Cornell Bowers CIS
Computer Science

CS3410: Computer Systems and Organization

LEC28: Review of Calling Convention, Caches, Virtual Memory

Professor Giulia Guidi
Monday, December 8, 2025

Credits: Bala, Bracy, Garcia, Guidi, Kao, Sampson, Sirer, Weatherspoon

Final Exam

GDB **not** on the final, but everything else covered in class or lab can be in the exam

Our **final exam** is coming up **Saturday, December 13**:

- 7 PM in Kennedy Hall KND116 (same as main prelim)

Quick reminders:

- Please **use the restroom before the exam**
- During testing **no devices** (phones, calculators, watches, headphones, etc.) are permitted
- Don't forget to **bring your Cornell ID** since we'll use it to check you in and swap for your netID

Calling convention review

Purpose of Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

- E.g., how **arguments** are passed
- E.g., how **return values** are returned
- E.g., how **registers** and the **stack** are managed
- E.g., **who** is responsible for saving/restoring what

The essentially the "**contract**" between the caller (e.g., `main`) and callee, e.g., function `f()`

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **ra** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **sp** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (a0–a1 only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

RISC-V Calling Convention

I have **8** registers for the **arguments**. But what happens if my function has **10** arguments?

Register	Use
`x1 → ra`	The **return** address
`x2 → sp`	The **stack** pointer
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved)
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved)

The first 8 integer arguments → `a0–a7`

Remaining arguments → pushed onto the **stack**

RISC-V Calling Convention

A calling convention is a set of rules that defines **how functions communicate**:

Register	Use
`x1 → ra`	The **return** address — If `main` calls a function `add`, **ra** stores the address of the instruction following the call to `add` in `main`
`x2 → sp`	The **stack** pointer — **sp** stores the address in memory of the top of the **stack**
`x5–x7 → t0–t2`	The **temporary** registers (caller-saved) — The subroutine **can** modify these values
`x10–x17 → a0–a7`	The function **arguments** and **return** values (`a0–a1` only)
`x8–x9 → s0–s1`	The **saved** registers (callee-saved) — The subroutine e.g., `add` can touch these values, but it **must** restore them

Caller-saved: Caller must save if it wants the value after the call

Callee-saved: Callee must preserve them across the call

Function Call

Let's go through the `addOne` function **execution**:

```
int addOne(int i) {
  return i + 1;
}
```

`addOne` is a leaf function, meaning it does **not** call another fuction, so we do **not** actually need to store `ra` on the stack

The assembly can be just:

```
# body
addi a0, a0, 1   # i + 1 (i passed to addOne in a0) and place return value in a0
ret
```

Function Call

Let's go through the addTwo function **execution**:

```
int addTwo(int i) {
  i = incrementOne(i)
  return i + 1;
}
```

```
int incrementOne(int x) {
  x = x + 1;
  return x;
}
```

3

```
# incrementOne body (leaf)
addi a0, a0, 1 # x = x + 1
ret            # leaf, ra untouched
# remember "ret = jr ra = jalr x0, 0(ra)"
```

1

```
# addTwo prologue
addi sp, sp, -8   # make stack space
sd   ra, 0(sp)    # save the caller return address onto the stack
```

2

```
# addTwo body (non-leaf)
call incrementOne   # overwrites ra
addi a0, a0, 1      # place return value in a0
```

4

```
# addTwo epilogue
ld   ra, 0(sp) # non-leaf, ra restored
addi sp, sp, 8
ret
```

I assumed 64-bit architecture and I did not enforce 16-byte alignment

Function Call Example

a0 a1 a2 a3

g, h, i, and j are **arguments** so they are stored in a0–a7 (a0–a3, in this case)

```
int Leaf (int g, int h, int i, int j)
{
  int f;  ——→ s0
  f = (g + h) – (i + j);
  return f;
}
```

callee-saved

In this example, I'm using s0 as a temporary register to store the result of the computation

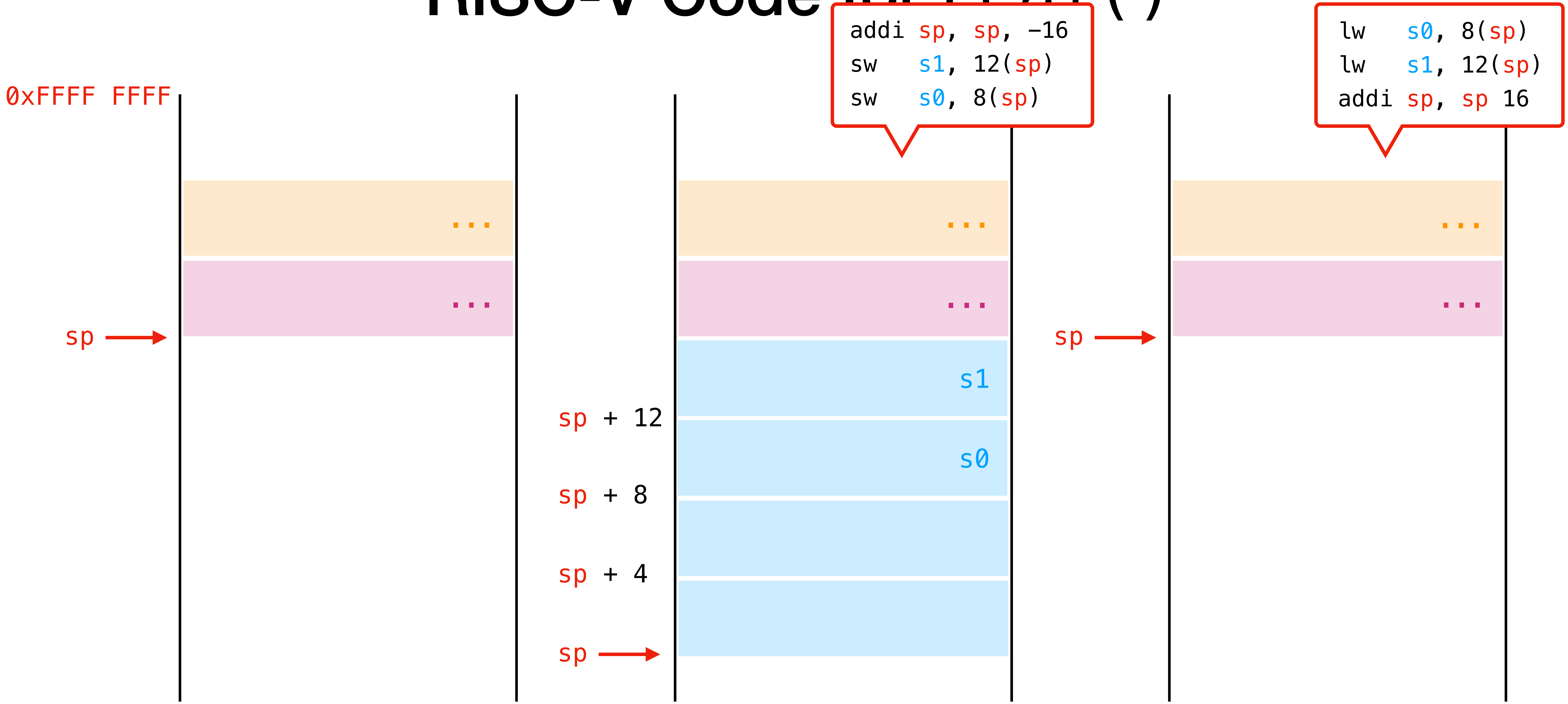
RISC-V Code for `Leaf()`

```
# int Leaf(int g, int h, int i, int j)
# a0 = g, a1 = h, a2 = i, a3 = j
# return in a0

Leaf:
    addi sp, sp, -16 # stack allocates 16 bytes (for s0 and s1)
    sw   s1, 12(sp)  # save s1 for use afterward callee-saved
    sw   s0, 8(sp)   # save s0 for use afterward callee-saved
    add  s0, a0, a1, # f  = g + h
    add  s1, a2, a3, # s1 = i + j
    sub  a0, s0, s1, # return value (g + h) – (i + j)

    lw   s0, 8(sp)   # restore register s0 for caller
    lw   s1, 12(sp)  # restore register s1 for caller
    addi sp, sp 16   # stack deallocates 16 bytes
    jr   ra          # jump back to calling routine
```

