Computer Systems Design

Lesson 8

Operating systems.

Memory virtualization.

Alexander Antonov, Assoc. Prof., ITMO University

Hangzhou, 2025

ITMO

Outline the lesson

- Basic construction principles
- OS/apps timeline
- Privilege levels
- Kernel and process context
- Address translation
- System call interface

ITMO

Operating systems (OS)

Operating system: system software providing the following capabilities:

- Isolation of application resources from each other
- Protection of system resources from applications
  system code and structures, critical data, I/O
- Abstraction from selected details of HW organization
  access to platform capabilities through standardized API
- Resource management
  CPU time, main memory, disk space, peripherals

Includes: ***kernel***, device drivers, interface subsystem, additional services
Manages execution of multiple ***processes***

Basic terms

Process – instance of executing program

Thread – instruction sequence executed on processor core

Single process might include several threads sharing common resources

Monolithic kernel – kernel executing most its functions on its own (in kernel itself)

Micro-kernel – kernel implementing only basic functions and offloading its most services as individual, special-purpose processes

Micro-kernels are typically more secure, but suffer from performance penalties (more switches between kernel and processes needed)

Basic OS construction principles

Hardware should provide ***at least 2 modes of execution***:

- Privileged (“supervisor”) mode: full access to the whole memory and I/O
  OS kernel works in this mode

- Unprivileged (“user”) mode: no access to anything rather than process memory
  application processes work in this mode

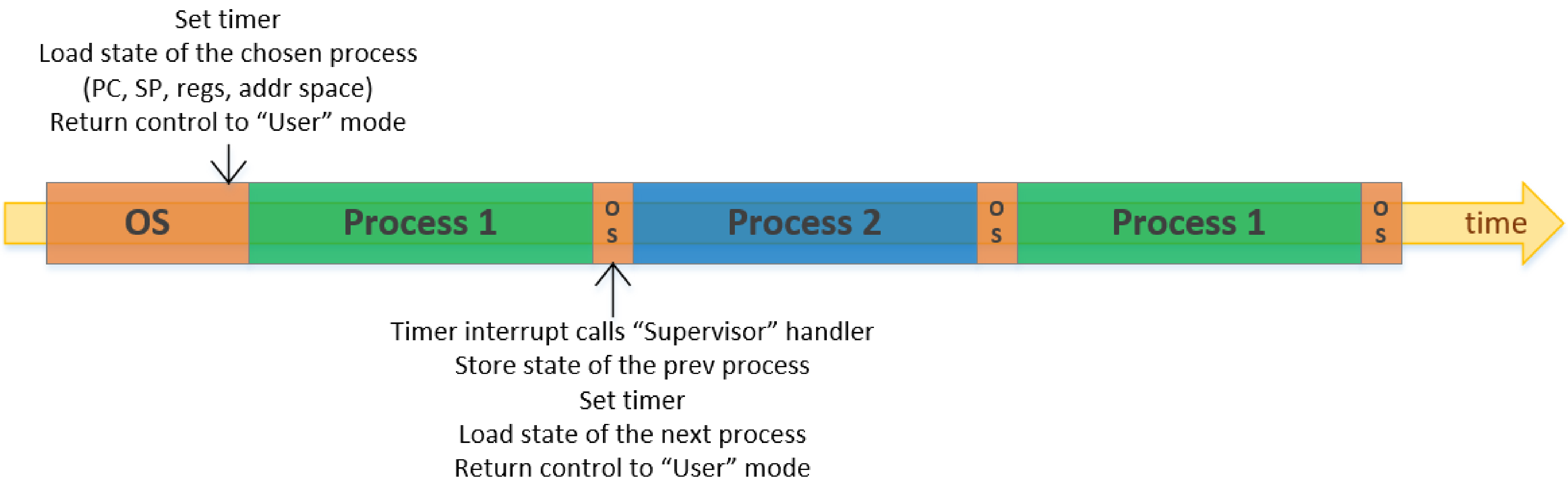
OS “gives” the apps some time to work, than returns control (on ***system timer***, or if ***system call*** requested).

