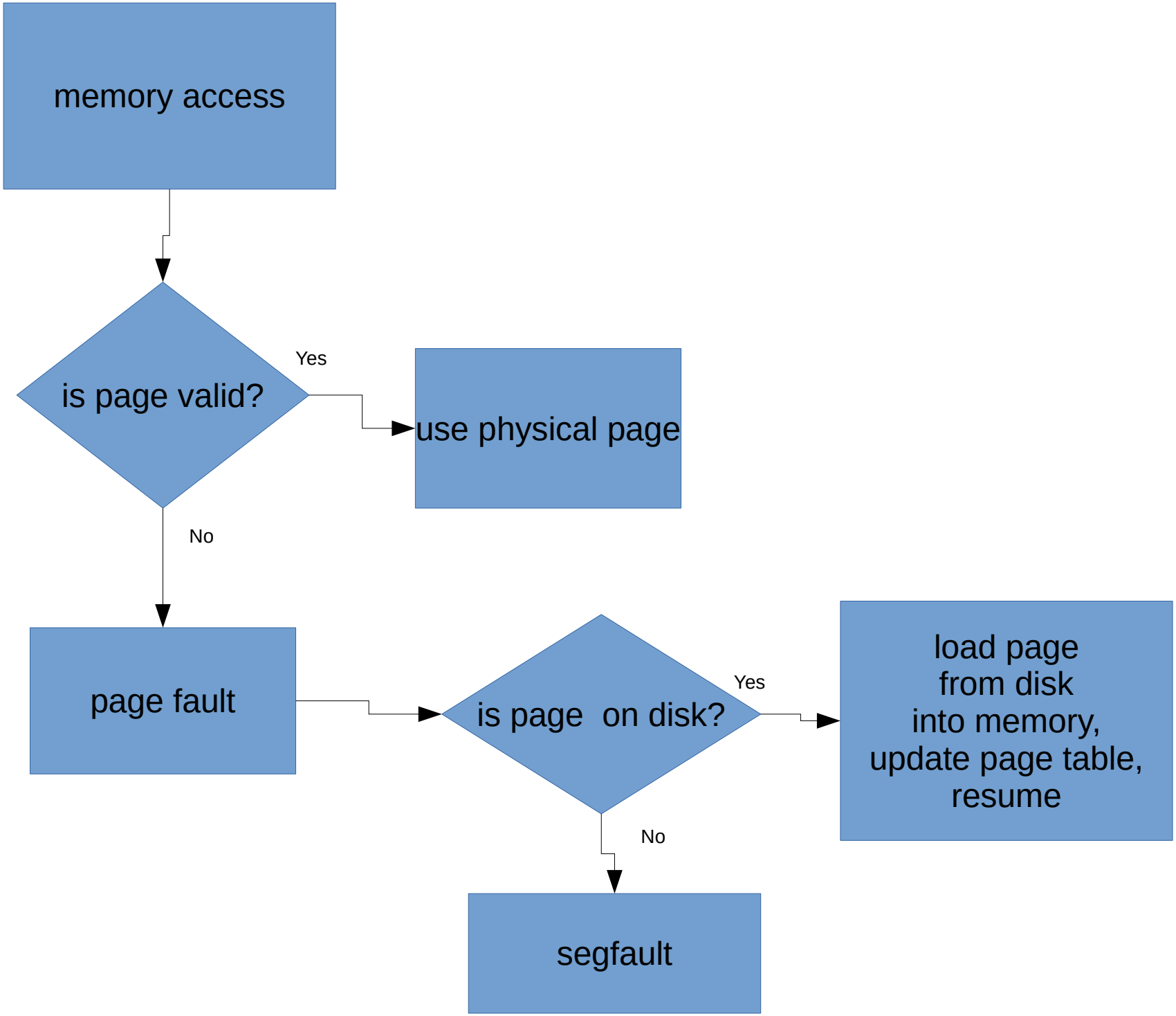
The swapping mechanism hopefully will:

give us the **size** of the disk,

with the **speed** of memory.

The design of virtual memory is based on the slowness of disk access.

We want to keep reduce misses.


memory access
is page valid?
Yes
use physical page
No
page fault
is page on disk?
Yes
load page
from disk
into memory,
update page table,
resume
No
segfault

Swapping

Swapping

The OS is responsible for allocating memory to applications.

If there is memory pressure (i.e. programs are using all available physical memory), what can the OS do?

Reject memory requests. Can't start programs, existing processes fail.

Or, the OS can choose to evict pages from memory and *swap* them to disk.

The corresponding page will be marked as invalid, so that when it's accessed, it will generate a page fault, whereupon it will be swapped back into memory.

The OS should prefer to swap seldom-used pages.

A page table

When we evict a page from memory, we don't always have to save it to disk:

- The page may be empty or unused.
- The page may have been loaded from a file (e.g. an executable), so there's no reason to save it to disk again.
- The page may already be on disk, from when it was previously swapped out.

In none of the above cases apply, we have to write the page to disk.

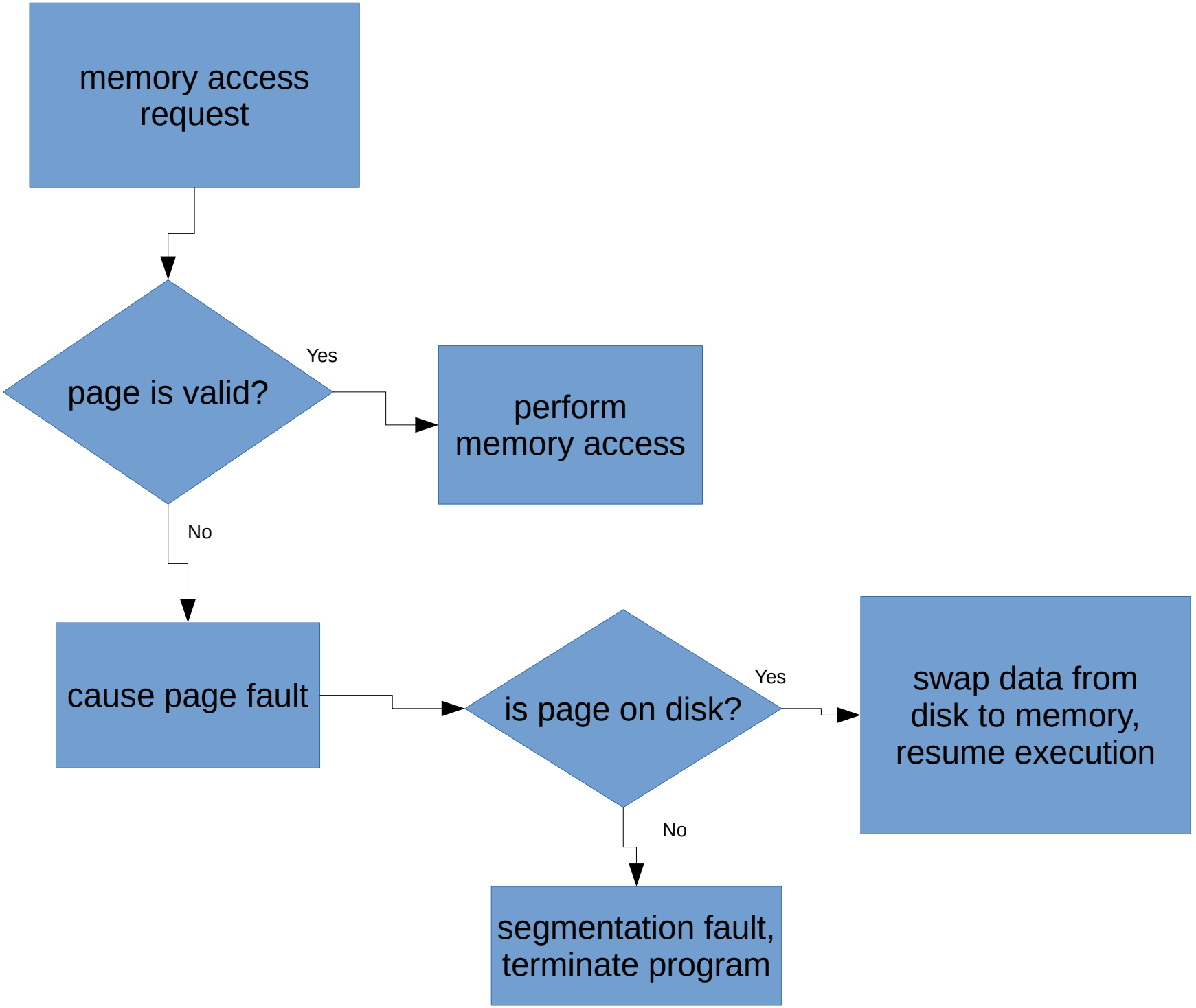
Virtual Page	V	Dirty	Physical Page
0	1	1	34
1	0		
2	1	0	0
3	1	1	50
4	1	0	13
5	1	0	14
....			...