I'm **enforcing** the 16-byte alignment

Can find this in lecture 13 (assuming 32-bit architecture)

RISC-V Code for `Leaf()`



Before the call to **`Leaf()`**

During the call to **`Leaf()`**

After the call to **`Leaf()`**

Can find this in lecture 13 (assuming 32-bit architecture)

RISC-V Code for `Leaf()`

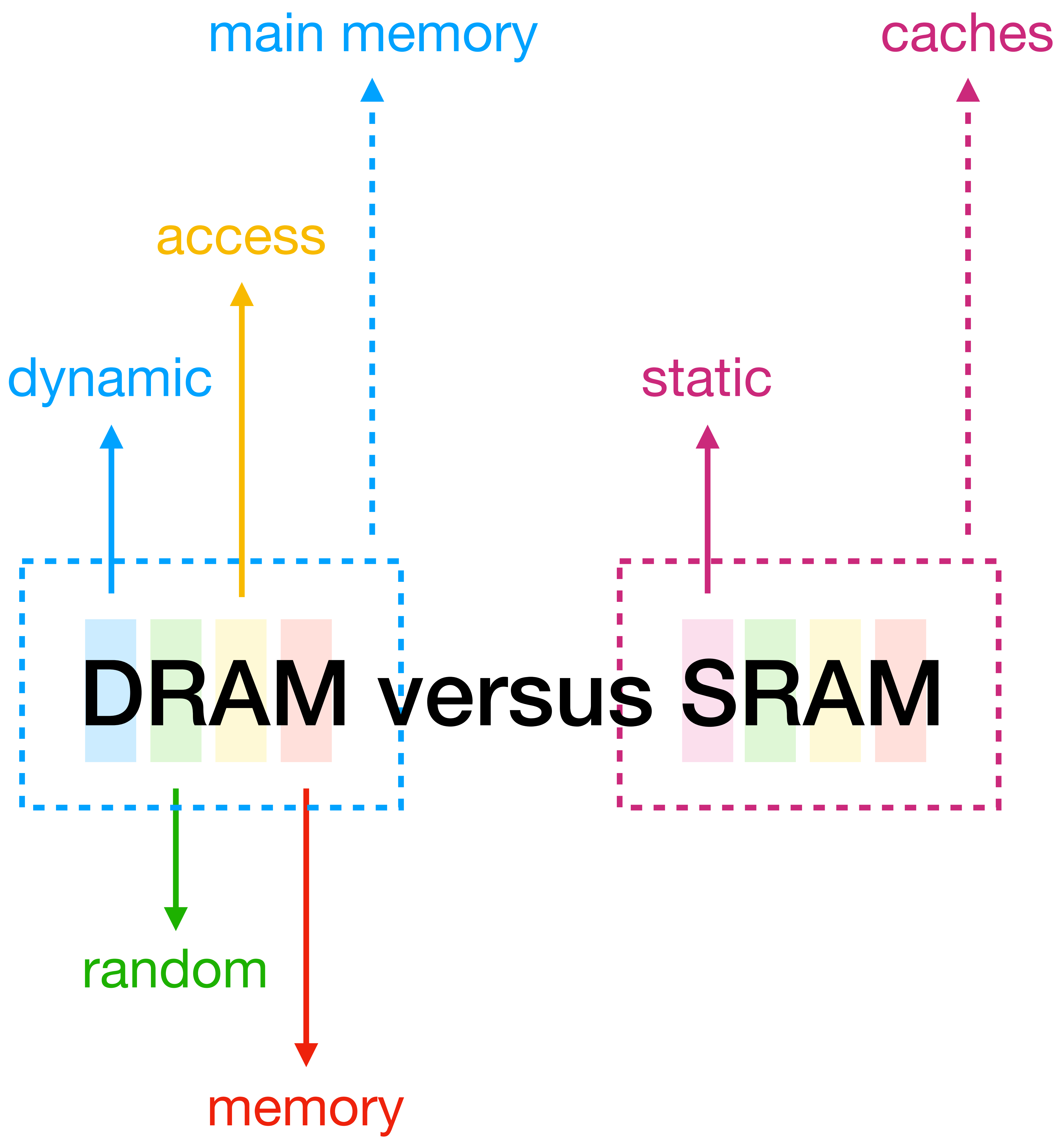
```
# int Leaf(int g, int h, int i, int j)
# a0 = g, a1 = h, a2 = i, a3 = j
# return in a0

Leaf:
    add   t0, a0, a1    # t0 = g + h
    add   t1, a2, a3    # t1 = i + j
    sub   a0, t0, t1    # a0 = (g + h) − (i + j)
    jr    ra            # return to caller
```

caller-saved

caller-saved

Do **not** need to store the current value of `t0` and `t1` on the stack—the caller was responsible for saving the content, if needed

