OS review

Instructor: Youngjin Kwon

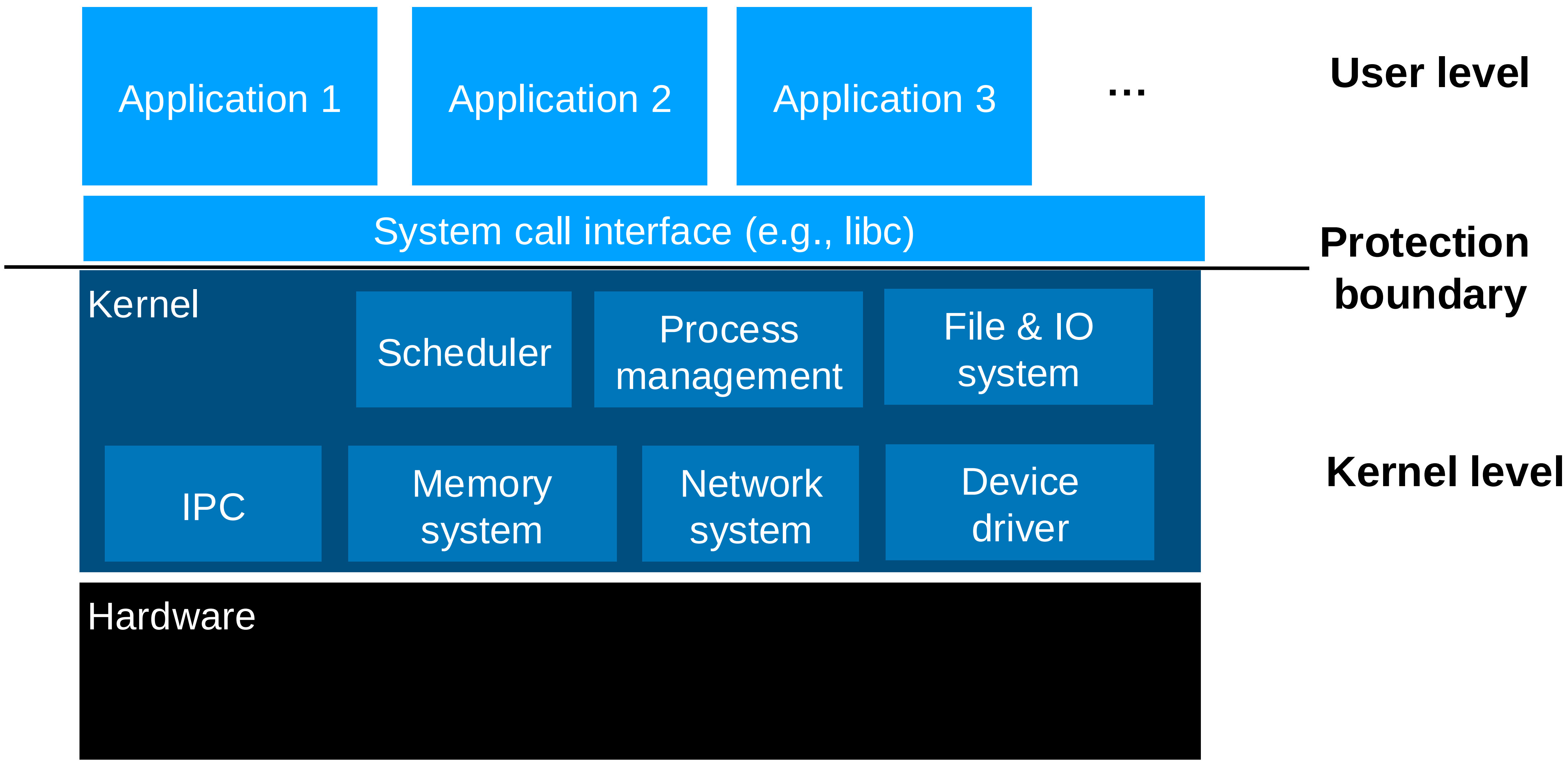
What's going on?

```c
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/time.h>

int main(void)
{
  void *addr;
  struct timeval start, end, elap;

  addr = mmap(NULL, 1 << 20, PROT_READ | PROT_WRITE,
      MAP_ANONYMOUS| MAP_PRIVATE, -1, 0);

  gettimeofday(&start, NULL);
  memset(addr, 1, 1 << 20);
  gettimeofday(&end, NULL);

  timersub(&end, &start, &elap);

  printf("Time taken to memset1 %ld usec\n", elap.tv_usec);

  gettimeofday(&start, NULL);
  memset(addr, 2, 1 << 20);
  gettimeofday(&end, NULL);

  timersub(&end, &start, &elap);

  printf("Time taken to memset2 %ld usec\n", elap.tv_usec);

  munmap(addr, 1 << 20);

  return 0;
}
```

MAP size is way beyond CPU cache size!

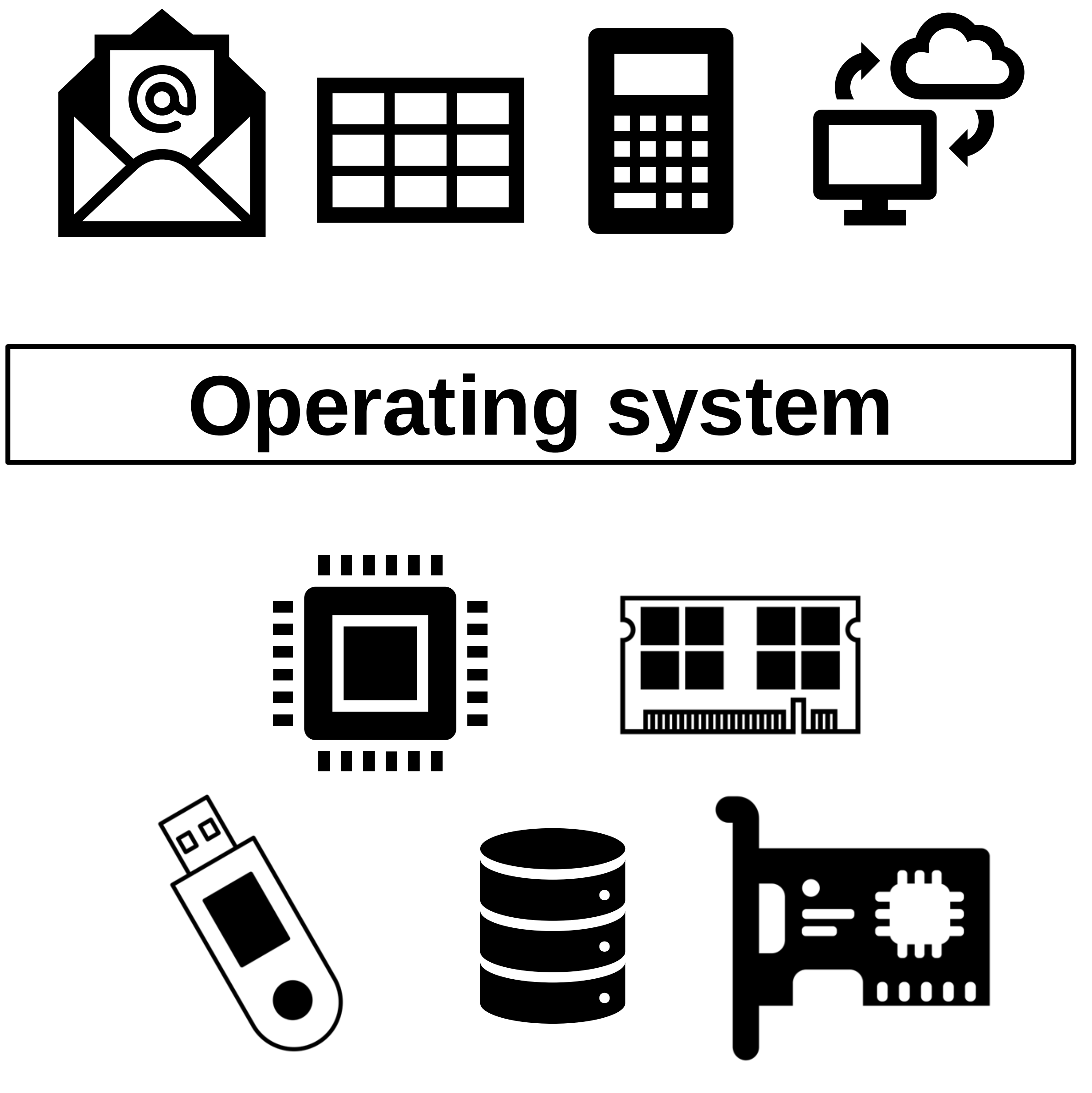
```
#include <stdio.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/time.h>

#define MAP_SIZE (1 << 30)

int main(void)
{
  void *addr;
  struct timeval start, end, elap;

  addr = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
      MAP_ANONYMOUS| MAP_PRIVATE, -1, 0);

  gettimeofday(&start, NULL);
  memset(addr, 1, MAP_SIZE);
  gettimeofday(&end, NULL);

  timersub(&end, &start, &elap);

  printf("Time taken to memset1 %0.2lf msec\n",
      (((double)elap.tv_sec * 1000000.0) + (double)elap.tv_usec) / 1000.0 );

  gettimeofday(&start, NULL);
  memset(addr, 2, MAP_SIZE);
  gettimeofday(&end, NULL);

  timersub(&end, &start, &elap);

  printf("Time taken to memset2 %0.2lf msec\n",
      (((double)elap.tv_sec * 1000000.0) + (double)elap.tv_usec) / 1000.0 );

  munmap(addr, MAP_SIZE);

  return 0;
}
```

Bird eye view of OS



Why OS is required?

- To programs,
  - Providing application programming interface (API) to use hardware
  - Hide details of hardware

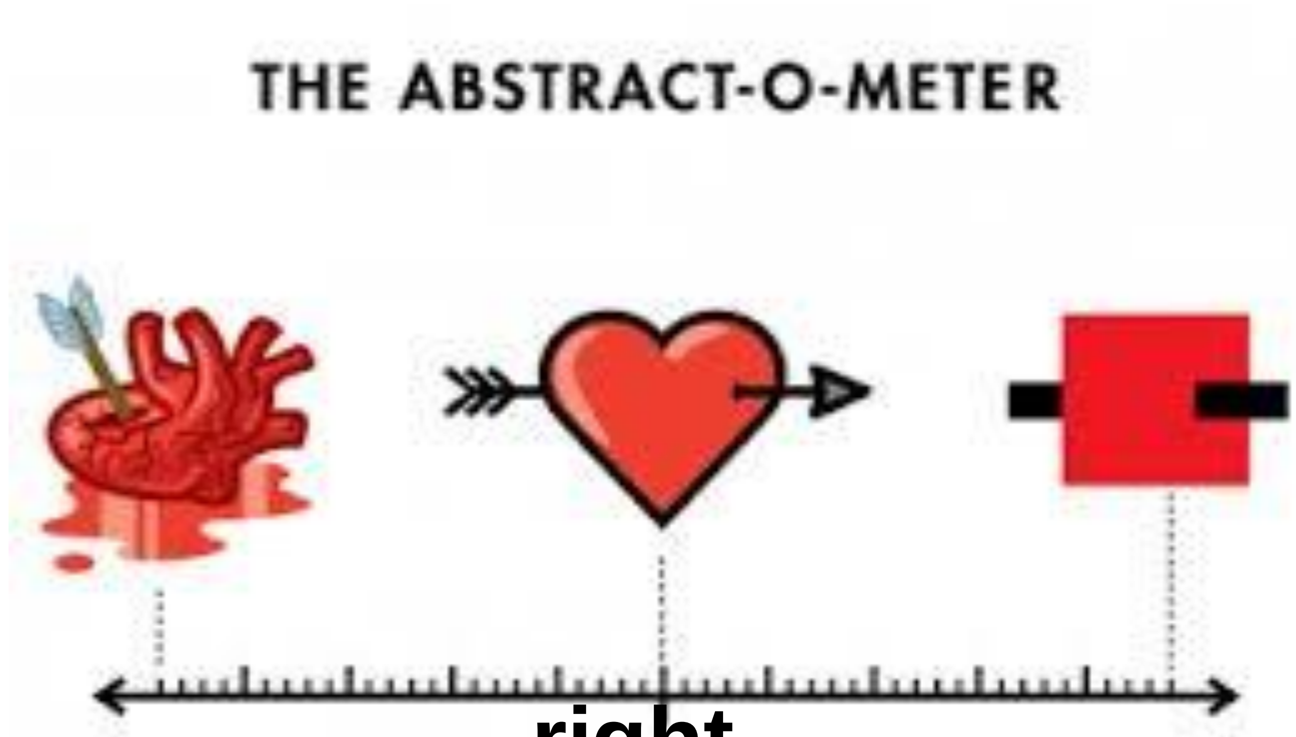


Key roles of OS

- Design **abstractions** to use hardware
  - Define APIs for applications to use
- **Protection & Isolation**
  - Contain malicious or buggy behaviors of applications
  - Protecting OS from malicious or buggy applications
  - Isolating one application from another
- **Sharing** resources
  - Multiplex hardware resources

What is “Abstraction”?

- The **process or outcome** of making something **easier to understand** by **ignoring some of details** that may be unimportant



Too realistic **right abstraction** **Too abstract**

OS designers’ first thought

- No one want to write programs directly handling hardware details (**easy to program**)
- To utilize hardware resources, OS has to run multiple applications (**management unit of execution**)
- Protect applications from each other (**protection unit of execution**)

What is the conclusion?

Building an abstraction that gives an illusion that each application runs on a single machine

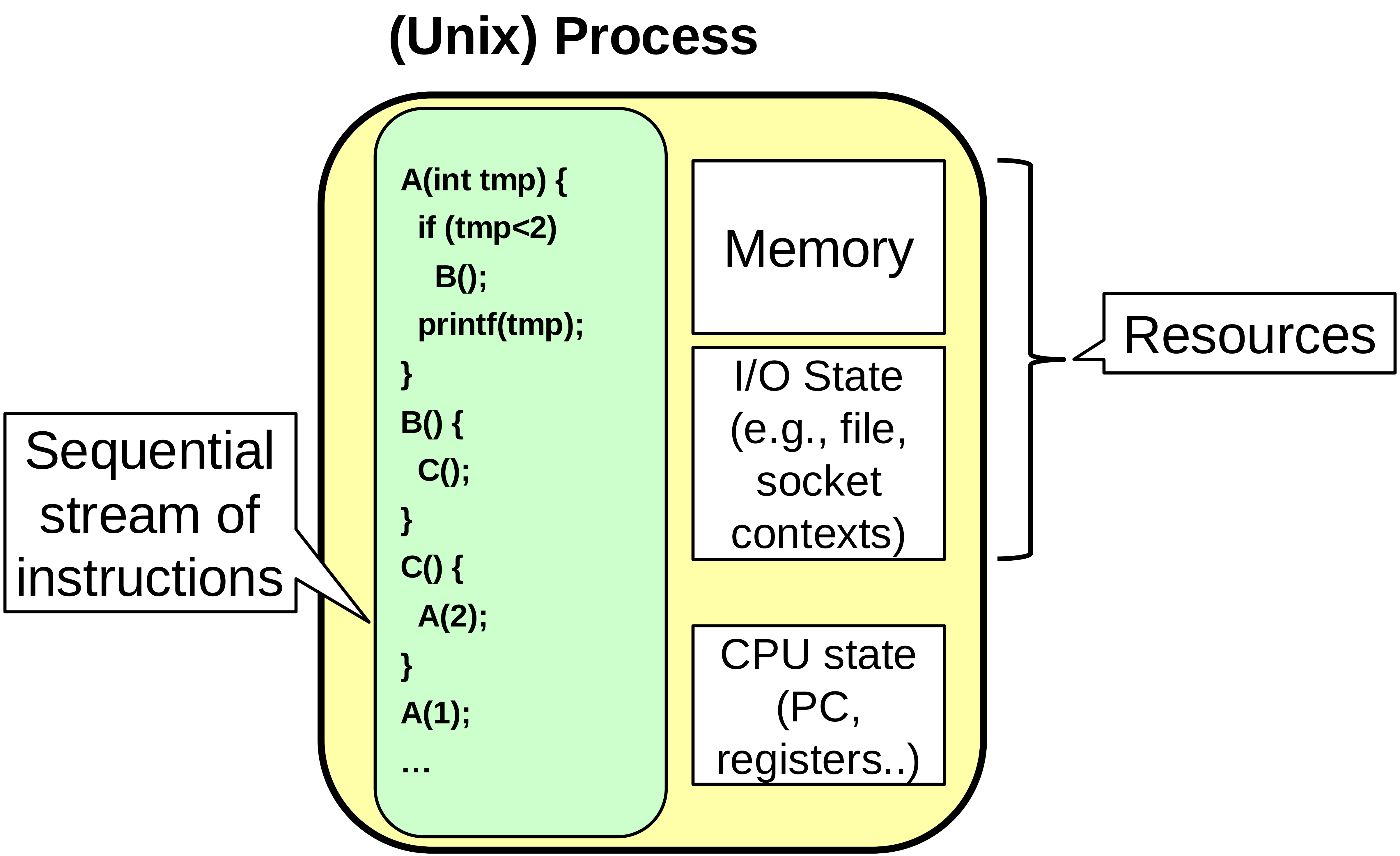
Let’s call it process (= executed application)

How to make it easy to use hardware?

