CS614: Linux Kernel Programming

Logistics, Introduction

Debadatta Mishra, CSE, IIT Kanpur

Course and instructors

$whereis cs614

Mon, Wed 2PM - 3.15PM RM-101
https://www.cse.iitk.ac.in/users/deba/cs614/

Piazza link: https://piazza.com/iitk.ac.in/secondsemester2023/cs614
Canvas: https://canvas.cse.iitk.ac.in/login

$whereis deba

KD 212, deba@cse.iitk.ac.in , Meeting hours: 3PM - 5PM Thursday

$ ls TAs

Arun KP (kparun@cse.iitk.ac.in), Rohit Singh (rohit@cse.iitk.ac.in)

Course policy

Add/Drop

Course registration and drop:- Ideally before 16th Jan 2023

Class guidelines

Keep your mobile phones switched off / silent

Ask questions and interact (both in class and in Piazza)

Be on time!

Evaluation

1. Quizzes (10%)
2. Assignments (40%)
3. Mid-semester (15%)
4. End-semester (35%)

Evaluation

1. Quizzes (10%)
2. Assignments (40%)
3. Mid-semester (15%)
4. End-semester (35%)

References

Operating Systems: Three Easy Pieces. Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau.
Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati.
Linux Kernel Development, 3rd Edition, Robert Love.
Linux Device Drivers, 3rd Edition, By Jonathan Corbet, Greg Kroah-Hartman, Alessandro Rubini.
Linux kernel documentation, Research papers

Evaluation

1. Quizzes (10%)
2. Assignments (40%)
3. Mid-semester (15%)
4. End-semester (35%)

“Take pride in honest hard work ”

“Cheating implies accepting defeat”

“If you are here to learn, never defeat the purpose by cheating”

https://www.cse.iitk.ac.in/pages/AntiCheatingPolicy.html

Homework -1

- Personal laptops with decent backup desirable
- **HW1: Setup a Virtual machine for the course (Due before next lecture)**
  - Create a Linux VM (Ubuntu Linux recommended) (KVM is prefered)
  - Download the Linux kernel version - linux-6.1.4
  - Compile and boot the latest kernel

OS refresher: True/False

1. Superuser (e.g., root user in UNIX) in a multi-tasking OS can execute all instructions provided by the hardware instruction set architecture (ISA).
2. Every process in a computer system is guaranteed to be in running state at least once during its life time.
3. A critical section consisting of a single instruction may require mutual exclusion.
4. A user process interrupted by a device interrupt is always scheduled immediately after the interrupt is serviced.
5. A page fault can be handled without changing any page table entry.

OS refresher: Quiz

In a uniprocessor system, in which of the following case(s), a process executing in user mode can cause an entry into the OS?

A. accessing a general purpose register like RAX

B. executing a JMP (jump) instruction

C. decrementing an unsigned integer value stored in a register beyond zero

D. executing a printf( ) statement

E. returning from a function

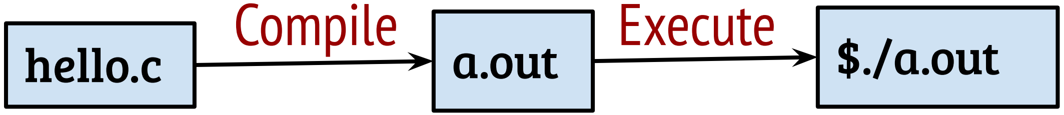
Revisiting Hello World!

hello.c

```
int main(void)
{
   printf(“Hello world”);
   while(1);
}
```

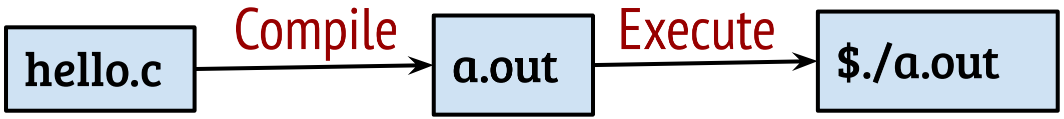
- What will be the execution behavior and why?
- Alternate ways to print “hello world” on screen?

Program execution



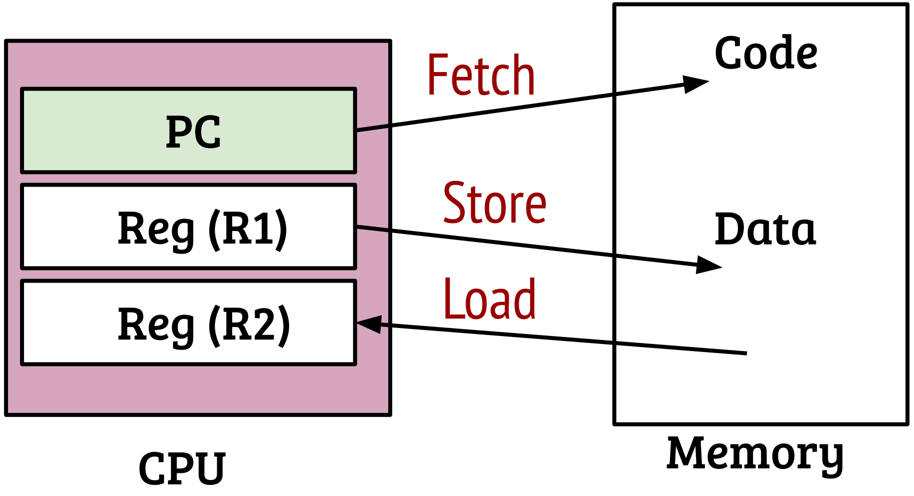
You said only CPU can execute!

Inside program execution



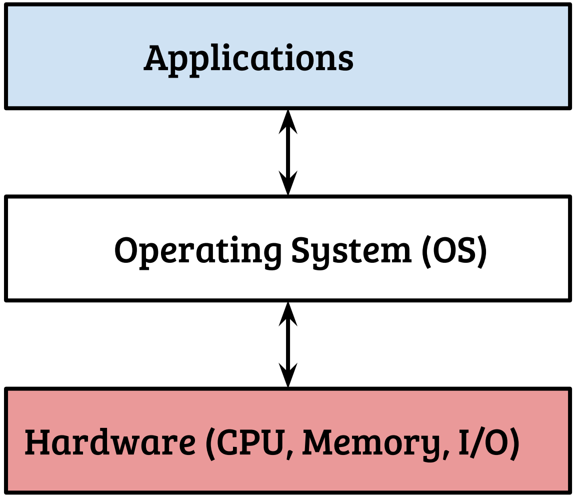
You said only CPU can execute!

CPU execution (from hardware perspective)



- Loads instruction pointed to by PC
- Decode instruction
- Load operand into registers
- Execute instruction (ALU)
- Store results

Role of the OS



- OS bridges the *semantic gap* between the notions of application execution and real execution
  - OS loads an executable from disk to memory, allocates/frees memory dynamically
  - OS initializes the CPU state i.e., the PC and other registers
  - OS provides interfaces to access I/O devices
- OS facilitates hardware resource sharing and management (How?)

Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU

Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space

Virtual view of resources

- Process
    - Each running process thinks that it owns the CPU
- Address space
    - Each process feels like it has a huge address space
- File system tree
    - The user feels like operating on the files directly

Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?

Virtual view of resources

- Process
  - Each running process thinks that it owns the CPU
- Address space
  - Each process feels like it has a huge address space
- File system tree
  - The user feels like operating on the files directly
- What are the OS responsibilities in providing the above virtual notions?
  - The OS performs multiplexing of physical resources efficiently
  - Maintains mapping of virtual view to physical resource

Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
    - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?

Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Virtualization: Efficiency/performance

- Resource virtualization should not add *excessive* overheads
- Efficient when programs use the resources directly, infrequent OS mediation
  - Example: when a process is scheduled on CPU, it should execute without OS intervention
- What is the catch?
  - Loss of control e.g., process running an infinite loop on a CPU
  - Isolation issues e.g., process accessing/changing OS data structures

Conclusion: Some limits to direct access must be enforced.

Limited direct execution

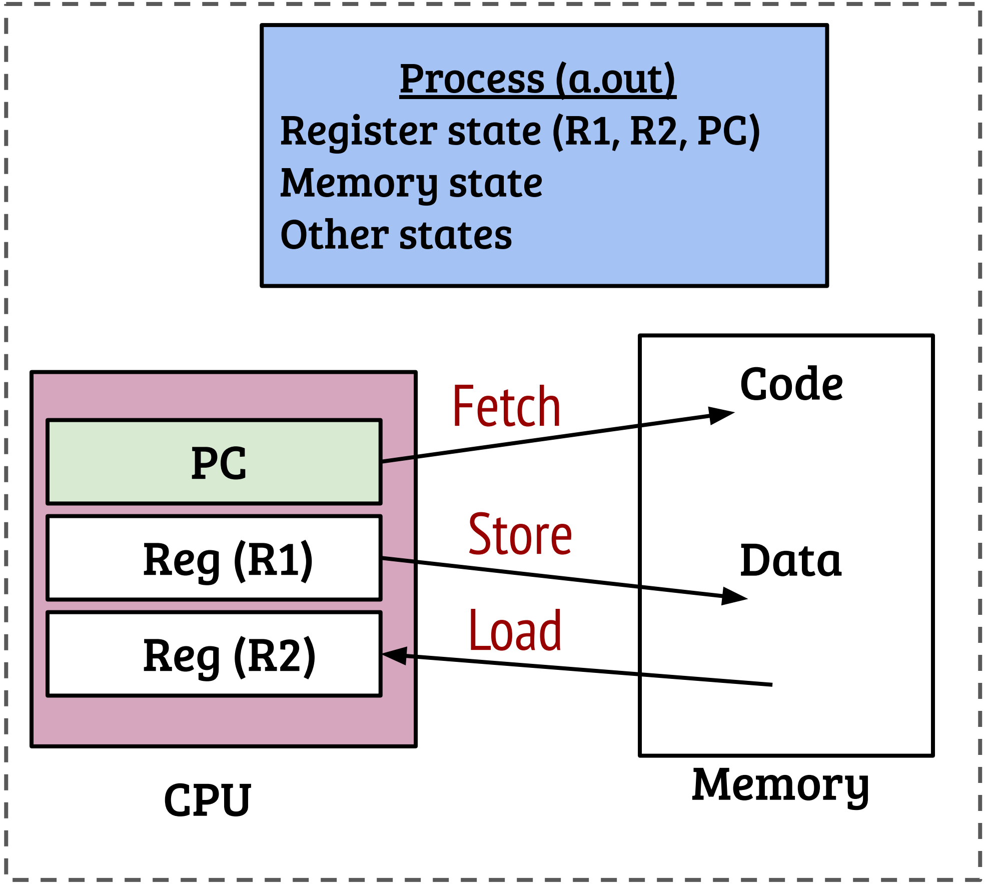
- Can the OS enforce limits to an executing process by itself?

A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

OS

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



Process (a.out)
Register state (R1, R2, PC)
Memory state
Other states

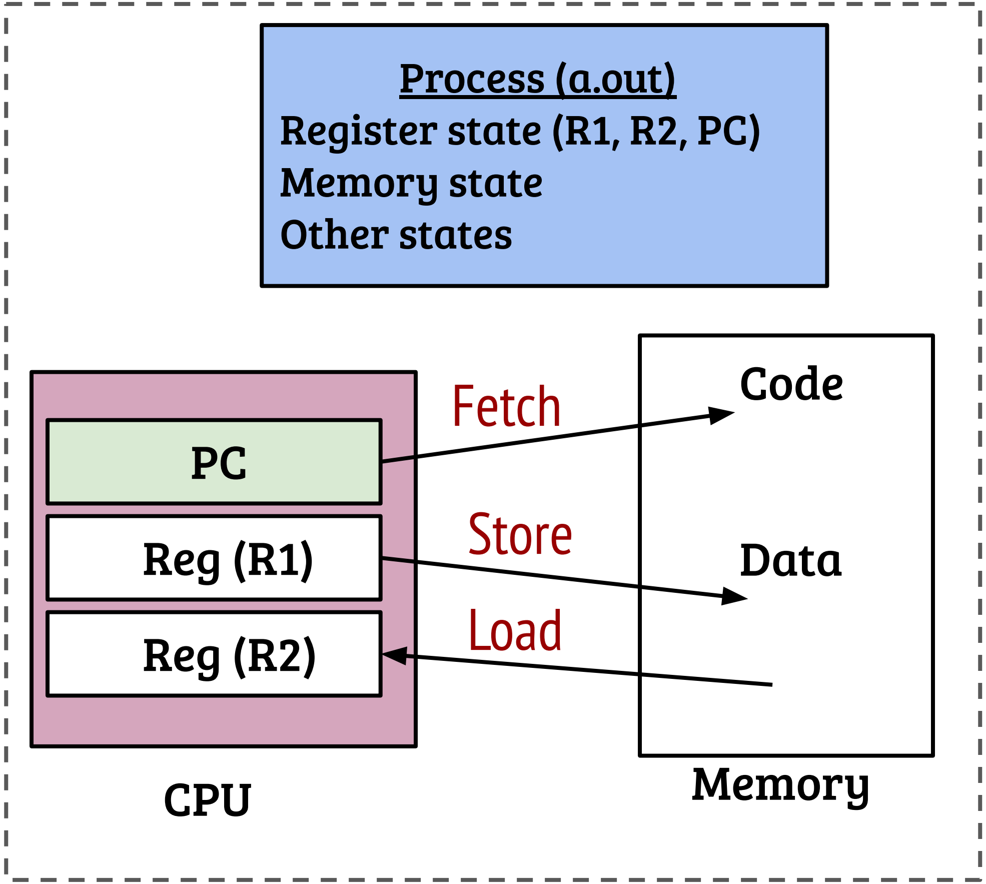
A process in execution

I want to take control of the CPU from this process which is executing an infinite loop, but how?

OS

Help me!

I want to restrict this process accessing memory of other processes, but how? Monitoring each memory access is not efficient!



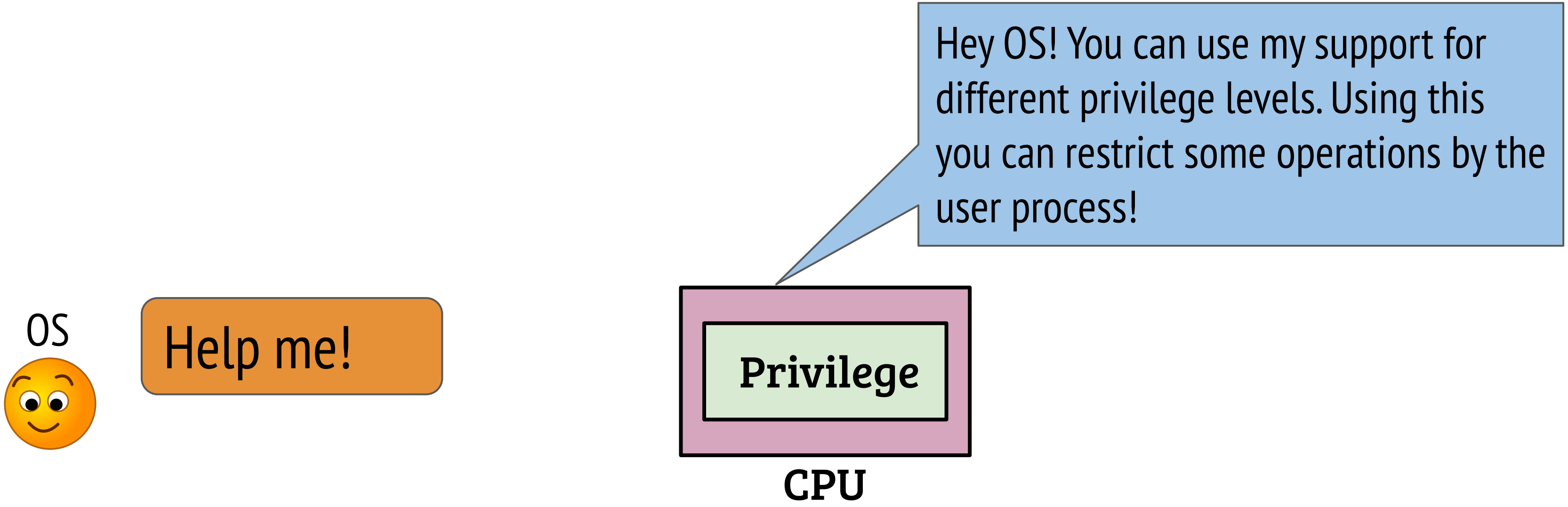
Limited direct execution

- Can the OS enforce limits to an executing process by itself?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!

Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?

Hardware support: Privilege levels



OS

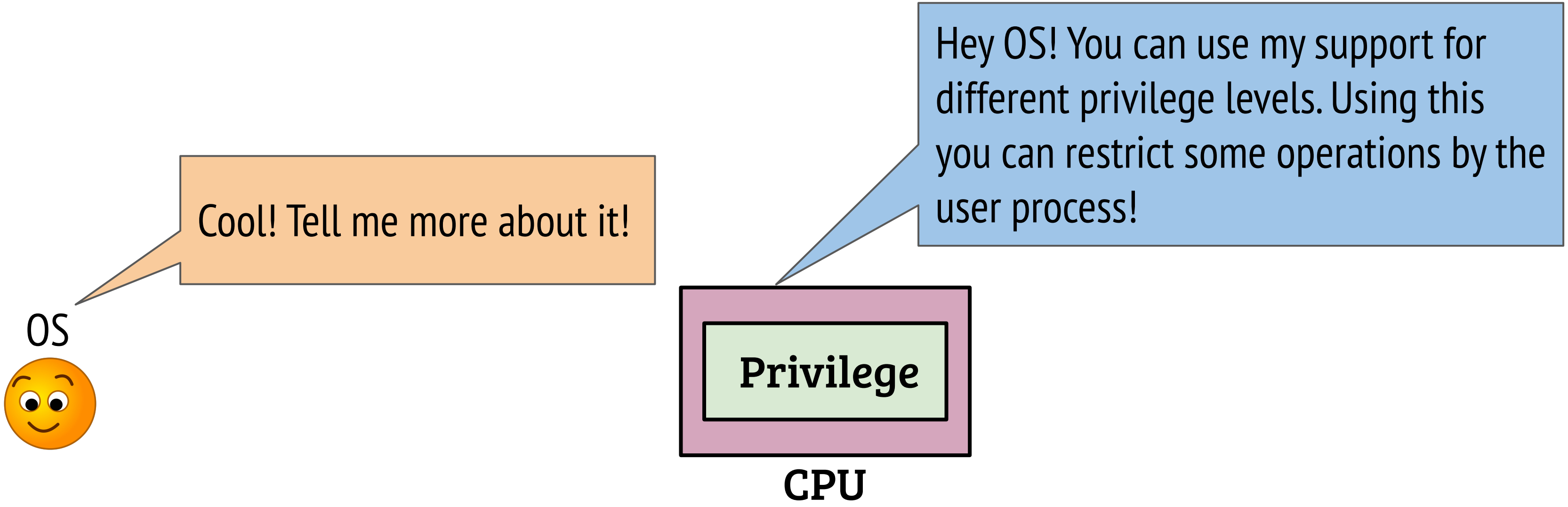
Help me!

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

Privilege

CPU

Hardware support: Privilege levels



Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

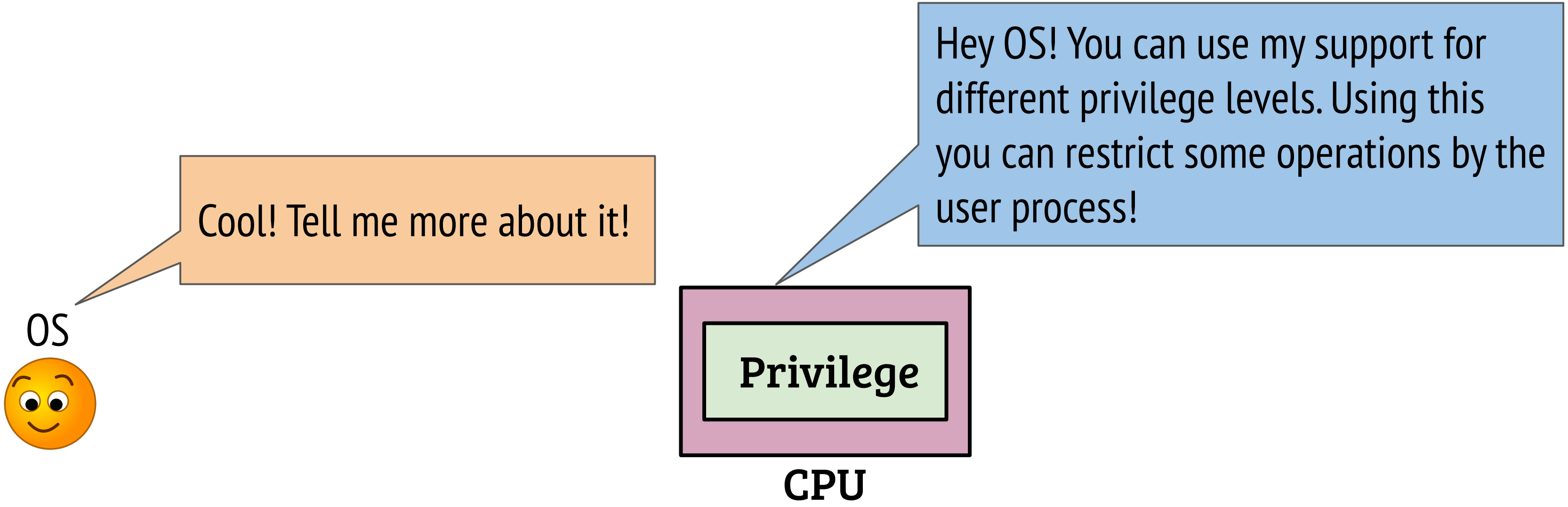
Cool! Tell me more about it!

OS

Privilege

CPU

Hardware support: Privilege levels

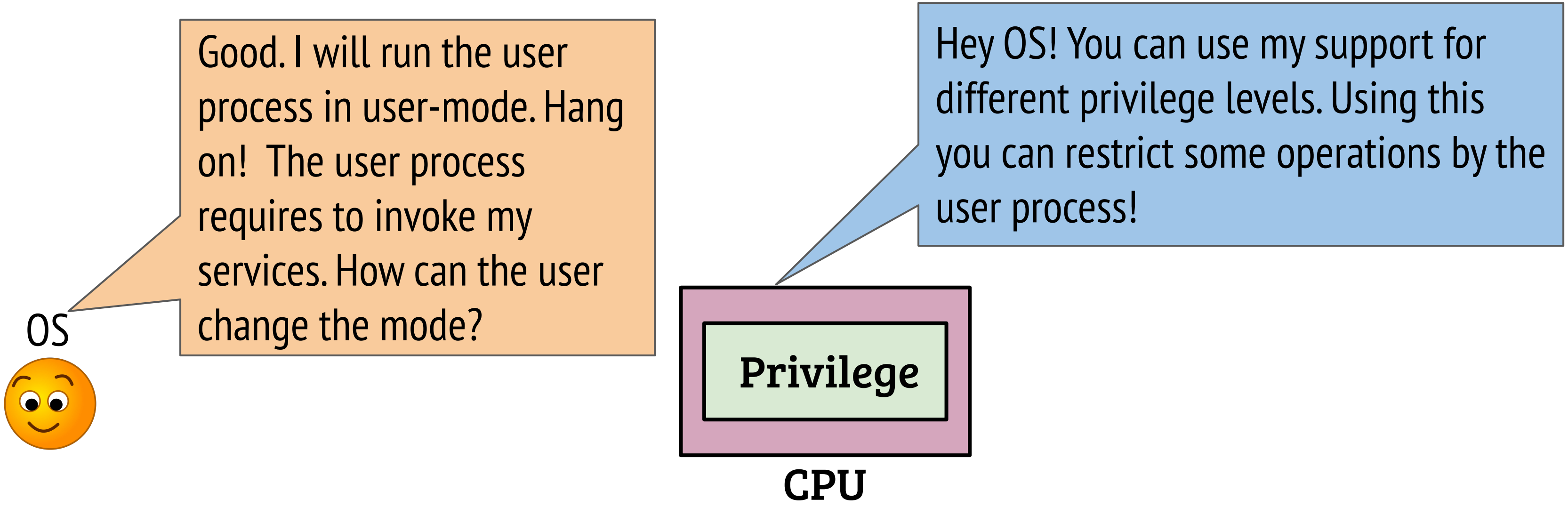


Cool! Tell me more about it!

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

- CPU can execute in two modes: *user-mode* and *kernel-mode*
- Some operations are allowed only from kernel-mode (privileged OPs)
  - If executed from user mode, hardware will notify the OS by raising a fault/trap

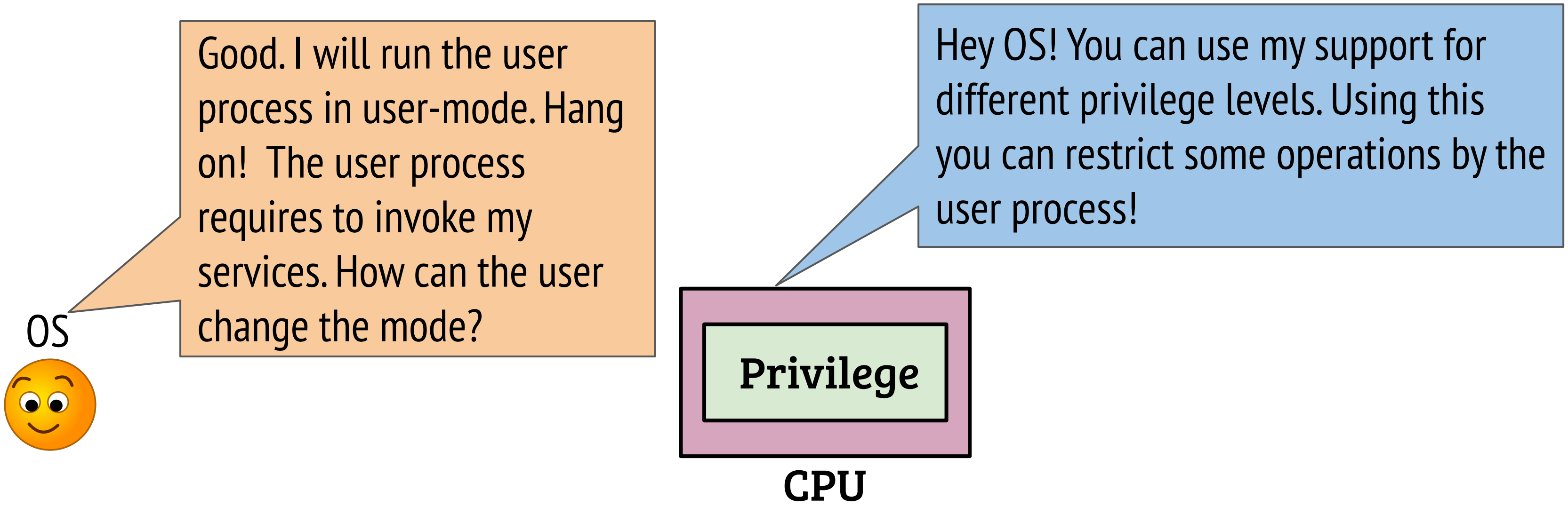
Hardware support: Privilege levels



Good. I will run the user process in user-mode. Hang on! The user process requires to invoke my services. How can the user change the mode?

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

Hardware support: Privilege levels



Good. I will run the user process in user-mode. Hang on! The user process requires to invoke my services. How can the user change the mode?

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

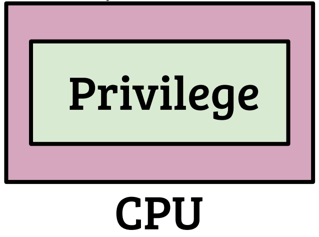
- From user-mode, privilege level of CPU can not be changed directly
- The hardware provides entry instructions from the user-mode which causes a mode switch
- The OS can define the handler for different entry gates

Hardware support: Privilege levels

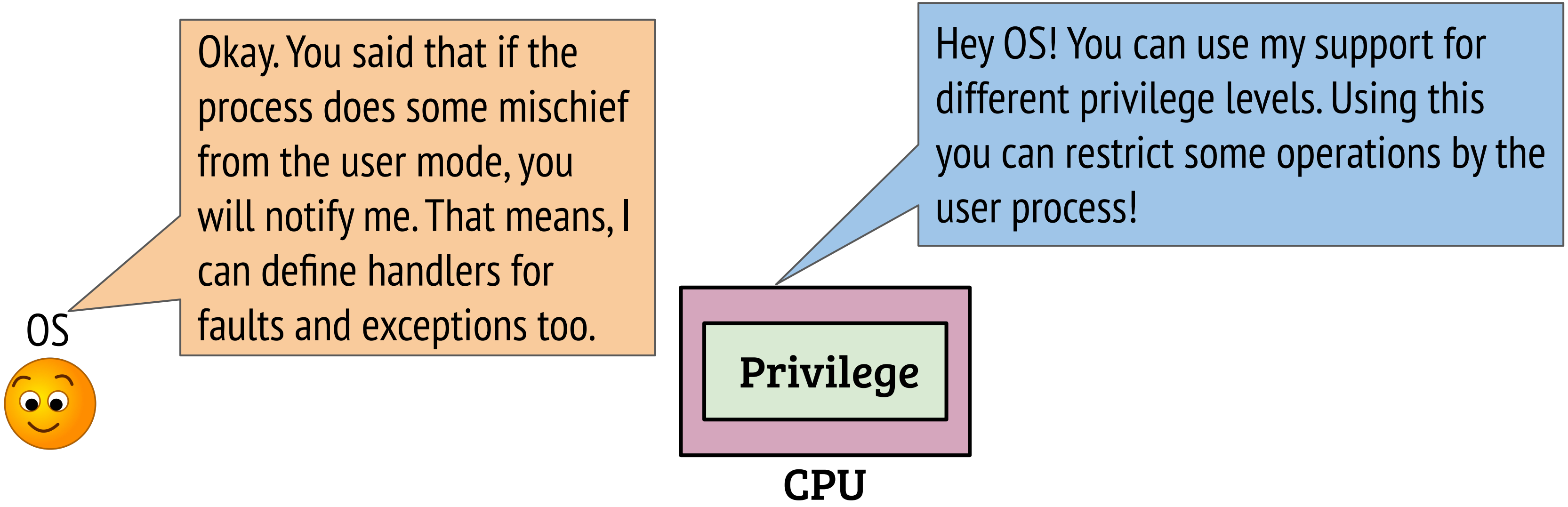
OS

Okay. You said that if the process does some mischief from the user mode, you will notify me. That means, I can define handlers for faults and exceptions too.

