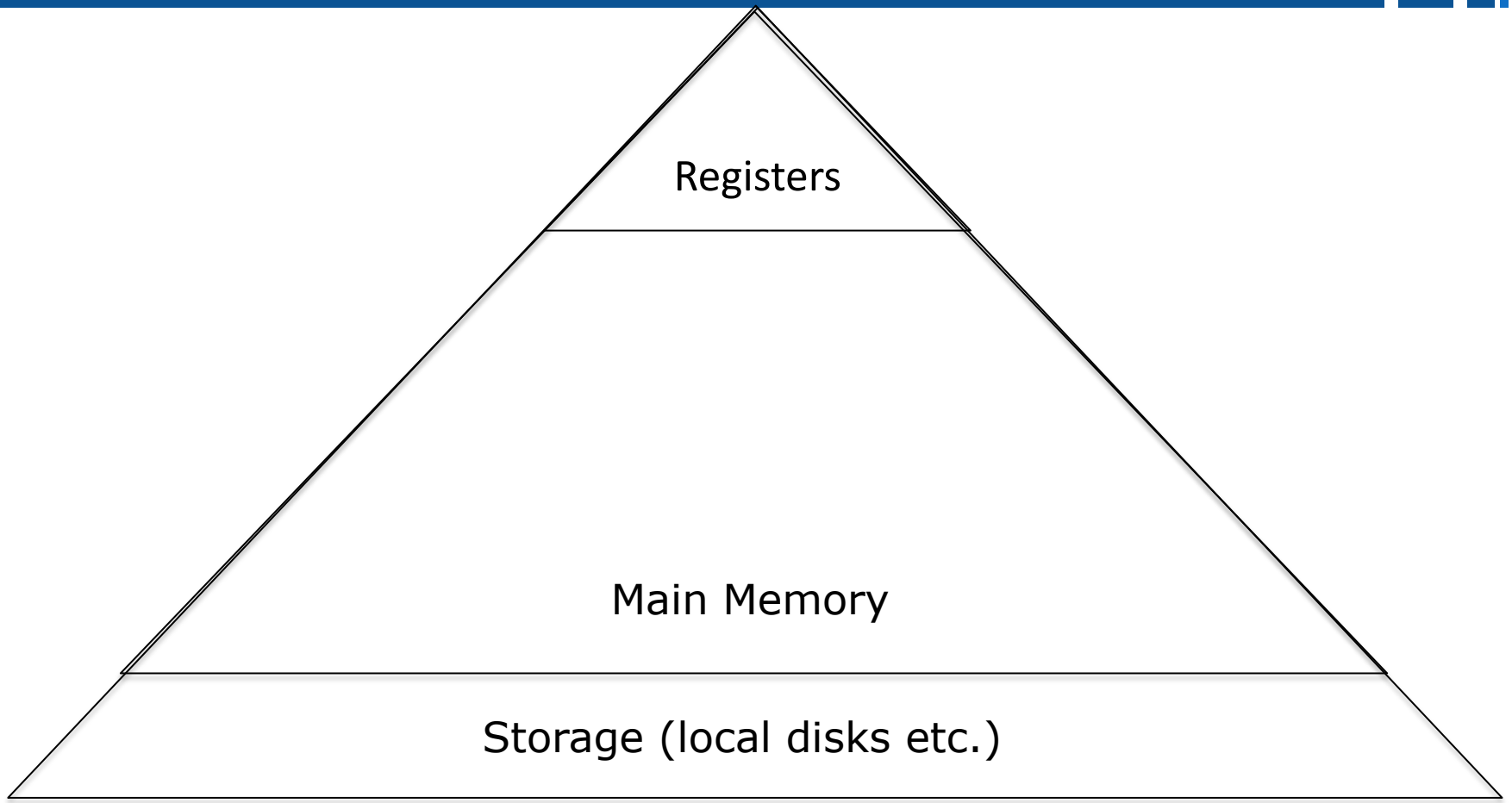
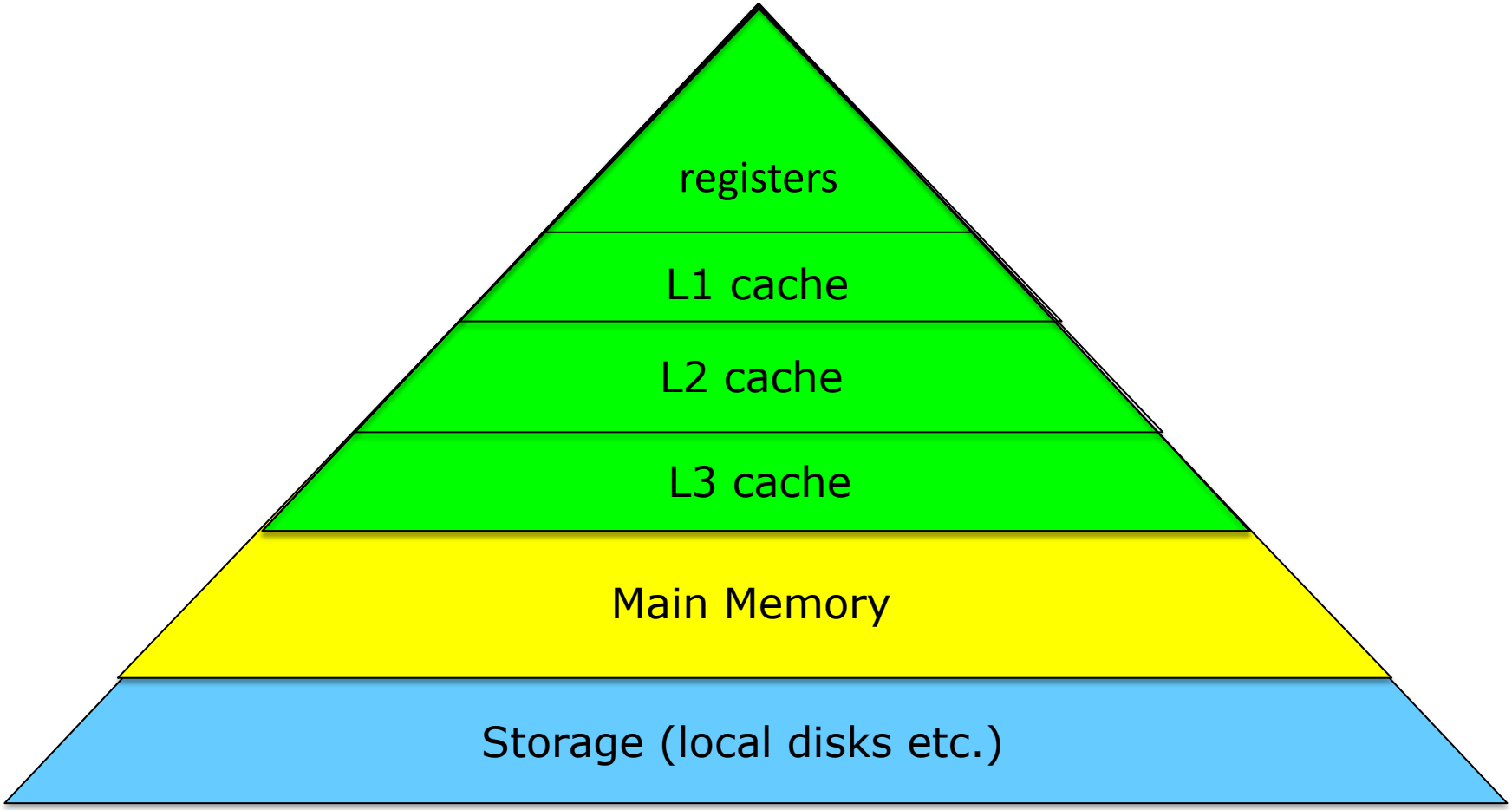
Operating Systems
Memory Management

David Hay
Dror Feitelson

Memory So Far

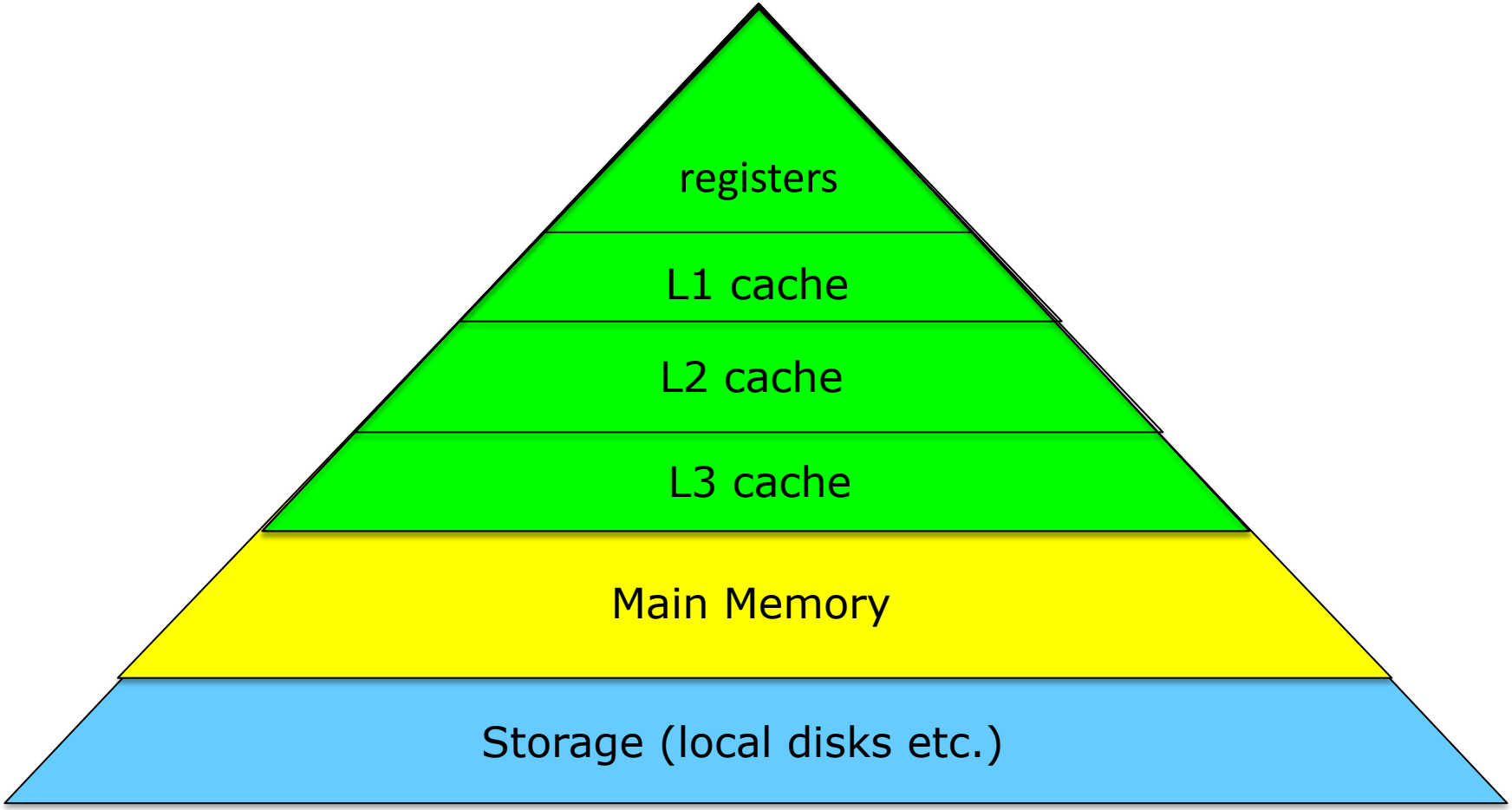


The Memory Hierarchy



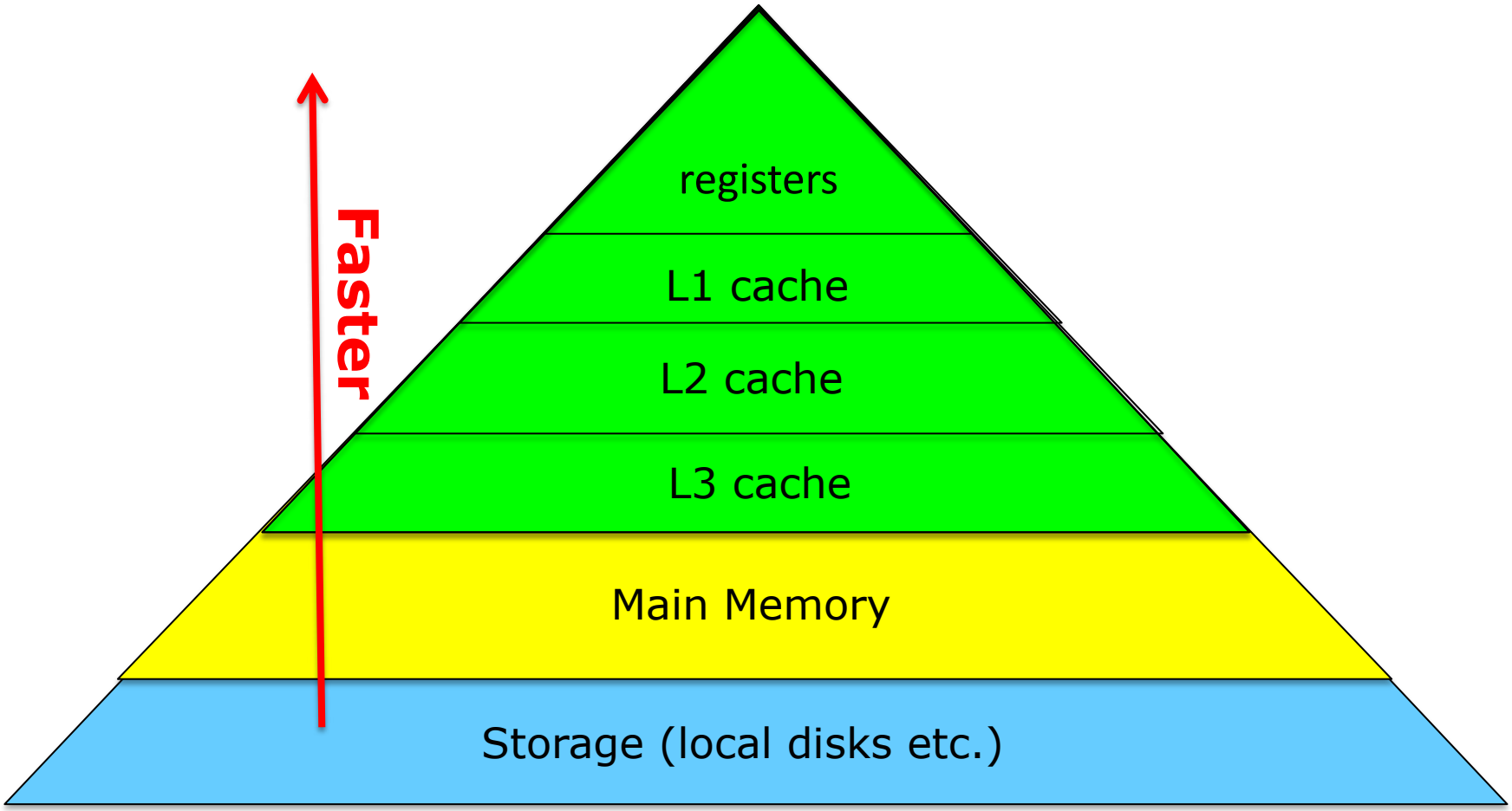
Remote storage

The Memory Hierarchy



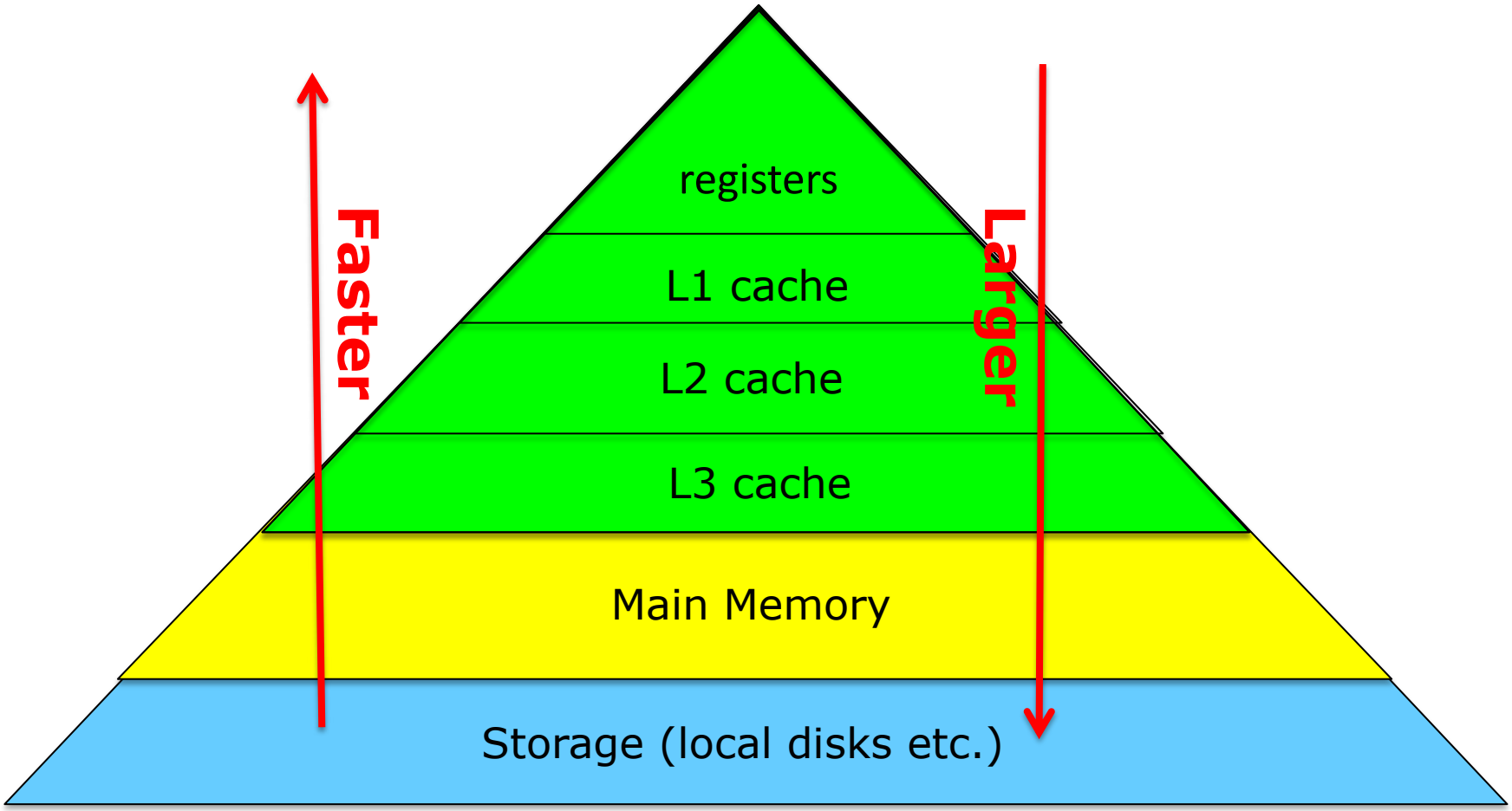
Remote storage

The Memory Hierarchy



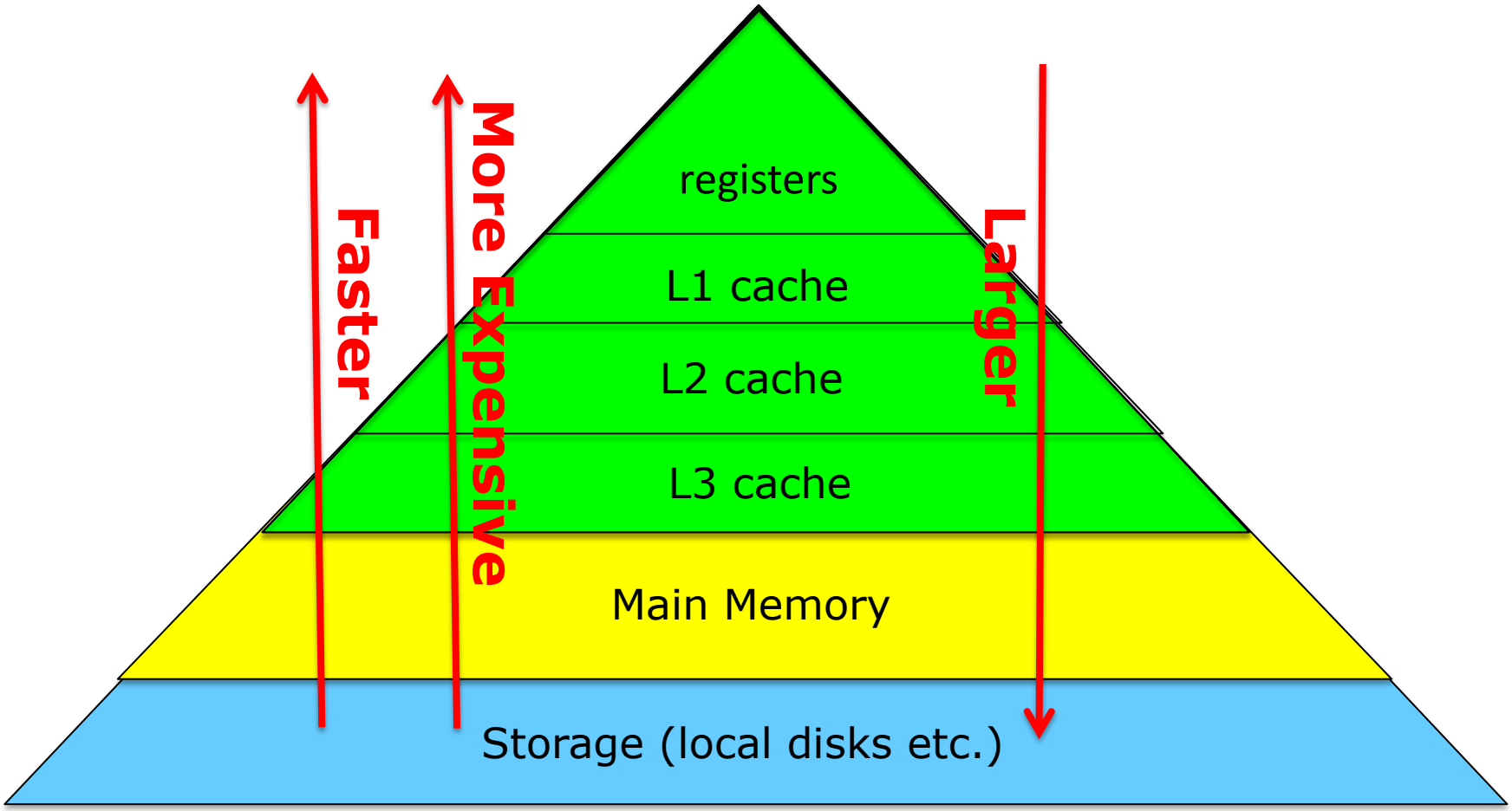
Remote storage

The Memory Hierarchy

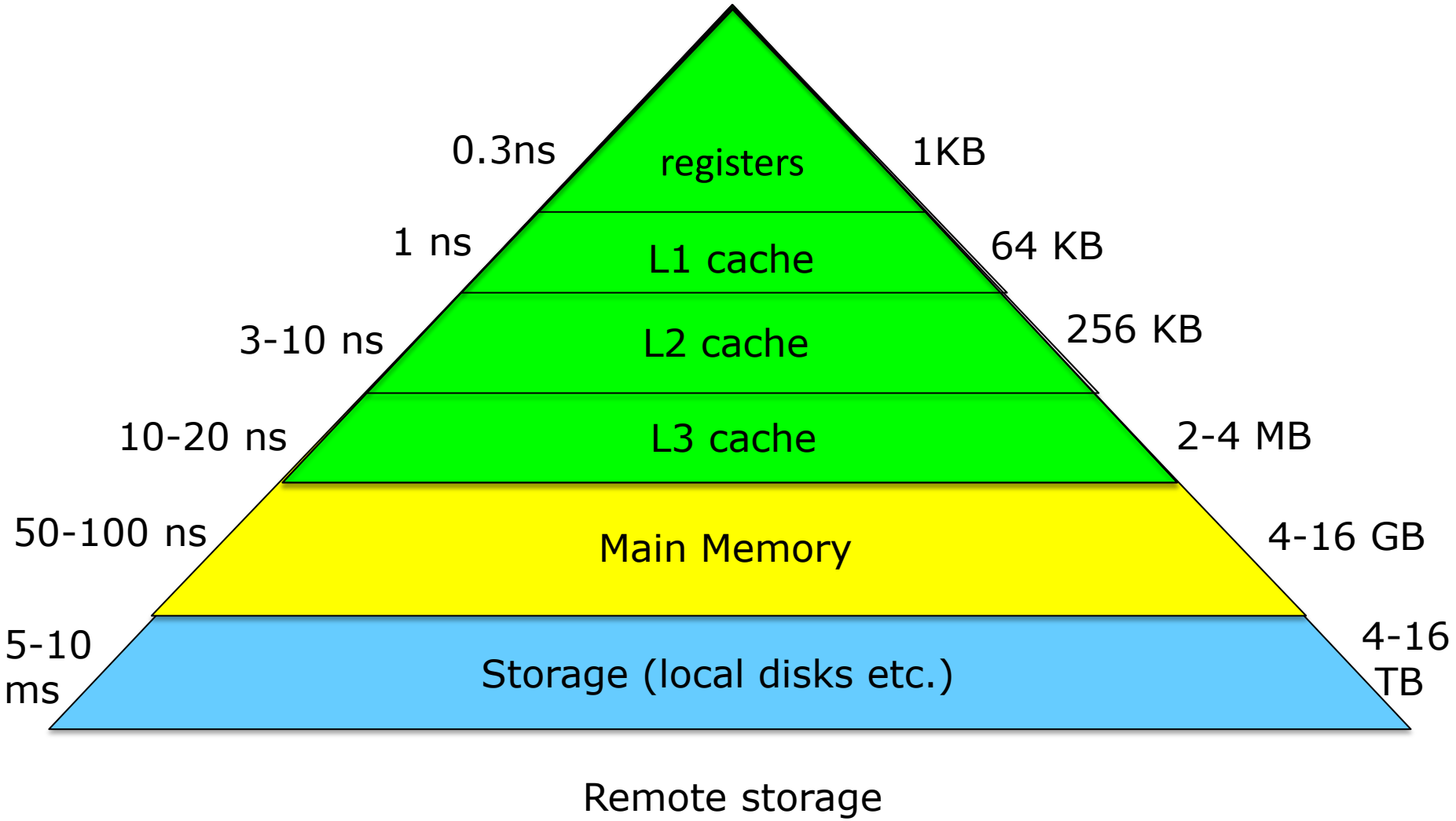


Remote storage

The Memory Hierarchy

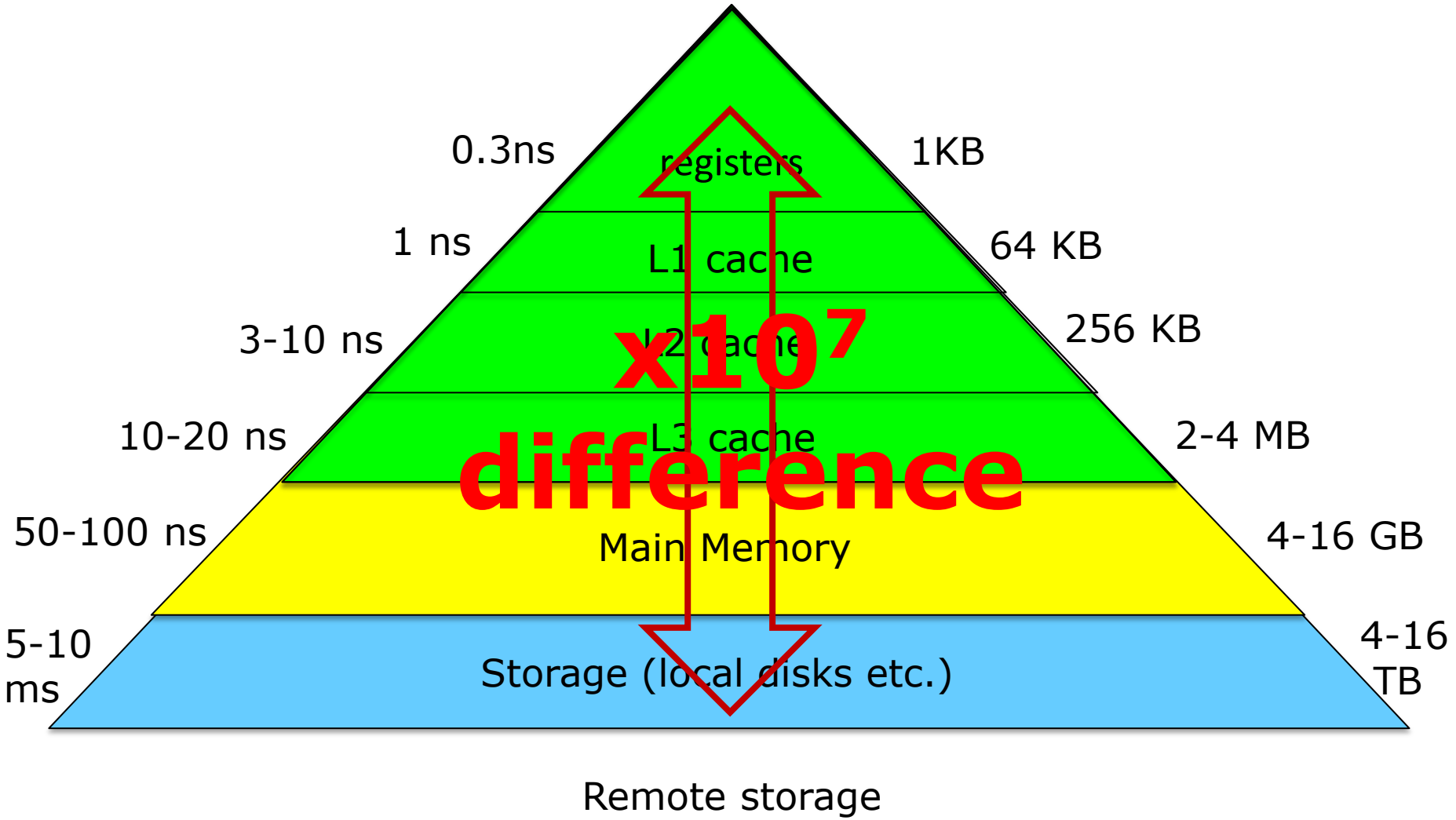


The Memory Hierarchy



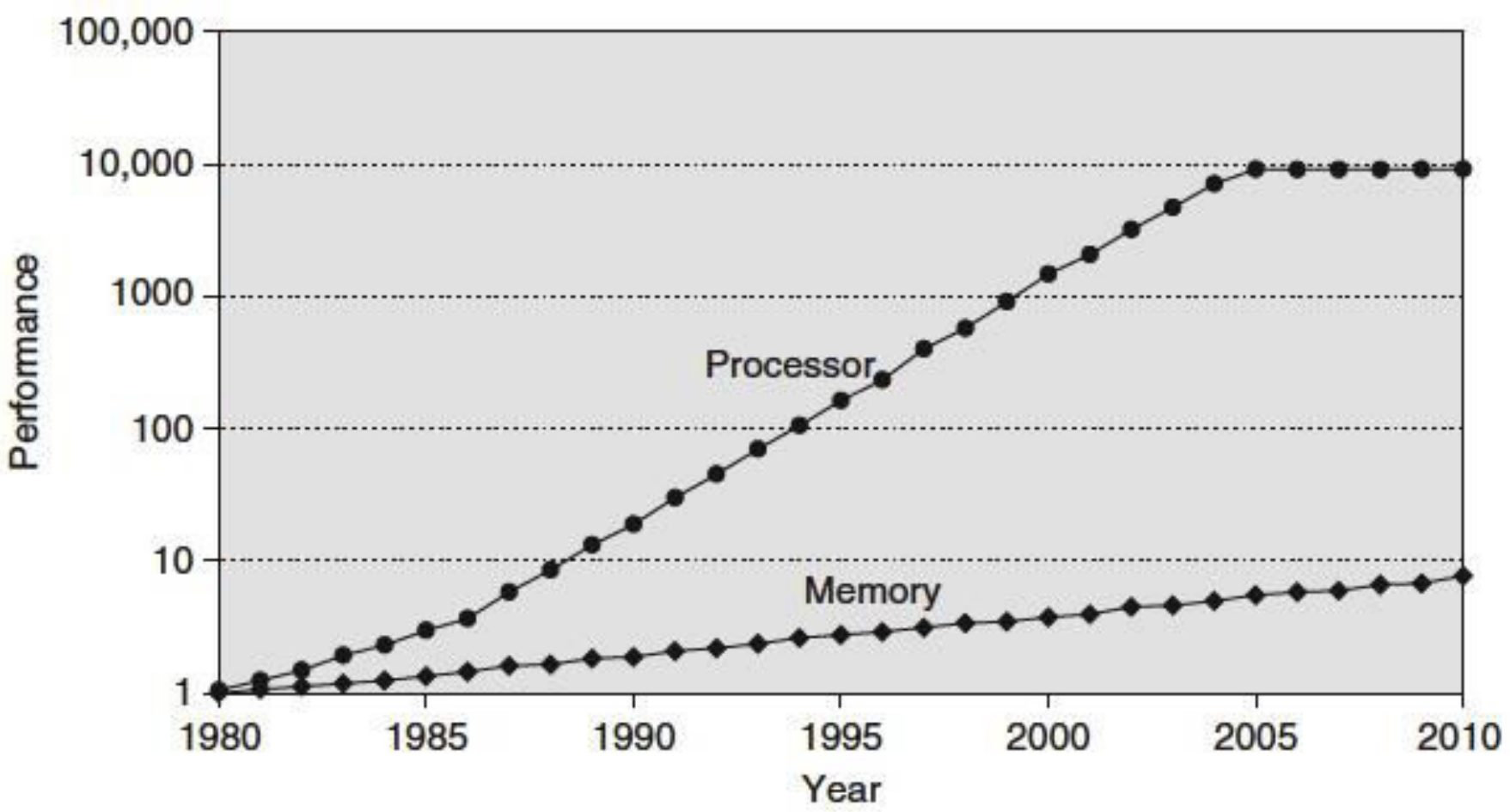
Hennessy and Patterson, September 2012, servers