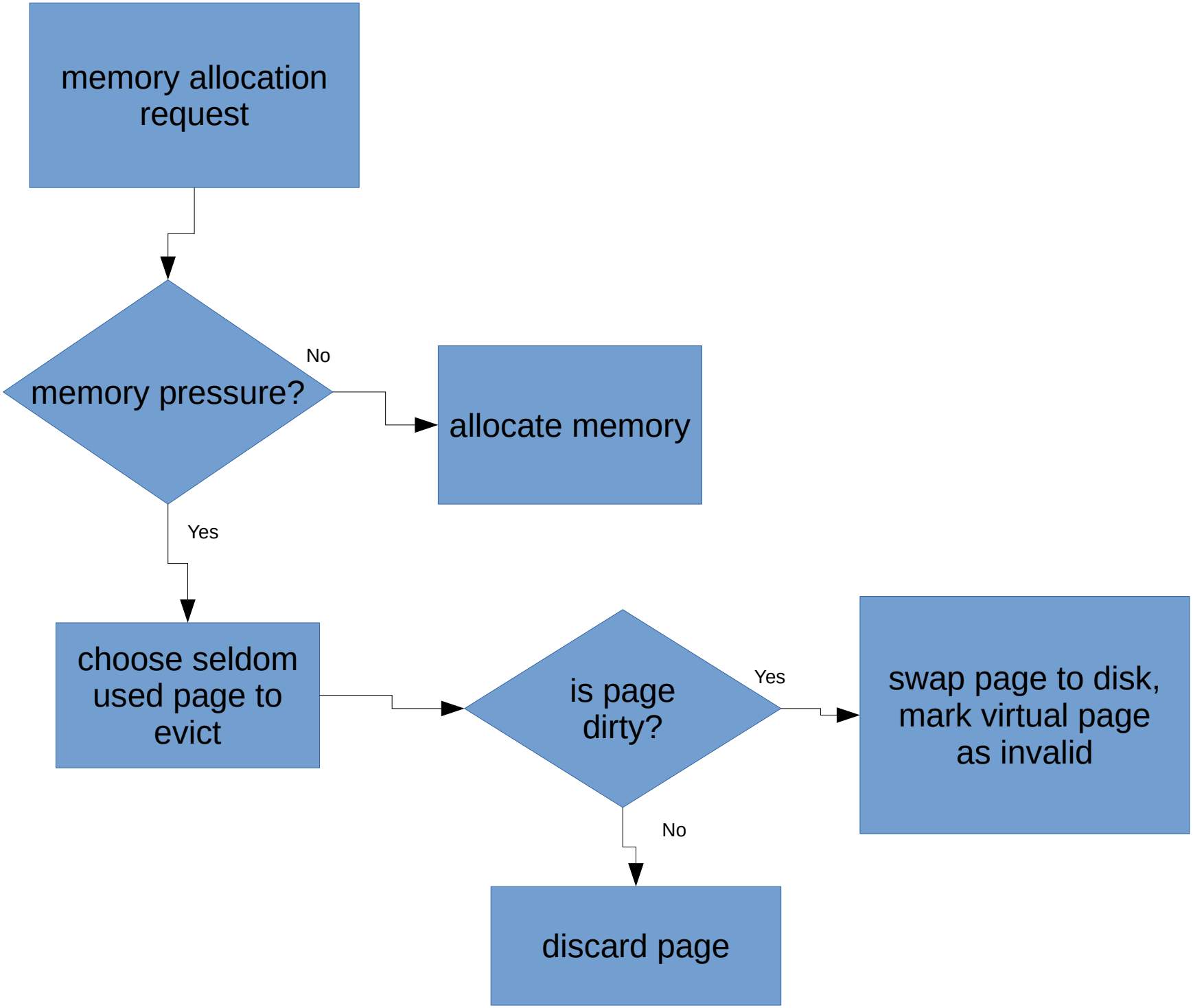
Every page initially has its dirty bit set to zero.

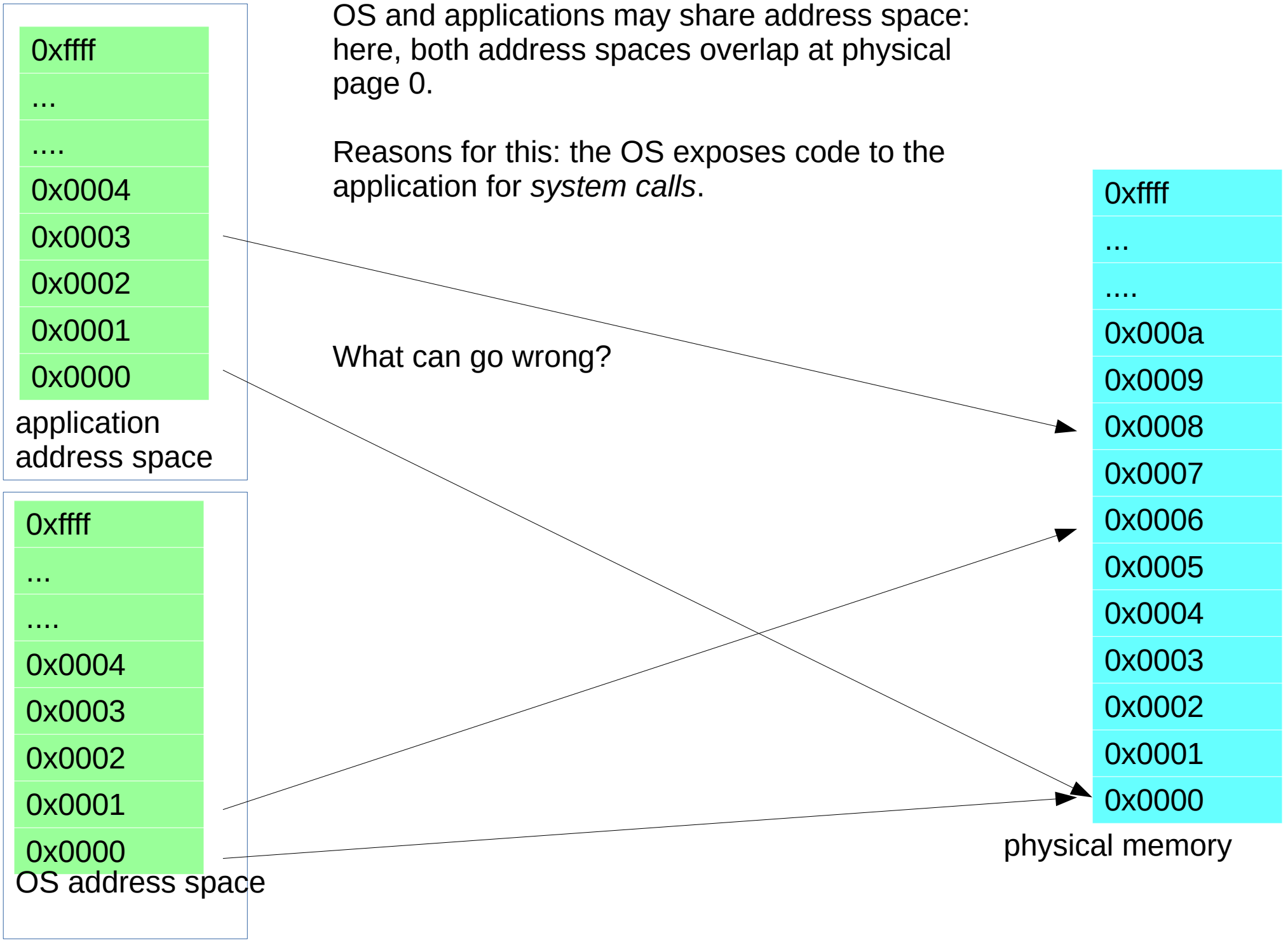
When a program does a write to that page, the processor sets the dirty bit to 1.