Memory Management Unit decouples address spaces (setup by OS, described later)

To do something “interesting” (access something else: storage, graphics, network, etc.), processes should ***“ask”*** OS to access it (fire system call).

OS decides how to process the system call.

OS/apps timeline



ITMO

System call examples

- Halting process
- Creation of new processes
- Memory (de)allocation
- Interprocess communication (IPC) request
- Peripheral access (storage, graphics, network)
- Waiting for events
- Getting information about the system

RISC-V privilege levels

RISC-V ISA defines the following privilege levels:

- **Machine**
  - basic mode, must be present in the system
  - the highest level of trust
- **Supervisor**
  - used by the operating system
  - does not have access to the **Machine** level
- **User**
  - used by applications
  - the lowest level of trust, does not have access to the **Machine** and **Supervisor** levels

Number of levels	Supported Modes	Intended Usage
1	M	Simple embedded systems
2	M+U	Secure embedded systems
3	M+S+U	Systems running Unix-like operating systems

Kernel/process context

When kernel and processes switch, the ***context should be changed***

Context elements:

- Registers (PC, general purpose registers)
- Memory mappings

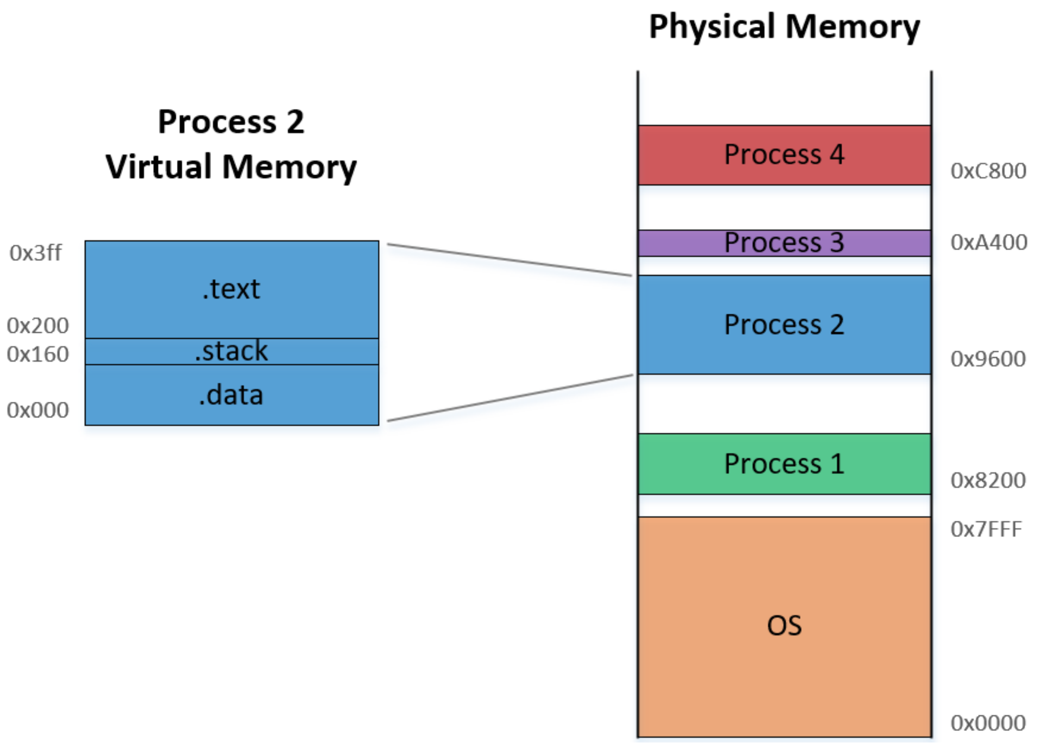
  Virtual address: address defined in program

  Physical address: translated address in physical memory

Address translation is done continuously during process execution by special hardware block: ***Memory Management Unit (MMU)***

- Other resources

  shared data, opened files, I/O, etc.