The Memory Hierarchy

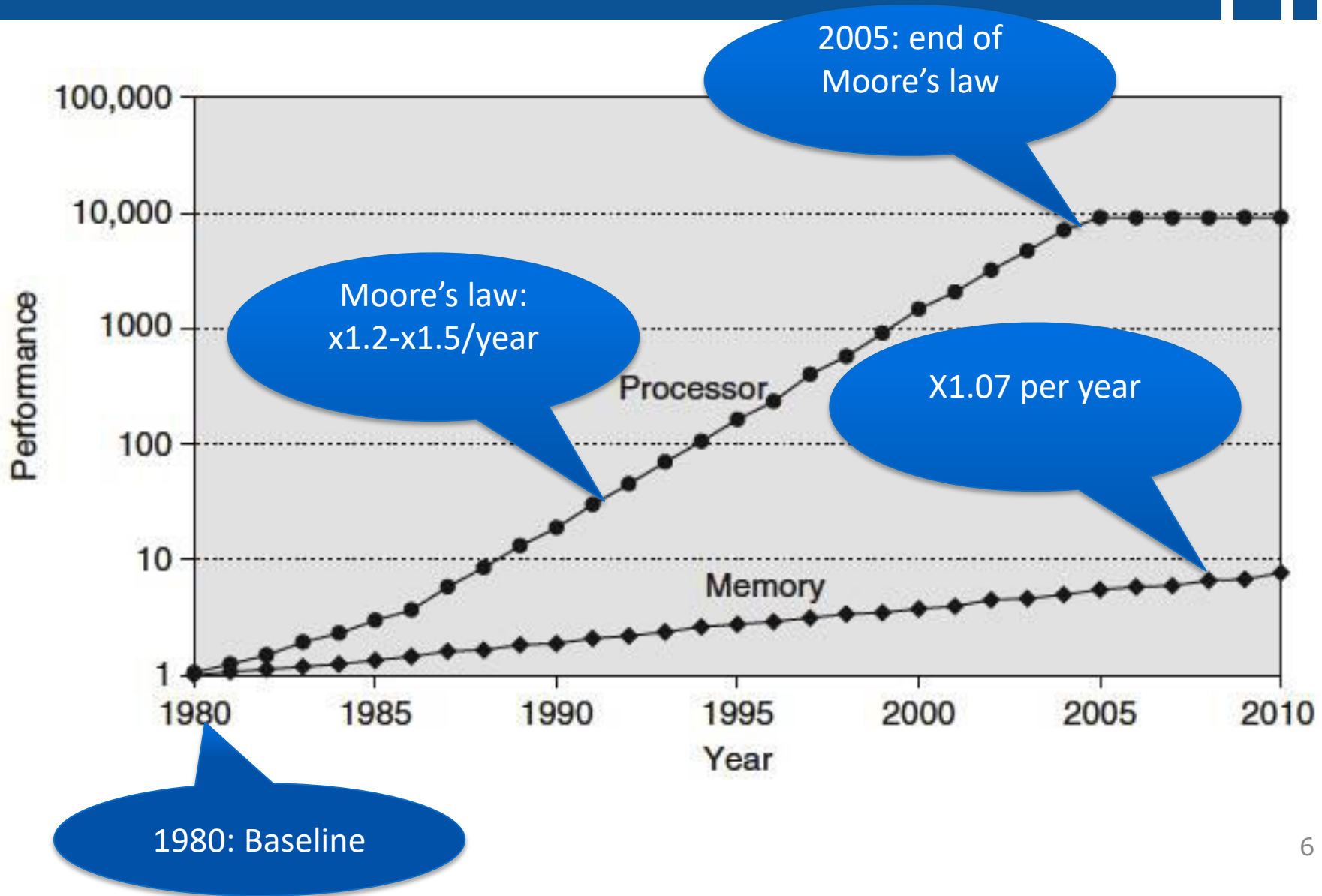


Hennessy and Patterson, September 2012, servers

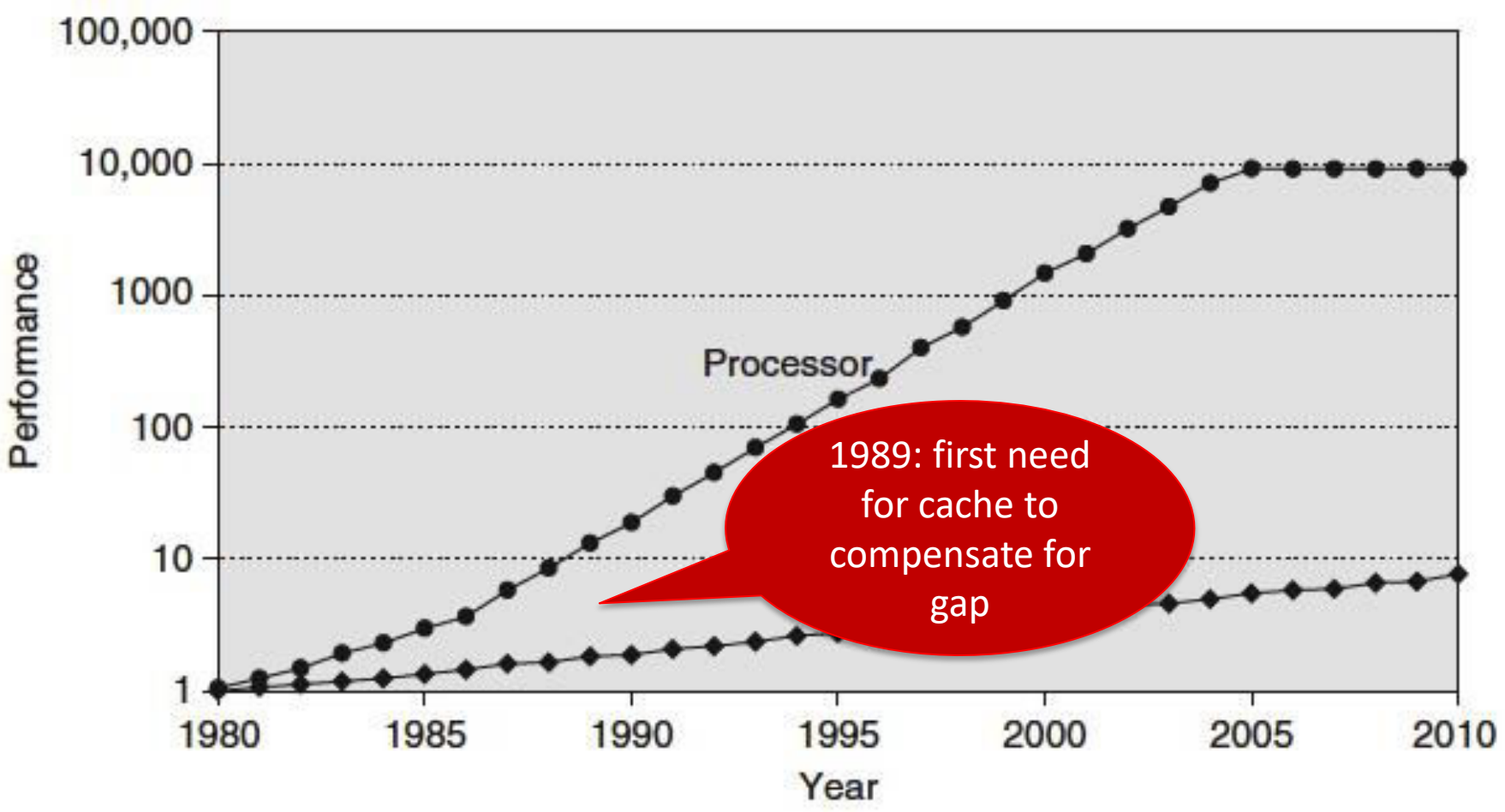
CPU-DRAM Gap 1980-2010



CPU-DRAM Gap 1980-2010



CPU-DRAM Gap 1980-2010



Caching

Small fast memory serves as cache for large slow memory

- How to find things in the cache?
- Which items should be stored in the cache?
- Which items should be evicted from the cache?

Similar considerations and solutions in HW caching and in OS memory management

Principle of Locality

Temporal locality:

If we accessed a certain address, the chances are high to access it again shortly.

- Data: updating
- Instructions: loops

Principle of Locality

Temporal locality:

If we accessed a certain address, the chances are high to access it again shortly.

- Data: updating
- Instructions: loops

Spatial locality:

If we accessed a certain address, the chances are high to access its neighbors.

- Data: arrays
- Instructions: sequential execution

Principle of Locality

Temporal locality:

If we accessed a certain address, the chances are high to access it again shortly.

- Data
- Instr

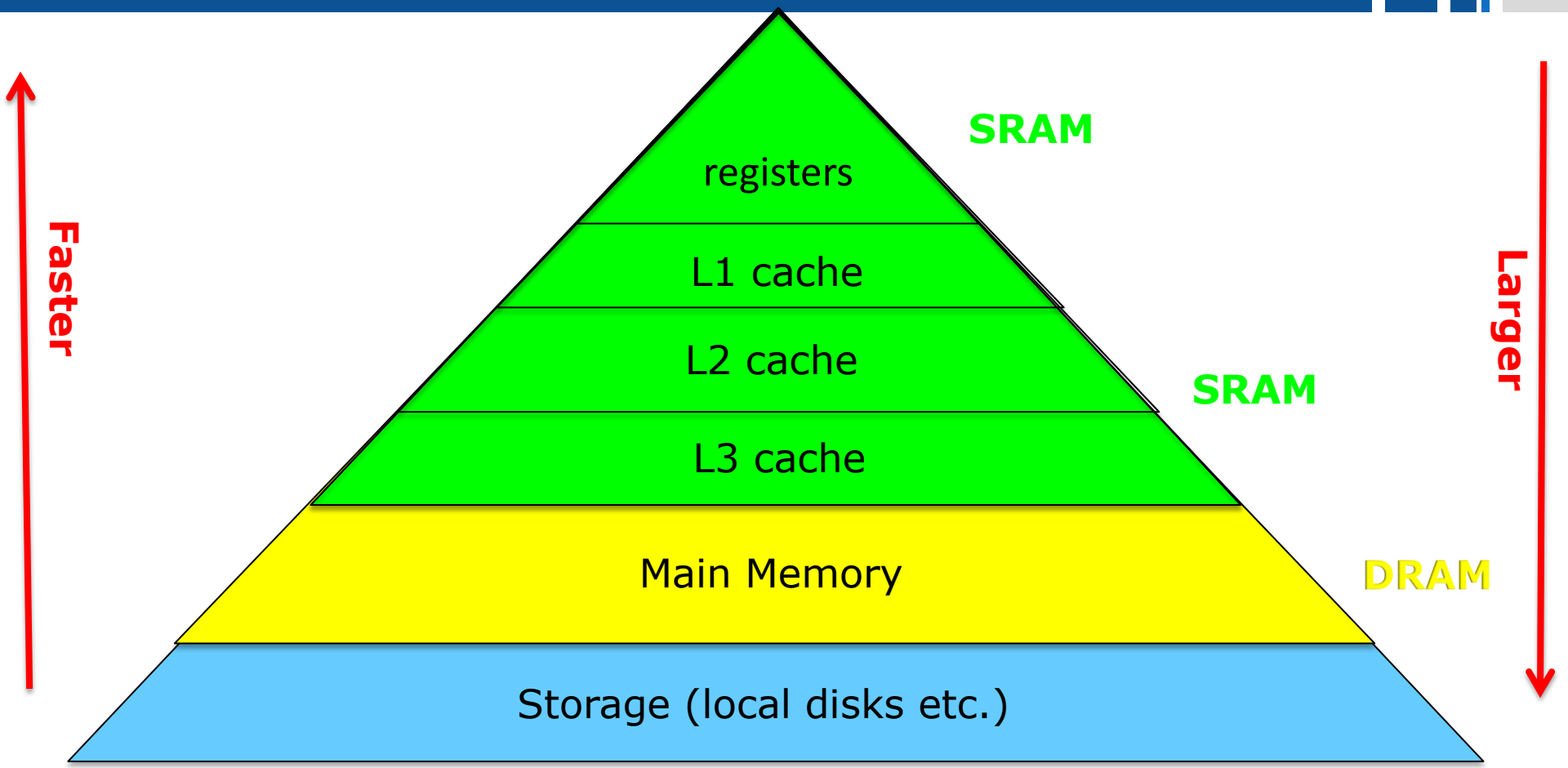
Idea: Keep data and instructions the CPU is **most likely to need next** in fast memory close to the CPU

Spati

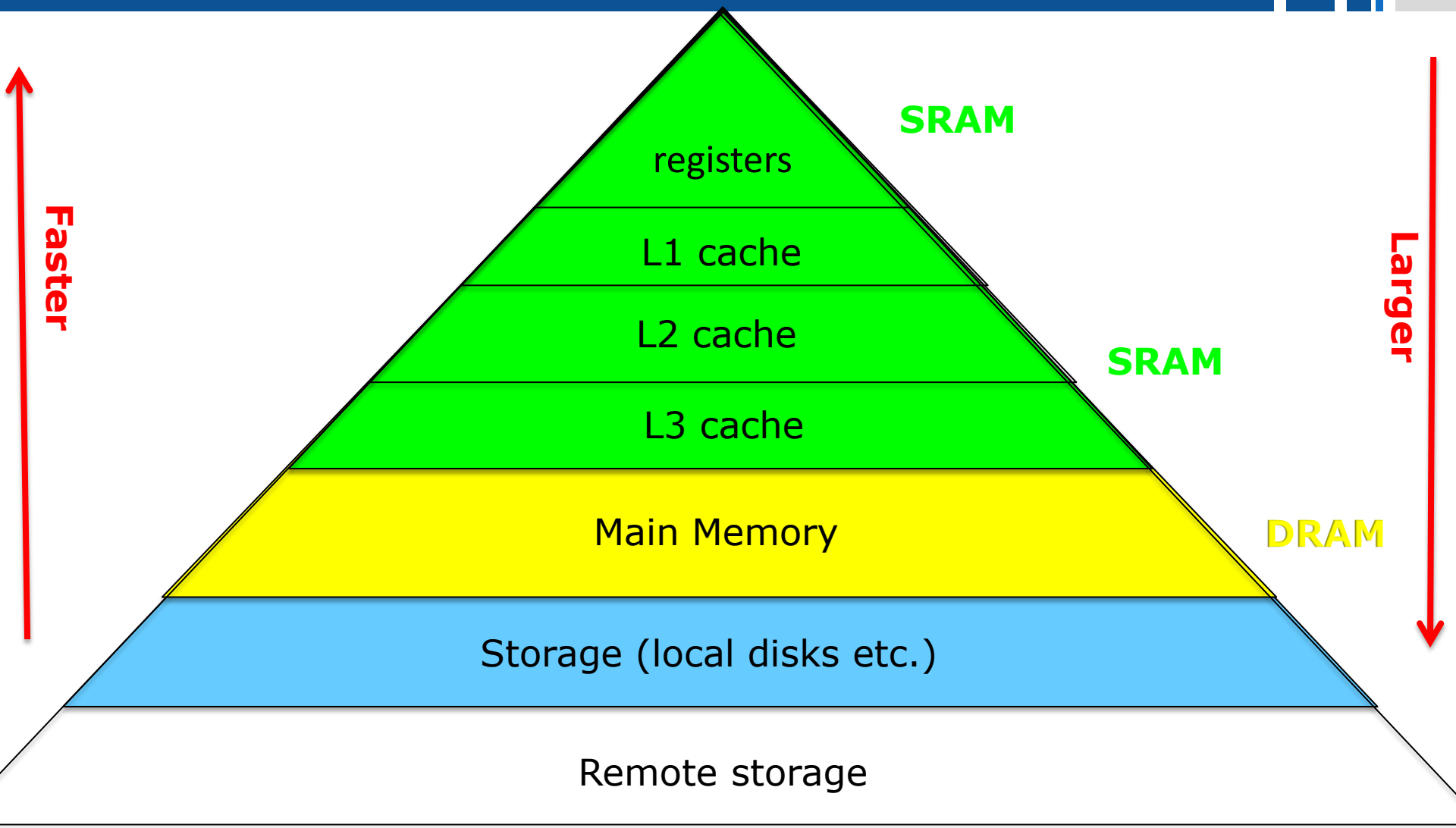
If we a are high to access its neighbors.