The OS will write the page to disk when swapping only if the dirty bit is 1.

This should all sound very familiar! With paging, *memory is a write-back cache of disk.*


memory access request
page is valid?
Yes
perform memory access
No
cause page fault
is page on disk?
Yes
swap data from disk to memory, resume execution
No
segmentation fault, terminate program


memory allocation request
memory pressure?
No
allocate memory
Yes
choose seldom used page to evict
is page dirty?
Yes
swap page to disk, mark virtual page as invalid
No
discard page


0xffff
...
....
0x0004
0x0003
0x0002
0x0001
0x0000
application address space
0xffff
...
....
0x0004
0x0003
0x0002
0x0001
0x0000
OS address space
OS and applications may share address space: here, both address spaces overlap at physical page 0.
Reasons for this: the OS exposes code to the application for system calls.
What can go wrong?
0xffff
...
....
0x000a
0x0009
0x0008
0x0007
0x0006
0x0005
0x0004
0x0003
0x0002
0x0001
0x0000
physical memory

A page table

To prevent the application from changing OS data, we introduce a *protect bit*. Pages marked with it cannot be written.

Virtual Page	V	Dirty	Protect	Physical Page
0	1	1	1	34
1	0			
2	1	0	0	0
3	1	1	0	50
4	1	0	0	13
5	1	0	0	14
....				...

A page table

Assume an 8-byte page size. For each of the following address, given a corresponding physical page, or indicate if a page fault will occur. If the page is not writeable, indicate so.

43, 32, 10, 5, 30

Virtual Page	V	Dirty	Protect	Physical Page
0	1	1	1	34
1	0			
2	1	0	0	0
3	1	1	0	50
4	1	0	0	13
5	1	0	0	14
....				...

A page table

Assume an 8-byte page size. For each of the following address, given a corresponding physical page, or indicate if a page fault will occur. If the page is not writeable, indicate so.

43, 32, 10, 5, 30

Virtual Page	V	Dirty	Protect	Physical Page
0	1	1	1	34
1	0			
2	1	0	0	0
3	1	1	0	50
4	1	0	0	13
5	1	0	0	14
....				...

virtual address 43. Page 5, offset 3. Physical address 115.
virtual address 32. Page 4, offset 0. Physical address 104.
virtual address 10. Page 1, offset 2. Page fault!
virtual address 5. Page 0, offset 5. Physical address 277. Read only.
virtual ddresss 30. Page 3, offset 6. Physical address 406.

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	1	0
3	1	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.
3) When the program writes to these new pages, they are marked dirty.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	1	1
3	1	1
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.
3) When the program writes to these new pages, they are marked dirty.
4) Now, some other application causes memory pressure, and the OS decides to swap out our program's pages 2 and 3 to disk. They are now marked invalid and not dirty.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}
swapfile -> {2,3}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.
3) When the program writes to these new pages, they are marked dirty.
4) Now, some other application causes memory pressure, and the OS decides to swap out our program's pages 2 and 3 to disk. They are now marked invalid and not dirty.

Why would the OS swap out 2 and 3,
instead of (non-dirty) 0 and 1?
It's possible the program was still running,
resulting in reads to machine code in pages 0 and 1.
Therefore pages 0 and 1 may be used more recently
than 2 and 3.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}
swapfile -> {2,3}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.
3) When the program writes to these new pages, they are marked dirty.
4) Now, some other application causes memory pressure, and the OS decides to swap out our program's pages 2 and 3 to disk. They are now marked invalid and not dirty.
5) When our program tries to access those pages, it will cause a page fault. The OS will swap them back in and mark them valid. They are still marked not dirty, because the copy on disk is still present.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	1	0
3	1	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}
swapfile -> {2,3}

Program walkthrough

1) When you run a program from an executable file, the content of the file is copied into memory. In this case, the program machine occupies virtual pages 0 and 1. They are valid, because they are in memory, and they are not dirty, because their content is directly copied from the file.
2) As the program runs, it requests additional memory from the operating system. The OS handles this by mapping additional pages 2 and 3 into its address space. They are now marked as valid. They are empty, and so are not dirty.
3) When the program writes to these new pages, they are marked dirty.
4) Now, some other application causes memory pressure, and the OS decides to swap out our program's pages 2 and 3 to disk. They are now marked invalid and not dirty.
5) When our program tries to access those pages, it will cause a page fault. The OS will swap them back in and mark them valid. They are still marked not dirty, because the copy on disk is still present.
6) If the OS decides again detects memory pressure, it is safe to simply discard these pages, so copies already exist on disk.

Virtual Page Num	Valid?	Dirty?
0	1	0
1	1	0
2	0	0
3	0	0
4	0	0
5	0	0
6	0	0

On disk:
program.exe -> {0,1}
swapfile -> {2,3}

How big is a page table?

Consider a system with 32-bit virtual addresses, 32-bit physical addresses, and 4KB pages. How big will the page tables be (including metadata)? Assume each memory cell is one byte.

How big is a page table?

Consider a system with 32-bit virtual addresses, 32-bit physical addresses, and 4KB pages. How big will the page tables be (including metadata)? Assume each memory cell is one byte.

There will be 2**20 page table entries.

Each entry will contain the 20-bit physical page, plus the dirty, protect, and valid bits.

So the total size is 2**20 * (20 + 3) = 24,117,248 bits, or 3,014,656 bytes.

