Virtual Memory

Drawing: Processes

- Take three minutes to draw multiple "processes"

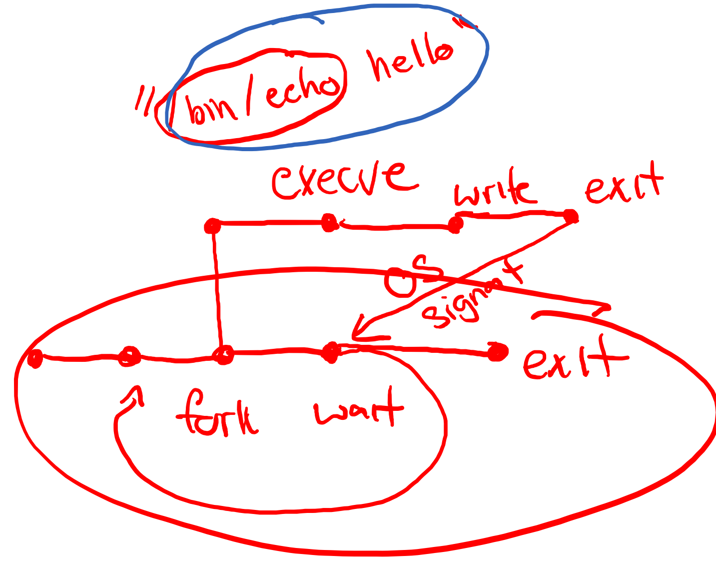
- Some reminders
  - Program vs process
  - Process control blocks
  - Context switching
  - Process life cycle
  - `fork`/`execve`/`waitpid`/`exit`
  - Process graphs
  - Scheduling (round robin; multi-level)

main

loop

fork()

wait()



/bin/sleep 10 &

run in

"background"

Virtual Memory

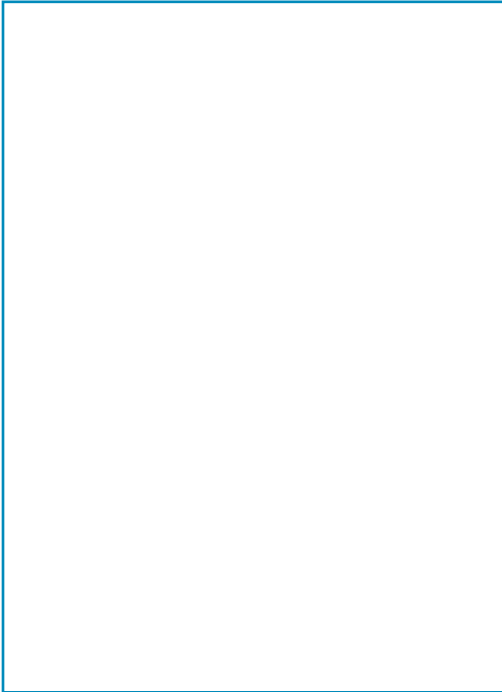
- Abstraction of physical memory
- Give appearance of a large, consistent amount of memory
- Handle loading from secondary (disk) storage

- A page table maps between a virtual address and physical address
- A memory management unit (MMU) implements the translation in hardware

- Closely connected with concurrency and multiple processes
- Managed by both the CPU and the OS