- OS designers **build each abstraction** of hardware resources and **bind it to process**
  - CPU -> Virtualizing CPU
  - Memory -> Virtual address space
  - Storage -> File
- OS designers provide APIs to applications to use the abstractions

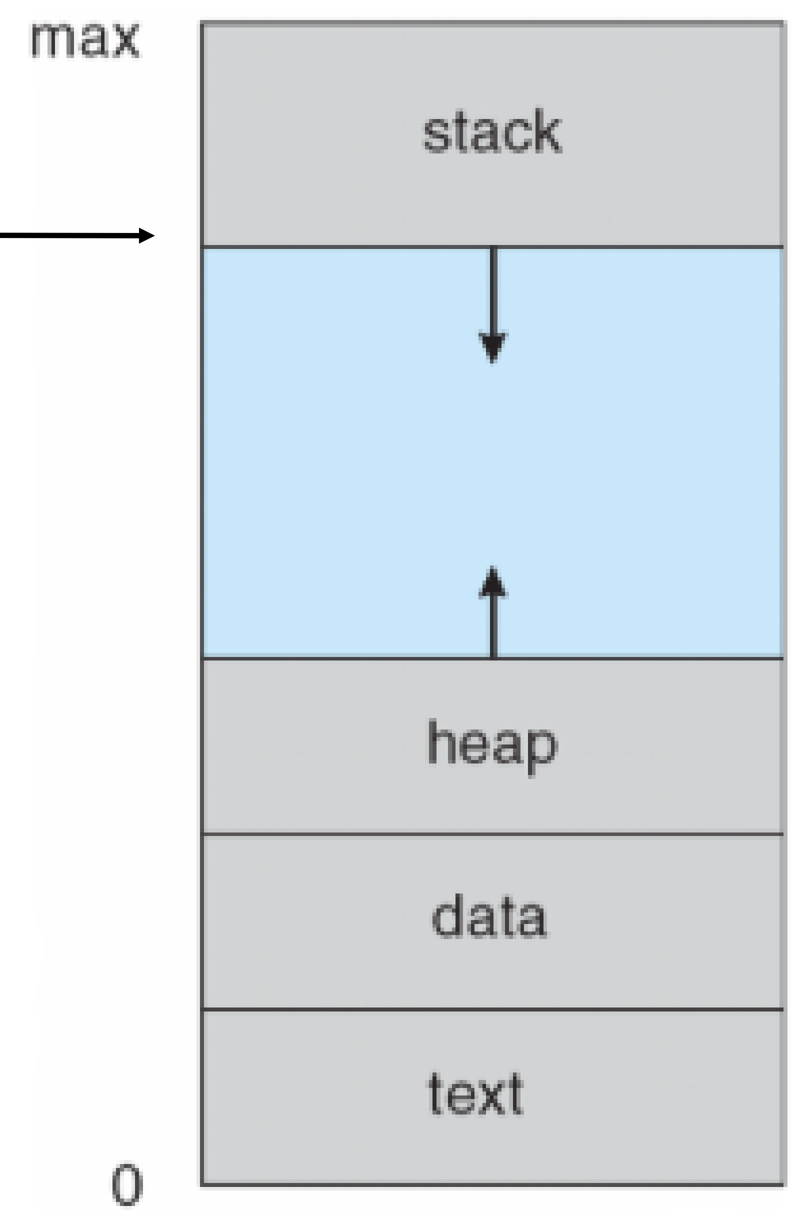
They call them as system calls

Process


(Unix) Process
A(int tmp) {
if (tmp<2)
B();
printf(tmp);
}
B() {
C();
}
C() {
A(2);
}
A(1);
…
Memory
I/O State (e.g., file, socket contexts)
CPU state (PC, registers..)
Resources
Sequential stream of instructions

What process abstracts?

- Each process has its own view of **(   )**
  - Own address space
  - Own virtual CPU
  - Own files

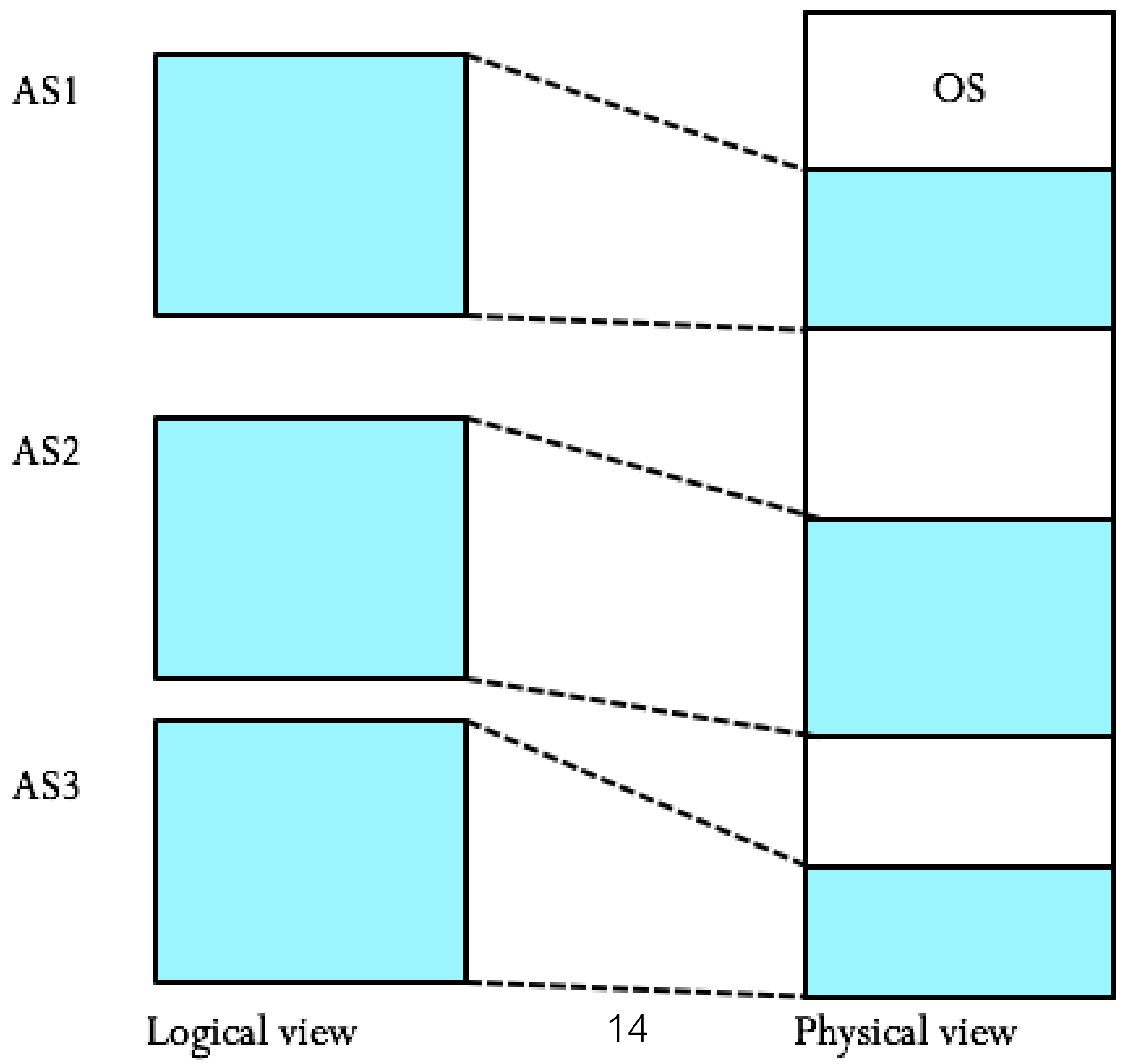


Nice clean abstraction!

The next question is how to design each abstraction?

Abstraction of address space

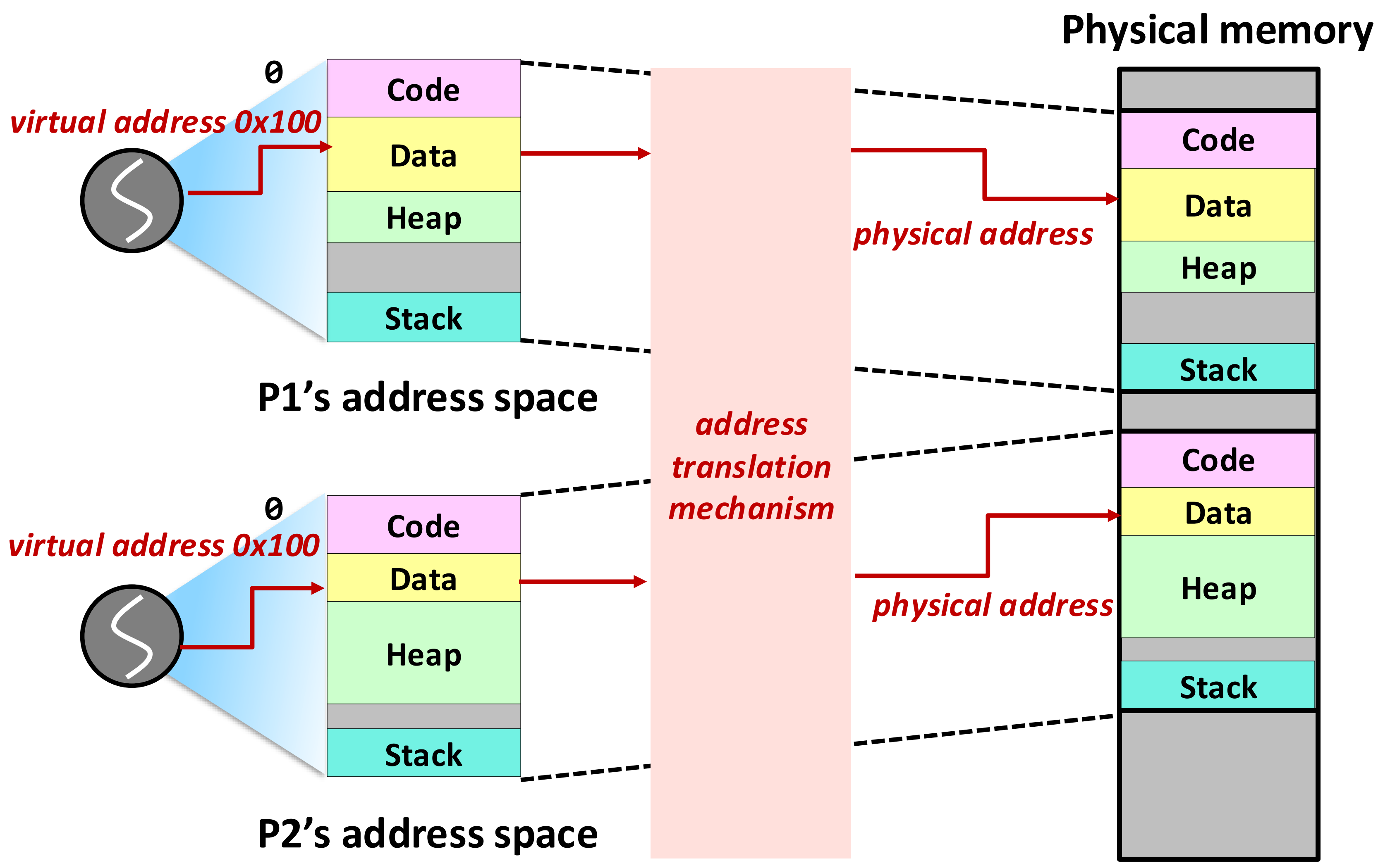
- How to associate virtual address to physical address?
  - Divide each physical memory to small chunk (called page)
  - Create mapping from virtual to physical address



Virtual address layout

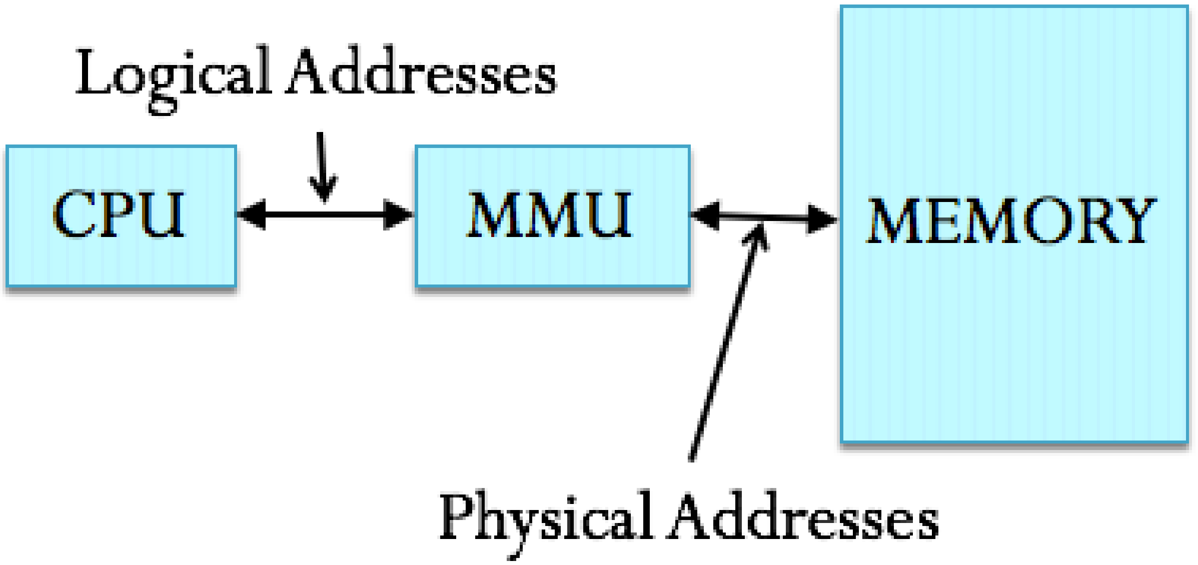
Virtual memory: Level of Indirection

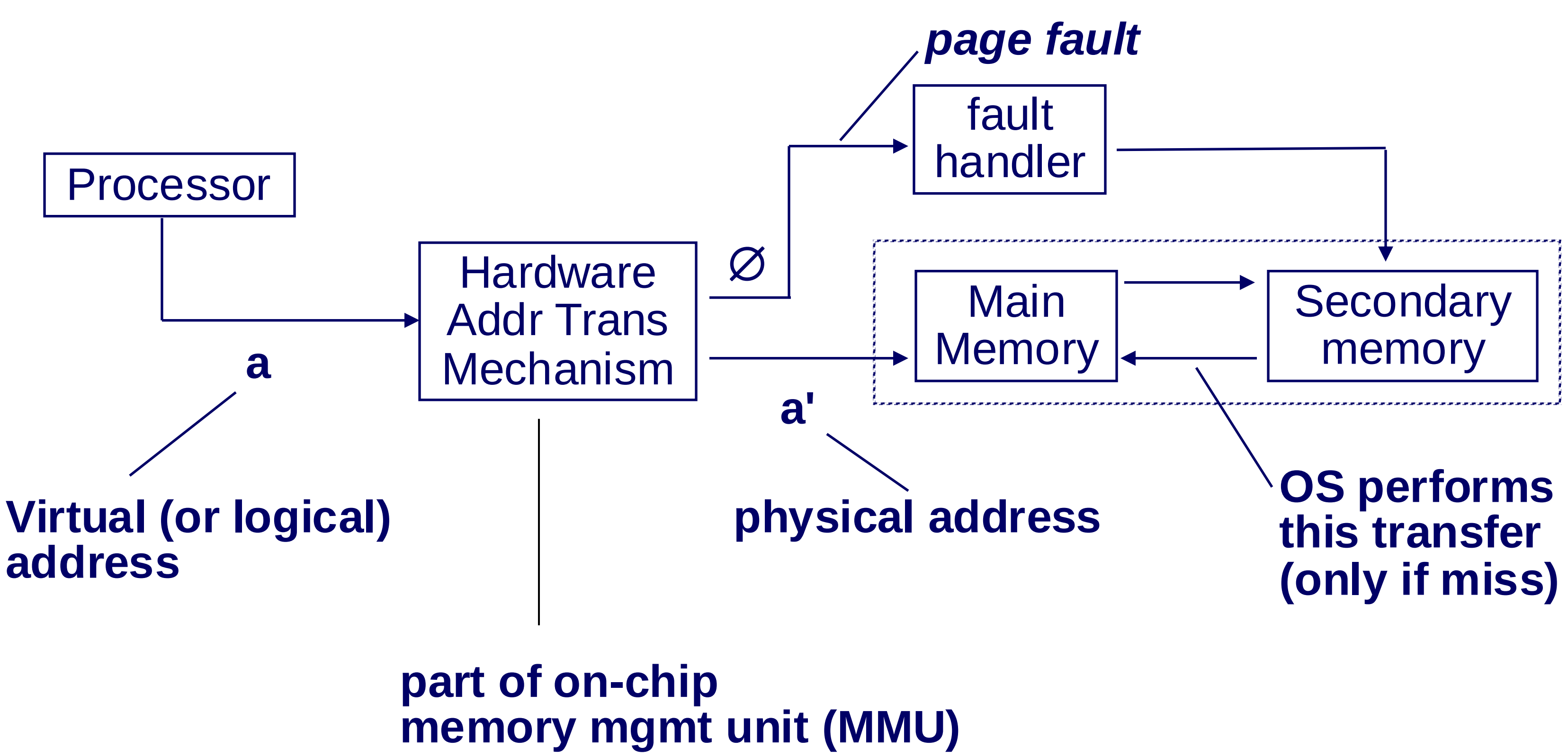
Level of indirection



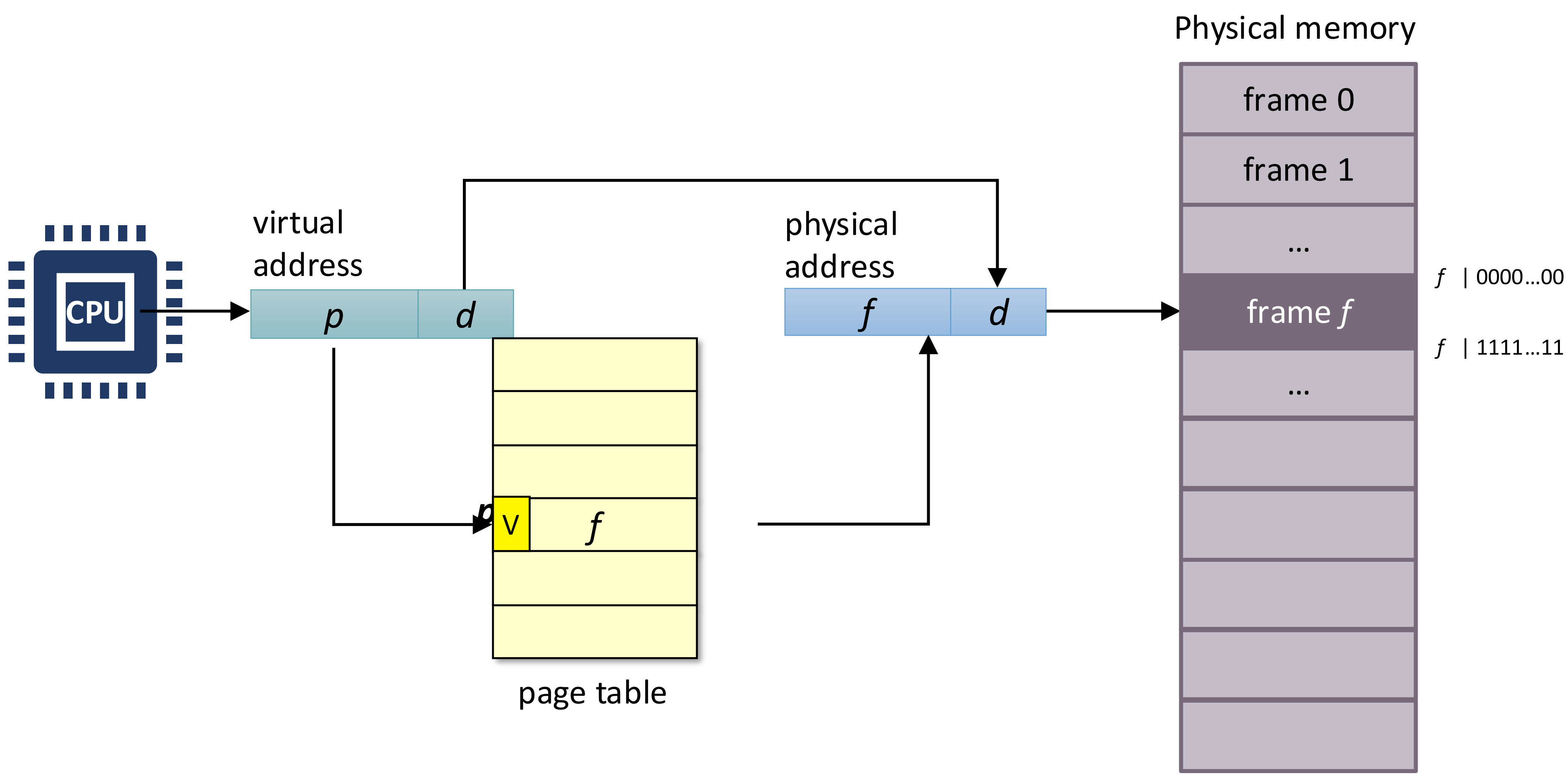
Abstraction of address space

- How to map virtual to physical address?
  - Segmentation
  - Paging (Single-level, multi-level …)
  - Segmented paging