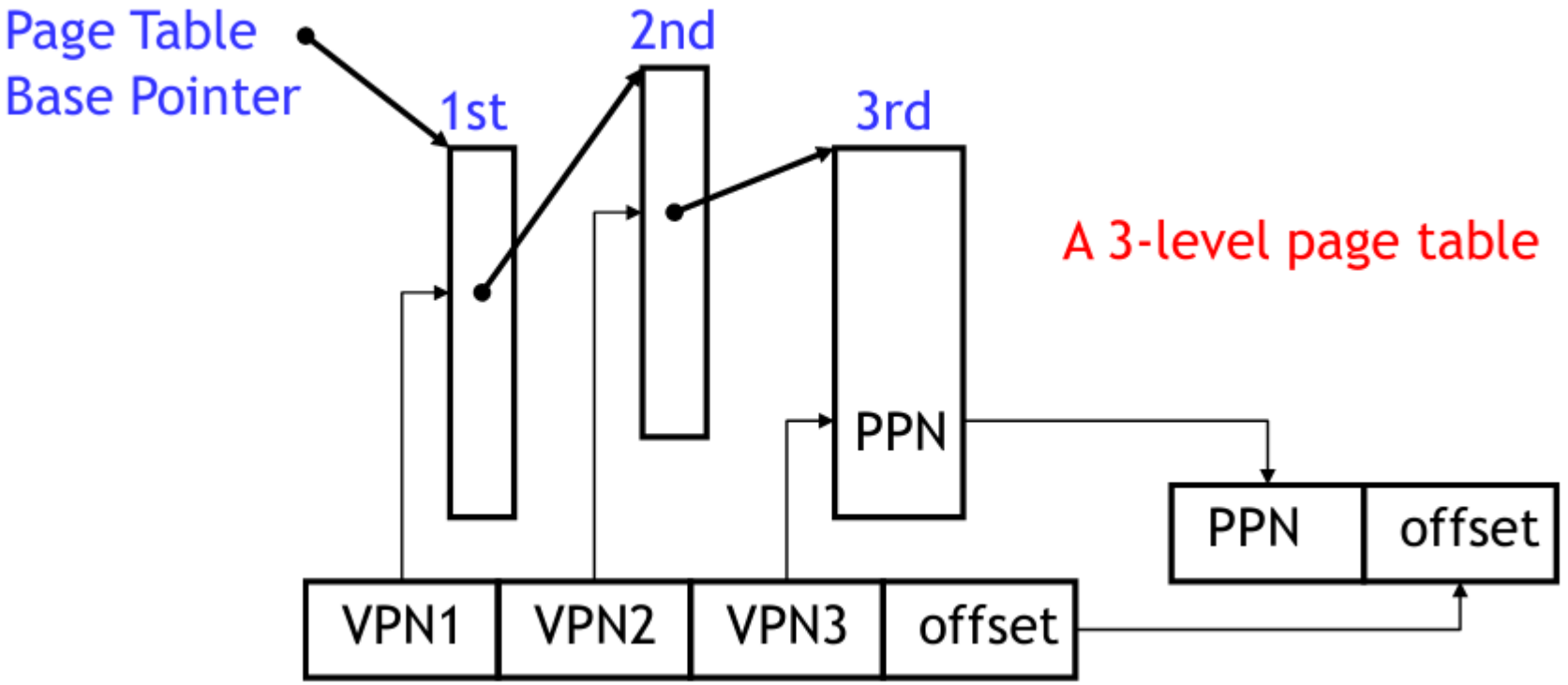
Each process will have a page table. That adds up!

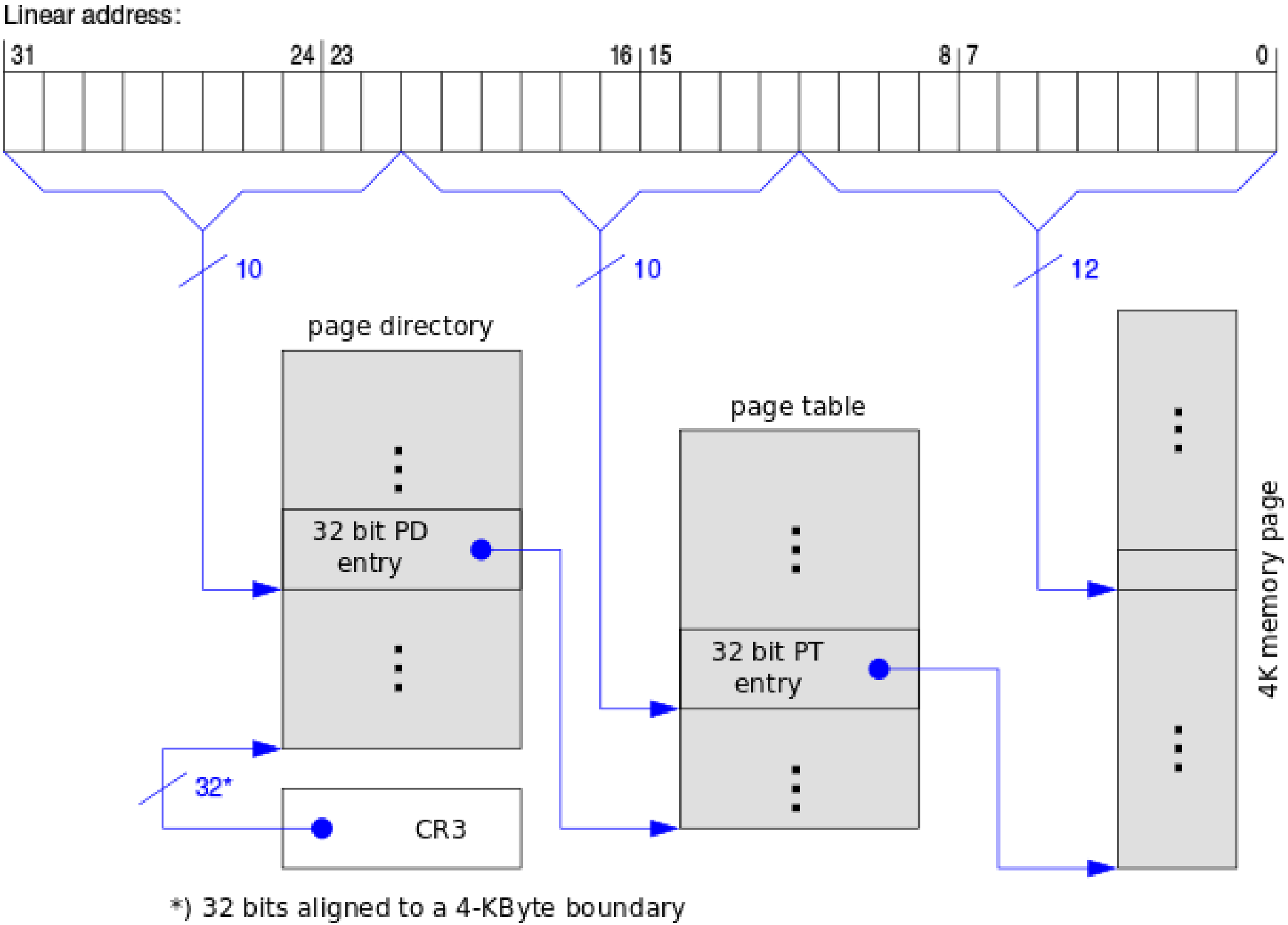
Further observation: most programs use only a small part of their address space. Most page tables are mostly empty.

Another observation: with a page table, every memory access by an application entails *two* (2) round-trip accesses to memory: first to translate the address in the page table, then to access the actual address. So we've just cut the speed of our memory unit in half.