Cache review


main memory
caches
access
dynamic
static
DRAM versus SRAM
random
memory

DRAM versus SRAM

Feature	DRAM (Dynamic RAM)	SRAM (Static RAM)
Data storage	Uses **1 transistor + 1 capacitor** per bit	Uses a **latch with 6 transistors** per bit
Refresh needed?	**Yes**, must be refreshed periodically	**No** refresh required
Speed	**Slower** than SRAM	**Fast**
Cost	**Cheaper** (fewer transistors)	**Expensive** (more transistors)
Density	**High density** (more bits per chip)	**Low density** (takes more space)
Power usage	**Lower idle power**, but refresh consumes energy	**Higher idle power**, but efficient during access
Volatility	**Volatile** (data lost without power)	**Volatile** (data lost without power)
Typical use	**Main memory**	**CPU caches (L1/L2/L3)**
Access time	~10–100 ns (nanoseconds)	~1–2 ns (nanoseconds)
Cost per bit	**Low**	**High**

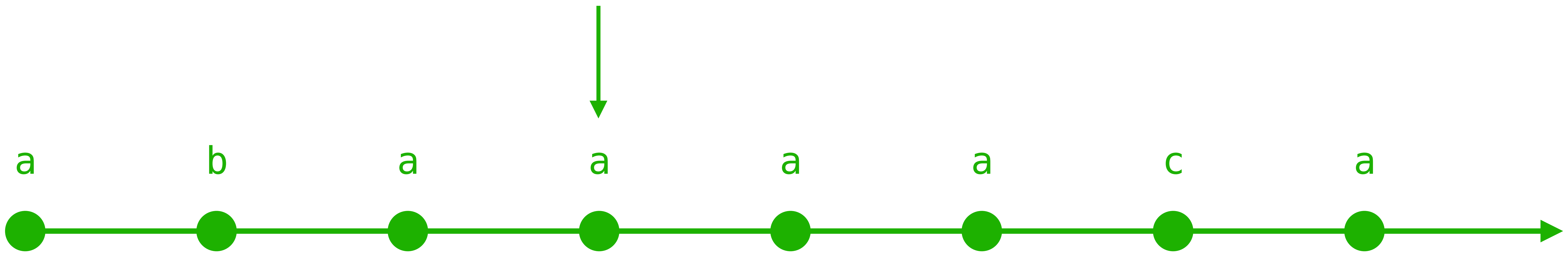
Locality Locality Locality

2 main categories!

If you ask for something, you're likely to ask for:

- The same piece of data again soon — **temporal locality**

The same data **a** is *reused* (short time apart)



program timeline accessing **variables**

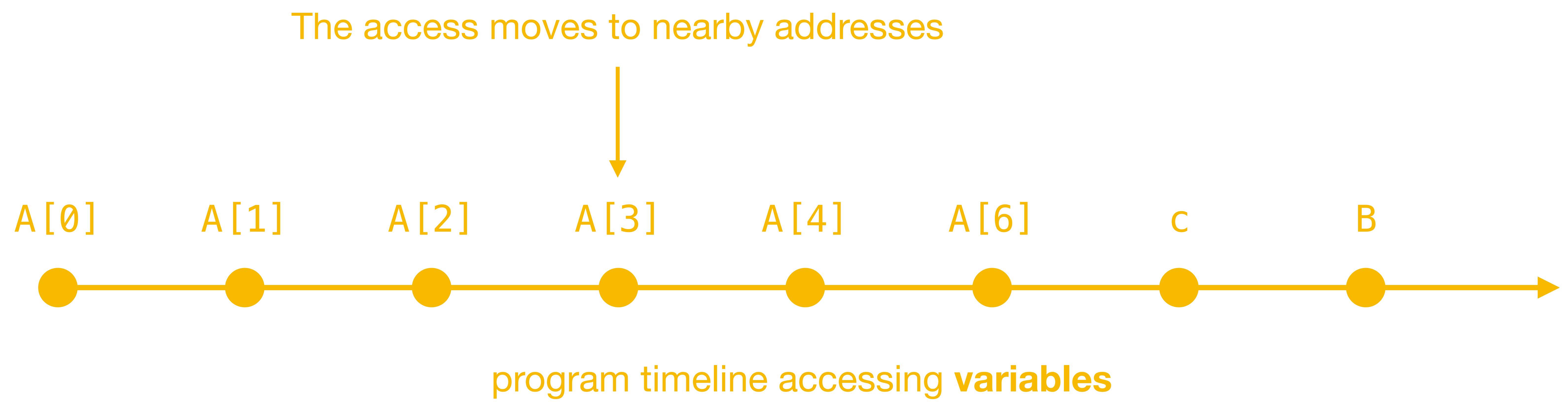
Locality Locality Locality

2 main categories!

If you ask for something, you're likely to ask for:

- Piece of data that's near the previous piece of data

spatial locality



On to set associative cache design (middleground)

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	0	xx	X | X
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

Set associative design:

Divide the storage into <u>sets</u> of size 2^k
The cache has 2^k <u>ways</u>:

tag | index | offset

XXXX

There are: $2^n / 2^k = 2^{n-k}$ sets

Each set has one index; do the “associative” thing within each set

Other designs are special cases:

- Direct mapped: k = 0
- Fully associative: k = n

The access algorithm

CACHE

WAY 0

index or set	V	Tag	Data
0	0	xx	X | X
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

1. Check every cache line **in the set in parallel**

~~1~~ Split the address between **tag** `t` and **index** `i`

~~2~~ Check the entry `i`

3. Is it valid? If no, cache miss!

4. Is it the tag `t`? If no, cache miss!

4. Otherwise, cache hit!

tag | index | offset

XXXX

2. If not valid OR not tag `t`, try other indeces **in the set**

3. If no match (cache miss), **evict** some line **in the set** and load new block

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	0	xx	X | X
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

```
load 1100
load 1101
load 0100
load 1100
```

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101`

`load 0100`

`load 1100`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100`

`load 1100`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
0	xx	X | X
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100`

`load 1100`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
1	01	E | F
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache miss! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

	WAY 0			WAY 1		
index or set	V	Tag	Data	V	Tag	Data
0	1	11	M | N	1	01	E | F
1	0	xx	X | X	0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache <u>miss</u>! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache <u>hit</u>! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache <u>miss</u>! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100` Cache <u>hit</u>! **M** from `1101` is **still** in the cache!