Address translation: Paging



- TLB caches page table entries
- Page number: logical address
- Frame number: physical address

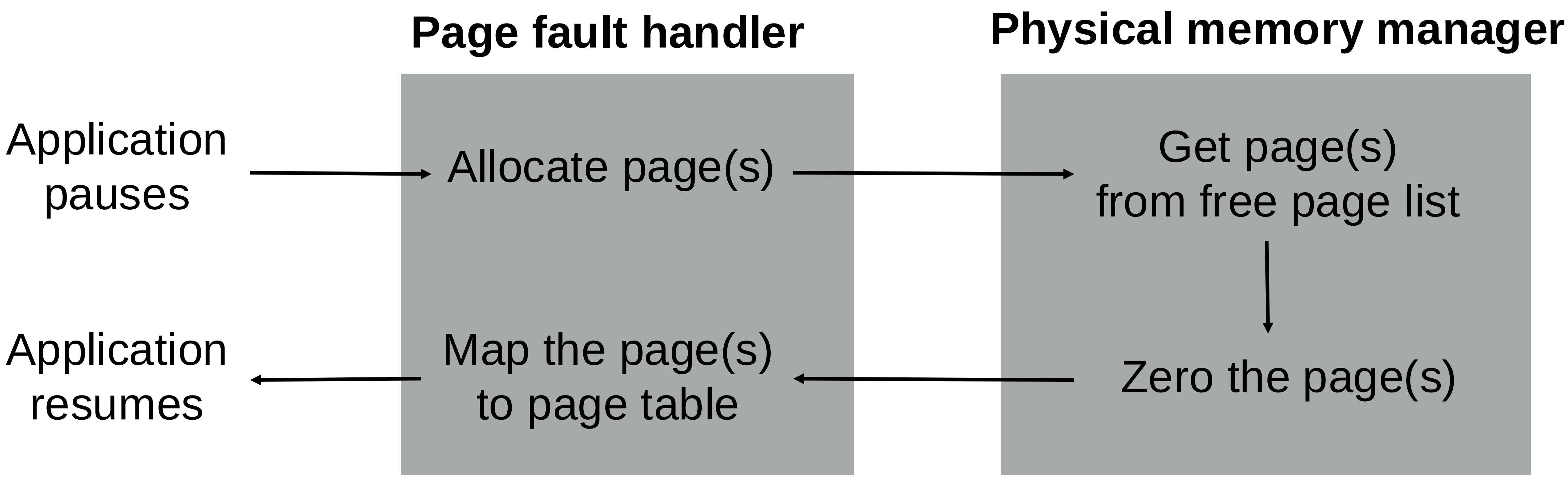
Abstraction of address space

Think about these questions

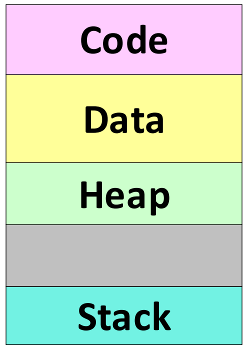
- Where is the page tables stored?
- What are role(s) of software (OS) for paging?
- What are role(s) of hardware for paging?

When to allocate physical memory?

- Demand paging
  - Application first accesses unallocated physical memory

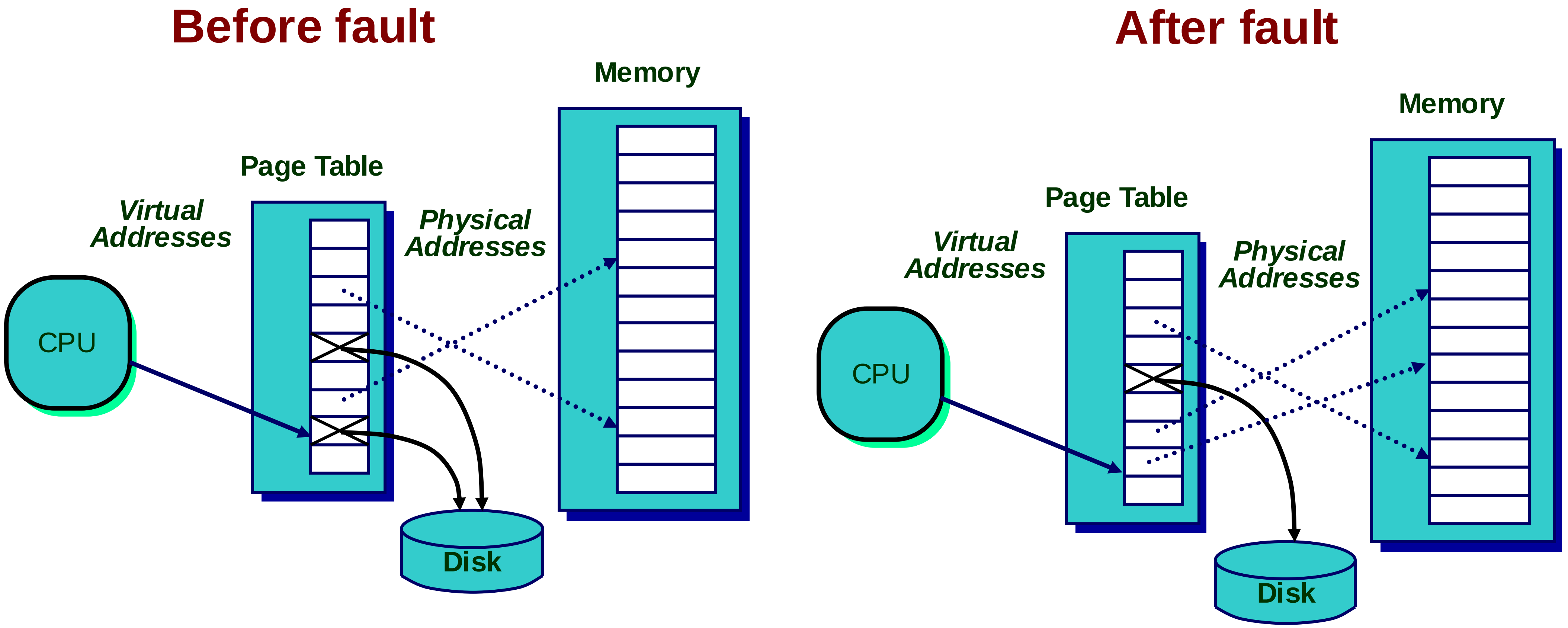


Page fault handling



Two types of memory: (and)

How to group?



Page fault handling

Processor signals controller

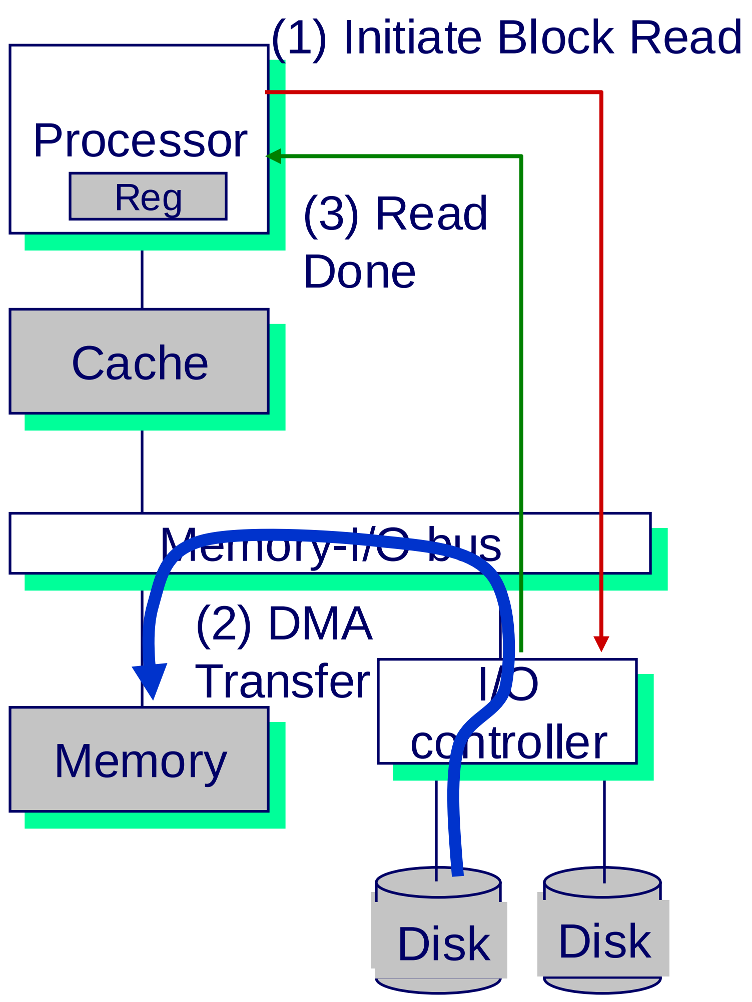
- Read block of length P starting at disk address X and store starting at memory address Y

Read occurs

- Direct Memory Access (DMA)
- Under control of I/O controller

I / O controller signals completion

- Interrupt processor
- OS resumes suspended process



Abstraction of storage

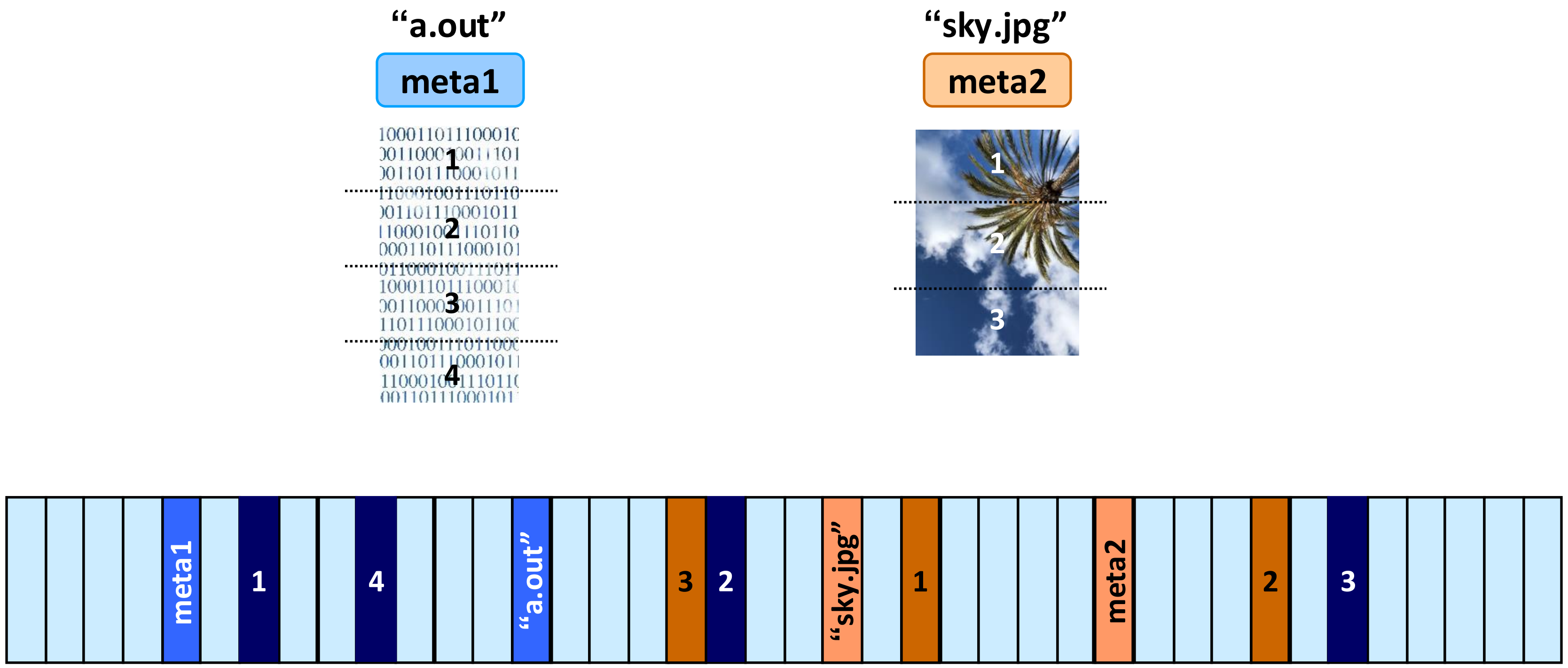
- File: a logical unit of storage
  - Identifier : pathname (path + filename)
  - Location of data is identified by (   )

OS subsystem maps the file to physical storage
Let's call it (   )

- Analogy
  - Virtual memory is an abstraction of physical memory
    - Level of indirection: (   ) → (   )
  - File is an abstraction of physical storage
    - Level of indirection: (   ) → (   )

File System: A Mapping Problem

- <filename, data, metadata> → <a set of blocks>



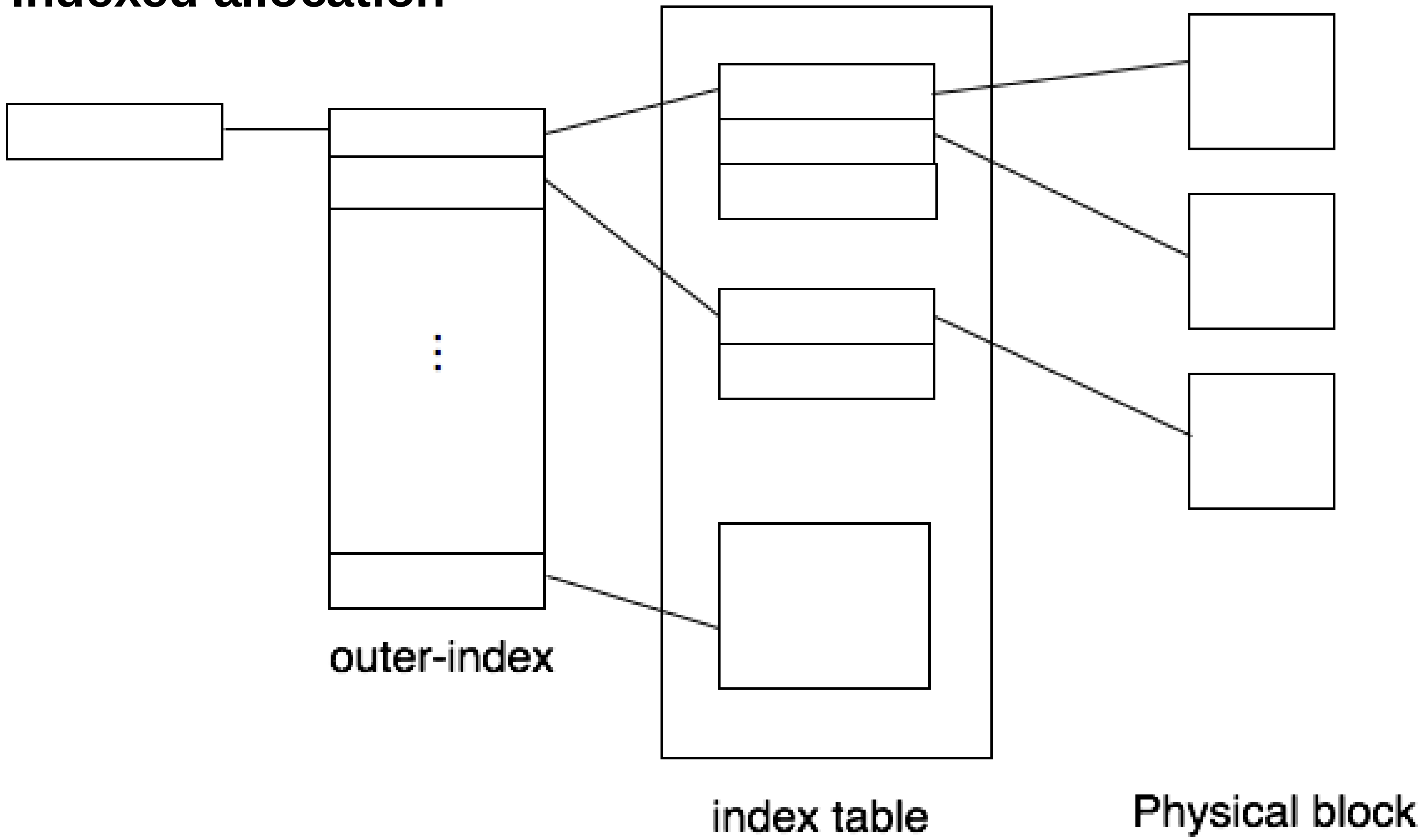
- How to map file to storage media?
  - Divide a file to small chucks (called block)
  - Create mappings from each block to a storage location (called block address)

Abstraction of storage

- How to create the mappings from file to storage?