Page tables

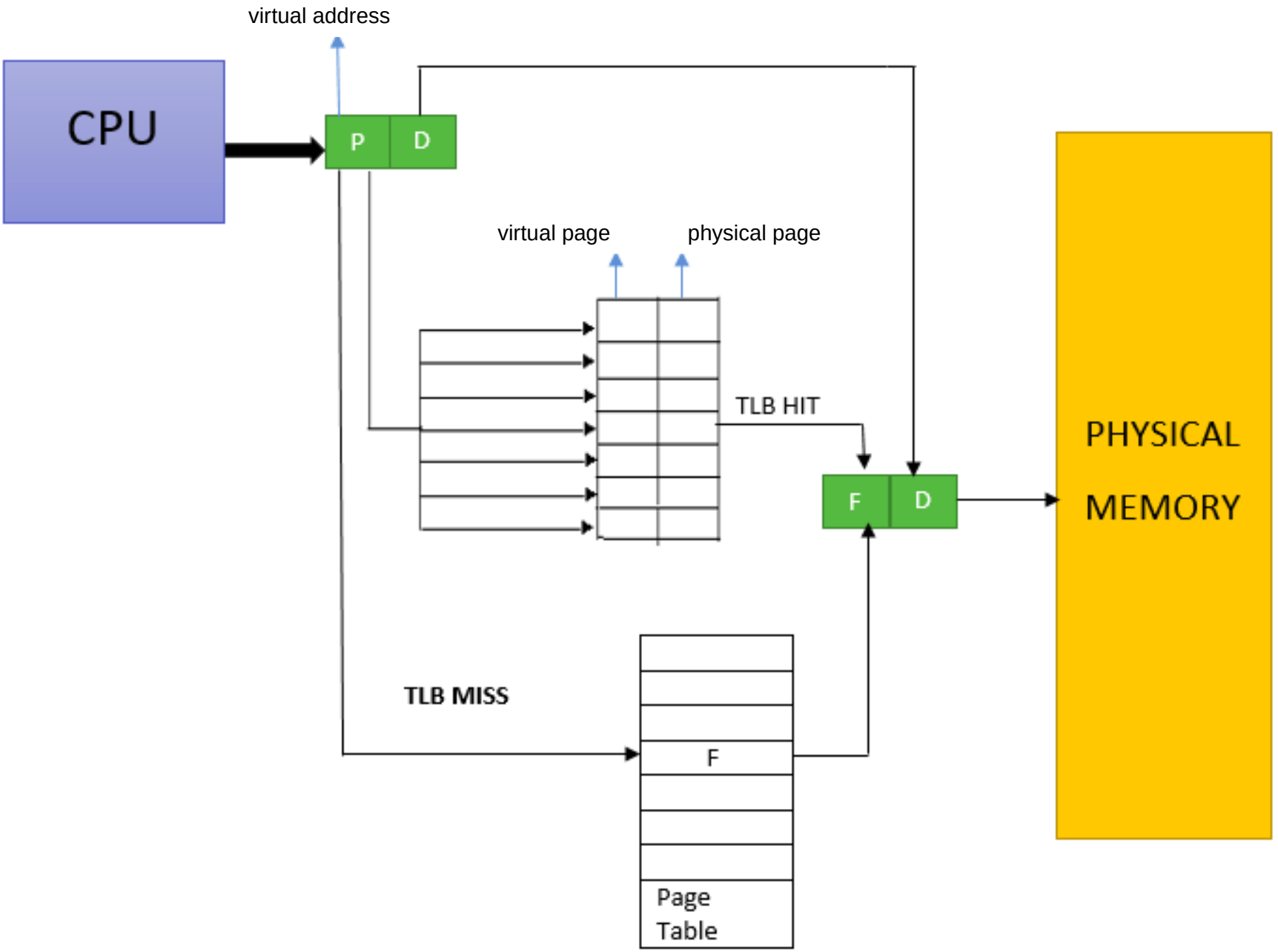
- Solutions to too-big page table
  - We swap it out, of course
  - The location of the page table is stored by another page table
  - We thus have a hierarchy of page tables, much as we had a hierarchy of caches
- Solutions to too-slow page table
  - We cache it, of course
  - Virtual-to-physical page translations are cached in the Translation Lookaside Buffer (TLB)


Page Table
Base Pointer
1st
2nd
3rd
A 3-level page table
PPN
PPN
offset
VPN1
VPN2
VPN3
offset

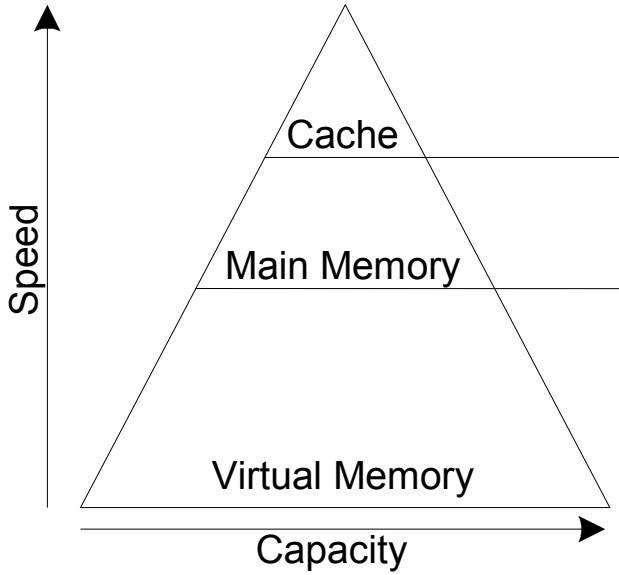

Linear address:
31
24
23
16
15
8
7
0
10
10
12
page directory
32 bit PD entry
page table
32 bit PT entry
4K memory page
32*
CR3
*) 32 bits aligned to a 4-KByte boundary

TLB

- The TLB (Translation Lookaside Buffer) is a cache of the page table
- It maps virtual addresses to physical addresses
  - The tag is the virtual page
  - The value stored is the physical page
  - Also has valid, dirty, and protect bits
  - TLB is consulted before checking the page table
- It's usually a fully-associative cache.


virtual address
CPU
P
D
virtual page
physical page
TLB HIT
F
D
PHYSICAL
MEMORY
TLB MISS
F
Page
Table

Memory Hierarchy



Technology	Price / GB	Access Time (ns)	Bandwidth (GB/s)
SRAM	$10,000	1	25+
DRAM	$10	10 - 50	10
SSD	$1	100,000	0.5
HDD	$0.1	10,000,000	0.1

Virtual Memory Example

- **System:**
  - Virtual memory size: 4 GB = 2^{32} bytes
  - Physical memory size: 1 GB = 2^{30} bytes
  - Page size: 4 KB = 2^{12} bytes
- **Organization:**
  - Virtual address: **32** bits
  - Physical address: **30** bits
  - Page offset: **12** bits
  - # Virtual pages = $2^{32}/2^{12}$ = $\mathbf{2^{20}}$ (VPN = 20 bits)
  - # Physical pages = $2^{30}/2^{12}$ = $\mathbf{2^{18}}$ (PPN = 18 bits)