- Data: arrays
- Instructions: sequential execution

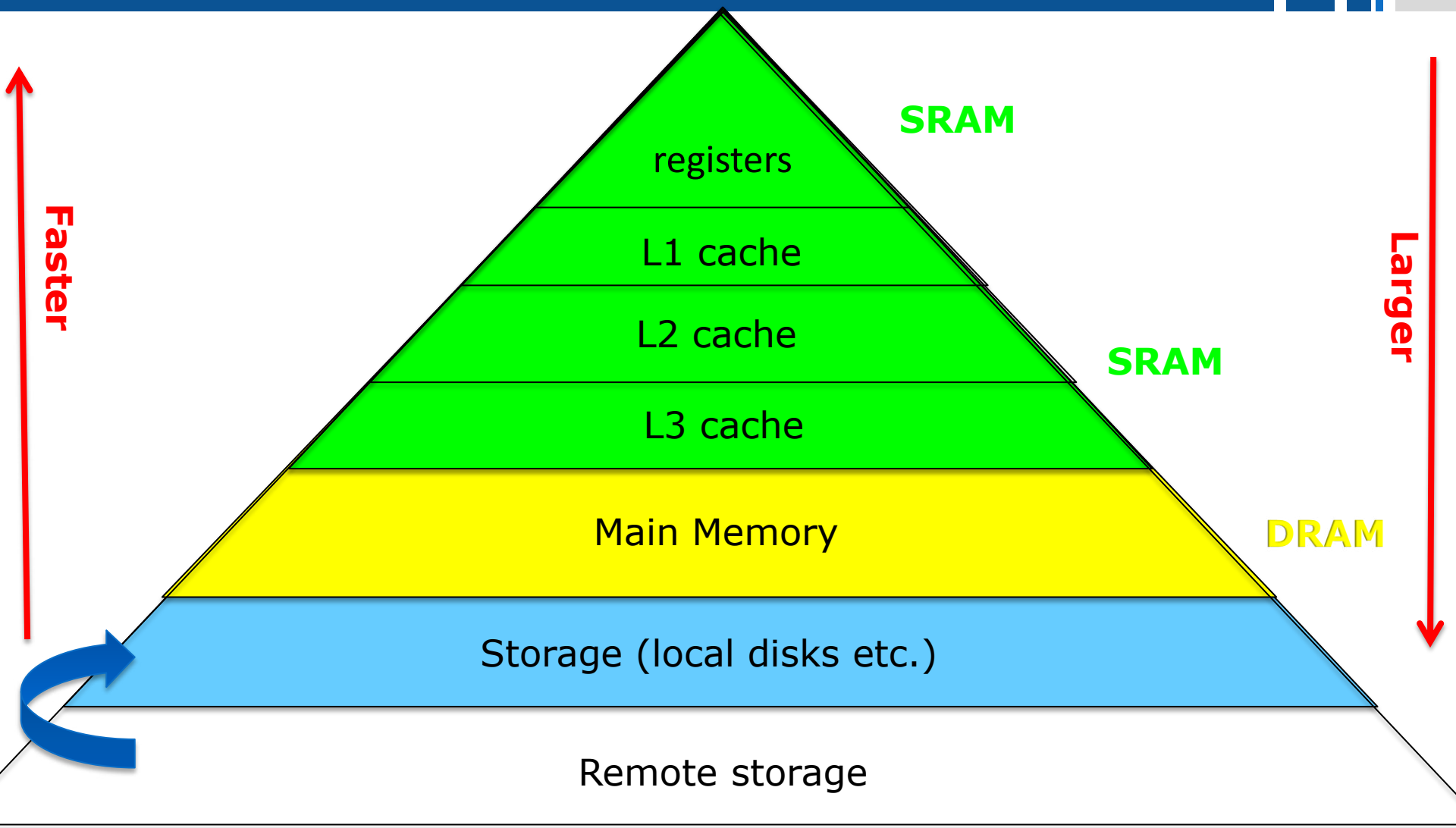
Caching in the Memory Hierarchy



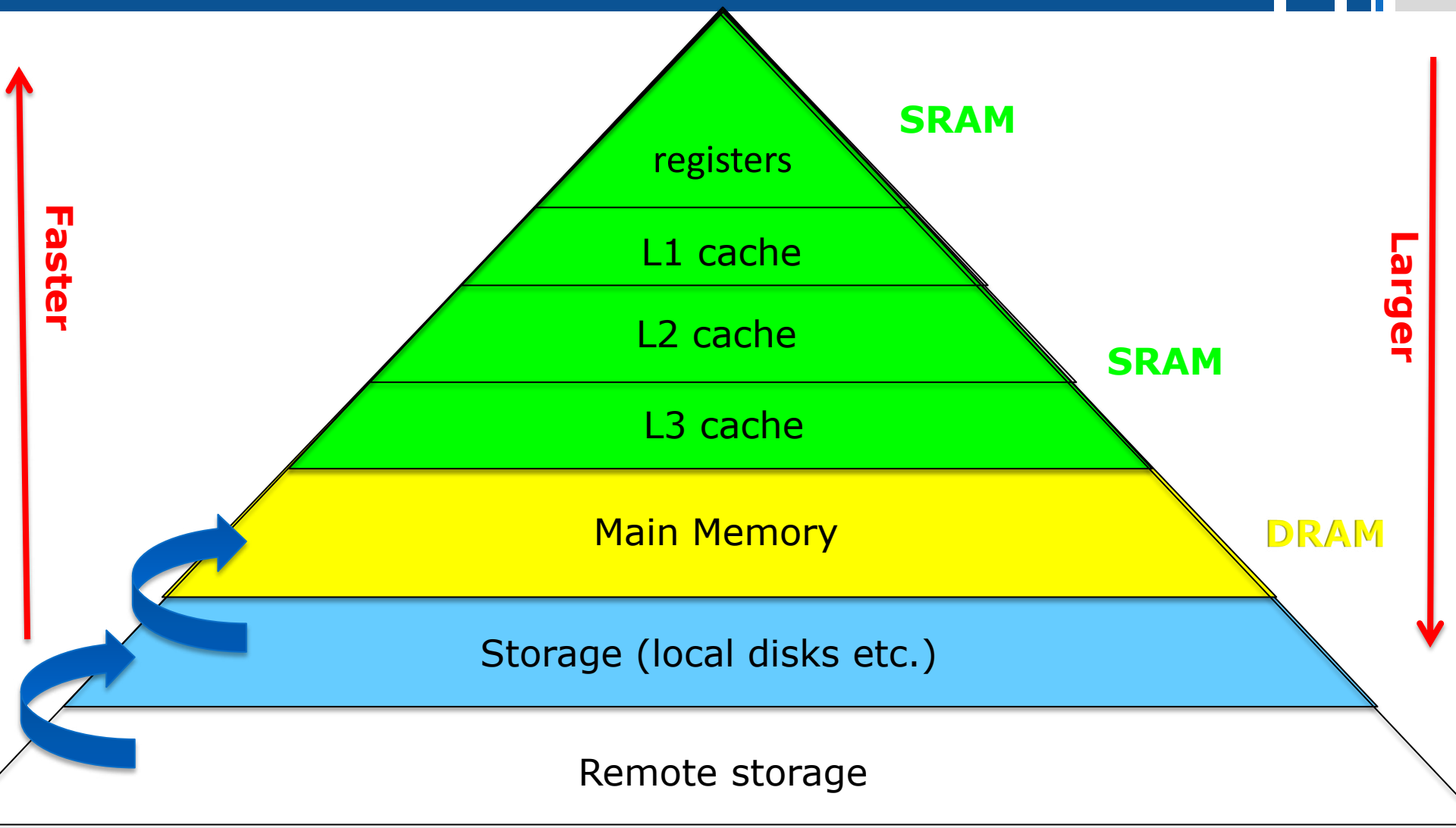
Caching in the Memory Hierarchy



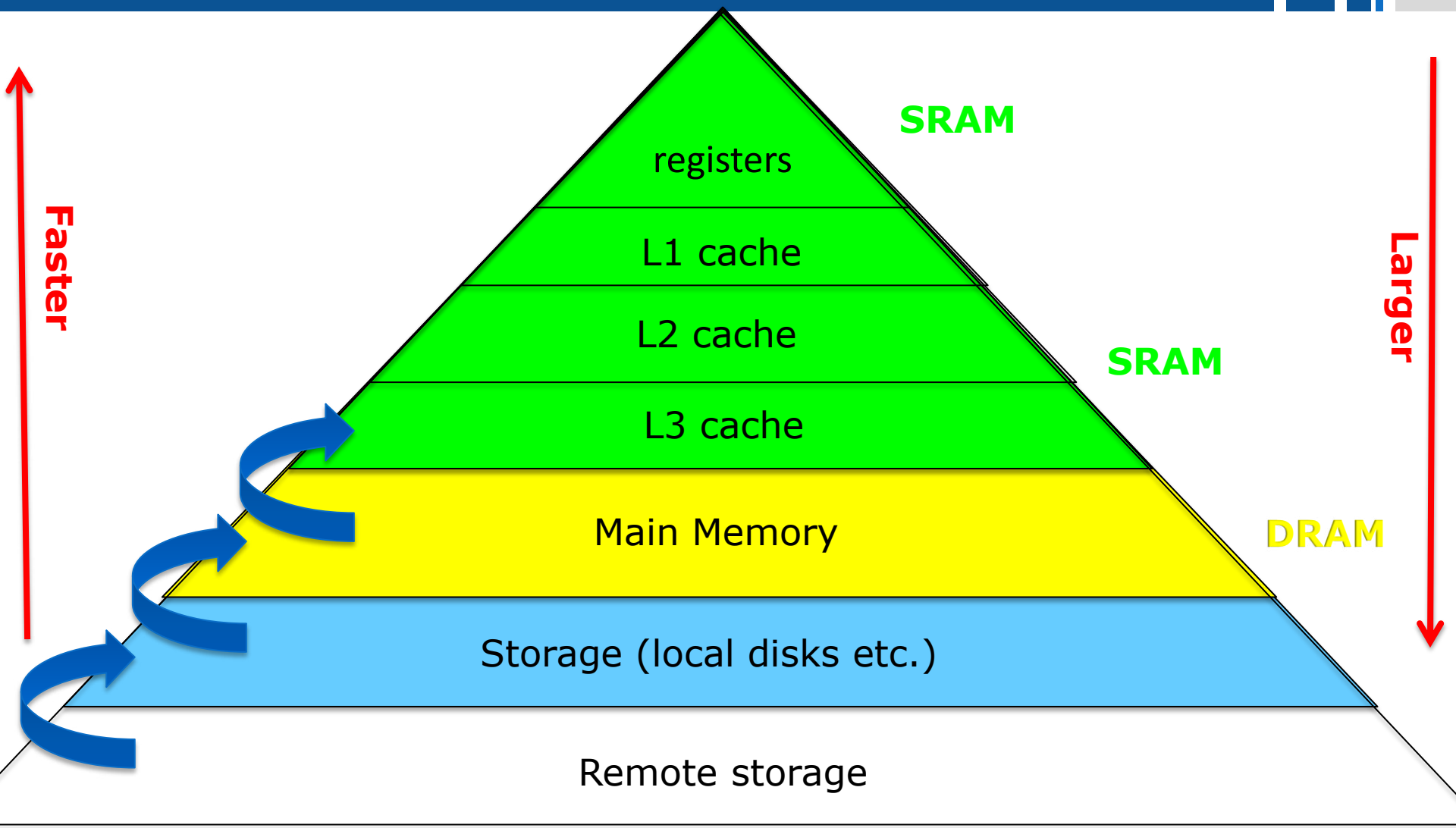
Caching in the Memory Hierarchy



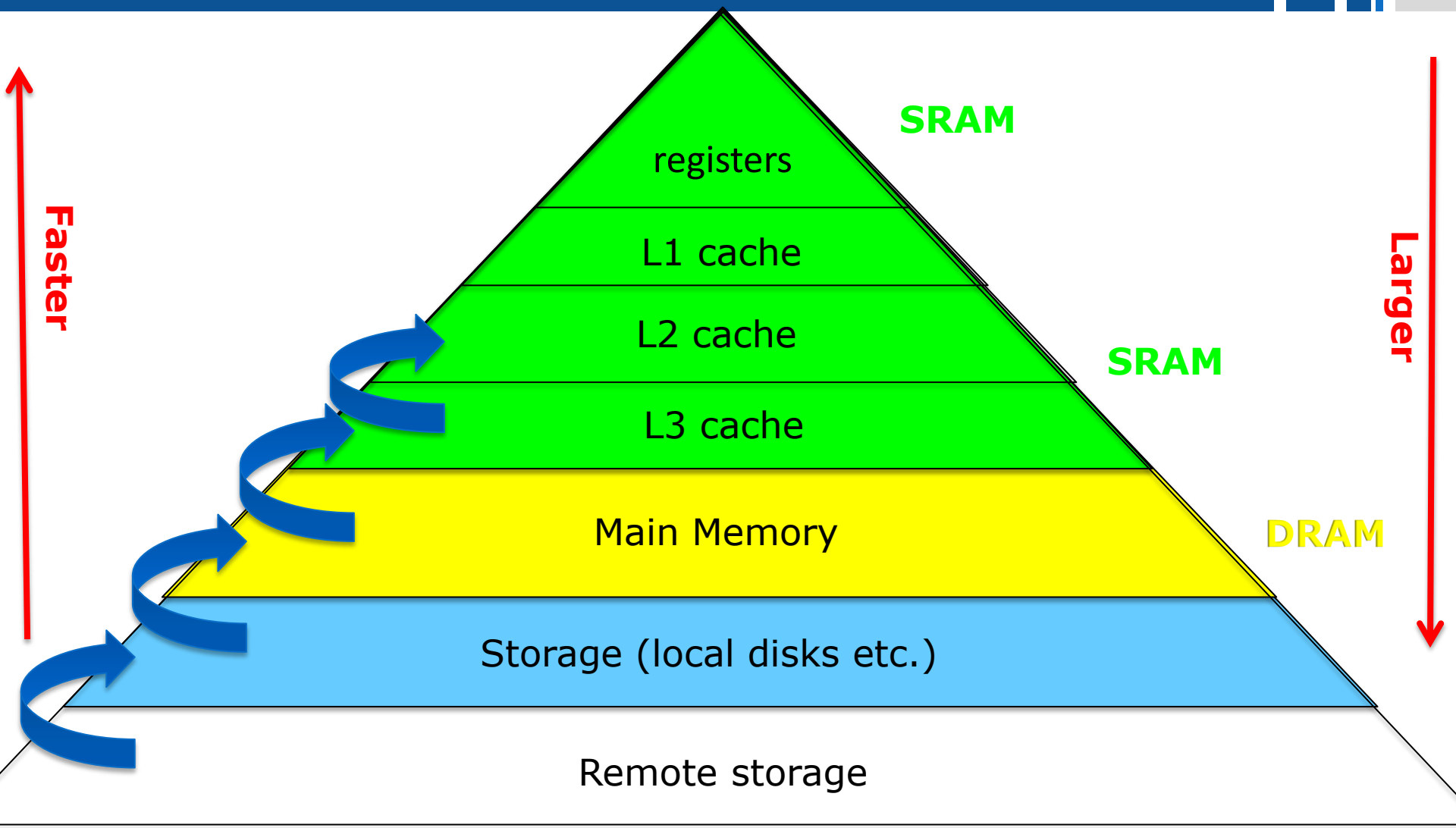
Caching in the Memory Hierarchy



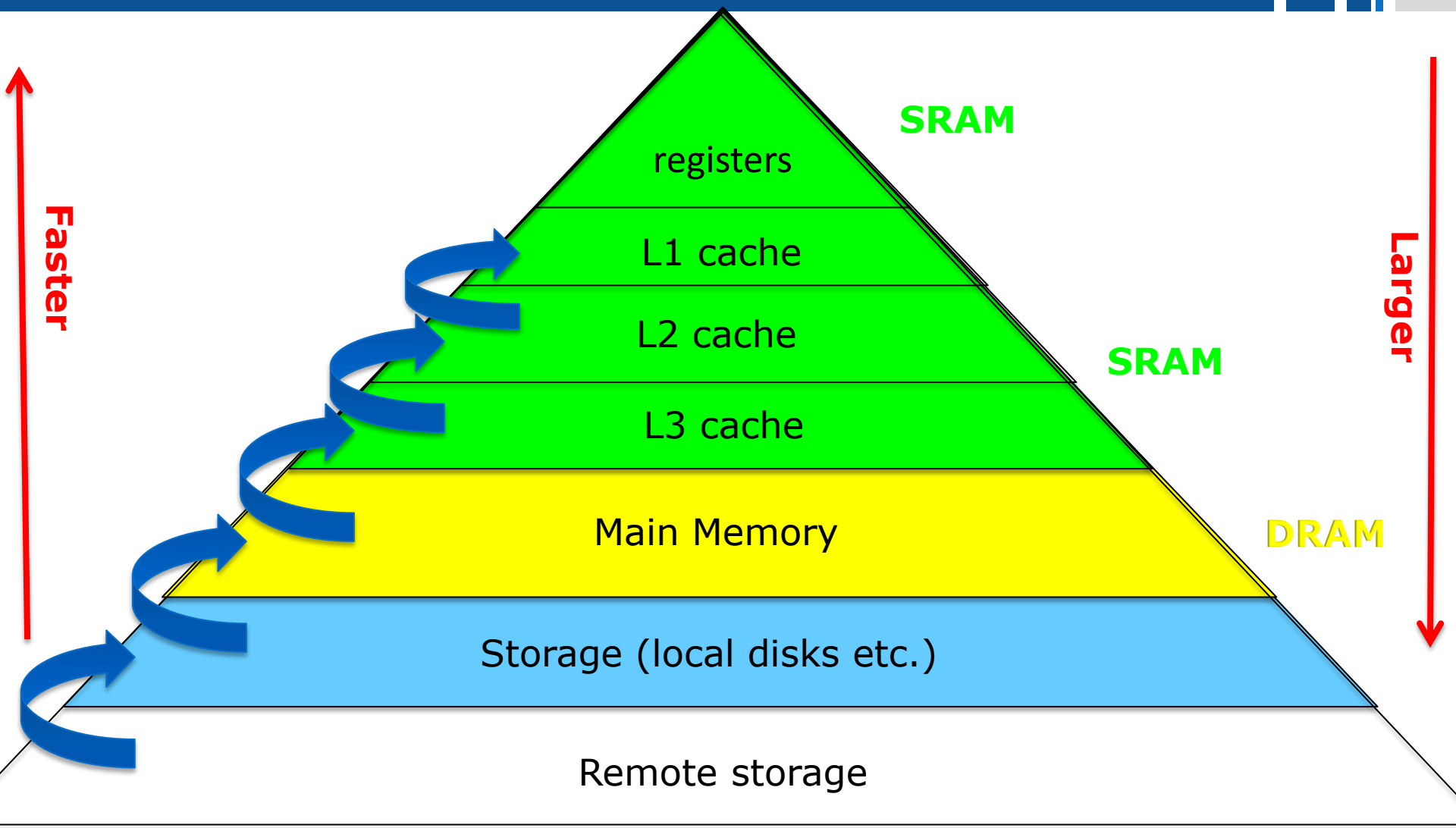
Caching in the Memory Hierarchy



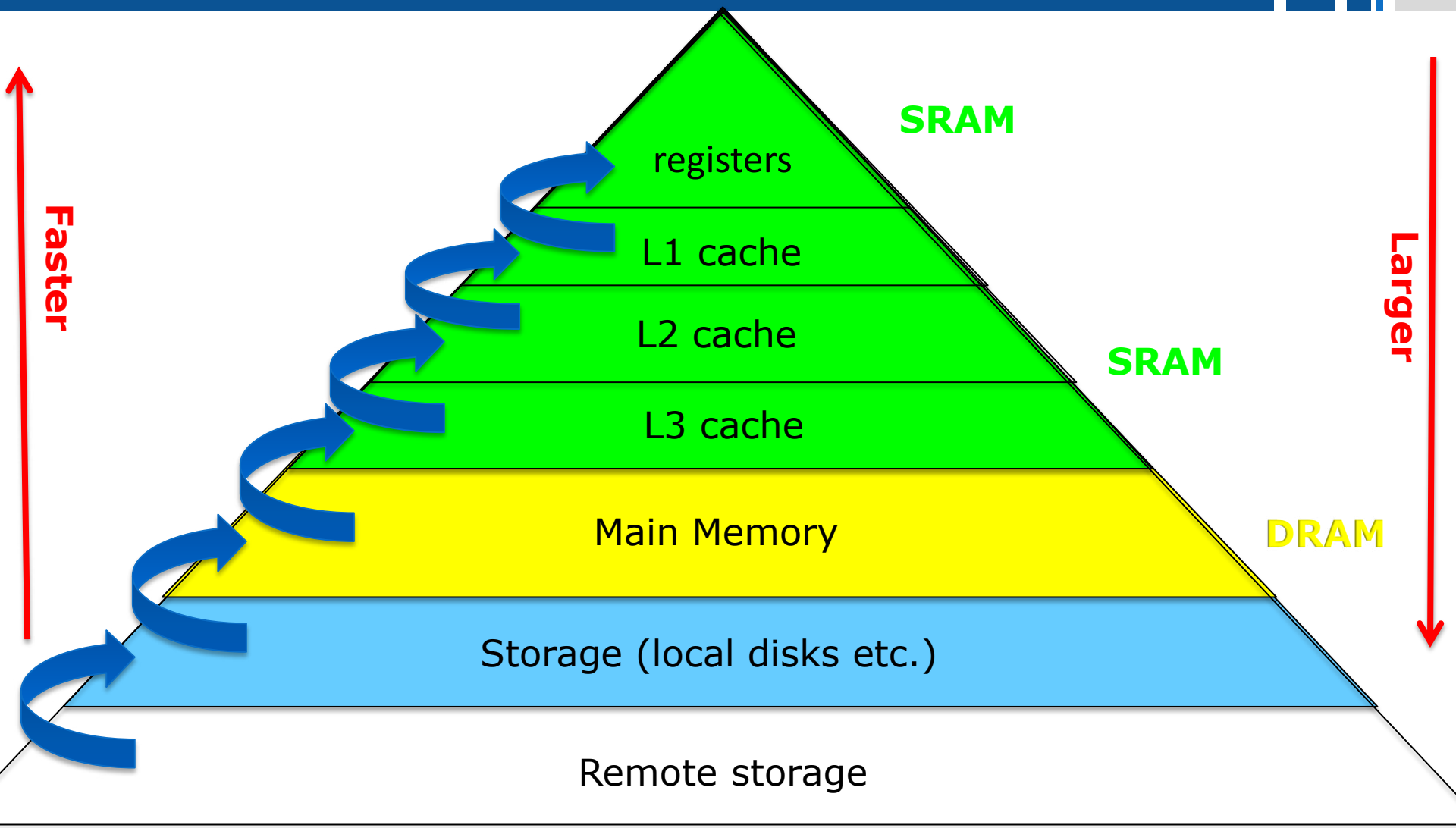
Caching in the Memory Hierarchy



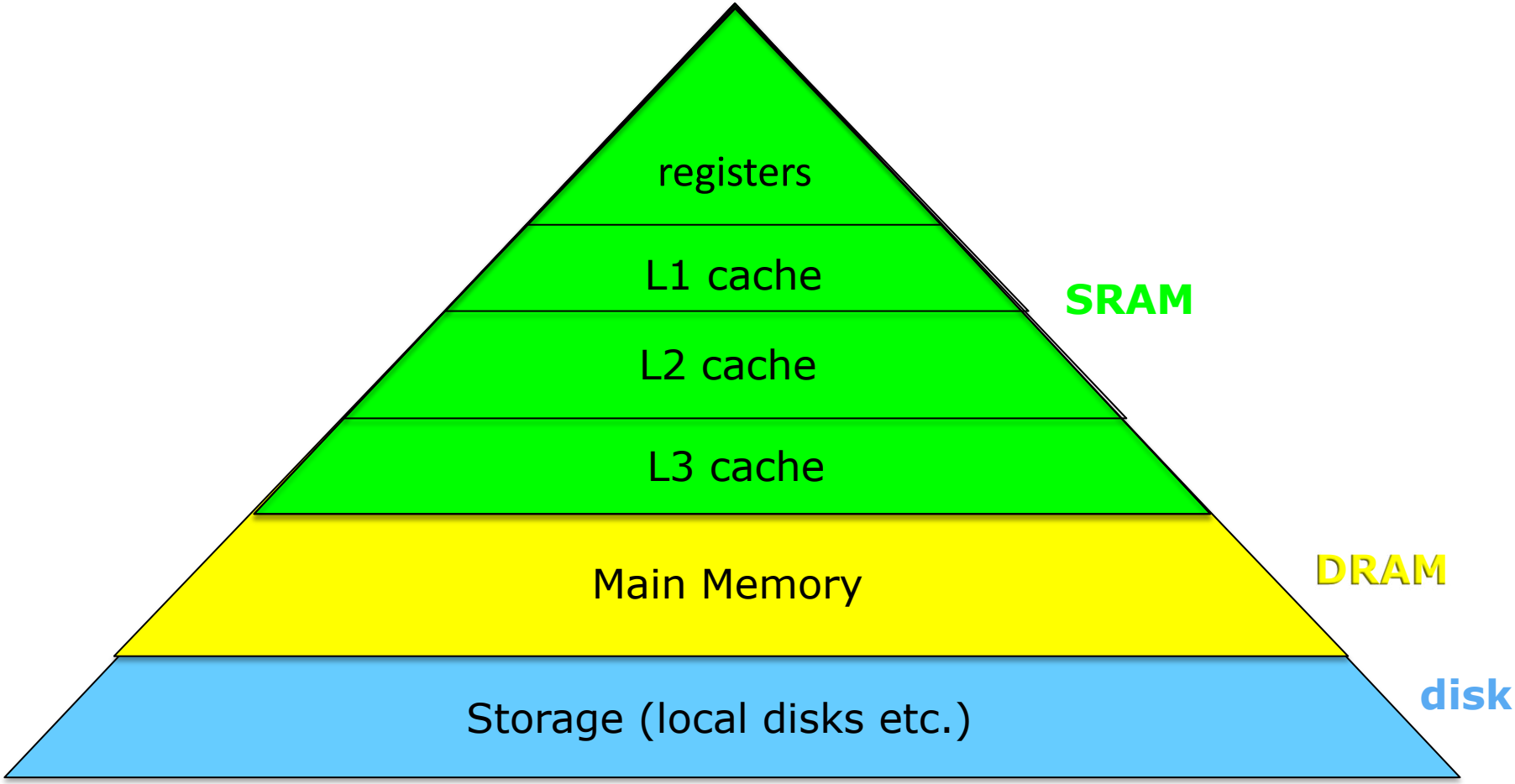
Caching in the Memory Hierarchy



Caching in the Memory Hierarchy



Technology



Remote storage

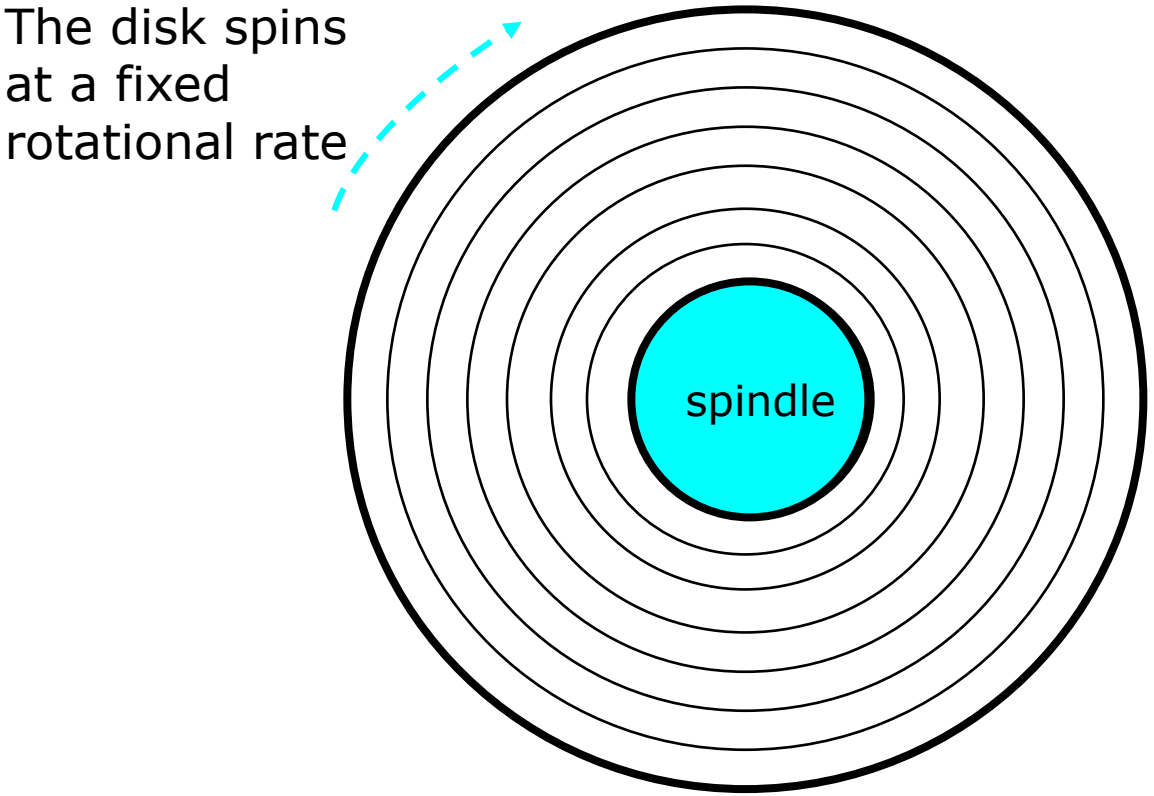
Random-Access Memory (RAM)

- Static RAM (SRAM)
  - Retains value indefinitely, as long as it is kept powered
  - Relatively insensitive to disturbances such as electrical noise
  - Faster and more expensive than DRAM
  - Used for registers and caches
- Dynamic RAM (DRAM)
  - Value must be refreshed every 10-100 ms
  - Sensitive to disturbances
  - Slower and cheaper than SRAM
  - Used for main memory

Memory	Single-chip Capacity	$/chip	$/MByte	Access speed (ns)	Watts/chip	Watts/MByte
DRAM	128MB	$10-$20	$0.08-$0.16	40-80	1-2	0.008-0.016
SRAM	9MB	$50-$70	$5.5-$7.8	3-5	1.5-3	0.17-0.33

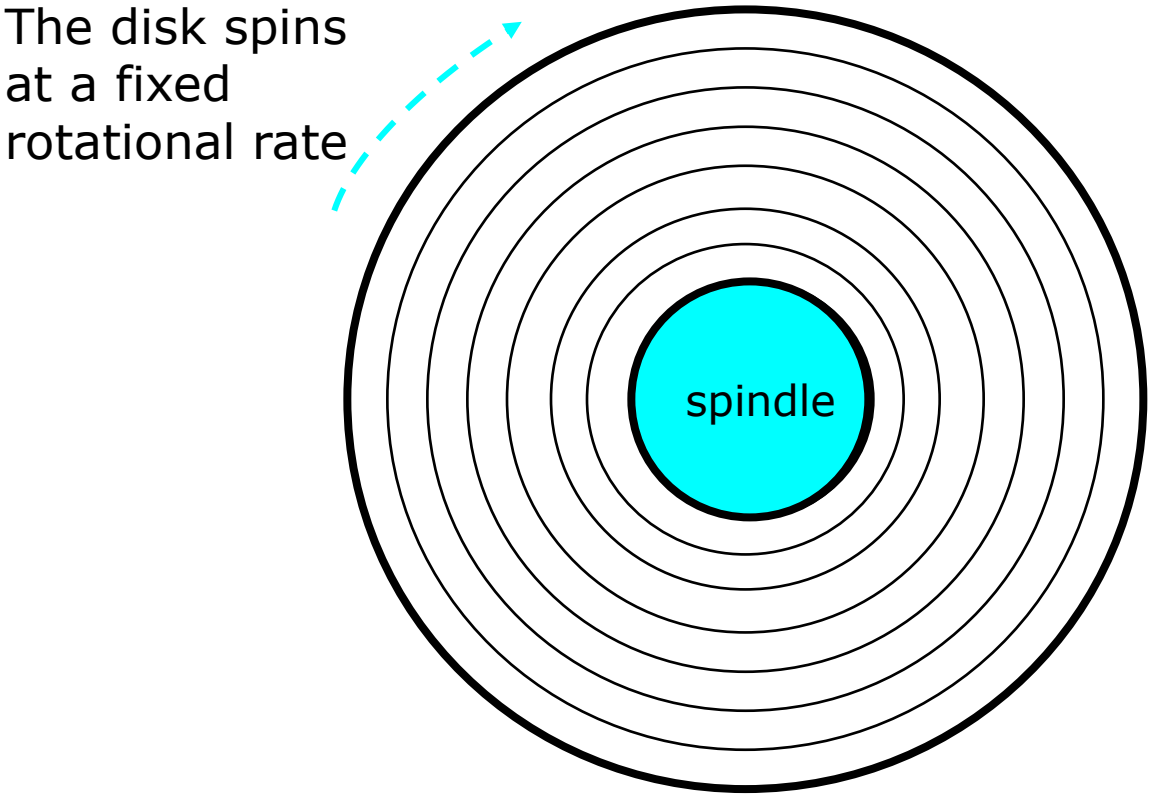
2010 numbers

Disk Operation (Single-Platter View)



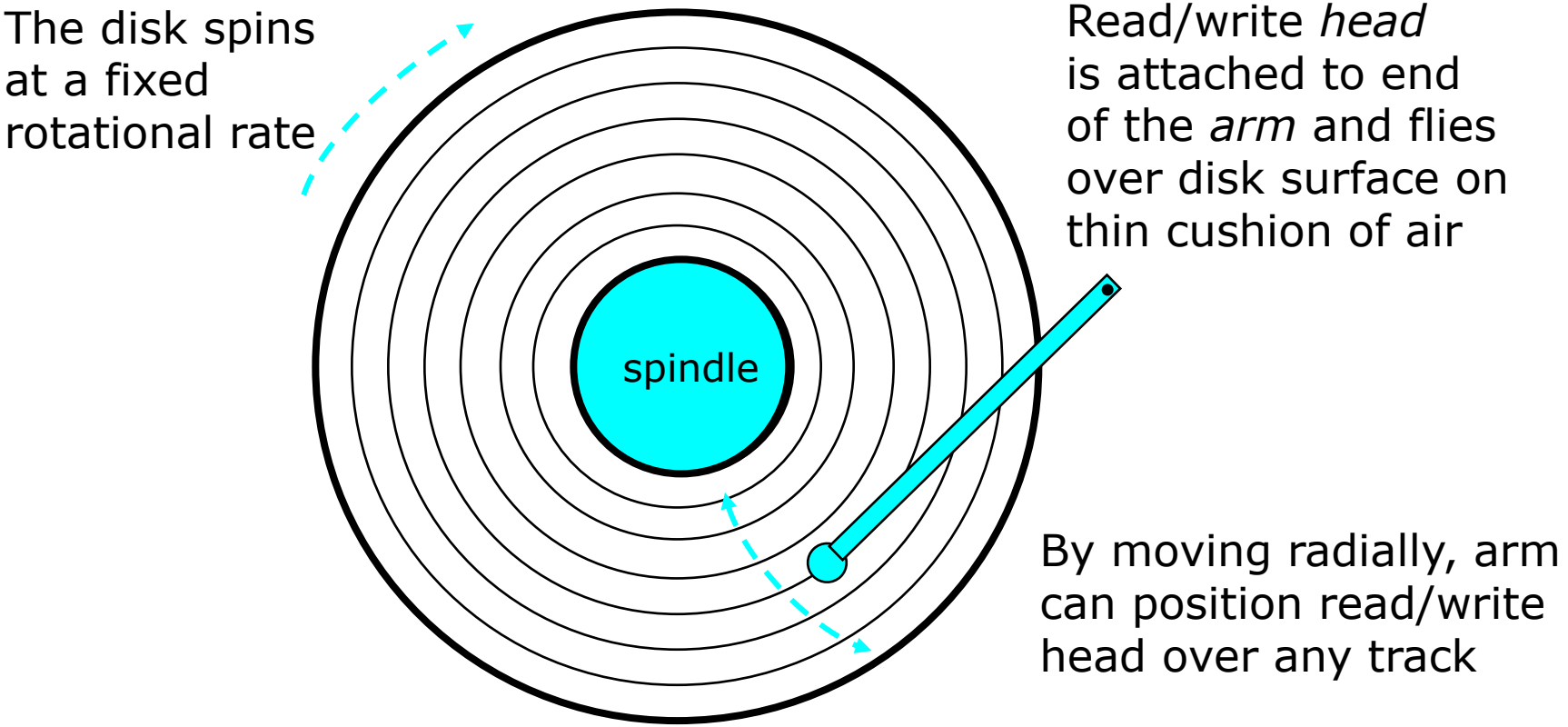
- Disk access time is **much** slower than DRAM
- **Fast** disks have access time of 5-10 **milliseconds**
  - Seek time and rotation delay are roughly equal

Disk Operation (Single-Platter View)



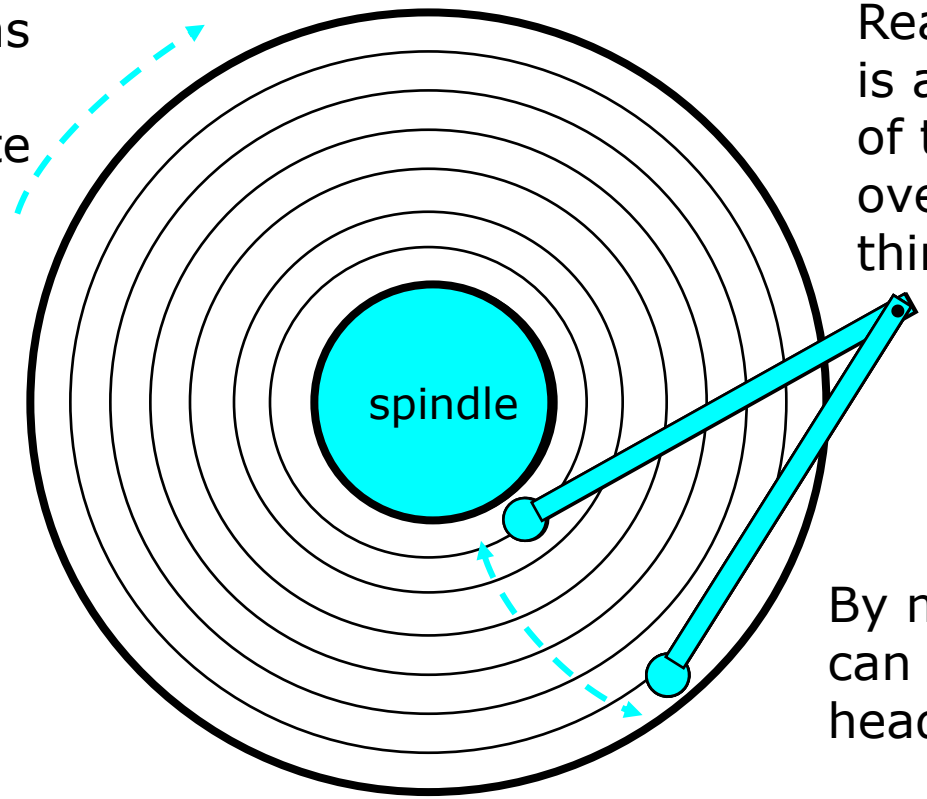
- Disk access time is **much** slower than DRAM
- **Fast** disks have access time of 5-10 **milliseconds**
  - Seek time and rotation delay are roughly equal

Disk Operation (Single-Platter View)



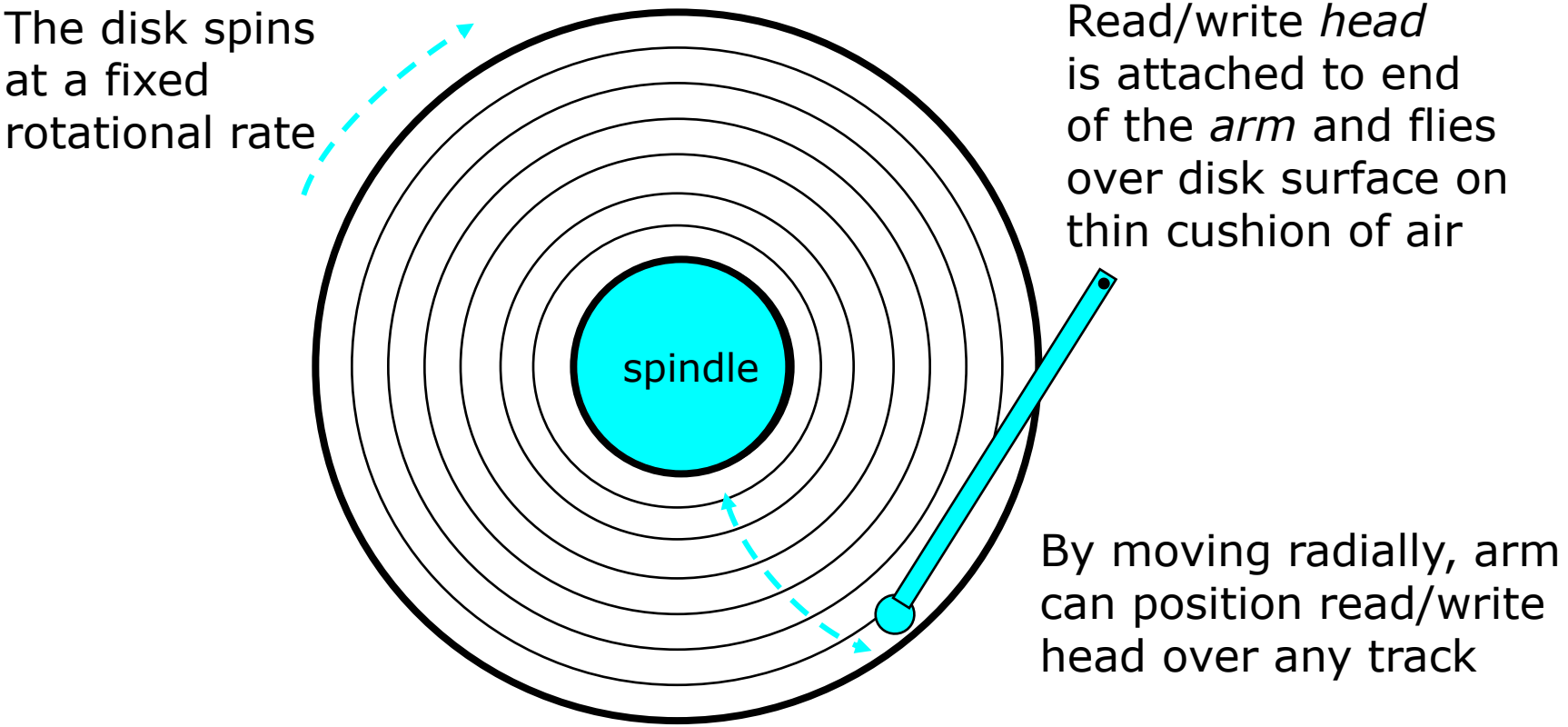
- Disk access time is **much** slower than DRAM
- **Fast** disks have access time of 5-10 **milliseconds**
  - Seek time and rotation delay are roughly equal

Disk Operation (Single-Platter View)



- Disk access time is **much** slower than DRAM
- **Fast** disks have access time of 5-10 **milliseconds**
  - Seek time and rotation delay are roughly equal

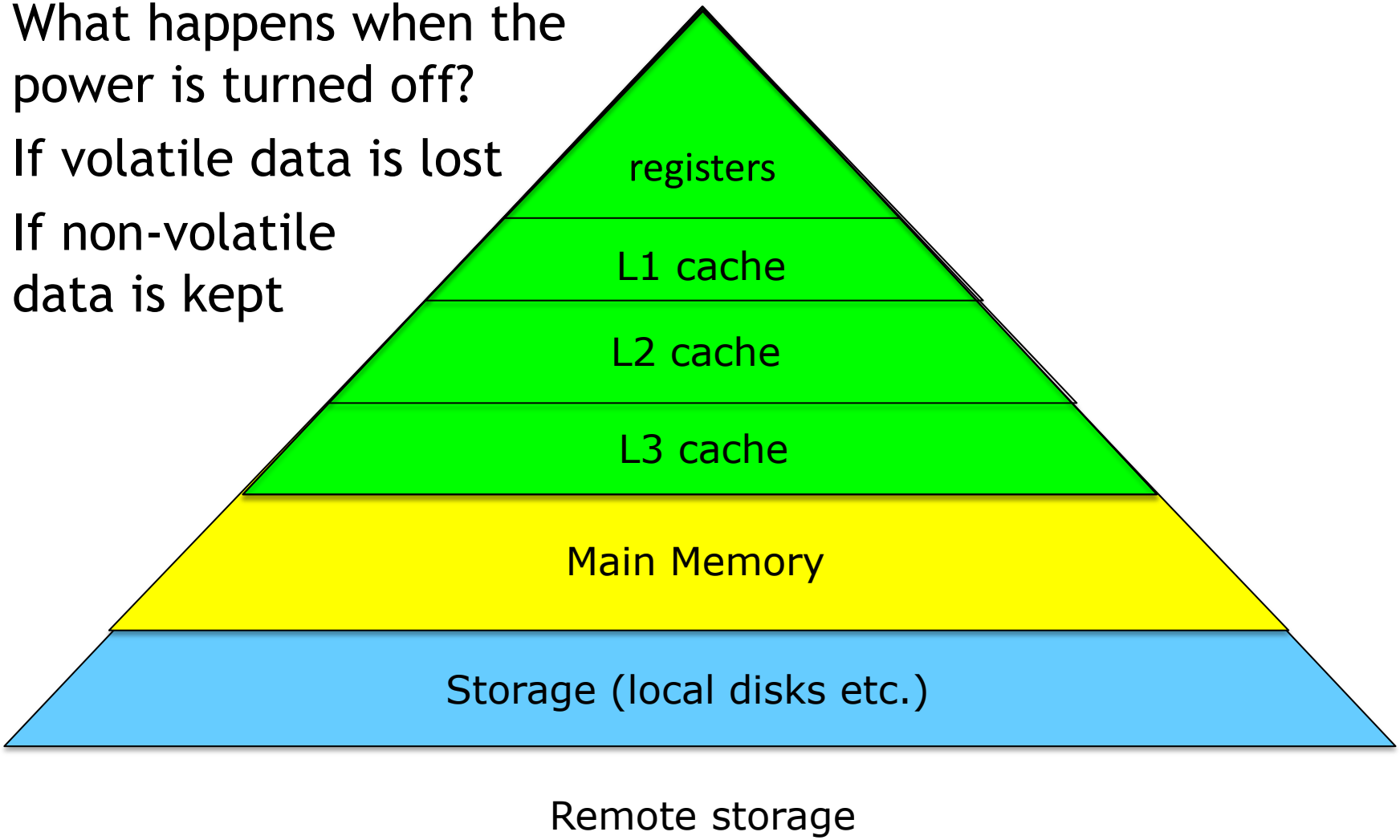
Disk Operation (Single-Platter View)



- Disk access time is **much** slower than DRAM
- **Fast** disks have access time of 5-10 **milliseconds**
  - Seek time and rotation delay are roughly equal

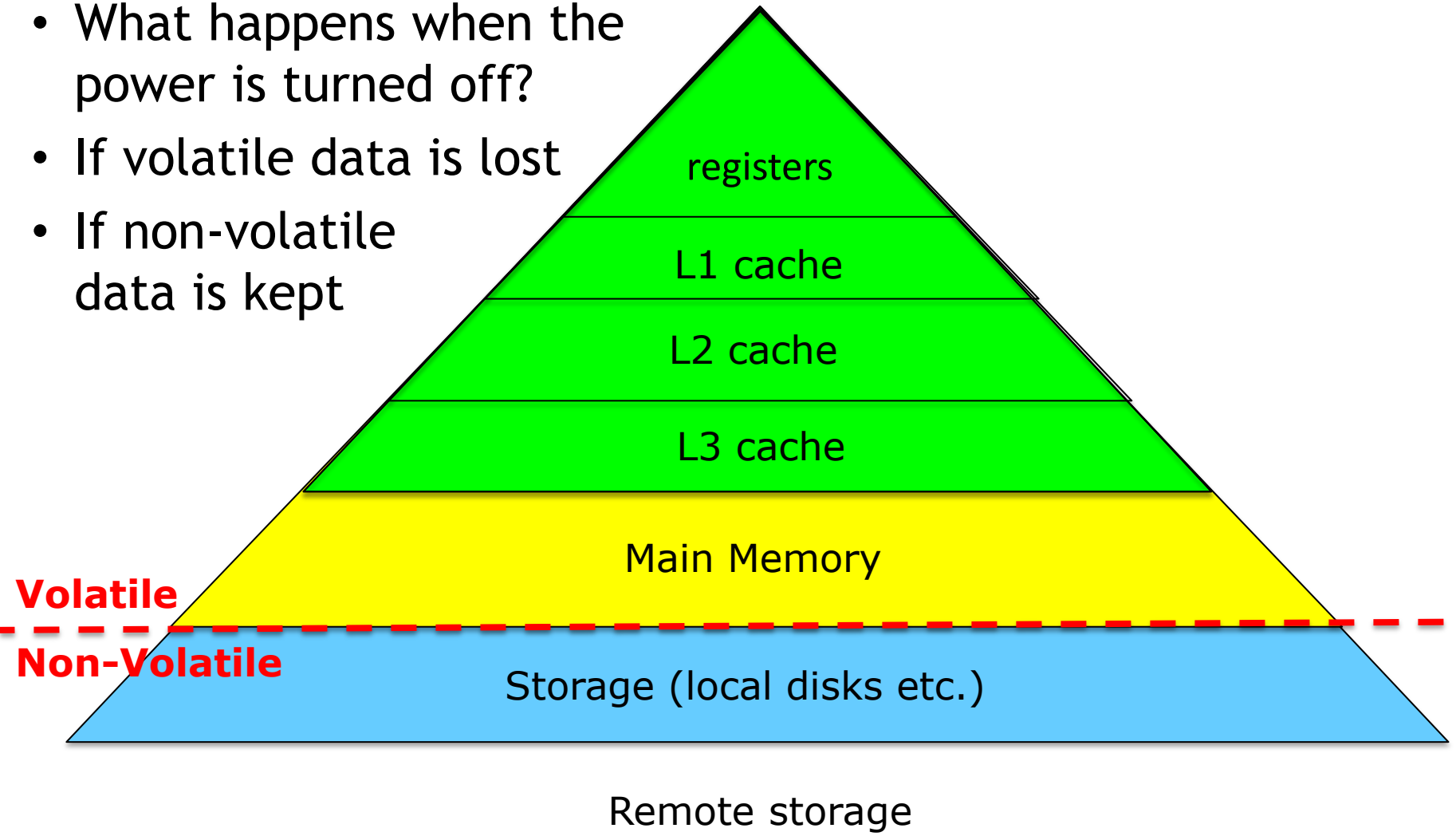
Volatility (volatile = נדיף)