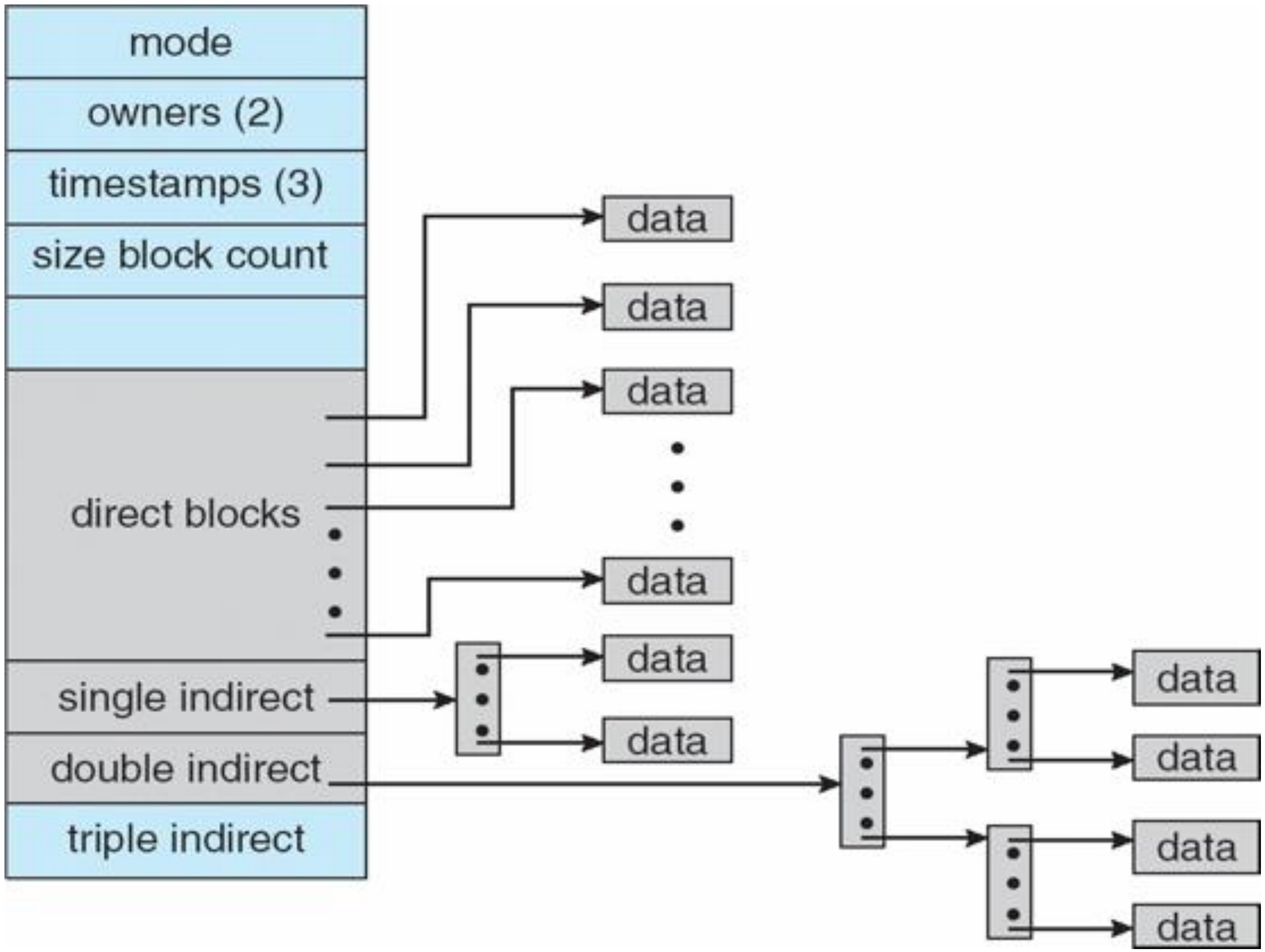
File’s logical block -> Storage physical block

Indexed allocation



Abstraction of storage

Metadata of a file

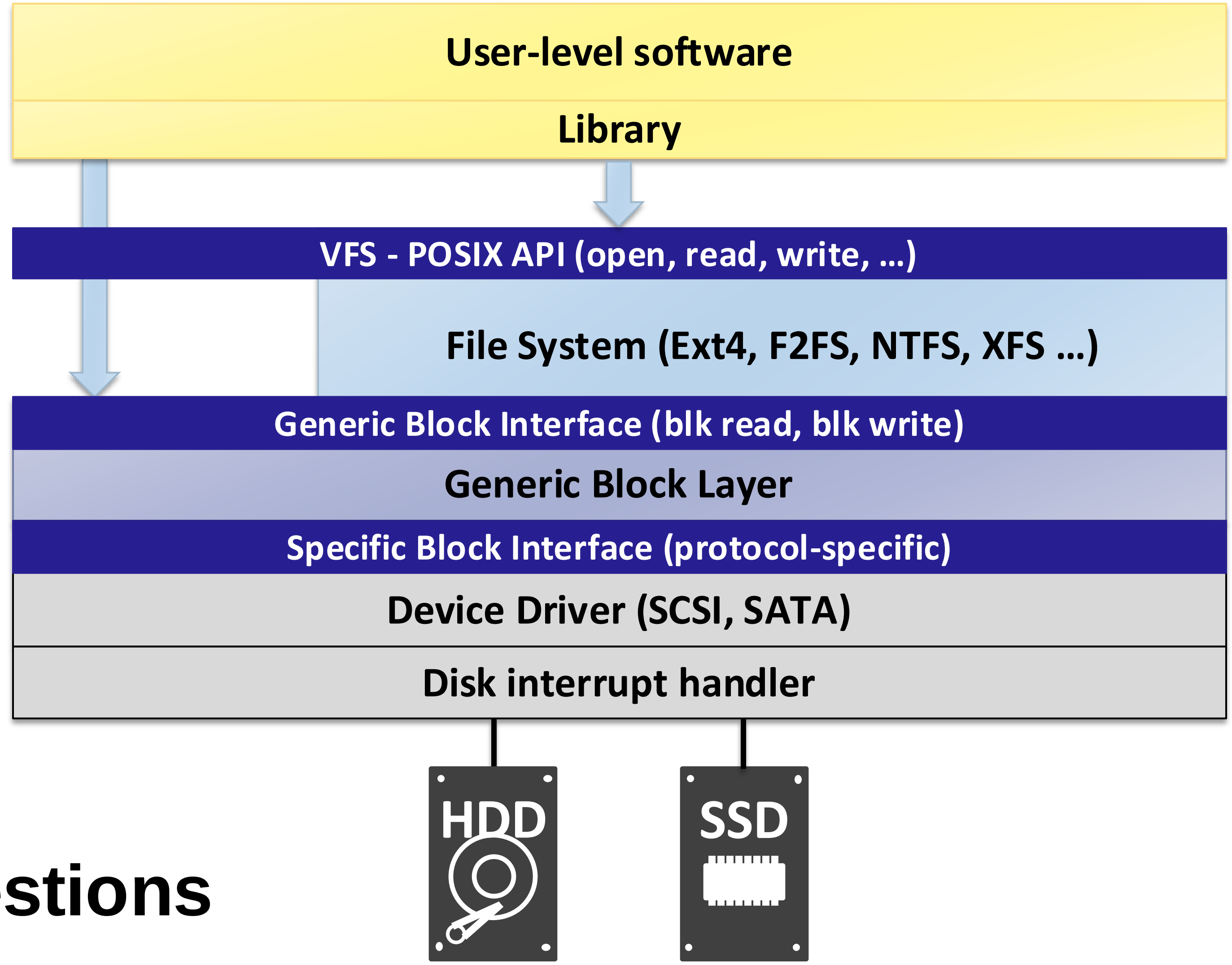


Abstraction of storage

Think about these questions

- Where are the internal nodes in the index? (memory? storage? or both?)
- Does hardware help for the indexing?
  - If yes, what is role(s) of hardware?
  - If not, why?
- When to allocate physical block?
- Any performance optimization for slow storage device?

Storage stack overview

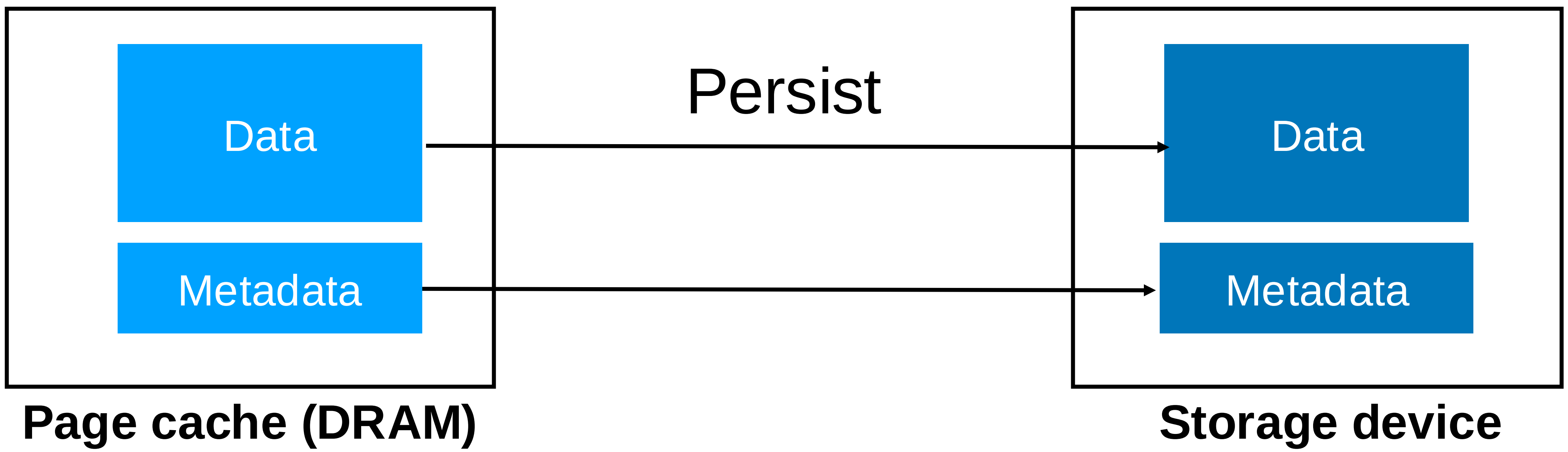


Think about these questions

- Why VFS is required?
- Why Generic Block Layer is required?
- Where is IO queues implemented?

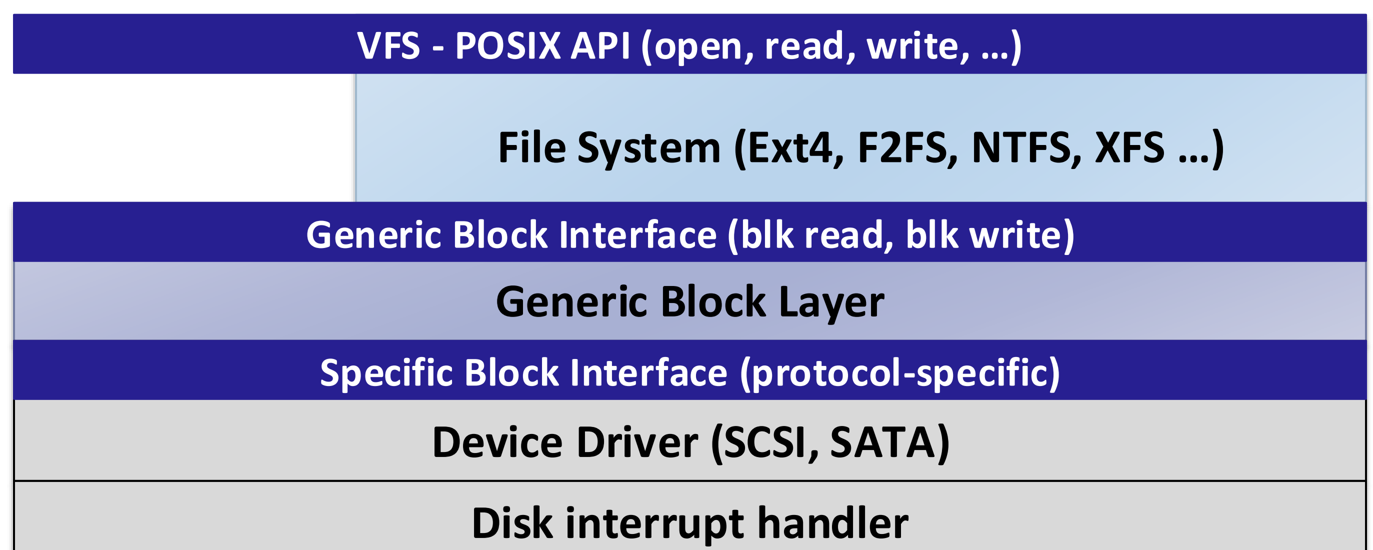
Layer of abstraction

Page cache



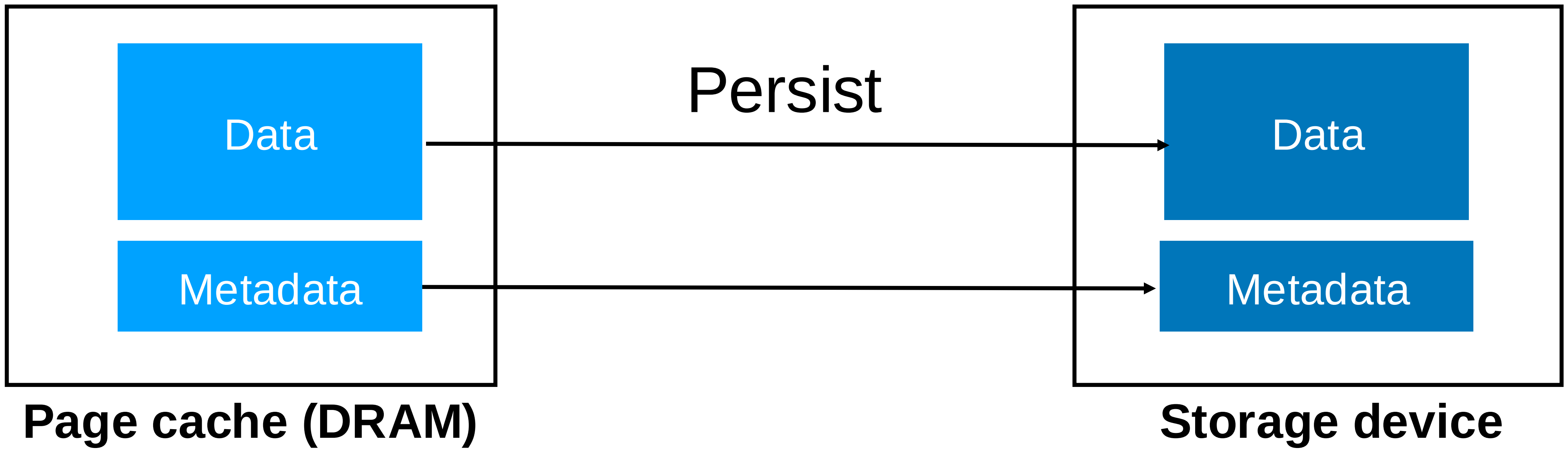
- Kernel read file data to page cache for performance

- What layer includes page cache?



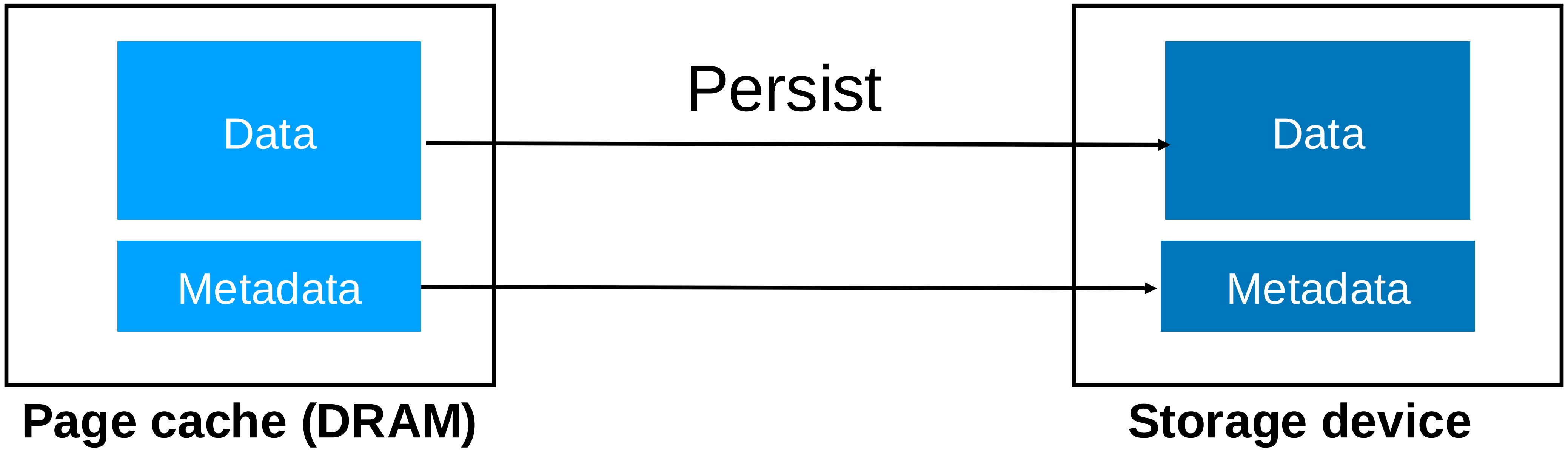
Problems of using buffer cache

Data is stored to two types of devices (two copies)



- Which data is up-to-date?
- When is the data persisted?
- Do you recognize any problems?

Consistency: Atomicity and Durability



- Atomicity: data in memory must be applied to storage device atomically
- Durability: data must be persisted to storage

Name

fsync, fdatasync - synchronize a file's in-core state with storage device

Synopsis

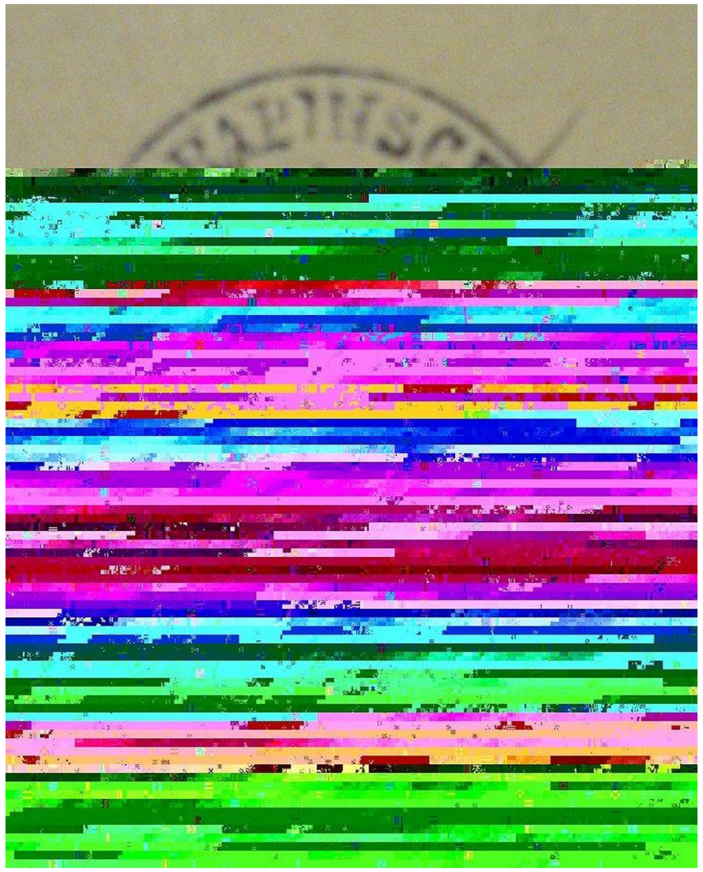
#include <unistd.h>

int fsync(int *fd*);

Non-atomic update

- Each storage has a unit of atomic updates
  - e.g., 4 KB in harddisk
- When you write data bigger than the atomic update size, it is possible to

OS may reorder data when writing to storage



Crash consistency example

- Assume storage can update 1B atomically

Goal

/a/file

Foo

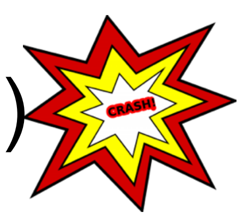
Update →

/a/file

Bar

1. A single write

write(/a/file, “Bar”)



Possible cases

Fao

For

…

Not atomic!