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!



Hardware support: Privilege levels

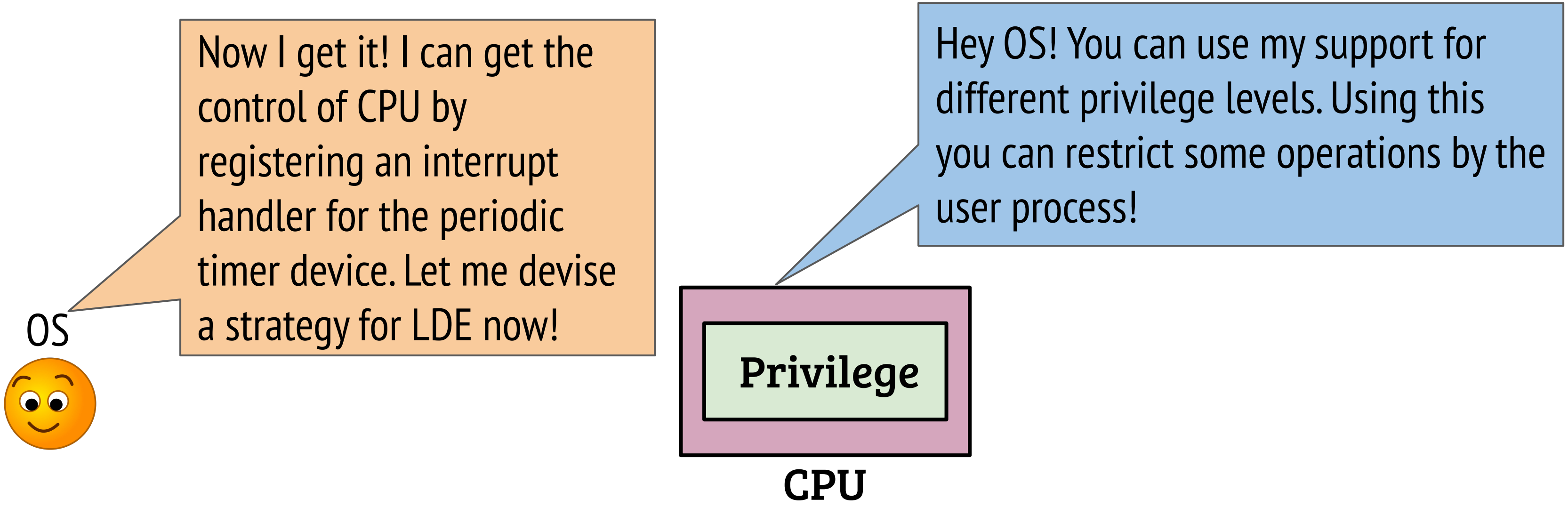


Okay. You said that if the process does some mischief from the user mode, you will notify me. That means, I can define handlers for faults and exceptions too.

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

- The OS can register the handlers for faults and exceptions
- The OS can also register handlers for device interrupts
- *Registration of handlers is privileged!*

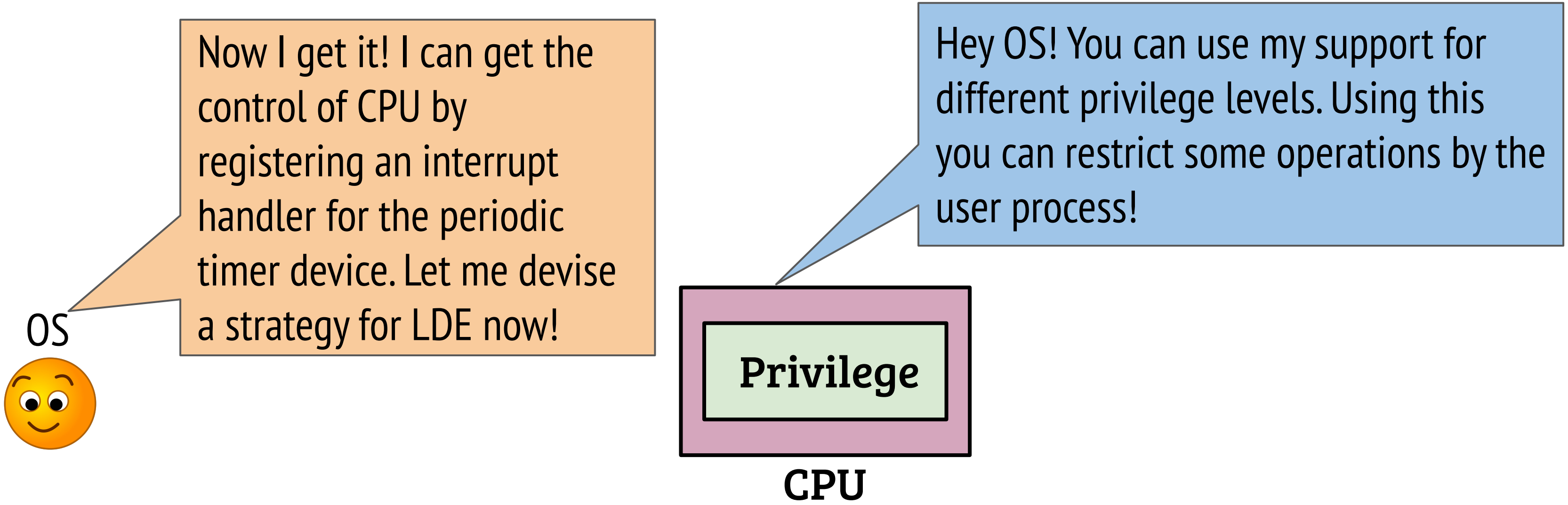
Hardware support: Privilege levels



Now I get it! I can get the control of CPU by registering an interrupt handler for the periodic timer device. Let me devise a strategy for LDE now!

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

Hardware support: Privilege levels

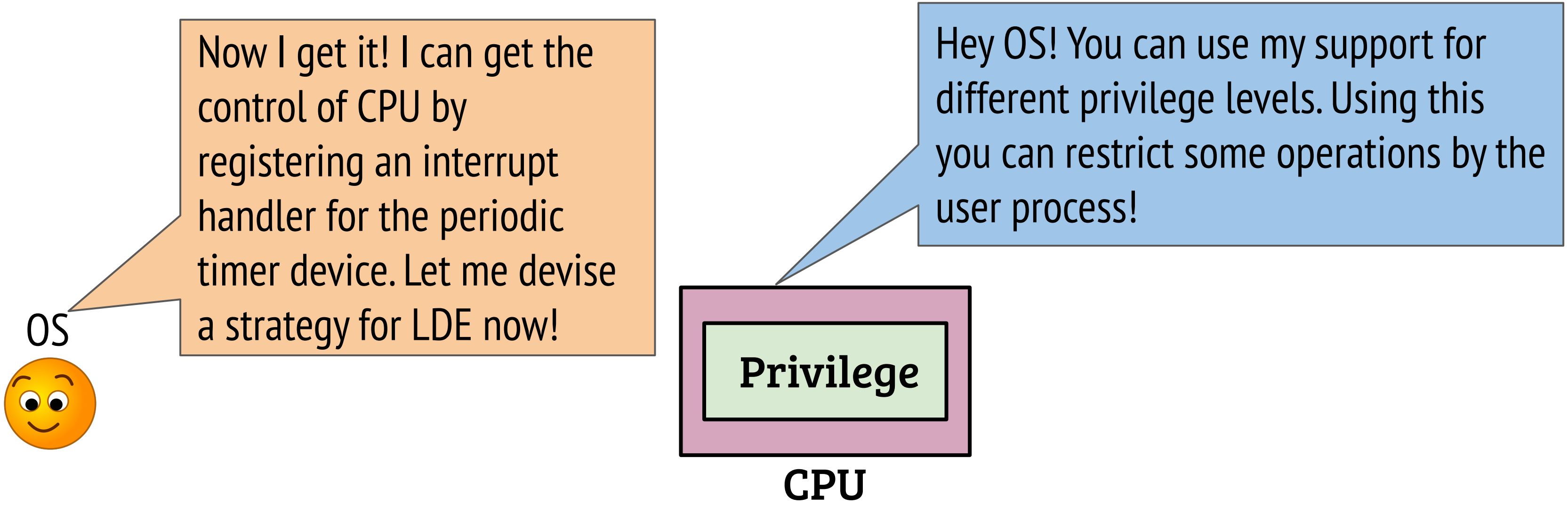


Now I get it! I can get the control of CPU by registering an interrupt handler for the periodic timer device. Let me devise a strategy for LDE now!

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts

Hardware support: Privilege levels



Now I get it! I can get the control of CPU by registering an interrupt handler for the periodic timer device. Let me devise a strategy for LDE now!

Hey OS! You can use my support for different privilege levels. Using this you can restrict some operations by the user process!

- After the boot, the OS needs to configure the handlers for system calls, exceptions/faults and interrupts
- The handler code is invoked by the OS when user-mode process invokes a system call or an exception or an external interrupt

Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
    - How to prove using a user program executing an infinite loop?

Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
    - How to prove using a user program executing an infinite loop?
    - CPU usage in user mode should dominate
    - Is this evidence enough?

Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
    - How to prove using a user program executing an infinite loop?
    - CPU usage in user mode should dominate
    - Is this evidence enough? OS intervention yet to be shown!
    - If a program does division by zero infinitely, we can prove OS intervention. How to go about it?

Evidence based Proof of LDE!

- “Proof” is a term associated with formal world
- “Evidence based proof” is very important for this course
- Proving LDE in the Linux kernel
    - How to prove using a user program executing an infinite loop?
    - CPU usage in user mode should dominate
    - Is this evidence enough? OS intervention yet to be shown!
    - If a program does division by zero infinitely, we can prove OS intervention. How to go about it?
    - Handle the signal and ignore it or something better (modify division-by-zero handler in OS) ⇒ First hand evidence

CS614: Linux Kernel Programming

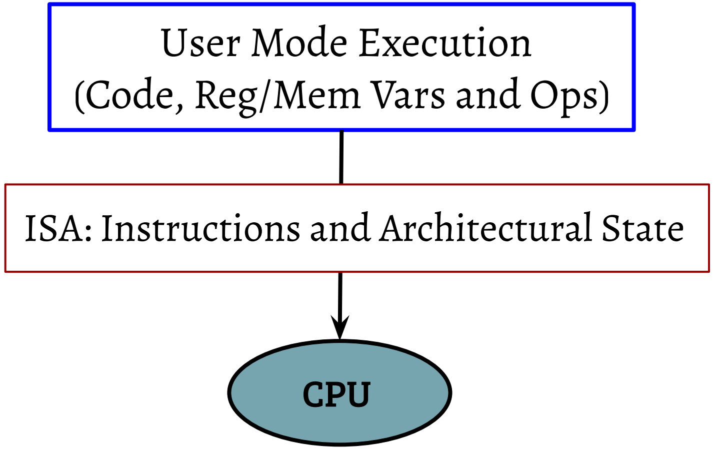
Privileges and Execution Contexts

Debadatta Mishra, CSE, IIT Kanpur

Recap: Limited direct execution

- Can the OS enforce limits to an executing process?
- No, the OS can not enforce limits by itself and still achieve efficiency
- OS requires support from hardware!
- What kind of support is needed from the hardware?
- CPU privilege levels: user-mode vs. kernel-mode
- Switching between modes, entry points and handlers

User Mode Execution: ISA and hardware resources

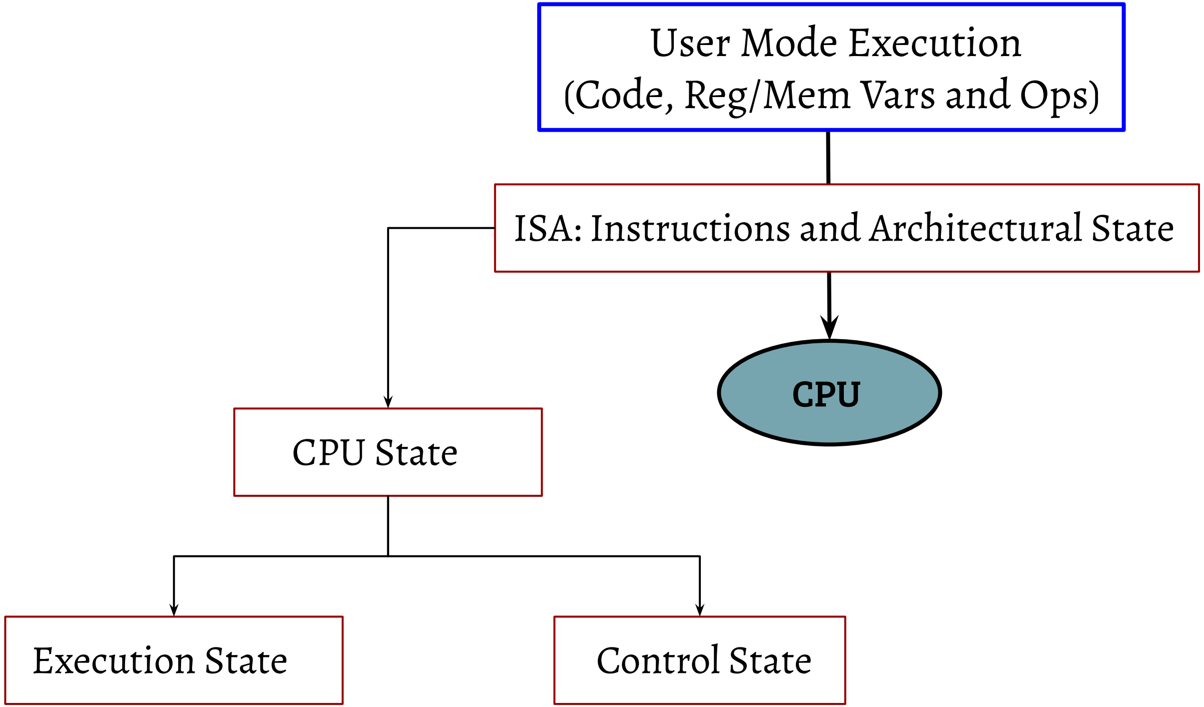


User Mode Execution
(Code, Reg/Mem Vars and Ops)

ISA: Instructions and Architectural State

CPU

User Mode Execution: ISA and hardware resources

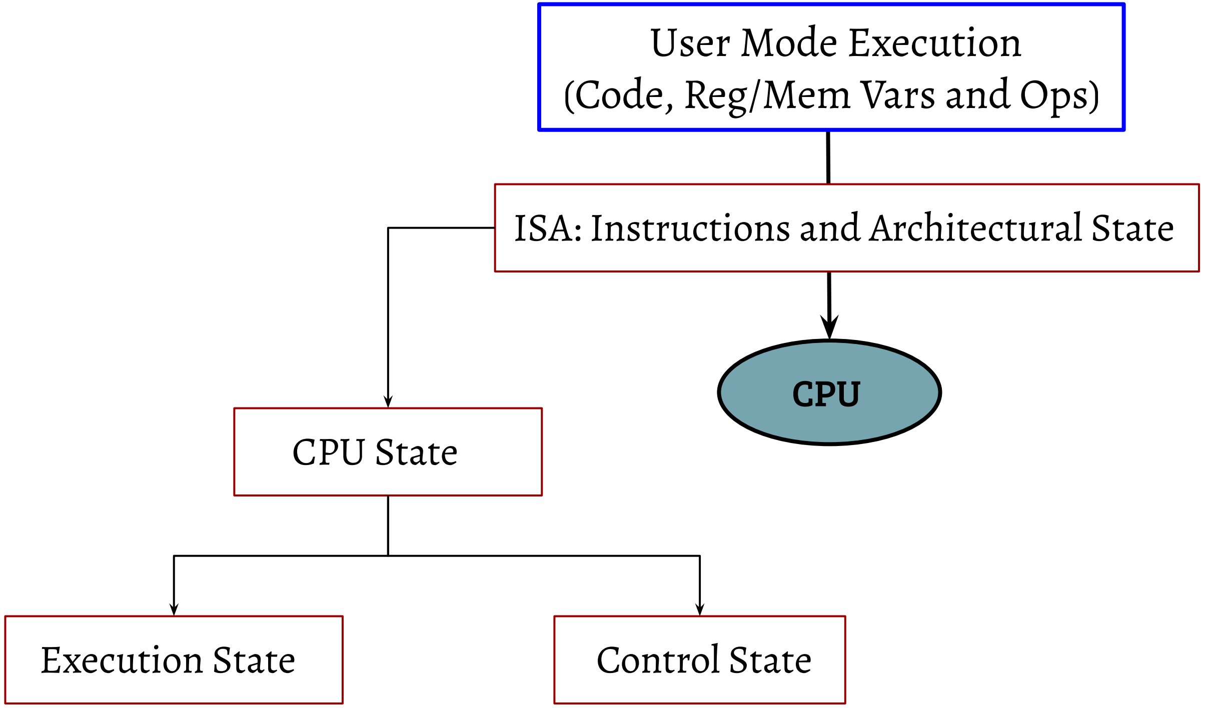

User Mode Execution
(Code, Reg/Mem Vars and Ops)
ISA: Instructions and Architectural State
CPU
CPU State
Execution State
Control State

General purpose registers and special registers (IP, SP etc.)

Control registers dictating the CPU behavior (e.g., CRs in X86)

- What is the OS role to ensure correct user mode execution?
- What about memory state? Is the stack a memory state or a register state?

User Mode Execution: ISA and hardware resources

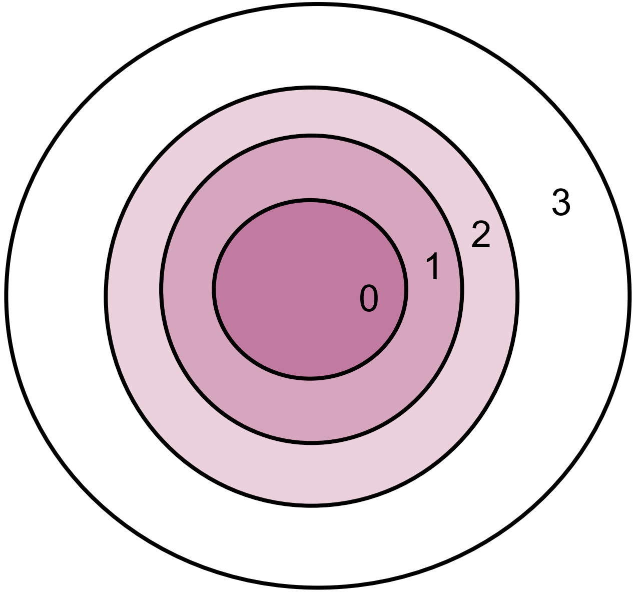

User Mode Execution
(Code, Reg/Mem Vars and Ops)
ISA: Instructions and Architectural State
CPU
CPU State
Execution State
Control State

General purpose registers and special registers (IP, SP etc.)

Control registers dictating the CPU behavior (e.g., CRs in X86)

- What is the OS role to ensure correct user mode execution? OS intervention should not mess-up the state
- What about memory state? Is the stack a memory state or a register state? Stack is maintained in memory, accessed using SP

X86: rings of protection



- 4 privilege levels: 0⟶ highest, 3⟶ lowest
- Some operations are allowed only in privilege level 0
- Most OSes use 0 (for kernel) and 3 (for user)
- Different kinds of privilege enforcement
  - Instruction is privileged
  - Operand is privileged

Privileged instruction: HLT (on Linux x86_64)

```
int main( )
{
  asm(“hlt;”);
}
```

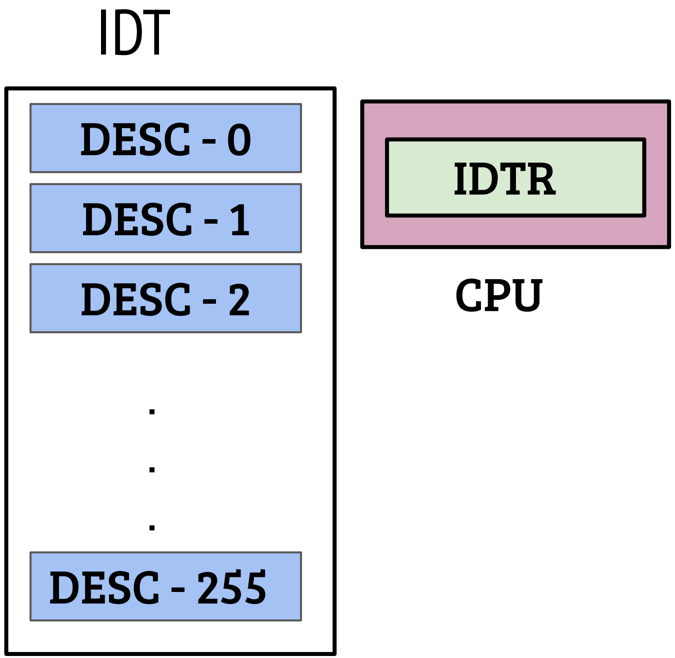
- HLT: Halt the CPU core till next external interrupt
- Executed from user space results in protection fault
- Action: Linux kernel kills the application

Privileged operation: Read CR3 (Linux x86_64)

```
#include<stdio.h>
int main( ){
  unsigned long cr3_val;
  asm volatile("mov %%cr3, %0;"
            : "=r" (cr3_val)
            :: );
 printf("%lx\n", cr3_val);
}
```

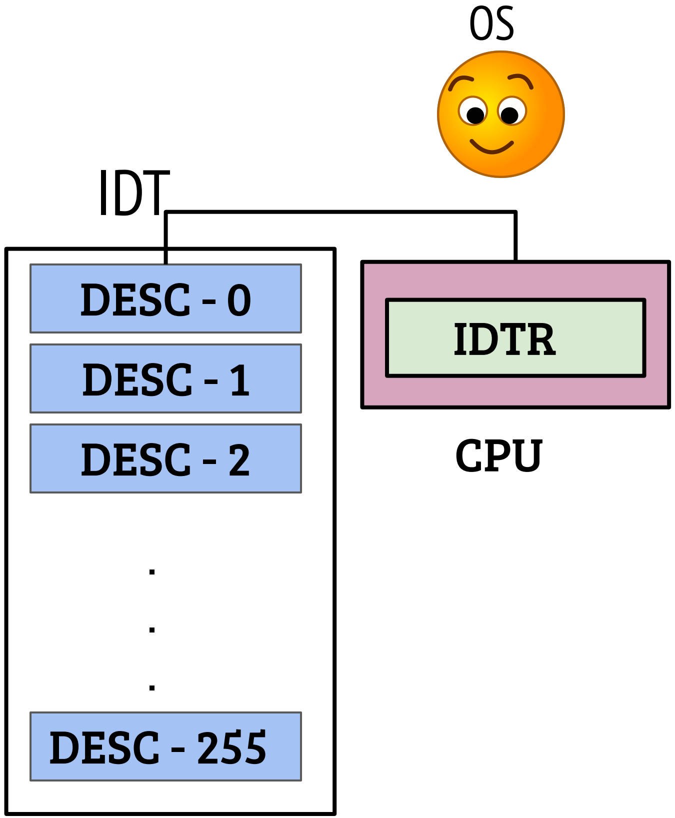
- CR3 register points to the address space translation information
- When executed from user space results in protection fault
- “mov” instruction is not privileged per se, but the operand is privileged

Interrupt Descriptor Table (IDT): gateway to handlers



- Interrupt descriptor table provides a way to define handlers for different events like external interrupts, faults and system calls by defining the descriptors
- Descriptors 0-31 are for predefined events e.g., 0 → Div-by-zero exception etc.
- Events 32-255 are user defined, can be used for h/w and s/w interrupt handling

Defining the descriptors (OS boot)



- Each descriptor contains information about handling the event
  - Privilege switch information
  - Handler address
- The OS defines the descriptors and loads the IDTR register with the address of the descriptor table (using *LIDT* instruction)

System call INT instruction (Conventional Method)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function. How?

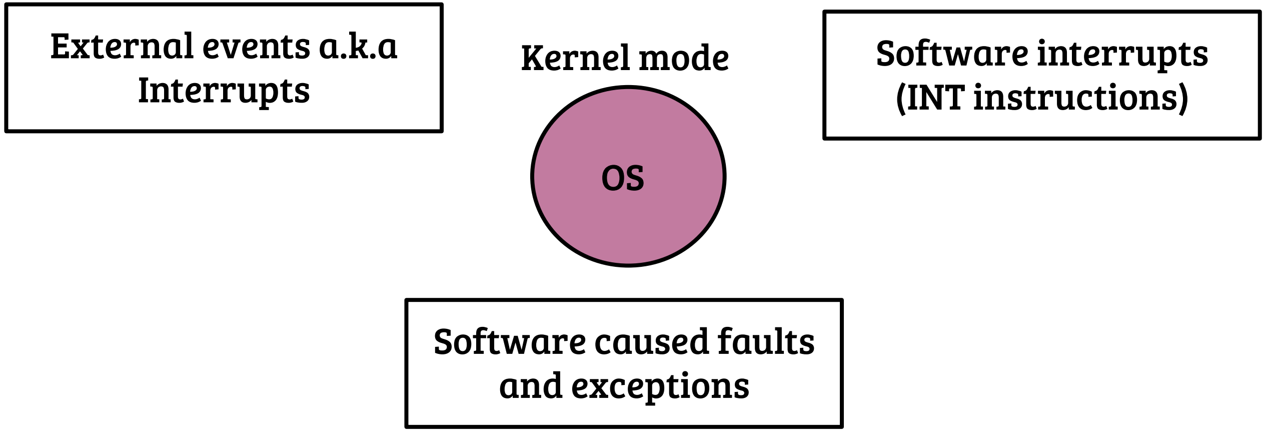
System call INT instruction (Conventional Method)

- INT #N: Raise a software interrupt. CPU invokes the handler defined in the IDT descriptor #N (if registered by the OS)
- Conventionally, IDT descriptor 128 (0x80) is used to define system call entry gates
- The generic system call handler invokes the appropriate handler function, How?
    - Every system call is associated with a number (defined by OS)
    - User process sends information like system call number, arguments through CPU registers which is used to invoke the actual handler

System call in Linux Kernel (using syscall inst.)

- X86 provides a fast system call method through the “syscall” instruction
- OS configures designated privileged registers with the entry address (and other information related to privilege change)
- The hardware saves the next instruction address (user return address) into RCX, change privilege levels and sets RIP to the syscall entry address. (SP and CR3 are not modified)
- Arguments and return value
    - RAX: System call # and return value
    - Arguments passed: RDI, RSI, RDX, R10, R8, R9

Post-boot OS execution



External events a.k.a Interrupts

Kernel mode

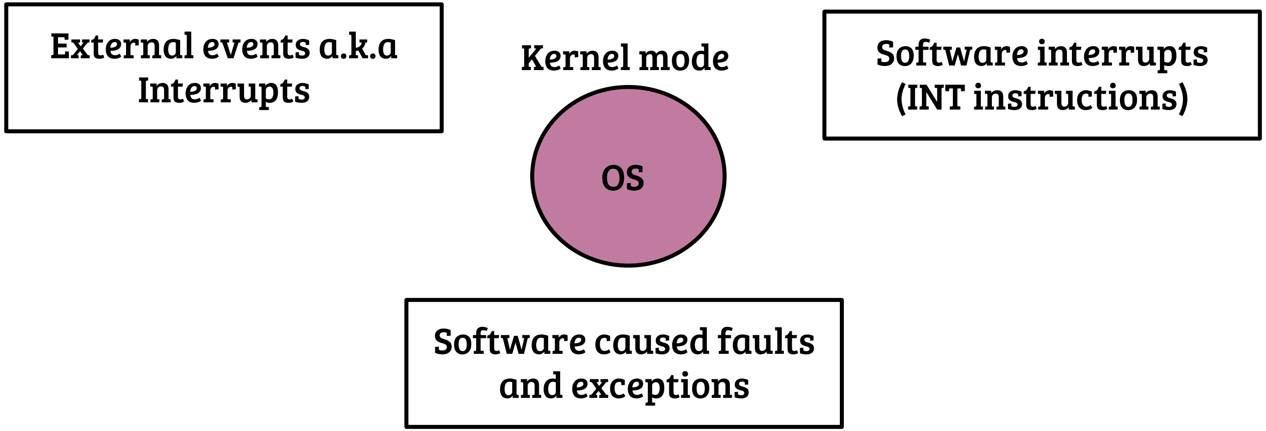
OS

Software interrupts (INT instructions)

Software caused faults and exceptions

- OS execution is triggered because of interrupts, exceptions or system calls

Post-boot OS execution



External events a.k.a Interrupts

Kernel mode

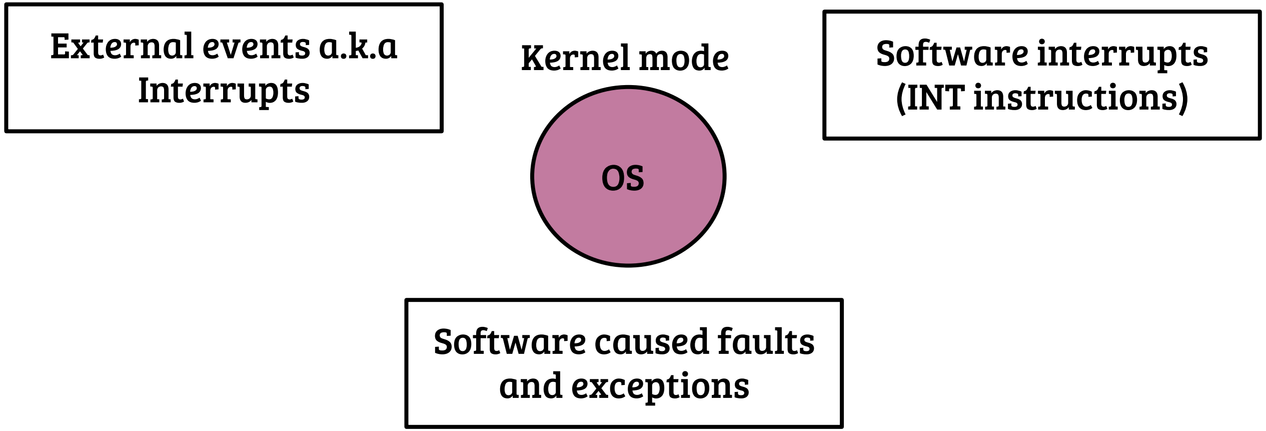
OS

Software interrupts (INT instructions)

Software caused faults and exceptions

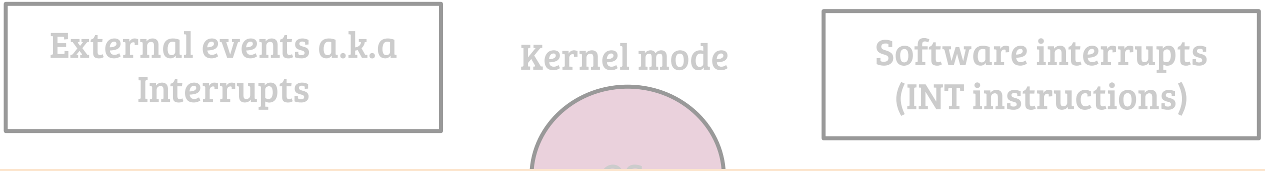
- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?