- What happens when the power is turned off?
- If volatile data is lost
- If non-volatile data is kept



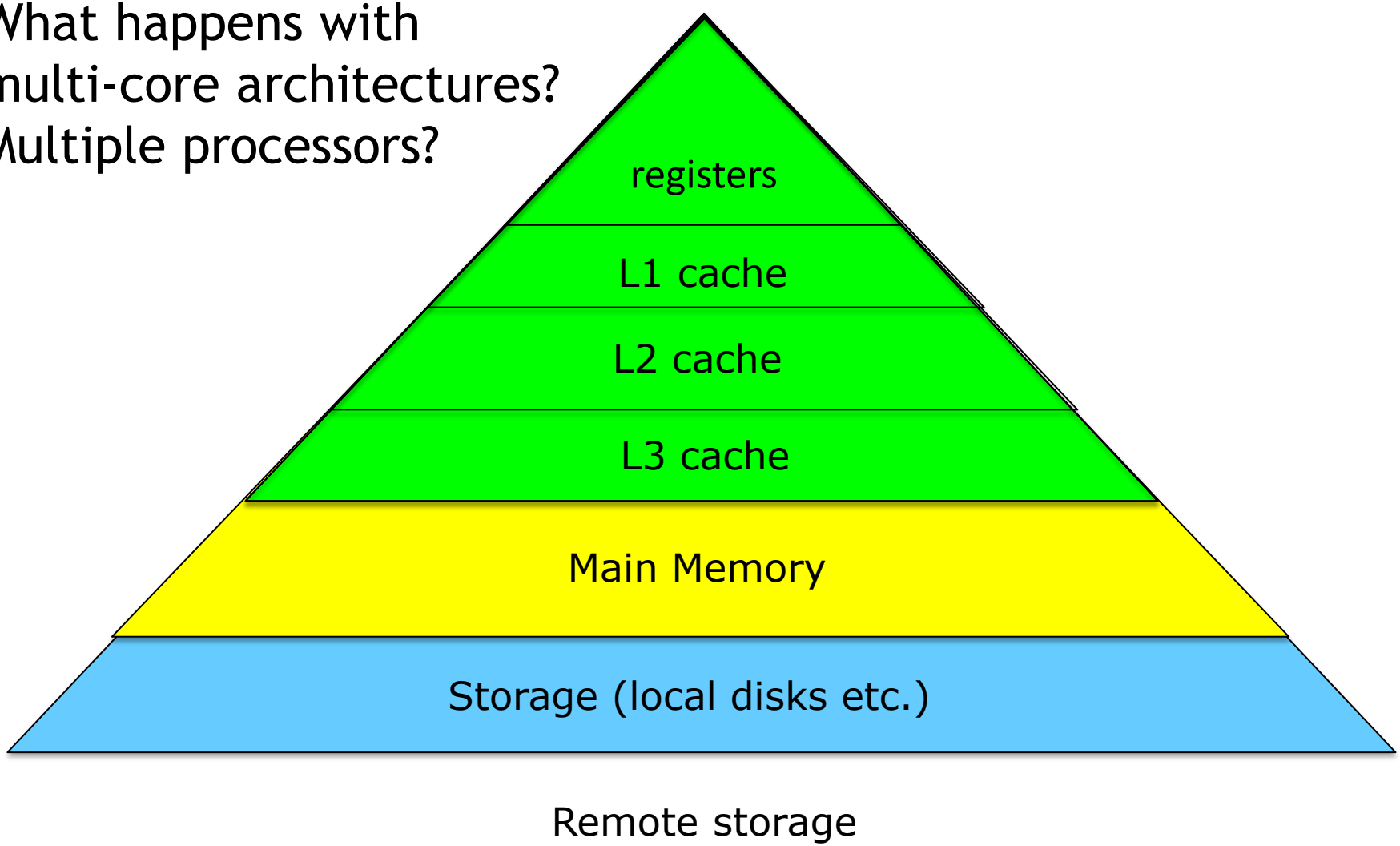
Volatility (volatile = נדיף)

- What happens when the power is turned off?
- If volatile data is lost
- If non-volatile data is kept



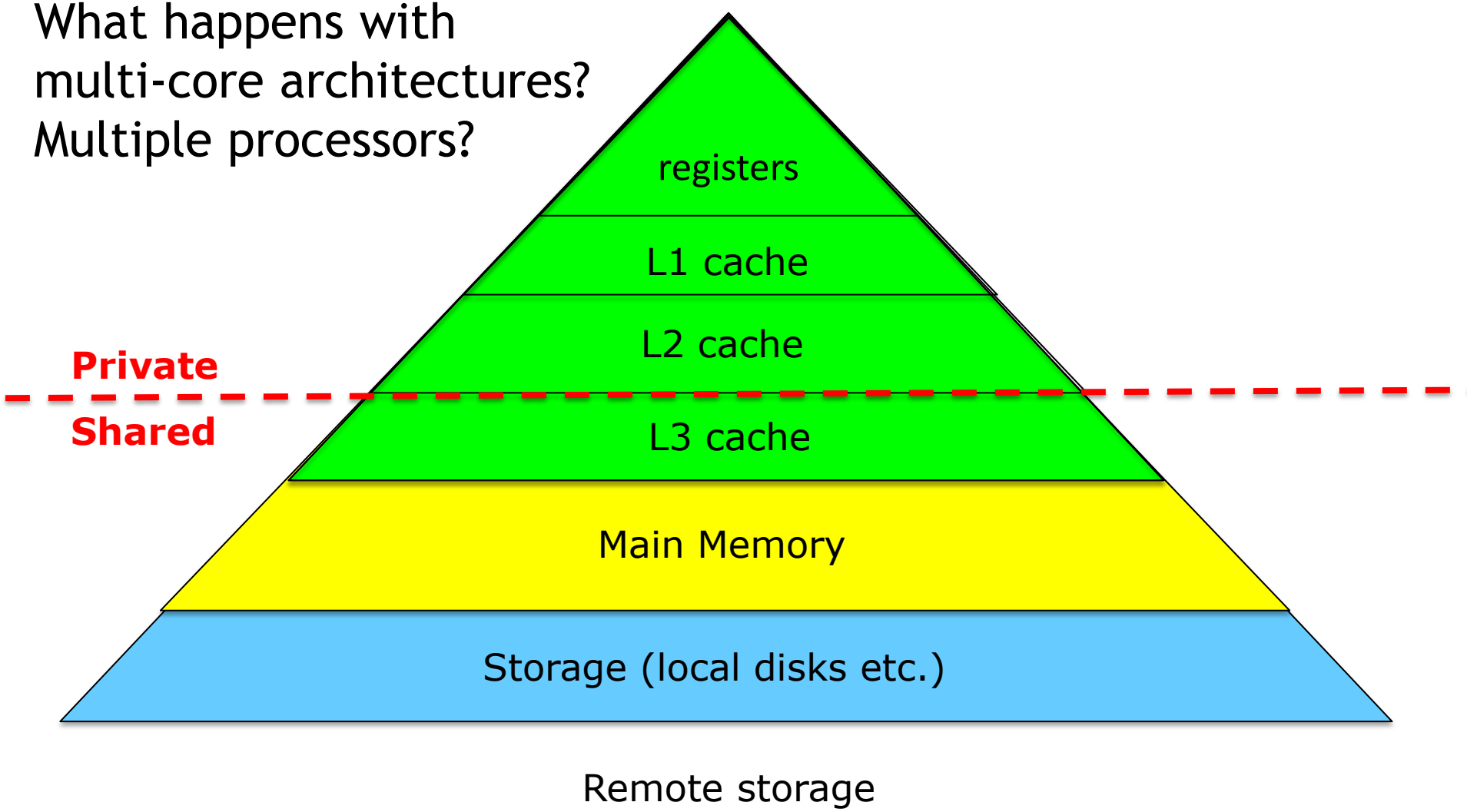
Modern Architectures

What happens with multi-core architectures? Multiple processors?



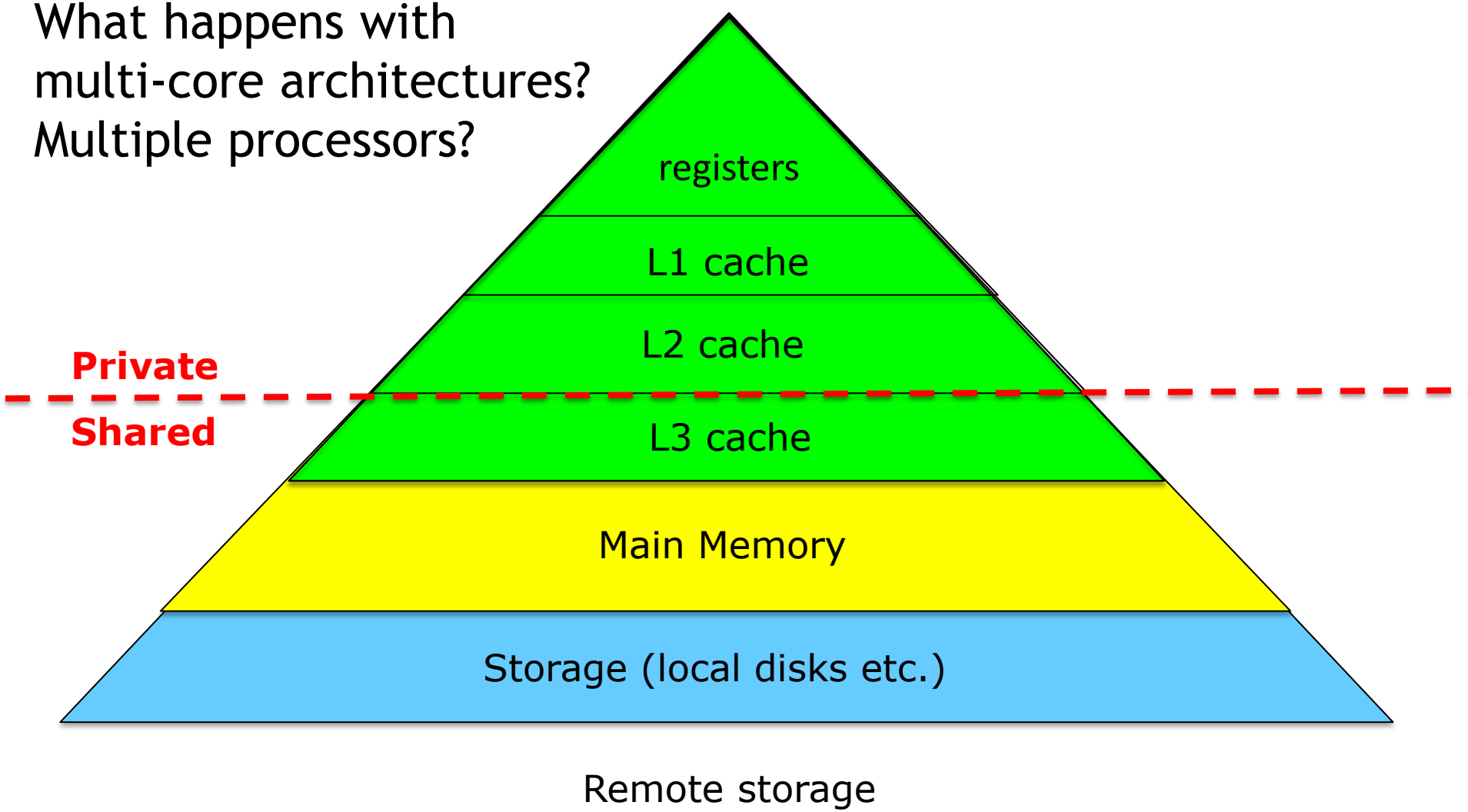
Modern Architectures

What happens with multi-core architectures? Multiple processors?

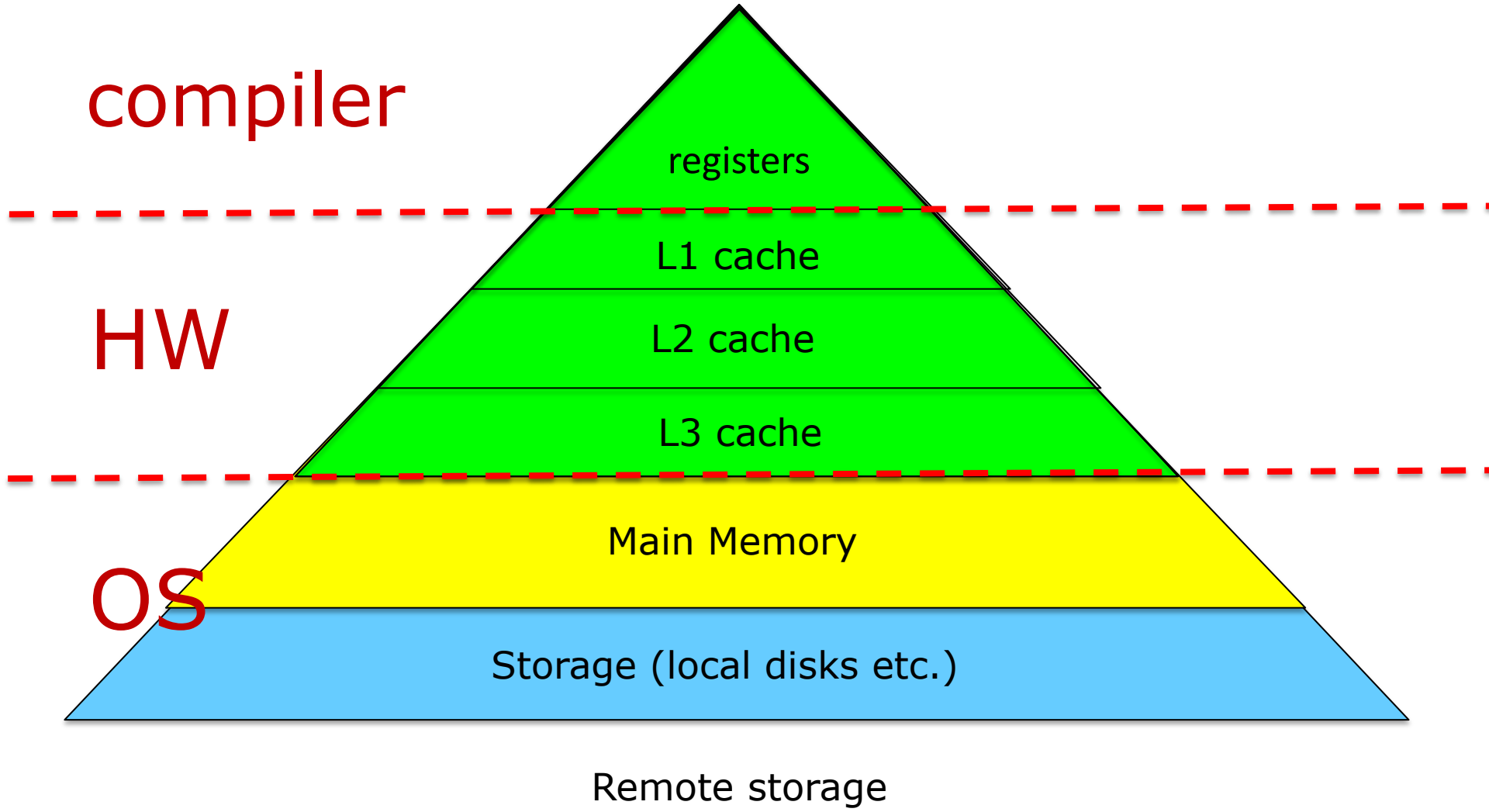


Modern Architectures

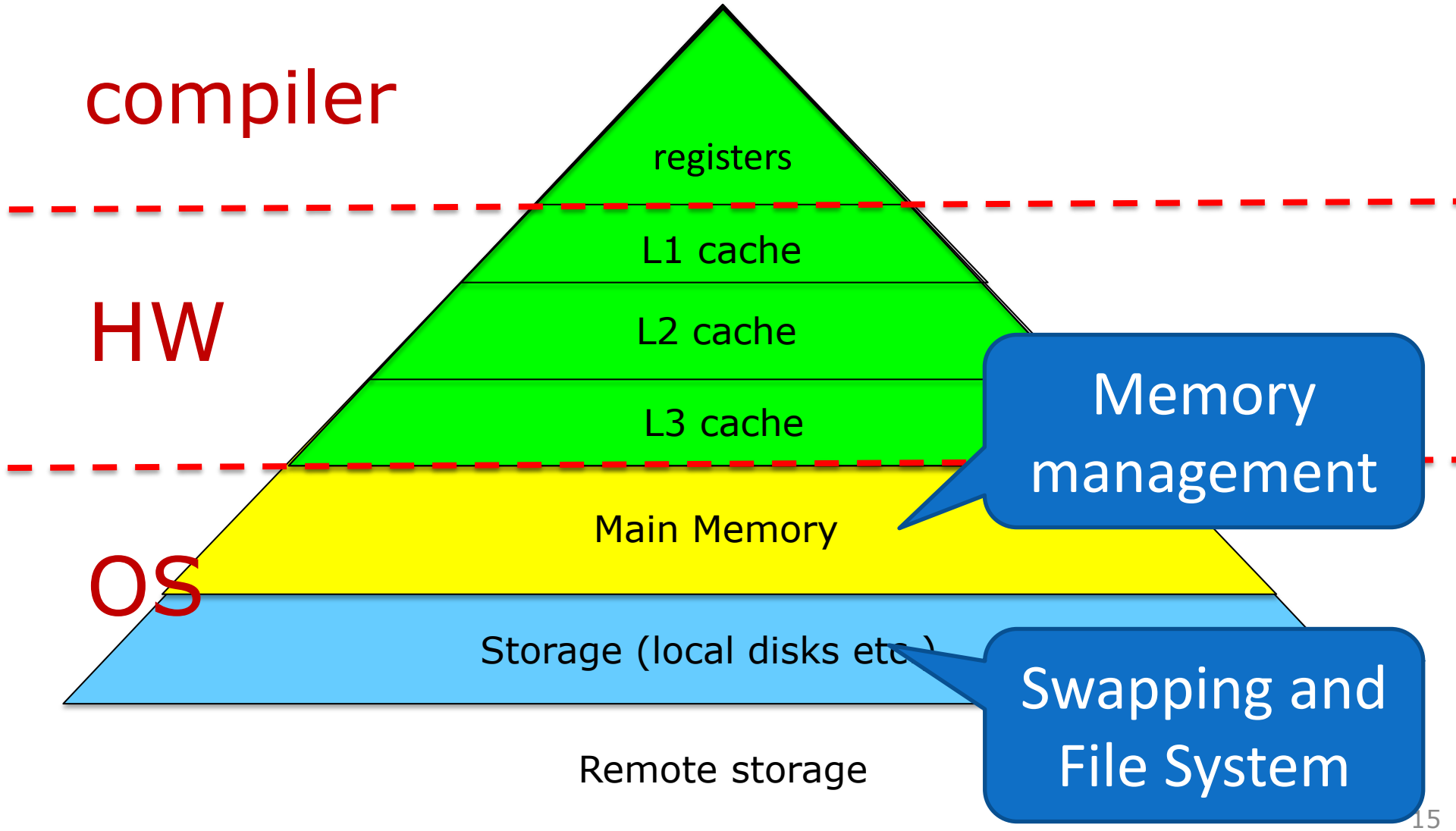
What happens with multi-core architectures? Multiple processors?



Responsibility



Responsibility



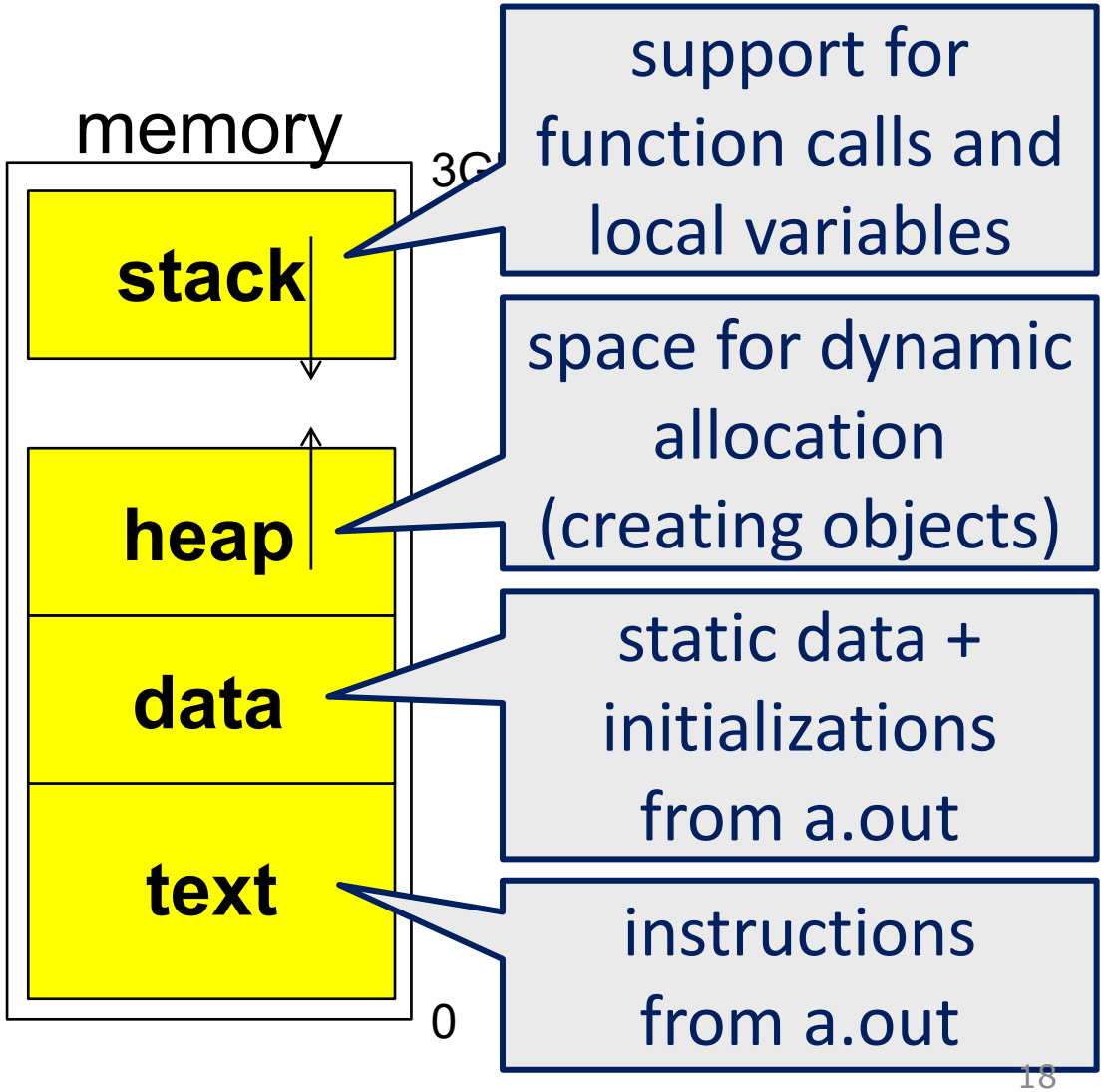
THE ADDRESS SPACE

“Address Space”

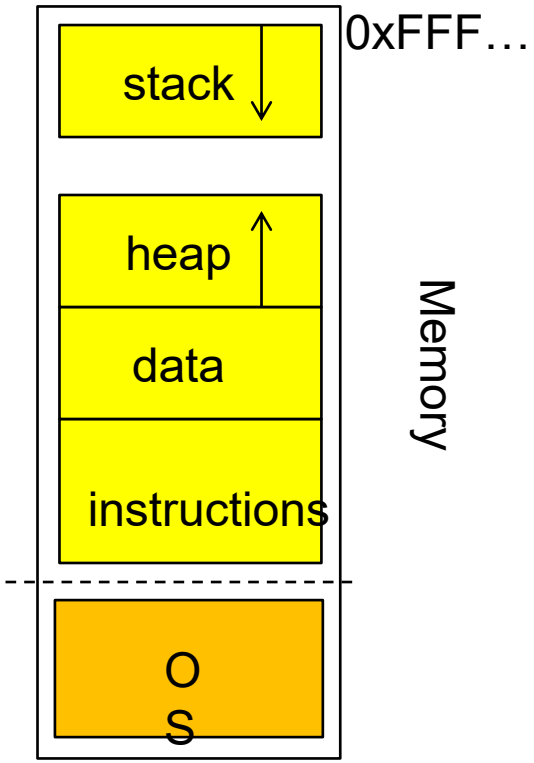
- All memory addressable by the program
- Depends on the architecture
- 32-bit architecture → 4GB memory
- Some set aside for OS
  - Windows left 2GB for user process
  - Linux left 3GB for user process
- This is **for each and every process**

Memory Segments

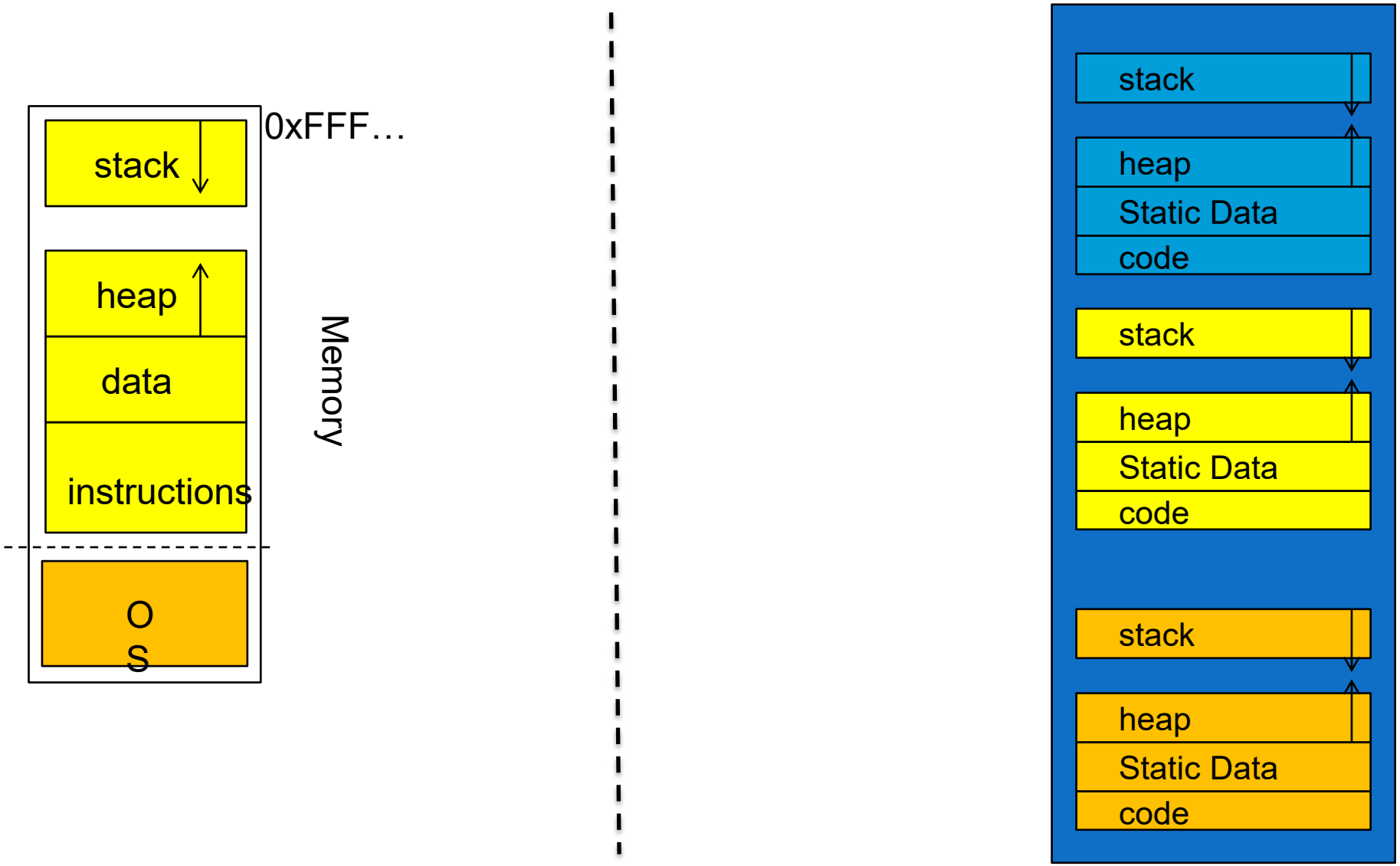
- The address space is divided into 4 segments with different uses
- Stack and heap placed to leave space for growth