Page-based address translation

Historically, segmented (base-and-bound) translation was used. Actual implementation: ***paging***
Memory space of all processes is split into ***pages***: equal blocks of memory

usually 4KiB - 64KiB

- “Hot” pages reside in main memory, currently unused: in disk

  disk “expands” main memory

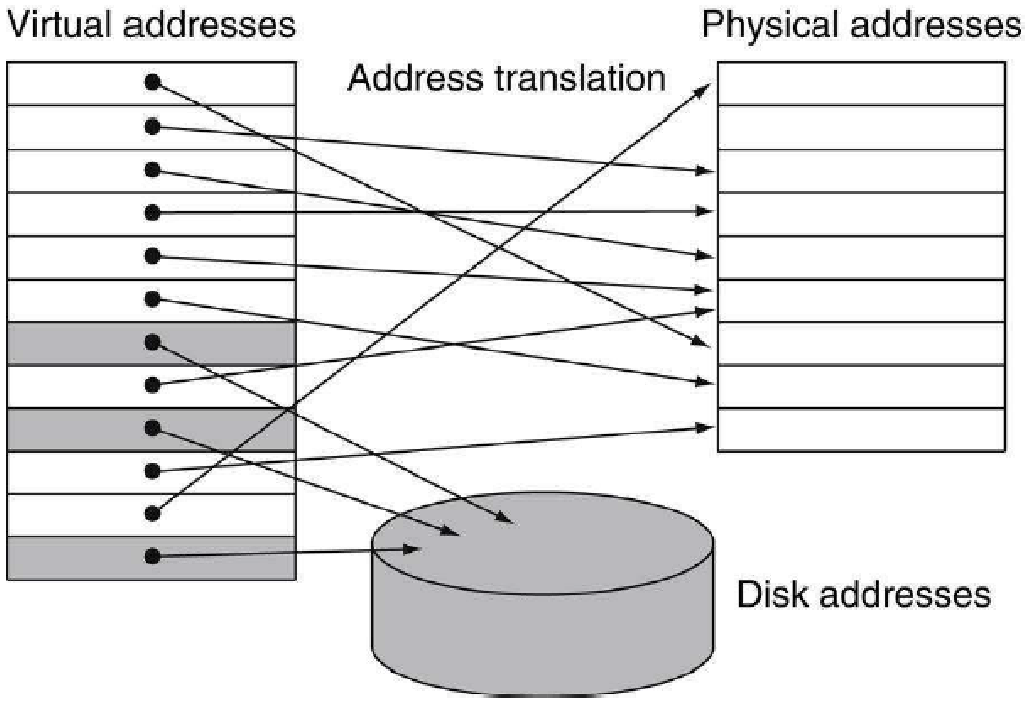
- MMU automatically translated addresses while process works