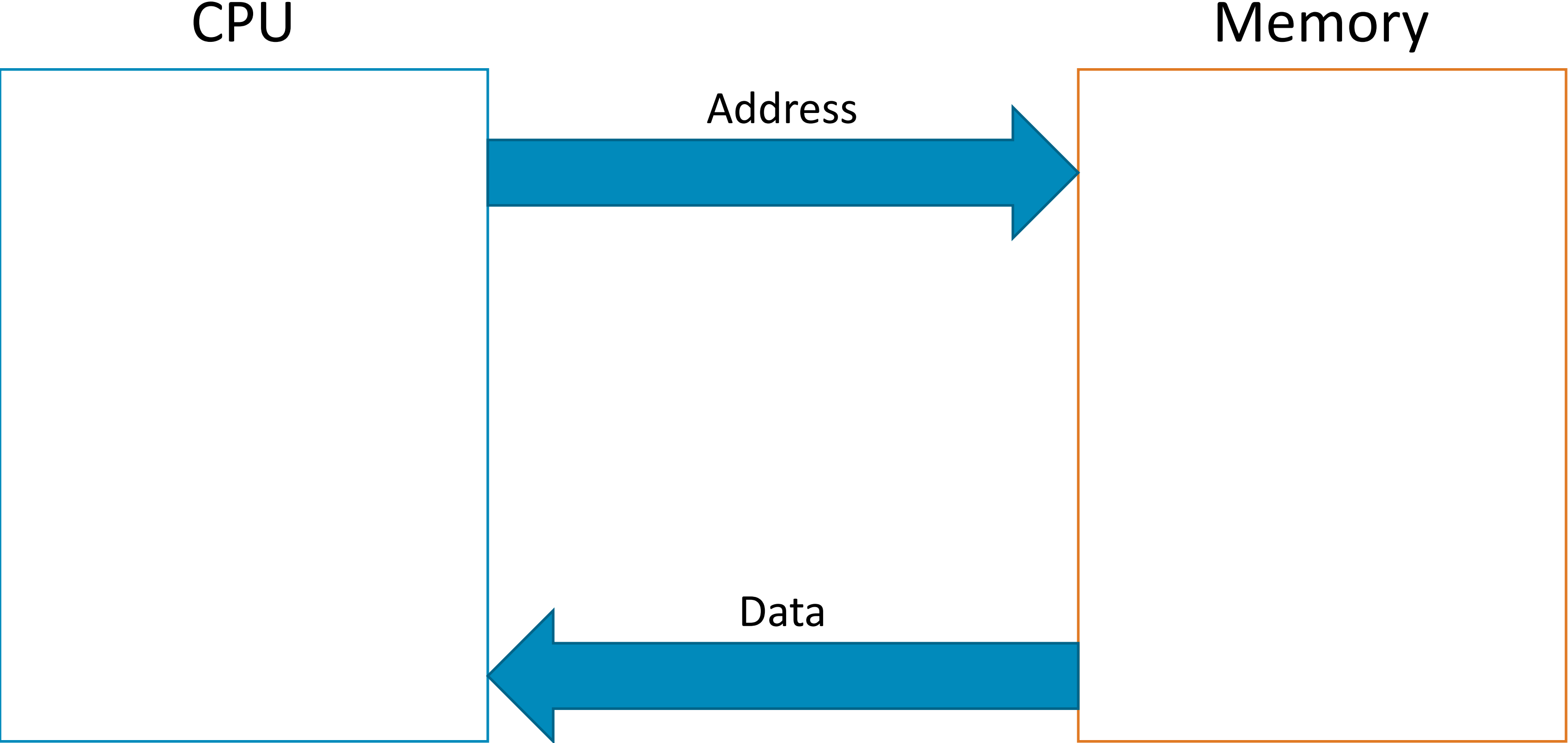
Abstractions

CPU

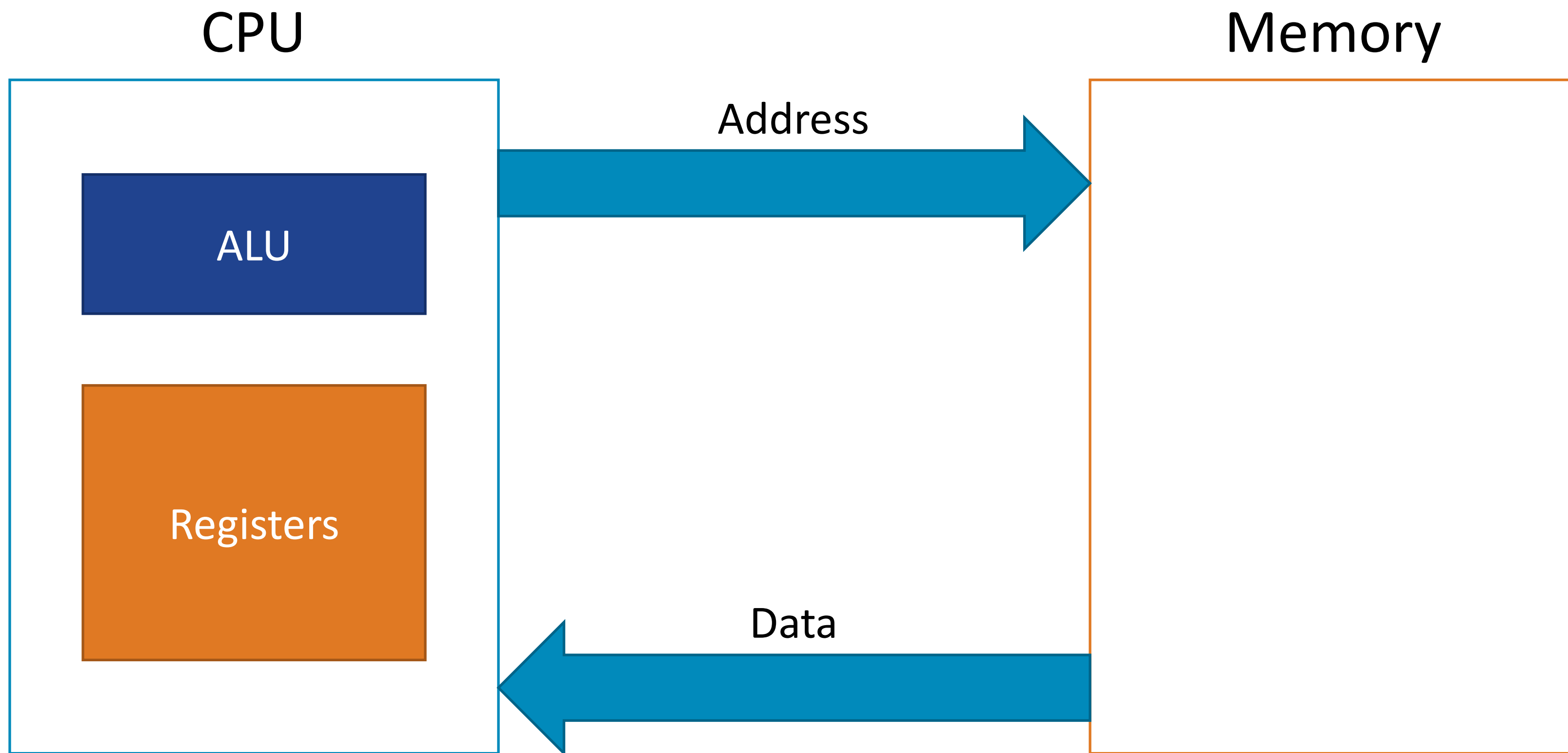


Memory

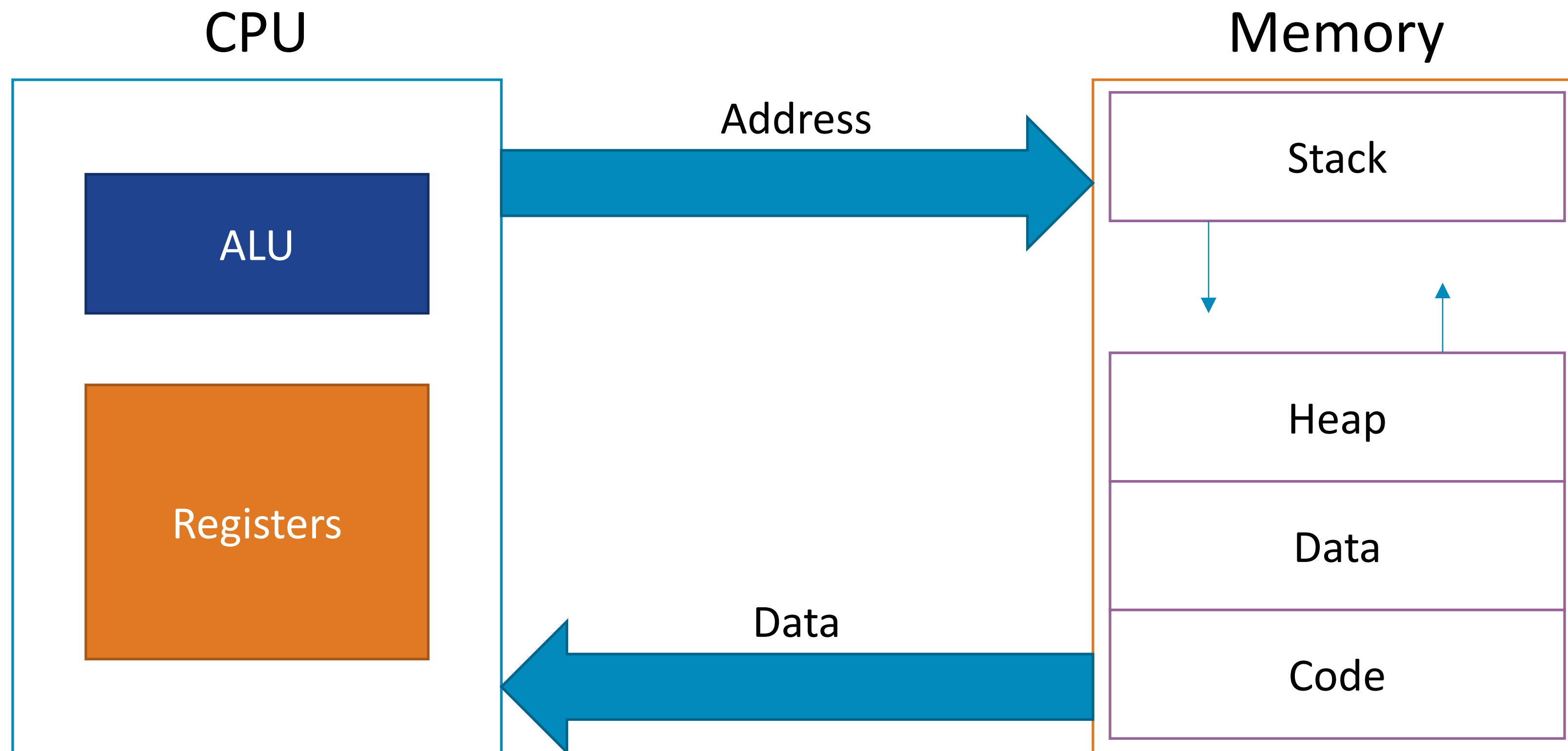
Abstractions



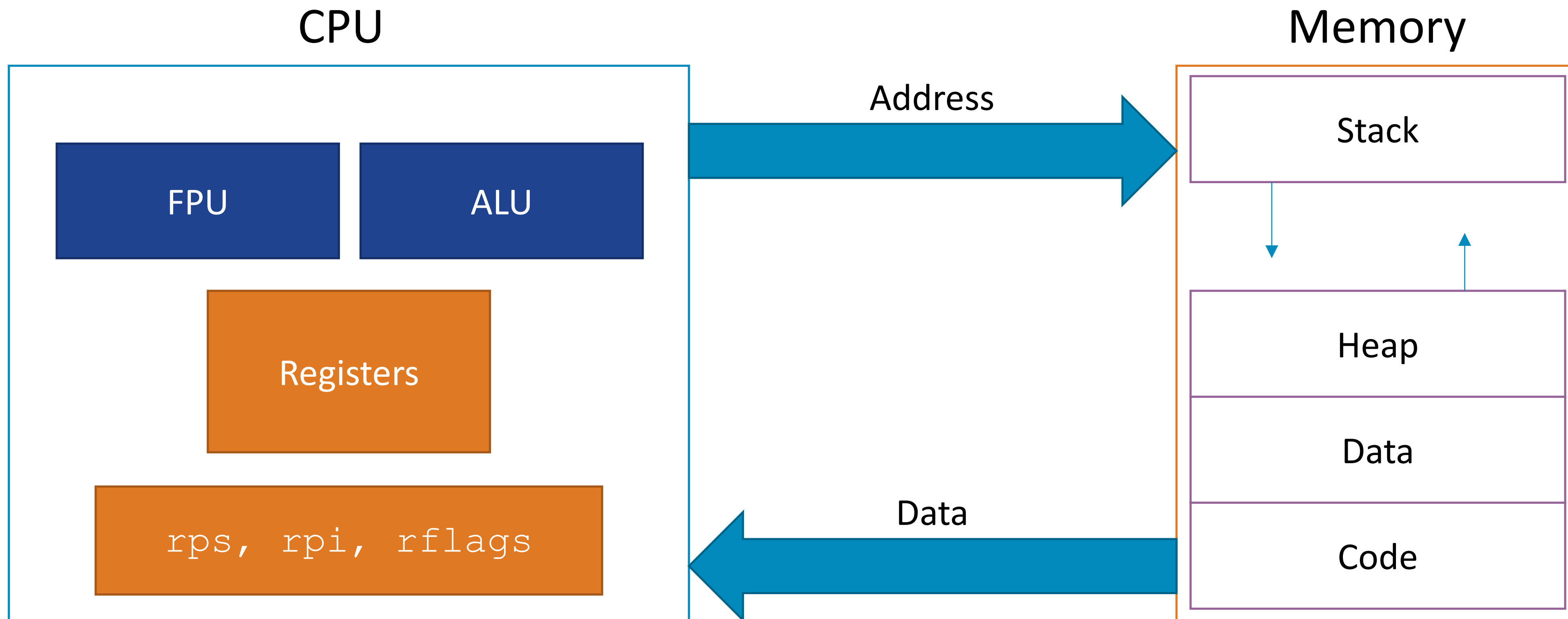
Abstractions



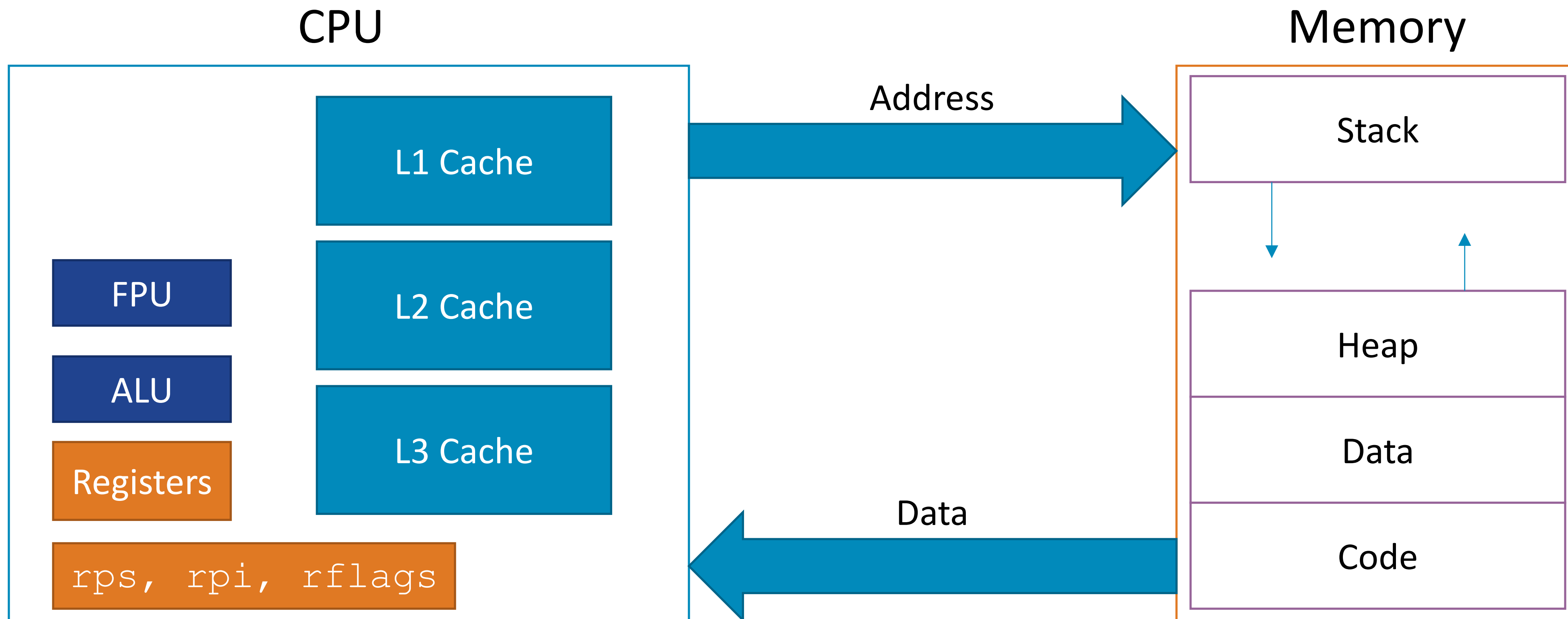
Abstractions



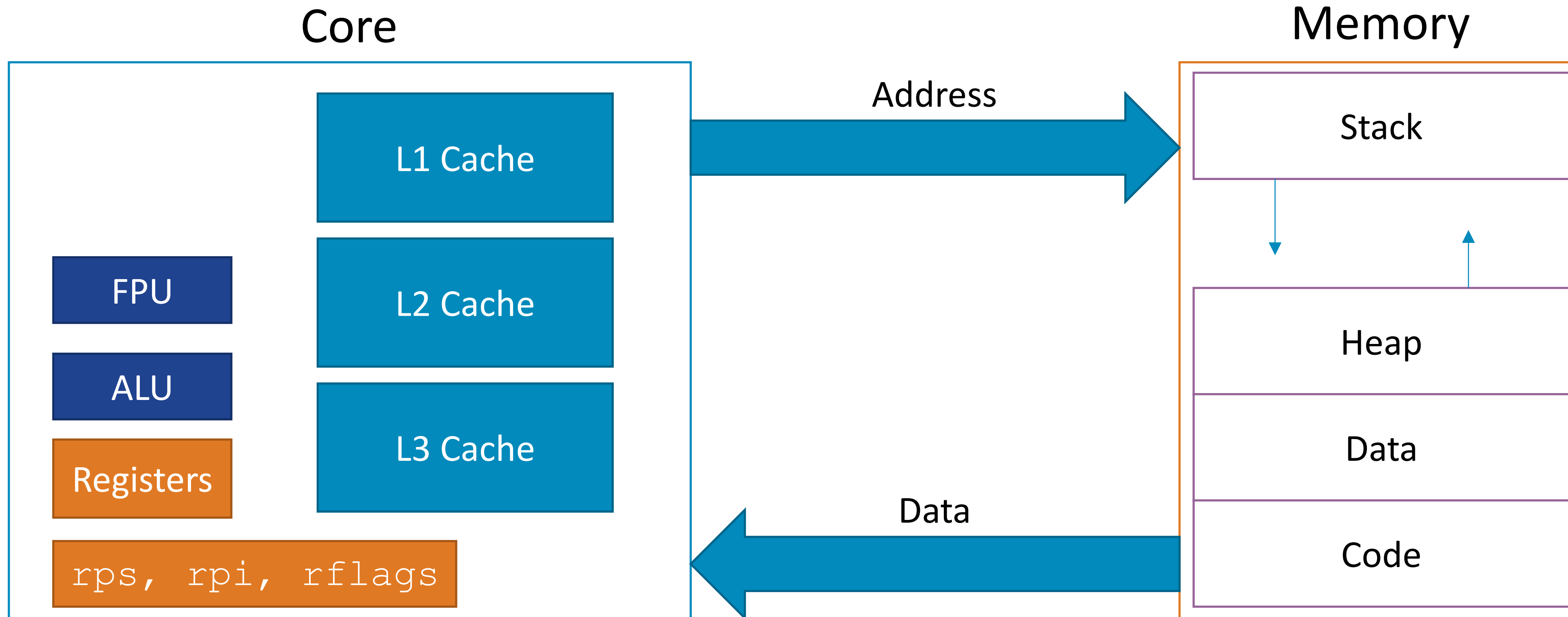
Abstractions



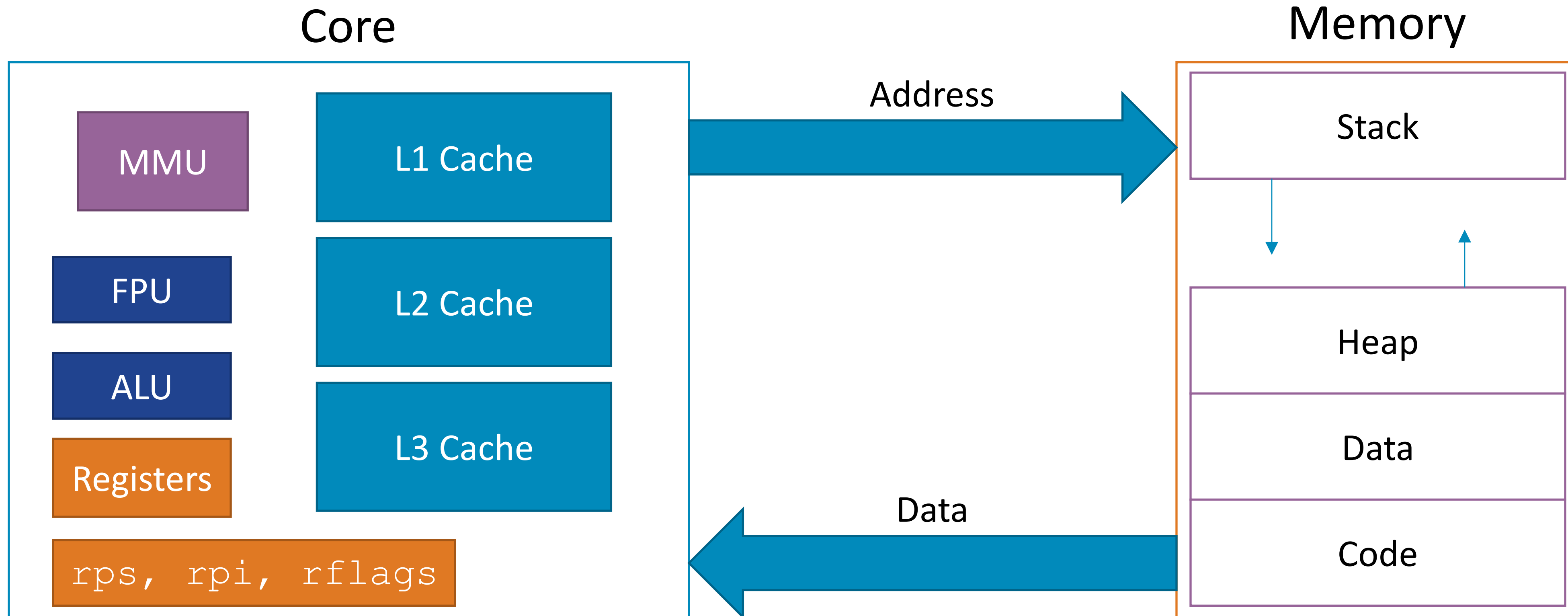
Abstractions



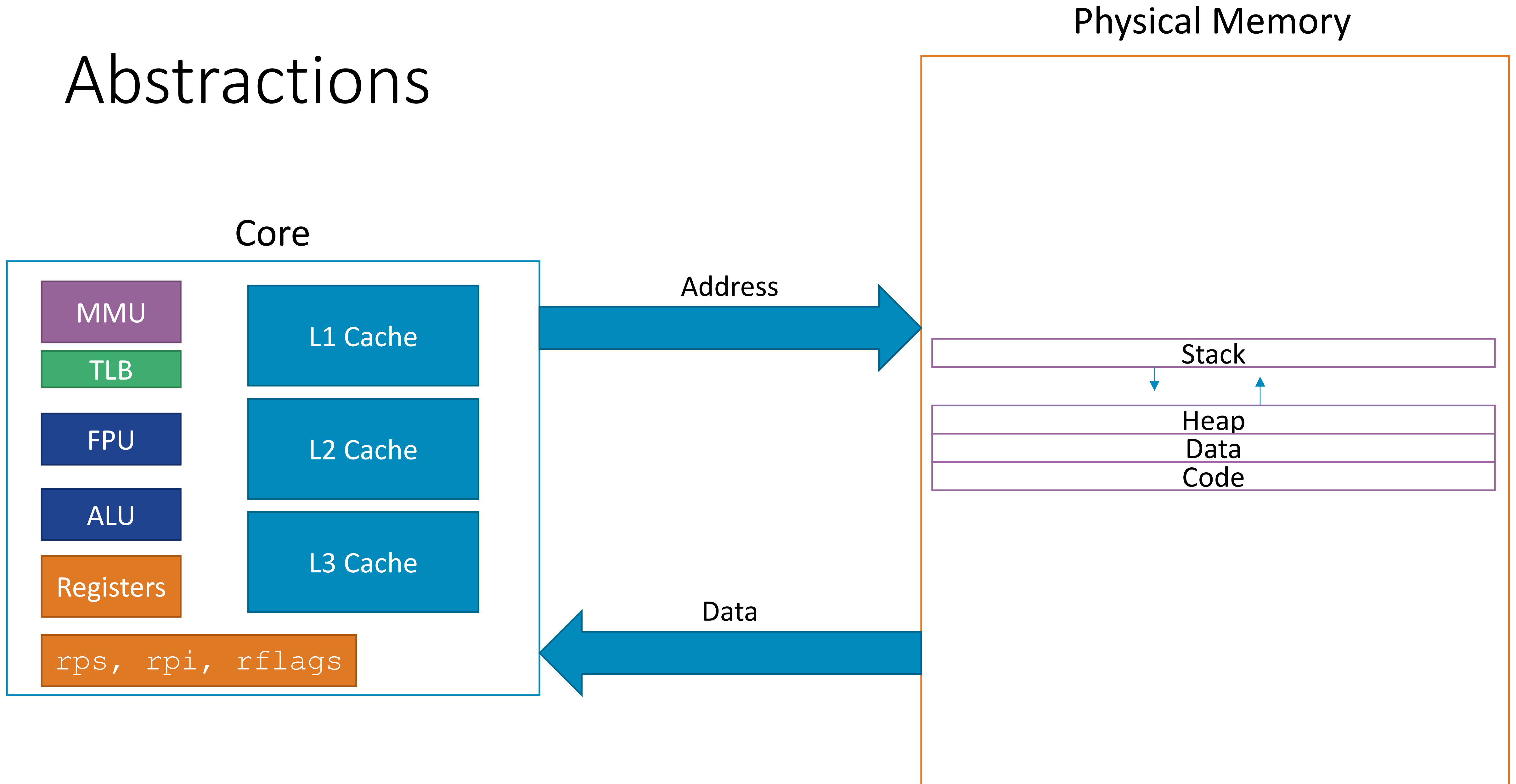
Abstractions