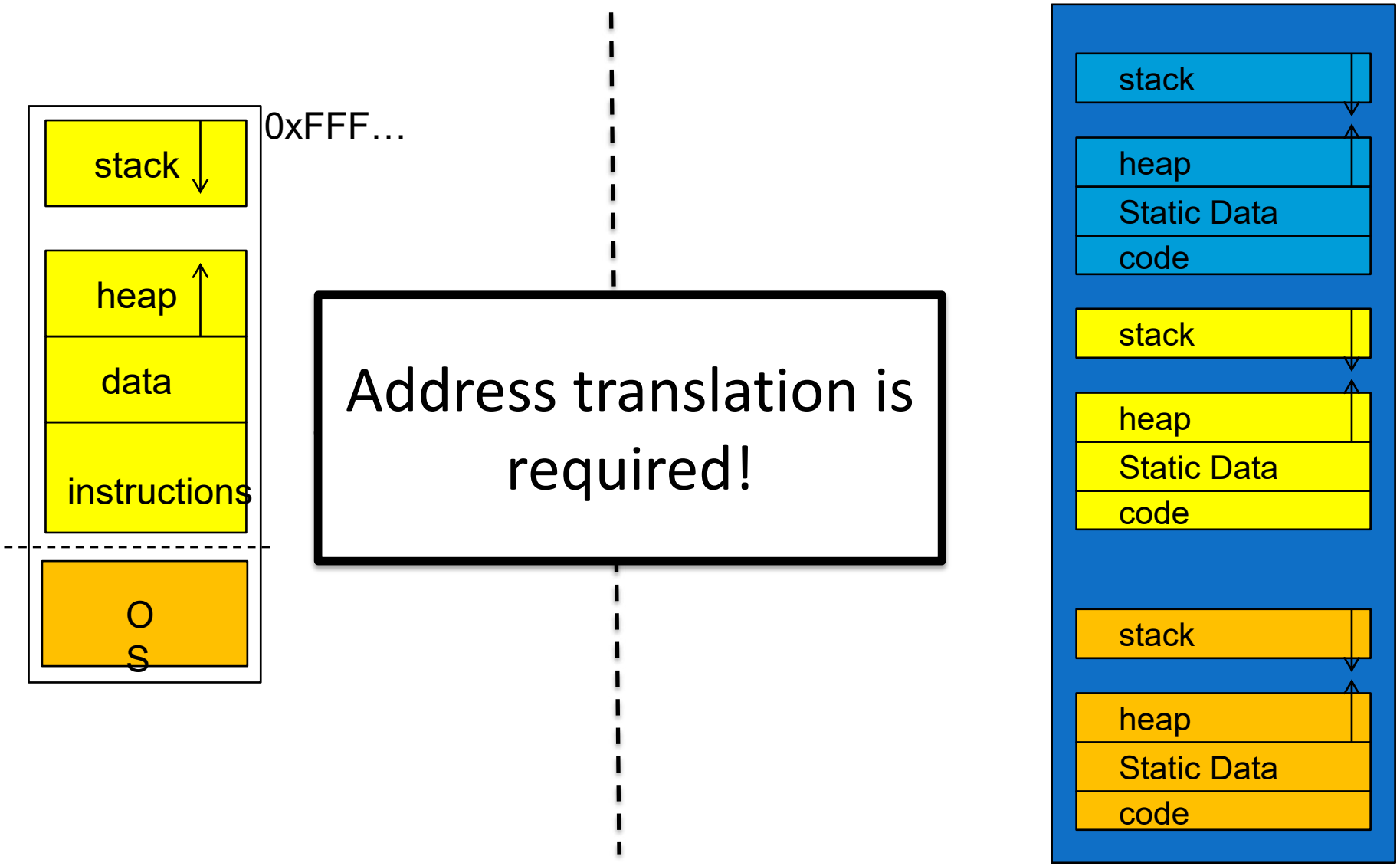
Process View Vs. Reality



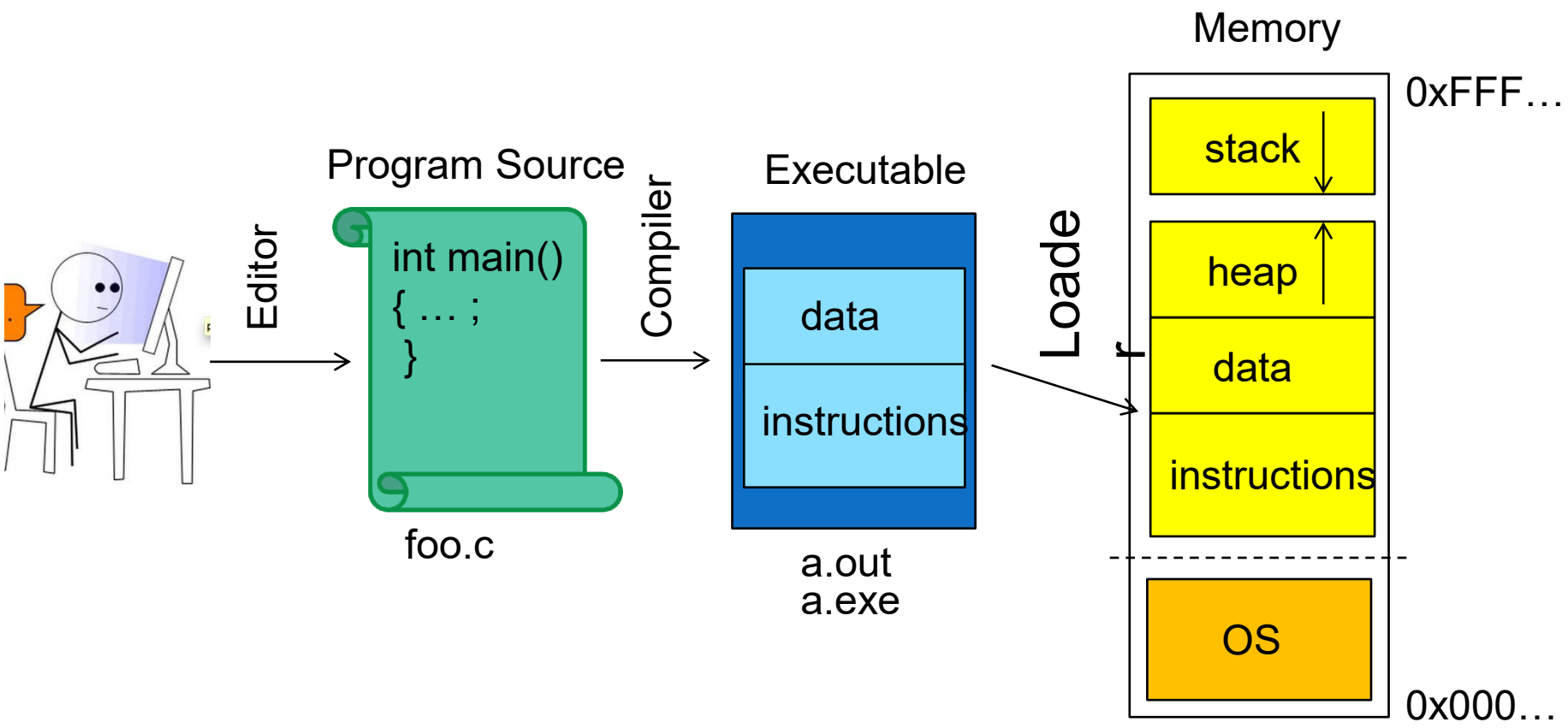
Process View Vs. Reality



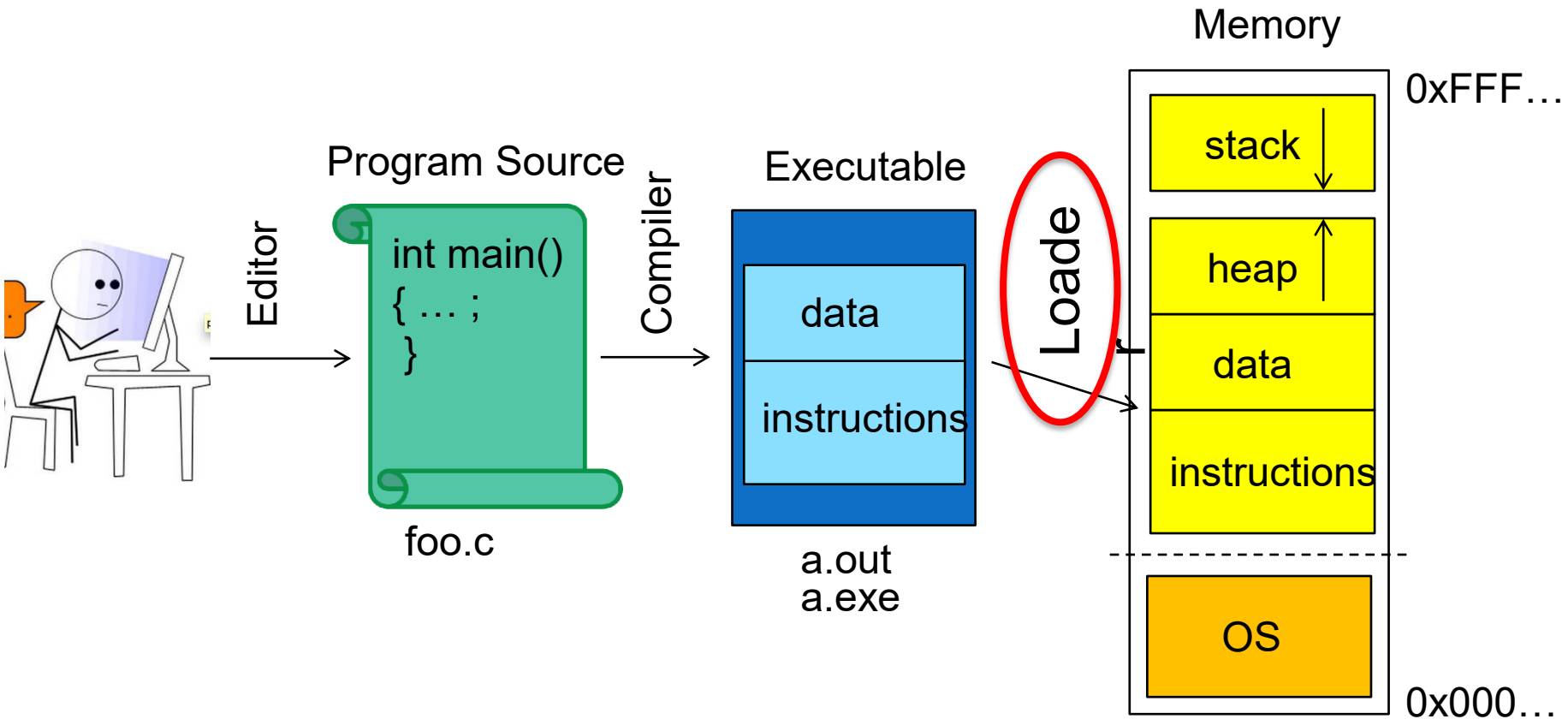
Process View Vs. Reality



When are Addresses Created?

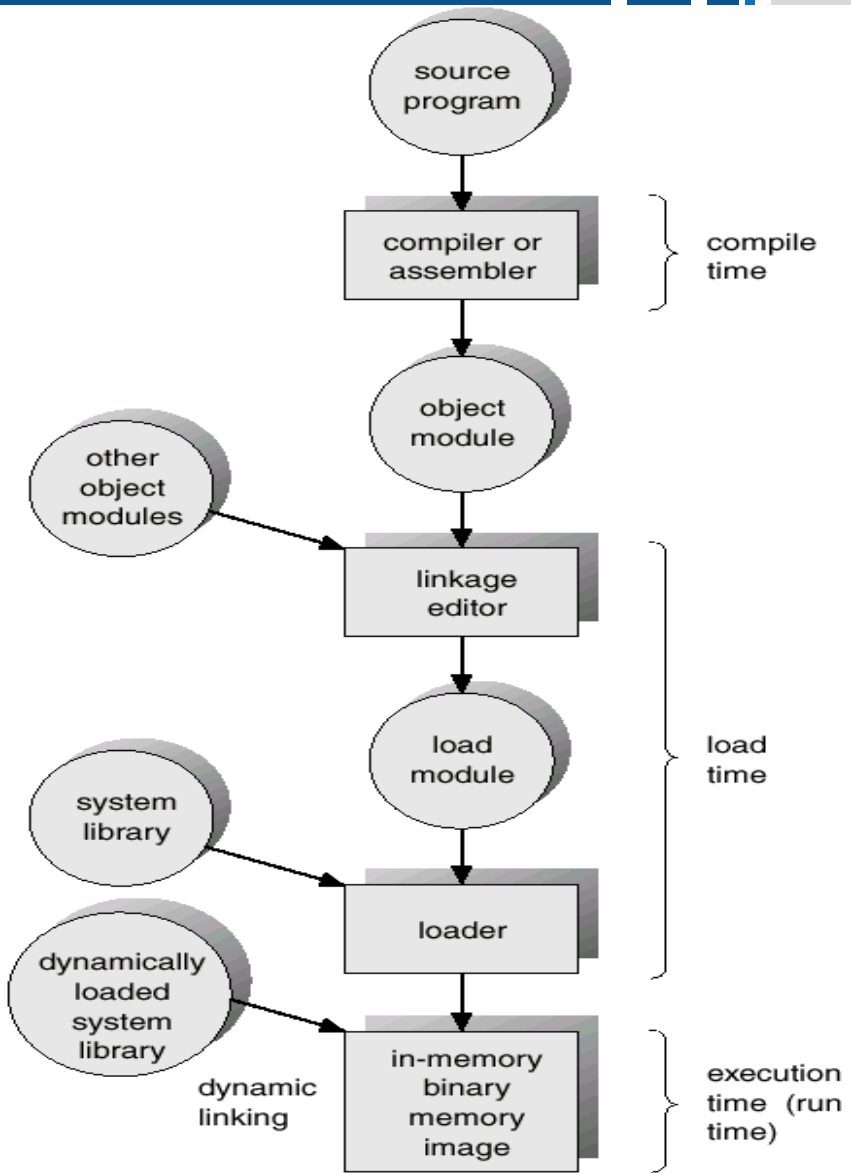


When are Addresses Created?

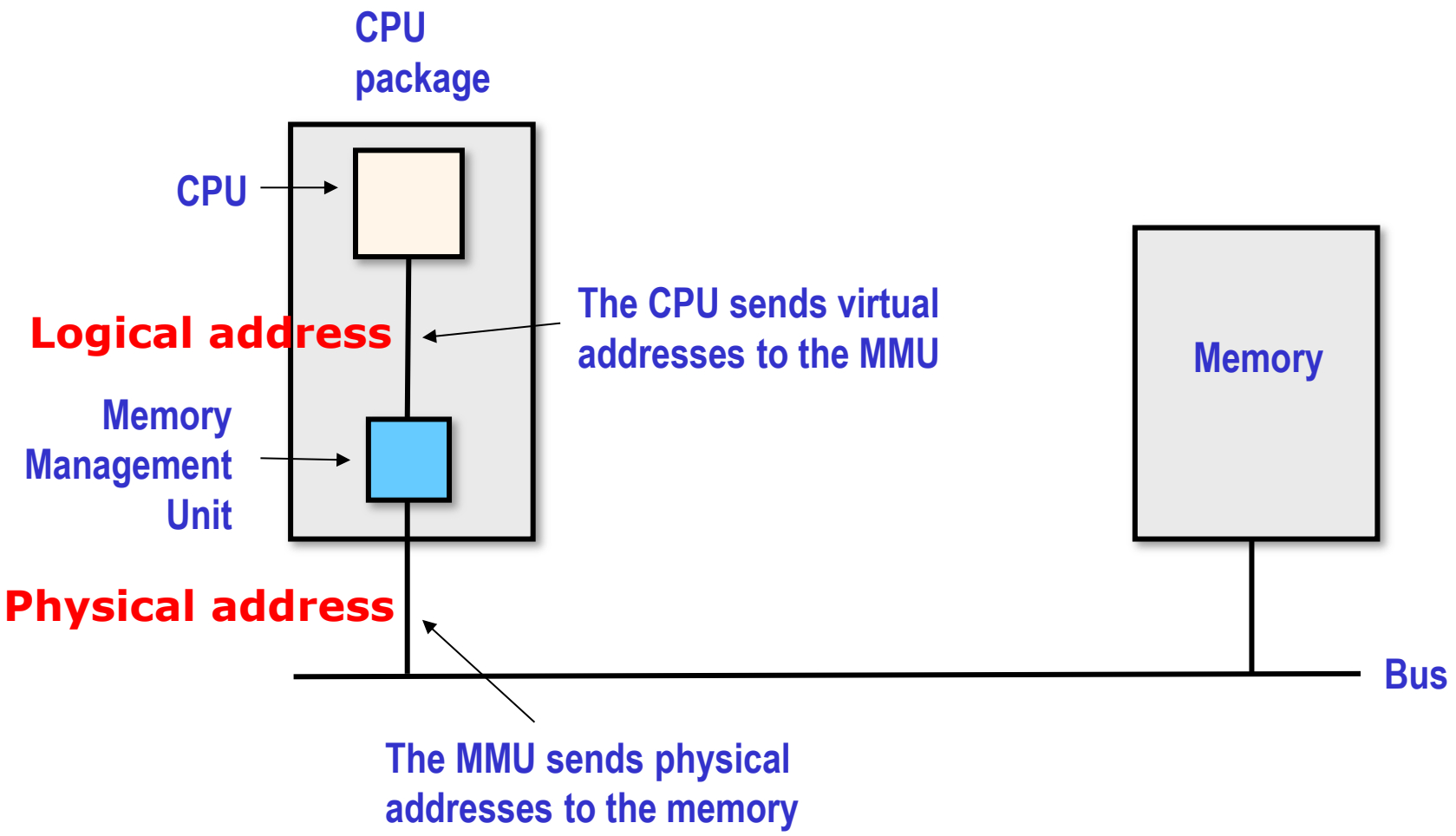


From Source to Execution…

- Programs need to be loaded to main memory before being executed
- Most addresses are known only after the load module (e.g., EXE file) is loaded into the main memory
  - Load module contains **relocatable** addresses



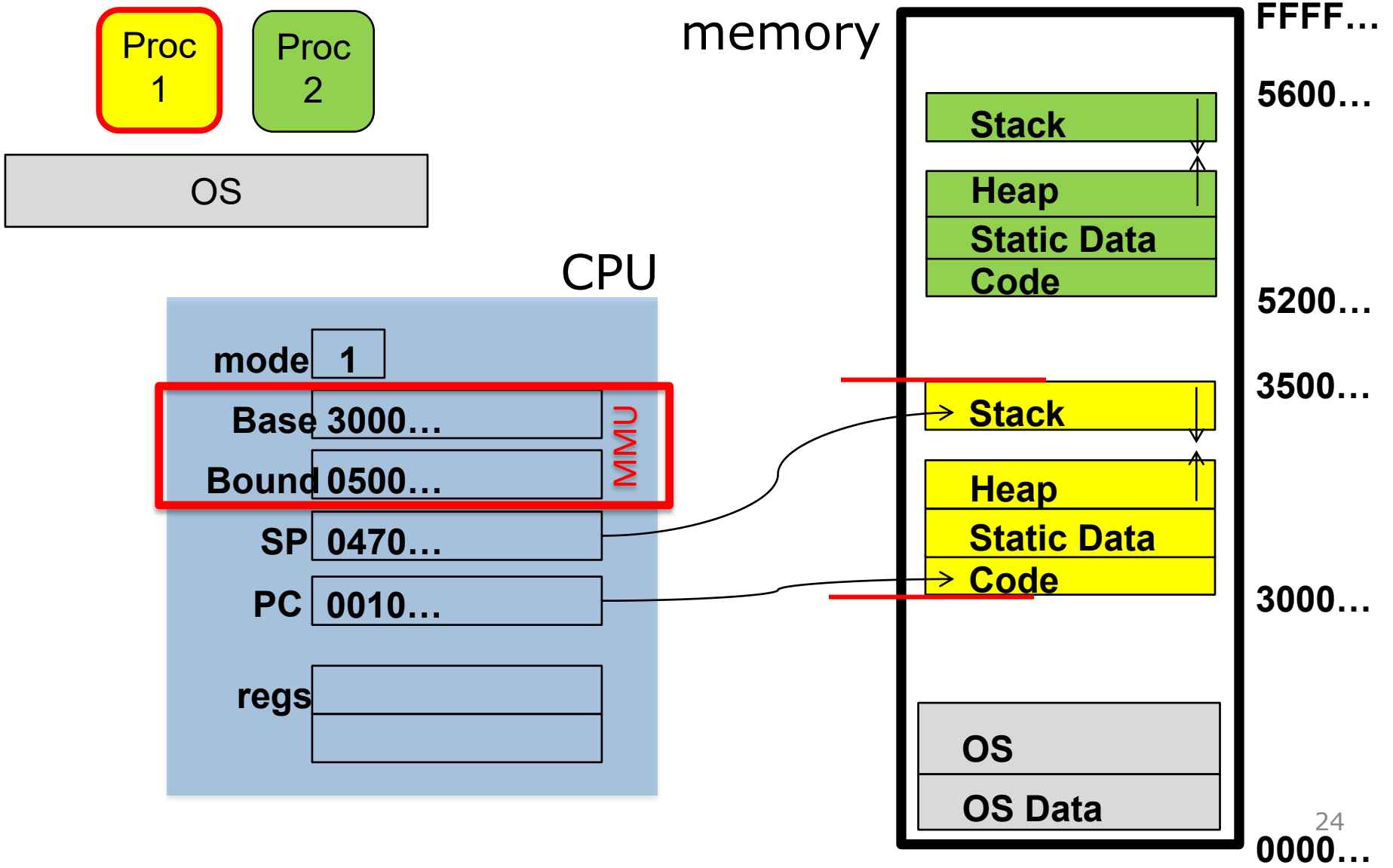
So, who is in charge of the translation?



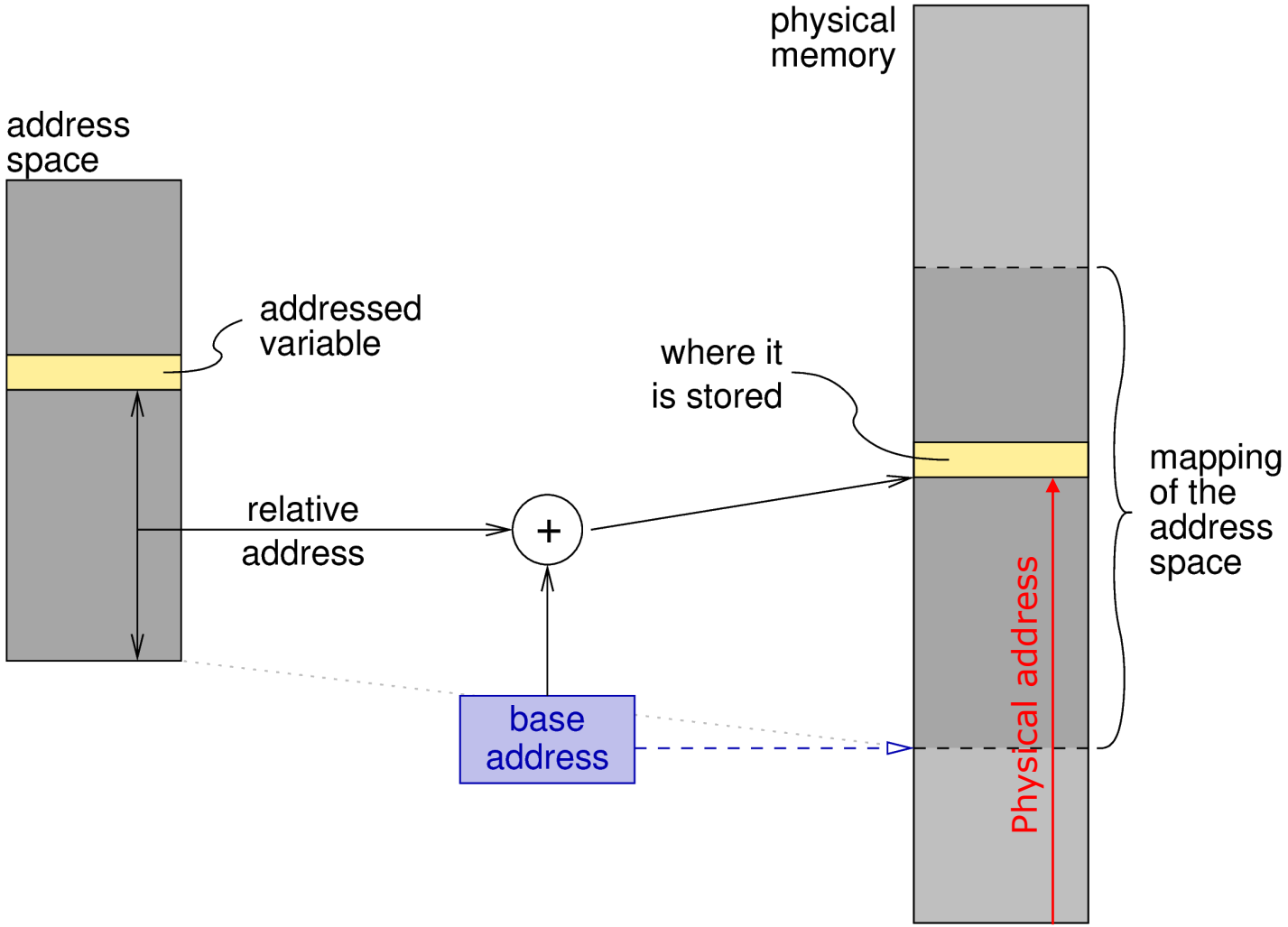
Contiguous Allocation

- Each process address space appears as a block in physical memory
- Address translation:
  - Each process gets a ***base*** address
  - **Logical address x** in the process is mapped to **physical address base+x** in the memory
- Very simple MMU implementation
  - Need to allocate "space to grow" → internal fragmentation

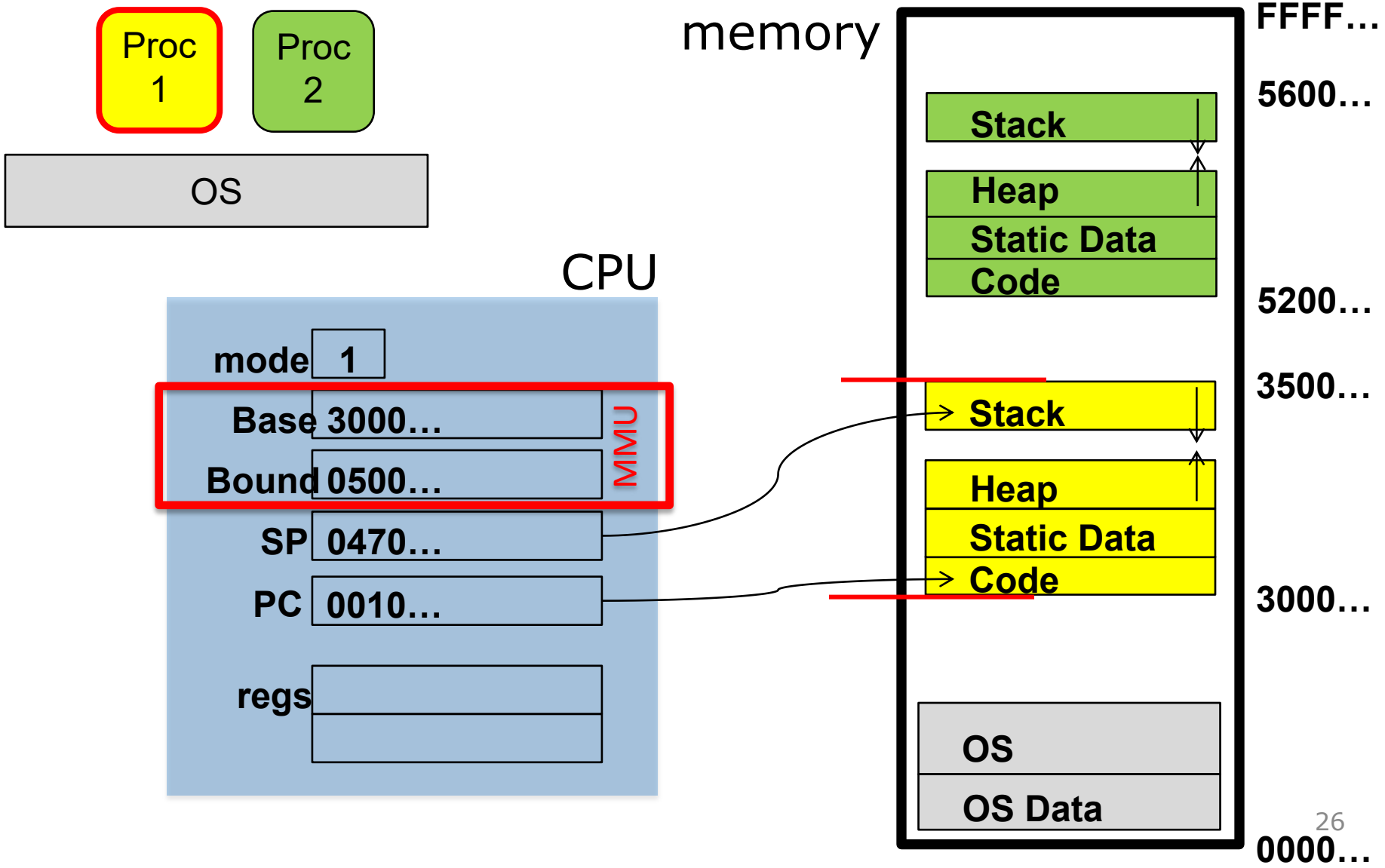
Simple Mapping Example



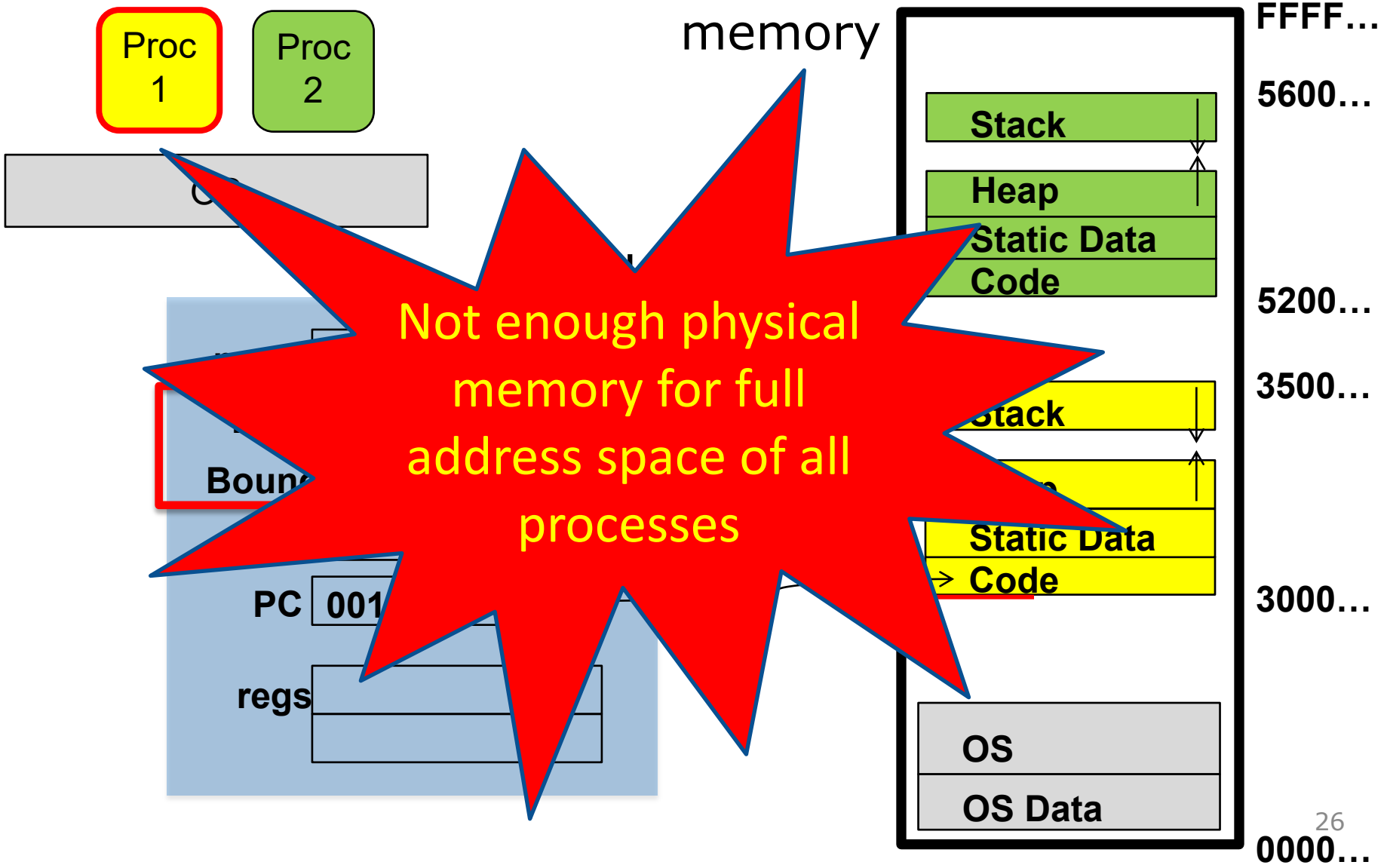
Address Translation Implementation



Problem


Proc 1
Proc 2
OS
CPU
mode 1
Base 3000…
Bound 0500…
SP 0470…
PC 0010…
regs
MMU
memory
FFFF…
5600…
Stack
Heap
Static Data
Code
5200…
3500…
Stack
Heap
Static Data
Code
3000…
OS
OS Data
0000…

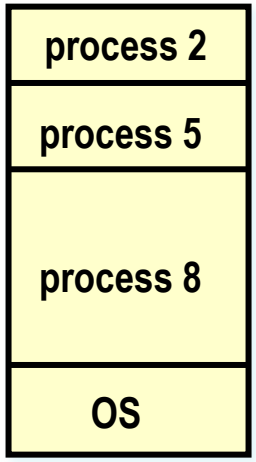
Problem


Proc 1
Proc 2
memory
FFFF…
5600…
Stack
Heap
Static Data
Code
5200…
3500…
Stack
Static Data
Code
3000…
OS
OS Data
0000…
Not enough physical memory for full address space of all processes
PC
001
regs