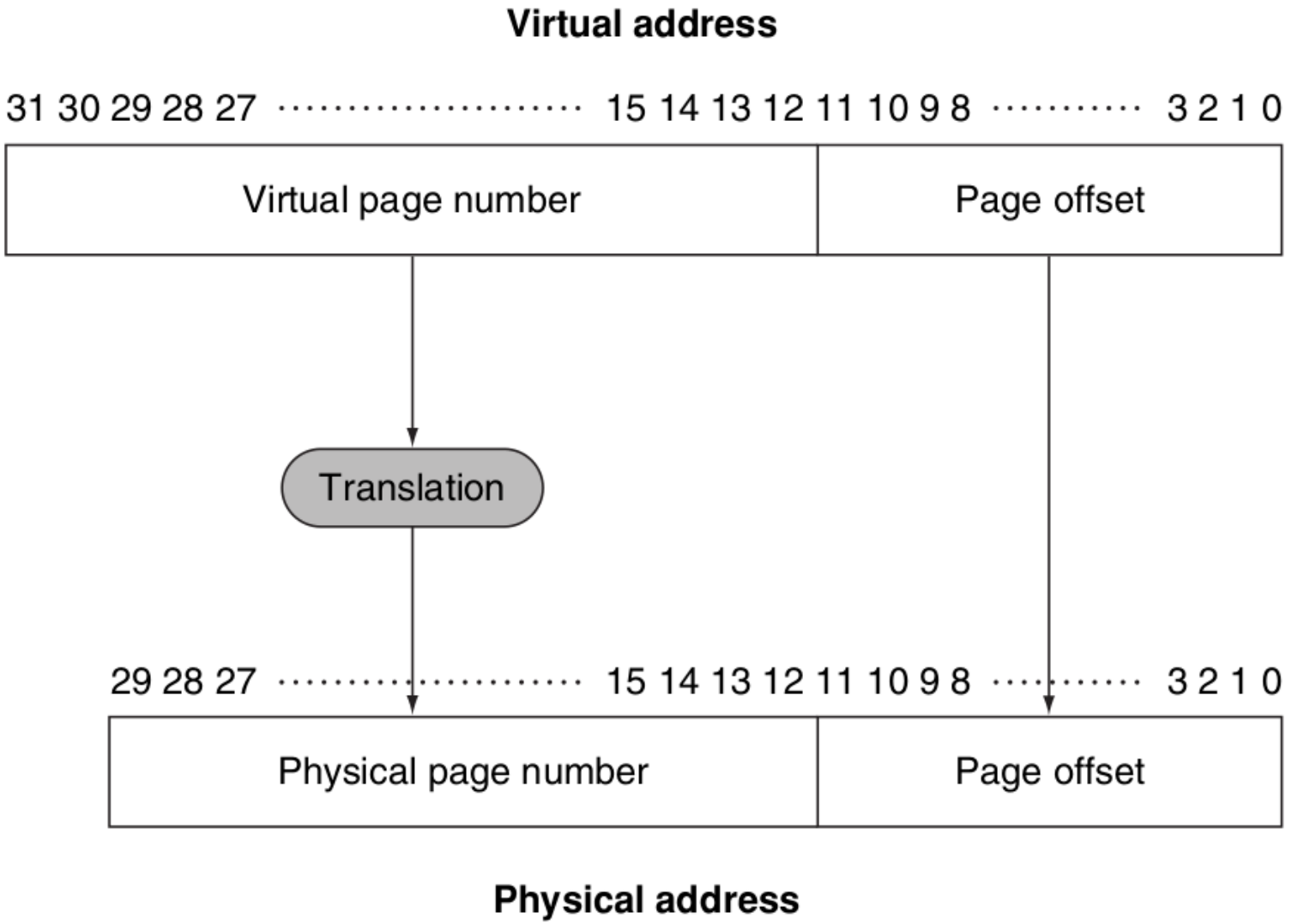
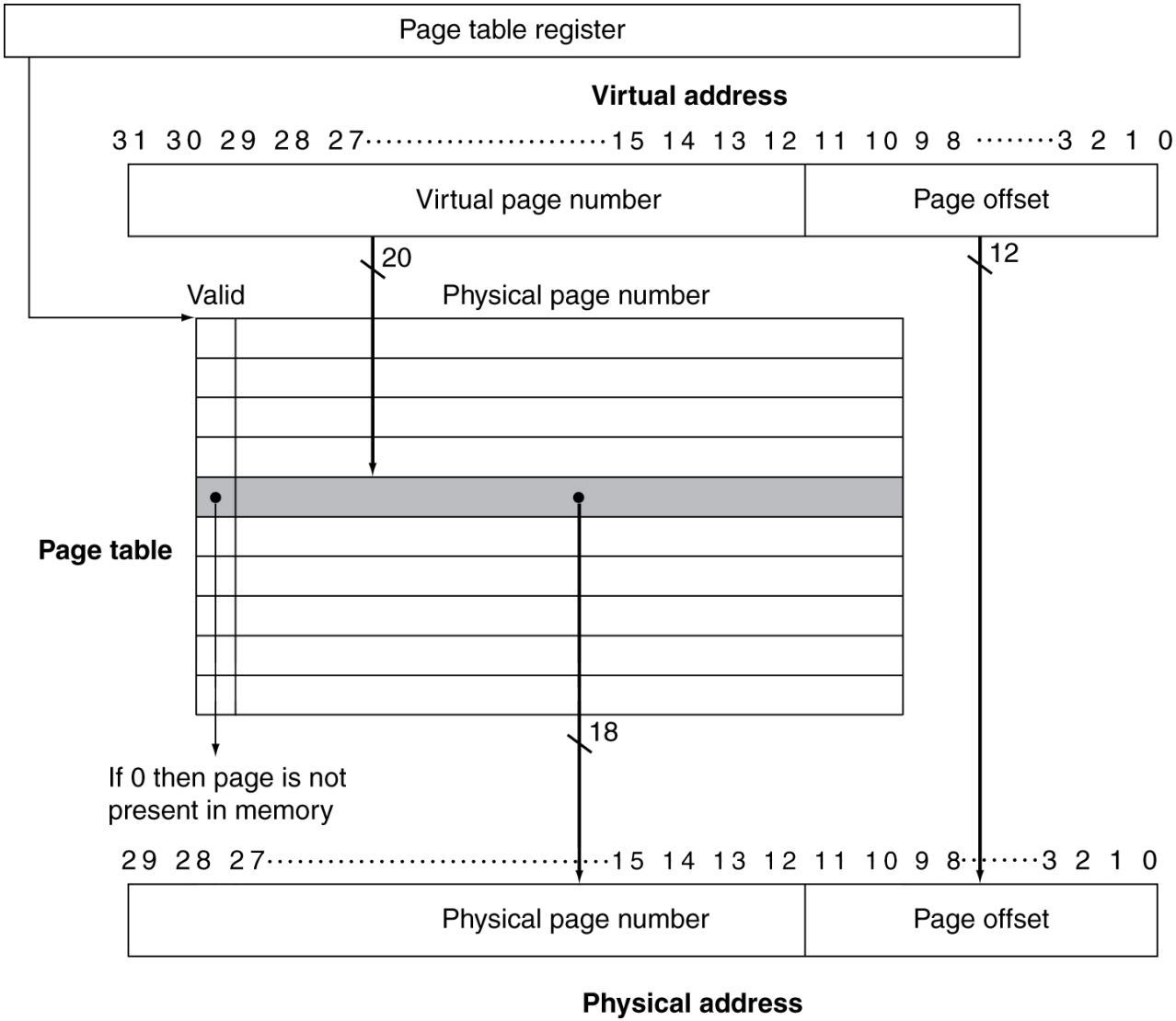
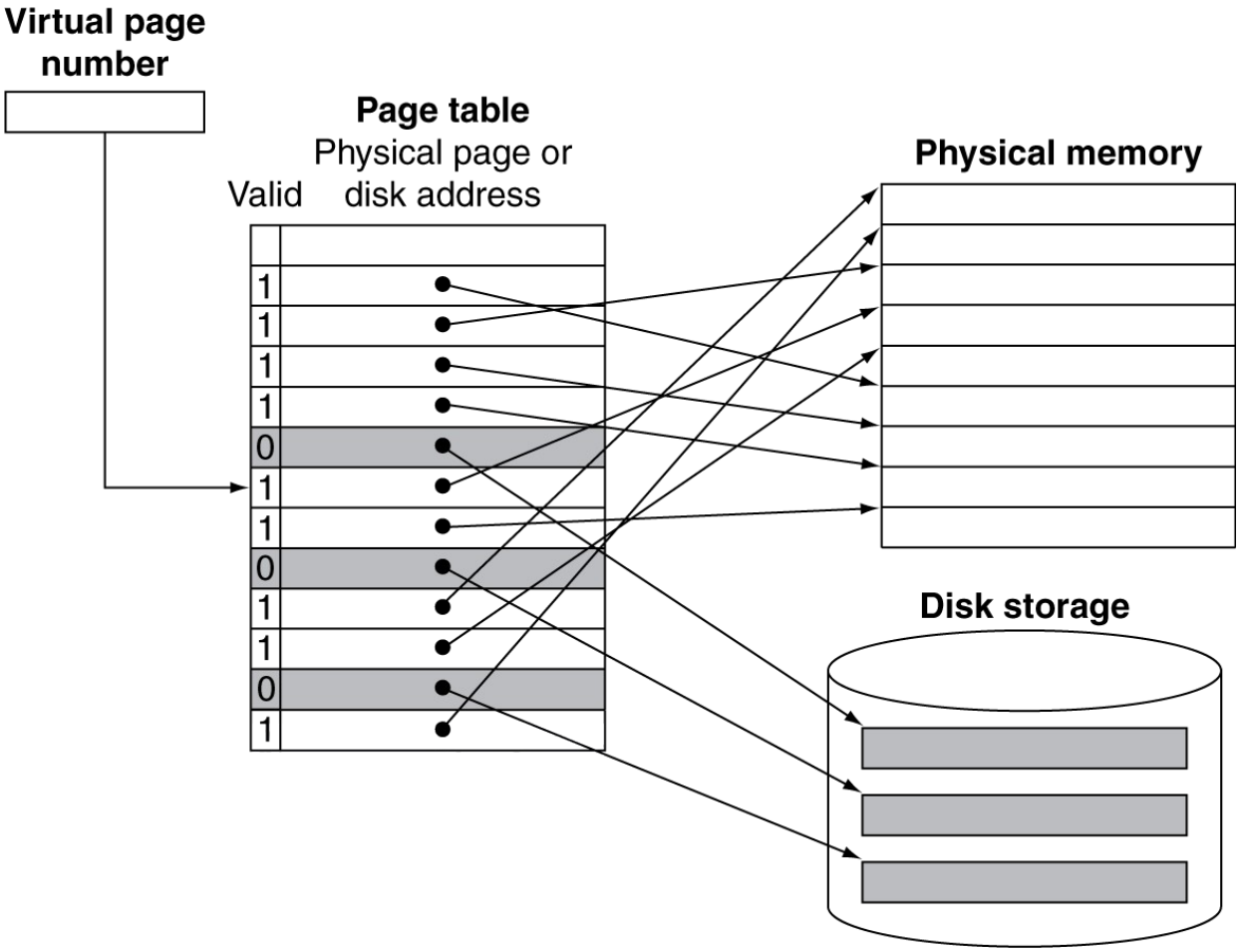