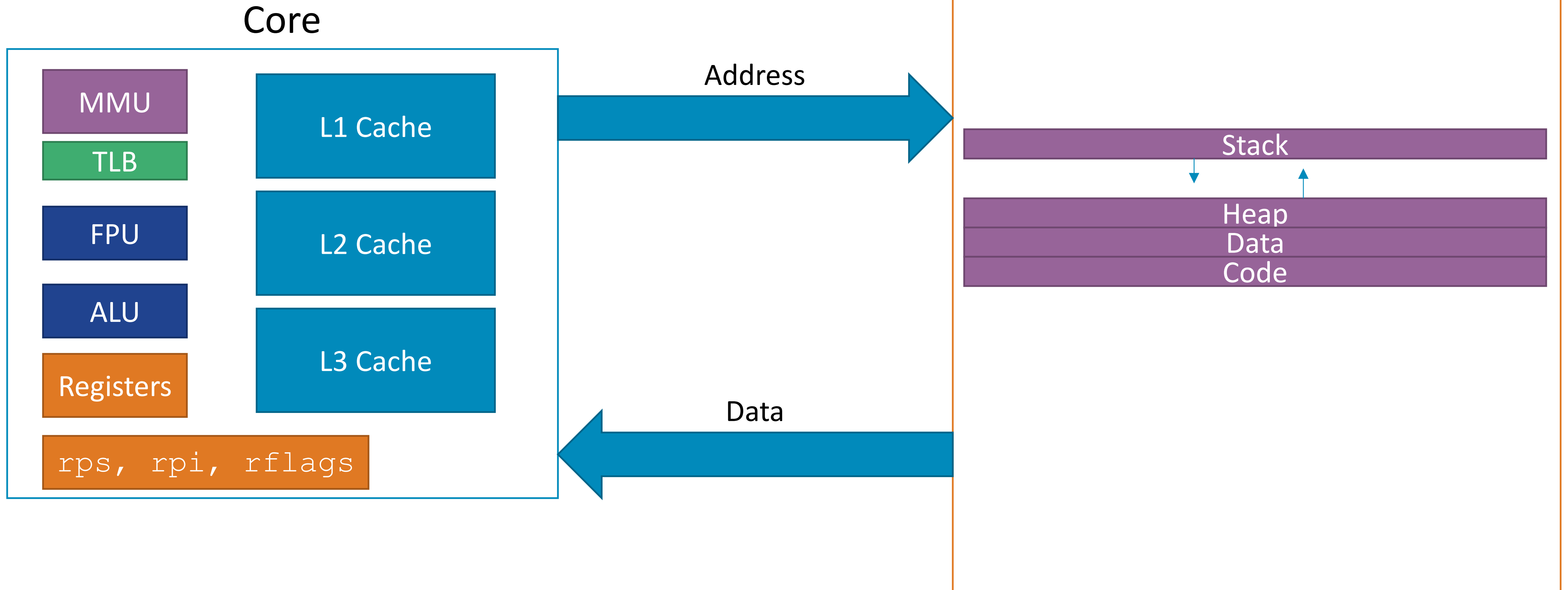
Abstractions



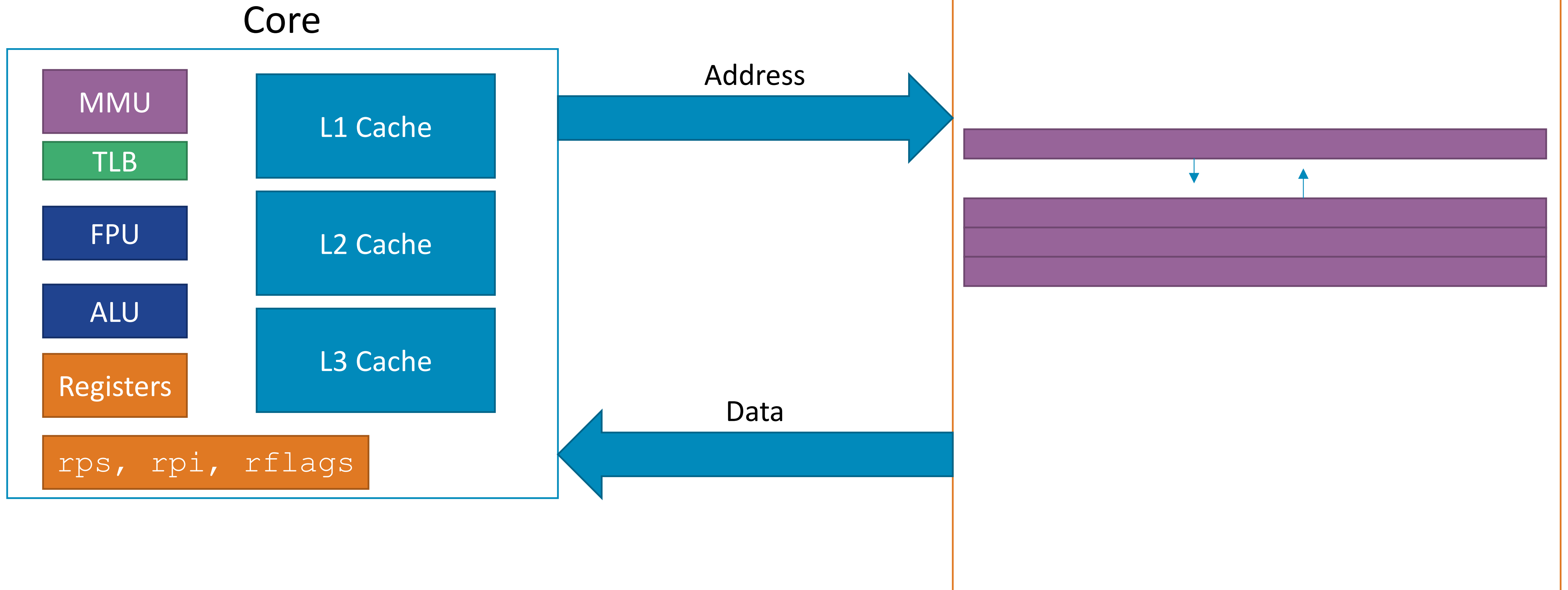
Abstractions



Abstractions

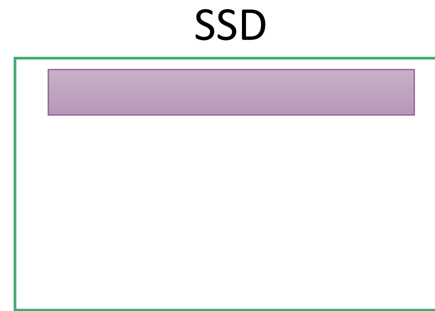


Abstractions



Abstractions

SSD



Physical Memory

Core

MMU

TLB

FPU

ALU

Registers

L1 Cache

L2 Cache

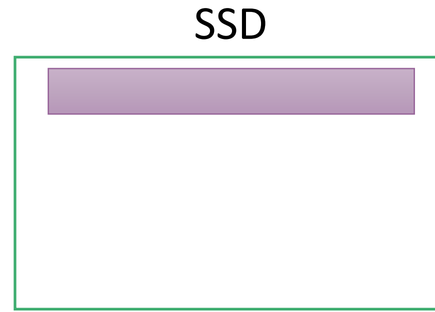
L3 Cache

`rps, rpi, rflags`

Address

Data

Abstractions



Physical Memory

Core

MMU

TLB

FPU

ALU

Registers

L1 Cache

L2 Cache