- If process accesses the page not currently in memory

  1) ***page fault*** is fired: special exception returning control to OS

  2) OS loads the page from disk to main memory

  3) OS returns control to the process



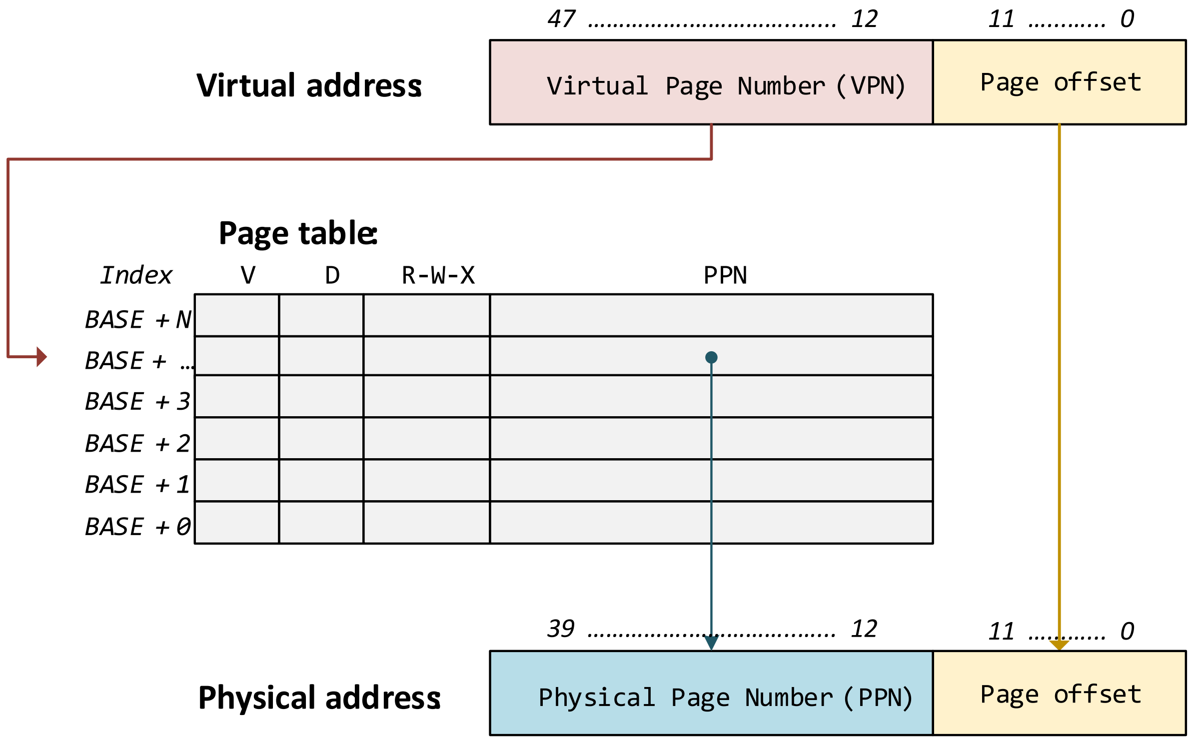
Page table

Page table: data structure describing translations for various pages

created by the OS for executable processes
stored in main memory (or disk)
processed by MMU

Contents:

- **validity bit** – a presence of a page in RAM (otherwise the page is on disk)
- physical page numbers (**PPN**)
- modification bit (**dirty**)
- permission bits (**r-w-x**)



Page address translation

Problem: too many pages

e.g. 1M 4-KiB pages for 32-bit address space – for each process!

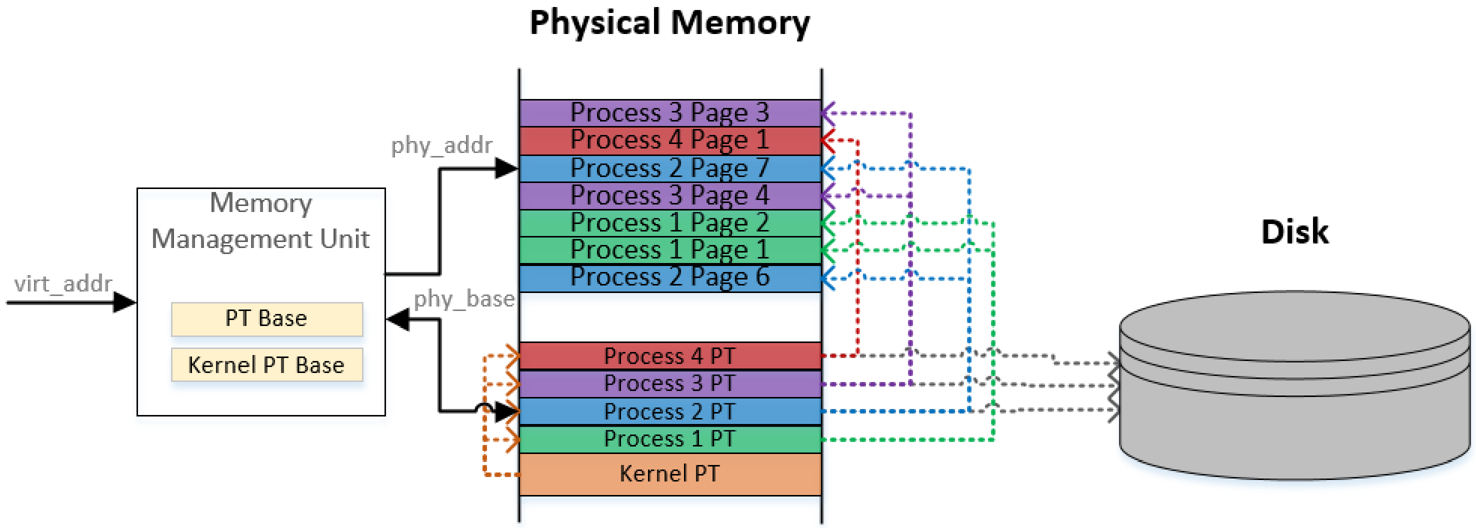
Solution: additional level of indirection:

- **Kernel PT** – table with PTs for all processes

  Kernel PT base written to MMU, indexed by process ID

- **Process PT** – PT for individual process

  PT base fetched from Kernel PT, indexed by VPN

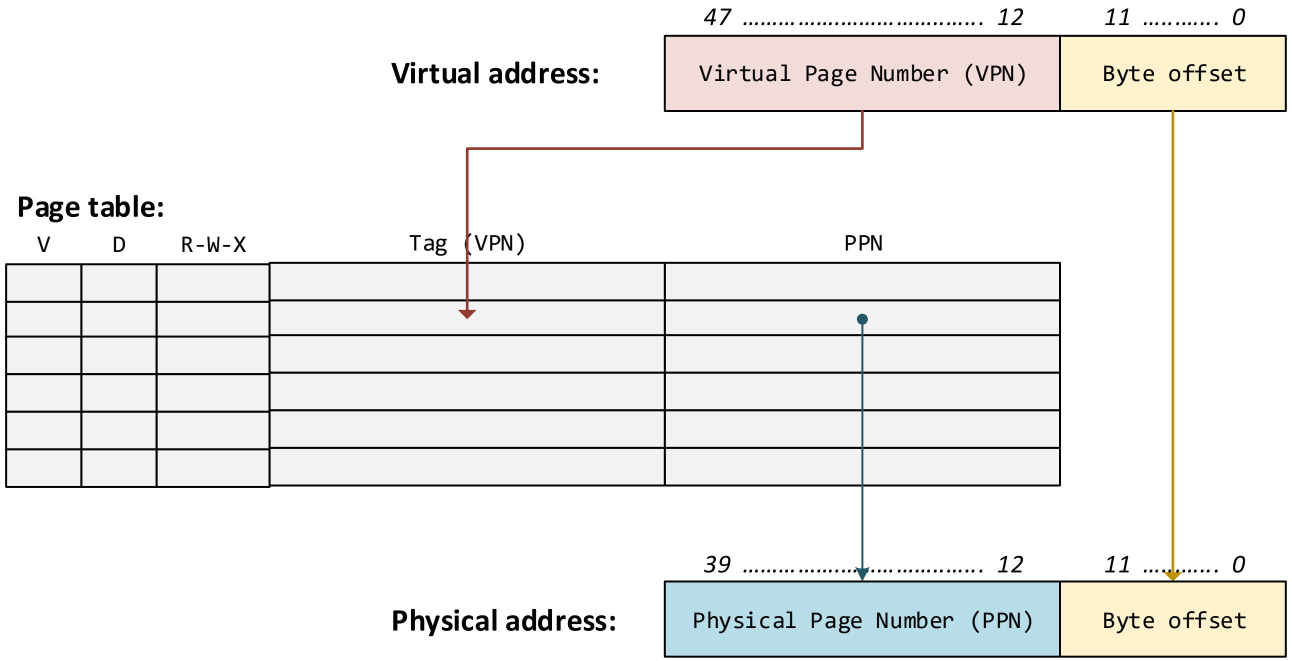


Translation lookaside buffer (TLB)