Post-boot OS execution


External events a.k.a Interrupts
Kernel mode
OS
Software interrupts (INT instructions)
Software caused faults and exceptions

- OS execution is triggered because of interrupts, exceptions or system calls
- Exceptions and interrupts are abrupt, the user process may not be prepared for this event to happen. What can go wrong and how to handle it?
- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

Post-boot OS execution


External events a.k.a Interrupts
Kernel mode
Software interrupts (INT instructions)

- Does the OS need a separate stack?
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

for this event to happen. What can go wrong and how to handle it?

- The interrupted program may become corrupted after resume! The OS need to save the user execution state and restore it on return

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
    - The user may have an invalid SP at the time of entry
    - OS need to erase the used area before returning

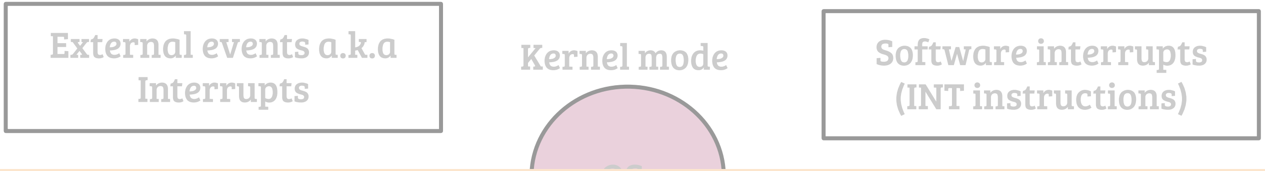
The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?

The OS stack

- OS execution requires a stack for obvious reasons (function call & return)
- Can the OS use the user stacks?
- No. Because of security and efficiency reasons,
  - The user may have an invalid SP at the time of entry
  - OS need to erase the used area before returning
- If OS has its own stack, who switches the stack on kernel entry?
- On X86 systems, the hardware (or OS in case of "syscall") switches the stack pointer to the stack address configured by the OS

Post-boot OS execution



External events a.k.a Interrupts

Kernel mode

Software interrupts (INT instructions)

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- How the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

The interrupted program may become corrupted after resume. The OS need to save the user execution state and restore it on return

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working?

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
    - The OS configures the kernel stack address of the currently executing process in the hardware
    - The hardware switches the stack pointer on system call or exception

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
  - The OS configures the kernel stack address of the currently executing process in the hardware
  - The hardware switches the stack pointer on system call or exception
- What about external interrupts?

Management of OS stacks

- A per-process OS stack is required to allow multiple processes to be in OS mode of execution simultaneously
- Working
    - The OS configures the kernel stack address of the currently executing process in the hardware
    - The hardware switches the stack pointer on system call or exception
- What about external interrupts?
    - Separate interrupt stacks are used by OS for handling interrupts

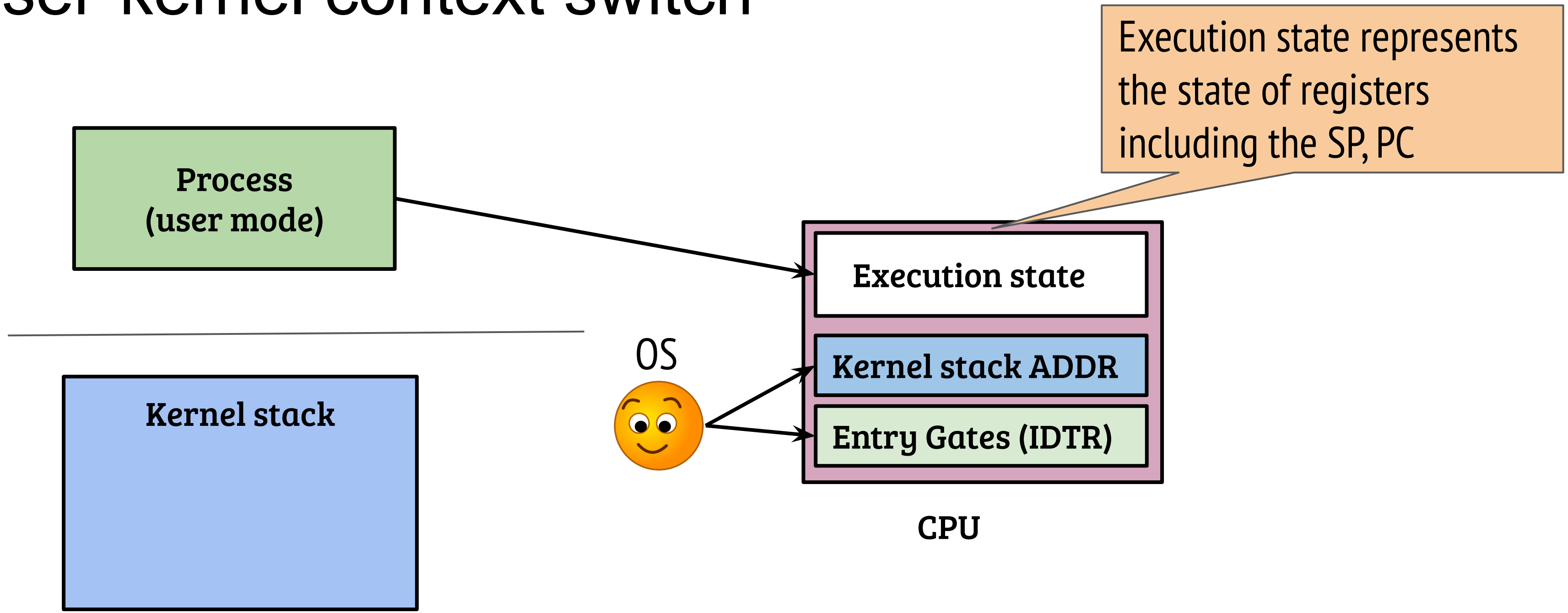
Post-boot OS execution

External events a.k.a Interrupts
Kernel mode
Software interrupts (INT instructions)

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How is the user process state preserved on entry to OS and restored on return to user space?
- Which address space the OS uses?

The interrupted program may become corrupted after resume. The OS need to save the user execution state and restore it on return

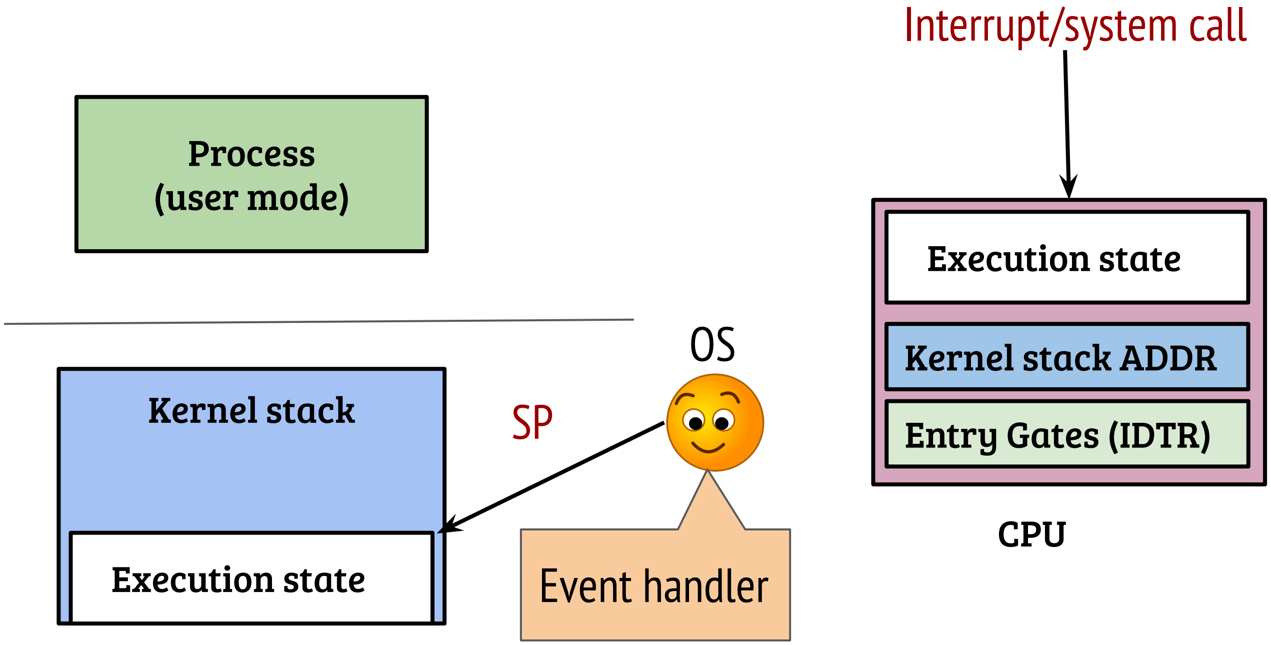
User-kernel context switch



Execution state represents the state of registers including the SP, PC

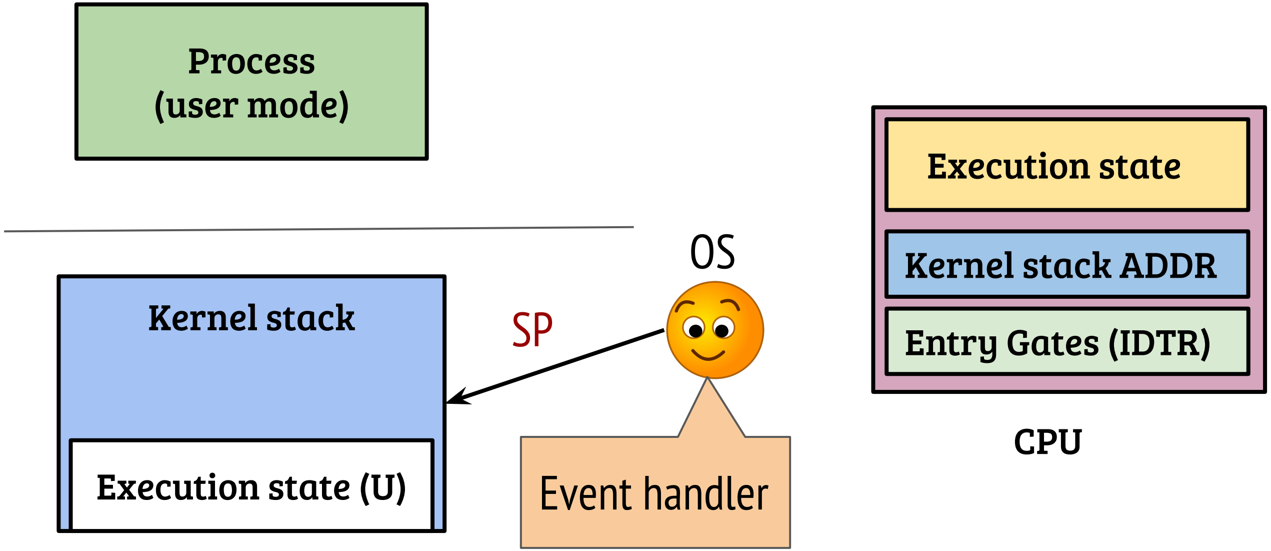
- The OS configures the kernel stack of the process before scheduling the process on the CPU

User-kernel context switch



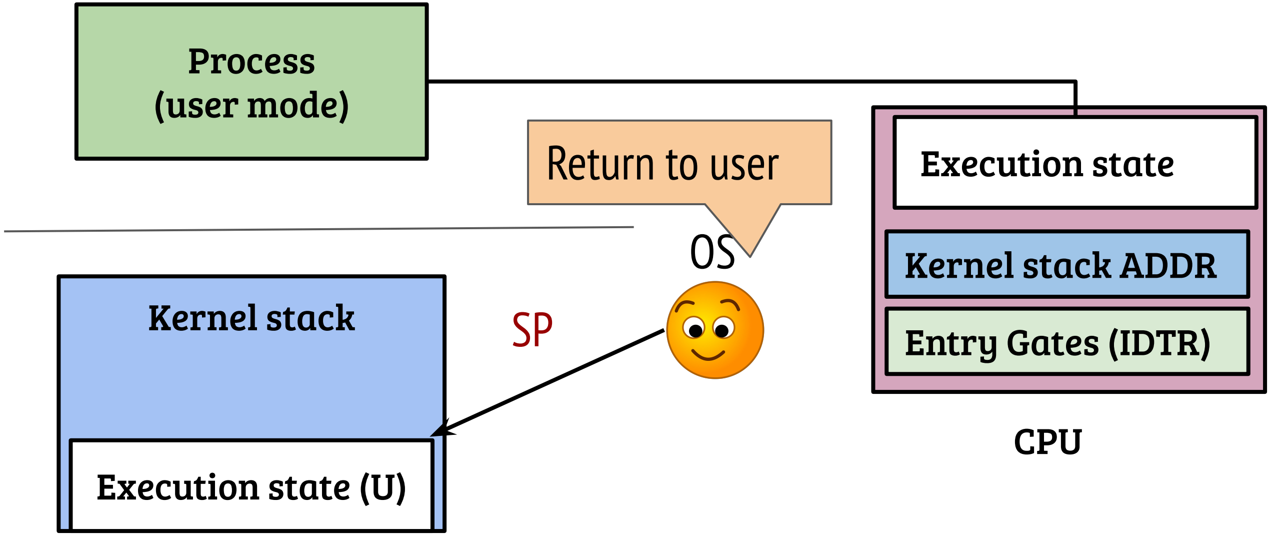
- The CPU saves the execution state onto the kernel stack
- The OS handler finds the SP switched with user state saved (fully or partially depending on architectures)

User-kernel context switch



- The OS executes the event (syscall/interrupt) handler
  - Makes uses of the kernel stack
  - Execution state on CPU is of OS at this point

User-kernel context switch



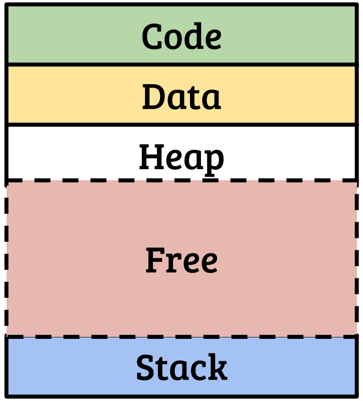
- The kernel stack pointer should point to the position at the time of entry
- CPU loads the user execution state and resumes user execution

Post-boot OS execution

- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?

to save the user execution state and restore it on return

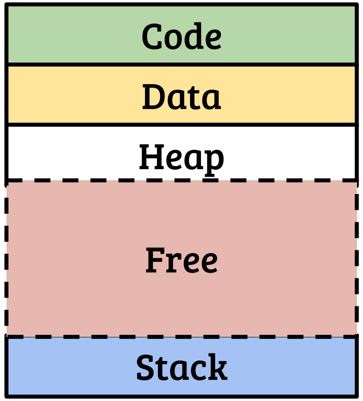
The OS address space

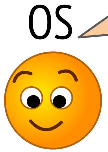


OS

Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

The OS address space





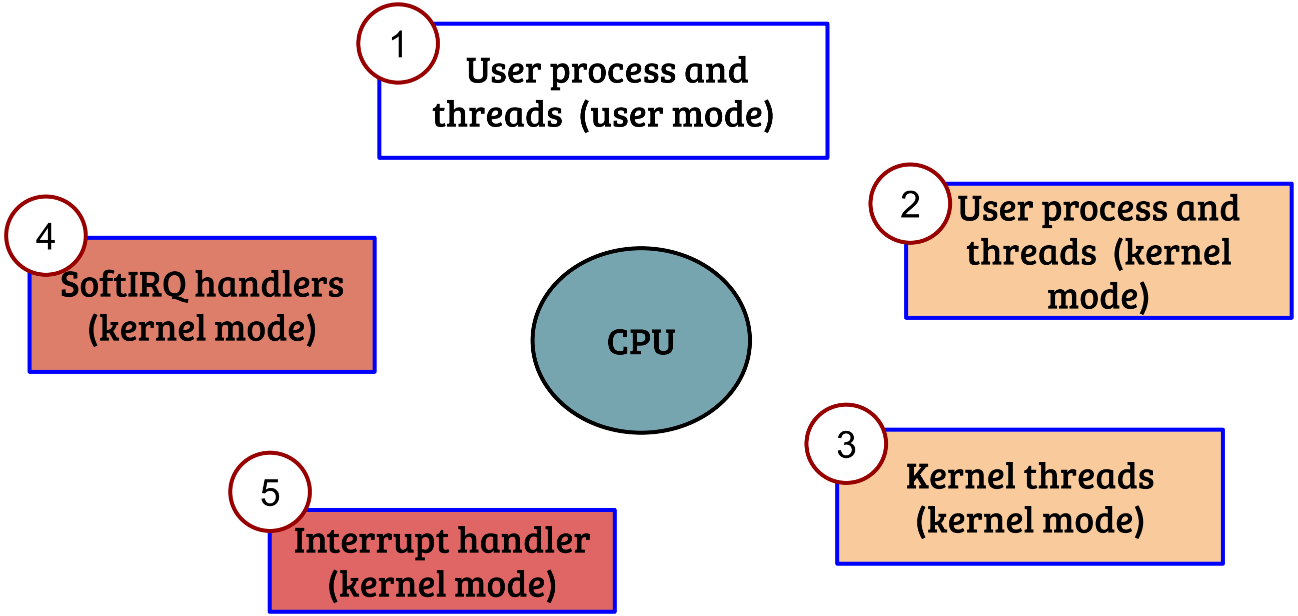
Not only I have to enable address space for each process, I need an address space myself which is protected from the user processes. Design?

- Two possible design approaches
  - Use a separate address space for the OS, change the translation information on every OS entry (inefficient)
  - Consume a part of the address space from all processes and protect the OS addresses using H/W assistance (most commonly used)

Post-boot OS execution

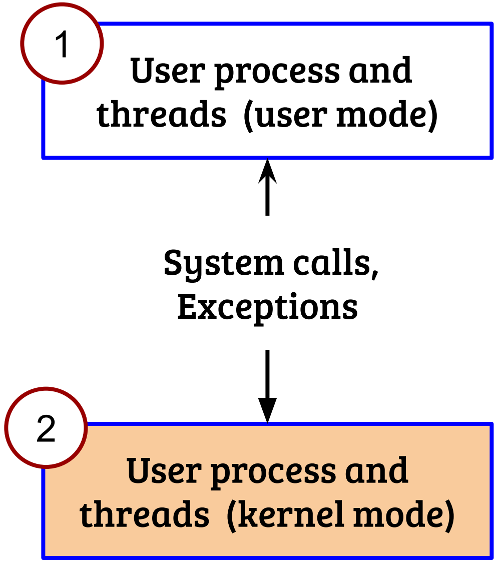
- Does the OS need a separate stack?
- Yes, the hardware switches the SP to point it to a configured OS stack
- How many OS stacks are required?
- For every process, a kernel stack is required
- How the user process state preserved on entry to OS and restored on return to user space?
- The user execution state is saved/restored using the kernel stack by the hardware (and OS)
- Which address space the OS uses?
- A part of the process address space is reserved for OS and is protected

Execution contexts in Linux


1
User process and threads (user mode)
2
User process and threads (kernel mode)
4
SoftIRQ handlers (kernel mode)
CPU
3
Kernel threads (kernel mode)
5
Interrupt handler (kernel mode)

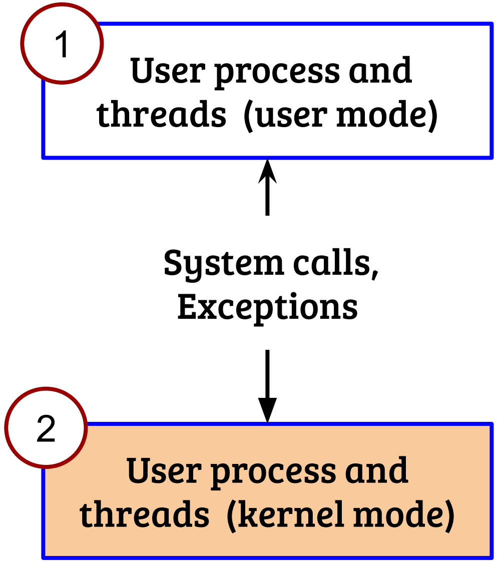
- In a linux system, the CPU can be executing in one of the above contexts
- For (3), (4) and (5), the context is not associated with any user process

User contexts


1
User process and threads (user mode)
System calls, Exceptions
2
User process and threads (kernel mode)

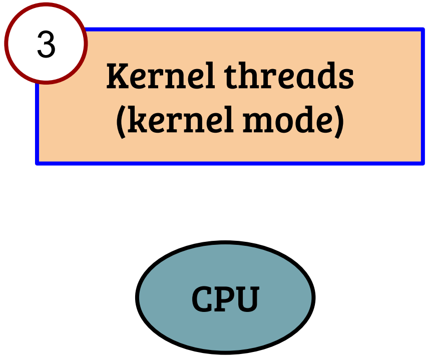
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can a process sleep { in (1) and (2) }?
- Can a process in user mode preempted?
- Can a process in kernel mode preempted?

User contexts


1
User process and threads (user mode)
System calls, Exceptions
2
User process and threads (kernel mode)

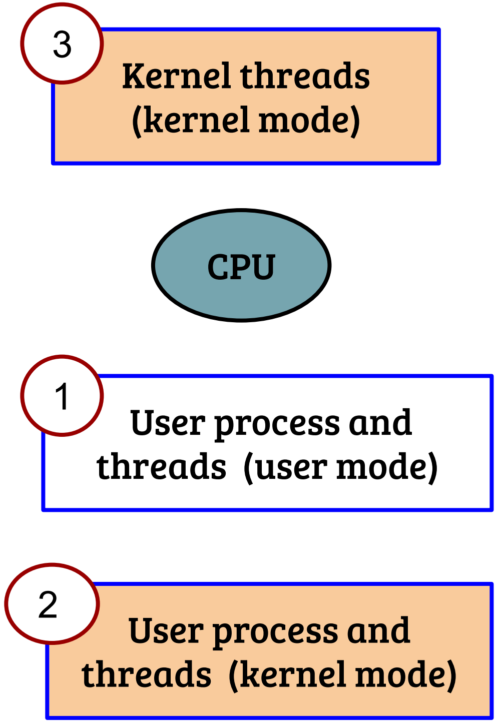
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change, CR3 changes if PTI enabled
- Can a process sleep { in (1) and (2) }?
- Yes, it can (lock holding conditions apply for 2)
- Can a process in user mode be preempted?
- Yes
- Can a process in kernel mode be preempted?
- Yes (if not explicitly disabled)

Kernel threads



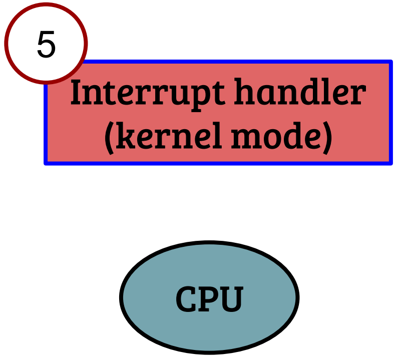
- Kernel threads are independent of user processes and threads
- Created in kernel using *kthread_create*
- How is a kernel thread different?
- Can it sleep?
- Can it be be preempted?
- Which contexts can preempt a kernel thread?

Kernel threads


3
Kernel threads (kernel mode)
CPU
1
User process and threads (user mode)
2
User process and threads (kernel mode)

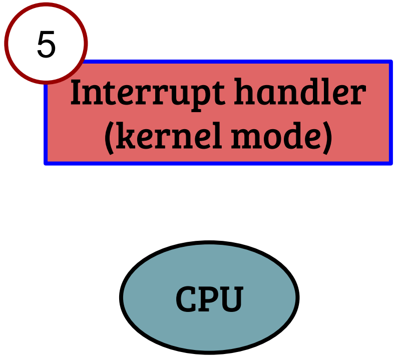
- How is a kernel thread different?
- Kernel thread never executes in user mode
- Does not require a MM context of its own
- Can it sleep?
- Yes, it can (lock holding conditions apply)
- Can it be be preempted?
- Yes (if not explicitly disabled)
- Which contexts can preempt a kernel thread?
- User, Interrupt and SoftIRQ

Hardware interrupts (Background)



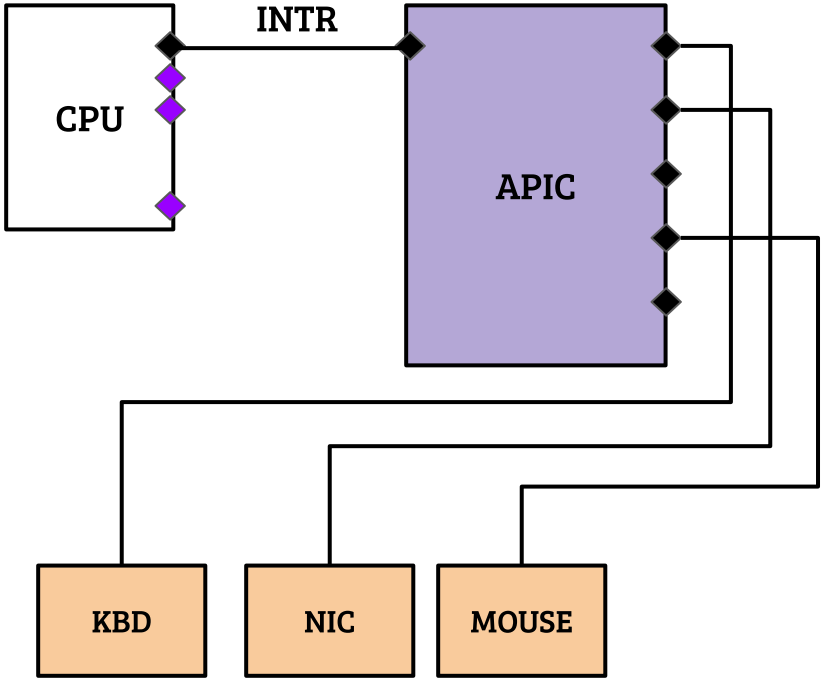
- Why interrupts?
- Example: Receive a packet from network
- What are the architectural support?

Hardware interrupts (Background)



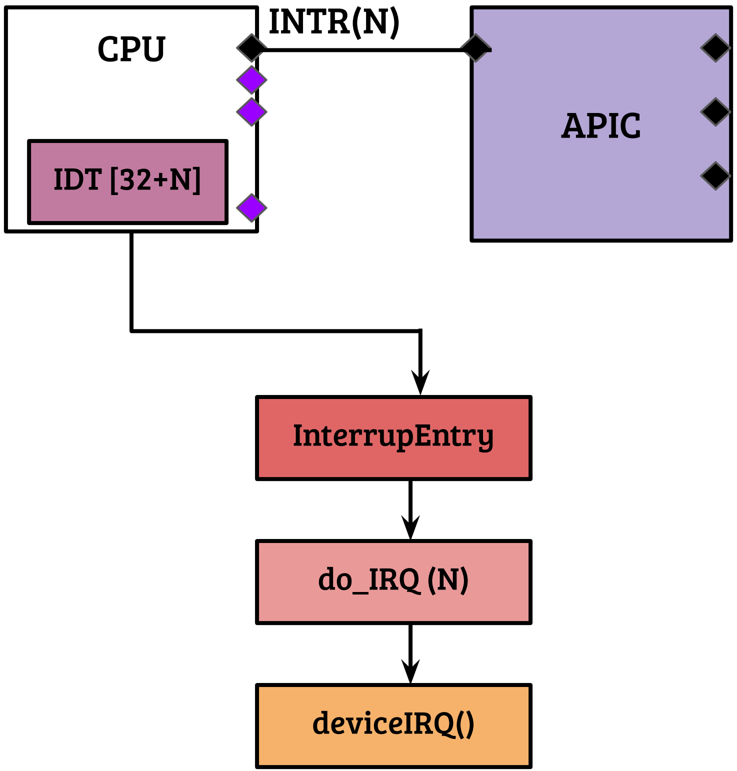
- Why interrupts?
- Example: Receive a packet from network
- Avoid CPU wastage due to polling
- Responsive and scalable systems
- What are the architectural support?
- CPU has limited #of interrupt PINs ⟶ How to multiplex many devices?