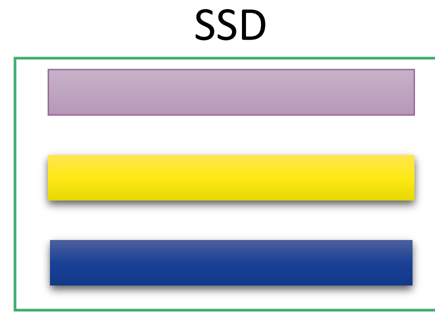
L3 Cache

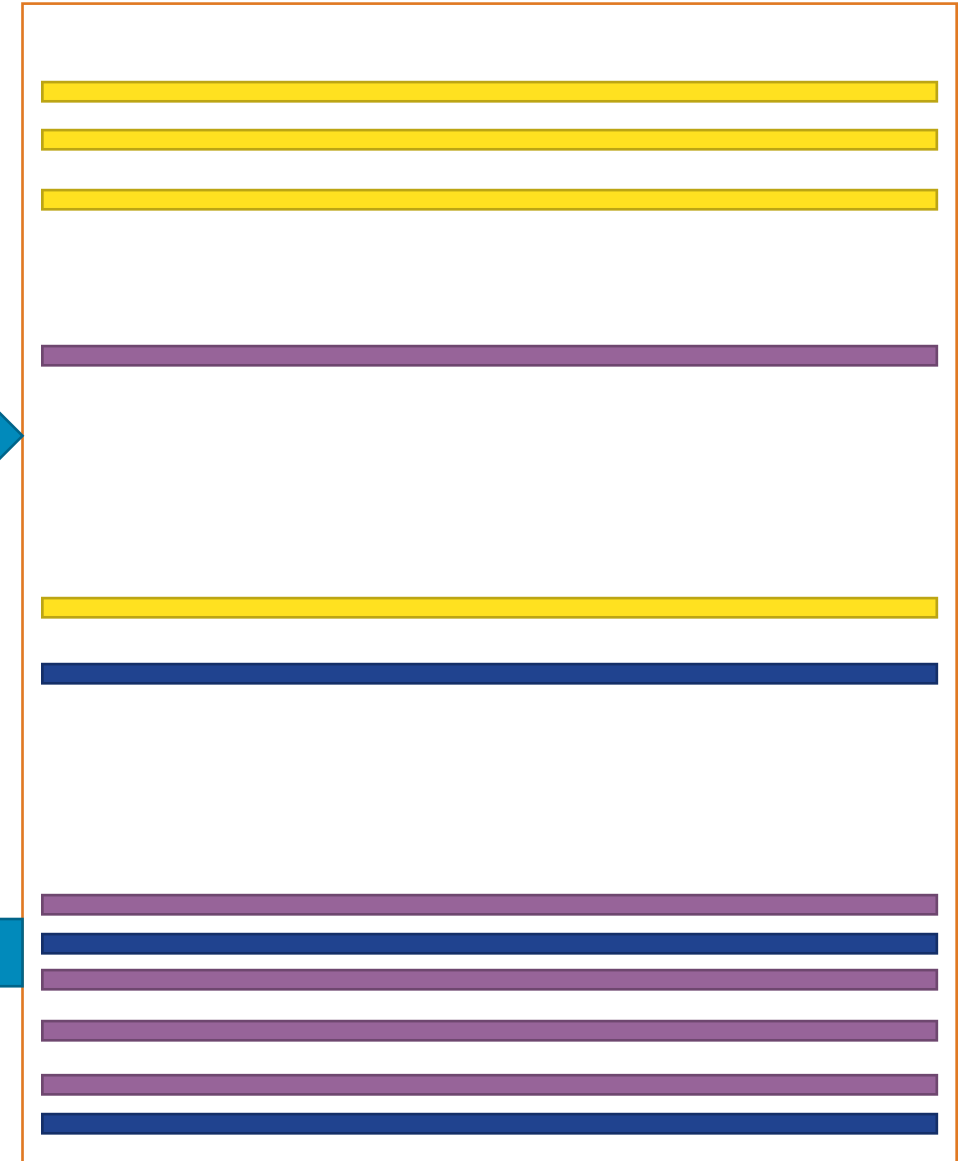
`rps, rpi, rflags`

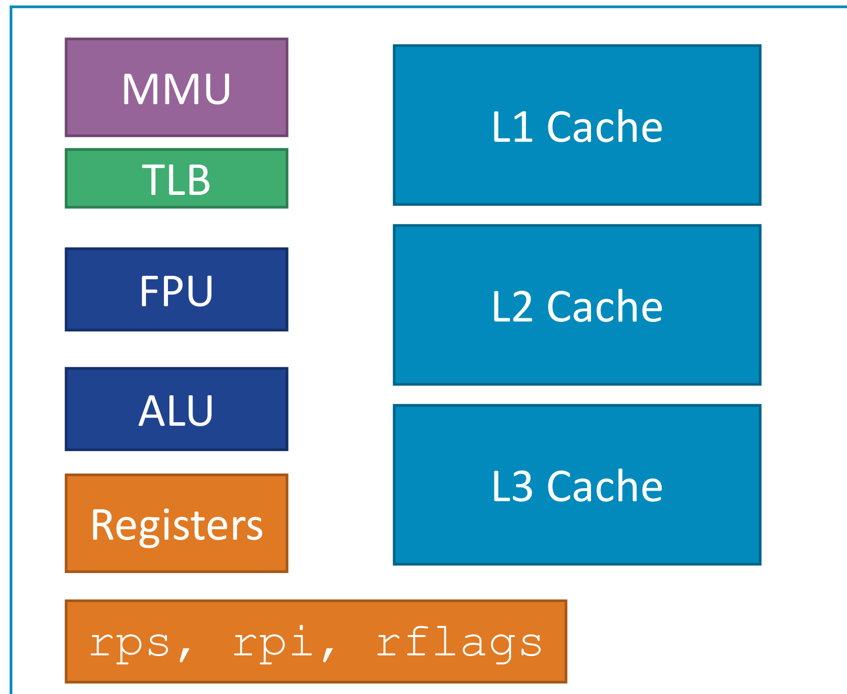
Address

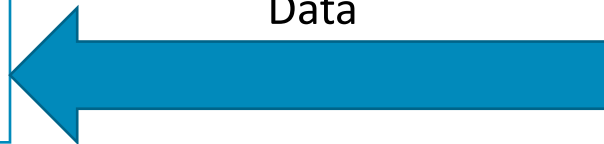
Data

Abstractions









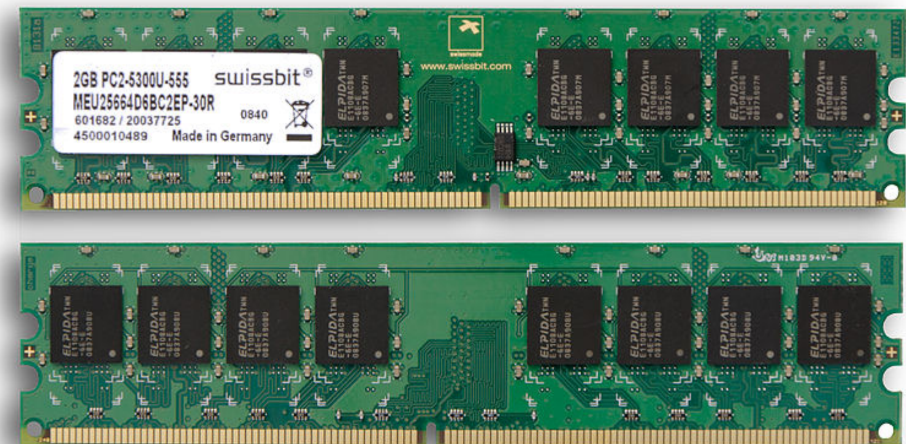
Memory

Virtual

- Not a real thing (no chip)
- Terabytes
- Supported by OS and CPU
- Every process things it owns all physical memory
- MMU on CPU translates the virtual address into physical