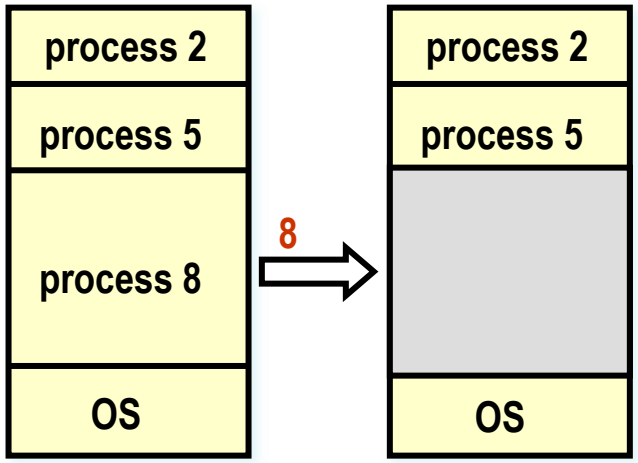
Fragmentation

Allocation Dynamics

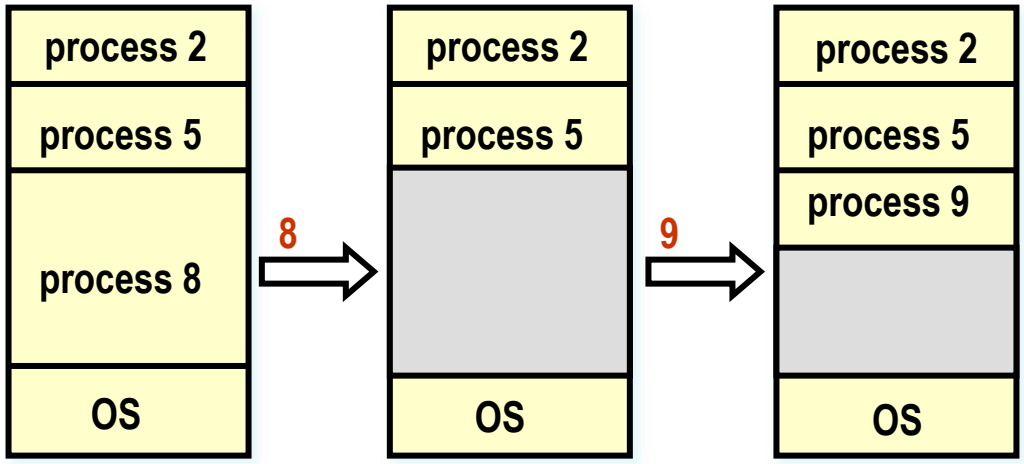
Allocation Dynamics



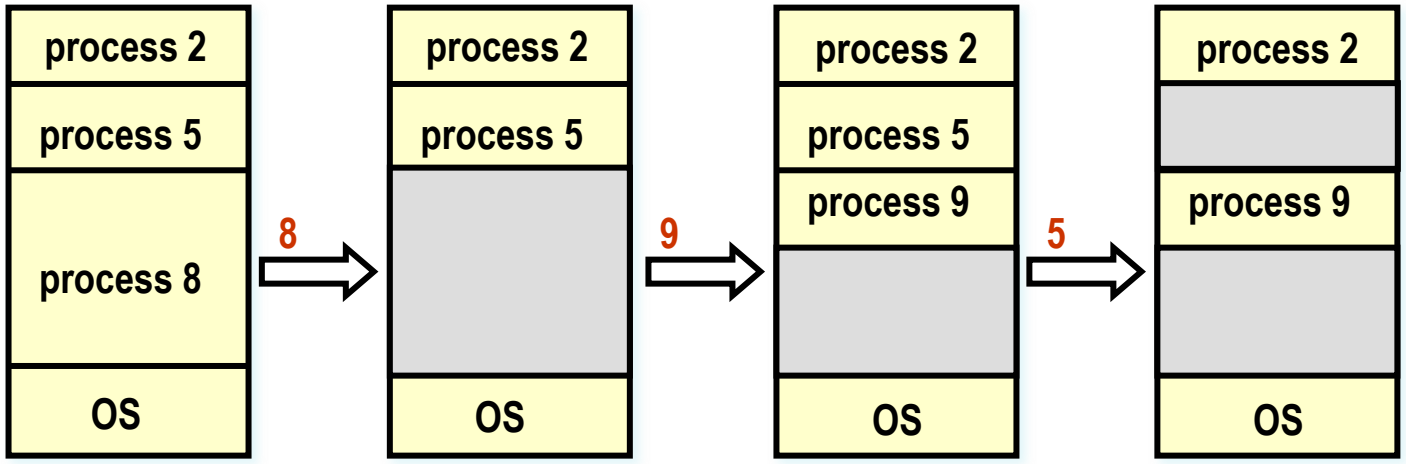
Allocation Dynamics



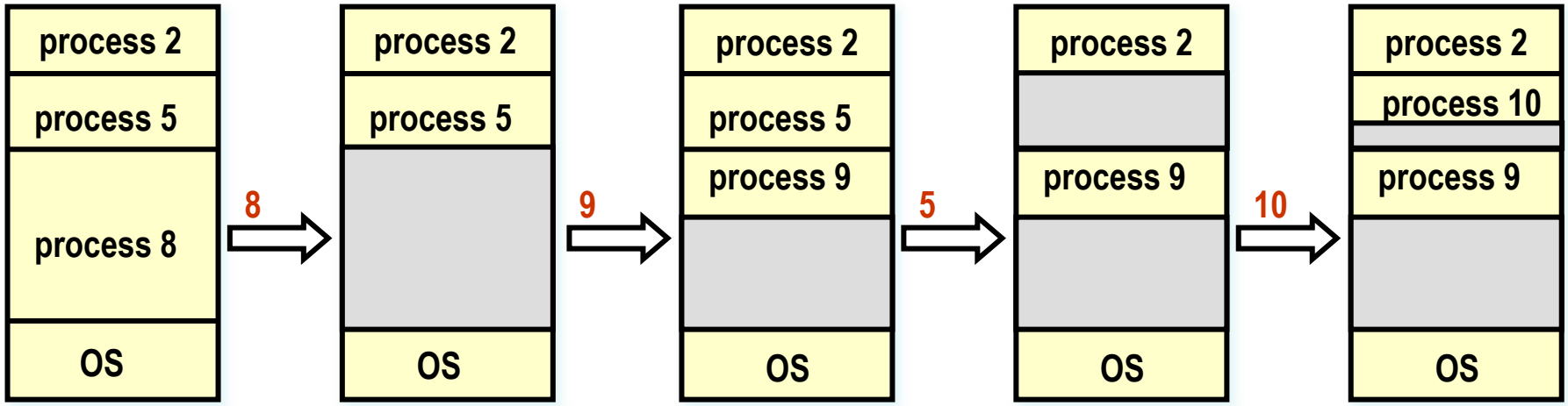
Allocation Dynamics



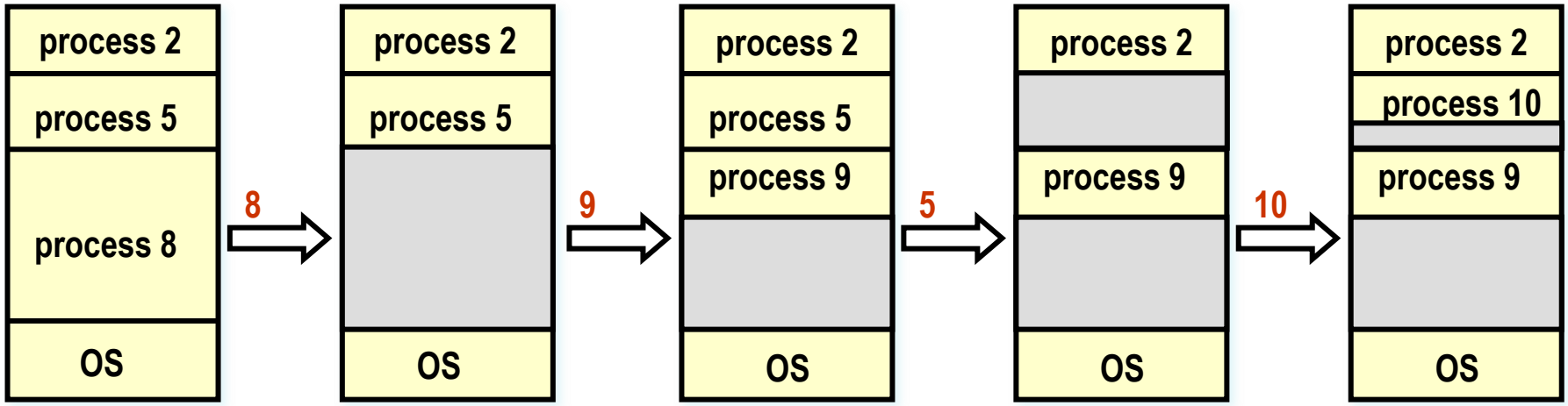
Allocation Dynamics



Allocation Dynamics



Allocation Dynamics



Result: External Fragmentation

Fragmentation

- Situation in which we have enough memory to allocate to a process, but not contiguously, as the holes are scattered all over memory
  - Internal Fragmentation: free memory inside process' allocation
  - External Fragmentation: free memory between processes' allocations
- We try to solve only external fragmentation!

Allocation Algorithms

- Input:
  - List of current free ranges
  - Request for new allocation
- Output:
  - Decision which free range to use
  - Can use only part

Allocation Algorithms

- **First fit:**

  Scan the list and select the first range that fits
- Linear time, constant < 1
- May cause excessive fragmentation at low addresses

Allocation Algorithms

- **Next fit:**

  Scan the list starting from where you left off last time and select the first range that fits
- Linear time, constant < 1
- Spreads fragmentation more evenly

Allocation Algorithms

- **Best fit:**

  Scan the and select the range that provides the snuggest fit
- Linear time
- Maintains large ranges
- Causes small fragments

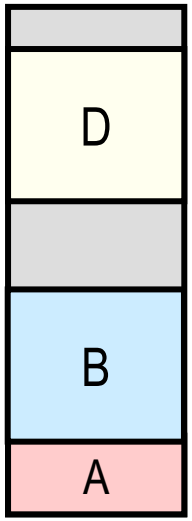
Fragmentation Solution: Compaction

Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes

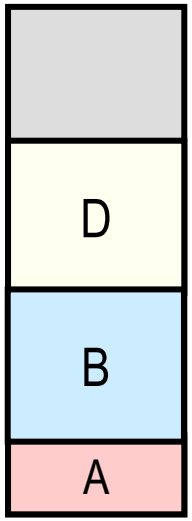
Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes



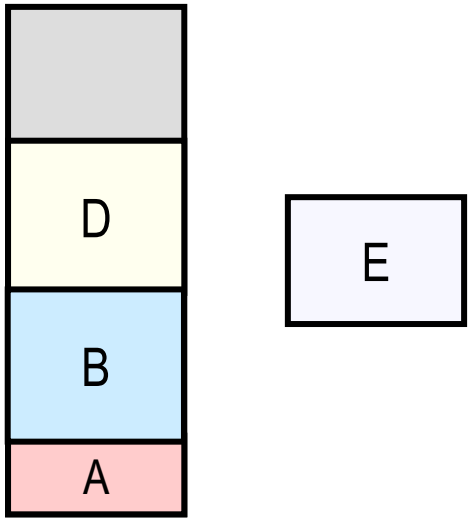
Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes



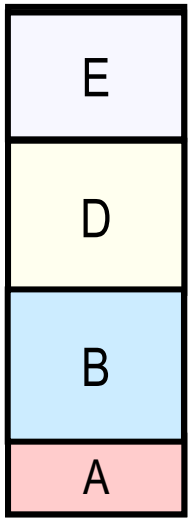
Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes



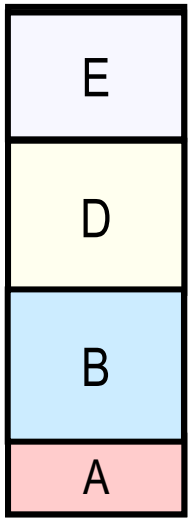
Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes



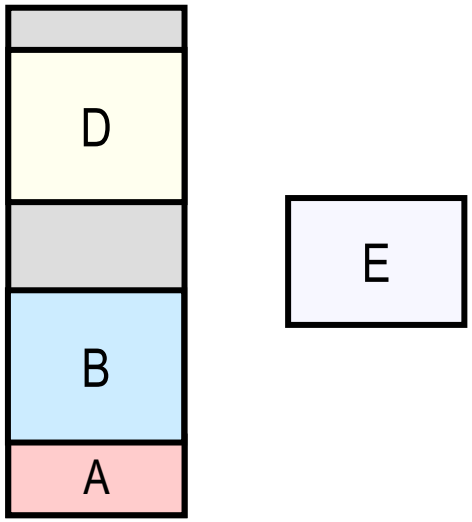
Fragmentation Solution: Compaction

- Move around data to unify holes and create large enough holes to accommodate future processes
- **Problem:** Very costly operation. Sometime not enough space to move data



Better Solution: Using Pages

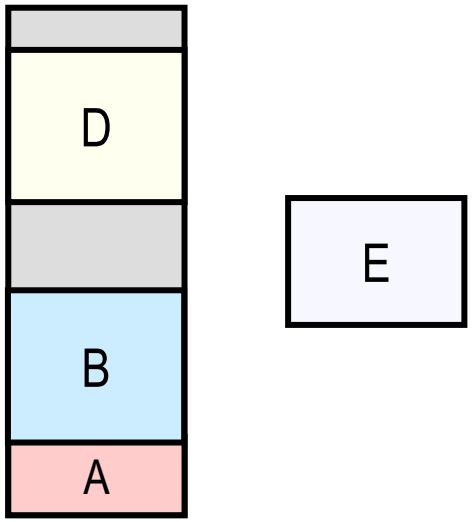
- Divide process memory space into fixed size **pages**
- Map each page independently



Better Solution: Using Pages

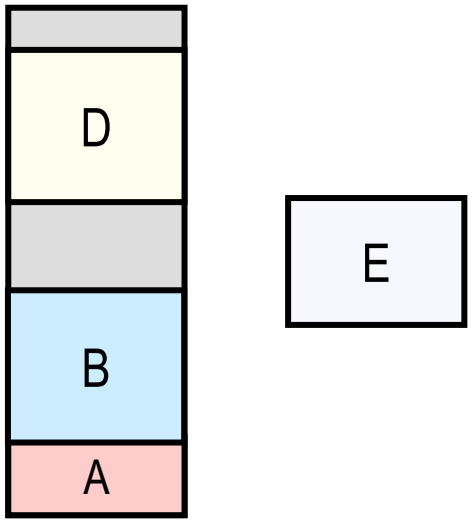
- Divide process memory space into fixed-size **pages**
- Map each page independently

Segmentation



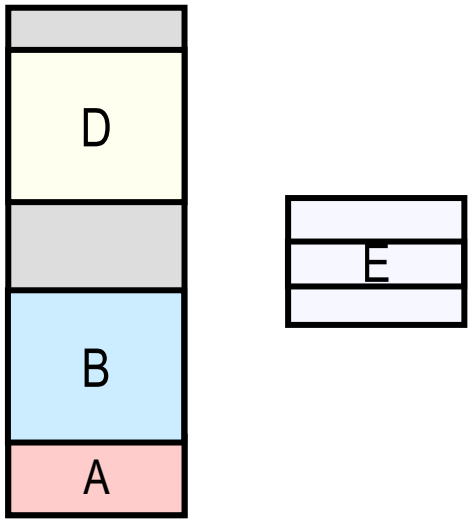
Better Solution: Using Pages

- Divide process memory space into fixed size **pages**
- Map each page independently



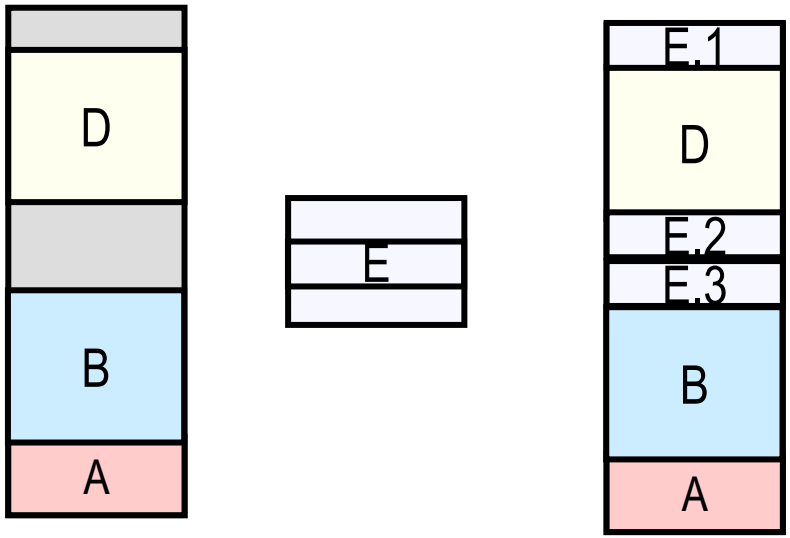
Better Solution: Using Pages

- Divide process memory space into fixed size **pages**
- Map each page independently



Better Solution: Using Pages

- Divide process memory space into fixed size **pages**
- Map each page independently



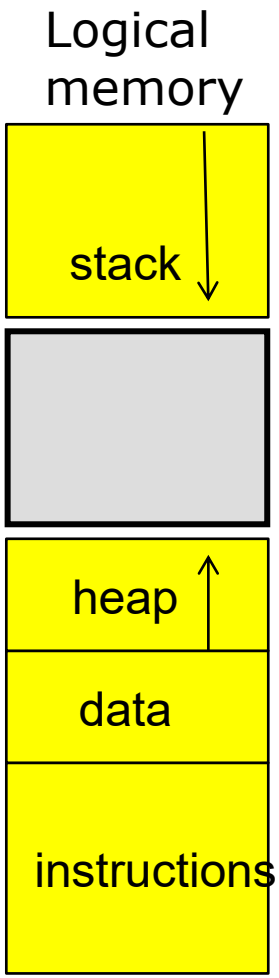
PAGING

The Concept of Paging

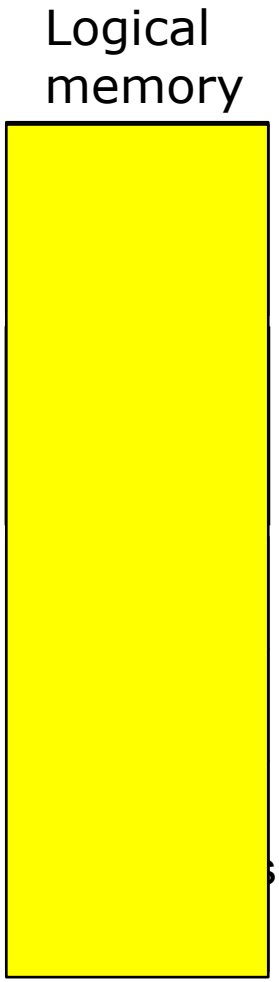
- The address space is divided into pages
- All pages have a fixed size
  - Typically 4KB (4096 bytes)
- The physical memory is (conceptually) divided into frames
  - The same size as pages
- Any page can be mapped to any frame
  - The mapping is done by the OS ...
  - ... and used by the MMU to access memory

Pages and Frames

Pages and Frames

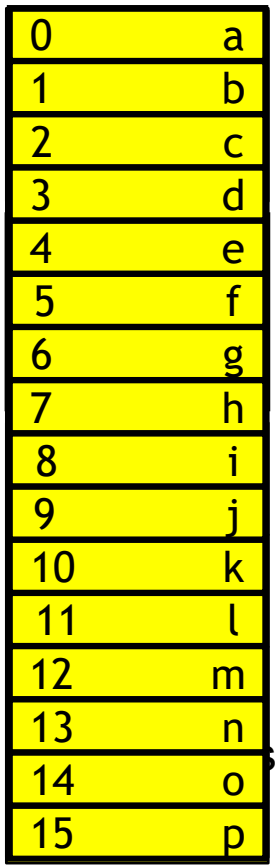


Pages and Frames



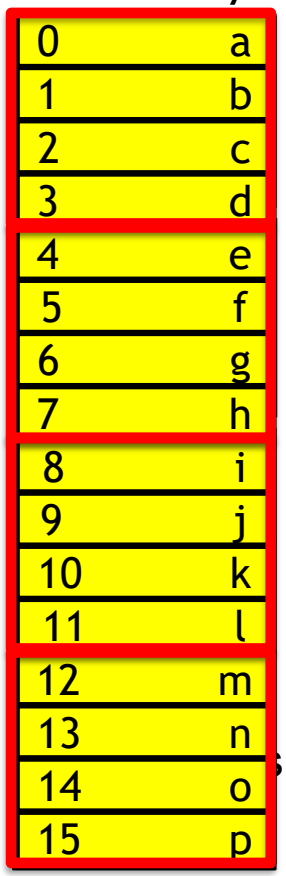
Pages and Frames

Logical memory



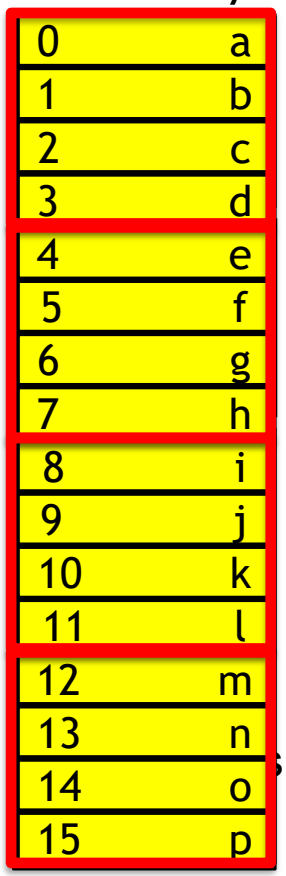
Pages and Frames

Logical memory

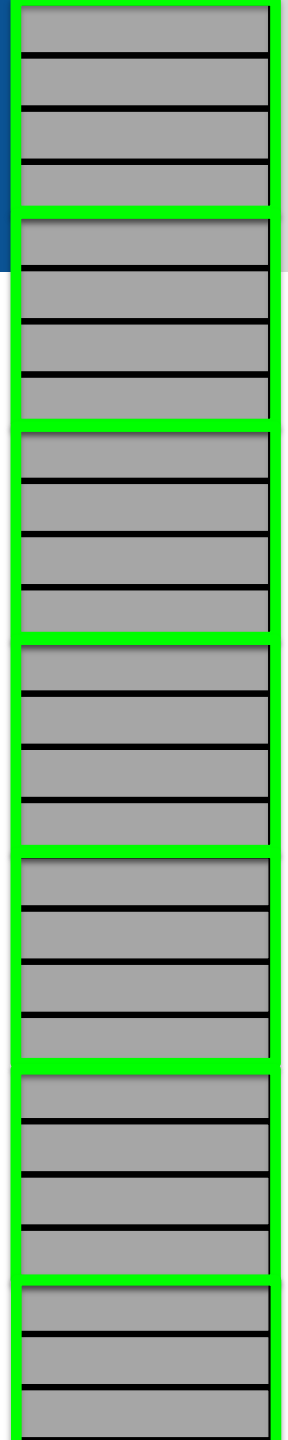


Pages and Frames

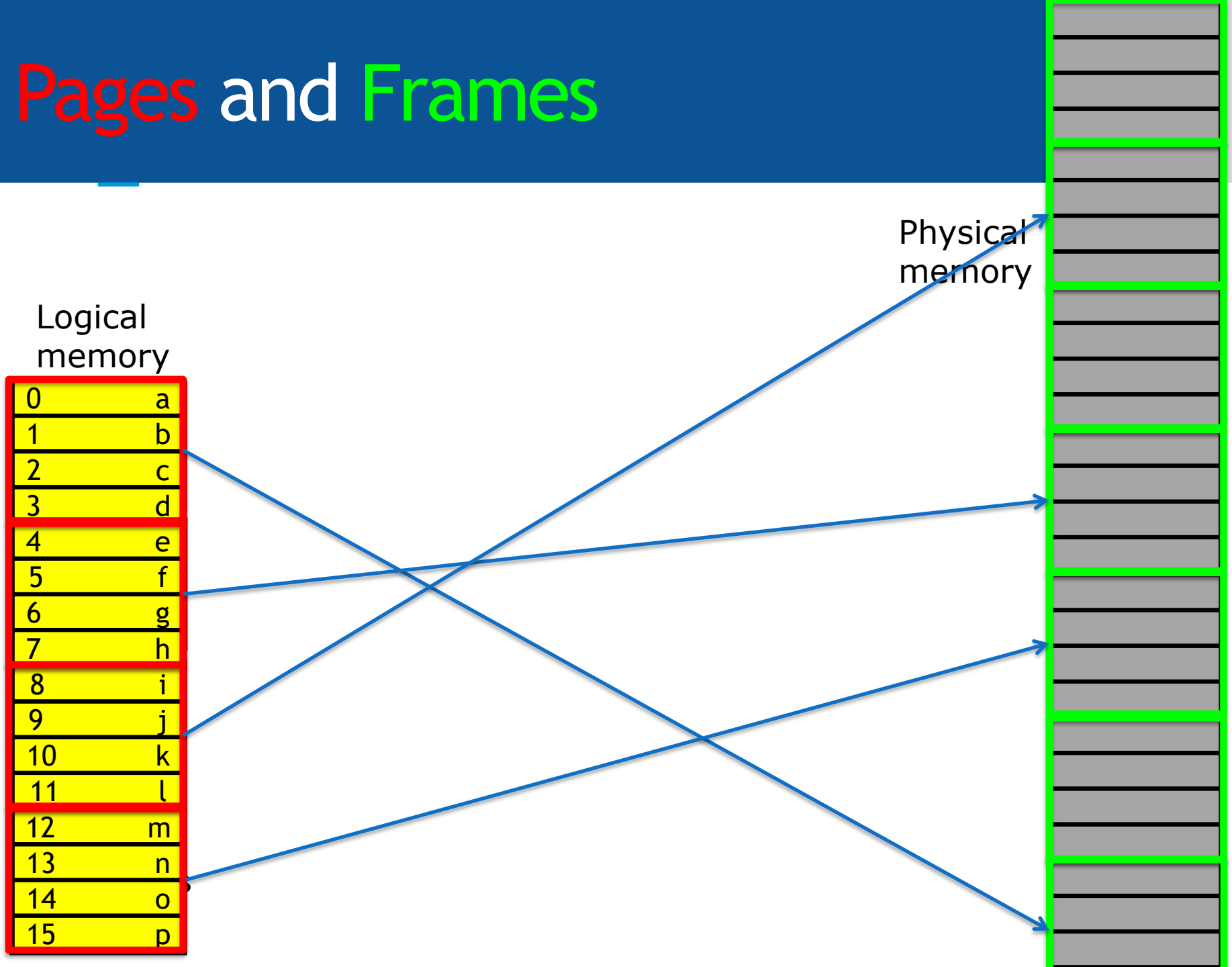
Logical memory



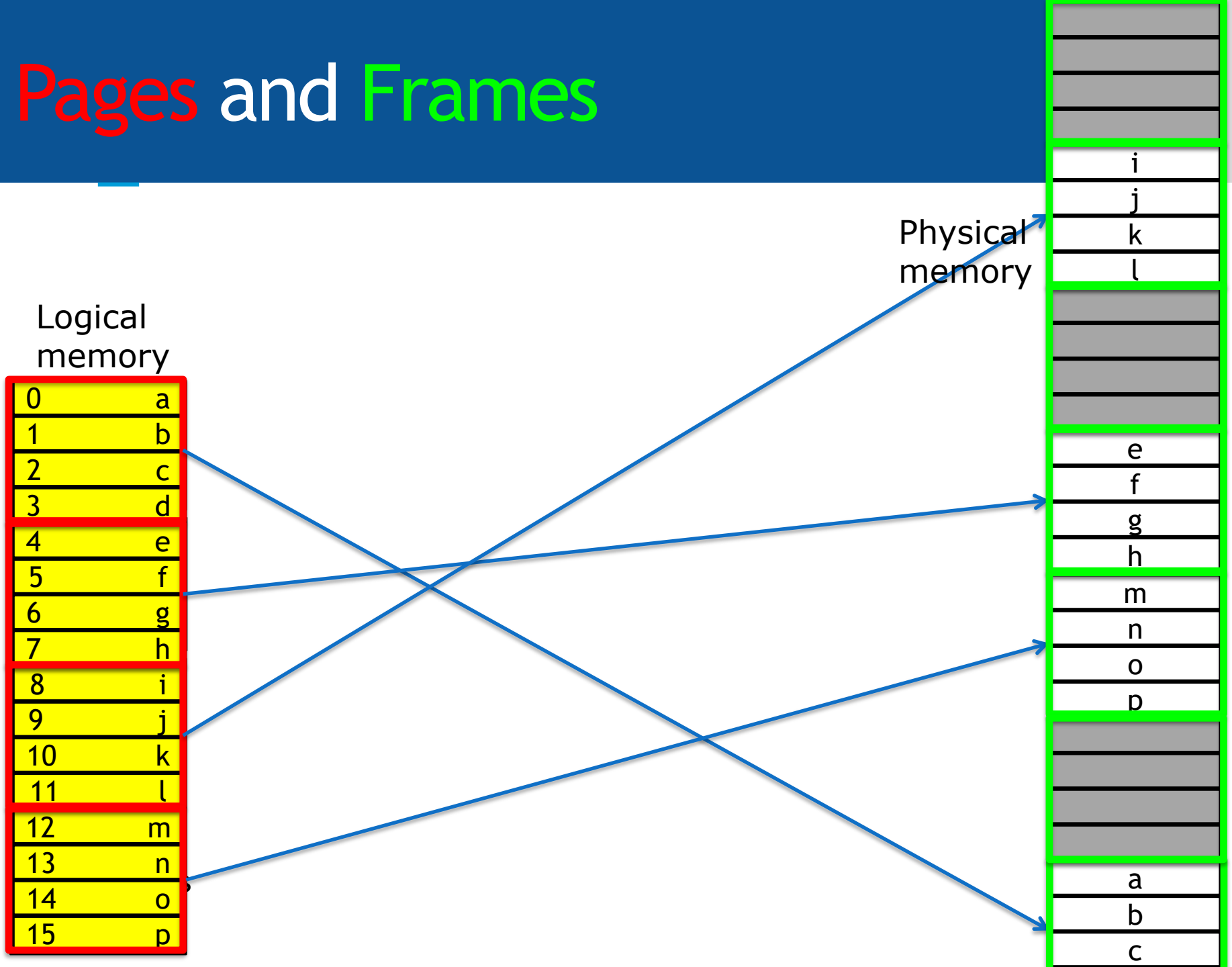
Physical memory



Pages and Frames

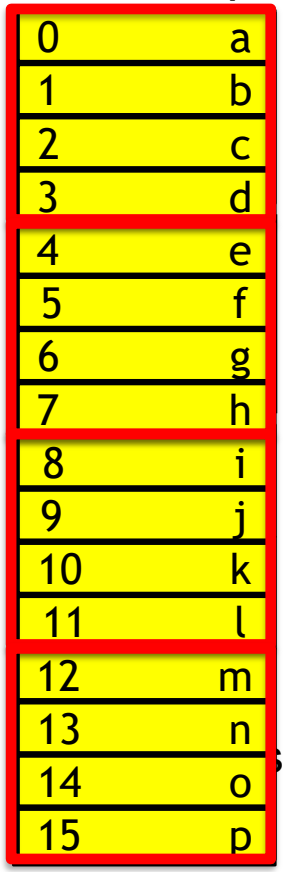


Pages and Frames

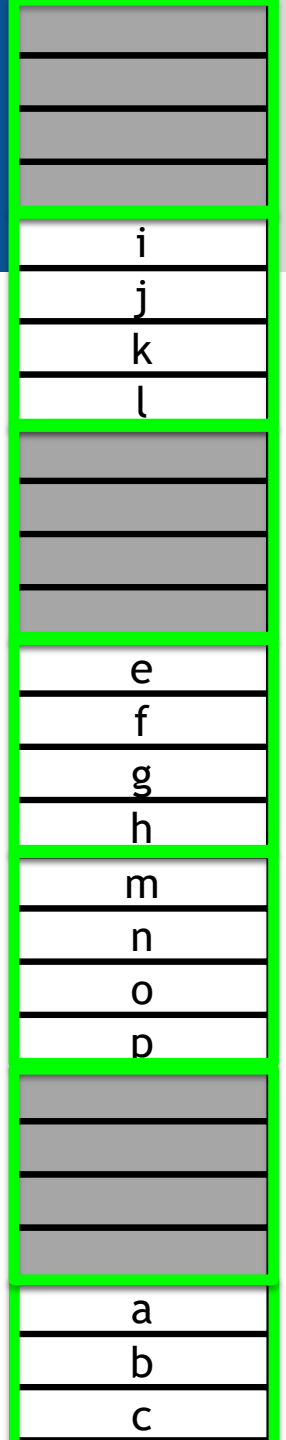


Pages and Frames

Logical memory

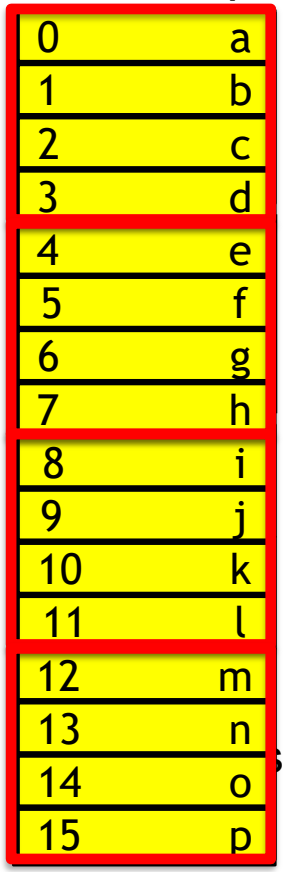


Physical memory



Pages and Frames

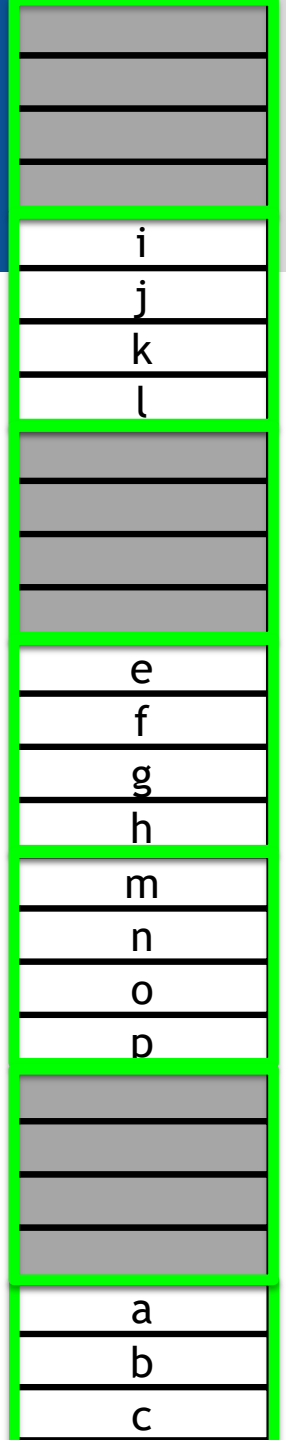
Logical memory



Page Table

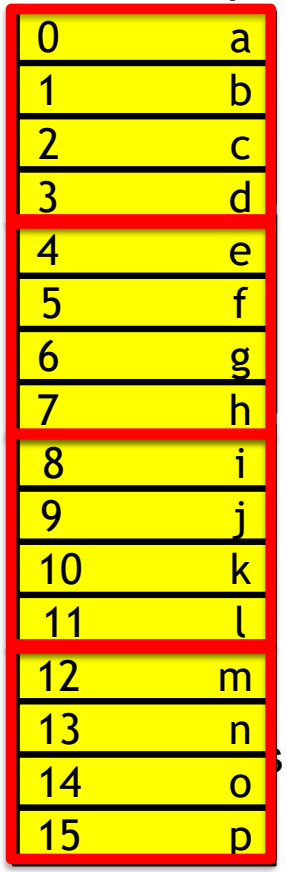
Page	Frame
0	6
1	3
2	1
3	4

Physical memory



Pages and Frames

Logical memory

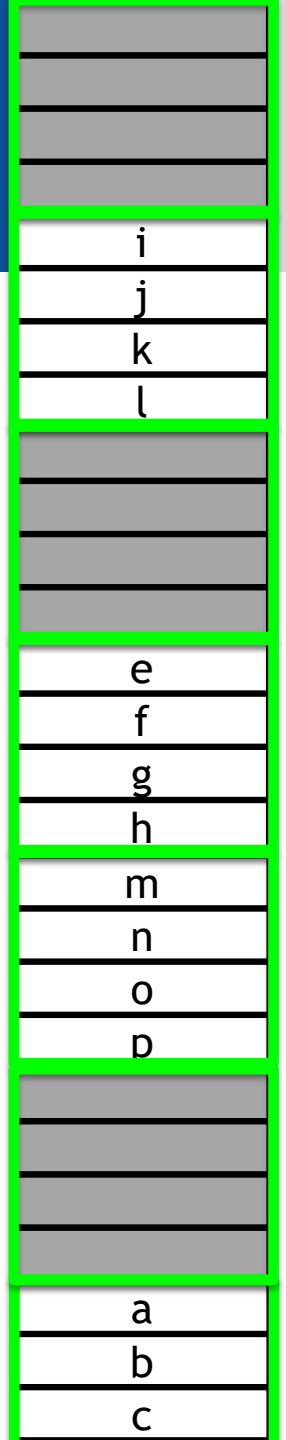


Page Table

Page	Frame
0	6
1	3
2	1
3	4

(per process)