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
1	01	E | F
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache miss! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100` Cache hit! **M** from `1101` is **still** in the cache!

`load 1010`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	1	10	K | L

WAY 1

V	Tag	Data
1	01	E | F
0	xx	X | X

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache miss! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100` Cache hit! **M** from `1101` is **still** in the cache!

`load 1010` Cache miss! **K** from `1010` is now in the cache together with **L** from `1011`

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
1	01	E | F
0	xx	X | X

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache miss! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100` Cache hit! **M** from `1101` is **still** in the cache!

`load 1000`

tag | index | offset

XXXX

Set Associative Cache

CACHE

WAY 0

index or set	V	Tag	Data
0	1	11	M | N
1	0	xx	X | X

WAY 1

V	Tag	Data
1	10	I | J
0	xx	X | X

MEMORY

addr	Data
1111	P
1110	O
1101	N
1100	M
1011	L
1010	K
1001	J
1000	I
0111	H
0110	G
0101	F
0100	E
0011	D
0010	C
0001	B
0000	A

1. Check every cache line **in the set in parallel**
2. If no valid bit or no tag `t`, then evict **from the set** and load new line

`load 1100` Cache miss! **M** from `1100` is now in the cache together with **N** from `1101`

`load 1101` Cache hit! **N** from `1101` is already in the cache! I moved it together with **M**

`load 0100` Cache miss! **E** from `0100` is now in the cache together with **F** from `0101`

`load 1100` Cache hit! **M** from `1101` is **still** in the cache!

`load 1000` Cache miss! I need to **evict** one line (LRU); then load **I** from `1000` in the cache together with **J** from `1001`

Cache performance

The average access time t_{avg}:

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

$t_{avg} = 4 + 5\% * 100$

$t_{avg} =$ **9 cycles**

The average access time t_{avg}:

$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$

$t_{avg} = 1\ ns + 5\% * 50\ ns$

$t_{avg} =$ **3.5 ns**

Three types of <u>cache misses</u> (3 Cs):

- **Cold** or **Compulsory:** first access ever to a block
- **Capacity:** the cache is too small
- **Conflict:** mapping collision (esp. direct mapped), the associativity is too low

Cache performance

Three types of <u>cache misses</u> (3 Cs):

- **Cold** or **Compulsory:** first access ever to a block
- **Capacity:** the cache is too small
- **Conflict:** mapping collision (esp. direct mapped), the associativity is too low

If I want to achieve a **lower miss rate**, how can I change my cache?

- **I can build a larger cache**
- **I can increase associativity**

Cache performance

Three types of <u>cache misses</u> (3 Cs):

- **Cold** or **Compulsory:** first access ever to a block
- **Capacity:** the cache is too small
- **Conflict:** mapping collision (esp. direct mapped), the associativity is too low

If I want to achieve a **lower miss rate**, how can I change my cache?

- **I can build a larger cache**
- **I can increase associativity**

If I want to achieve a **lower miss rate**, can I change my code? I can increase **locality**!

Cache exercise for practice

- Byte-addressable memory
- 2-way set associative cache
- Cache size: 32 bytes
- Block size: 8 bytes; 32 / 8 = 4 cache lines in total (2 per set)
- LRU replacement policy and write-allocate and write-back policy
- Cache starts empty

For each instruction, determine: **hit**, **cold**, **conflict**, or **capacity miss**:

```
lw x1,  0(x0)
sw x2, 16(x0)
lw x3,  8(x0)
sw x4,  0(x0)
lw x5, 24(x0)
```

Review of virtual memory and swap space

Page Table Overhead

- How large is Page Table?
- The virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
    - # entries in PageTable?
    - number of pages = virtual memory / page size
    - number of pages = $2^{32} / 2^{12} = 2^{20}$ = 1,048, 576 pages ~ 1 million pages

Page Table Overhead

- How large is Page Table?
- The virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
    - # entries in PageTable?
    - size of PageTable? (in bytes)
    - A page table entry (PTE) usually stores the **PFN** plus some metadata (valid bit, protection bits, etc.)
    - Typically, we use 4 bytes (32 bits) per PTE (common for 32-bit physical addresses)
    - Page Table size = $2^{20} \times 4$ bytes = 4×2^{20} bytes = 4 MB

Page Table Overhead

- How large is Page Table?
- The virtual address space (for each process):
  - Given: total virtual memory: 2^{32} bytes = 4GB
  - Given: page size: 2^{12} bytes = 4KB
    - # entries in PageTable? ~ 1 million pages
    - size of PageTable? (in bytes) ~ 4 MB
- The physical address space:
  - Total physical memory: 2^{29} bytes = 512MB
    - Overhead for 10 processes?

- number of physical frames = physical memory / page size = 2^{29} / 2^{12} = 2^{17} = 131, 072 frames
- pages per process = 2^{32} / 2^{12} = 2^{20} pages
- page table size per process = 4 MB
- page table **overhead** size= 4 MB x 10 = 40 MB
- fraction overhead = 40 / 512 MB = **7.8%**

Paging

- But what if process requirements > physical memory?
  - Then, *virtual starts earning its name*

- E.g., a process needs **1 GB**, but physical memory is only **512 MB**
- Can’t fit the entire virtual address space in DRAM at once
- **Paging** allows this to work by mapping only the pages that are actively used

Paging

- But what if process requirements > physical memory?
  - Then, *virtual starts earning its name*

- The main memory acts as a cache for secondary storage (disk):
  - **Swap** memory pages out to disk when not in use
  - **Page** them back in when needed
  - If a process accesses a page not in memory, a **page fault** occurs