Physical

- RAM
- Gigabytes
- Isolation handled by OS



Courtesy of Wikipedia

Wikipedia

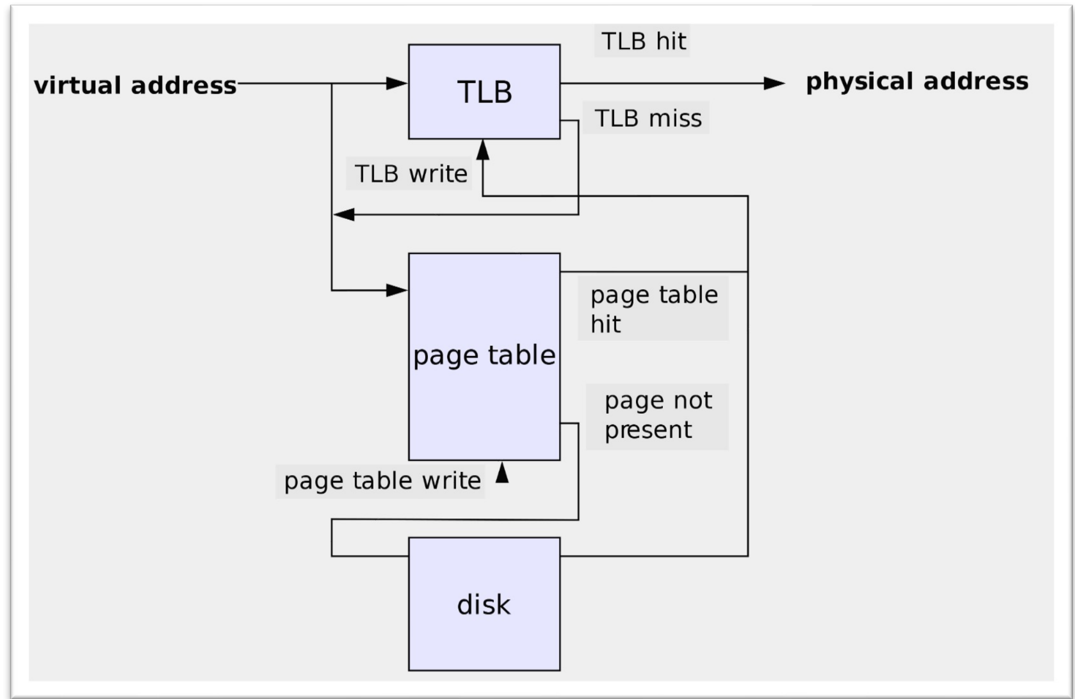
Virtual address space

Physical address space

0x00000000
0x00010000
text
0x10000000
data
stack
0x7fffffff

0x00000000
0x00ffffff

page belonging to process
page not belonging to process



https://www.kernel.org/doc/html/latest/x86/x86_64/mm.html

48-Bit Virtual Address Space

Region	Address	Size
Kernel Address Space	0xffff ffff ffff ffff – 0xffff 8000 0000 0000	128 TB
Unused Address Space	0xffff 7fff ffff ffff – 0x0000 8000 0000 0000	
User Address Space	0x0000 7fff ffff ffff – 0x0000 0000 0000 1000	128 TB
Reserved (Protection)	0x0000 0000 0000 0000	Configurable

How many of you have a system with 128 TB?

How many of you have a system with 128 TB?

- Some system support 57-bit virtual addresses, which would 64 PB…

All user-processes share this portion of the address space. All

- Stacks
- Heaps
- Data
- Code

The first "page" is often reserved to prevent null-reference errors.

```
char *str = <zero for some reason>;
str[15] = 'a';
```

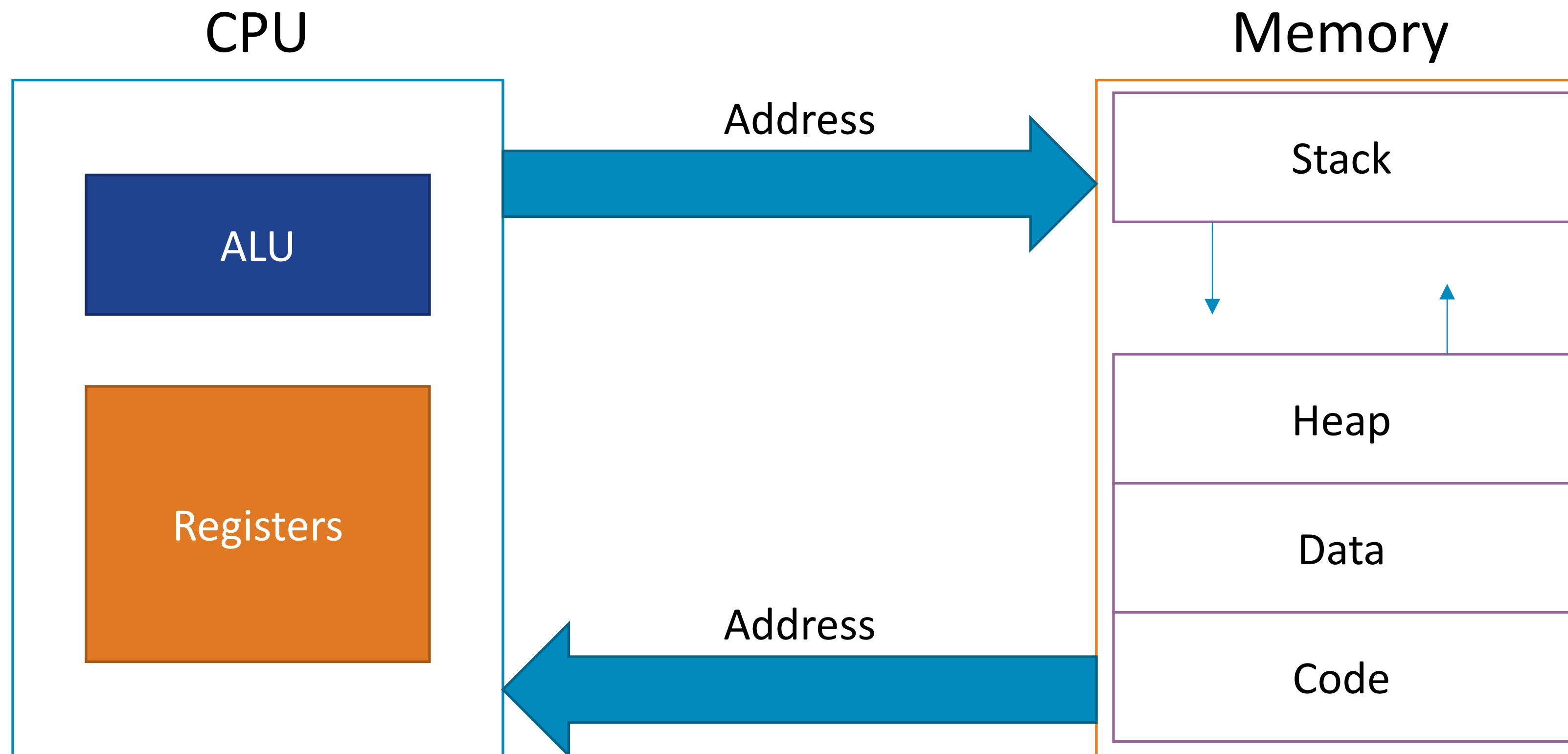
Another Form of Caching

- Similar in purpose to L1, L2, L3, etc. cache
- But now stored in RAM instead of on CPU

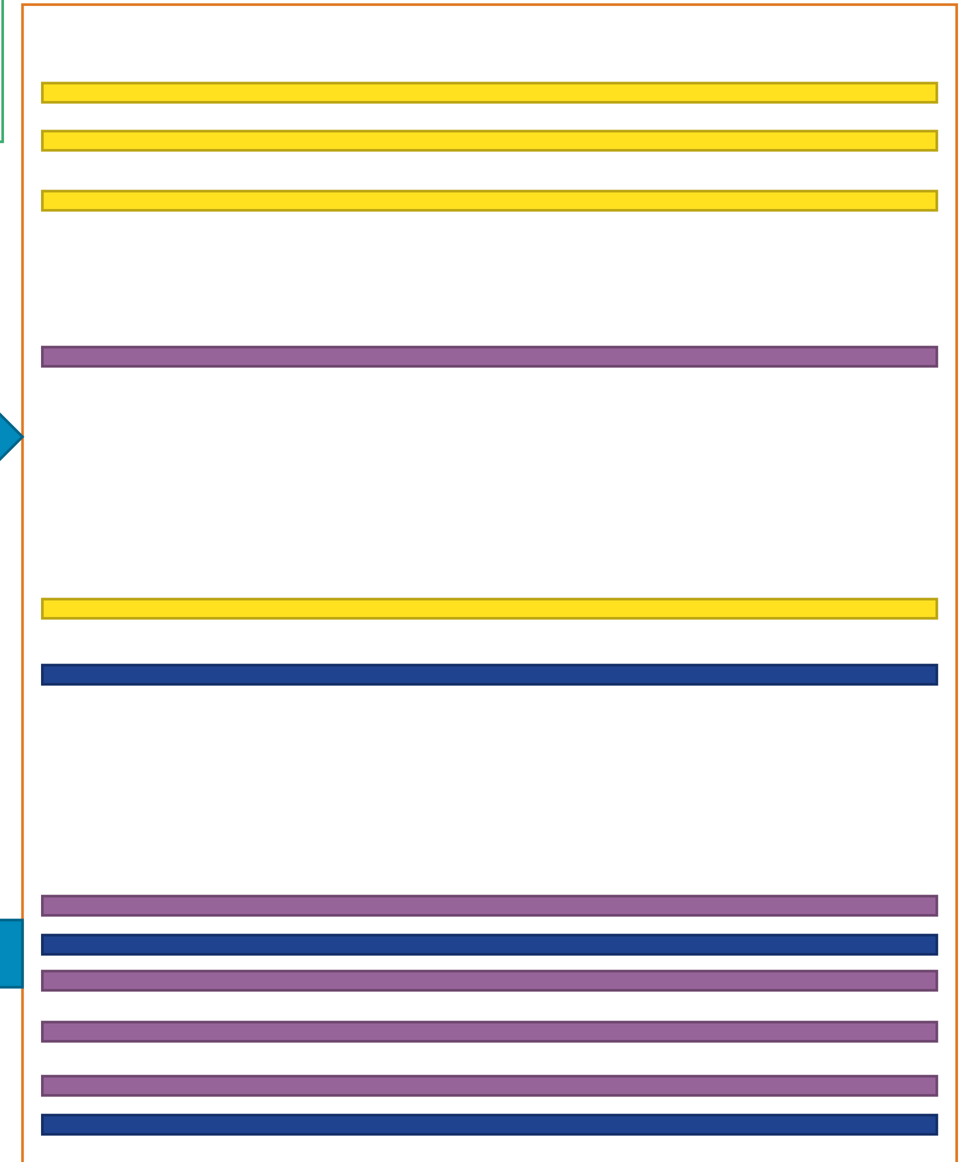
- Before: data block
- Now: page

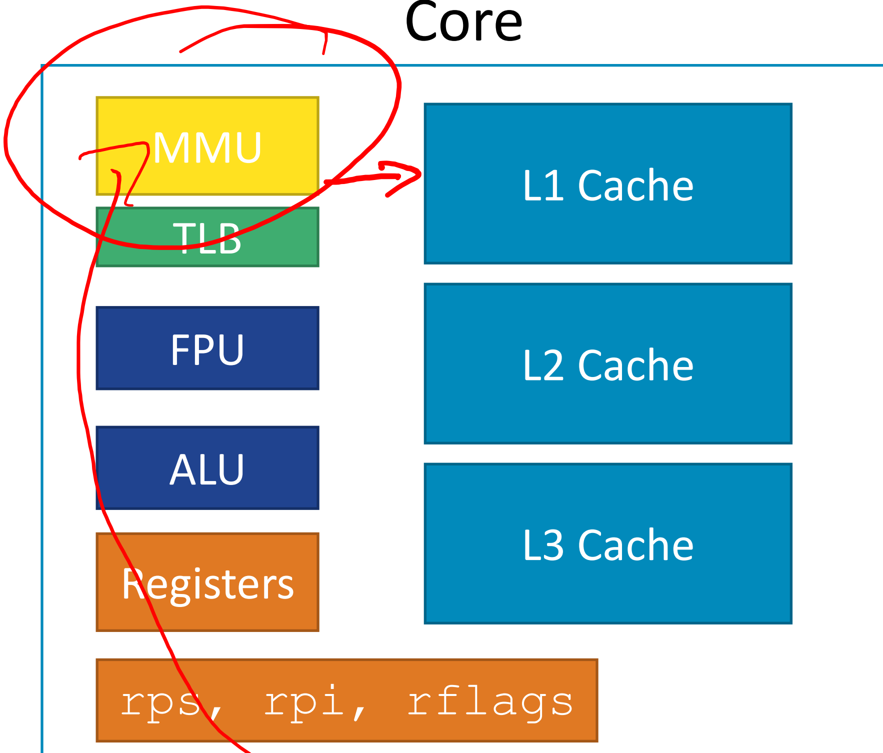
- Before: evicting/updating (or swapping)
- Now: paging