Crash consistency example

2. Rollback logging

creat(/a/log)

write(/a/log, “Foo”)

write(/a/file, “Bar”)

unlink(/a/log)

Reordered and

Possible cases

Fao

For

...

3. Rollback logging with ordering

creat(/a/log)

write(/a/log, “Foo”)

fsync(/a/log)

write(/a/file, “Bar”)

fsync(/a/file)

unlink(/a/log)

Possible cases

Fao

For

...

***/a/* may not contain**

***/a/*log**

Crash consistency example

4. Correct version

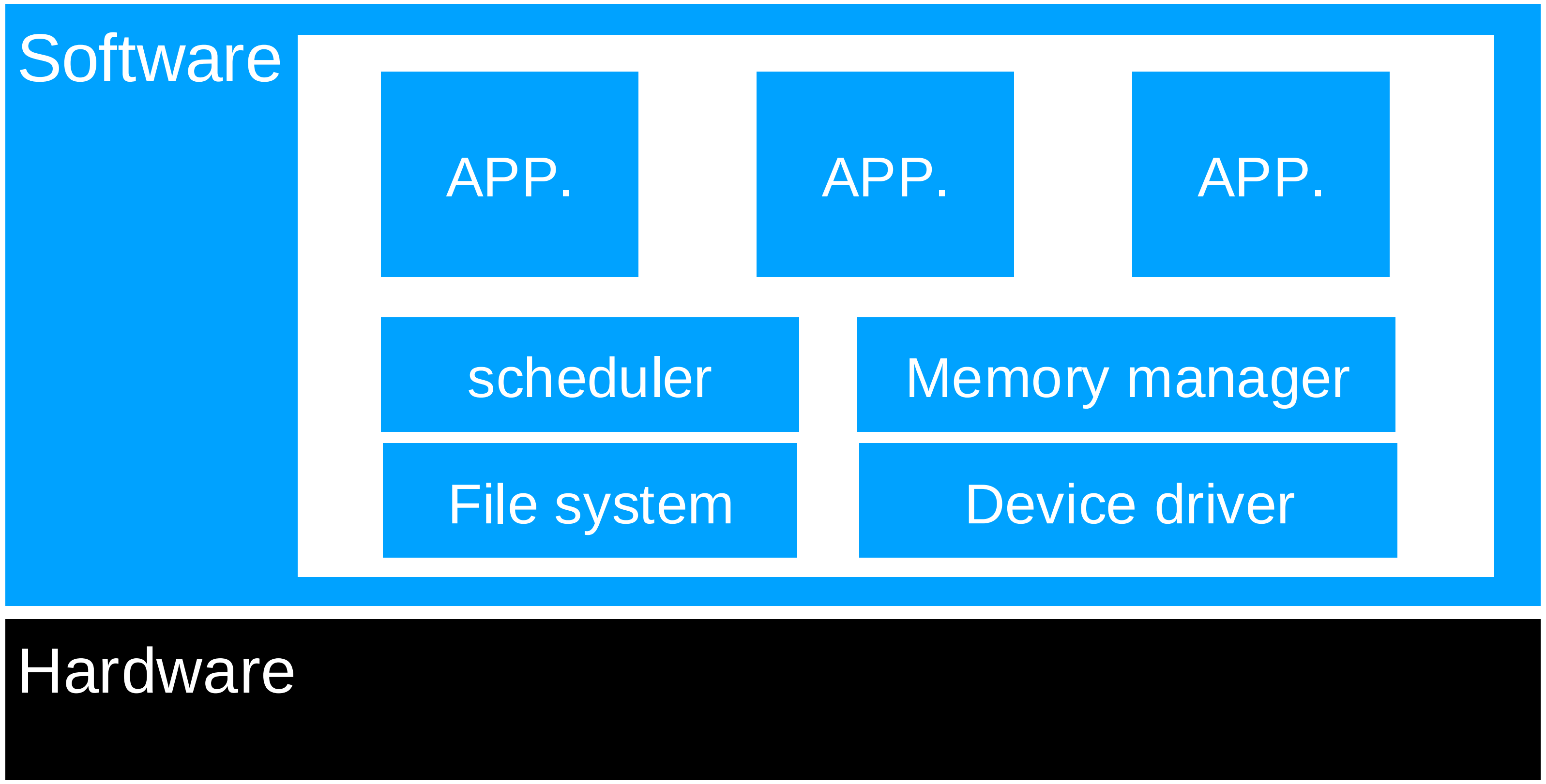
creat(/a/log)
write(/a/log, “Foo”)
fsync(/a/log)
fsync(/a/)
write(/a/file, “Bar”)
fsync(/a/file)
unlink(/a/log)

Must understand
atomicity, ordering, and durability
(including directory)

Key roles of OS

- Provide **abstraction** to use hardware
  - Provide APIs and semantics to applications
- **Protection & Isolation**
  - Contain malicious or buggy behaviors of applications
  - Protecting OS from malicious or buggy applications
  - Isolating one application from another
- **Sharing** resources
  - Multiplex hardware resources

The first design



- Any problems?
- Applications can do
  - Crash OS subsystems
  - Read and modify other applications' data
  - Hoard CPU time

Design: how to archive protection?

- Preventing applications from executing some important **instructions**
  - e.g., shutdown machine, load other applications' page table
- Preventing applications from reading/writing other applications' **memory**
- OS must **regain control** from applications
  - An application may go to infinite loop

Requirement for protection

- **Privileged instruction**
  - Preventing applications from executing some important **instructions**
- **Memory protection**
  - Preventing applications from reading/writing other applications' **memory**
- **(Timer) interrupt**
  - OS must regain control from applications

Separation of privilege

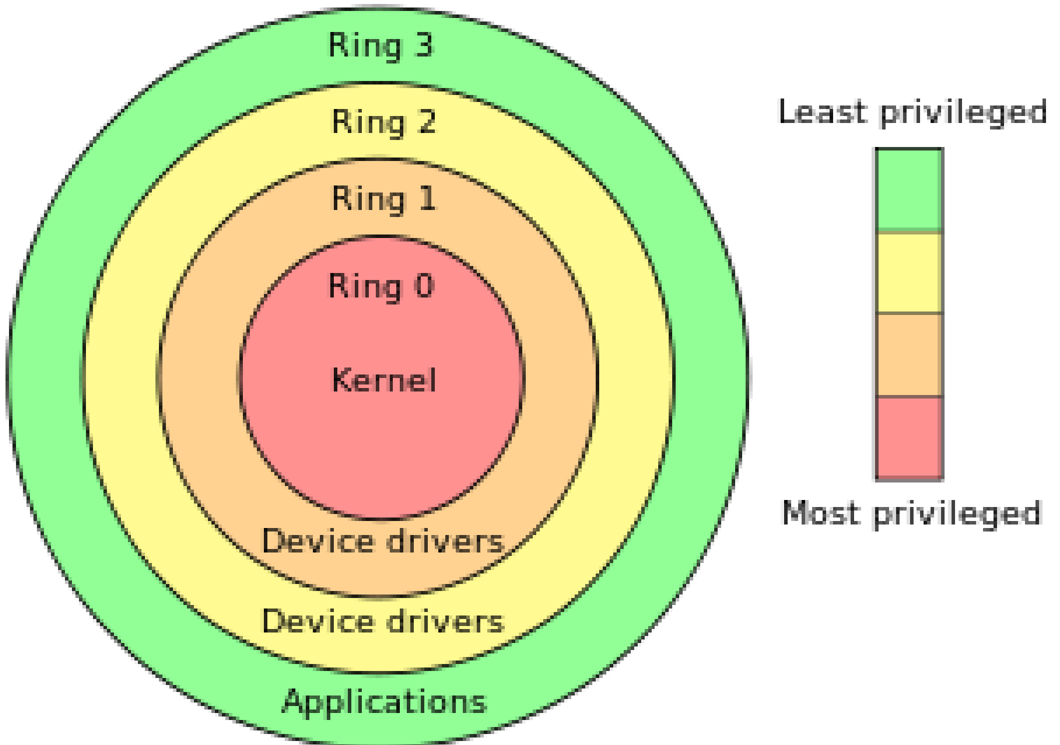
- Clearly, OS must have higher privilege than application
- How can guarantee (or define) the privilege level?

- HW vs SW. who has more privilege?

- HW endorses higher privilege to OS
  - OS can execute privileged instructions
  - OS can have privileged memory to prevent application from accessing OS code and data

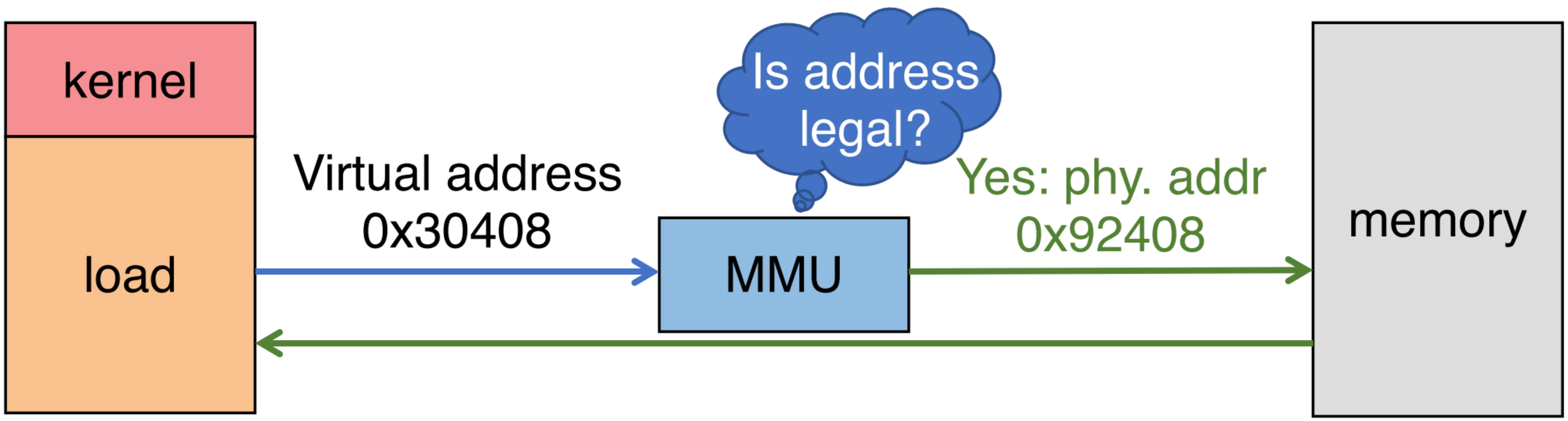
Hardware Protection Mechanisms

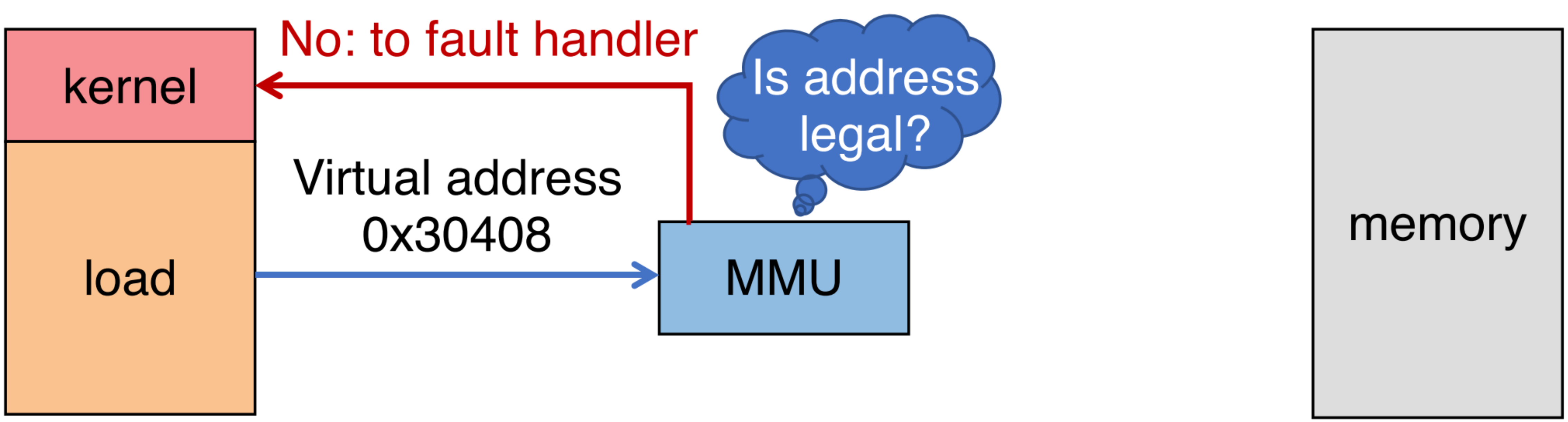
- Protection mechanisms
  - Dual mode operation (or ring mode)
    - mode bit is provided by hardware
  - Privilege I/O instructions
  - Memory protection mechanism (later in this semester)



Address Translation Concept

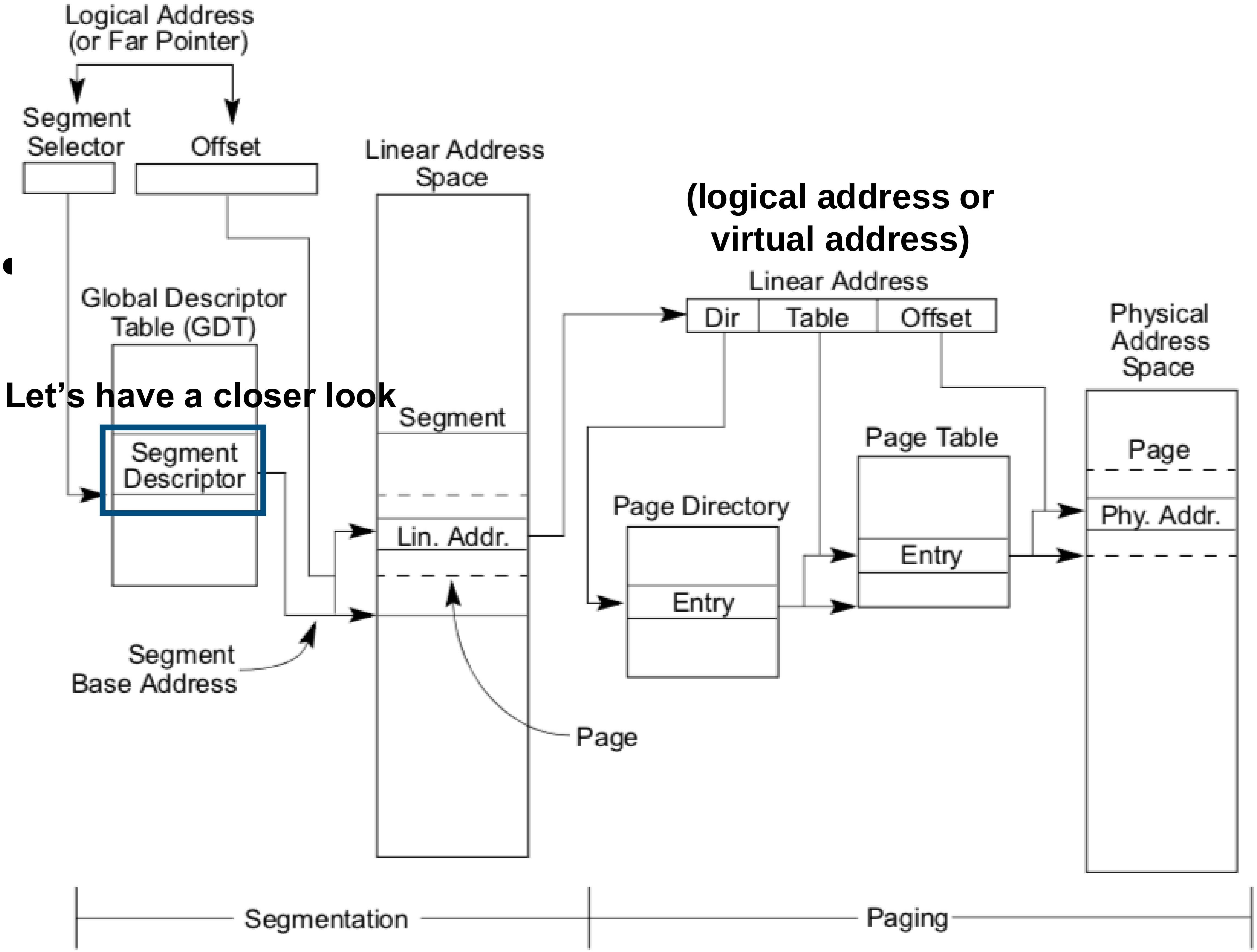
At runtime, Memory-Management Unit (MMU) relocates each load/store



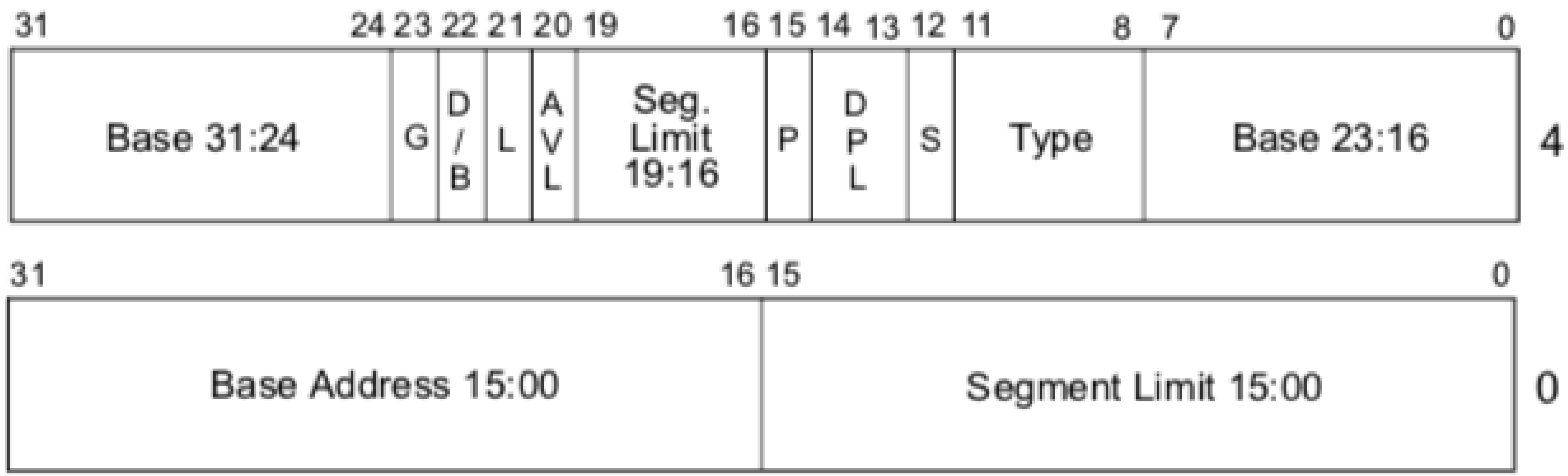


(logical address or virtual address)