Interrupt architecture - PIC and APIC



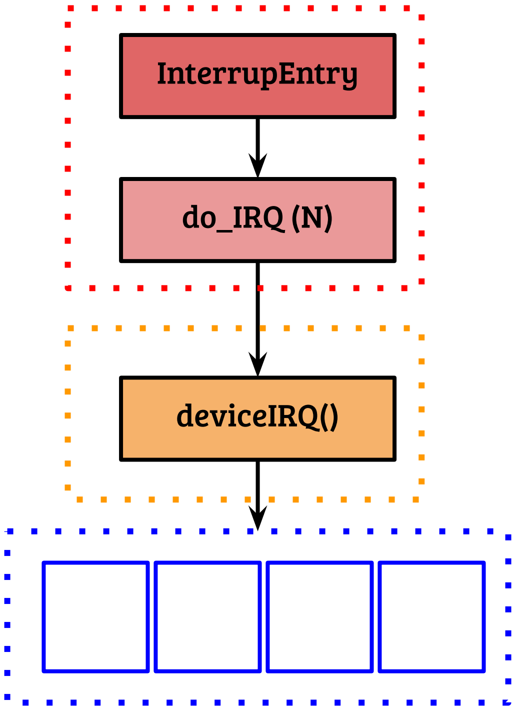
- Every device attached to the APIC is configured with a unique IRQ number
- APIC saves the IRQ in a control port register and raise CPU interrupt line on receipt of device interrupt
- CPU reads the IRQ number and invokes the interrupt handler
- Waits for acknowledgement before clearing the INTR line
- Selective disabling of IRQs possible
    - != cli (CPU interrupt disable)
    - New interrupts not lost

Interrupt handling



- IDT configured to load the interrupt execution context (CPL and stack)
- Interrupt entry: save regs, switch CR3 if needed
- do_IRQ checks the descriptor flags and invokes the real handler
- The device driver handler implements the device specific functionalities
- When is the interrupt acknowledged (i.e., INTR is cleared)?
- How long is the device interrupt masked?
- Not all interrupts can be handled quickly, e.g., NIC RCV

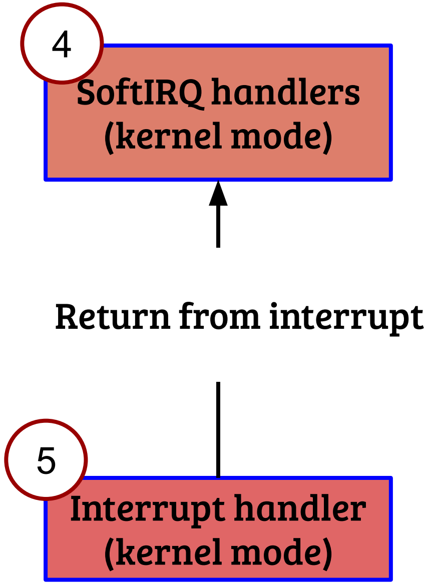
Interrupt handling in three stages



Events for Deferred Processing

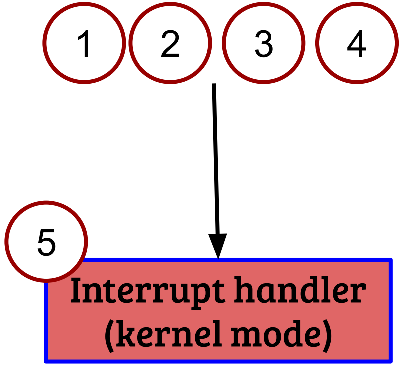
- Critical tasks: Interrupt context setup, APIC acknowledgement
- Semicritical: Accessing/updating device state, e.g., update receive queue pointers of a NIC
- Deferrable: Actions that are device independent e.g., Network stack processing

Interrupt handling: SoftIRQ


4
SoftIRQ handlers (kernel mode)
Return from interrupt
5
Interrupt handler (kernel mode)

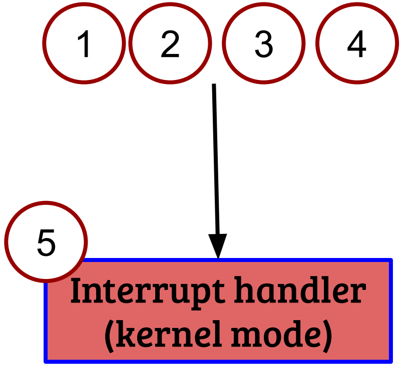
- Carry out deferrable operations, can be preempted by interrupts
- Like an interrupt, it can be raised, disabled, enabled, masked
- Executed by the local CPU kernel thread (*ksoftirqd*, one per CPU)
    - Infinite loop checking for pending softIRQ (set when softirq is raised)
    - Often scheduled on irq_exit( ) or explicit wakeup

Interrupt context



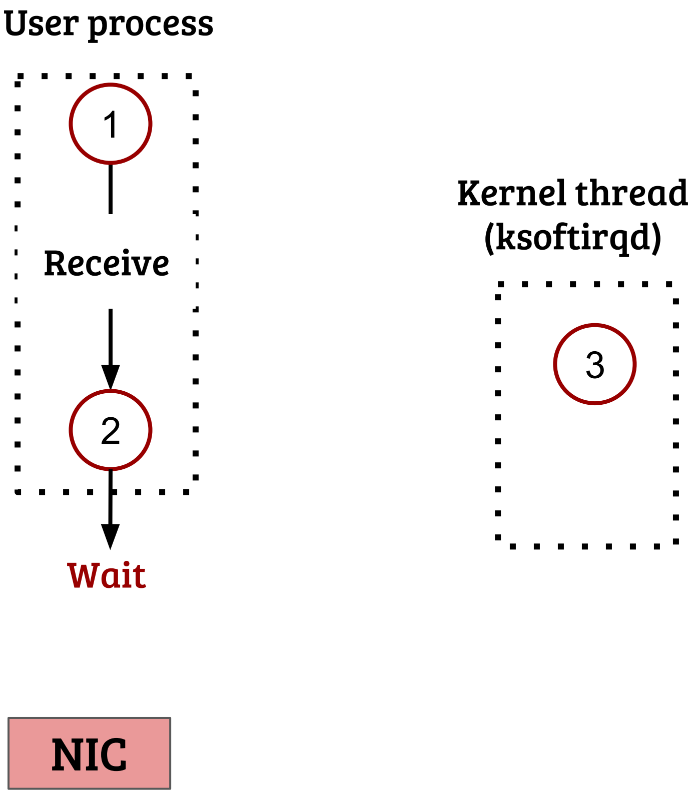
- What are the changes in the CPU state? {CPL, Stack, CR3}
- Can an interrupt handler sleep?
- Can it be preempted?

Interrupt context



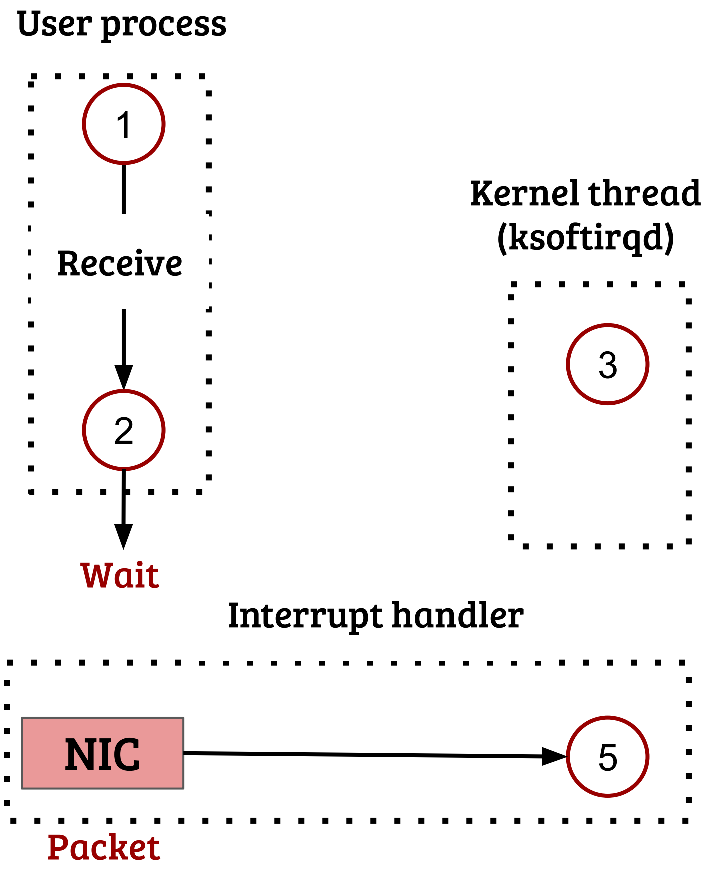
- What are the changes in the CPU state? {CPL, Stack, CR3}
- CPL and Stack change (interrupt stack used), CR3 changes if entering from user mode in a PTI enabled system
- Can an interrupt handler sleep?
- No, Linux does not allow sleeping (directly/indirectly) in an interrupt handler
- Can it be preempted?
- Only by another interrupt (if APIC Acked and interrupts enabled on CPU)

Contexts in action: network receive



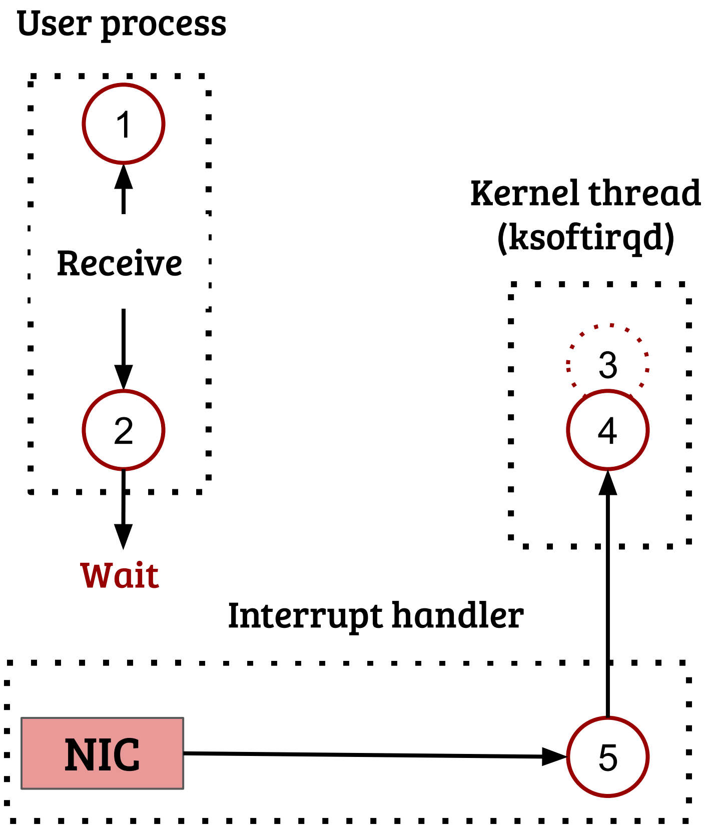
- The user process invokes recv( ) system call (blocking)
- No processed payload found, the process is descheduled and put into a wait queue
- Ksoftirqd is either suspended or processing other pending softIRQs

Contexts in action: network receive



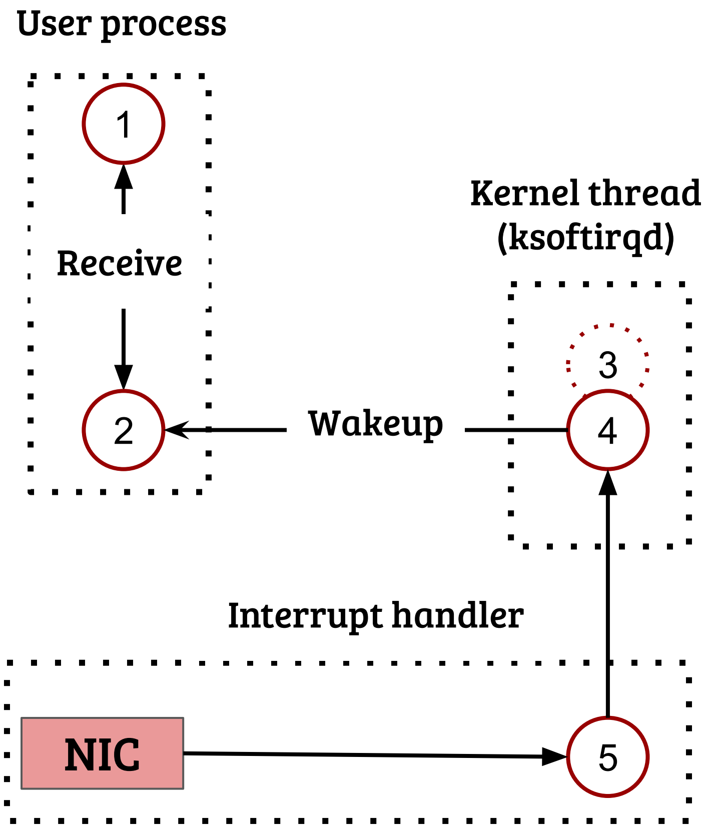
- The NIC copies the packet (using DMA) into memory buffers (a.k.a. skbuffs) and triggers the interrupt
- Before the device specific interrupt handling, APIC is acknowledged
- The device interrupt handler update the device state while masking device interrupts
- Queues the packet for further processing and triggers a softIRQ

Contexts in action: network receive



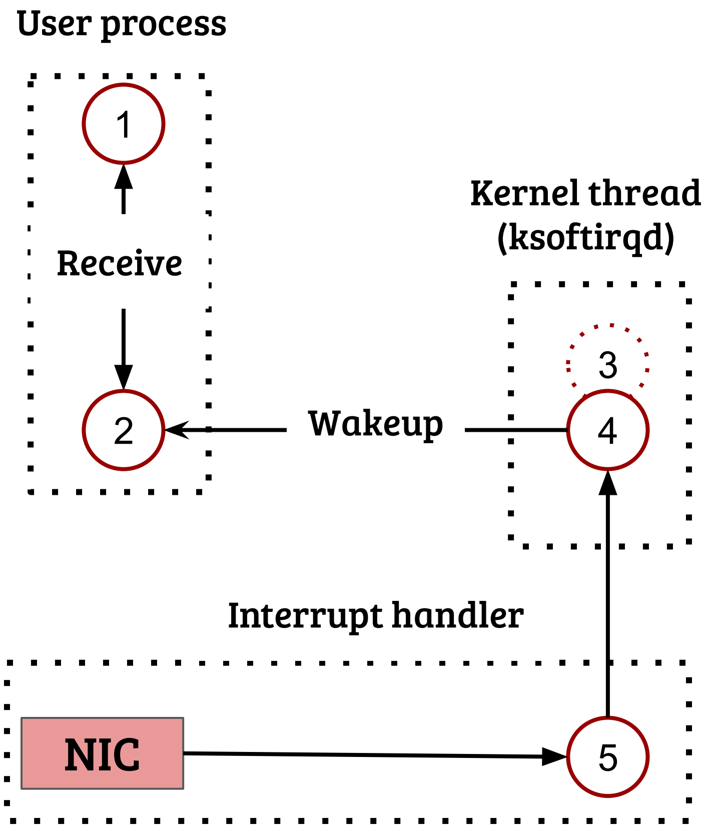
- The softIRQ is scheduled using the ksoftirqd kernel thread context
- Protocol stack processing is performed in this context
- As part of the protocol processing, the destination process is derived

Contexts in action: network receive



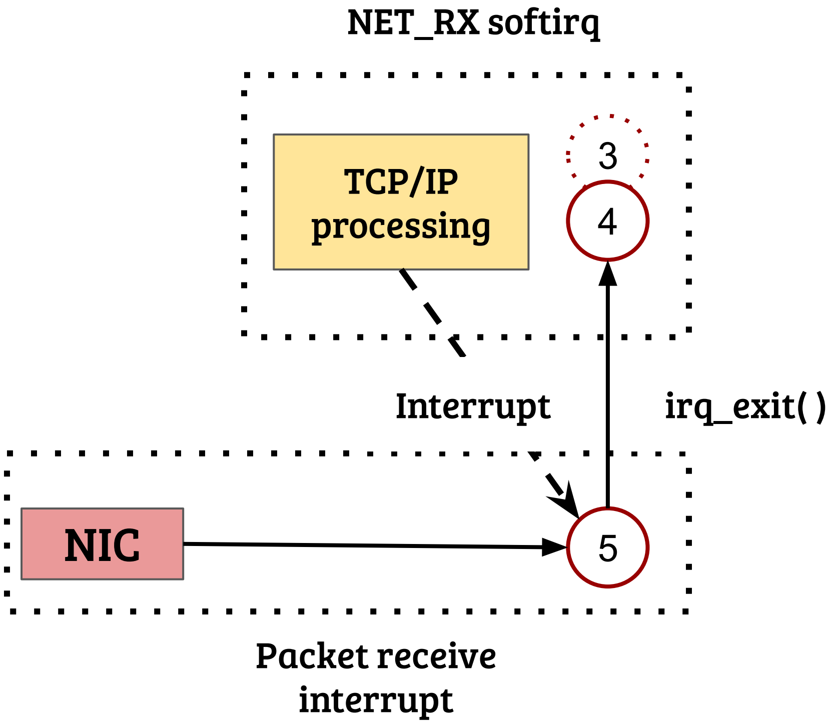
- The softIRQ processing wakes up the user process
- The user process returns from syscall (copy payload to user)
- Now, what could be the issues with this approach?

Challenges in network receive



- Minimize network packet copy across the contexts
- Precise scheduling: application progress and fairness
- Network is always overdriven and self-adjusting in nature → rate limit as early as possible
- Issues
  - Receive livelock: CPU is always handling interrupts
  - User process starvation due to softIRQ processing

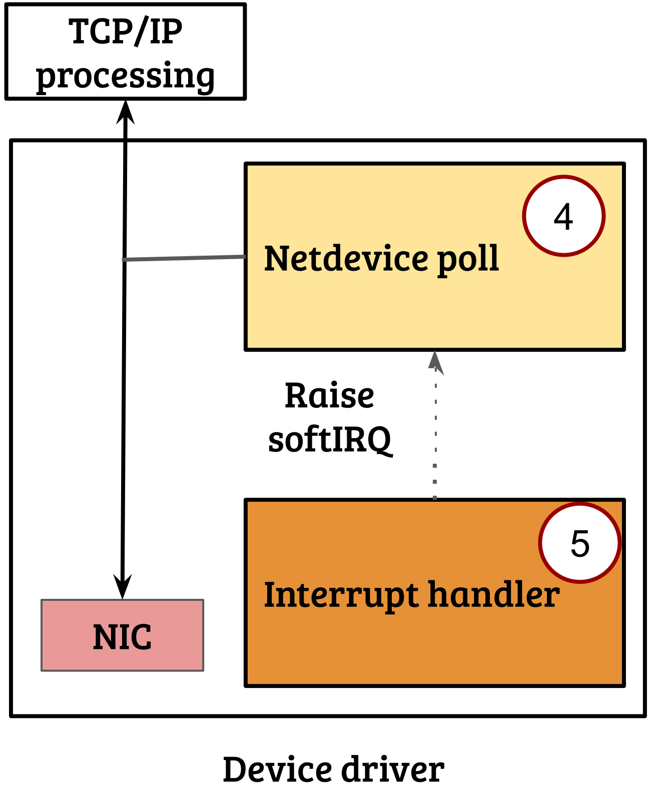
Receive livelock [1]



- Root cause: Interrupts have the highest priority over other contexts
- If the rate of interrupts is high, the system remains in interrupt handling mode, resulting in *receive livelock*
- Solution approach: Lower the priority of interrupts under heavy load
- How?

[1] https://www.usenix.org/legacy/publications/library/proceedings/sd96/mogul.html

NAPI: Interrupt + Polling



- Interrupt handler raises softIRQ after disabling packet receive interrupts
- Driver registered poll method is invoked
  - Executes till receive queue is empty or an upper threshold (budget)
  - Enable the interrupt (if queue is empty) and return
- Advantages
  - Low network load, more interrupt driven
  - High load, less interrupt processing
  - Avoid wasted work, drop packets early (in the device buffer)

Context related helper routines

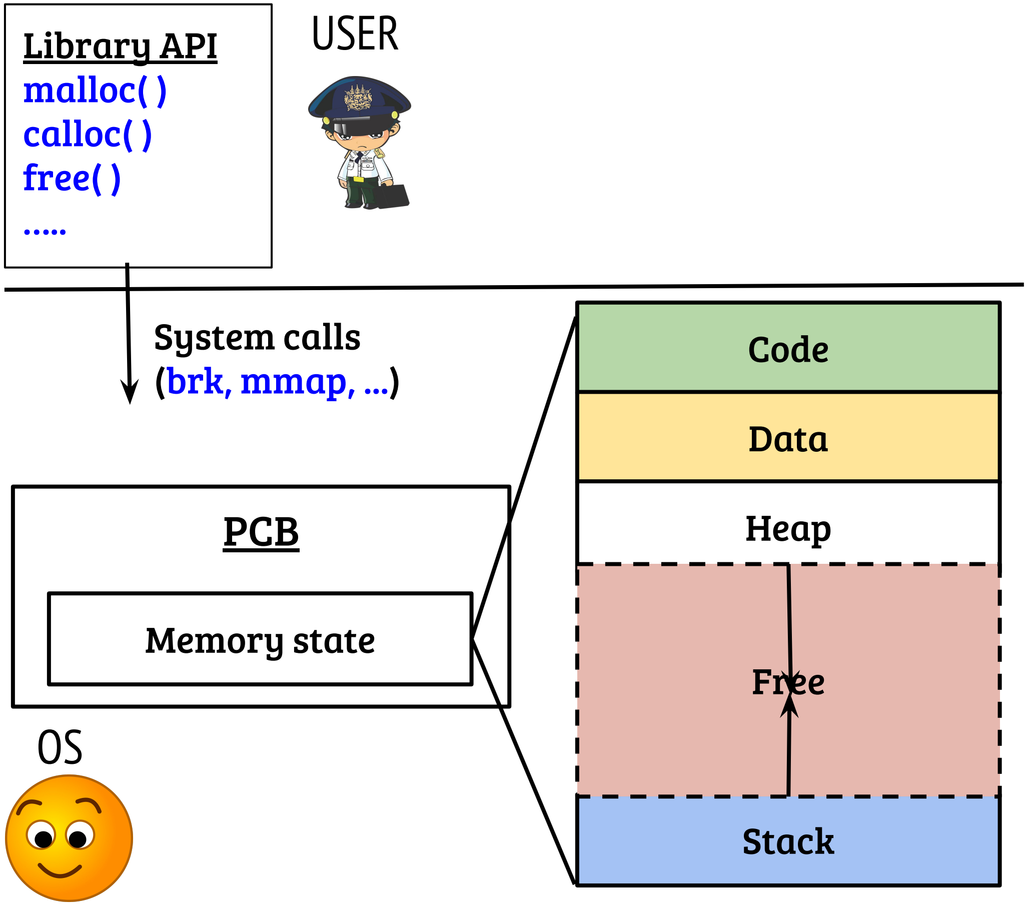
- bool in_irq( )
  - True if the current execution is in hardware interrupt
- bool in_softirq( )
  - True if the current execution is in a softIRQ or it is disabled
- bool in_interrupt( )
  - True if we are in NMI, IRQ, softIRQ context or have softIRQs disabled
- bool in_task( )
  - True if executing in a task context, *current* is valid
- Disabling/enabling interrupts
  - local_irq_disable/enable( )
- Disabling/enabling softIRQs
  - local_bh_disable/enable( )

CS614: Linux Kernel Programming

Virtual Memory

Debadatta Mishra, CSE, IIT Kanpur

User API for memory management



Library API

malloc( )
calloc( )
free( )
.....

USER

System calls
(brk, mmap, ...)

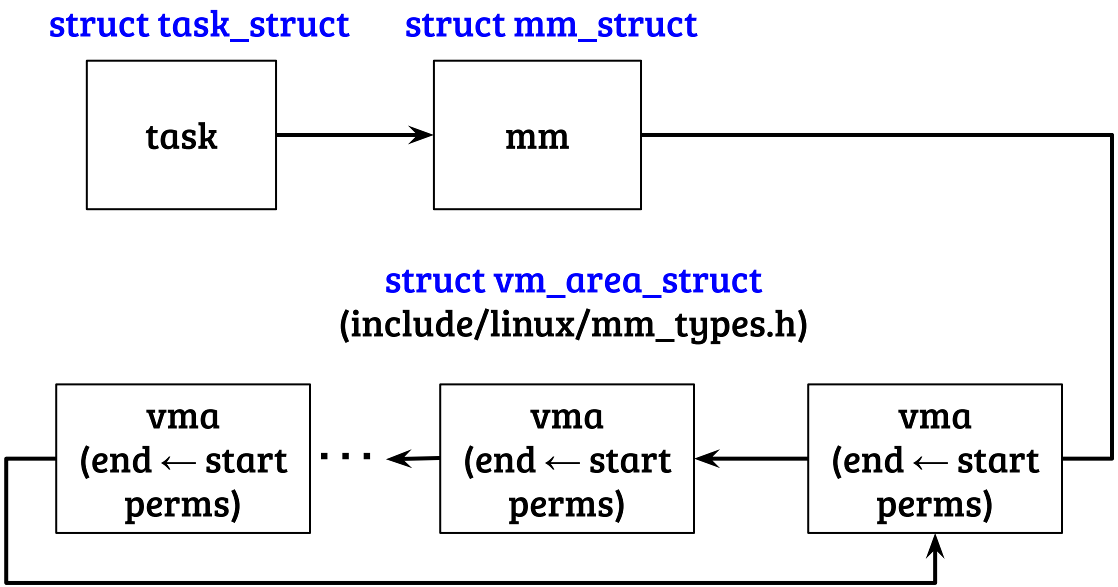
PCB

Memory state

OS

- Generally, user programs use library routines to allocate/deallocate memory
- OS provides some address space manipulation system calls (today’s agenda)

Virtual memory management



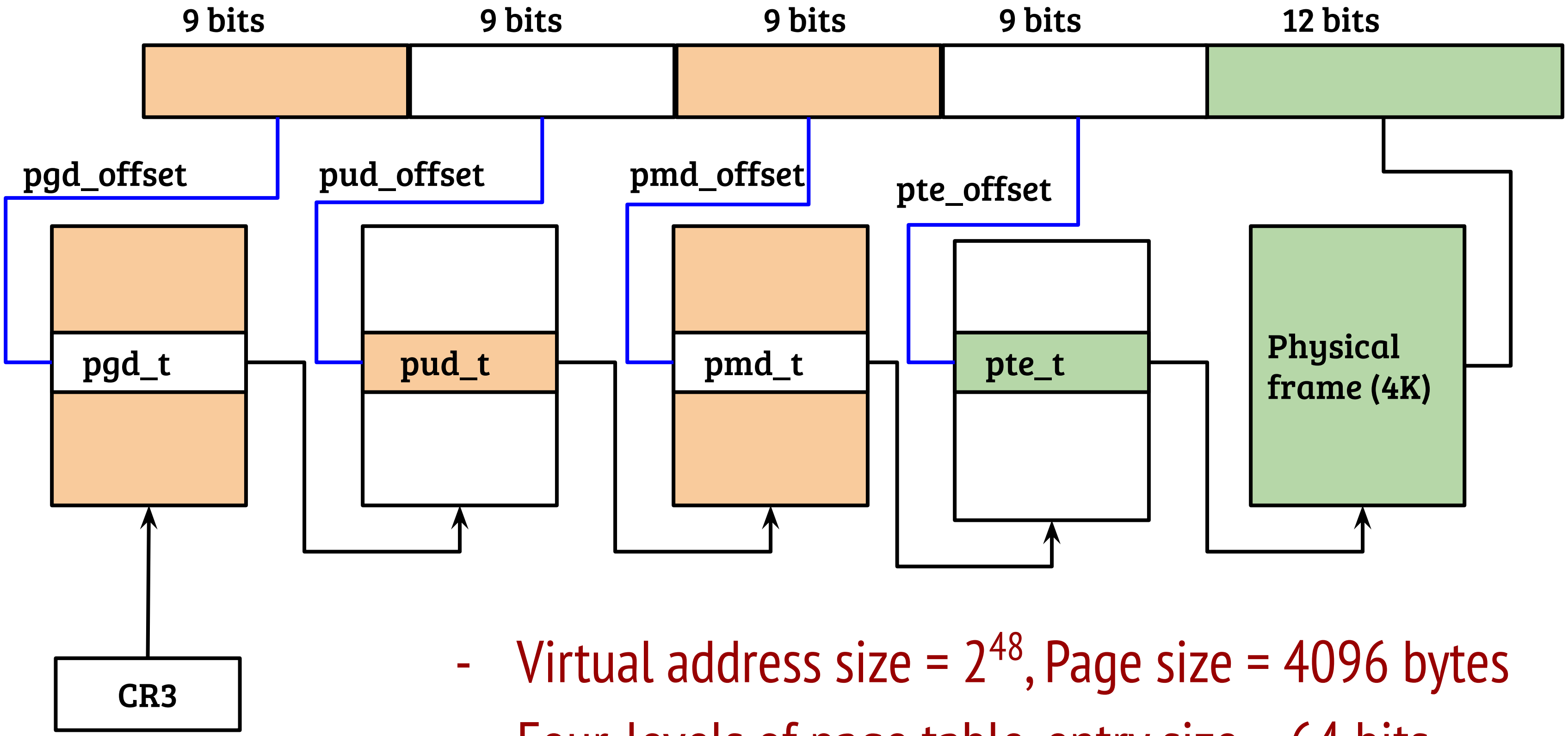
- start and end never overlaps between two vm areas
- can merge/extend vmas if permissions match
- linux maintains both rb_tree and a sorted list (see mm/filemap.c)

The OS implements VM system calls like mmap( ), mprotect( ) by manipulating the VMAs

Address translation: Paging

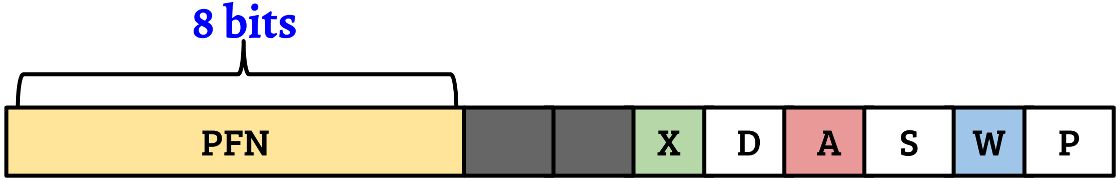
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = 4096 bytes
- Four-levels of page table, entry size = 64 bits

Paging example (structure of an example PTE)



- PFN occupies a significant portion of PTE entry (8 bits in this example)