FIGURE 5.26 Mapping from a virtual to a physical address. The page size is $2^{12} = 4$ KiB. The number of physical pages allowed in memory is 2^{18}, since the physical page number has 18 bits in it. Thus, main memory can have at most 1 GiB, while the virtual address space is 4 GiB.

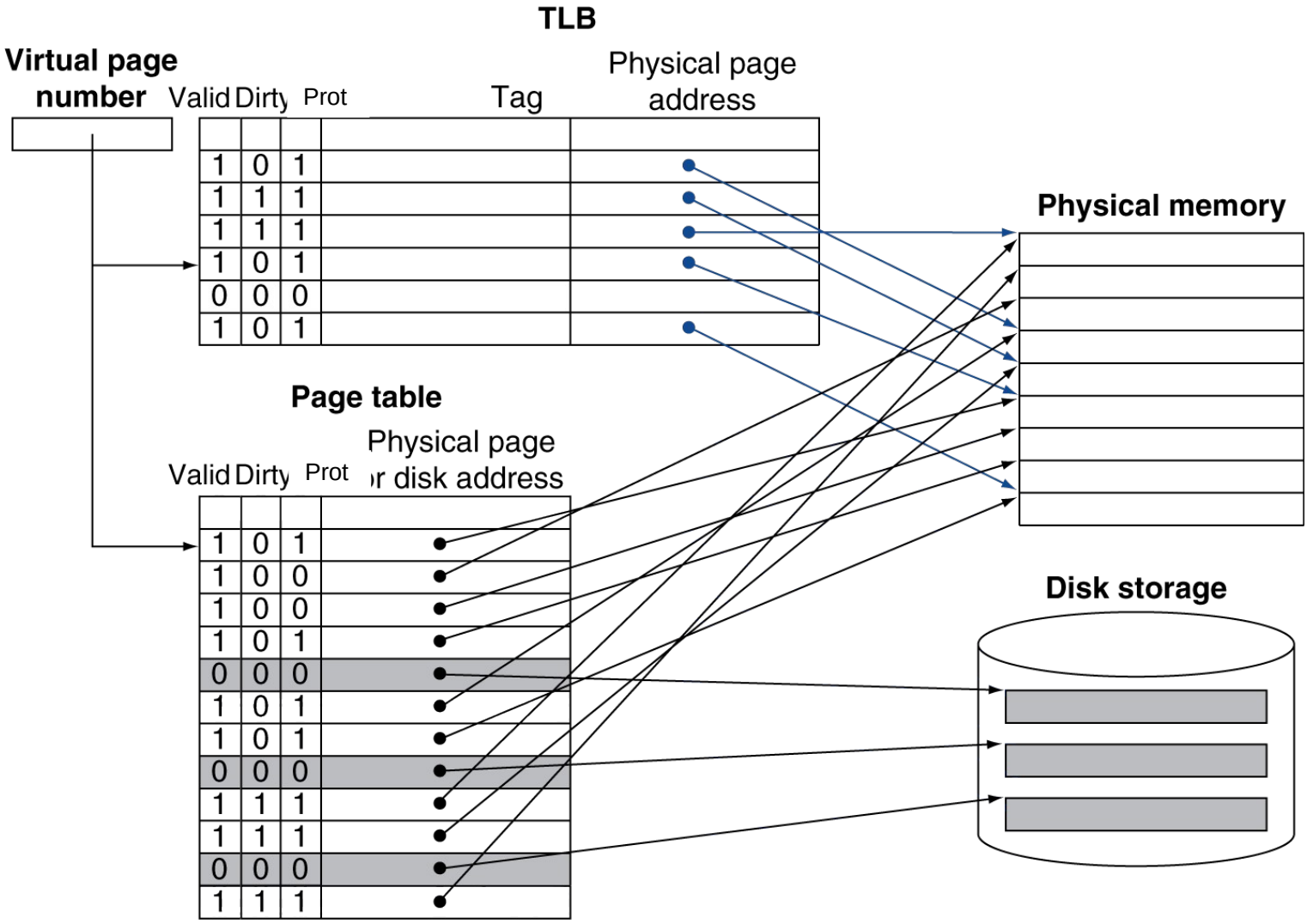
Translation Using a Page Table



Mapping Pages to Storage



Fast Translation Using a TLB



Hierarchical page tables

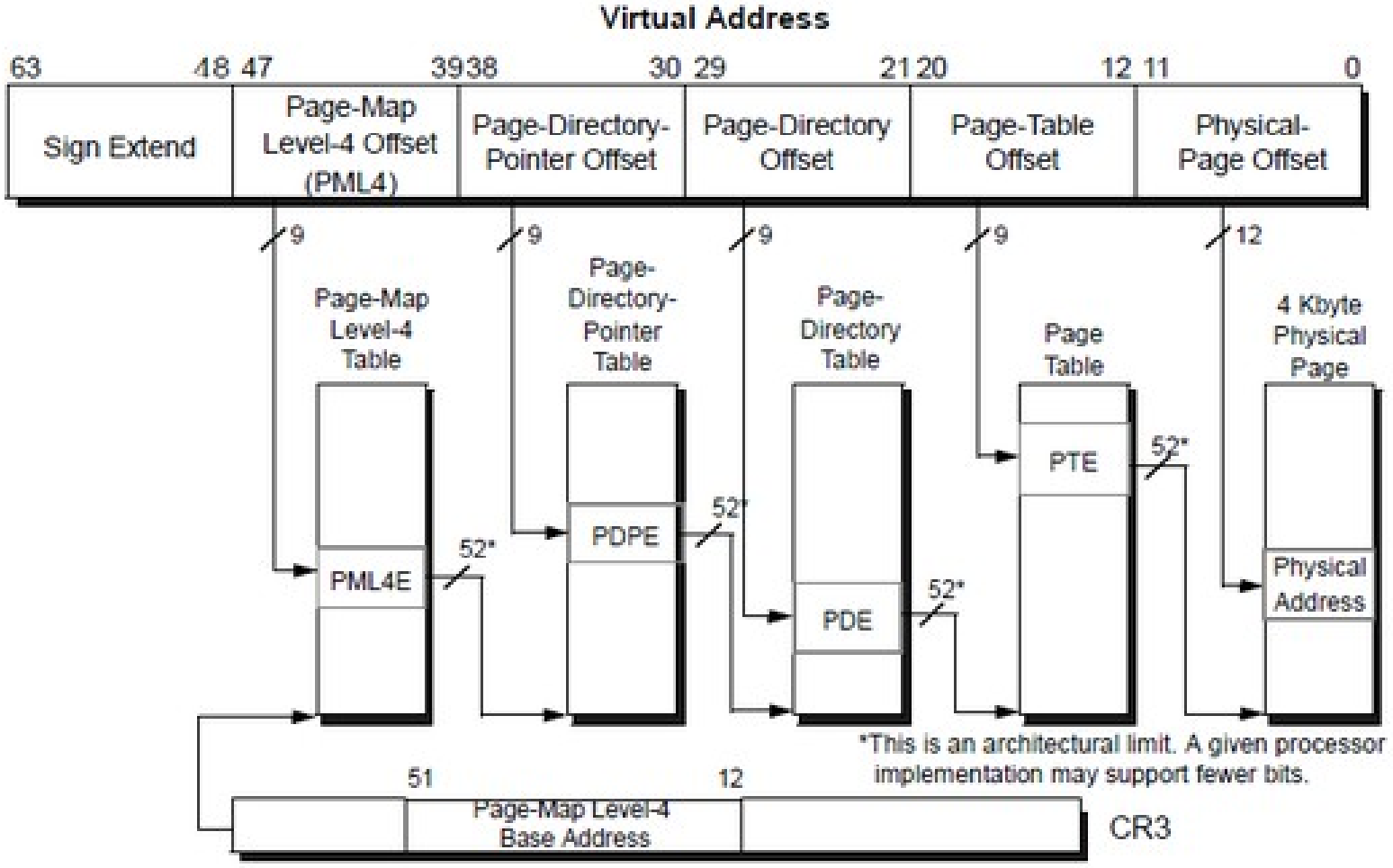
Modern 64-bit Intel-based systems have 4 or 5 levels of page tables.

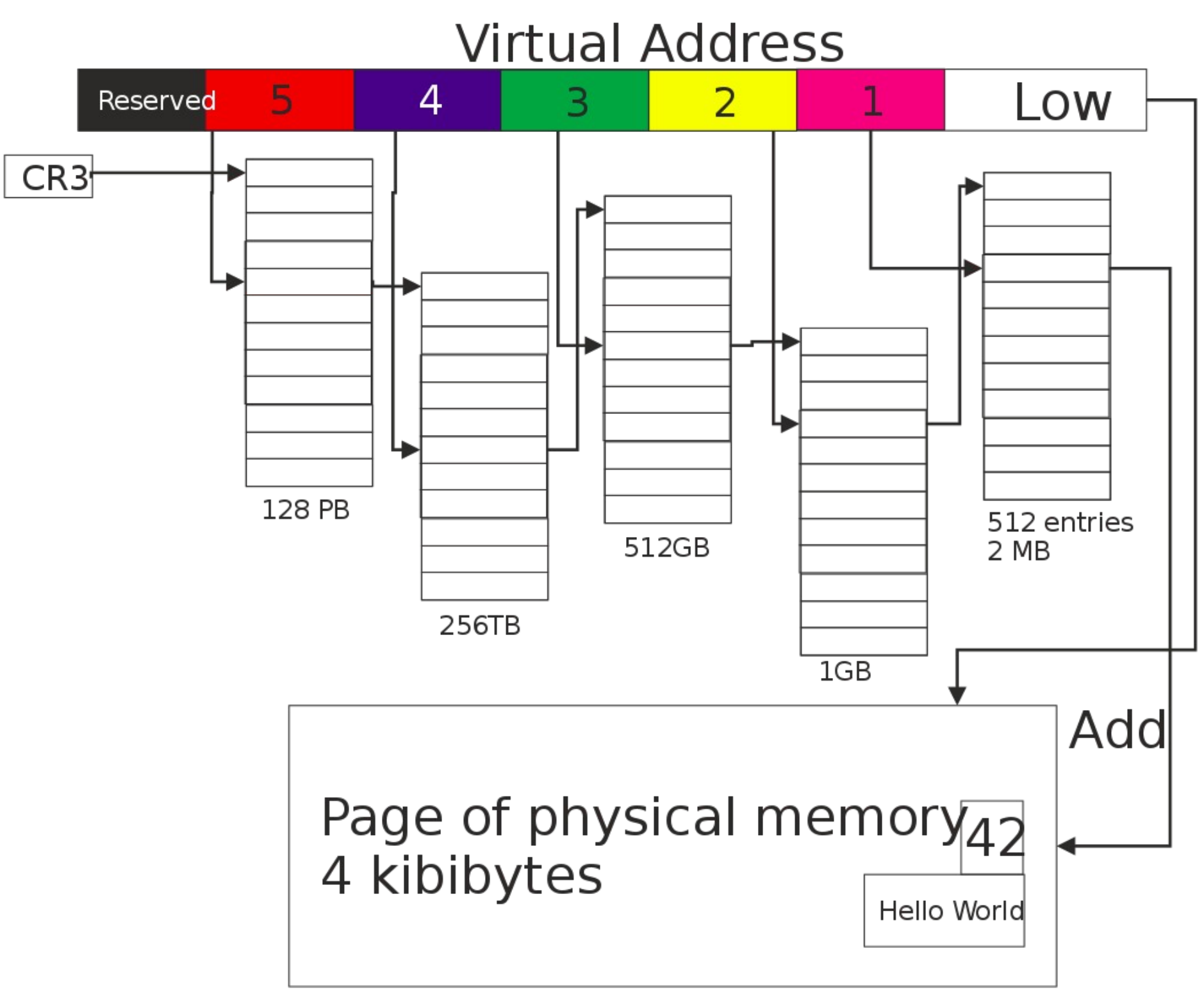
Each layer has a 9-bit index.

4-level page tables allow virtual address of 48 bits, address space of 256 TiB.

5-level page tables allow virtual address of 57 bits, address space of 128 PiB.

Typically, low virtual address are for userland, high virtual addresses are reserved for kernel.


Virtual Address
63
48
47
39
38
30
29
21
20
12
11
0
Sign Extend
Page-Map Level-4 Offset (PML4)
Page-Directory-Pointer Offset
Page-Directory Offset
Page-Table Offset
Physical-Page Offset
9
9
9
9
12
Page-Map Level-4 Table
Page-Directory-Pointer Table
Page-Directory Table
Page Table
4 Kbyte Physical Page
PML4E
PDPE
PDE
PTE
Physical Address
52*
52*
52*
52*
*This is an architectural limit. A given processor implementation may support fewer bits.
51
12
Page-Map Level-4 Base Address
CR3


Virtual Address
Reserved
5
4
3
2
1
Low
CR3
128 PB
256TB
512GB
1GB
512 entries
2 MB
Add
Page of physical memory
4 kibibytes
42
Hello World

Note that the most significant bits of virtual addresses are unused. Rather than simply be zero, they are *sign extended*, that is, they must all have the same value as the most significant used bit of the virtual address.