Let’s have a closer look

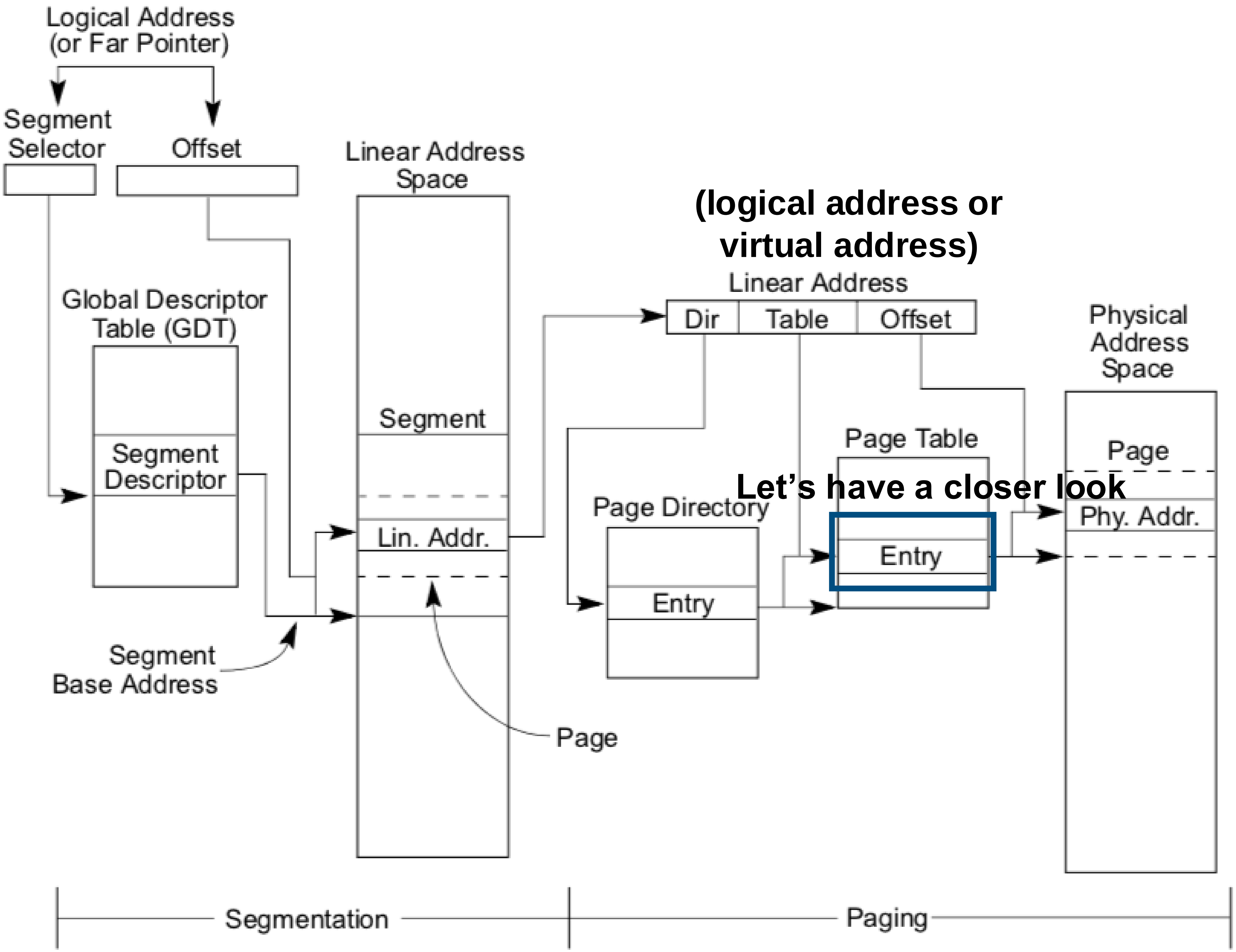


Segment descriptor (X86)

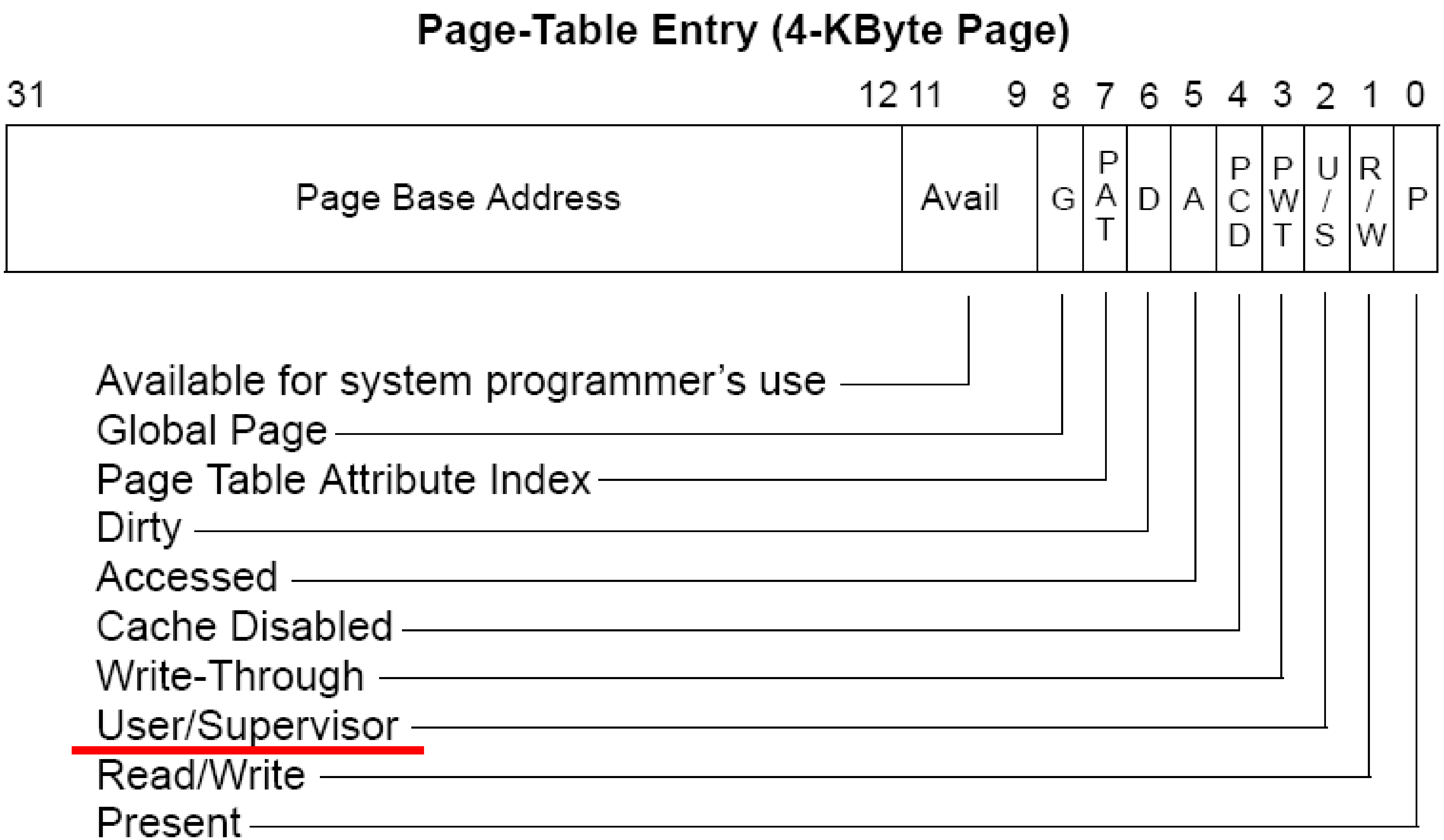


L — 64-bit code segment (IA-32e mode only)
AVL — Available for use by system software
BASE — Segment base address
D/B — Default operation size (0 = 16-bit segment; 1 = 32-bit segment)
DPL — Descriptor privilege level
G — Granularity
LIMIT — Segment Limit
P — Segment present
S — Descriptor type (0 = system; 1 = code or data)
TYPE — Segment type

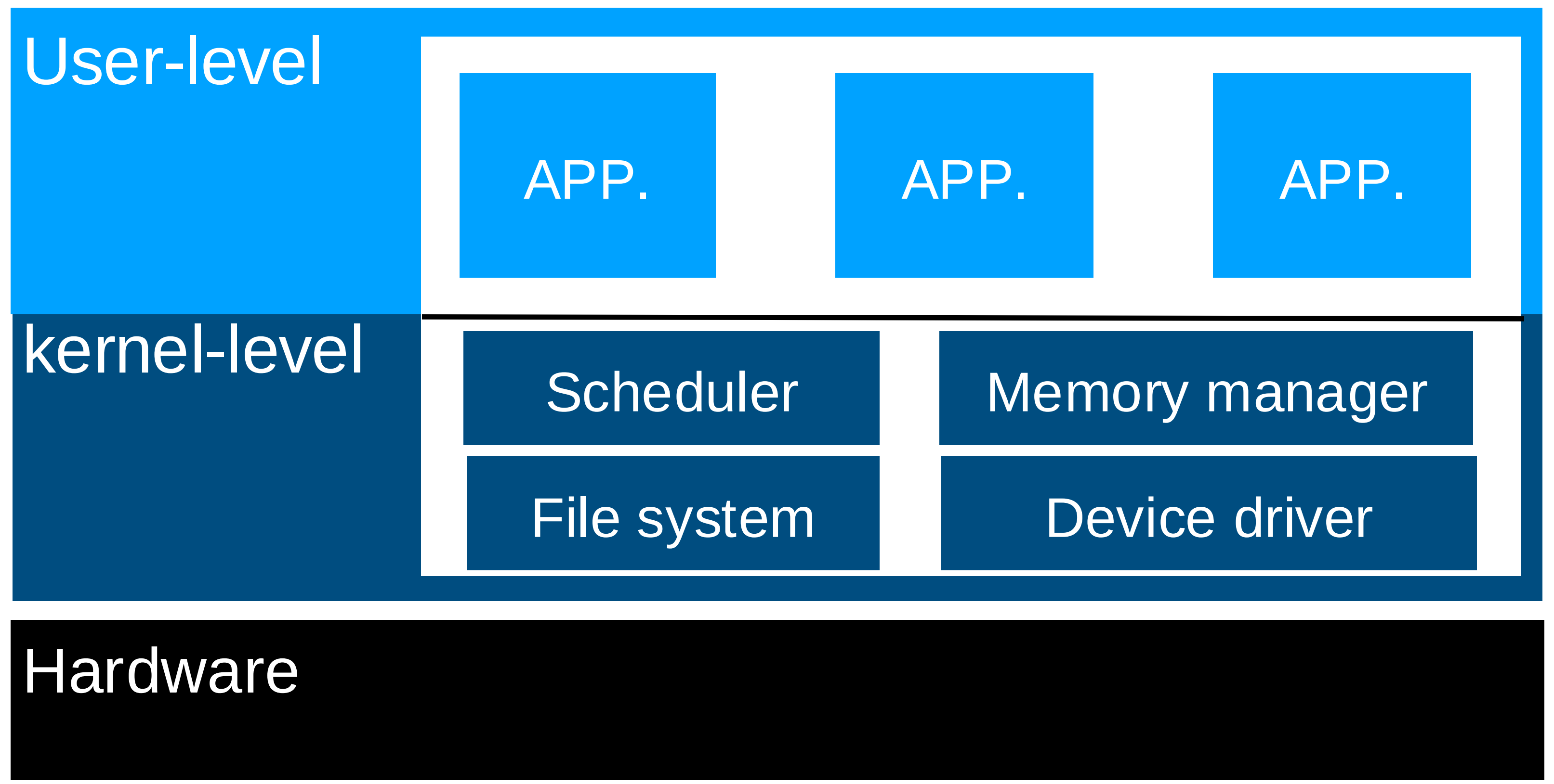
0: kernel (memory)
3: user (memory)


Logical Address
(or Far Pointer)
Segment
Selector
Offset
Linear Address
Space
(logical address or
virtual address)
Linear Address
Dir
Table
Offset
Physical
Address
Space
Global Descriptor
Table (GDT)
Segment
Segment
Descriptor
Page Table
Page
Let’s have a closer look
Page Directory
Phy. Addr.
Lin. Addr.
Entry
Entry
Segment
Base Address
Page
Segmentation
Paging

Page table entry



Dual-mode OS



- Do you see the problems?
- Buggy device drivers may shutdown entire system
- Performance impact to access fast devices

Key roles of OS

- Provide **abstraction** to use hardware
  - Provide APIs and semantics to applications
- **Protection & Isolation**
  - Contain malicious or buggy behaviors of applications
  - Protecting OS from malicious or buggy applications
  - Isolating one application from another
- **Sharing** resources
  - Multiplex hardware resources

Isolation by protection domain

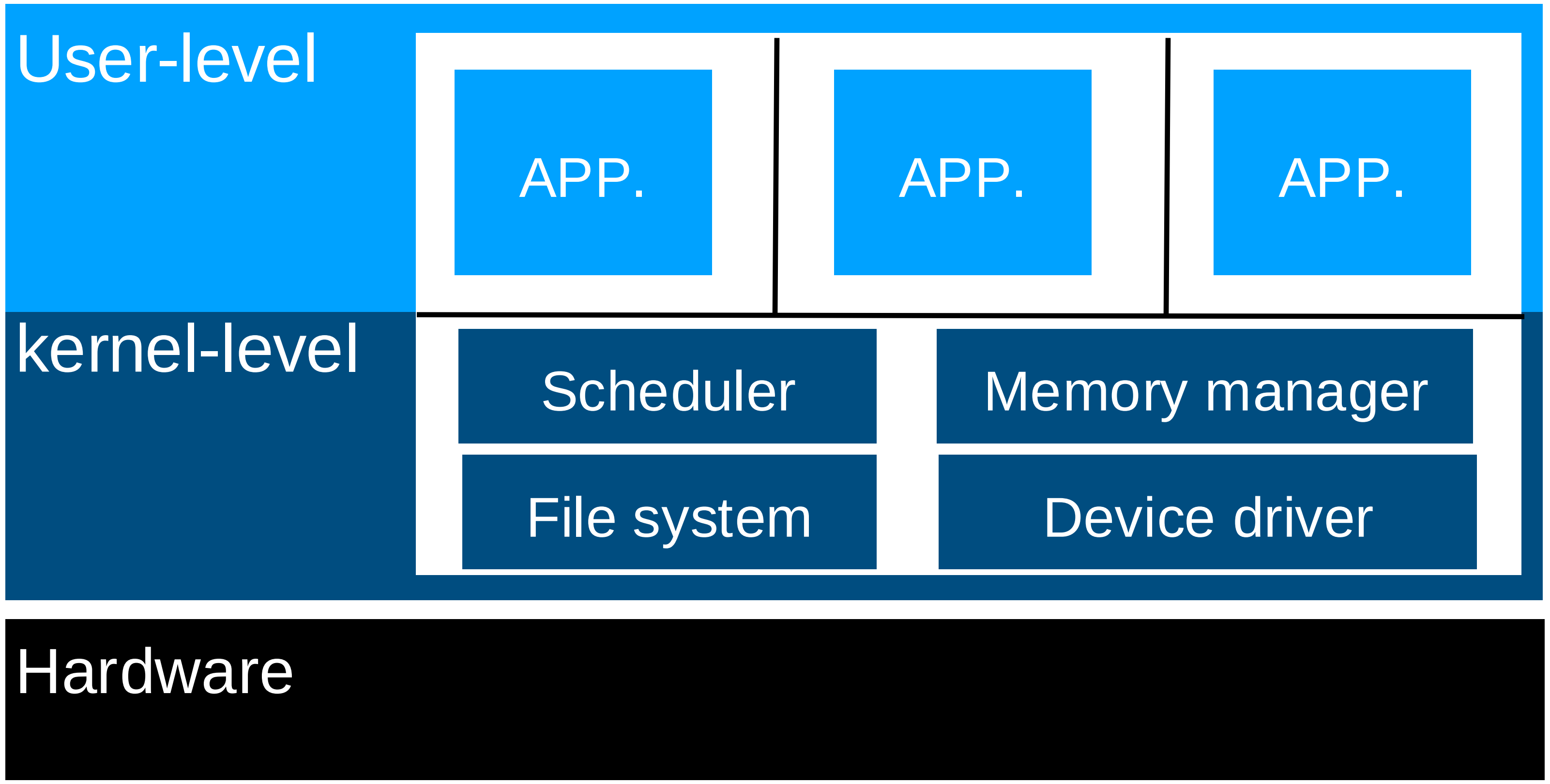
- The goal is isolating (protecting) application each other
- Hardware provides protection mechanisms
- OS Designers' first task is how to define protection unit and enforce the hardware mechanism

The first question

- Applying protection mechanism: raw hardware or abstraction?
  - File vs raw disk
  - Virtual address space vs physical memory
  - TCP connections vs ethernet packet