P Present bit, 1 ⇒ entry is valid

W Write bit, 1 ⇒ Write allowed

S Privilege bit, 0 ⇒ only kernel mode access is allowed

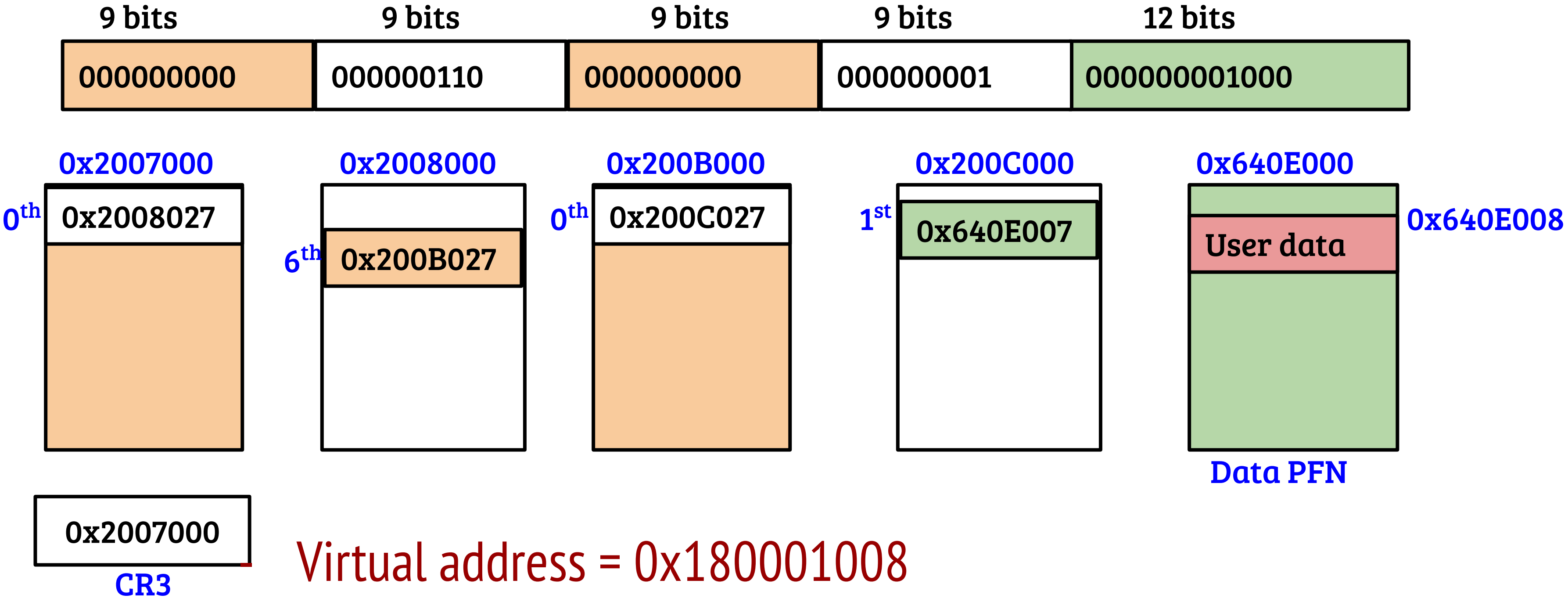
A Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)

D Dirty bit, 1 ⇒ Address written (set by H/W during walk)

X Execute bit, 1 ⇒ Instruction fetch allowed for this page

■ Reserved/unused bits

4-level page tables: example translation



- Virtual address = 0x180001008
- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
      sum += ctr;
```

```
0x20100:   mov $0, %rax;
0x20102:   mov %rax, (%rbp);      // sum=0
0x20104:   mov $0, %rcx;          // ctr=0
0x20106:   cmp $10, %rcx;         // ctr < 10
0x20109:   jge  0x2011f;          // jump if >=
0x2010f:   add %rcx, %rax;
0x20111:   mov %rax, (%rbp);      // sum += ctr
0x20113:   inc %rcx               // ++ctr
0x20115:   jmp 0x20106            // loop
0x2011f:   ………….
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:   mov $0, %rax;
0x20102:   mov %rax, (%rbp);      // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
        PageAddress P = V >> 12;
        TLBEntry entry = lookup(P);
        if (entry.valid) return entry.pte;
        entry = PageTableWalk(V);
        MakeEntry(entry);
        return entry.pte;
}
```

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

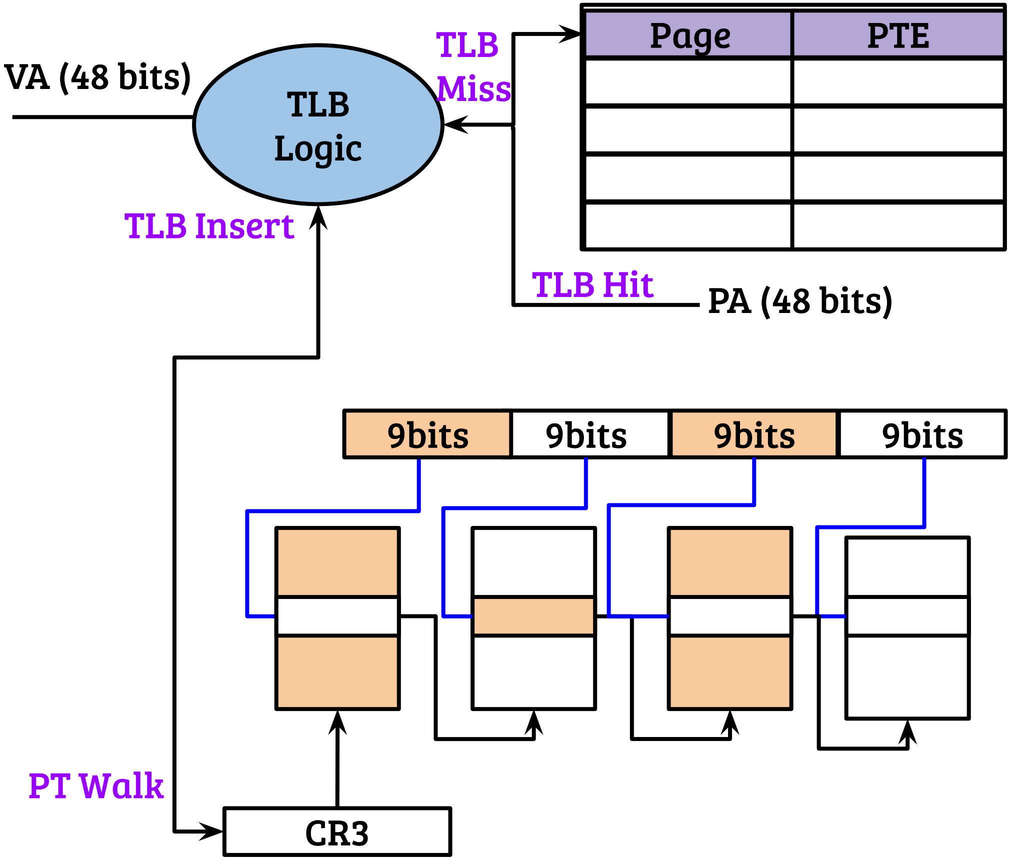
Paging: translation efficiency

```
0x20100:   mov $0, %rax;
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
    - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
    - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

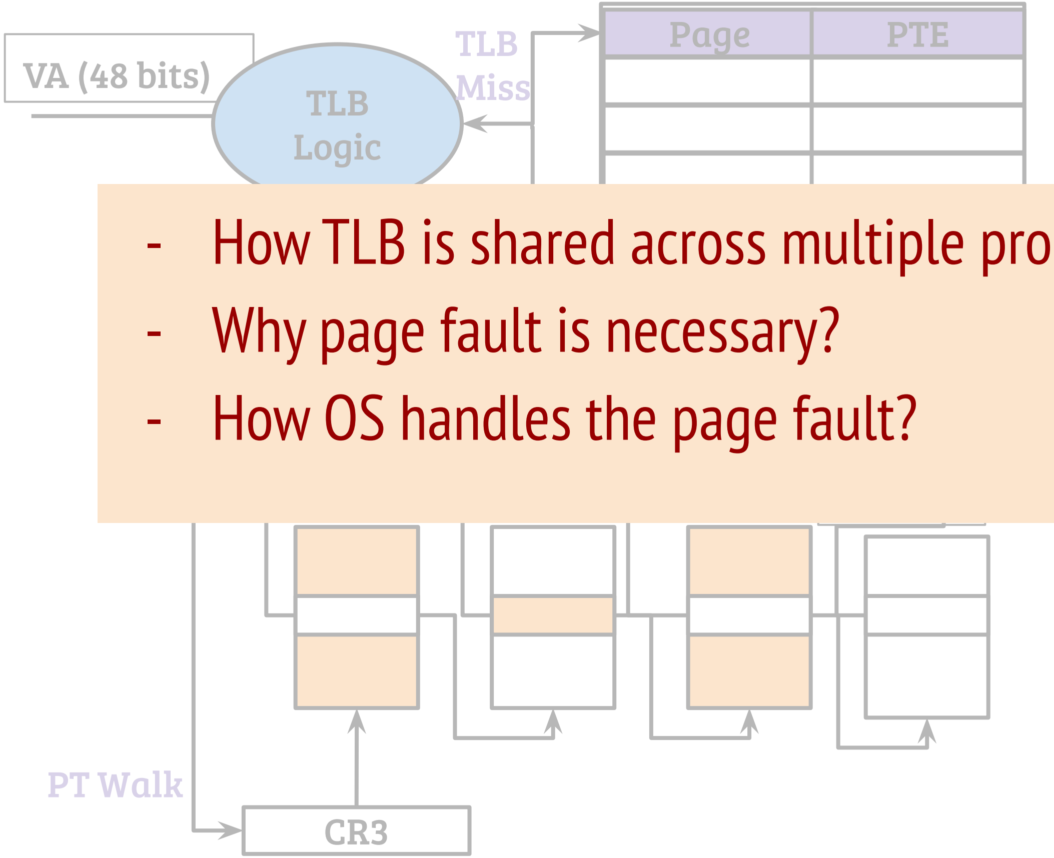
required (for translation) during the execution of the above code?

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



- TLB in the path of address

d

into the TLB directly, it can flush entries

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?

TLB: Sharing across applications

Process (A)

Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution

TLB: Sharing across applications

Process (A)

Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications

Process (A)

Process (B)

Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

Process (A)

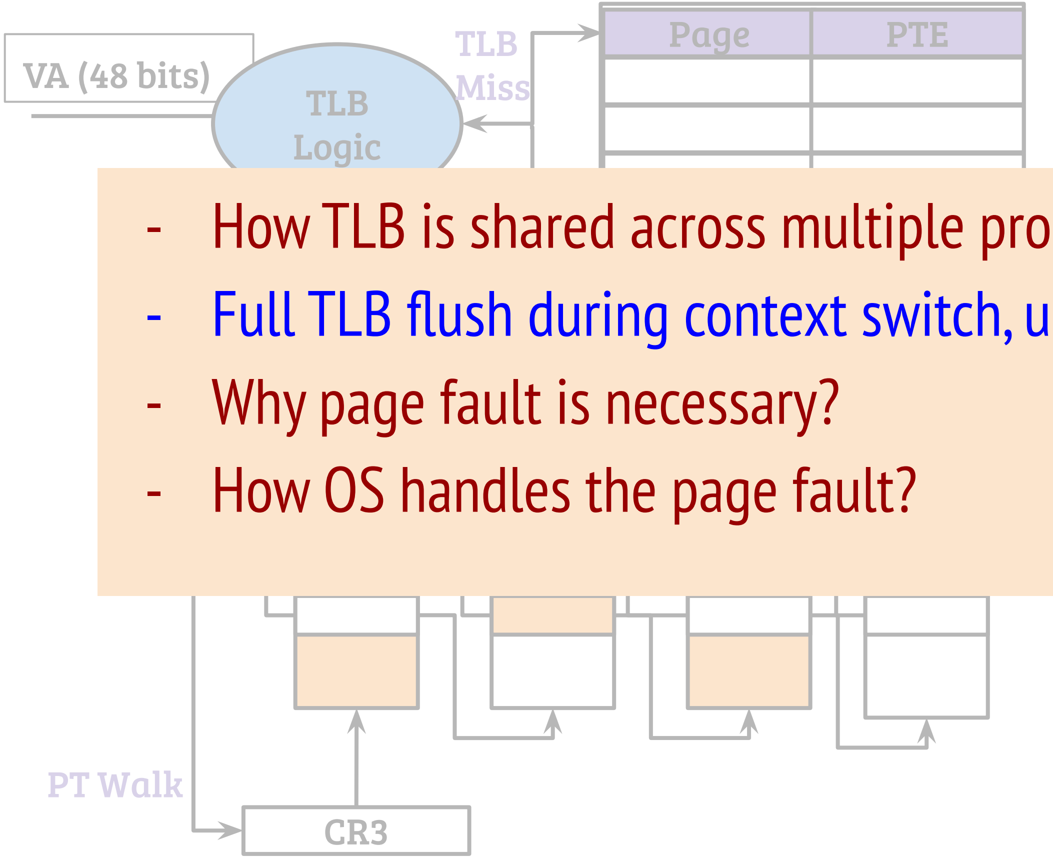
Process (B)

ASID	Page	PTE
A	0x100	ox200007
A	0x101	ox205007
B	0x100	0x301007
B	0x101	0x302007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)



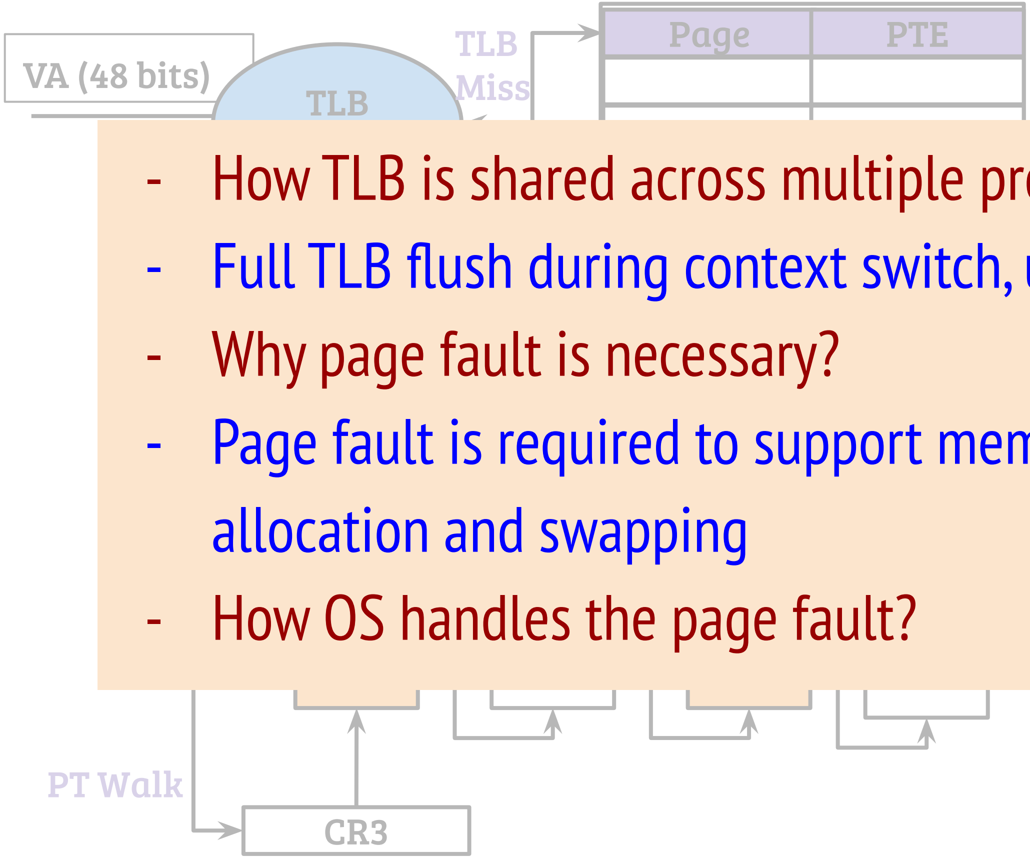
- TLB in the path of address

d

entries

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

Address translation (TLB + PTW)



- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

Page fault handling in X86: Hardware

```
If(  !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
         Raise pageFault;
} // Simplified
```

Page fault handling in X86: Hardware

```
If(  !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                    | access << 1
                    | cpl << 2;
        Raise pageFault;
} // Simplified
```

Error code

Other and unused	I	R	U	W	P

P **Present bit, 1 ⇒ fault is due to protection**

W **Write bit, 1 ⇒ Access is write**

U **Privilege bit, 1 ⇒ Access is from user mode**

R **Reserved bit, 1 ⇒ Reserved bit violation**

I **Fetch bit, 1 ⇒ Access is Instruction Fetch**

- Error code is pushed into the kernel stack by the hardware

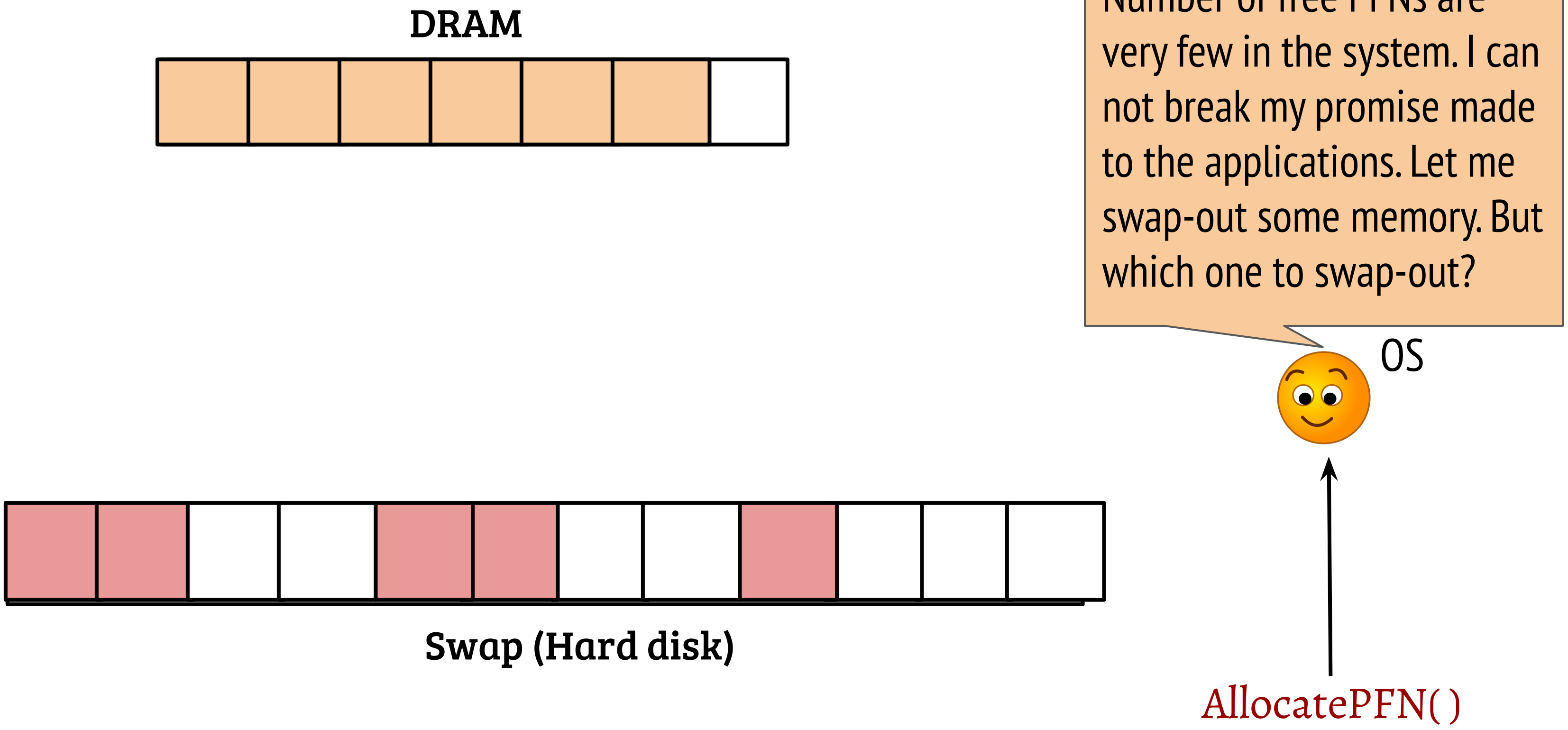
Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Address translation (TLB + PTW)

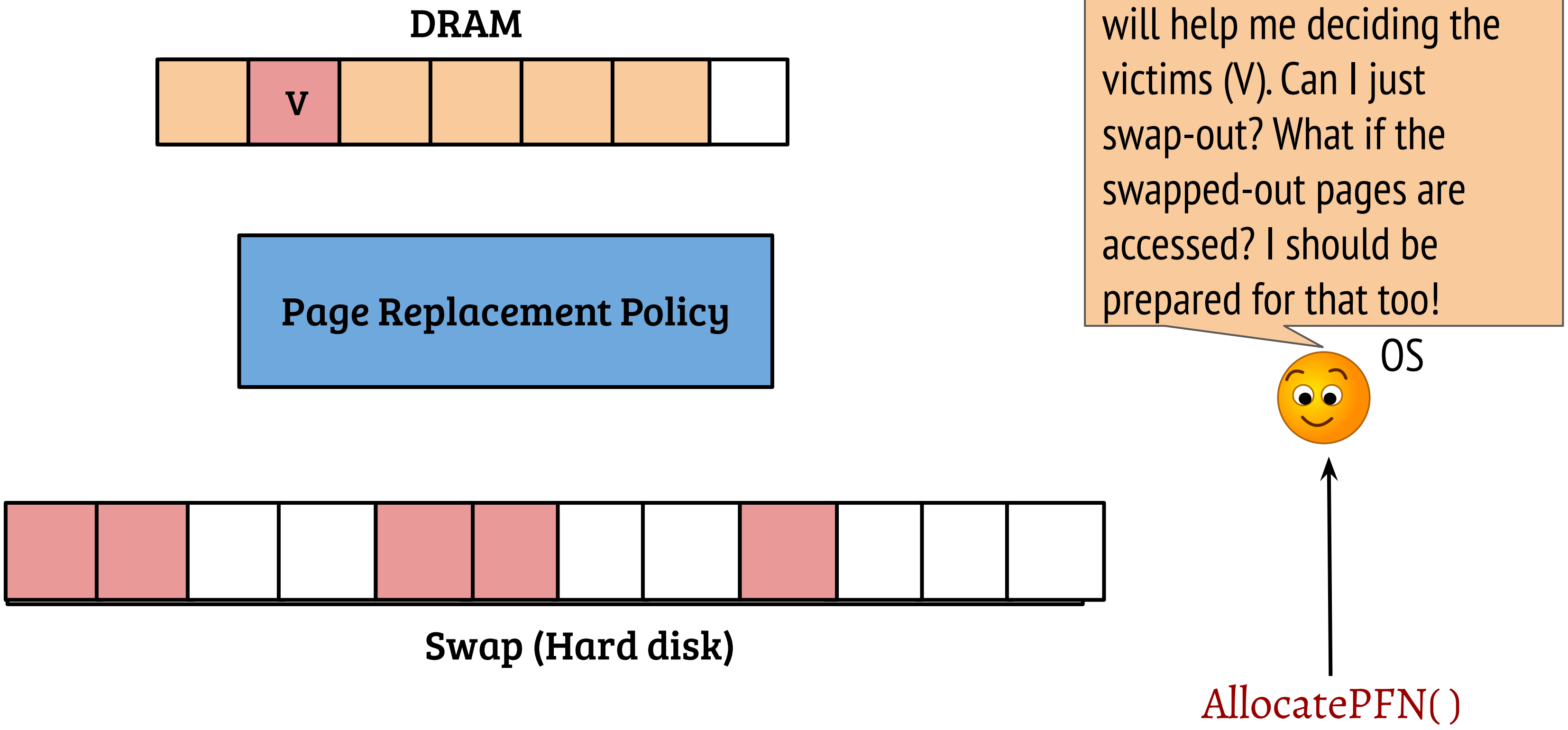
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

Swapping (swap-out)

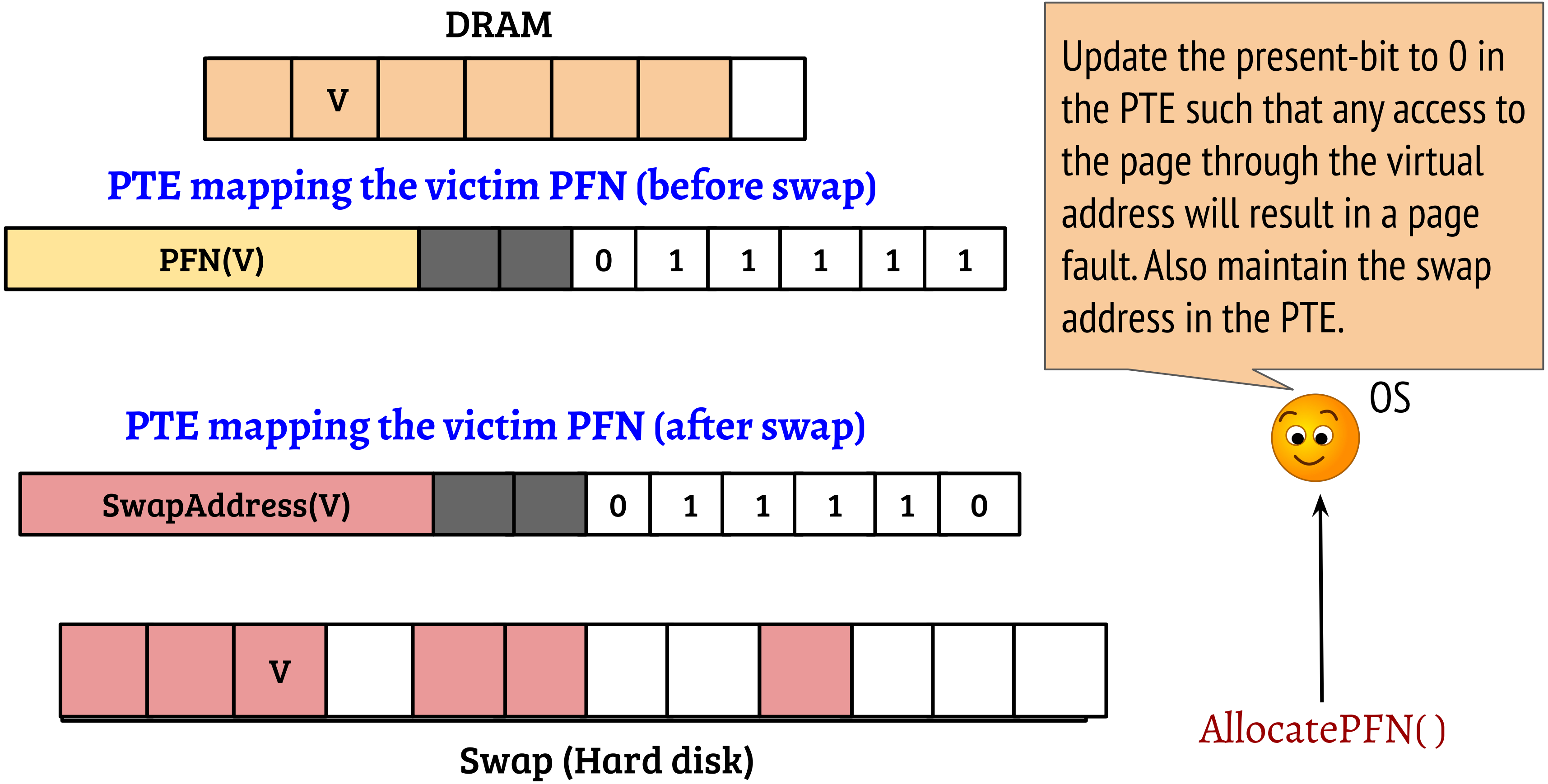


Number of free PFNs are very few in the system. I can not break my promise made to the applications. Let me swap-out some memory. But which one to swap-out?

Swapping (swap-out)

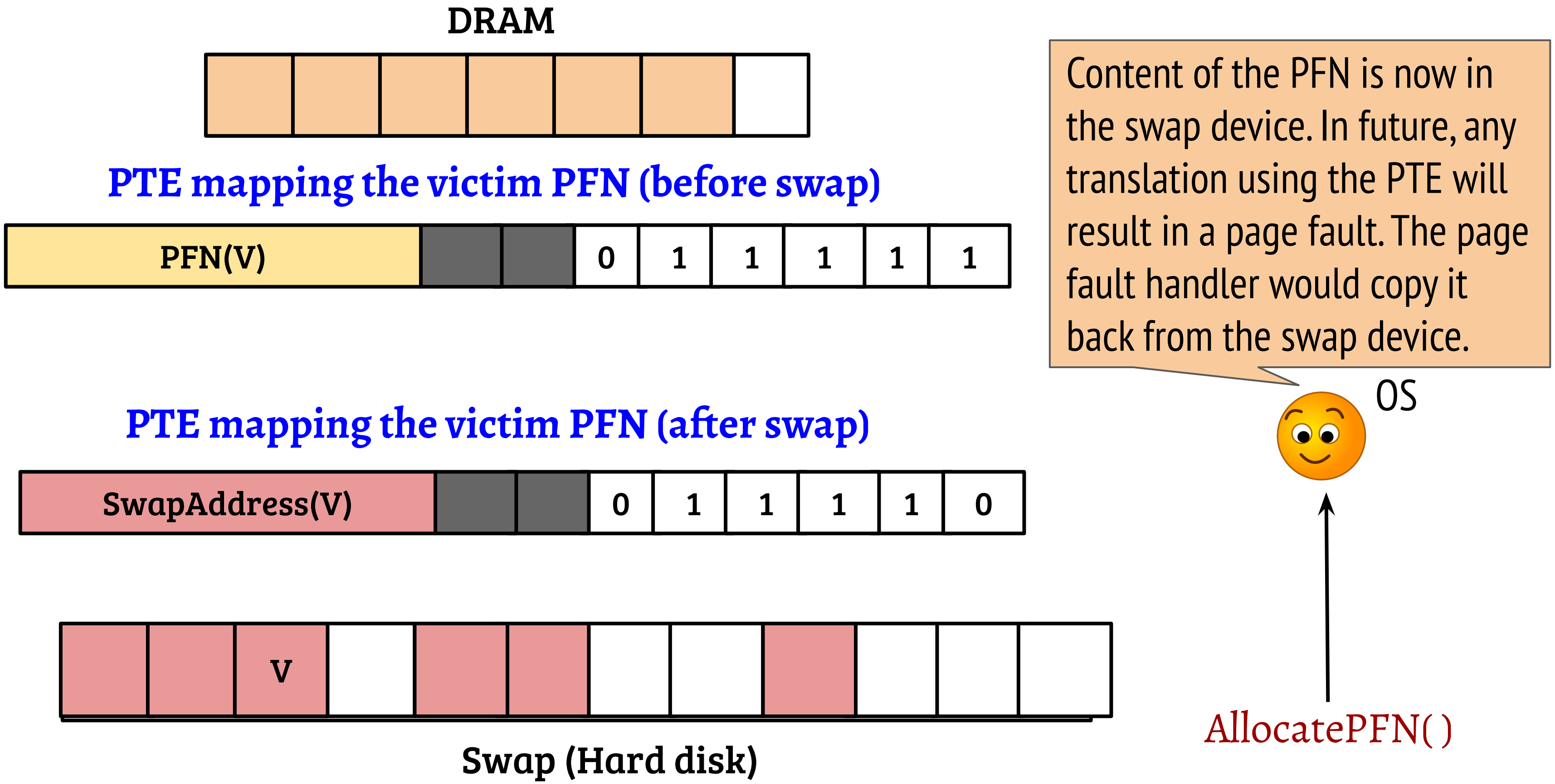

DRAM
V
Page Replacement Policy
Swap (Hard disk)
My page replacement policy will help me deciding the victims (V). Can I just swap-out? What if the swapped-out pages are accessed? I should be prepared for that too!
OS
AllocatePFN( )

Swapping (swap-out)


DRAM
V
PTE mapping the victim PFN (before swap)
PFN(V)
0
1
1
1
1
1
PTE mapping the victim PFN (after swap)
SwapAddress(V)
0
1
1
1
1
0
V
Swap (Hard disk)
OS
AllocatePFN( )

Update the present-bit to 0 in the PTE such that any access to the page through the virtual address will result in a page fault. Also maintain the swap address in the PTE.