Address Translation



Virtual Addressing

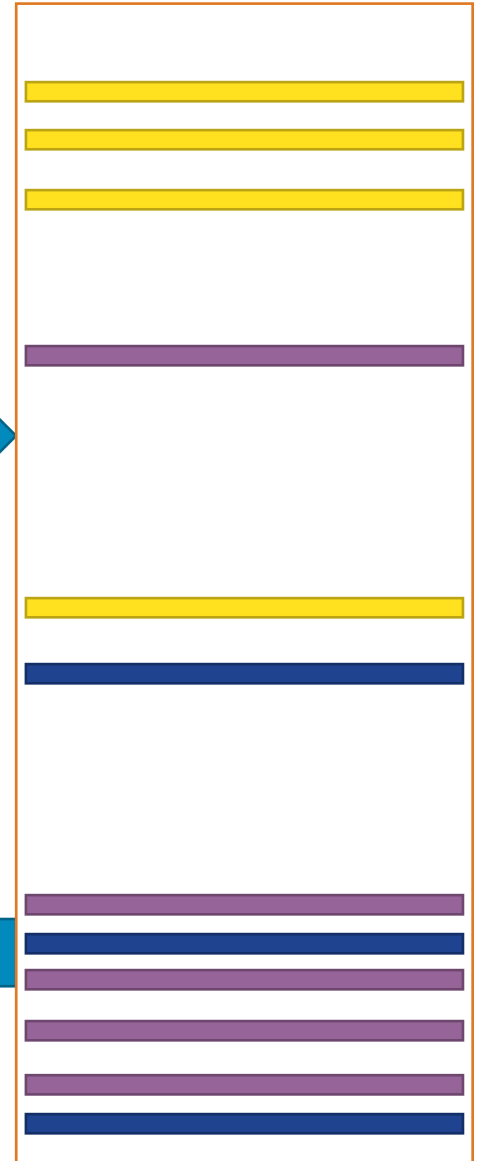




Virtual Addressing

SSD

Physical Memory



Exception

Core

Virtual Address

MMU

TLB

Physical Address

Cache

Physical Address

Data

rps, rpi, rflags

Registers

ALU

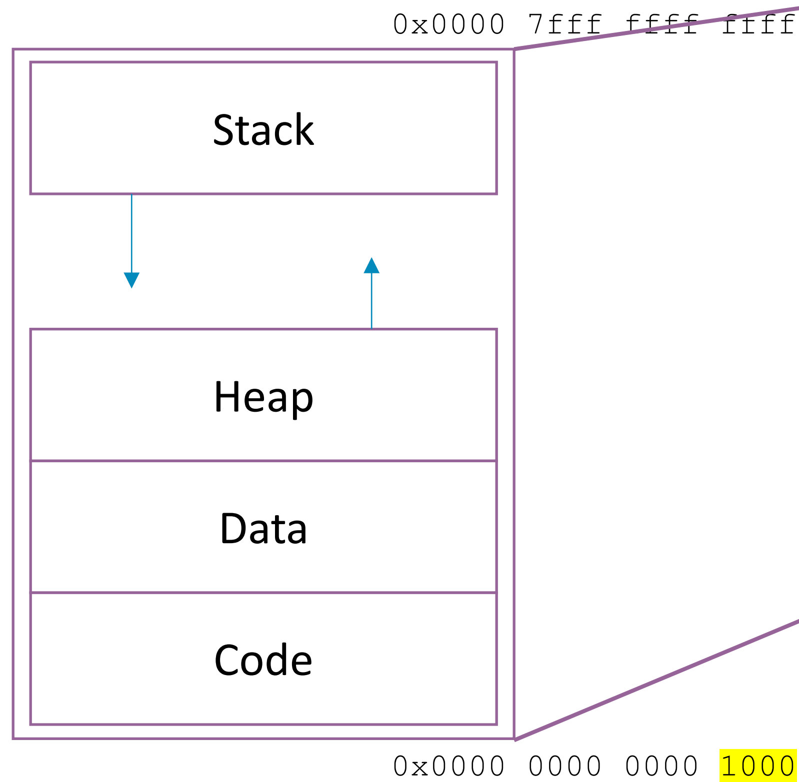
FPU

Possibilities

Let’s look at some good and bad possibilities

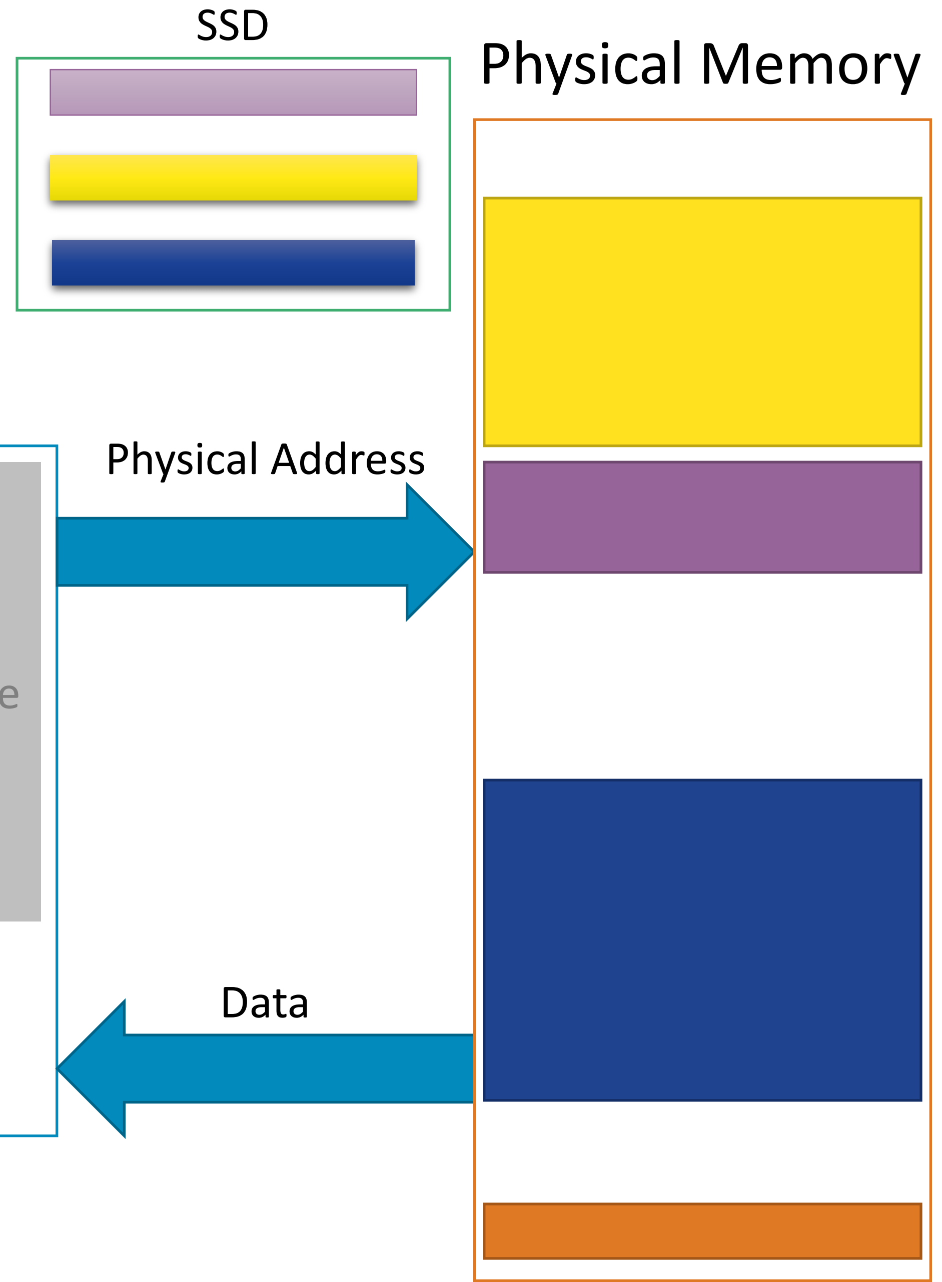
- Base-and-bound
- Segmentation
- Paging

Base-And-Bound



What if we introduce a fifth process?

Base-And-Bound Addressing



Exception

VA > Bound

Core

MMU

Process	Base	Bound
P1	0x…	size
P2	0x…	size
P3	0x…	size
P4	0x…	size

Virtual Address (VA)

Physical Address (PA)

Cache

PA = VA + Base

rps, rpi, rflags

Registers

ALU

FPU

What if we introduce a fifth process?

Base-And-Bound Addressing

Exception

VA > Bound

Core

Virtual Address (VA)

MMU

Process	Base	Bound
P1	0x…	size
P2	0x…	size
P3	0x…	size
P4	0x…	size

Physical Address (PA)

Cache

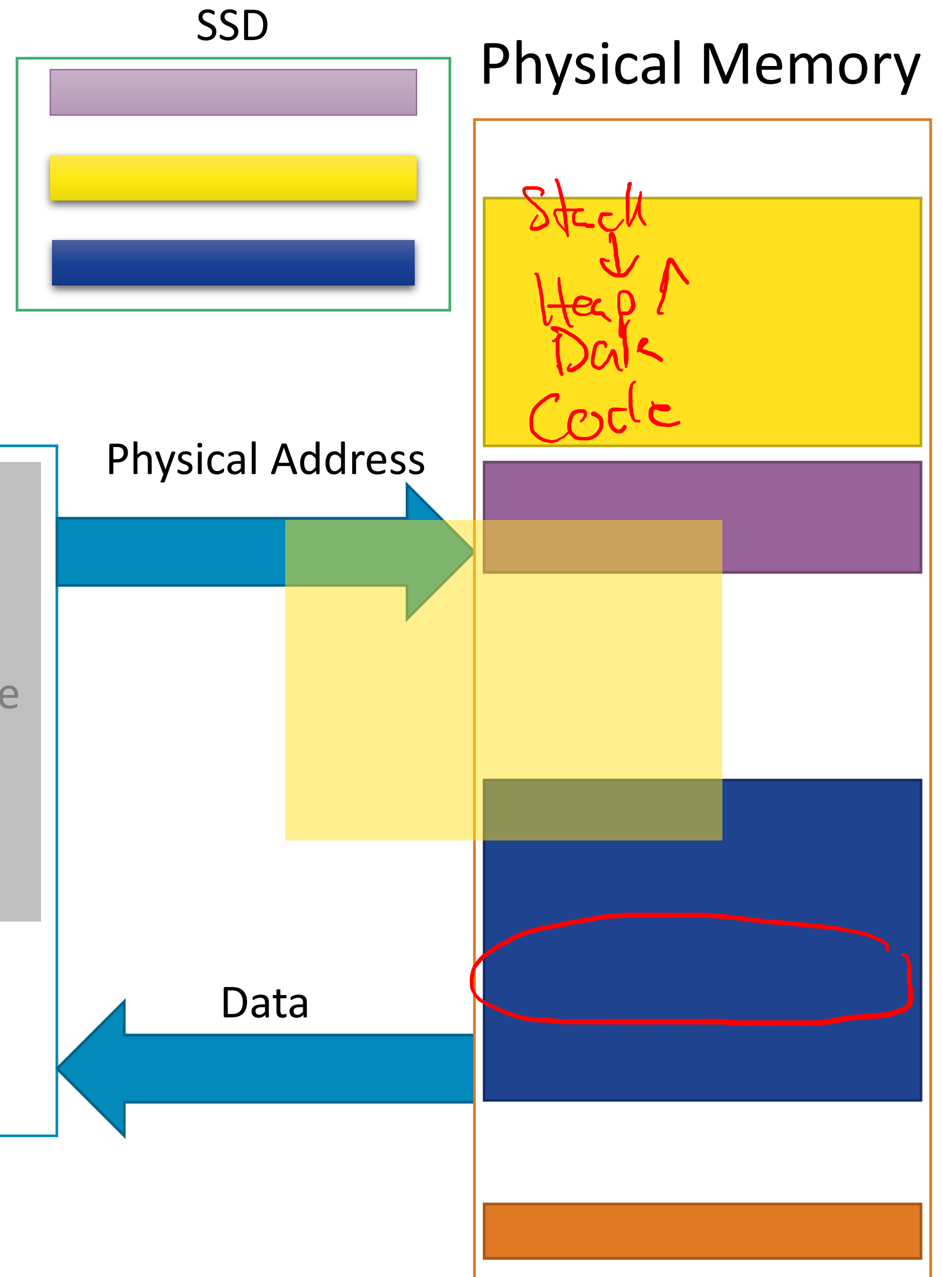
PA = VA + Base

rps, rpi, rflags

Registers

ALU

FPU



VA > Bound

PA = VA + Base

Practice with Base-and-Bound

Assume that you are currently executing a process P with Base `0x1234` and Bound `0x100`.