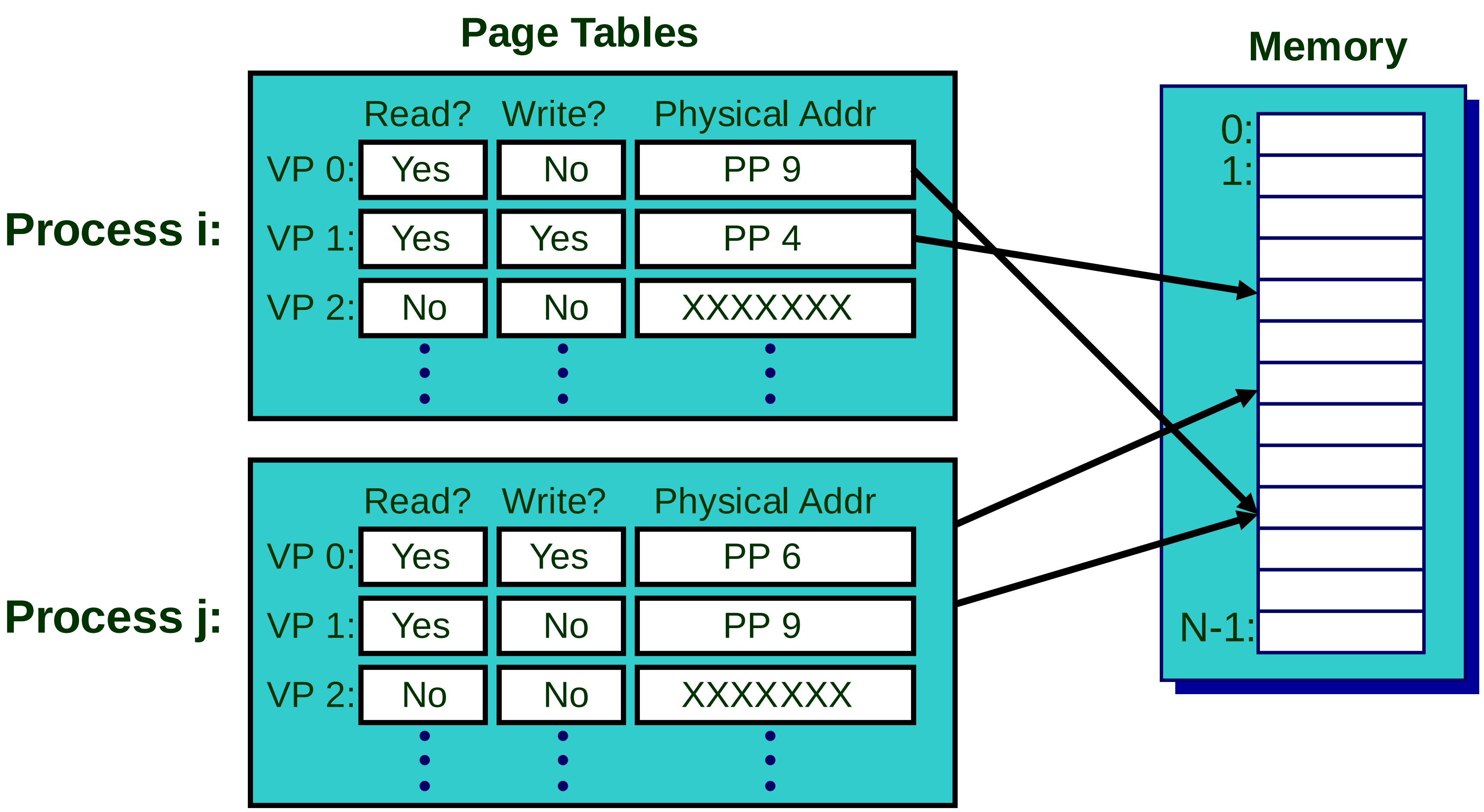
Isolation boundary

- What is a reasonable protection boundary in abstraction-level?
  - Process



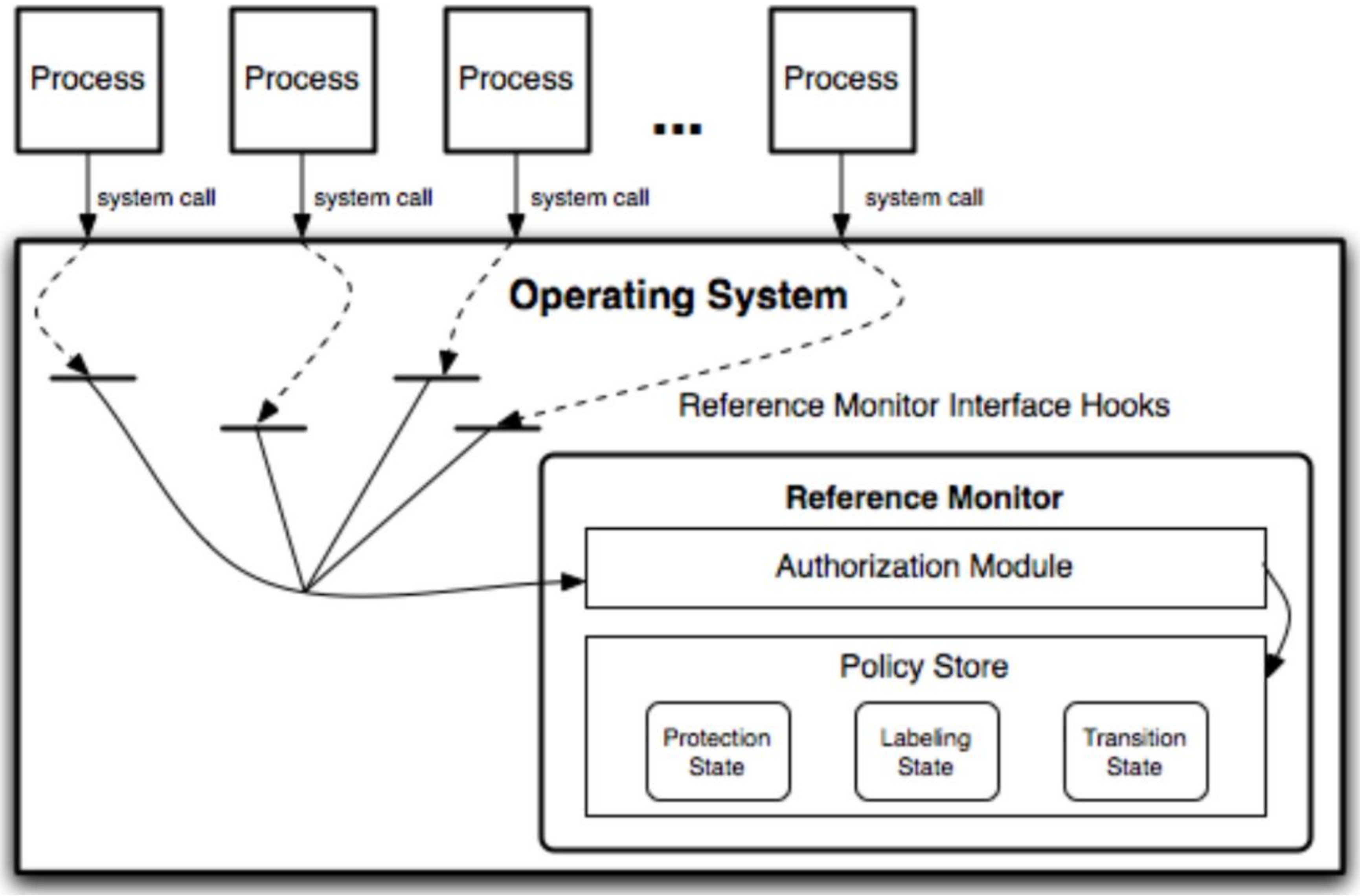
Isolation of memory

- Making virtual address private to each process
- Switching virtual address space when changing execution of process



Isolation of file

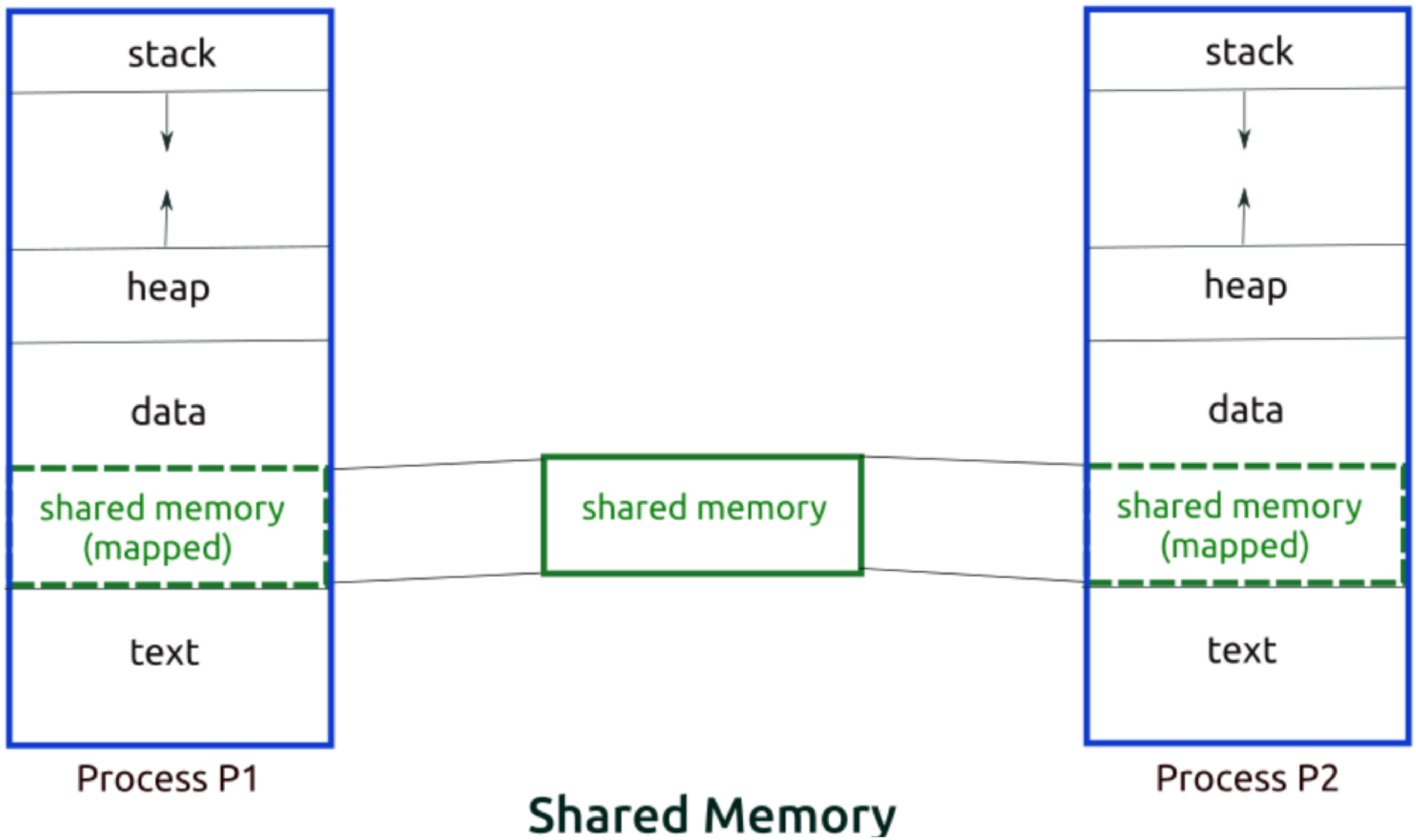
- File permission system to process (executed by a user)



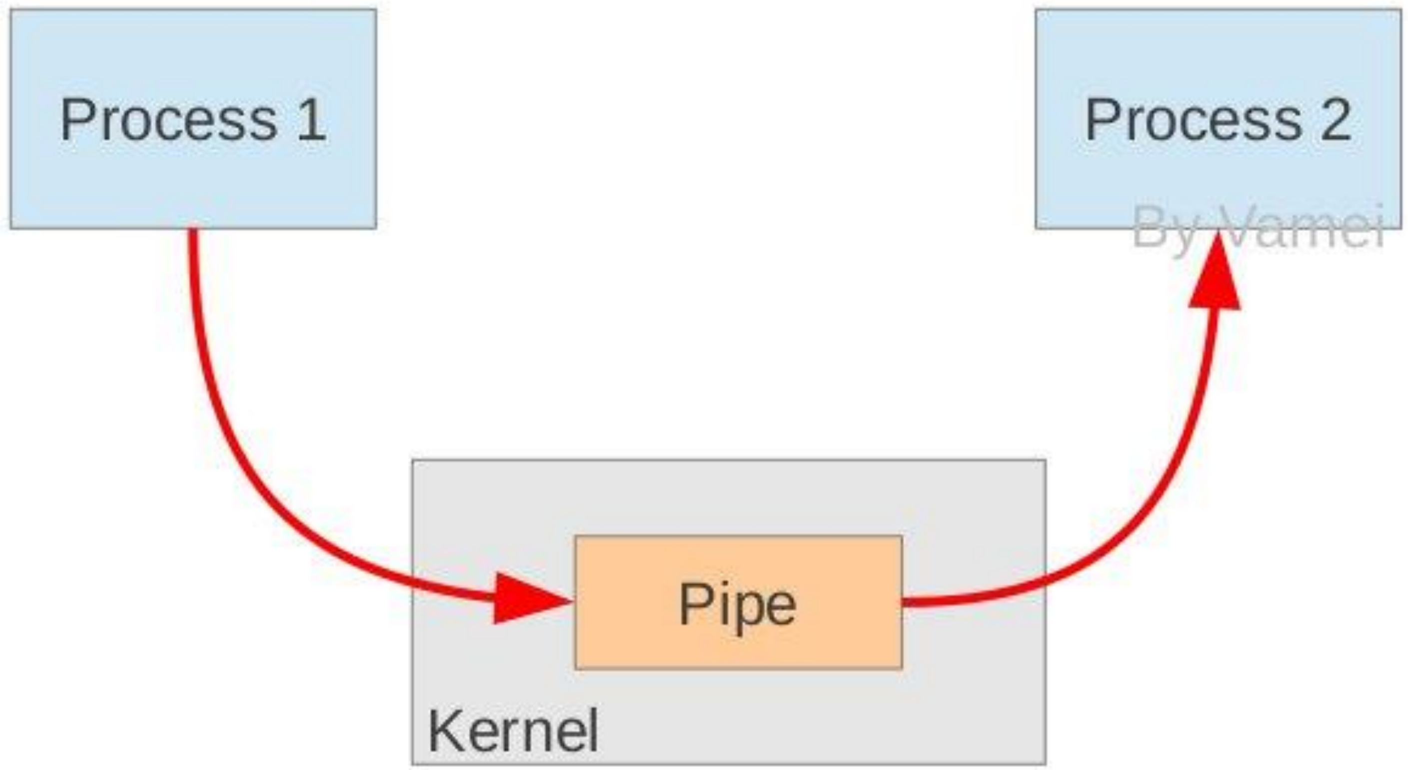
Will not cover (self study)

- Access control method
  - DAC and MAC
  - Capability-based access control
- Authentication of user and system (mutually distrust)
- Protected communication between the protection boundary
  - IPC

IPC: Shared memory



IPC: message passing

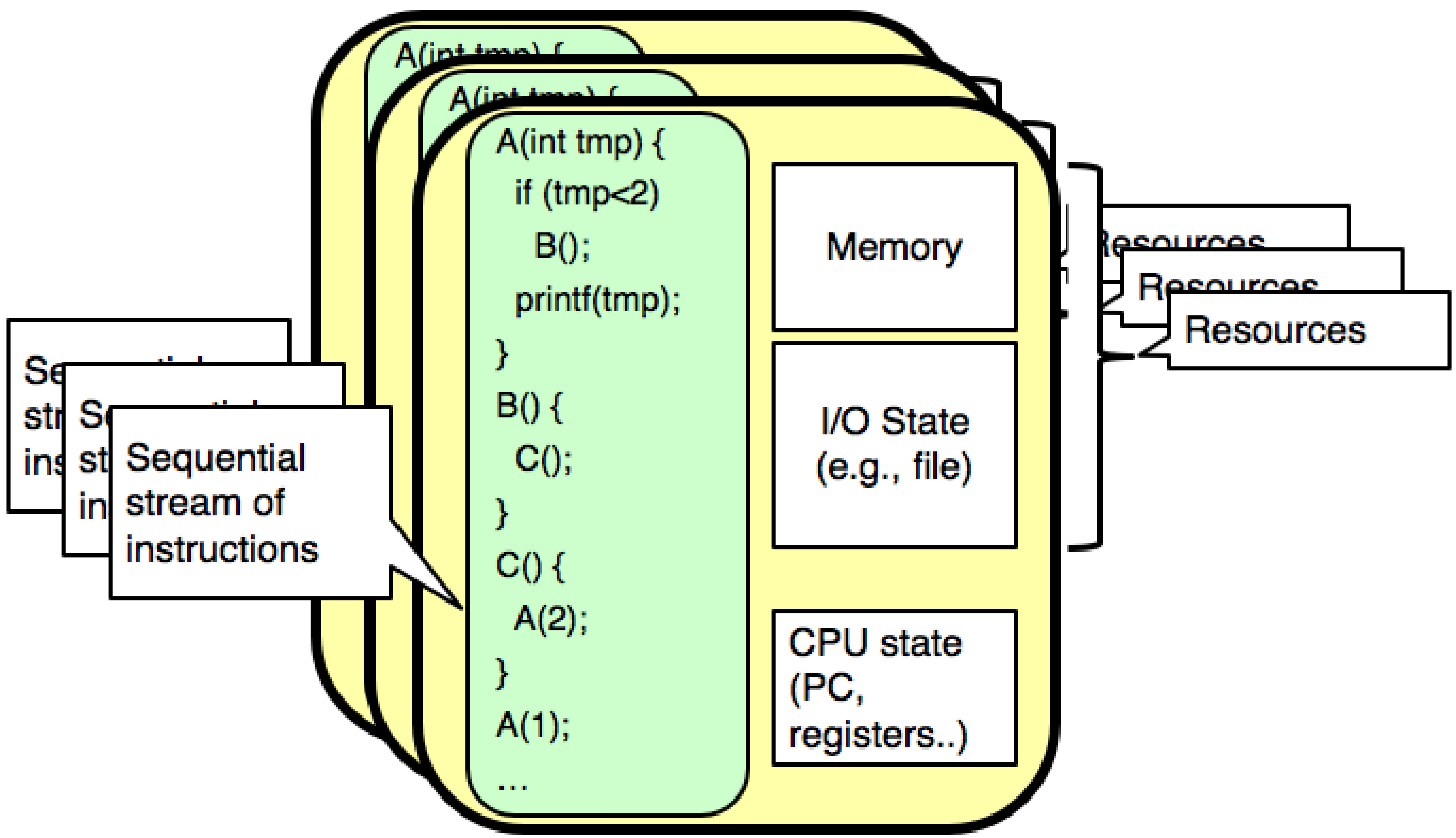


- Copying process 1's data to kernel buffer
- Copying kernel buffer to process 2's memory

Key roles of OS

- Provide **abstraction** to use hardware
  - Provide APIs and semantics to applications
- **Protection & Isolation**
  - Contain malicious or buggy behaviors of applications
  - Protecting OS from malicious or buggy applications
  - Isolating one application from another
- **Sharing** resources
  - Multiplex hardware resources