Swapping (swap-out)



DRAM

PTE mapping the victim PFN (before swap)

PFN(V)			0	1	1	1	1	1

PTE mapping the victim PFN (after swap)

SwapAddress(V)			0	1	1	1	1	0

		V									

Swap (Hard disk)

Content of the PFN is now in the swap device. In future, any translation using the PTE will result in a page fault. The page fault handler would copy it back from the swap device.

OS

AllocatePFN( )

Page fault with swap-in

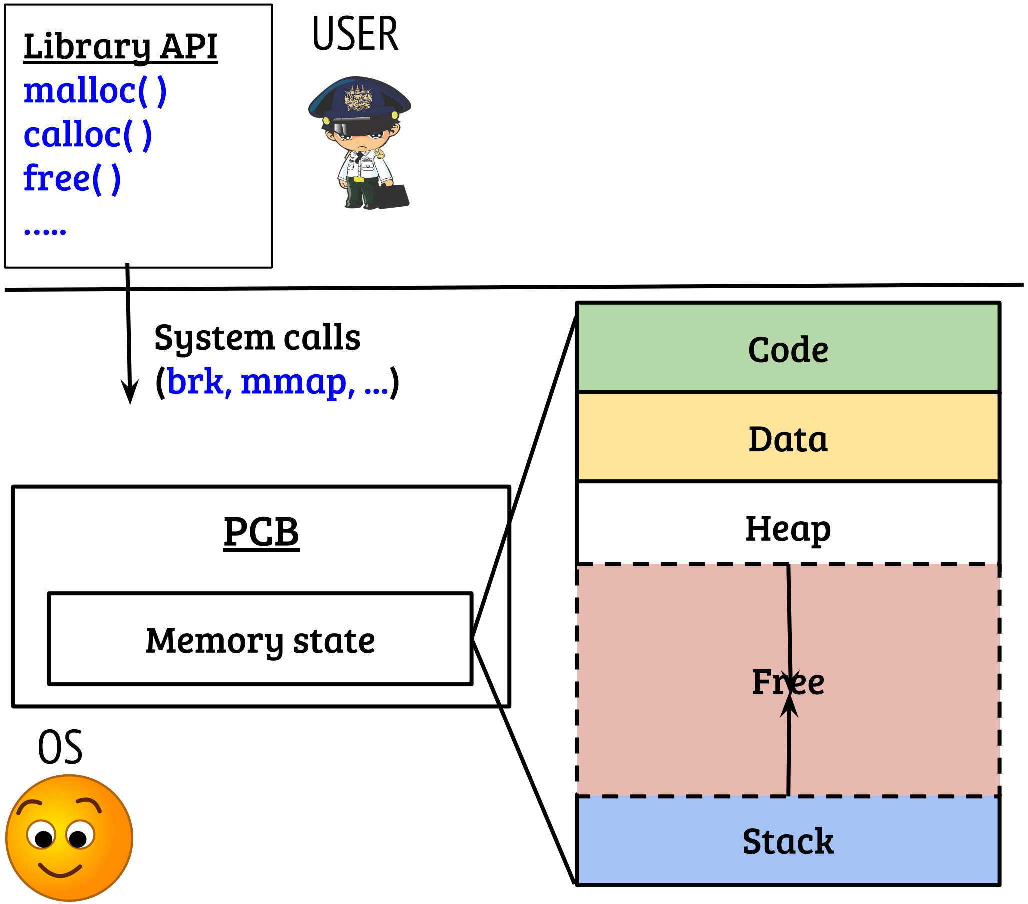
```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
              PFN = allocate_pfn( );
              If ( is_swapped_pte(address) )       // Check if the PTE is swapped out
                swapin(getPTE(address), PFN);   // Copy the swap block  to PFN
              install_pte(address, PFN);           // and update the PTE
             return;
        }
    RaiseSignal(SIGSEGV);
}
```

CS614: Linux Kernel Programming

Virtual Memory

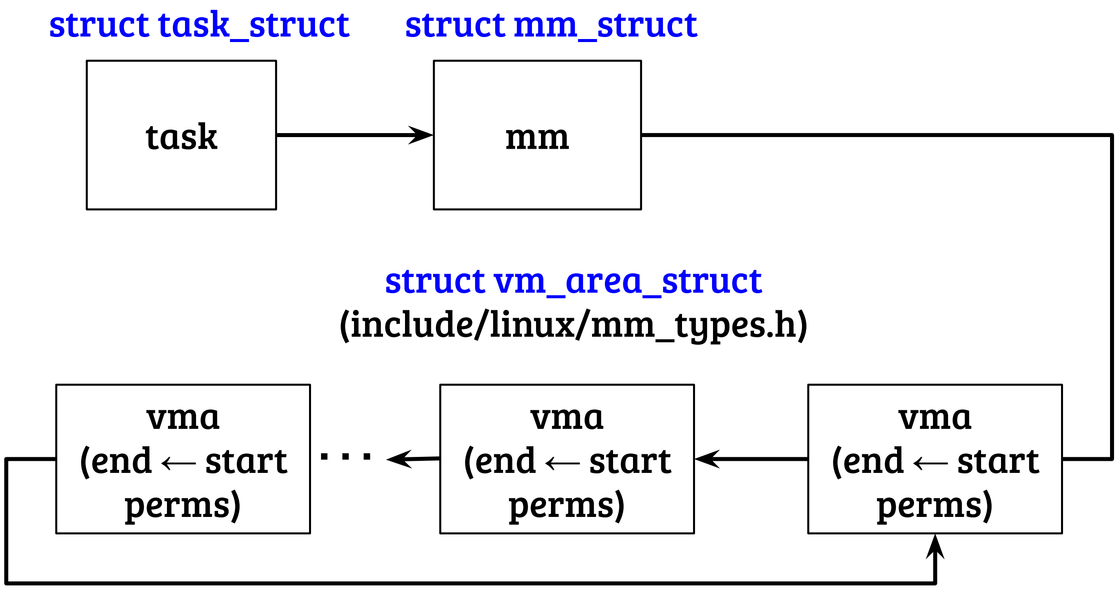
Debadatta Mishra, CSE, IIT Kanpur

User API for memory management



- Generally, user programs use library routines to allocate/deallocate memory
- OS provides some address space manipulation system calls (today's agenda)

Virtual memory management



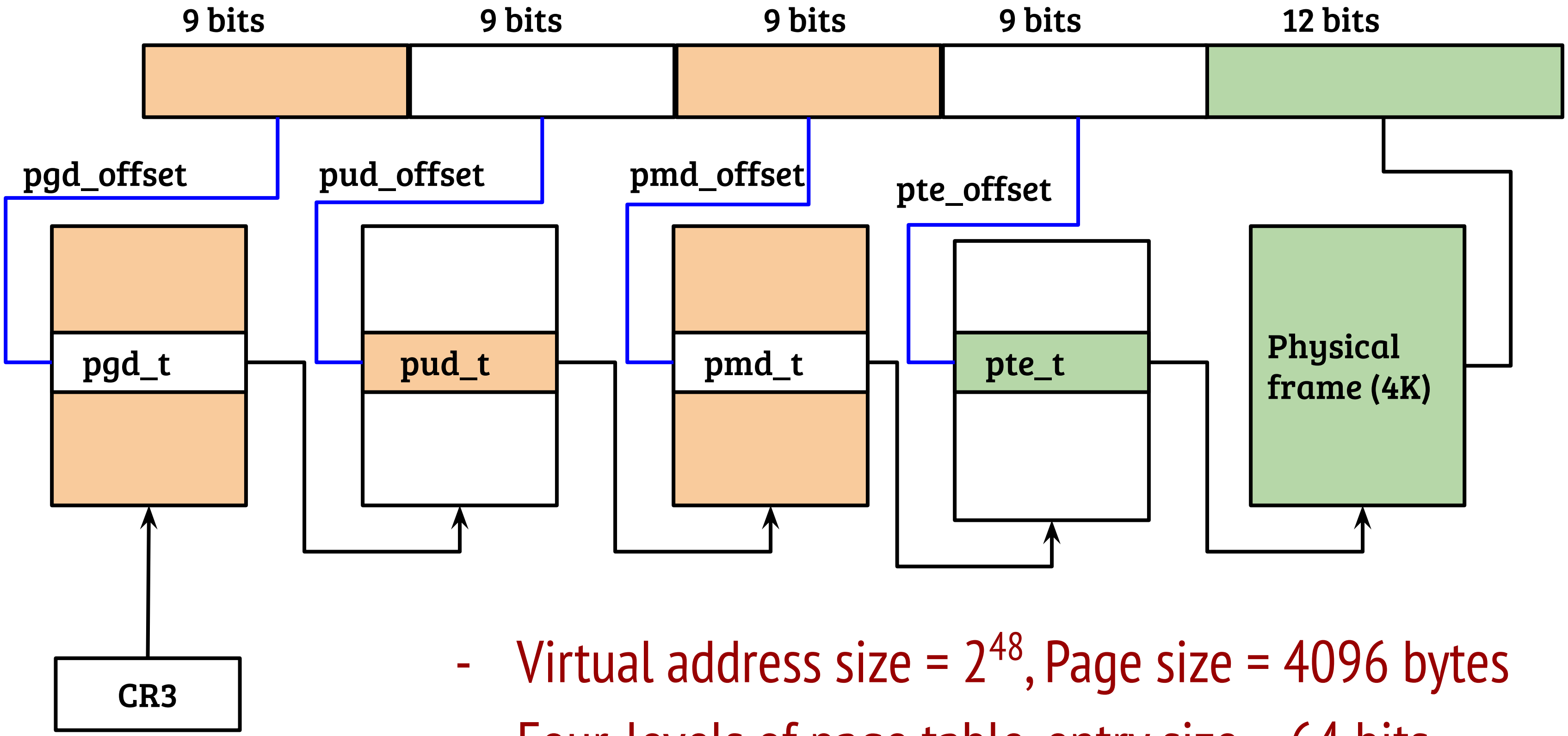
- start and end never overlaps between two vm areas
- can merge/extend vmas if permissions match
- linux maintains both rb_tree and a sorted list (see mm/filemap.c)

The OS implements VM system calls like mmap( ), mprotect( ) by manipulating the VMAs

Address translation: Paging

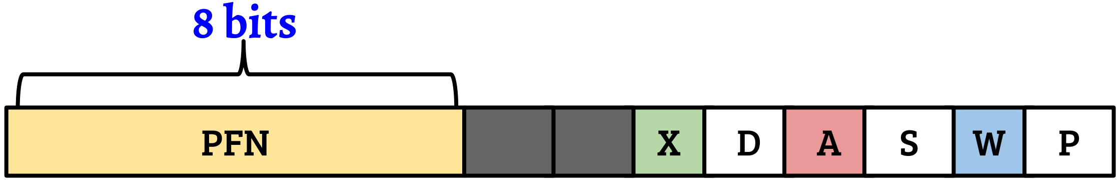
- The idea of paging
  - Partition the address space into fixed sized blocks (call it pages)
  - Physical memory partitioned in a similar way (call it page frames)
  - OS creates a mapping between *page* to *page frame* , H/W uses the mapping to translate VA to PA
- With increased address space size, single level page table entry is not feasible, because
  - Increasing page size increases internal fragmentation
  - Small pages may not be suitable to hold all mapping entries

4-level page tables: 48-bit VA (Intel x86_64)



- Virtual address size = 2^{48}, Page size = 4096 bytes
- Four-levels of page table, entry size = 64 bits

Paging example (structure of an example PTE)



- PFN occupies a significant portion of PTE entry (8 bits in this example)

P Present bit, 1 ⇒ entry is valid

W Write bit, 1 ⇒ Write allowed

S Privilege bit, 0 ⇒ only kernel mode access is allowed

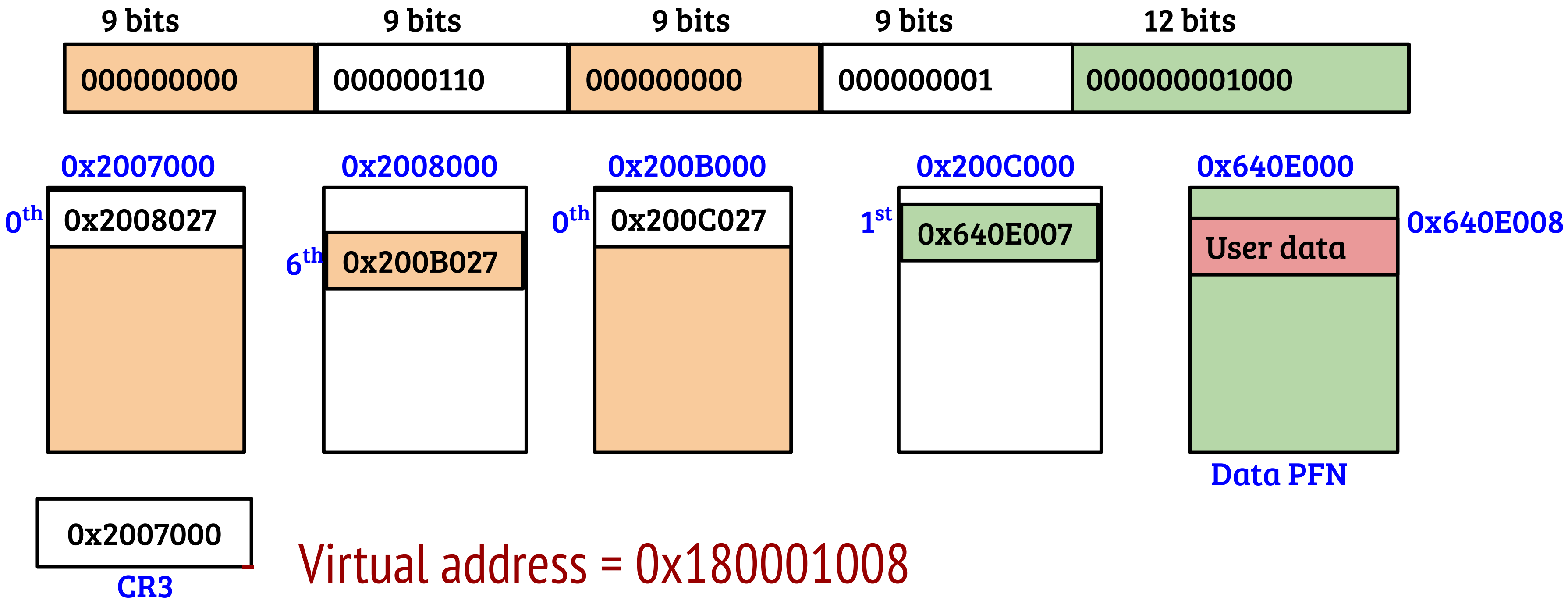
A Accessed bit, 1 ⇒ Address accessed (set by H/W during walk)

D Dirty bit, 1 ⇒ Address written (set by H/W during walk)

X Execute bit, 1 ⇒ Instruction fetch allowed for this page

■ Reserved/unused bits

4-level page tables: example translation



9 bits | 9 bits | 9 bits | 9 bits | 12 bits

000000000 | 000000110 | 000000000 | 000000001 | 000000001000

0x2007000

0th 0x2008027

0x2008000

6th 0x200B027

0x200B000

0th 0x200C027

0x200C000

1st 0x640E007

0x640E000

User data 0x640E008

Data PFN

0x2007000

CR3

- Virtual address = 0x180001008
- Hardware translation by repeated access of page table stored in physical memory
- Page table entry: 12 bits LSB is used for access flags

Paging: translation efficiency

```
sum = 0;
for(ctr=0; ctr<10; ++ctr)
    sum += ctr;
```

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp);     // sum=0
0x20104:  mov $0, %rcx;         // ctr=0
0x20106:  cmp $10, %rcx;        // ctr < 10
0x20109:  jge  0x2011f;         // jump if >=
0x2010f:  add %rcx, %rax;
0x20111:  mov %rax, (%rbp);     // sum += ctr
0x20113:  inc %rcx              // ++ctr
0x20115:  jmp 0x20106           // loop
0x2011f:  .............
```

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging: translation efficiency

```
0x20100:  mov $0, %rax;
0x20102:  mov %rax, (%rbp);      // sum=0
```

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?

- Considering four-level page table, how many memory accesses are required (for translation) during the execution of the above code?

Paging with TLB: translation efficiency

TLB

Page	PTE
0x20	0x750
0x7FFF	0x890

```
Translate(V){
        PageAddress P = V >> 12;
        TLBEntry entry = lookup(P);
        if (entry.valid) return entry.pte;
        entry = PageTableWalk(V);
        MakeEntry(entry);
        return entry.pte;
}
```

- TLB is a hardware cache which stores *Page* to *PFN* mapping
- After first miss for instruction fetch address, all others result in a TLB hit
- Similarly, considering the stack virtual address range as 0x7FFF000 - 0x8000000, one entry in TLB avoids page table walk after first miss

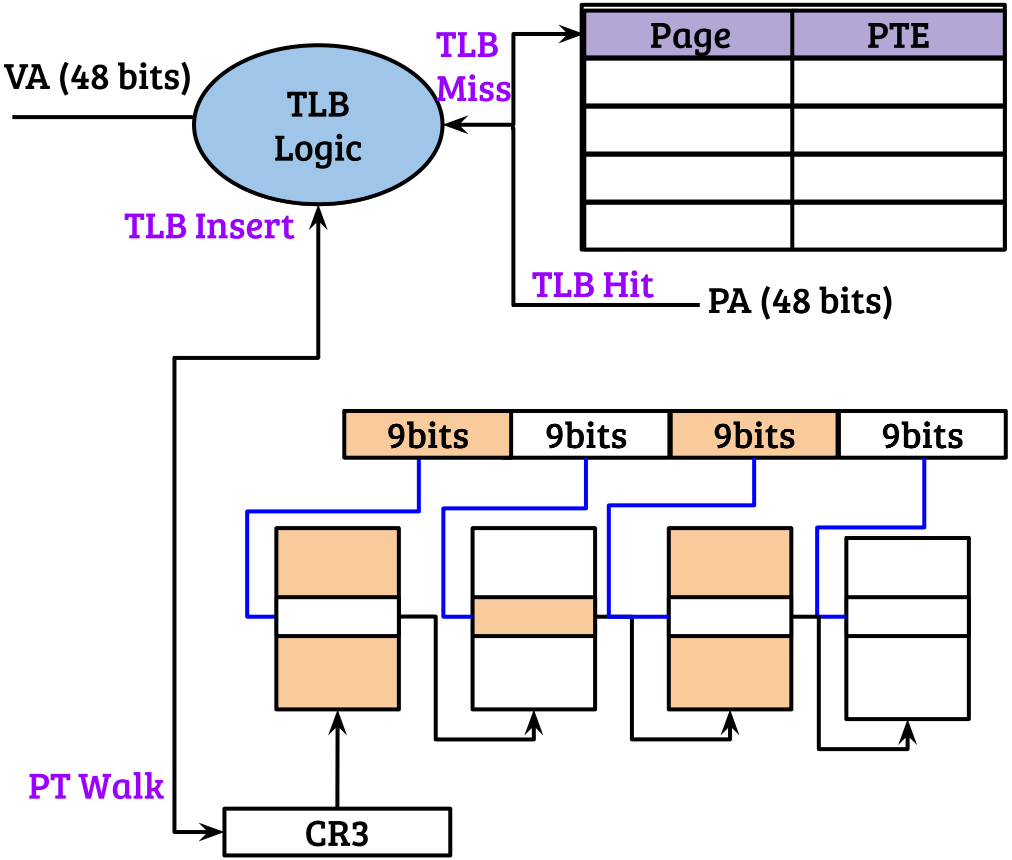
Paging: translation efficiency

0x20100: mov $0, %rax;

- Instruction execution: Loop = 10 * 6, Others = 2 + 3
  - Memory accesses during translation = 65 * 4 = 260
- Data/stack access: Initialization = 1, Loop = 10
  - Memory accesses during translation = 11 * 4 = 44
- A lot of memory accesses (> 300) for address translation
- How many distinct pages are translated?
- One code page (0x20) and one stack page (0x7FFF). Caching these translations, will save a lot of memory accesses.

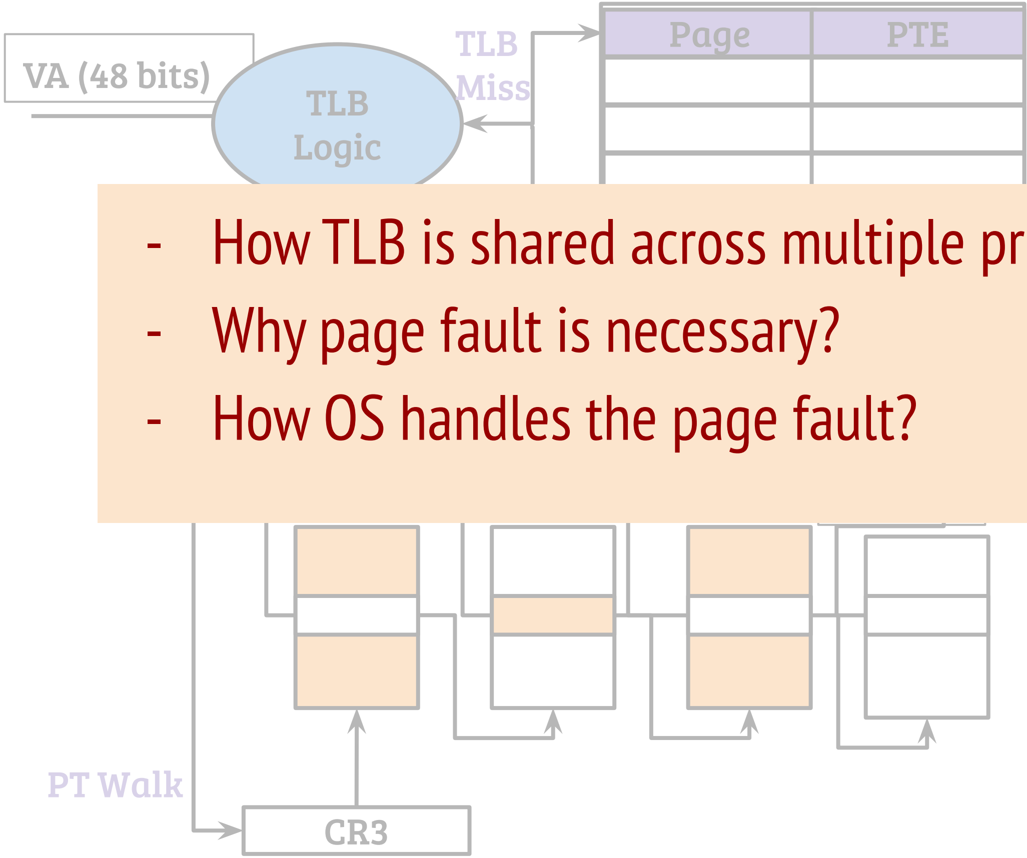
required (for translation) during the execution of the above code?

Address translation (TLB + PTW)



- TLB in the path of address translation
- Separate TLBs for instruction and data, multi-level TLBs
- In X86, OS can not make entries into the TLB directly, it can flush entries

Address translation (TLB + PTW)



- TLB in the path of address

d

into the TLB directly, it can flush entries

- How TLB is shared across multiple processes?
- Why page fault is necessary?
- How OS handles the page fault?

TLB: Sharing across applications

Process (A)

Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution

TLB: Sharing across applications

Process (A)

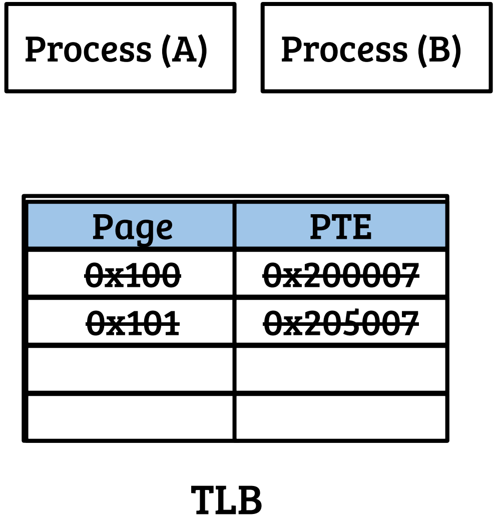
Process (B)

Page	PTE
0x100	0x200007
0x101	0x205007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Process B may be using the same addresses used by A. Result: Wrong translation

TLB: Sharing across applications



Process (A)

Process (B)

Page	PTE
~~0x100~~	~~0x200007~~
~~0x101~~	~~0x205007~~

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Correctness ensured. Performance is an issue (with frequent context switching)

TLB: Sharing across applications

Process (A)

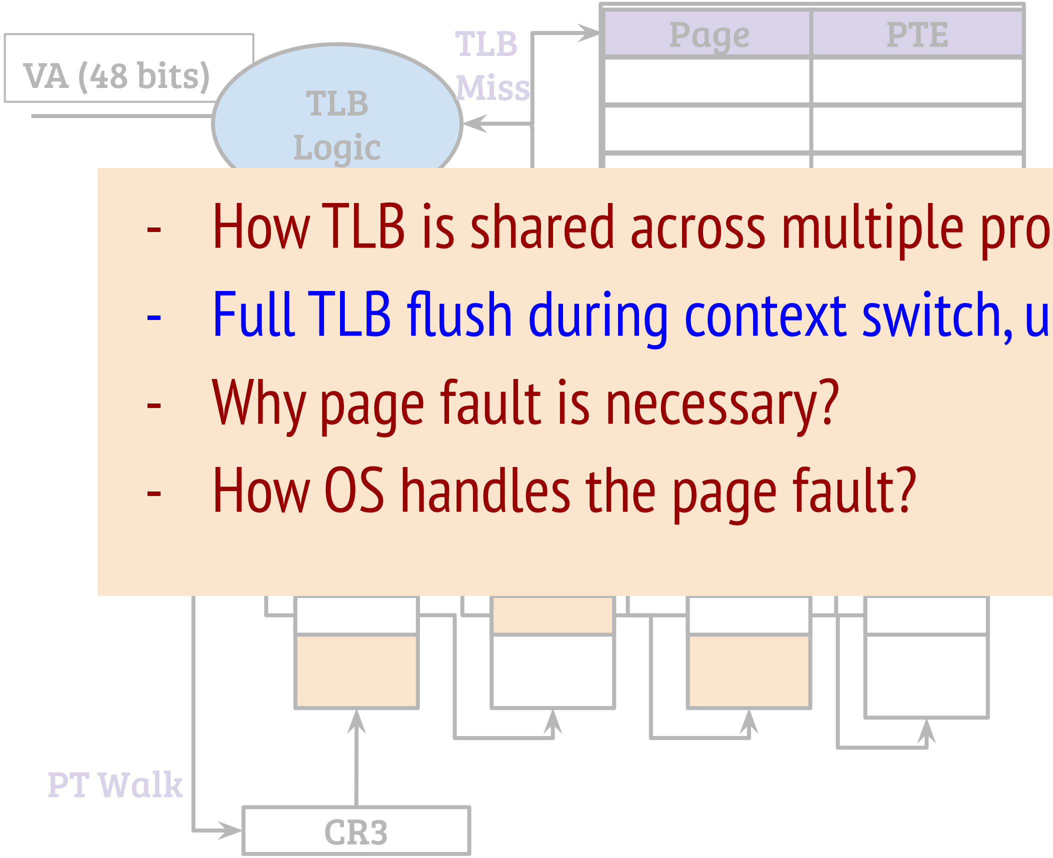
Process (B)

ASID	Page	PTE
A	0x100	ox200007
A	0x101	ox205007
B	0x100	0x301007
B	0x101	0x302007

TLB

- Assume that, process A is currently executing. What happens when process B is scheduled?
  - A) Do nothing
  - B) Flush the whole TLB
  - C) Some other solution
- Address space identified (ASID) along with each TLB entry to identify the process

Address translation (TLB + PTW)



- TLB in the path of address

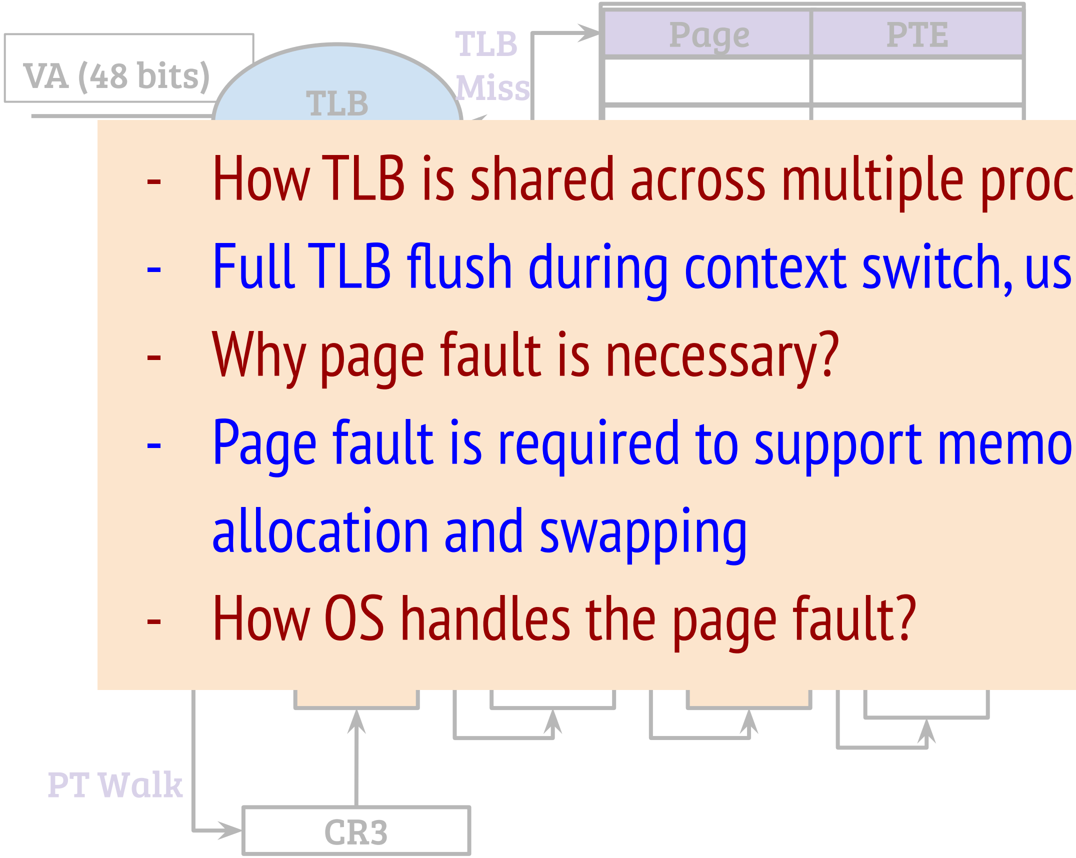
d

y,

entries

- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- How OS handles the page fault?

Address translation (TLB + PTW)



- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?

Page fault handling in X86: Hardware