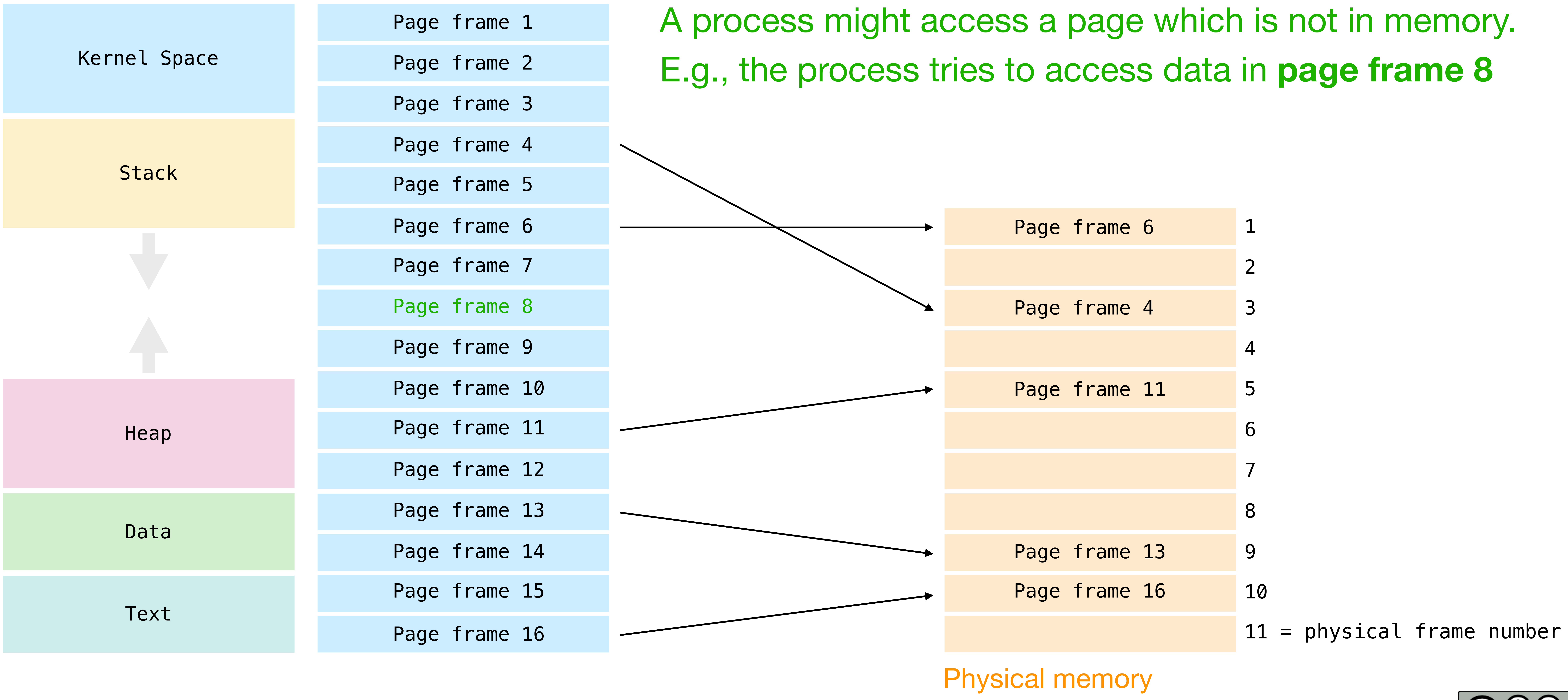
Paging

- But what if process requirements > physical memory?
    - Then, *virtual starts earning its name*
- The main memory acts as a cache for secondary storage (disk):
    - **Swap** memory pages out to disk when not in use
    - **Page** them back in when needed
    - If a process accesses a page not in memory, a **page fault** occurs
- Courtesy of Temporal & Spatial Locality (again!)

Picture Memory As ...

A process might access a page which is not in memory.

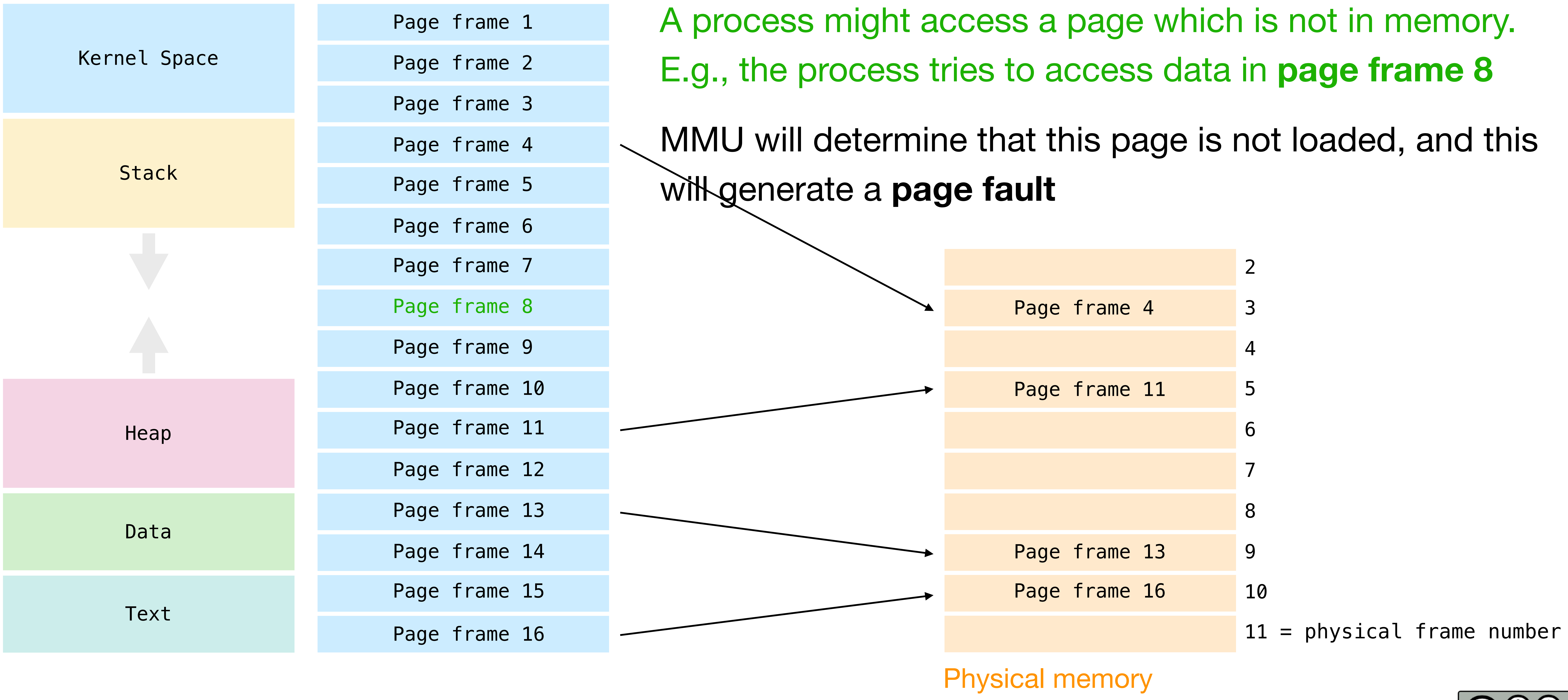
E.g., the process tries to access data in **page frame 8**