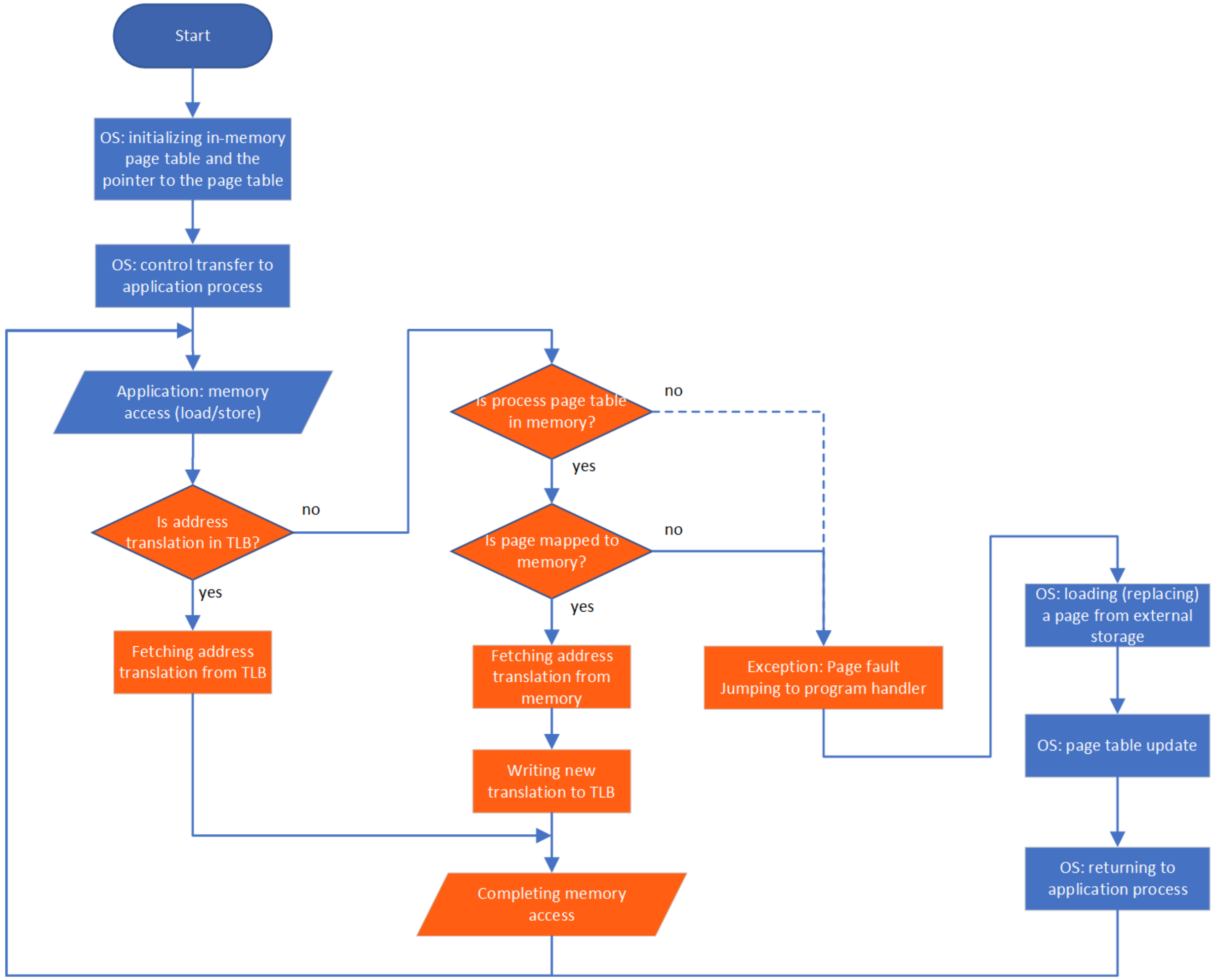
Translation lookaside buffer — specialized hardware cache inside processor containing page translations.