Physical memory



Pages and Frames

Logical memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

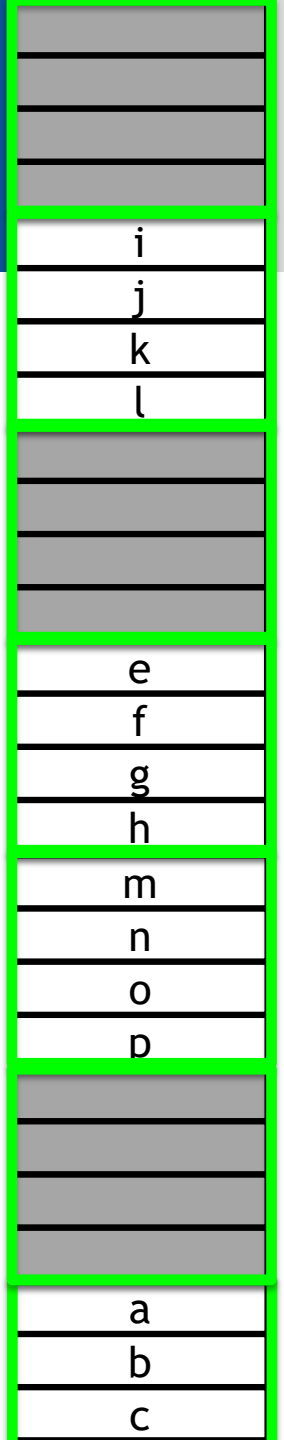
Page Table

Page	Frame
0	6
1	3
2	1
3	4

(per process)

Used by the MMU

Physical memory



Pages and Frames

Logical memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

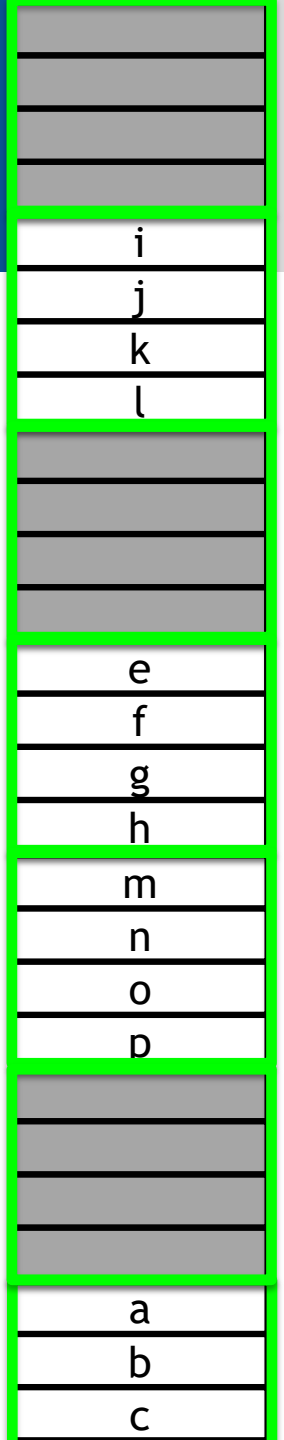
Page Table

Page	Frame
0	6
1	3
2	1
3	4

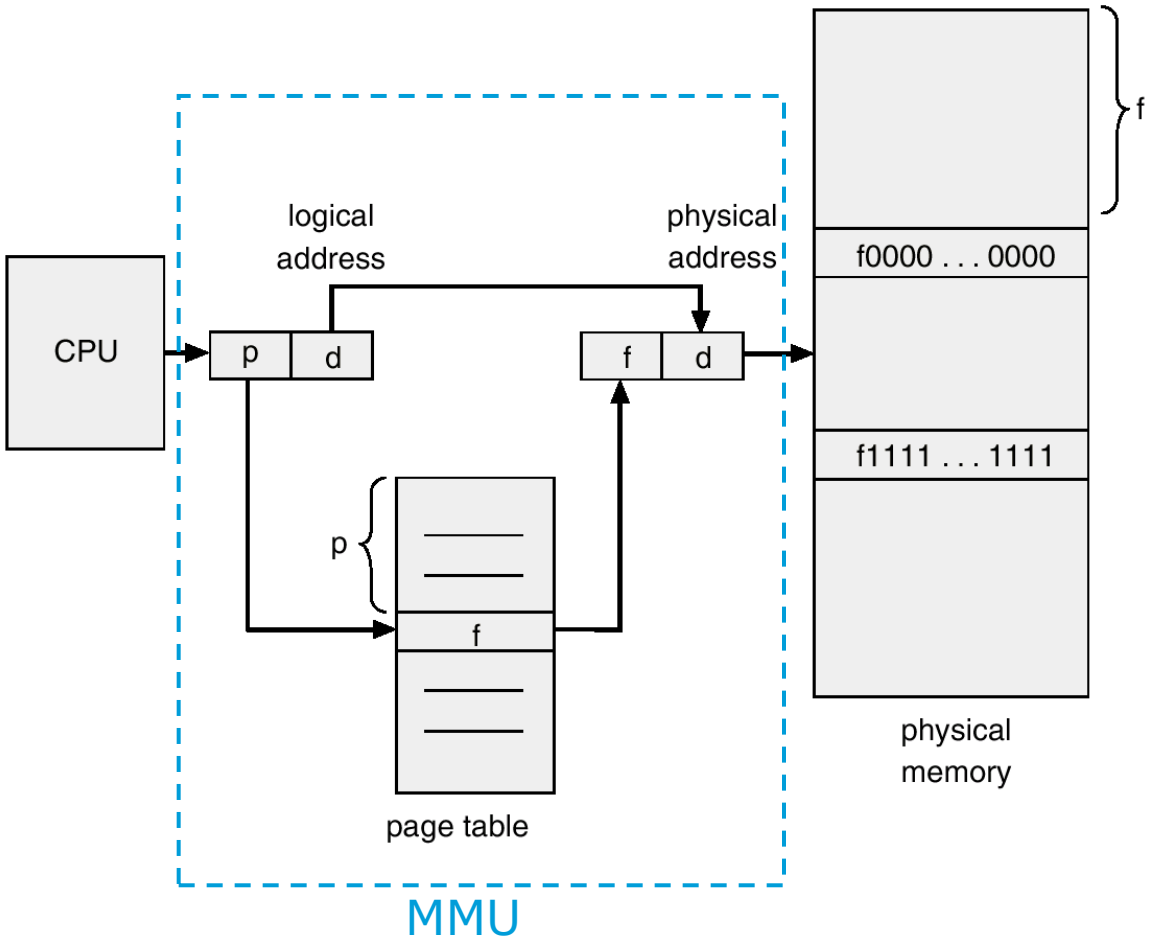
(per process)

Used by the MMU

Physical memory



Address Translation Architecture



Working with the page table

Logical memory

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

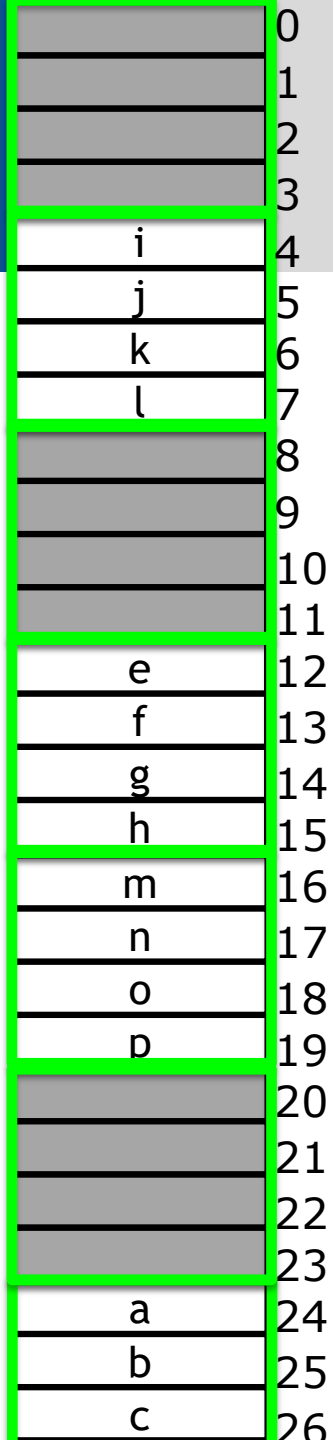
Page Table

Page	Frame
0	6
1	3
2	1
3	4

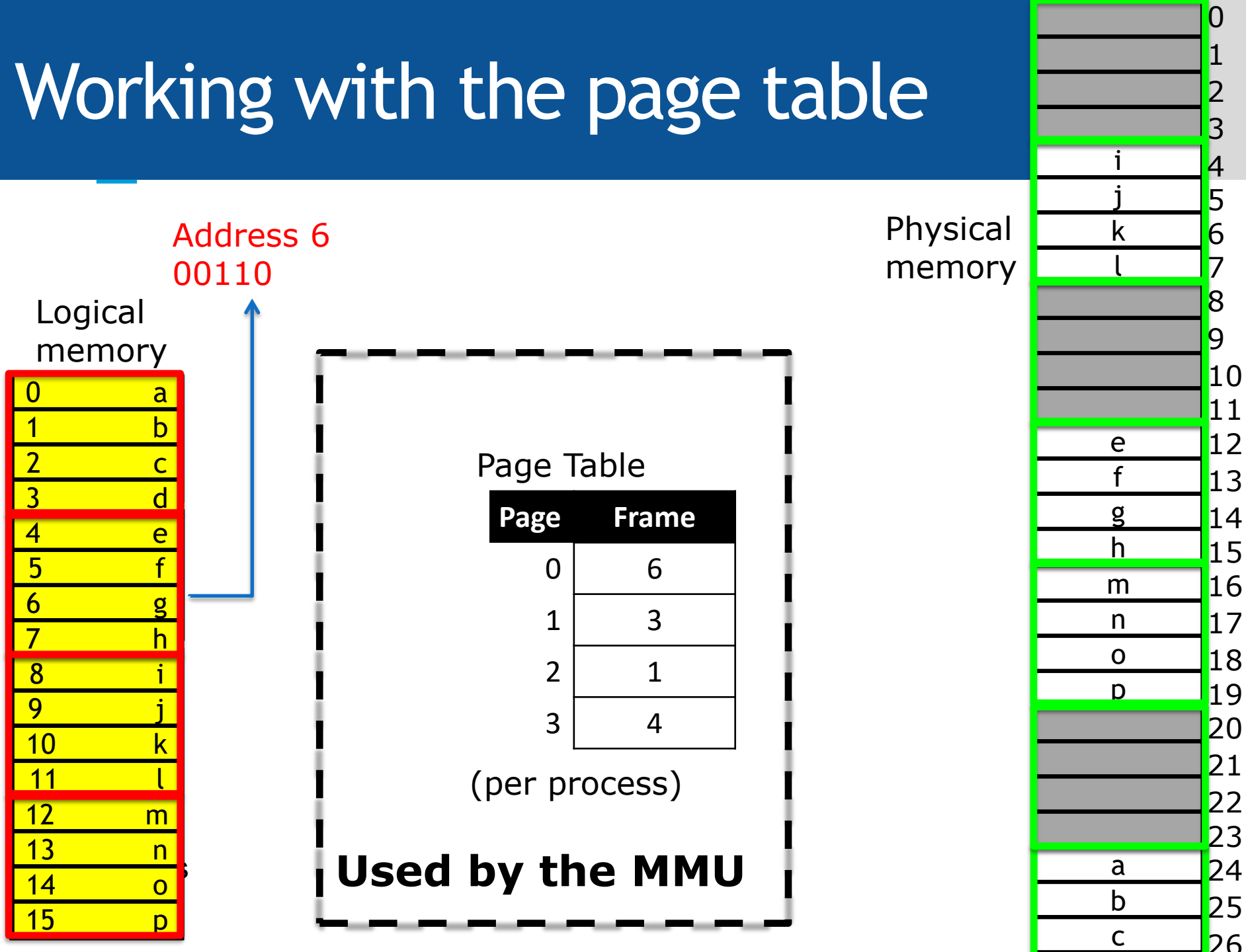
(per process)

Used by the MMU

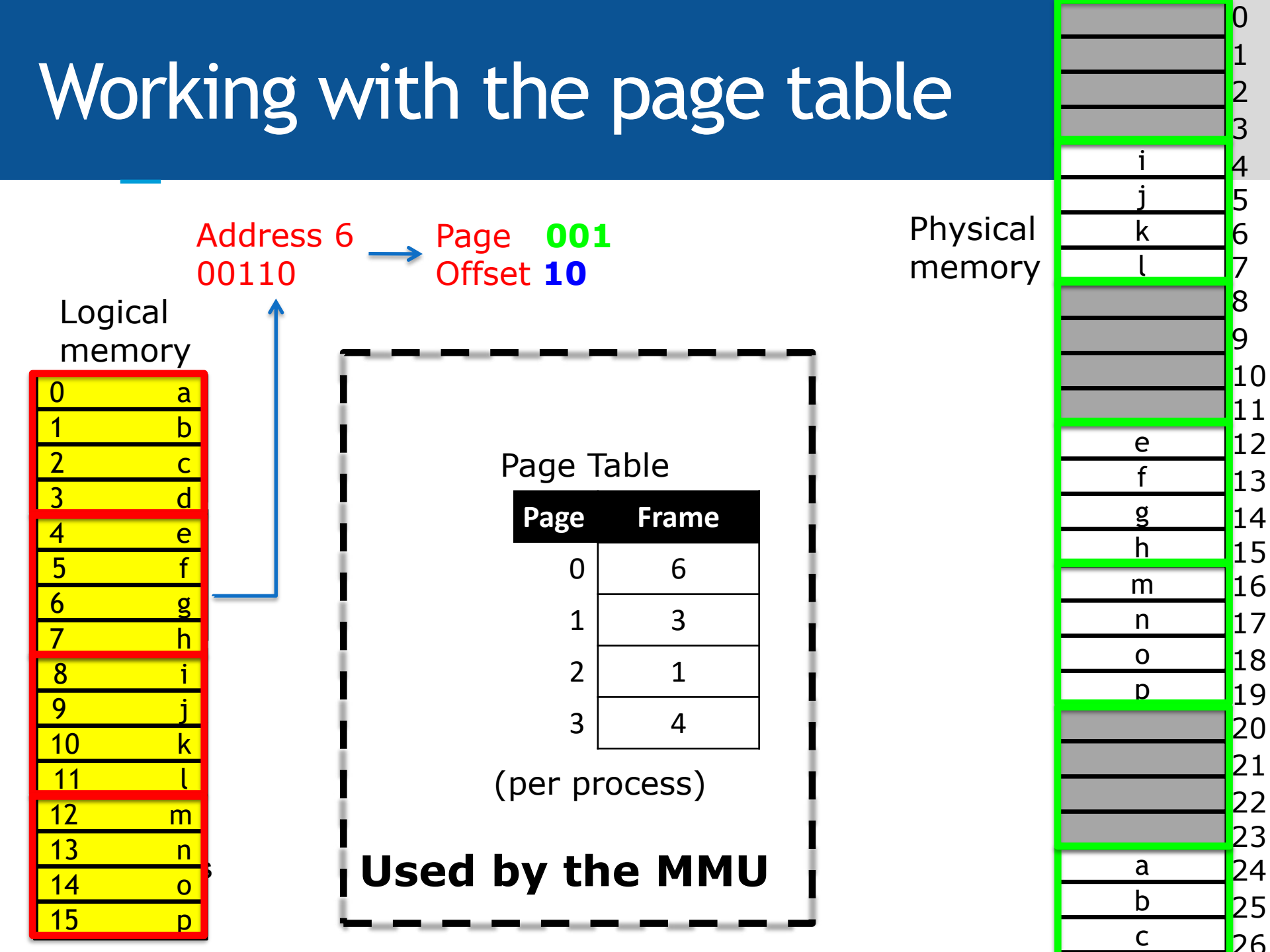
Physical memory



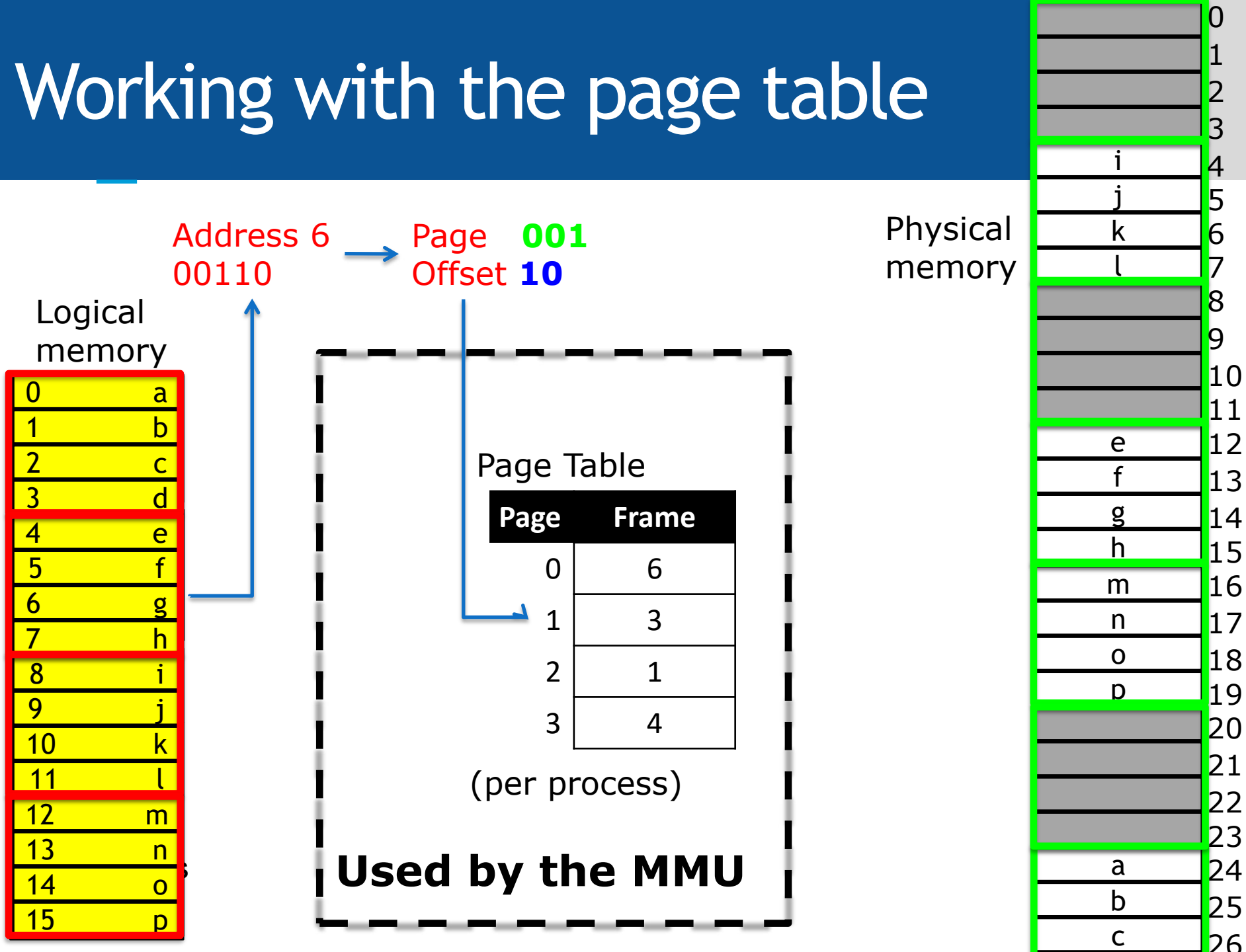
Working with the page table



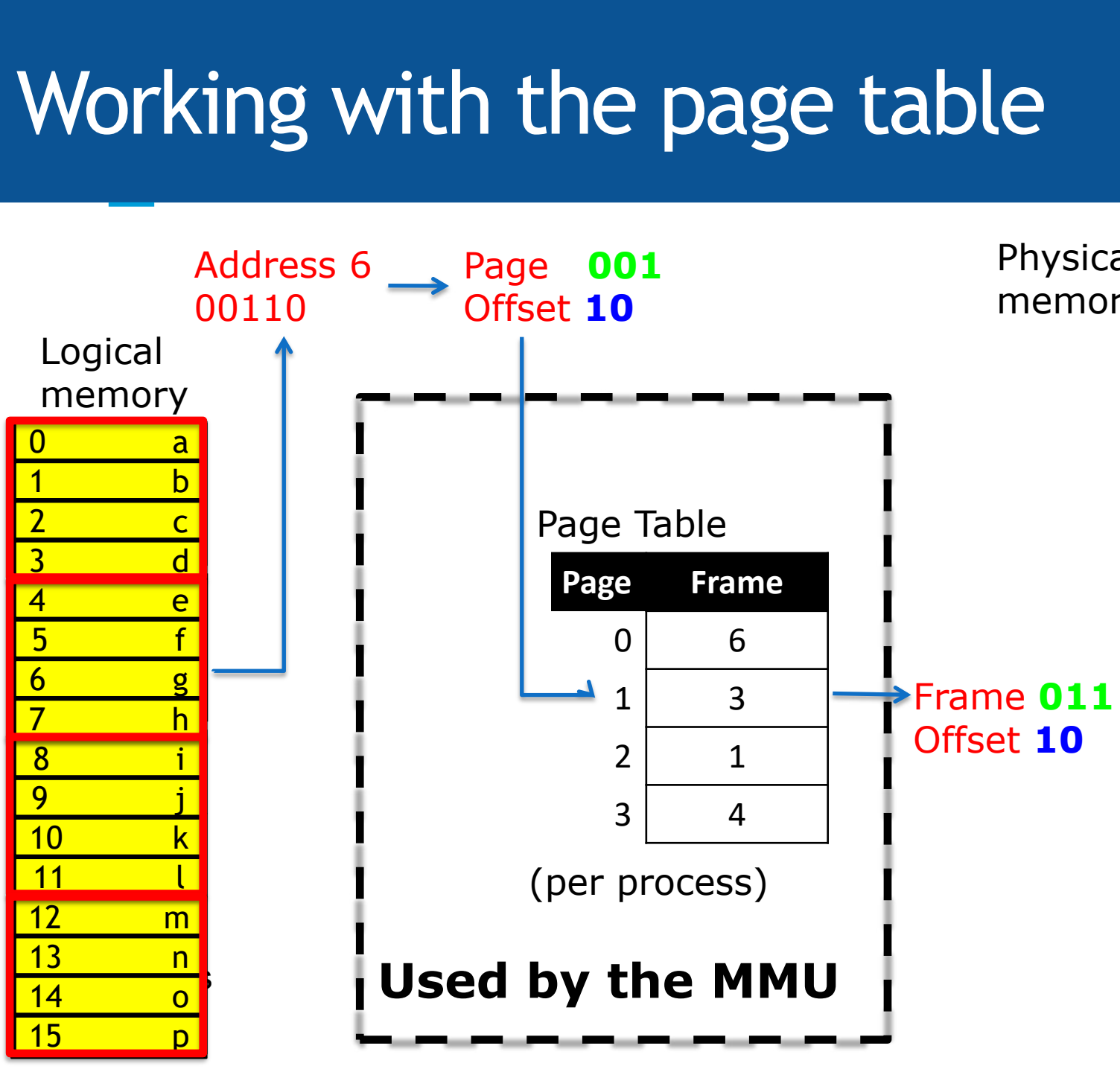
Working with the page table



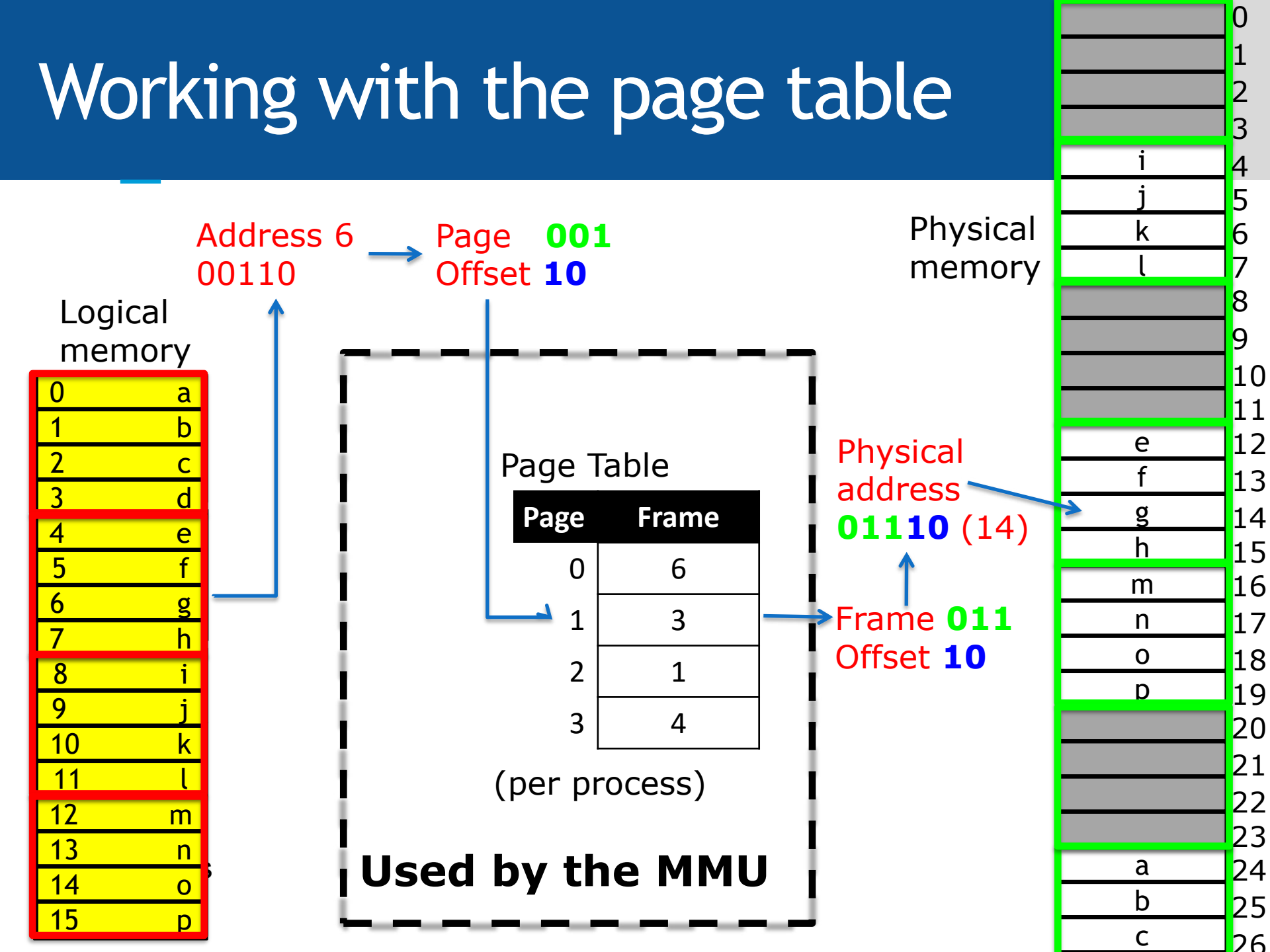
Working with the page table



Working with the page table



Working with the page table



Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables

Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables
- p is the page size
  s size of the process
  e size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables
- p is the page size
  s size of the process
  e size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Page table size: s/p pages → s/p entries, each of size e

Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables
- p is the page size
  s size of the process
  e size of the entry in the page table
  → **Overhead = (se/p)+p/2**

Page table size: s/p pages → s/p entries, each of size e

Internal fragmentation

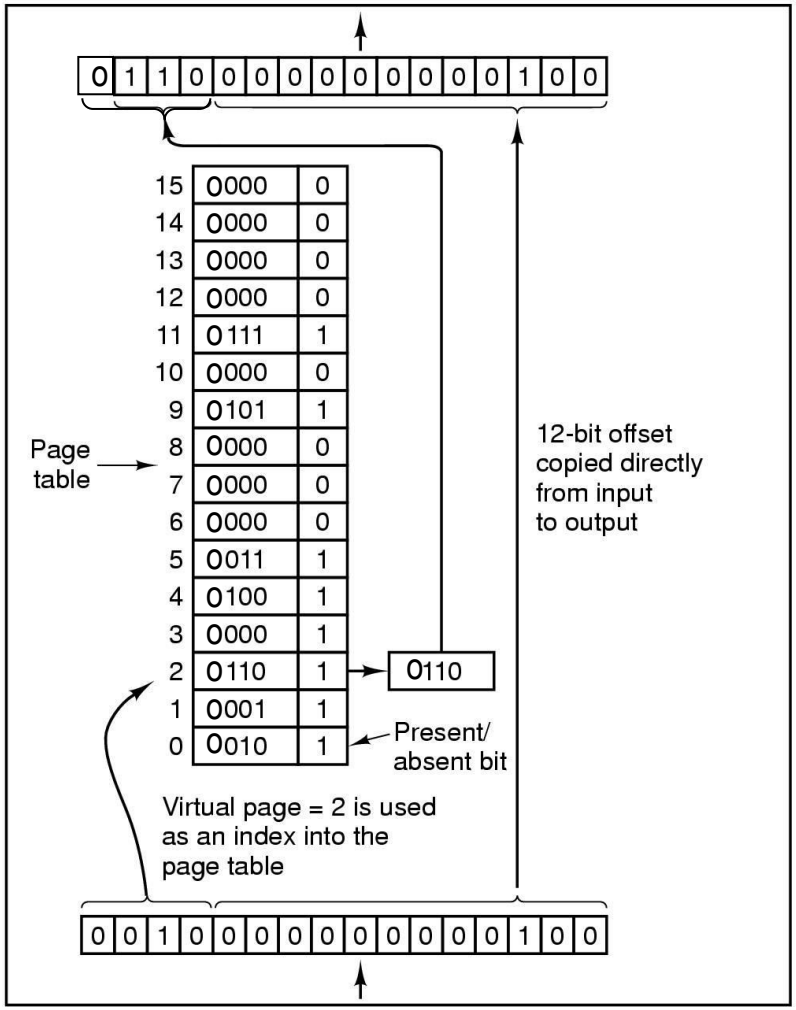
Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables
- p is the page size
  s size of the process
  e size of the entry in the page table
  → **Overhead = (se/p)+p/2**
  → **Optimal page size is √(2se)**

Optimal page size

- Small page size:
  - Less internal fragmentation
  - Larger page tables
- p is the page size
  s size of the process
  e size of the entry in the page table
  → **Overhead = (se/p)+p/2**
  → **Optimal page size is √(2se)**
- For s=1MB, e=64 bit → p=4KB

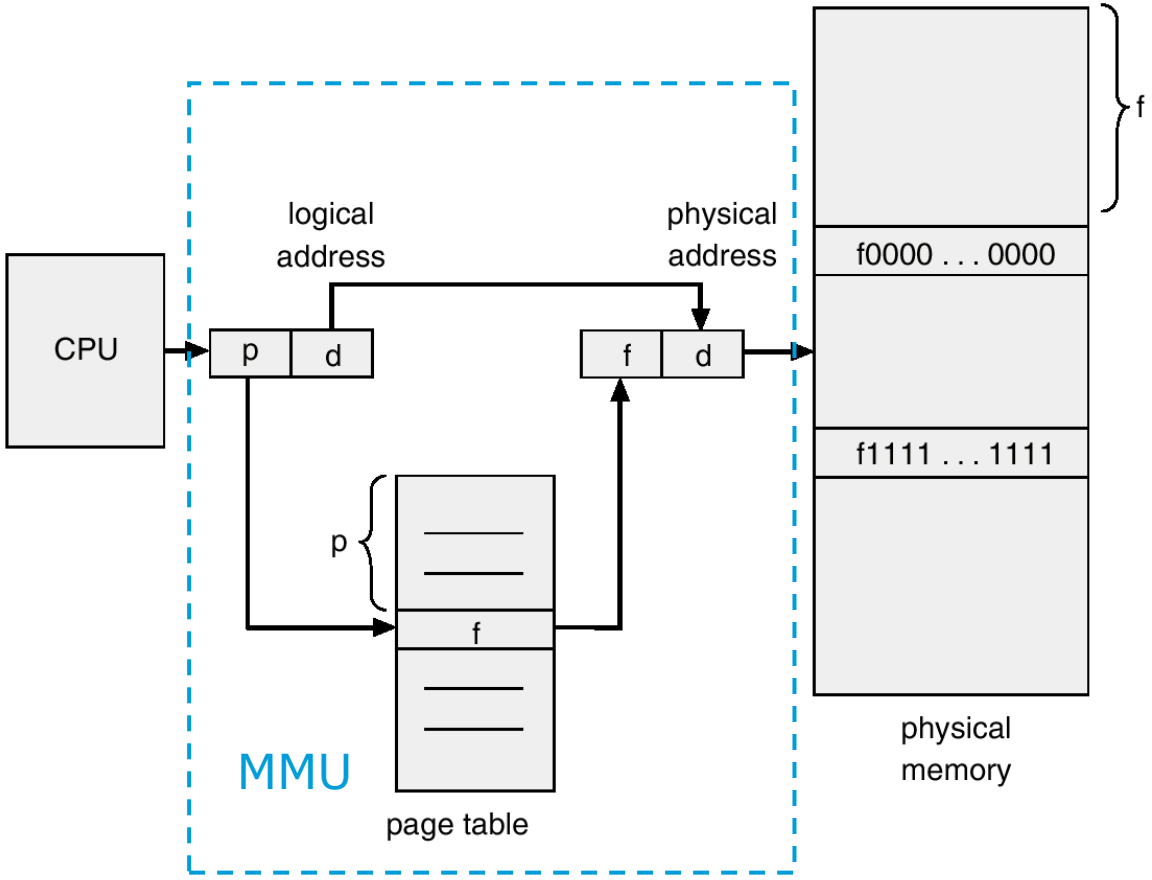
Internal operation of MMU with 16 4KB pages