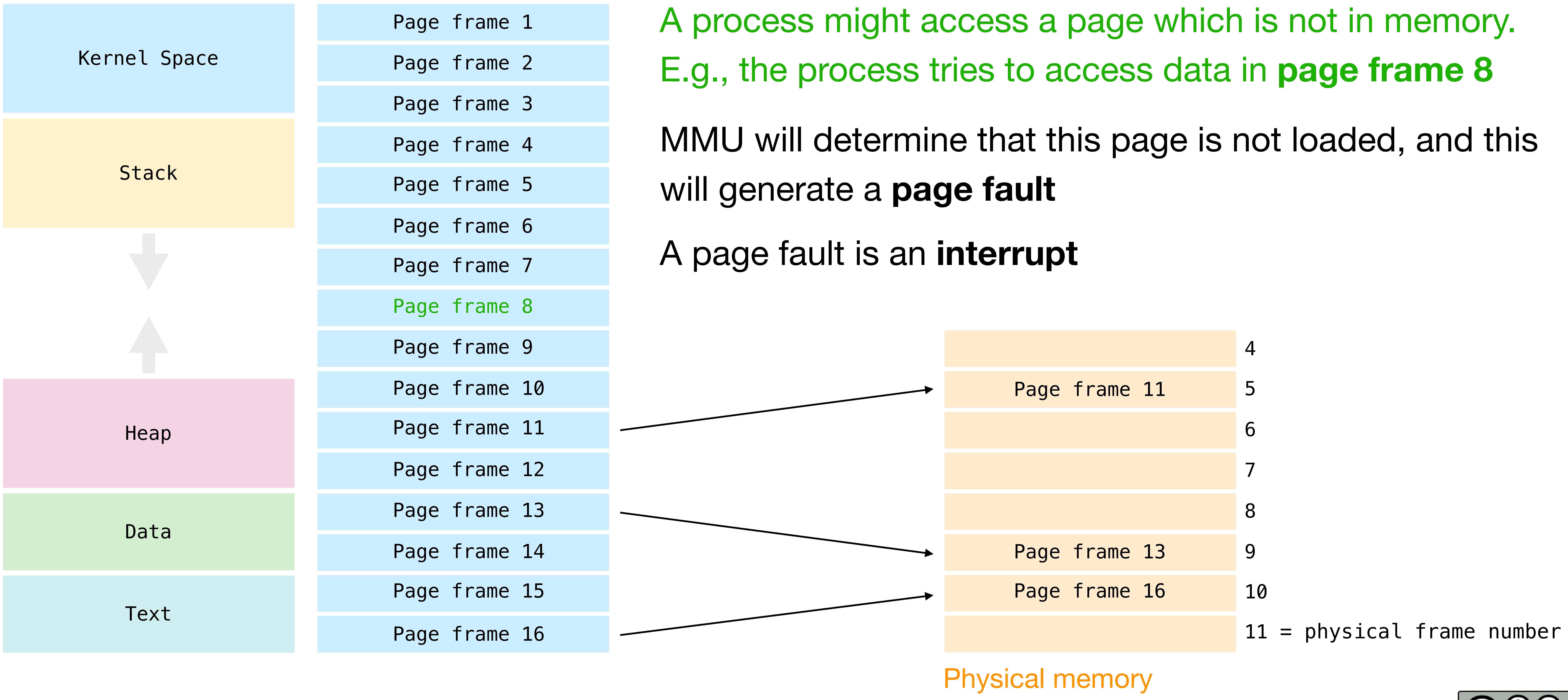
Picture Memory As ...

A process might access a page which is not in memory.
E.g., the process tries to access data in **page frame 8**

MMU will determine that this page is not loaded, and this will generate a **page fault**



Picture Memory As ...



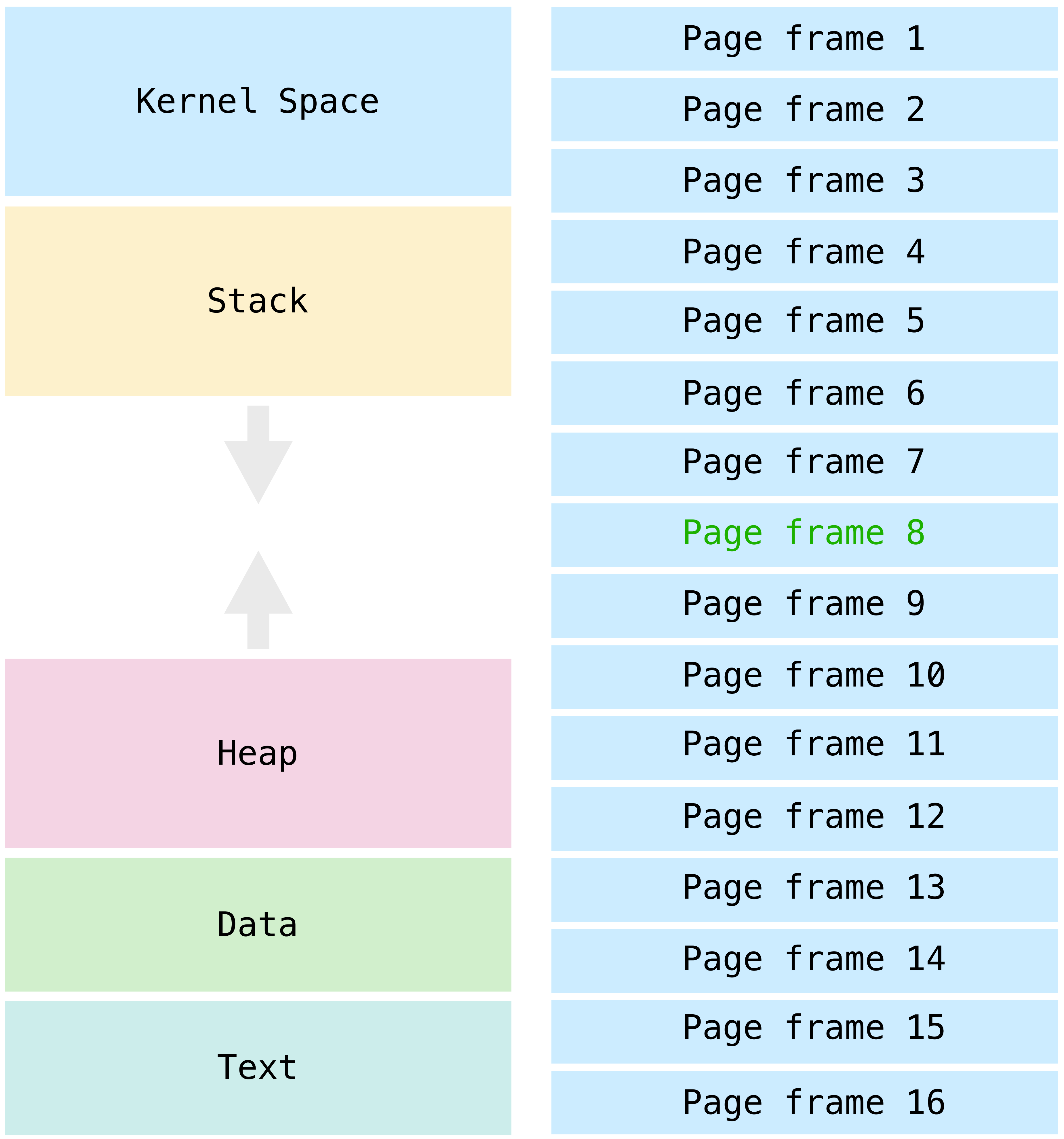
A process might access a page which is not in memory.