- What is the physical address that corresponds to the virtual address `0x47`?
- What is the physical address that corresponds to the virtual address `0x123`?

VA > Bound

PA = VA + Base

Practice with Base-and-Bound

Assume that you are currently executing a process P with Base `0x1234` and Bound `0x100`.

- What is the physical address that corresponds to the virtual address `0x47`?

  PA = `0x127b`

- What is the physical address that corresponds to the virtual address `0x123`?

  PA is invalid -> exception

Evaluating Base-and-Bound

- **Isolation:** don’t want different process states collided in physical memory

- **Efficiency:** want fast reads/writes to memory

- **Sharing:** want option to overlap for communication

- **Utilization:** want best use of limited resource

- **Virtualization:** want to create illusion of more resources

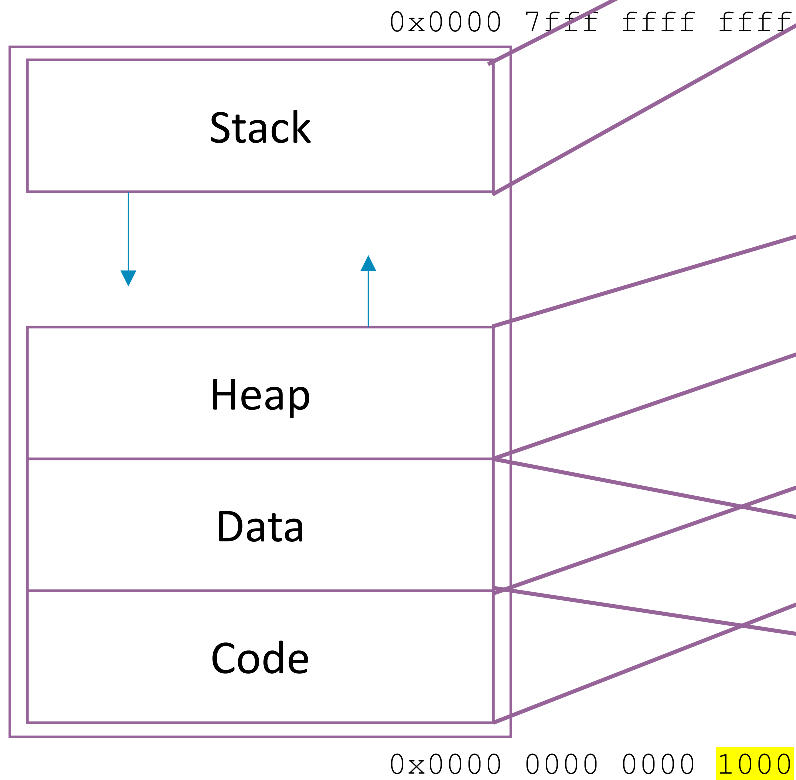
Possibilities

Let's look at some good and bad possibilities

- ~~Base-and-bound~~
- Segmentation
- Paging

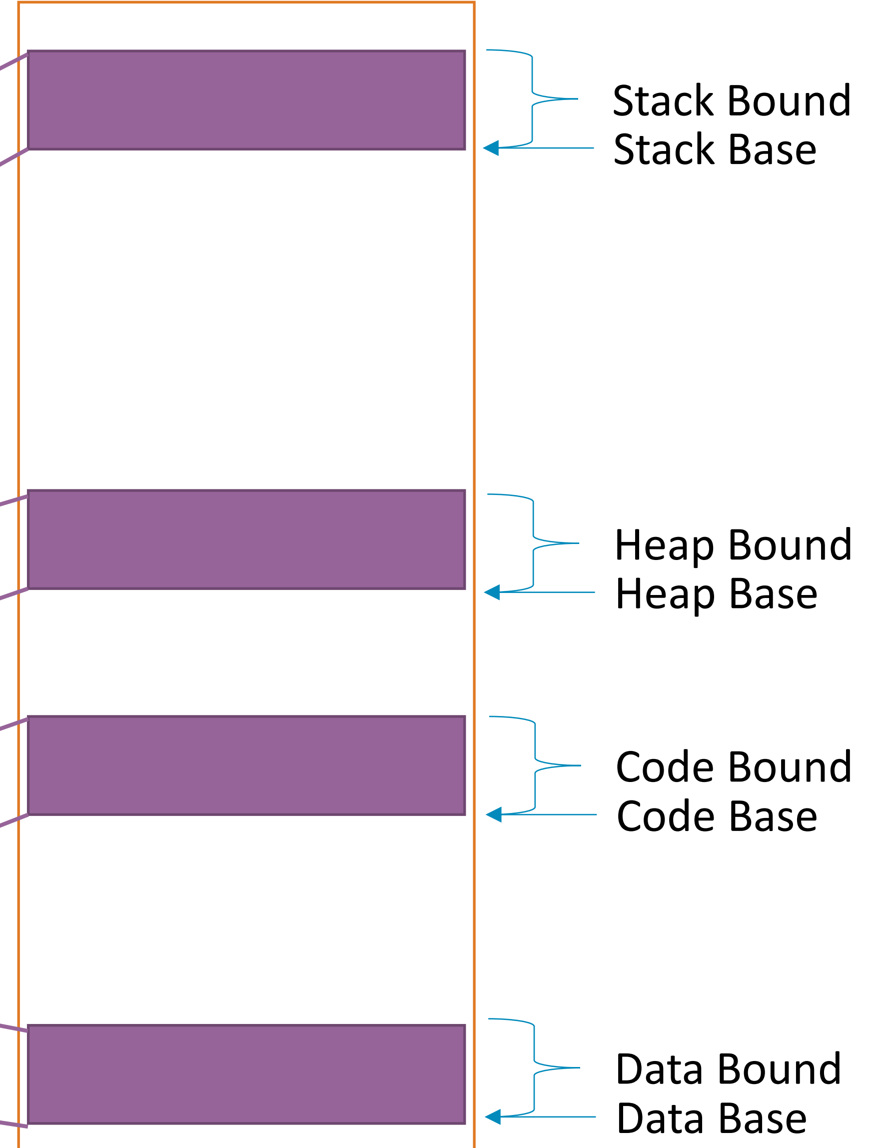
Segmentation

Process P1
Virtual Address Space



Not to scale

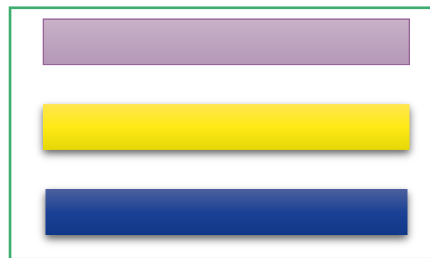
Physical Memory 16 GB



What if we introduce another process and the data segment is too big?

Segmentation

SSD



Physical Memory

Exception

Offset > Bound[Index] or invalid access

Core

Physical Address

MMU

Process	Base	Bound	Permissions
P1 Code	0x…	size	R-X
P1 Data	0x…	size	RW-
P1 Heap	0x…	size	RW-
P1 Stack	0x…	size	RW-
P2 …			

Virtual Address (VA)

Physical Address (PA)

Cache

PA = Base[Index] + Offset

Data

rps, rpi, rflags

Registers

ALU

FPU

VA must be decoded into Index and Offset (like caching).
Index gives segment
Offset gives byte address inside segment

VA: | Index | Offset |

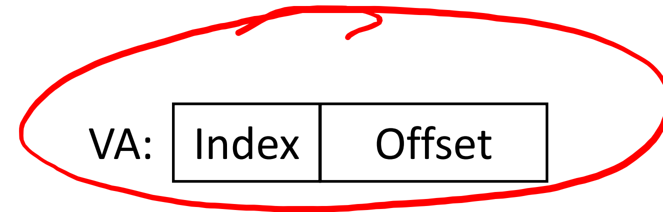
Offset > Bound[Index] or invalid access

Practice with Segmentation

PA = Base[Index] + Offset

Assume that you are currently executing a process P with the following segment table:

Base	Bound	Permissions
0x4747	0x080	RW-
0x2424	0x040	RW-
0x0023	0x080	RW-
0x1000	0x200	R-X



How many bits needed for the Index?

- What is the physical address that corresponds to the virtual address 0x001?
- What is the physical address that corresponds to the virtual address 0xD47?

Offset > Bound[Index] or invalid access

PA = Base[Index] + Offset

Practice with Segmentation

Assume that you are currently executing a process P with the following segment table:

Base	Bound	Permissions
0x4747	0x080	RW-
0x2424	0x040	RW-
0x0023	0x080	RW-
0x1000	0x200	R-X

VA: | Index | Offset |

How many bits needed for the Index?

- What is the physical address that corresponds to the virtual address 0x001?

| 00 | 00 0000 0001 |

0x4748

- What is the physical address that corresponds to the virtual address 0xD47?