```
If(  !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
          CR2 = Address;
          errorCode = pte.valid
                              | access << 1
                              | cpl << 2;
           Raise pageFault;
} // Simplified
```

Page fault handling in X86: Hardware

```
If(  !pte.valid ||
    (access == write && !pte.write) ||
    (cpl != 0 && pte.priv == 0)){
        CR2 = Address;
        errorCode = pte.valid
                        | access << 1
                        | cpl << 2;
        Raise pageFault;
} // Simplified
```

Error code

Other and unused	I	R	U	W	P

P	**Present bit, 1 ⇒ fault is due to protection**
W	**Write bit, 1 ⇒ Access is write**
U	**Privilege bit, 1 ⇒ Access is from user mode**
R	**Reserved bit, 1 ⇒ Reserved bit violation**
I	**Fetch bit, 1 ⇒ Access is Instruction Fetch**

- Error code is pushed into the kernel stack by the hardware

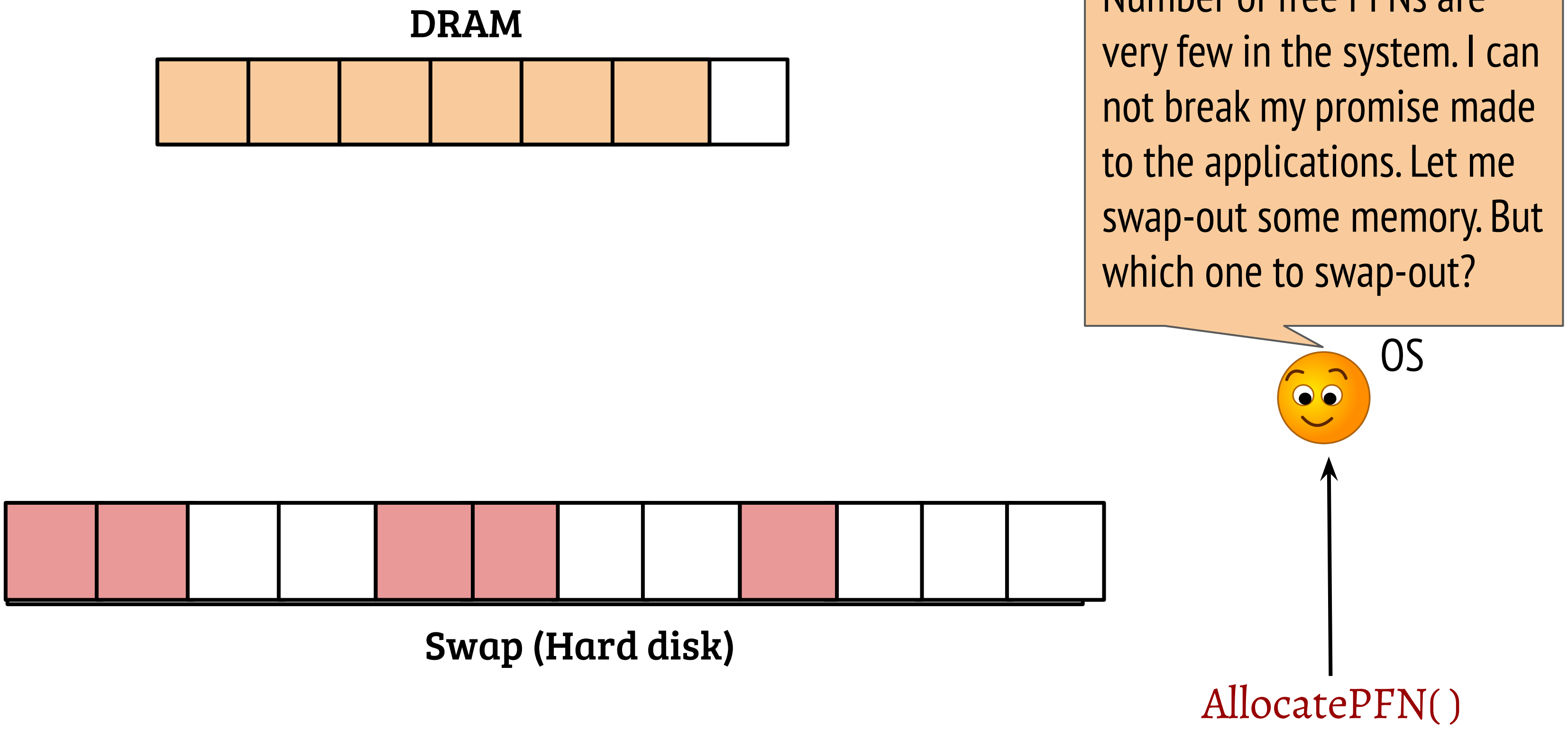
Page fault handling in X86: OS fault handler

```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            install_pte(address, PFN);
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Address translation (TLB + PTW)

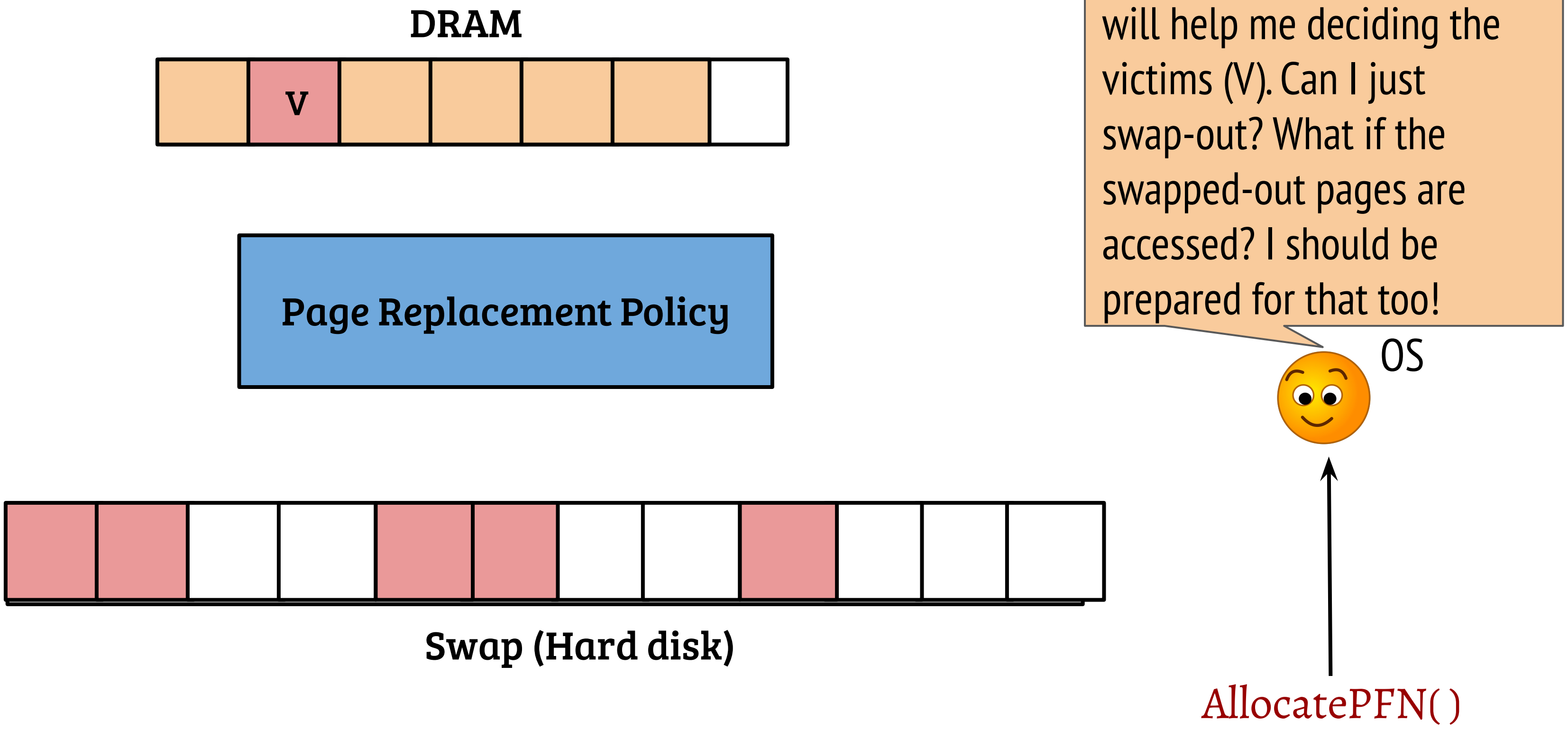
- How TLB is shared across multiple processes?
- Full TLB flush during context switch, using ASID
- Why page fault is necessary?
- Page fault is required to support memory over-commitment through lazy allocation and swapping
- How OS handles the page fault?
- The hardware invokes the page fault handler by placing the error code and virtual address. The OS handles the page fault either fixing it or raising a SEGFAULT.

Swapping (swap-out)



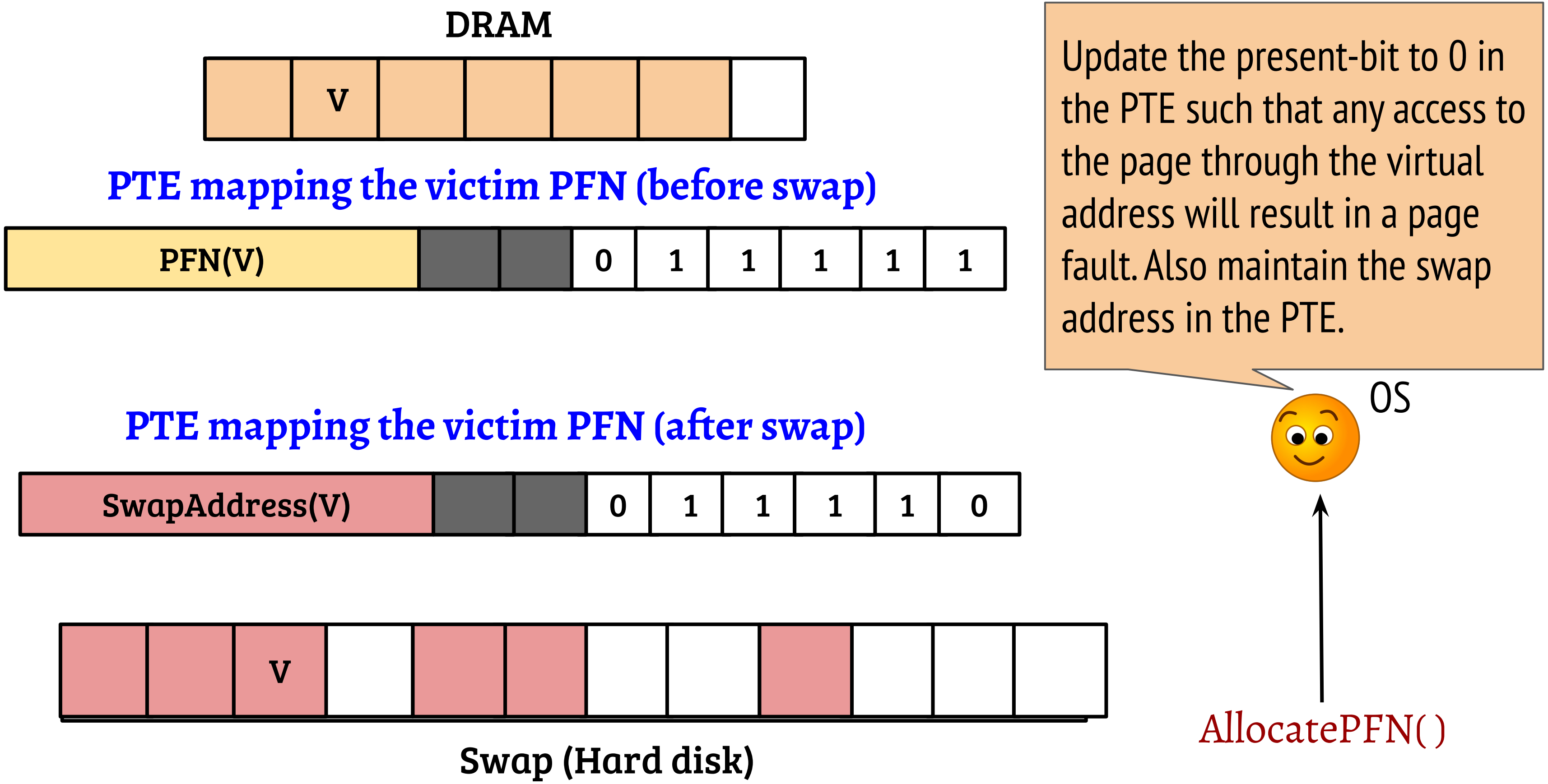
Number of free PFNs are very few in the system. I can not break my promise made to the applications. Let me swap-out some memory. But which one to swap-out?

Swapping (swap-out)



My page replacement policy will help me deciding the victims (V). Can I just swap-out? What if the swapped-out pages are accessed? I should be prepared for that too!

Swapping (swap-out)



PTE mapping the victim PFN (before swap)

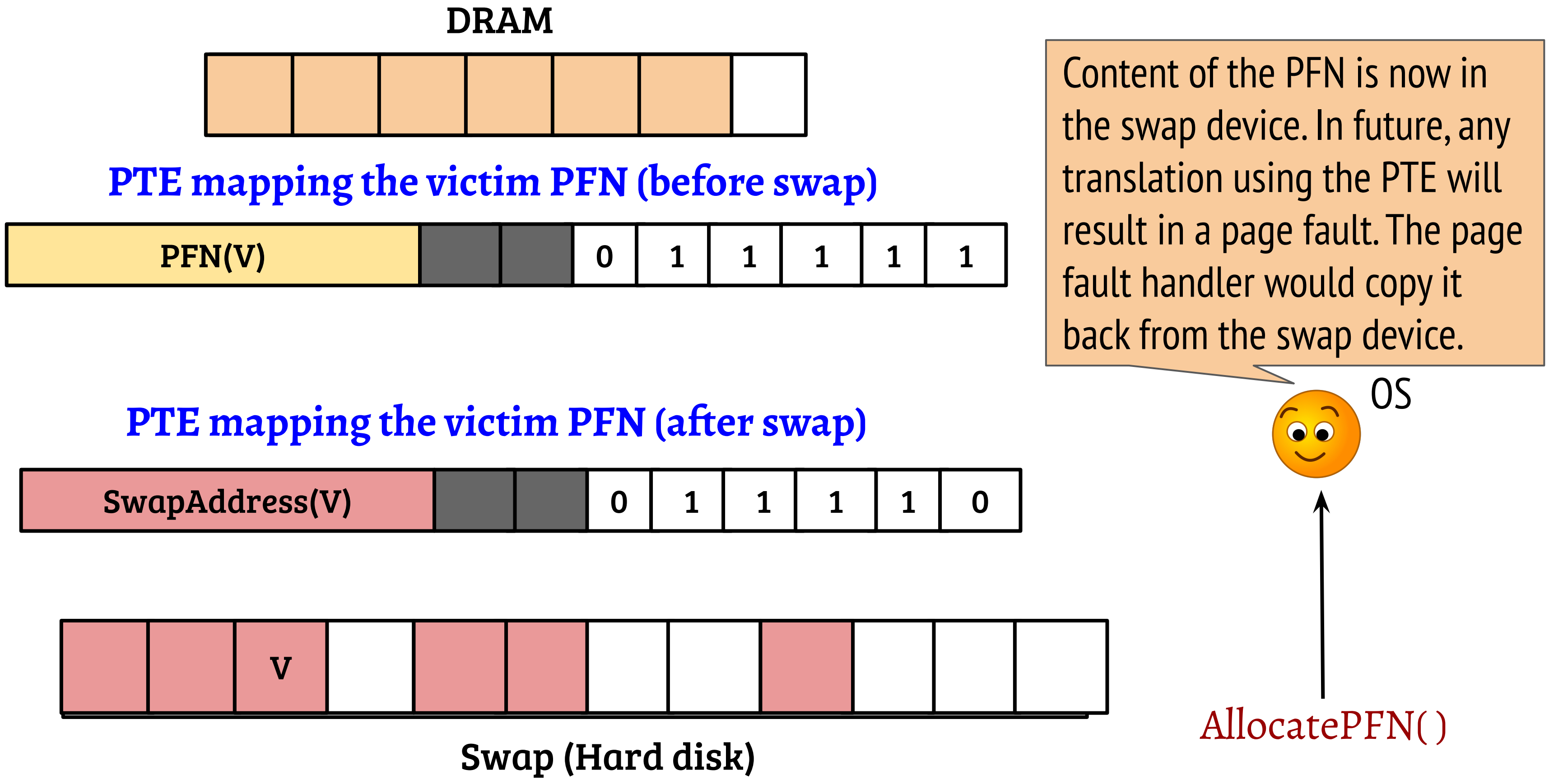
PFN(V)			0	1	1	1	1	1

PTE mapping the victim PFN (after swap)

SwapAddress(V)			0	1	1	1	1	0

Update the present-bit to 0 in the PTE such that any access to the page through the virtual address will result in a page fault. Also maintain the swap address in the PTE.

Swapping (swap-out)

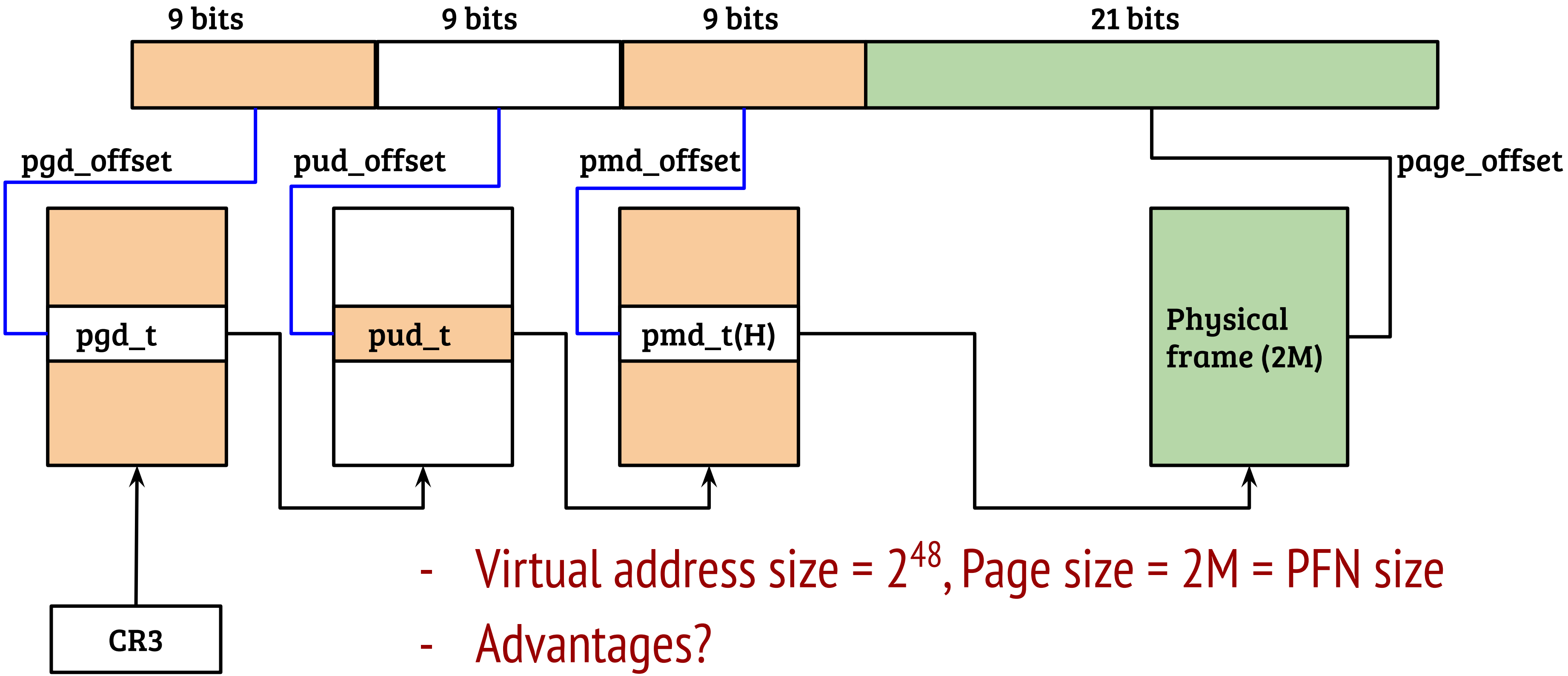

DRAM
PTE mapping the victim PFN (before swap)
PFN(V)
0 1 1 1 1 1
PTE mapping the victim PFN (after swap)
SwapAddress(V)
0 1 1 1 1 0
V
Swap (Hard disk)
OS
AllocatePFN( )

Content of the PFN is now in the swap device. In future, any translation using the PTE will result in a page fault. The page fault handler would copy it back from the swap device.

Page fault with swap-in

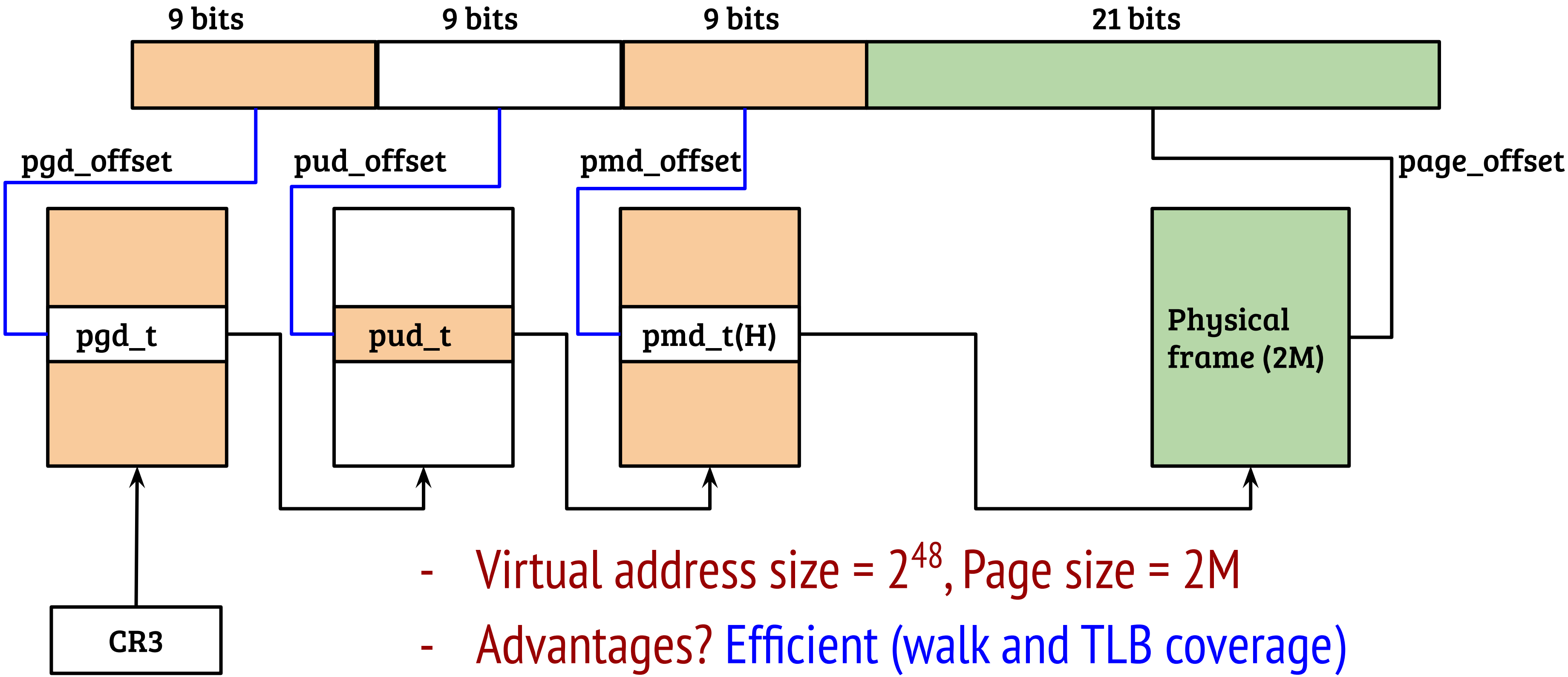
```
HandlePageFault( u64 address, u64 error_code)
{
    If ( AddressExists(current → mm_state, address) &&
        AccessPermitted(current → mm_state, error_code) {
            PFN = allocate_pfn( );
            If ( is_swapped_pte(address) )       // Check if the PTE is swapped out
             swapin(getPTE(address), PFN);       // Copy the swap block  to PFN
            install_pte(address, PFN);           // and update the PTE
            return;
        }
    RaiseSignal(SIGSEGV);
}
```

Efficient translation: Huge page support



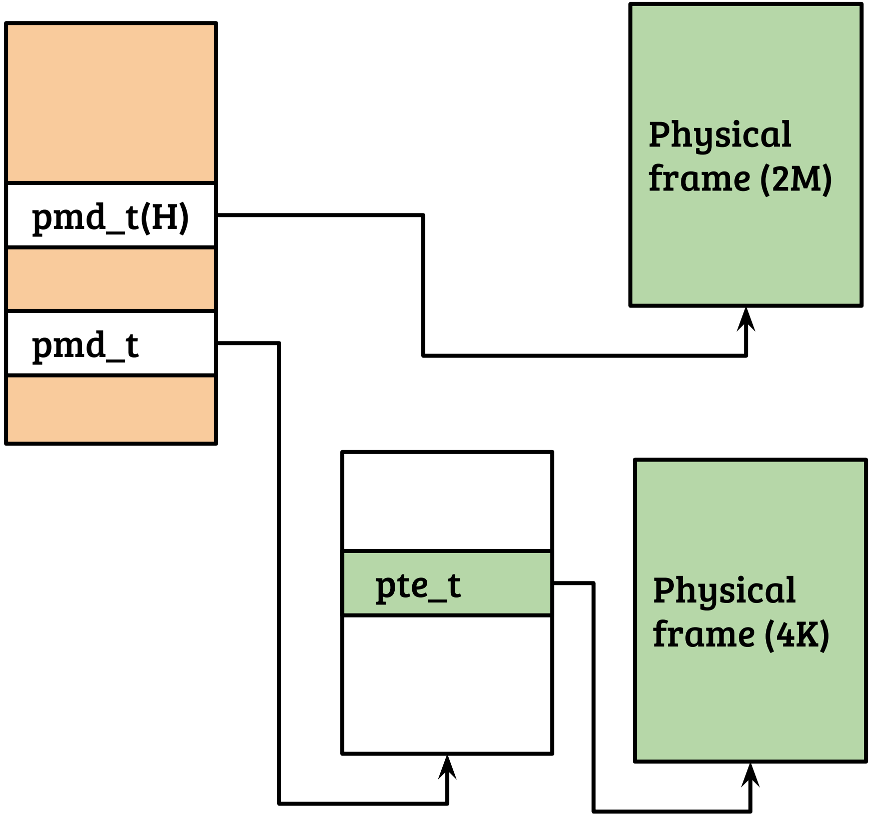
- Virtual address size = 2^{48}, Page size = 2M = PFN size
- Advantages?
- Disadvantages?