E.g., the process tries to access data in **page frame 8**

MMU will determine that this page is not loaded, and this will generate a **page fault**

A page fault is an **interrupt**

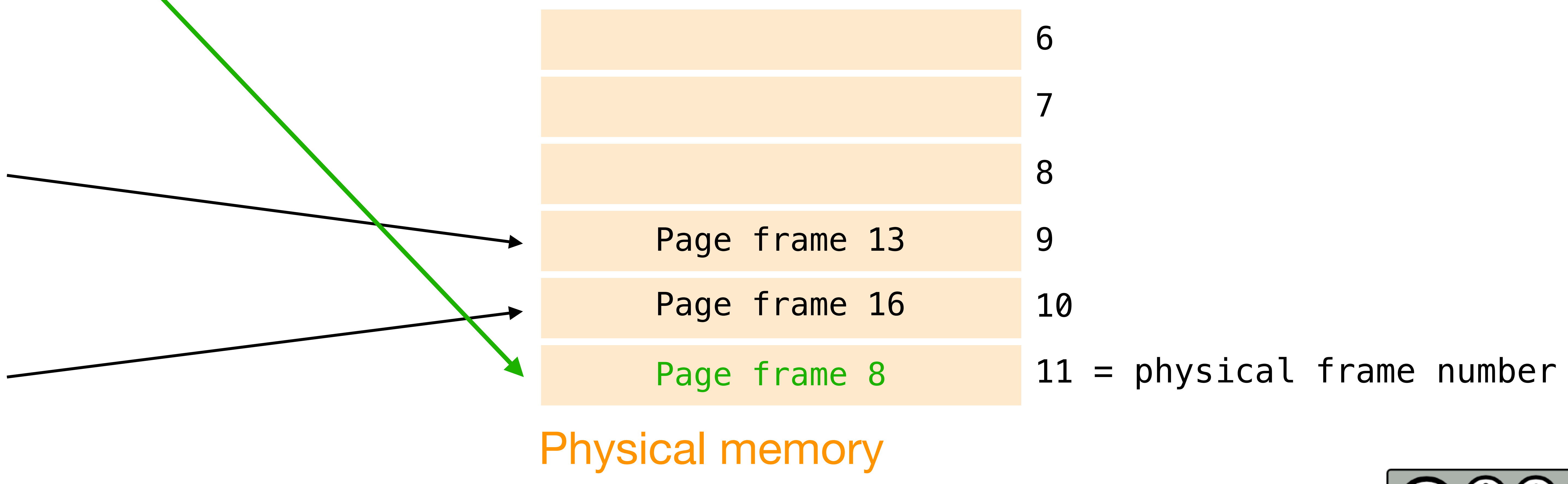
Picture Memory As …



A process might access a page which is not in memory. E.g., the process tries to access data in **page frame 8**

MMU will determine that this page is not loaded, and this will generate a **page fault**

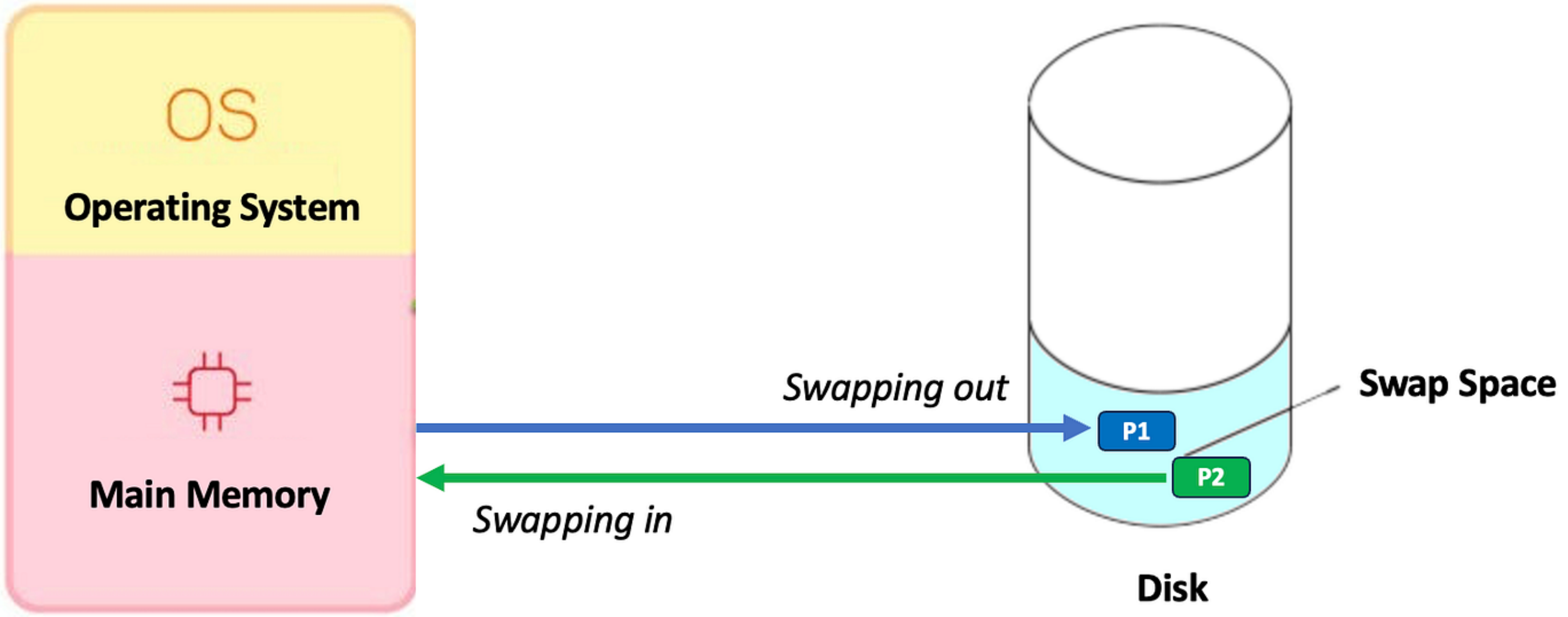
The **OS handler** for a page fault locates the needed page frame on the disk, copies it to a page in memory, and updates the page table