Now, we have an awesome abstraction



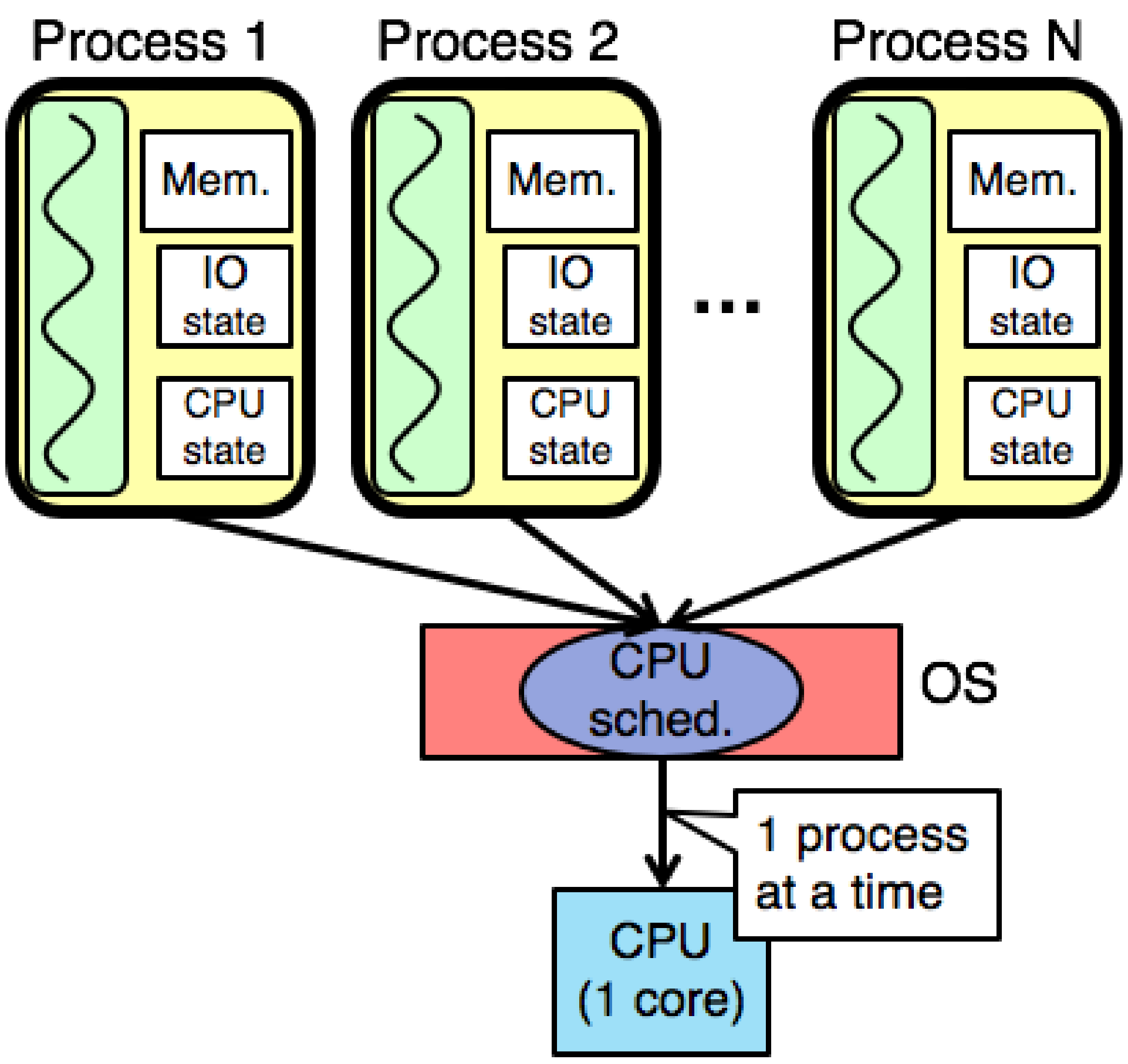
But my machine has only a single CPU and limited memory
So, processes must share the resources

Two types of sharing

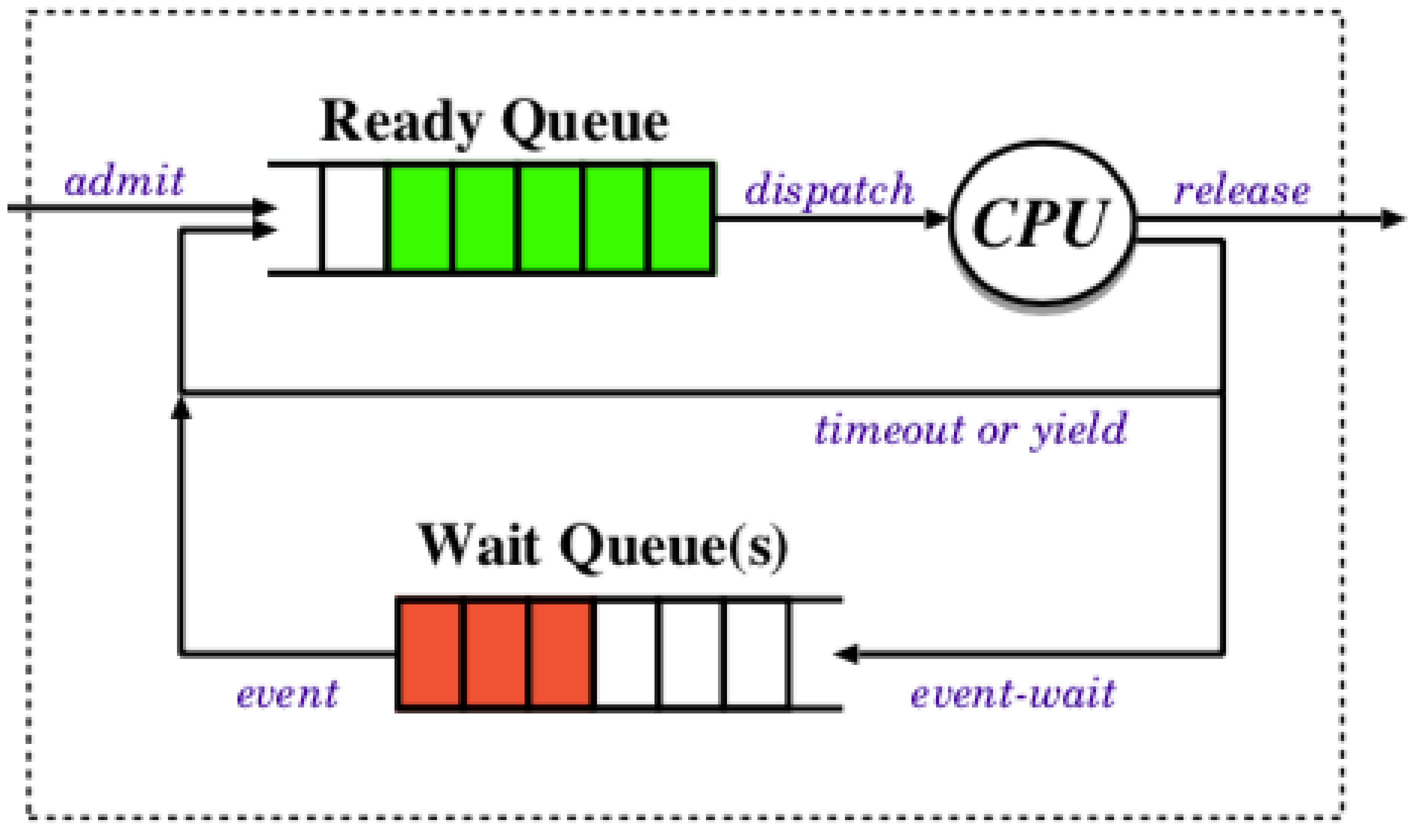
- **Time sharing**
    - CPU
    - Archived by scheduling

- **Space sharing**
    - Memory
    - Archived by virtual memory + space reclamation
        - e.g., page replacement (page eviction + swap)

Scheduling



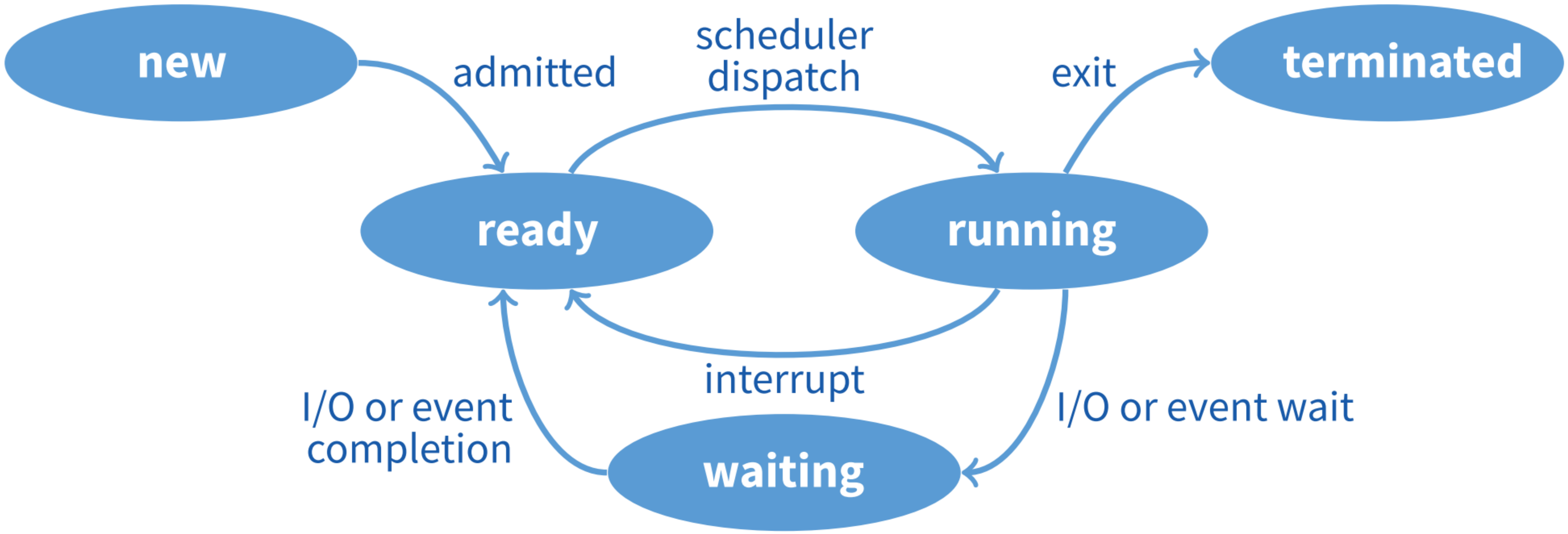
Closer look at scheduler



How to represent when processes is in ready or wait queue?

Design process state machine

When does OS invoke scheduler?



Preemptive scheduler:
1. Waiting → Ready
2. Running → Waiting
3. Running → Ready
4. New/waiting → Ready
5. Exit

Non-preemptive scheduler:
1. Running → Ready
2. Running → Waiting
3. Exit

Reminders

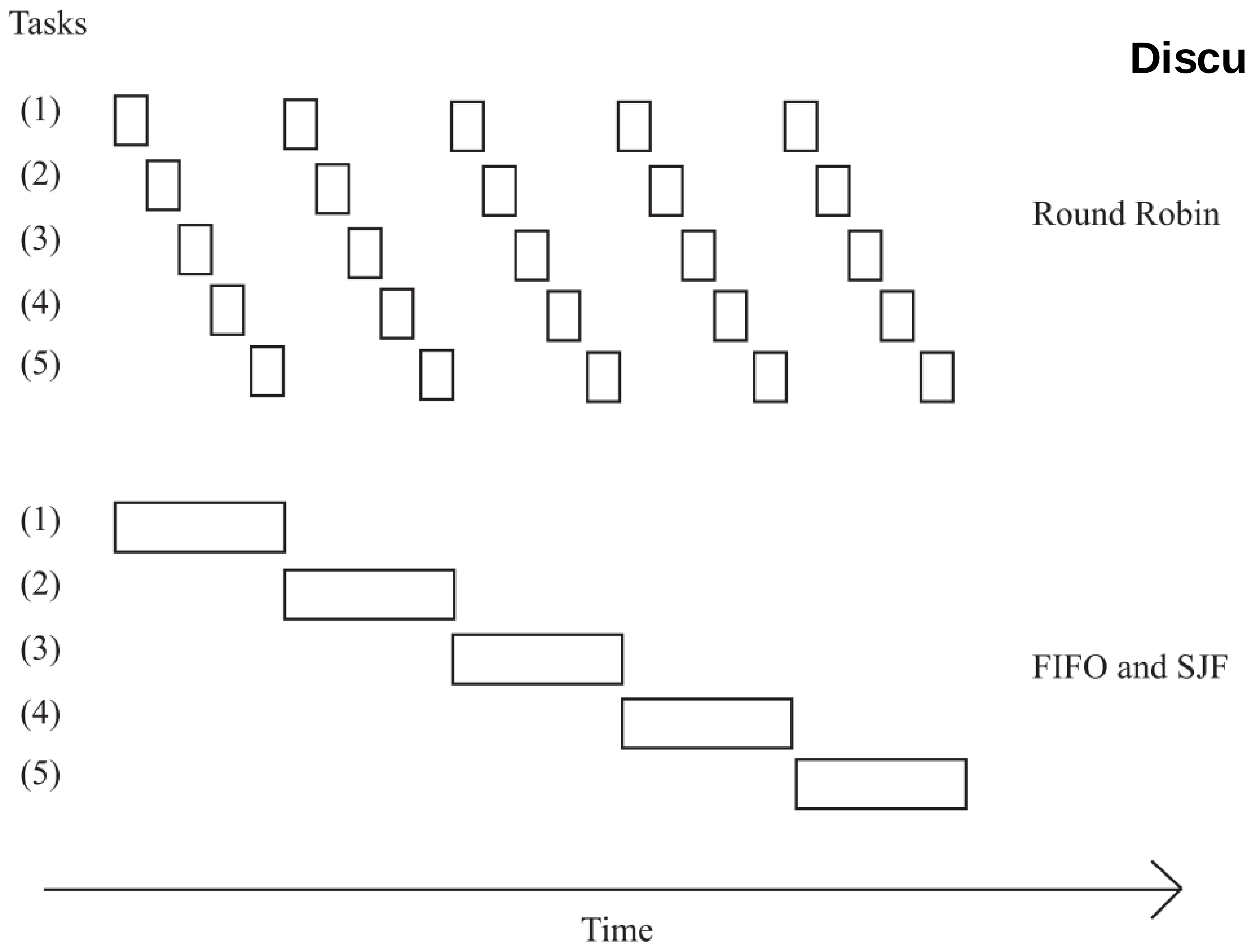
- FIFO
  - Pro:
  - Con:
- SJF
  - Pro:
  - Con:
- Round robin
  - Pro:
  - Con:

Reminders

- FIFO
  - Pro: Generally applicable
  - Con: Convoy effect (very high response time)
- SJF
  - Pro: Very good response time
  - Con: Starvation
- Round robin
  - Pro: No starvation, good response time
  - Con: Bad turnaround time

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?



Discuss response and turnaround time

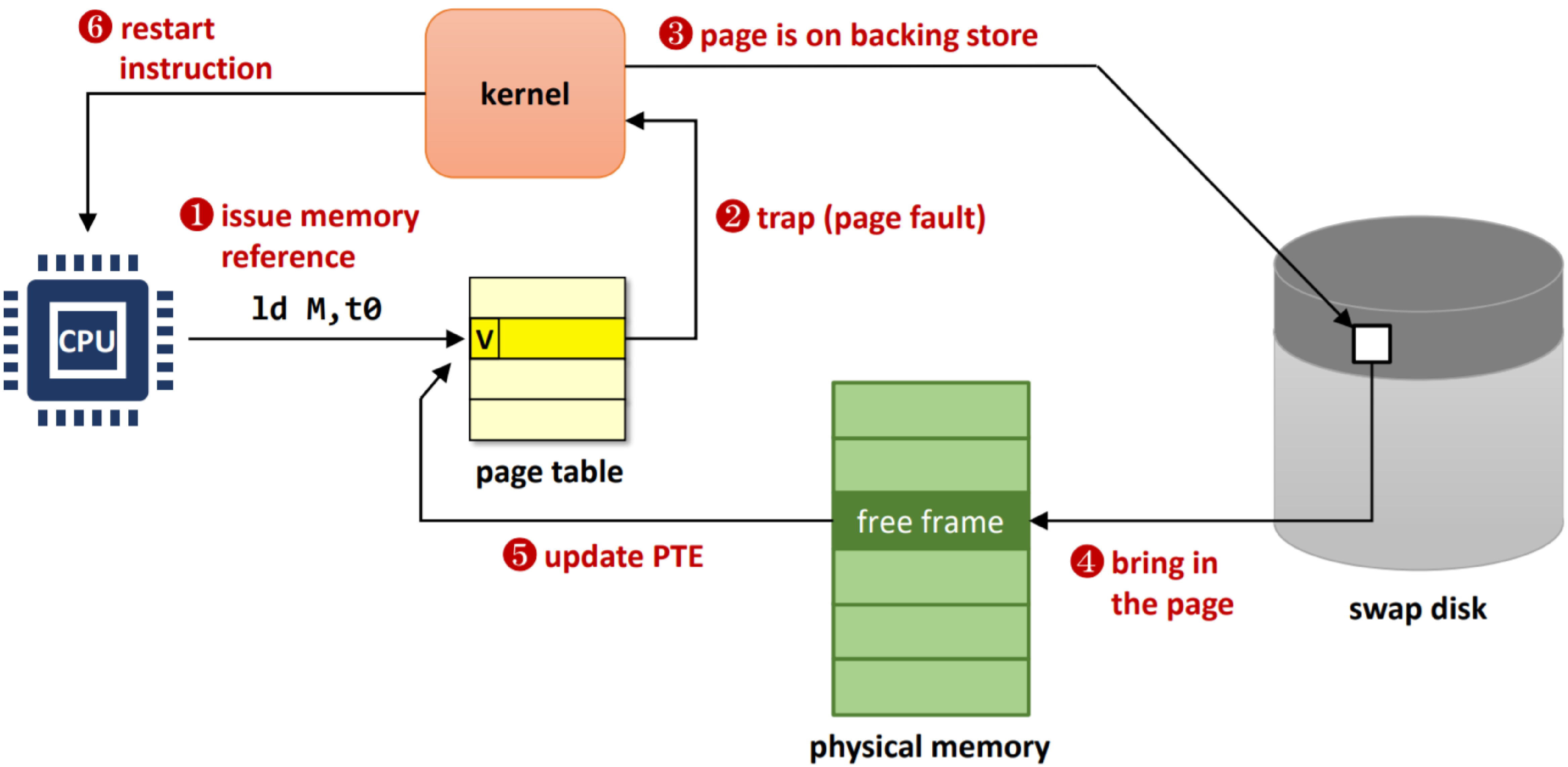
Good response time
Terrible turnaround time

Modest response time
Good turnaround time

Will not cover (self study)

- Time sharing
  - Scheduling policies
  - Multi-CPU scheduling

Paging review



Page replacement policy

- When a page fault occurs, the OS loads the faulted page from disk into a page frame of physical memory
- At some point, the process used all of the page frames it is allowed to use
  - This is likely (much) less than all of available memory
- When this happens, *the OS must replace a page for each page faulted in*
  - It must evict a page (called victim) to free up a page frame
- The **page replacement algorithm** determines how this is done

Policy goal: reduce cache misses

Improve expected case performance

Will not cover (self study)

- Space sharing
  - Page replacement policies
    - Optimal, LRU, Clock
    - Belady's anomaly

How to design OS

- Monolithic Kernel
  - All kernel components runs in the same kernel address space
- Microkernel
  - Moving some OS subsystems to user-level
- Exokernel
  - OS abstraction is bummer! No abstraction! (Whoa…)
  - Application must directly access hardware

Advanced topics of OS covered by graduate OS course