| 11 | 01 0100 0111 |

0x1147

Evaluating Segmentation

- **Isolation:** don’t want different process states collided in physical memory

- **Efficiency:** want fast reads/writes to memory

- **Sharing:** want option to overlap for communication

- **Utilization:** want best use of limited resource

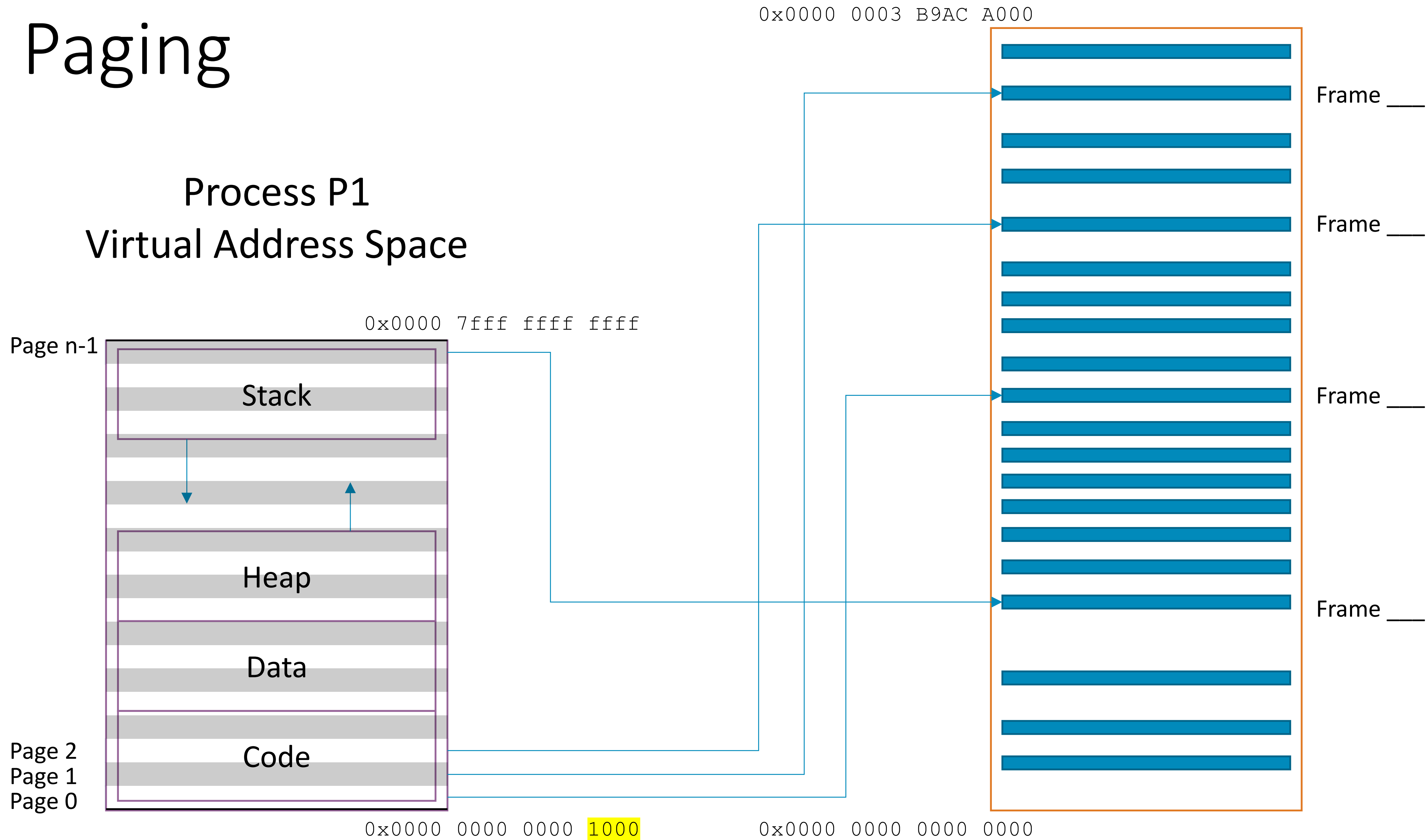
- **Virtualization:** want to create illusion of more resources

Possibilities

Let's look at some good and bad possibilities

- ~~Base-and-bound~~
- ~~Segmentation~~
- Paging

Paging



What if we introduce another process and we've run out of physical memory?

Paging

SSD

Exception

Invalid page or invalid access

Core

Virtual Address (VA)

MMU Page Table

P Frame	Permissions
47	R-X
NULL	RW-
13	RW-
42	RW-
105	R-X

Physical Address (PA)

Cache

Physical Address

Data

PA = Frame[Page] +++ Offset

Concatenation

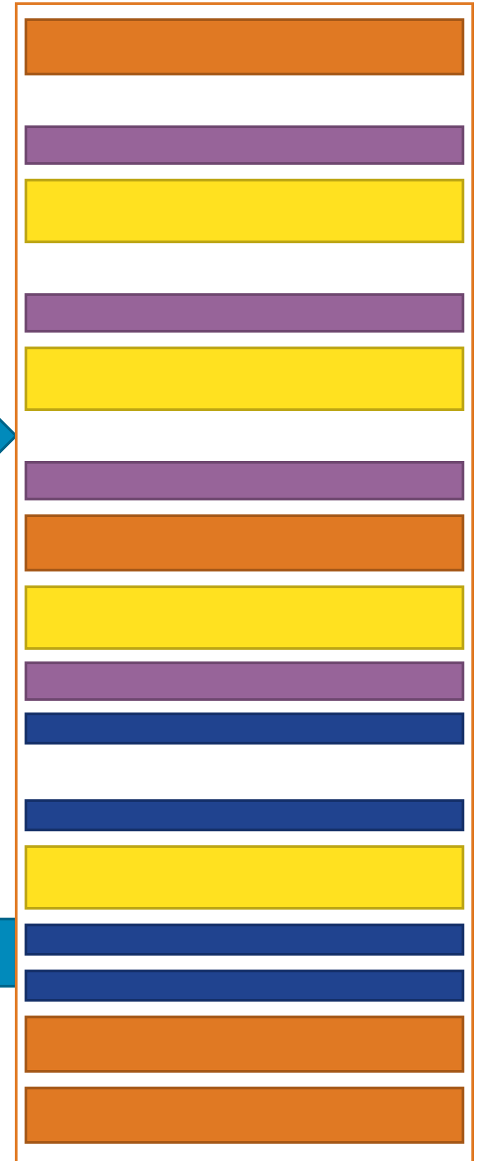
rps, rpi, rflags

Registers

ALU

FPU

Physical Memory



VA must be decoded into Page and Offset (like caching).
Page # is index into page table (gives P Frame)
Offset gives byte address inside P Frame

VA: | Page | Offset |

Invalid page or invalid access

PA = Frame[Page] +++ Offset

Practice with Paging

Assume that you are currently executing a process P with the following page table on a system with 16-byte pages:

Assume the page table is much bigger

Page	Frame	Permissions
0x17	0x47	RW-
0x16	0xF4	RW-
0x15	NULL	RW-
0x14	0x23	R-X

VA:

How many bits for the Page number?
How many bits for the Offset?

- What is the physical address that corresponds to the virtual address `0x147`?
- What is the physical address that corresponds to the virtual address `0x16E`?

Practice with Paging

Invalid page or invalid access

PA = Frame[Page] +++ Offset

Assume that you are currently executing a process P with the following page table on a system with 16-byte pages:

Assume the page table is much bigger

Page	Frame	Permissions
0x17	0x47	RW-
0x16	0xF4	RW-
0x15	NULL	RW-
0x14	0x23	R-X

VA: | Page | Offset |

How many bits for the Page number?
How many bits for the Offset?

- What is the physical address that corresponds to the virtual address `0x147`?

00010100	0111

0x237

- What is the physical address that corresponds to the virtual address `0x16E`?

00010110	1110

0xF4E

Invalid page or invalid access

PA = Frame[Page] +++ Offset

Practice with Paging

Assume that you are currently executing a process P with the following page table on a system with 16-byte pages:

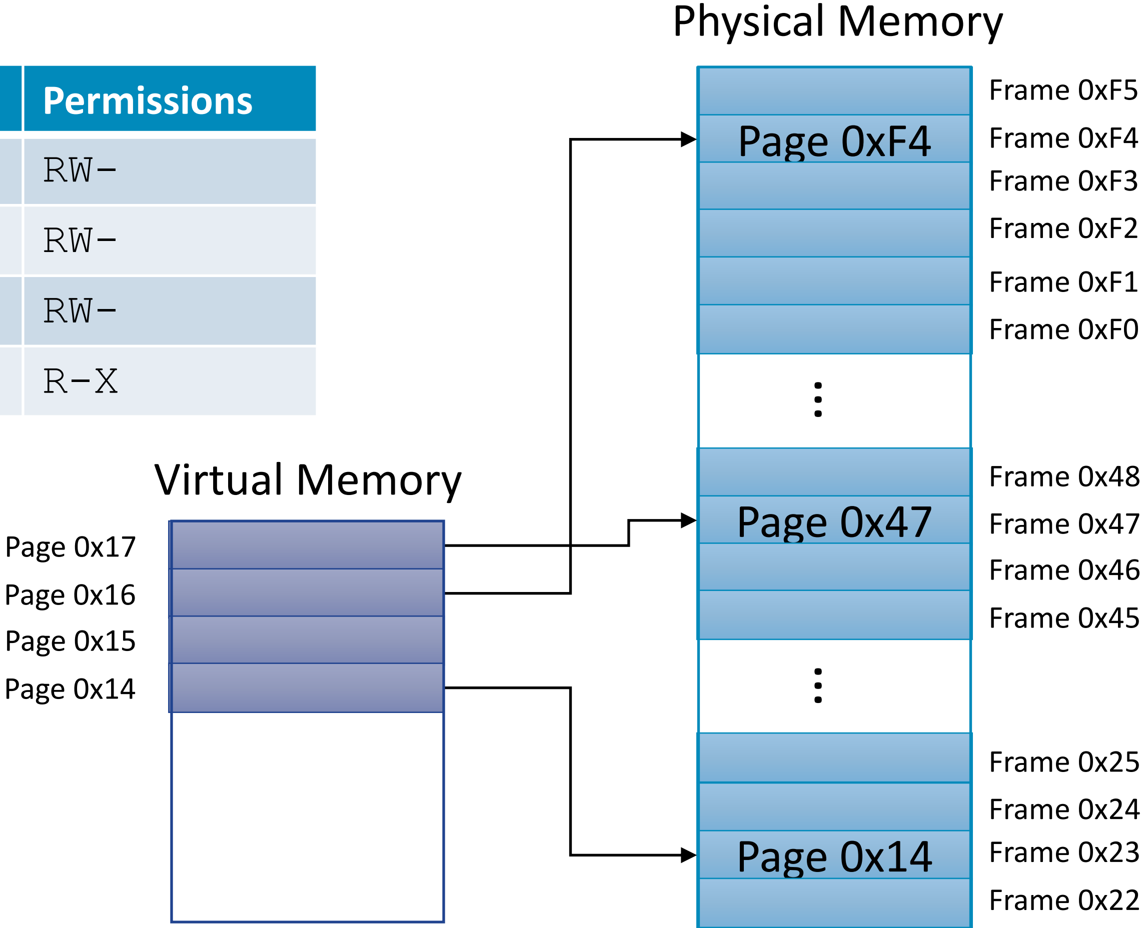
Page	Frame	Permissions
0x17	0x47	RW-
0x16	0xF4	RW-
0x15	NULL	RW-
0x14	0x23	R-X

Assume the page table is much bigger

How many bits for the Page number?
How many bits for the Offset?

What is the physical address that corresponds to the virtual address 0x147?

0x237



Memory as a Cache

- Each page table entry has a valid bit
- For valid entries, frame indicates physical address of page in memory
- A **page fault** occurs when a program requests a page that is not currently in memory
  - takes time to handle, so context switch
  - evict another page in memory to make space (which one?)

MMU

v	Frame	Permissions
1	47	RW-
0	NULL	RW-
0	13	RW-
1	42	R-X
⋮		

Page Replacement Algorithms

- **Random:** Pick any page to eject at random
  - Used mainly for comparison
- **FIFO:** The page brought in earliest is evicted
  - Ignores usage
- **OPT:** Belady's algorithm
  - Select page not used for longest time
- **LRU:** Evict page that hasn't been used for the longest
  - Past could be a good predictor of the future
- **MRU:** Evict the most recently used page
- **LFU:** Evict least frequently used page

Invalid page or invalid access

PA = Frame[Page] +++ Offset

More Paging Practice

- Assume that you are currently executing a process P with the following page table on a system with 256-byte pages:

Page	valid	P Frame	Permissions
…	…	…	…
0xFA	1	0x47	R,W
0xF9	1	0x24	R,W
0xF8	0	NULL	R,W
0xF7	0	0x23	R,X
…	…	…	…

How many bits for the Page number?
How many bits for the Offset?

- What is the physical address that corresponds to the virtual address 0xF947?

- What is the physical address that corresponds to the virtual address 0xF700?

More Paging Practice

Invalid page or invalid access

PA = Frame[Page] +++ Offset

- Assume that you are currently executing a process P with the following page table on a system with 256-byte pages:

Page	valid	P Frame	Permissions
…	…	…	…
0xFA	1	0x47	R,W
0xF9	1	0x24	R,W
0xF8	0	NULL	R,W
0xF7	0	0x23	R,X
…	…	…	…

VA:

How many bits for the Page number?
How many bits for the Offset?

- What is the physical address that corresponds to the virtual address 0xF947?

0xF9	0x47

0x2447

- What is the physical address that corresponds to the virtual address 0xF700?

0xF7	0x00

0x2300 Page fault

Evaluating Paging

- **Isolation:** don't want different process states collided in physical memory

- **Efficiency:** want fast reads/writes to memory

- **Sharing:** want option to overlap for communication

- **Utilization:** want best use of limited resource

- **Virtualization:** want to create illusion of more resources