Swap Space

A backup area where the OS can temporarily store parts of a process's memory that don't fit in DRAM

- The OS keeps **active pages** in DRAM and moves **inactive pages** to swap when RAM is full
- This way, the total "usable" memory = **DRAM + swap**

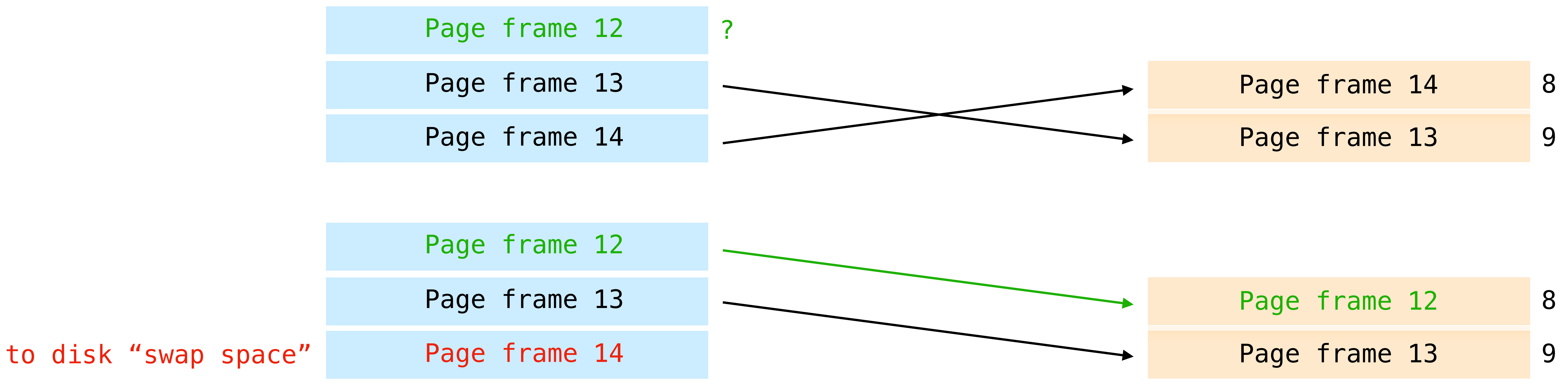


https://medium.com/@ahmedmansouri/the-diminishing-role-of-swap-space-in-cloud-environments-ab2e413c4013

Swap Space

If DRAM is full and a process needs a new page:

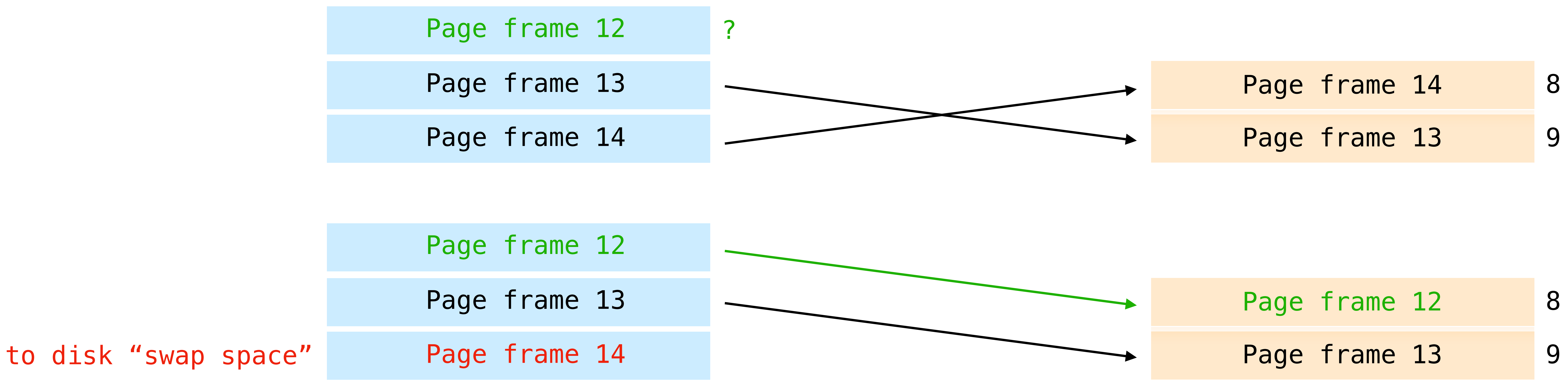
- The OS picks a page in DRAM to evict (LRU, etc.)
- If the page to be evicted has been modified, it's written to **swap**
- Then, the new page is loaded into that freed DRAM frame



Swap Space

Then, the OS updates the **page table** entry for the evicted page:

- It marks the page as not present in memory
- If stores the swap location (disk block) where the page lives



Swap Space

If DRAM is 8 KB and each page is 4 KB then only 2 pages can fit in DRAM at any given moment. If a process uses 4 pages A, B, C, and D, then:

Time	DRAM stores	Swap (disk) stores	The action
Process P begins	A, B	—	Load A and B
P accesses C	C, B	A	Page fault → Evict A, load C in
P accesses D	D, B	A, C	Page fault → Evict A, load D in
P accesses A	A, B	D, C	Page fault → Swap A in, D out

VM exercise for practice

Consider a system with 48-bit virtual addresses, 36-bit physical addresses, and 4 KiB pages. (One KiB is 2^{10} = 1024 bytes, so 4 KiB = 2^{12} bytes.) The memory is byte-addressable.

How many **bits** are in the:

- Physical page offset (PPO)?
- Physical page number (PPN)?
- Virtual page offset (VPO)?
- Virtual page number (VPN)?

Good luck on the exam!

Remember to fill out the course evaluation when the time comes!