Address Translation

- Divide virtual address into two parts:
  - Page number (top 20 bits)
  - Offset into page (bottom 12 bits)
- Use page number as index into page table
- Get frame number from page table (20 bit)
- Combine frame number and offset to create physical address

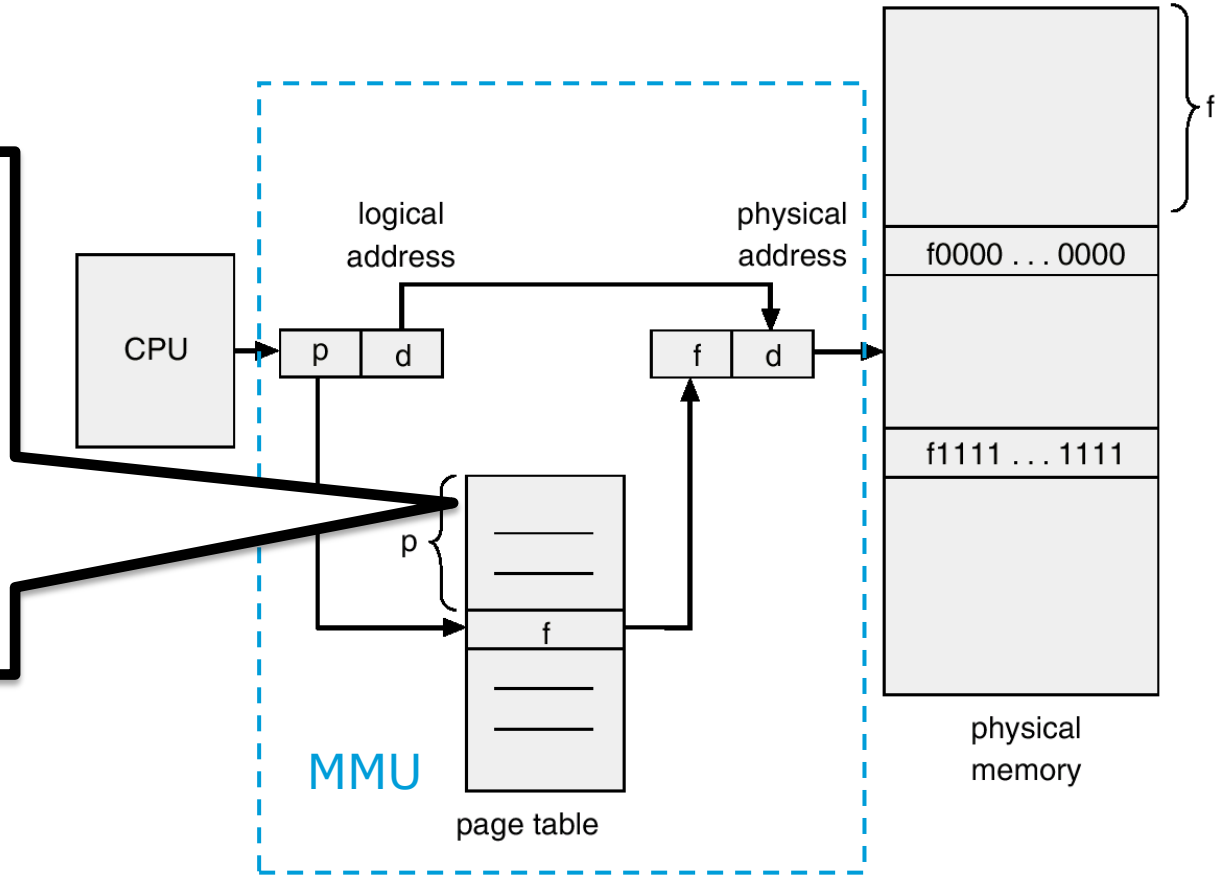
Address Translation Architecture



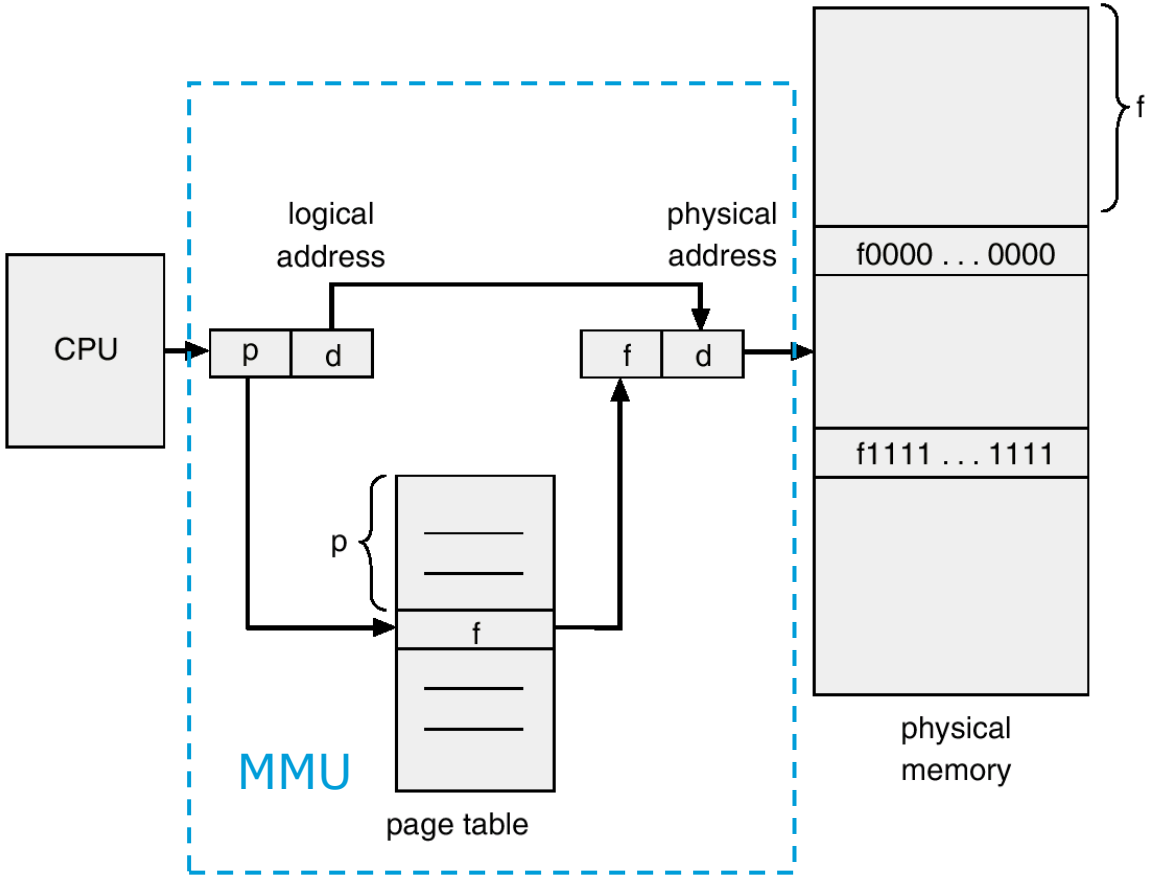
Address Translation Architecture

Page table size: se/p. 1MB process → 2KB *per process*.

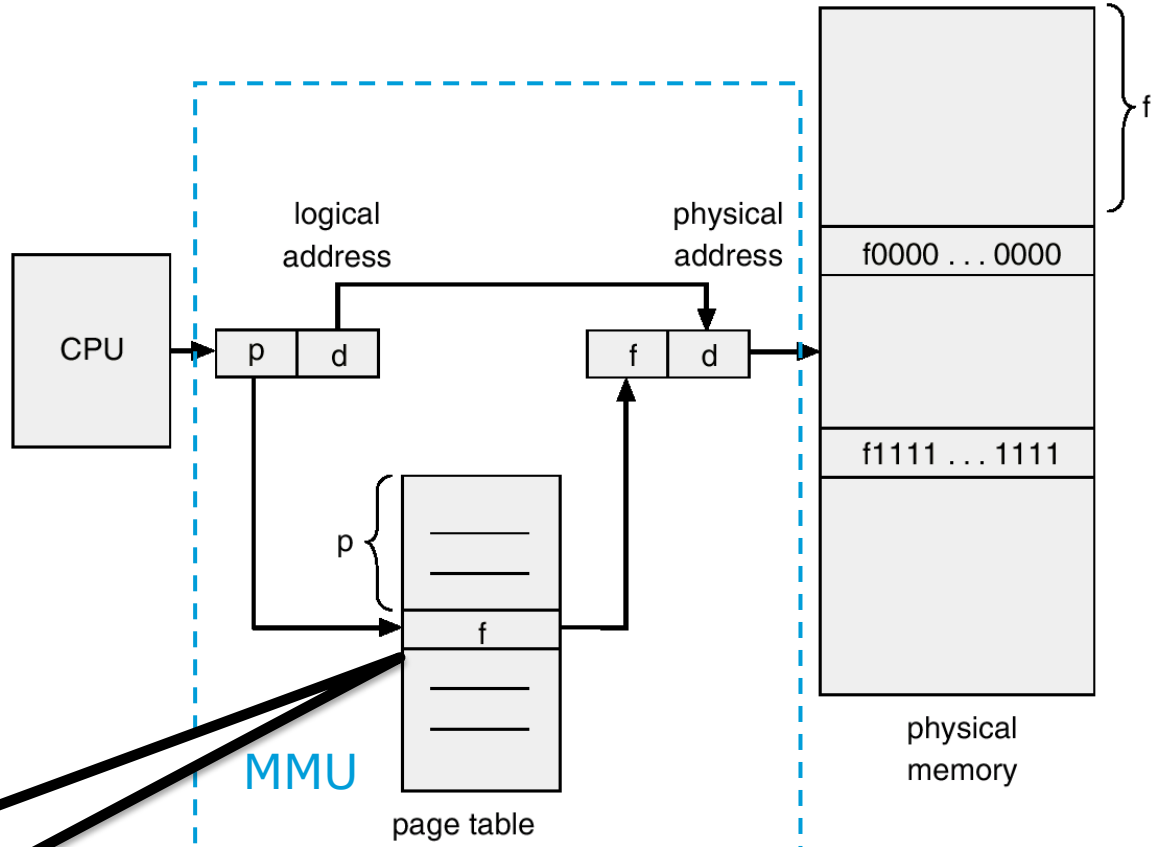
Stored in the physical memory!



Address Translation Architecture

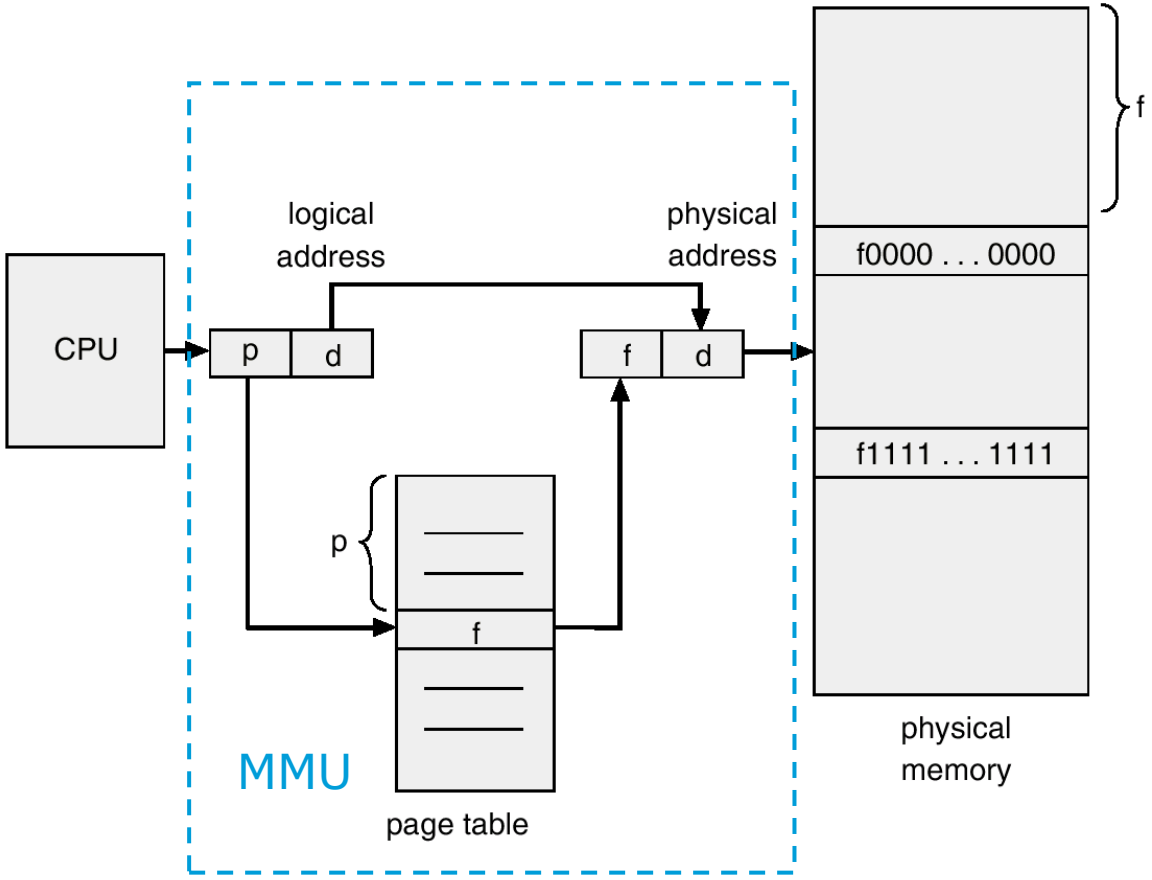


Address Translation Architecture

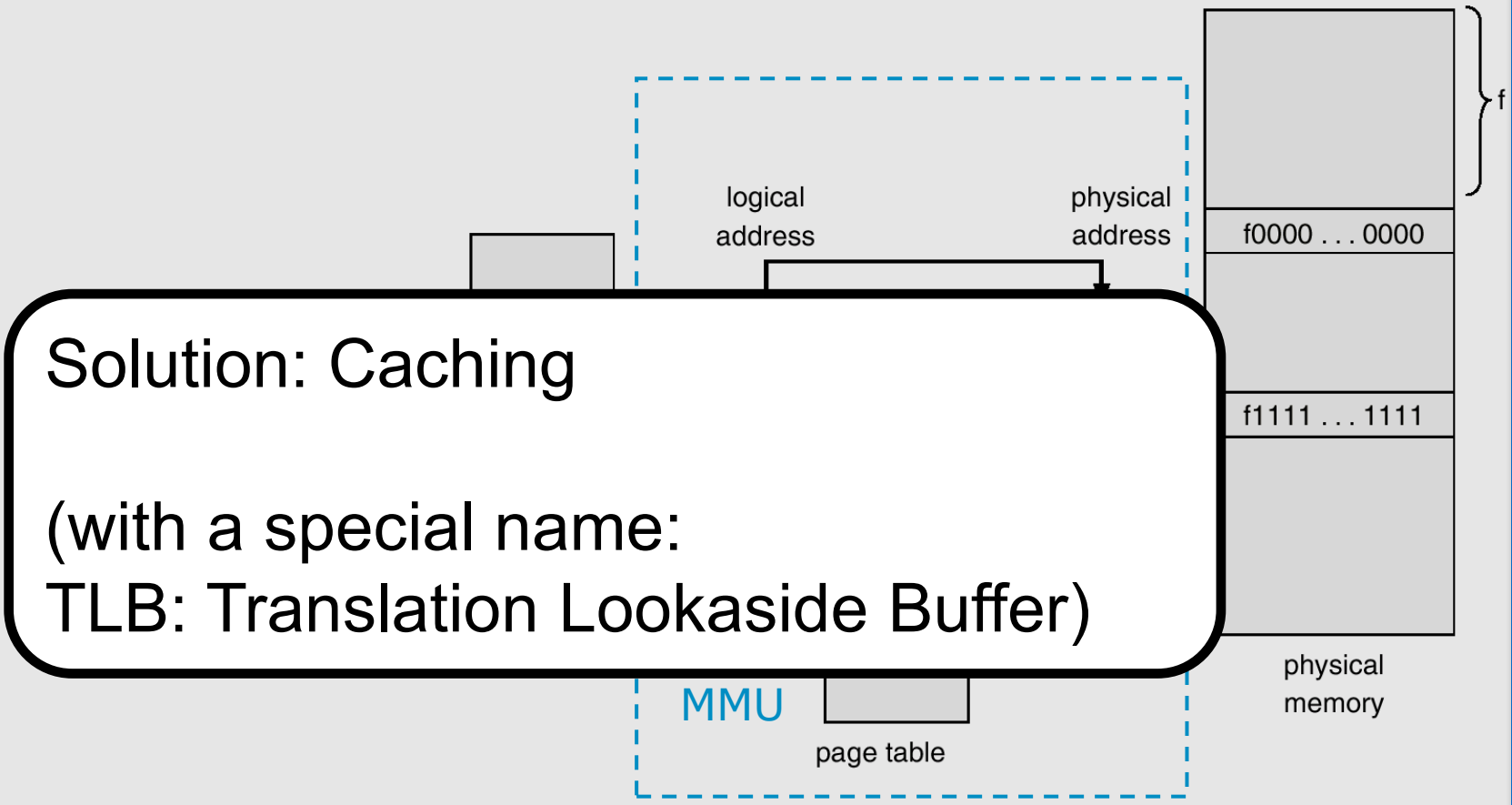


Extra memory access for each memory access!!

Address Translation Architecture



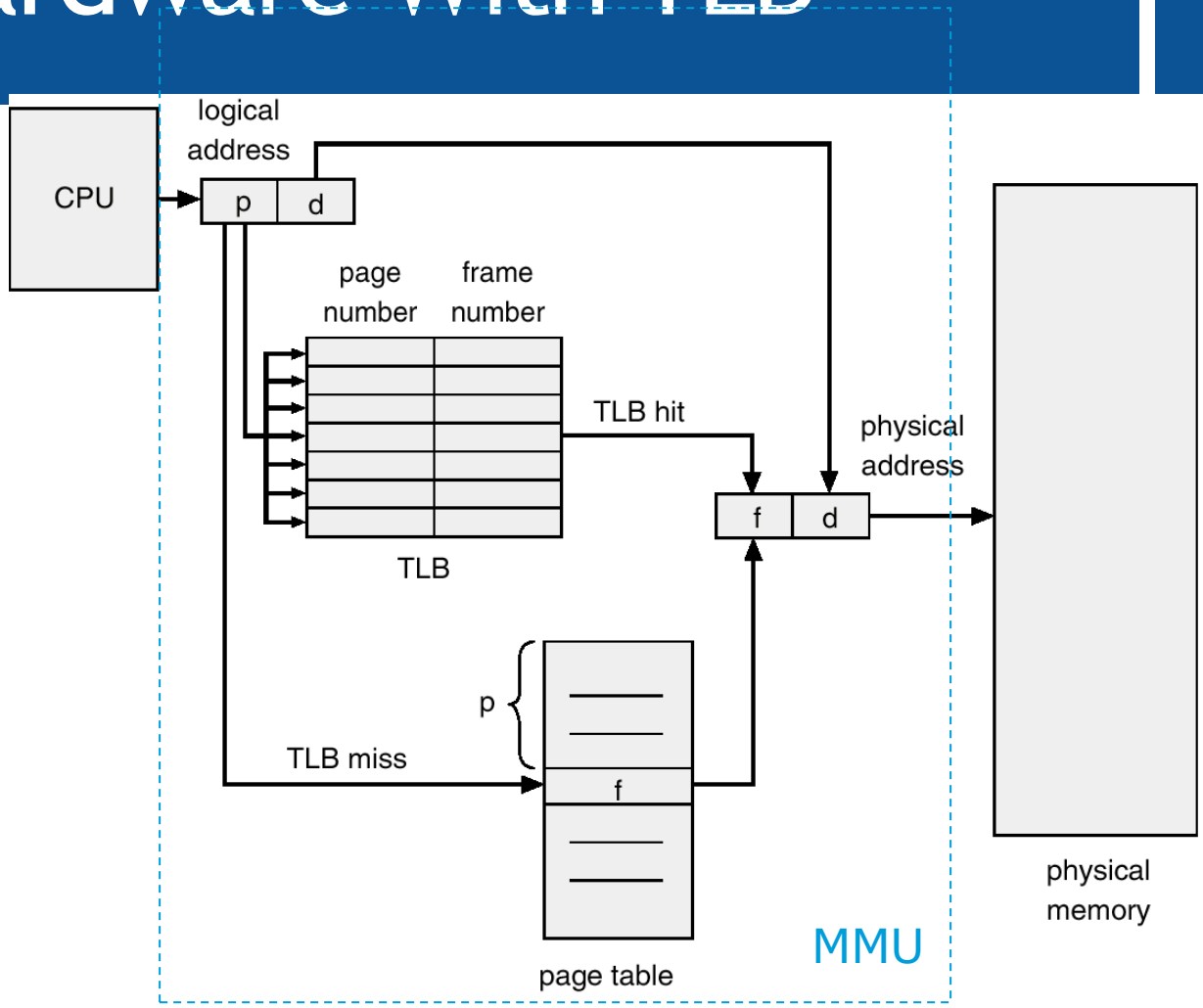
Address Translation Architecture



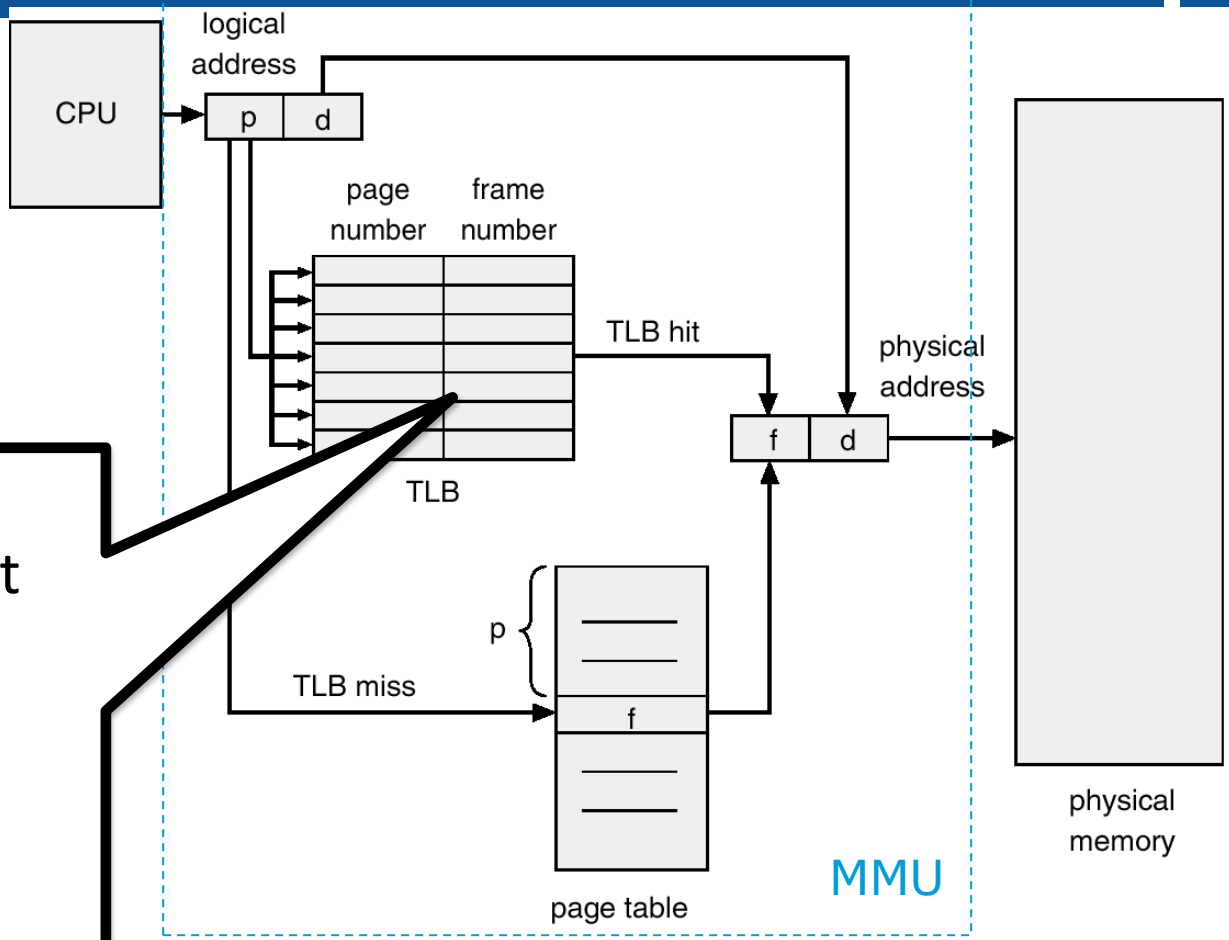
Solution: Caching

(with a special name:
TLB: Translation Lookaside Buffer)

Paging Hardware With TLB

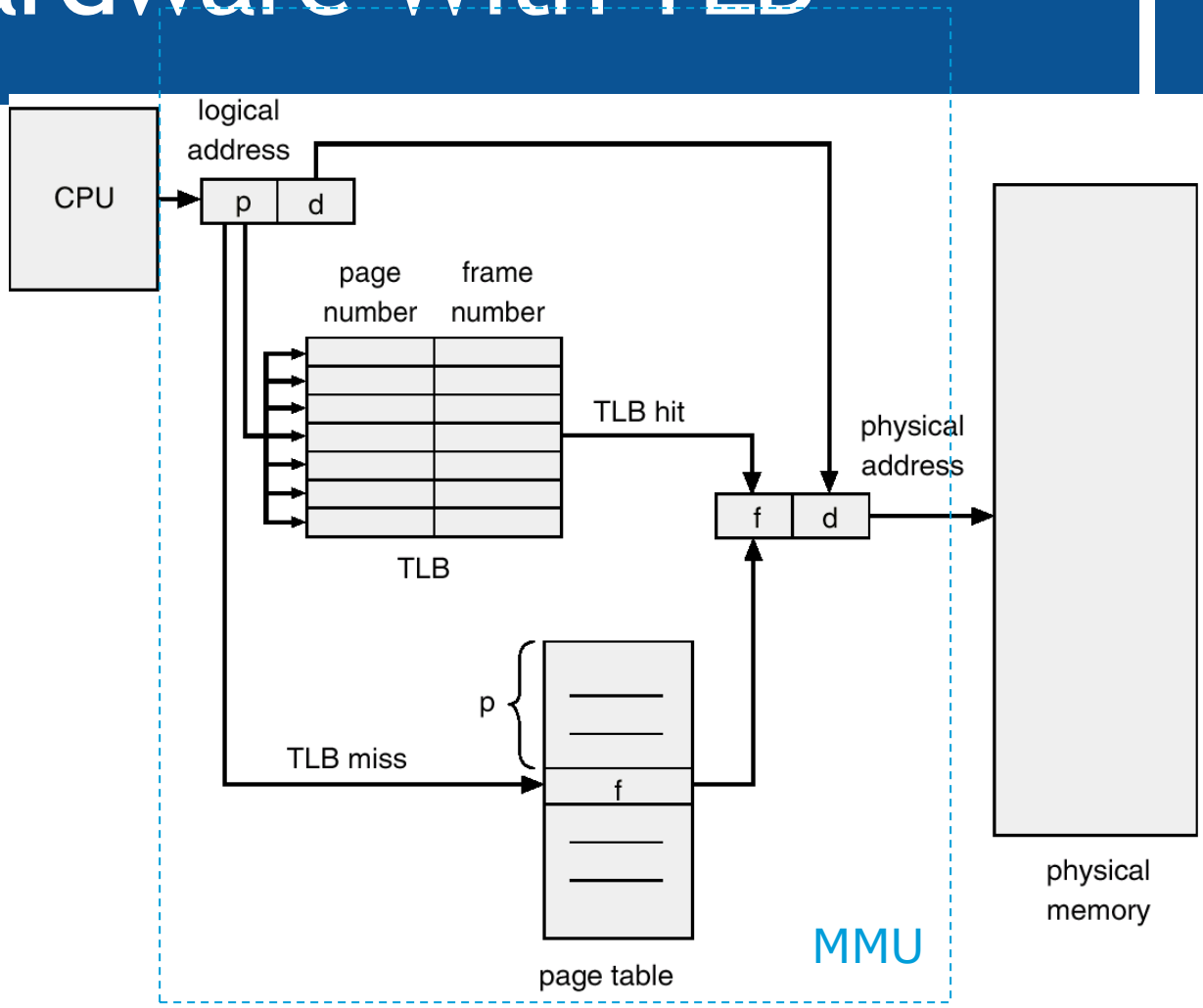


Paging Hardware With TLB



- UsuSmall, about 64 entries
- SRAM/Special hardware (very fast)

Paging Hardware With TLB



Paging

- OS keeps track of free frames
- When a process requires *n* pages, allocate any *n* free frames
  - Remove the frames from the list of free frames
  - Copy data to physical memory
  - Create page table for the process

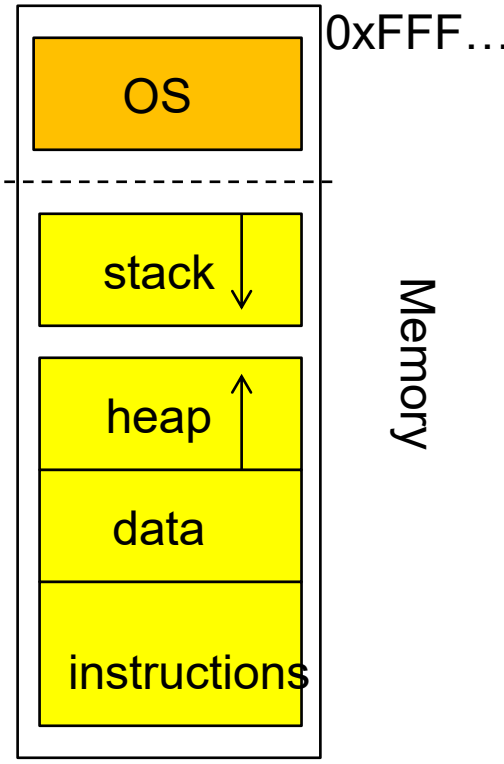
Virtual Memory

Logical vs. Physical Memory

- Which is larger?

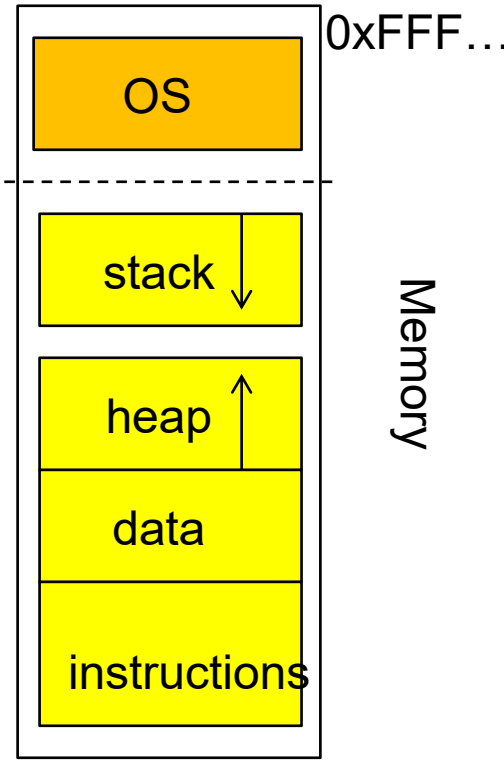
Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory



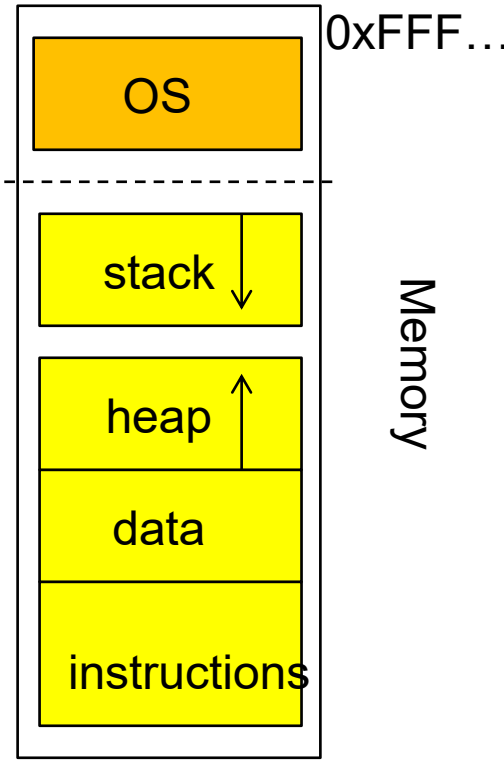
Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory
- The number of processes, in general, is not limited



Logical vs. Physical Memory

- Which is larger?
- **Each process** thinks it runs alone
  - May access the entire physical memory
- The number of processes, in general, is not limited



→ The required logical memory (sum of logical memory of all processes) is much larger than the actual physical memory

Virtual Memory

Virtual Memory

- Only part of the program & data needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows for more process to coexist in the main memory.
  - Allows for more efficient process creation.

Virtual Memory

- Only part of the program & data needs to be in memory for execution.
  - Logical address space can therefore be much larger than physical address space.
  - Allows for more process to coexist in the main memory.
  - Allows for more efficient process creation.
- Not currently used pages can be stored in the disk