*stores **subset** of all page translations ("hot" ones)*

*actual presence of page translation should be **checked** (by the VPN)*



Typical distribution of functionality between HW and SW

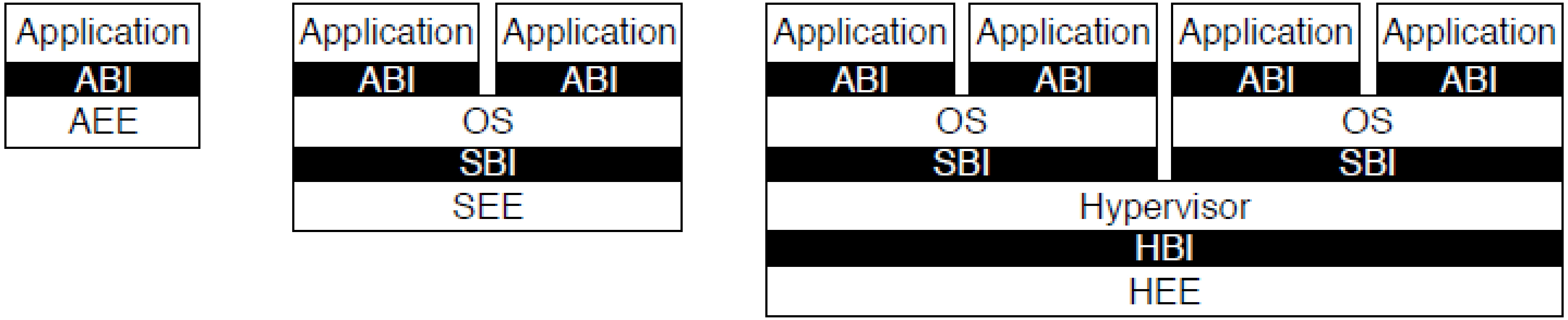


System call interface

OS and apps are typically compiled ***separately*** – interfaces have to be defined in ***binary form***

E.g. for RISC-V:

- Application Binary Interface (**ABI**):
  process – OS (or Application Execution Environment) interface
- Supervisor Binary Interface (**SBI**):
  OS – hypervisor (or Supervisor Execution Environment) interface
- Hypervisor Binary Interface (**HBI**):
  hypervisor – hardware (or Hypervisor Execution Environment) interface



Example: Linux system calls

Linux uses **a0-a6** registers for arguments, and **a7** register for system call number

```
#define SBI_CALL(which, arg0, arg1, arg2) ({          \
   register uintptr_t a0 asm ("a0") = (uintptr_t)(arg0);   \
   register uintptr_t a1 asm ("a1") = (uintptr_t)(arg1);   \
   register uintptr_t a2 asm ("a2") = (uintptr_t)(arg2);   \
   register uintptr_t a7 asm ("a7") = (uintptr_t)(which);  \
   asm volatile ("ecall"                   \
          : "+r" (a0)              \
          : "r" (a1), "r" (a2), "r" (a7)       \
          : "memory");             \
   a0;                       \
})
```

```
li a0, <arg 0>
li a1, <arg 1>
li a2, <arg 2>
li a7, <which>
ecall
```

Summary: typical reactions to memory-related events

Event	Registers
Processor cache miss	Handled by hardware. Data will be fetched from lower-level cache or main memory automatically (introducing delay, probably hitting performance)
TLB miss	Usually handled by hardware. Address translation will be fetched from lower-level TLB or main memory automatically (introducing delay, probably hitting performance)
Requested data page not in memory	Exception. Application is paused, control is transferred to OS. OS loads page with requested data from external memory to main memory, then resumes application.
PT not in memory	Exception. Application is paused, control is transferred to OS. OS loads PT from external memory to main memory, then attempts to resume application.
Read/write to incorrect address (without permission or out of allocated memory space by OS)	Exception (segmentation fault). Application stopped immediately, control is transferred to OS.
Read/write to incorrect address (within allocated memory space by OS, used afterwards)	Memory damage. Application might work incorrectly when accessing damaged data or crash.
Read/write to incorrect address (within allocated memory space by OS, never used afterwards)	Hidden memory damage. Application will work correctly but can crash after rebuilding and remapping data to memory.

Thank you for the lesson!

Alexander Antonov, Assoc. Prof., antonov@itmo.ru

Hangzhou, 2025