Efficient translation: Huge page support



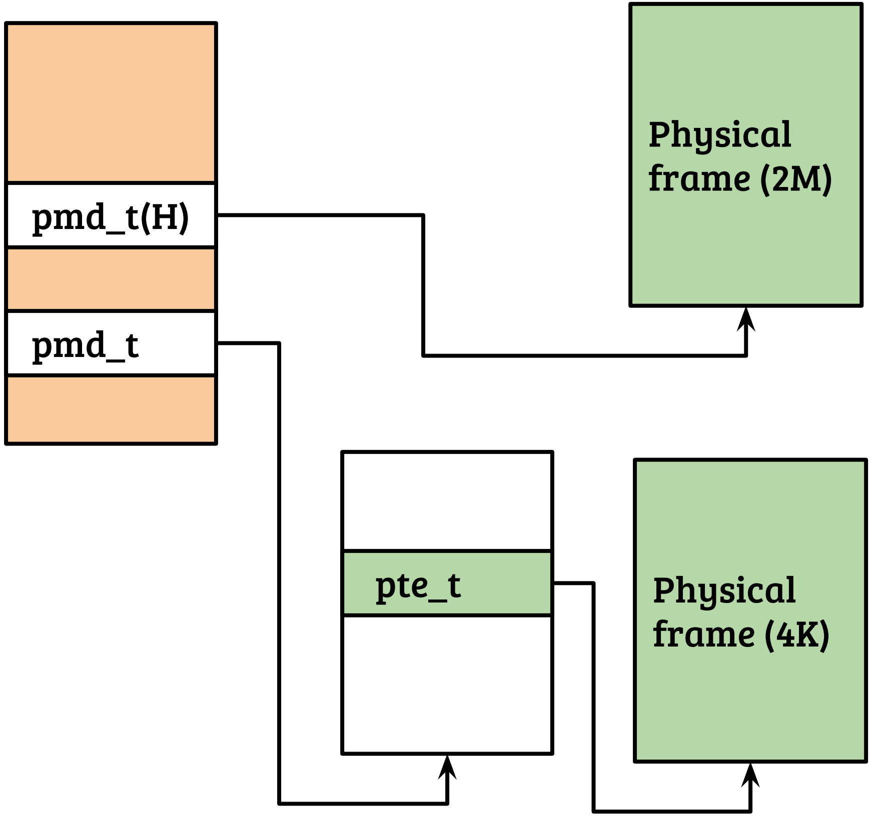
- Virtual address size = 2^{48}, Page size = 2M
- Advantages? Efficient (walk and TLB coverage)
- Disadvantages? Inefficient management

Mixed page size support



```
walk_pmd(pmd, vaddr) {
   if(pmd.H)
      paddr = pmd.nextL( ) + (vaddr & pmask);
  else
       pte = pmd.nextL( ) + pte_offset(vaddr)
} // Simplified H/W logic
```

Mixed page size support



```
walk_pmd(pmd, vaddr) {
   if(pmd.H)
      paddr = pmd.nextL( ) + (vaddr & pmask);
  else
      pte = pmd.nextL( ) + pte_offset(vaddr)
} // Simplified H/W logic
```

- The OS may use the hardware support to implement any policy
- Transparent hugepage (THP) in Linux trie to create huge page mapping in w/o explicit user space assistance
- Policy knobs through sysfs

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel

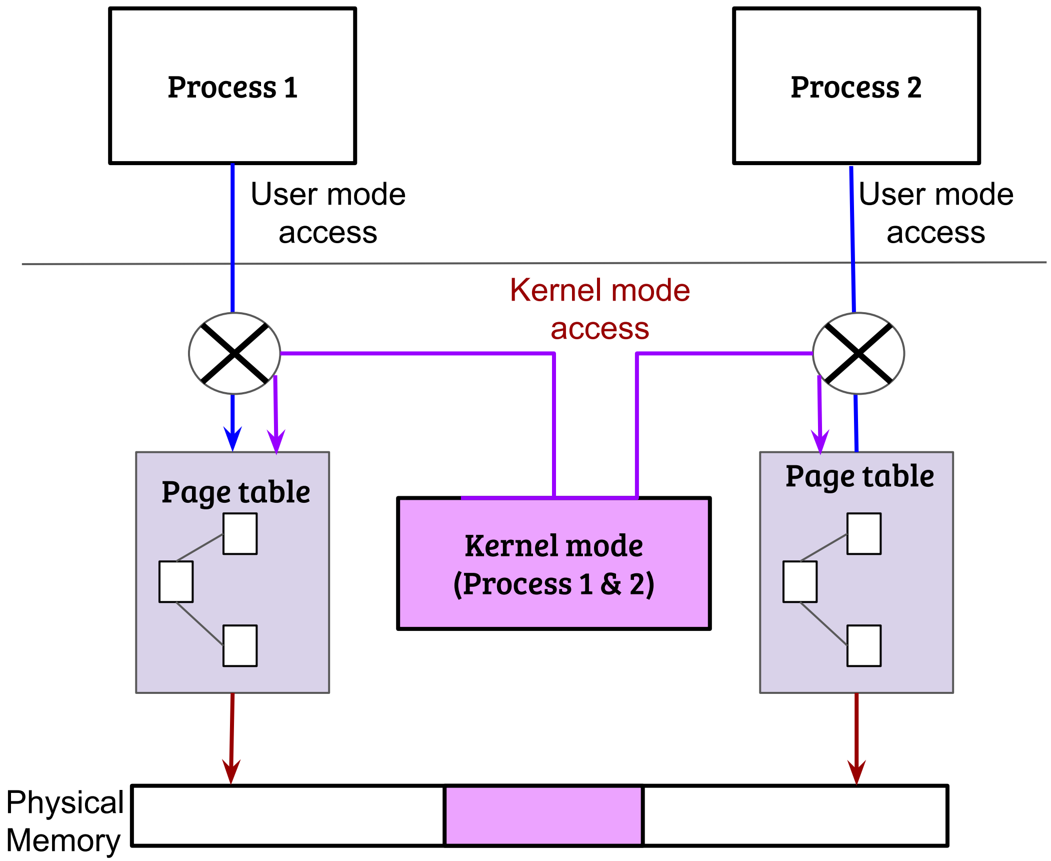
Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes can be accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel
  - Let kernel use the same MM context of the user process
  - No context switch, no problems of accessing user data

Kernel Virtual Memory

- Why not treat kernel as an isolated MM context?
  - Require MM context loading/unloading on user-kernel context switch
  - In kernel context, user data is accessed (a lot!) why?
  - Even worse, user data of many processes can be accessed
  - In X86, a small part of the kernel can not be isolated as HW does not perform MM context switch
- Requirement: efficient memory isolation between user and kernel
  - Let kernel use the same MM context of the user process
  - No context switch, no problems of accessing user data
- How kernel VM change propagated across processes? Isolation issues?

Issue of Kernel VM propagation



- Kernel virtual address mapping should be present in both process page tables.
- Ex: If kernel allocates memory while serving syscall from process-1, process-2 in kernel mode should see it!
- Solution should consider that, "processes and memory are dynamically created and destroyed"

Linux strives on family values!

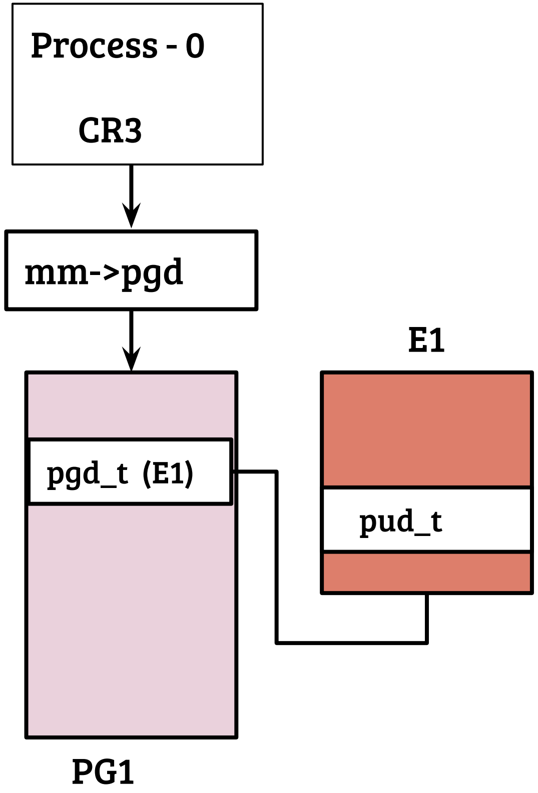
- A child process page table inherits the kernel mappings of the parent
- By implication, the inheritance tree is rooted at the first process
- Mapping changes → update mapping in every process?
    - Does not look good!

Linux strives on family values!

- A child process page table inherits the kernel mappings of the parent
- By implication, the inheritance tree is rooted at the first process
- Mapping changes → update mapping in every process?
  - Does not look good!

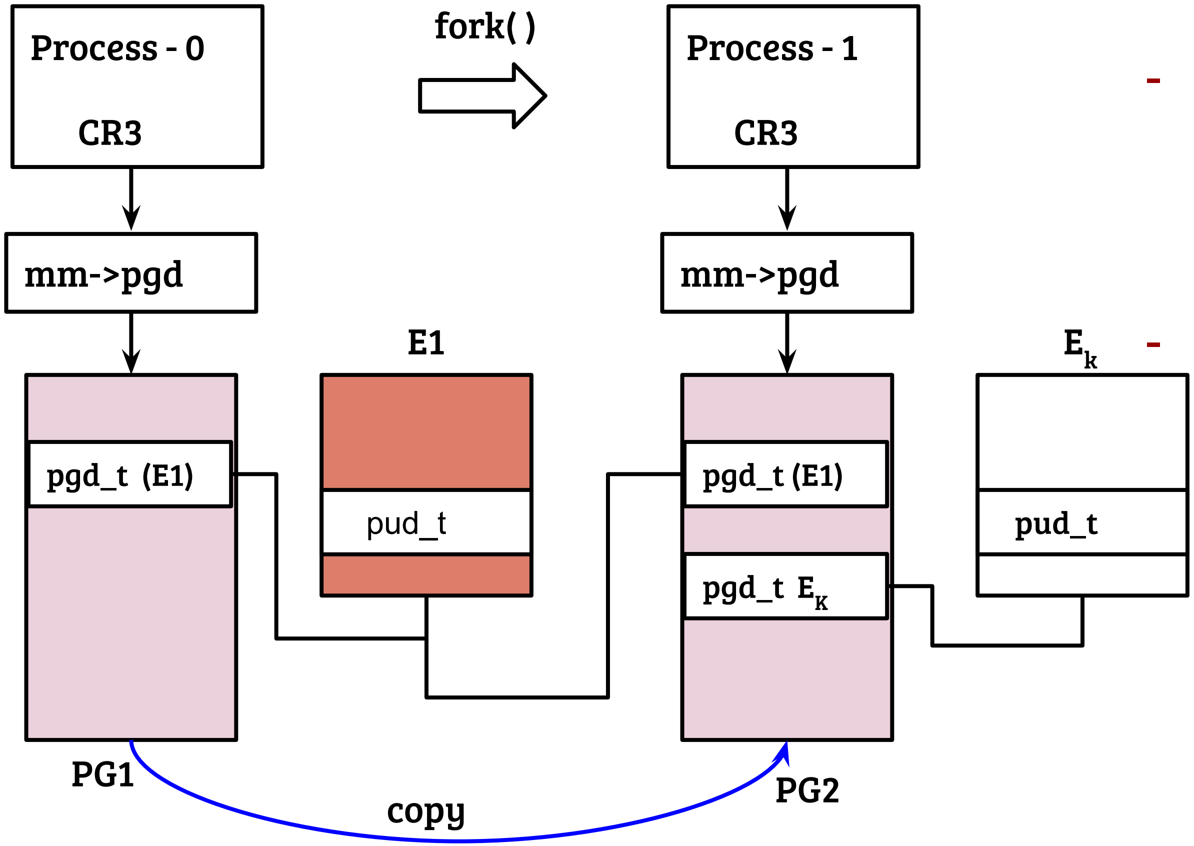
Solution: Every process owns its own **pgd** entries but inherits the kernel **pgd** entries from the parent :-)

Solution overview



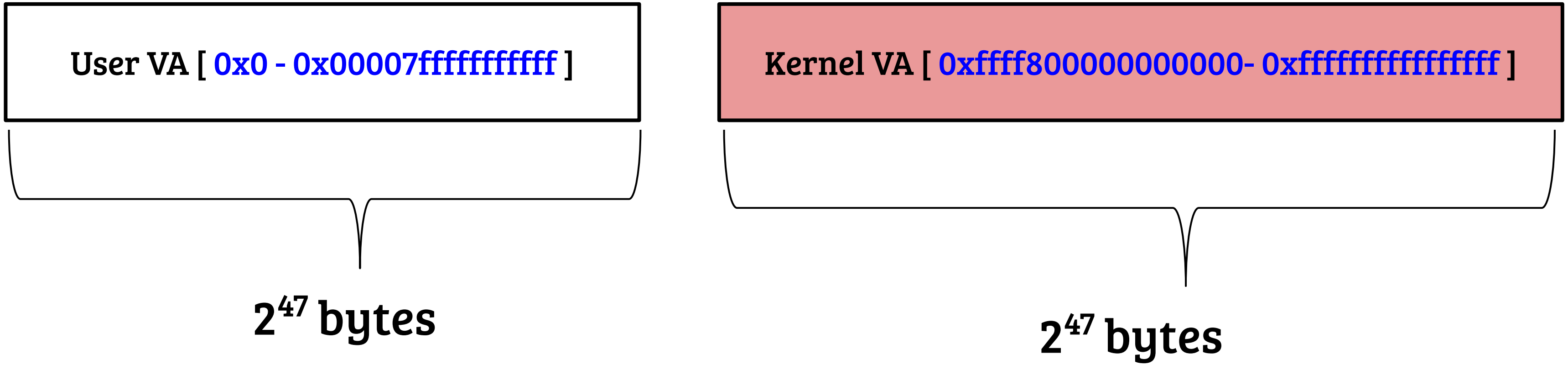
- One (or more) entries in PGD-level (level-4) reserved for kernel mapping
- How many?
- Depends on VA-range covered by one entry and the kernel VA size

Solution overview



- All updates to E1 are visible across all the processes
- So we are at peace! Not really.

Virtual memory layout (x86_64)



User VA [0x0 - 0x00007fffffffffff]

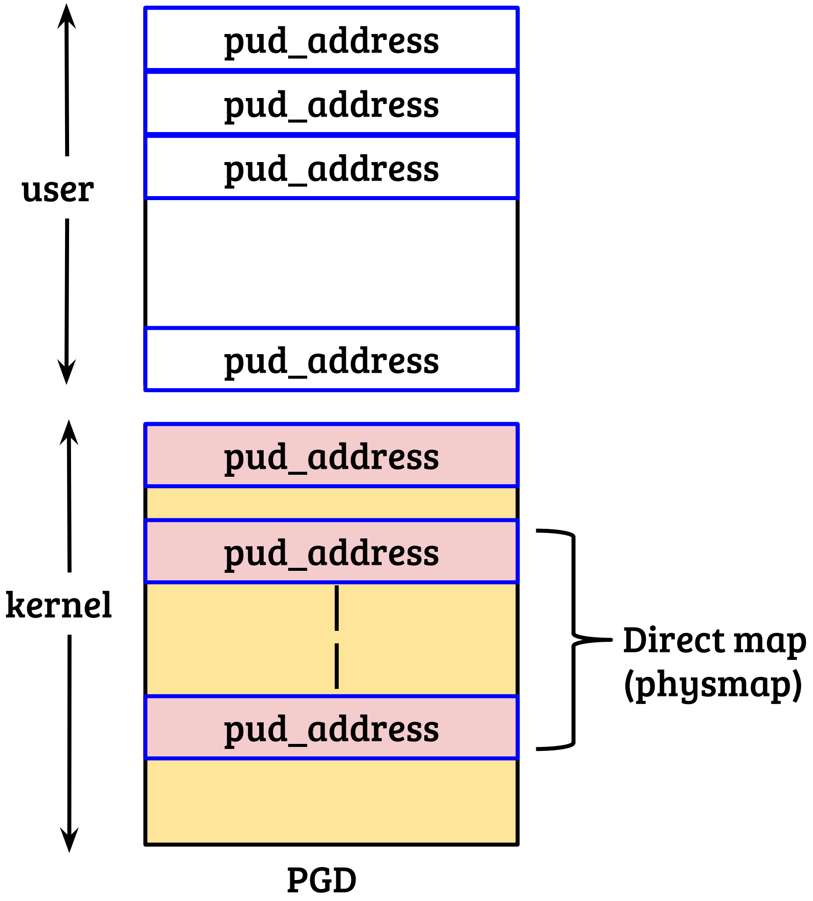
Kernel VA [0xffff800000000000- 0xffffffffffffffff]

2^{47} bytes

2^{47} bytes

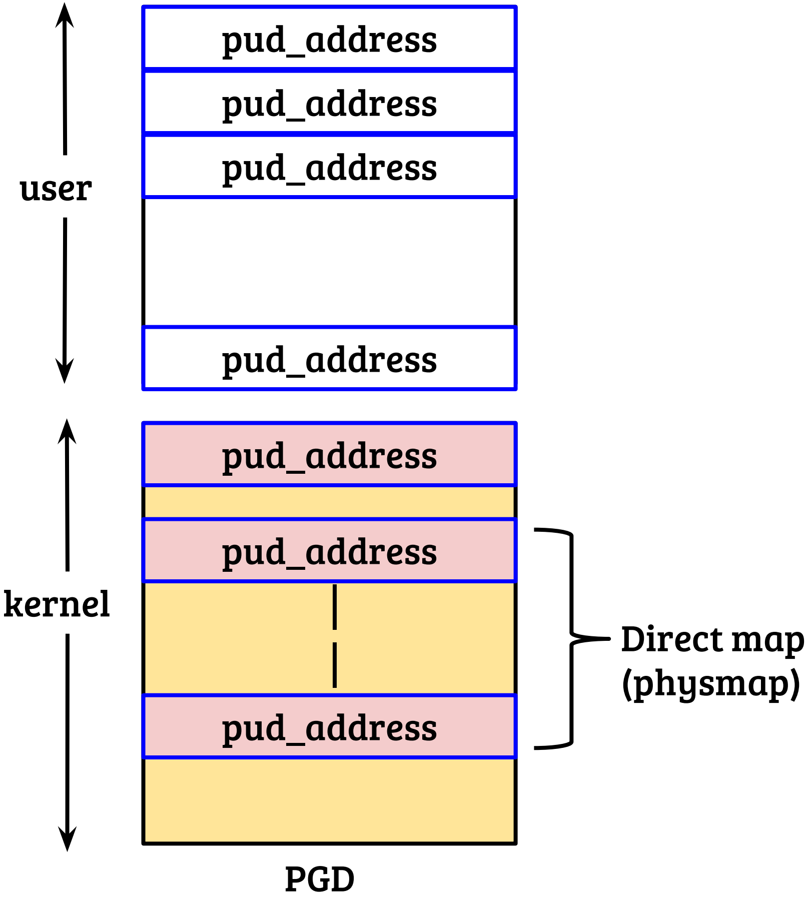
- User virtual addresses use the LSB 47 bits
- Kernel virtual address does not start from 0x800000000000, but from 0xffff800000000000
- Why? Because X86 hardware enforces if 47th bit is one, 48-63 must be set to one

Process address space (user + kernel)



- Virtual address space is split into two parts, user VA and kernel VA
- Kernel mappings are isolated from user through **S/U** bit of page table entry
- Advantages: isolation + efficiency
- What is the need for direct map?

Process address space (user + kernel)



- Virtual address space is split into two parts, user VA and kernel VA
- Kernel mappings are isolated from user through **S/U** bit of page table entry
- Advantages: isolation + efficiency
- What is the need for direct map? Helps in mapping physical address to an already mapped kernel vaddr

Issue with shared address space

```
char array[256 * 4096];        //__alligned(4k);
char secret = *(char *) 0xffff880000000000;
array[secret  << 12] = 0;
```

- This program will result in an exception → Segmentation fault
- Everything seems to be under control. What is the problem then?

Information leakage through out-of-order execution

```
1.   mov RCX, $0xFFFF888000000000;
2.   mov RBX, $array;
3.   mov AL, [RCX];
4.   Shl RAX, $0xC;
5.   mov RBX, qword [RBX + RAX];
```

Executed out-of-order

Exception handler

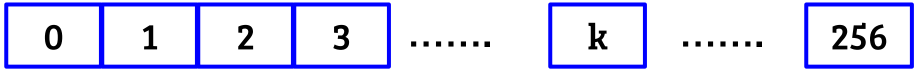
```
1.   cmp CR2, $userend;
2.   Jg raise_segv;
3.   ............
4.   ........
5.   raise_segv:
6.   ..........
```

- By the time the instruction in line#3 is committed (and a fault is raised), instructions in line#4 and #5 are completed out-of-order

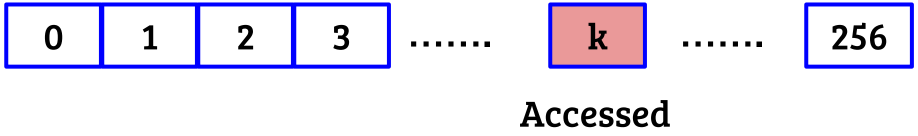
Side-effect: access footprint

```
1.  char array[256 * 4096];        //__alligned(4k);
2.  char secret = *(char *) 0xffff880000000000;
3.  array[secret  << 12] = 0;
```

Array (before the program execution) : block 0 == {0 - 4095} etc.

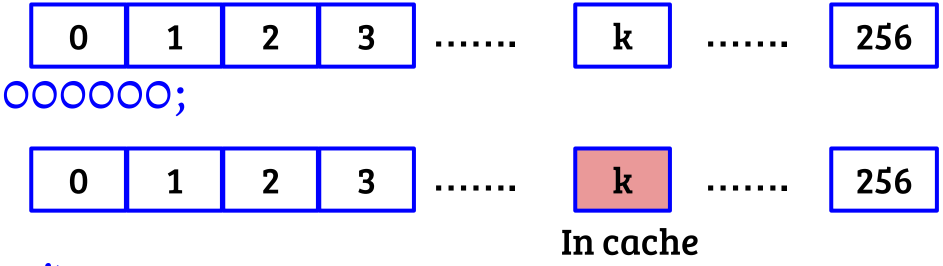


Array (after out-of-order execution of #3) {assume secret = k}



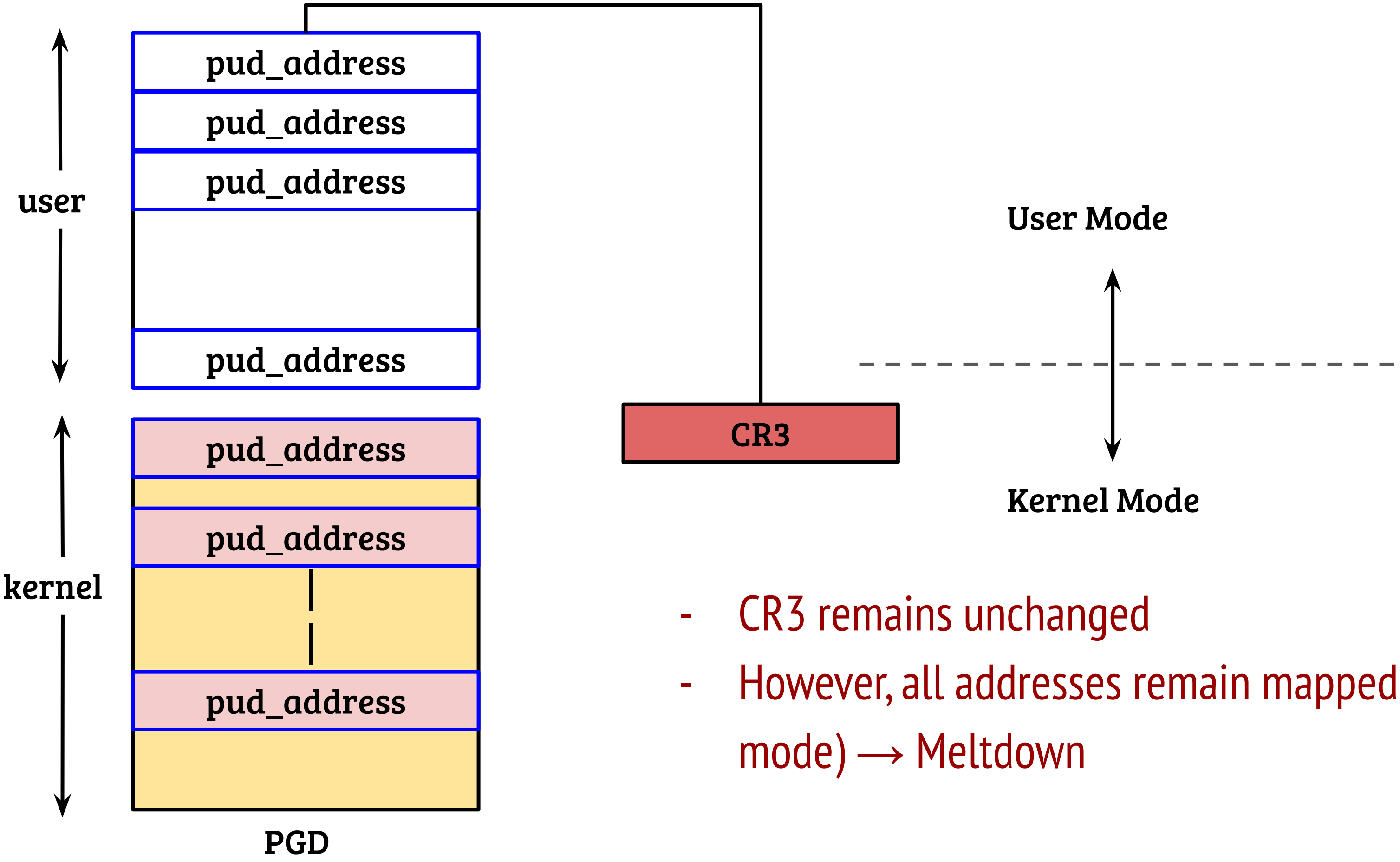
OOO vulnerability + Flush-Reload

```
unsigned time[256];
char array[256 * 4096];
flush_array(array);
char secret = *(char *) 0xffff888000000000;
array[secret  << 12] = 0;
for(i=0; i<256; ++i)
            access_and_time(array, time, i);
secret = find_index_with_min_time( time);
```



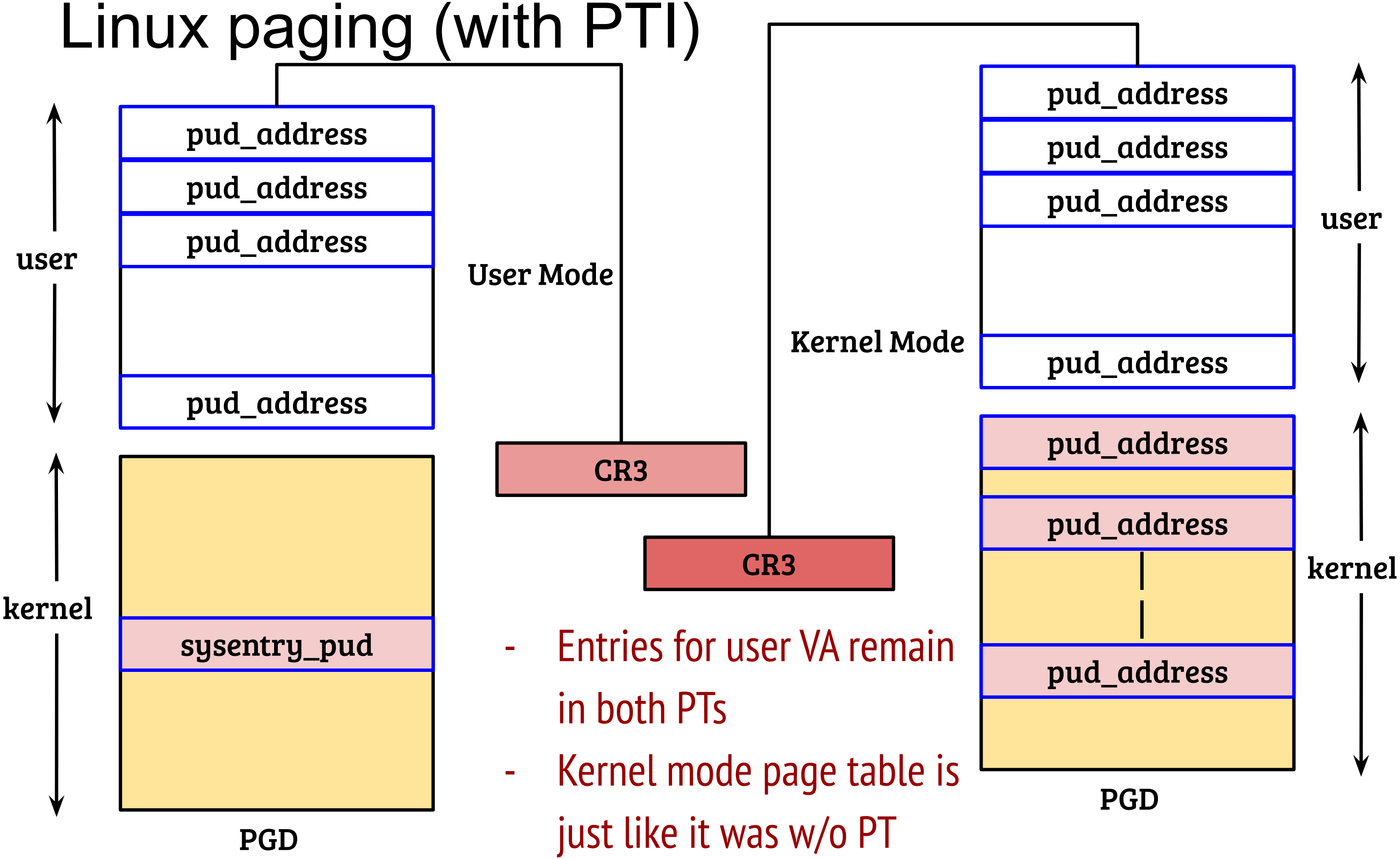
- Result: indirectly read the value of secret
- Meltdown is easy.... Some subtle points still remain
- What is the fix?

Linux paging (before PTI)



- CR3 remains unchanged
- However, all addresses remain mapped (even in user mode) ⟶ Meltdown

Linux paging (with PTI)



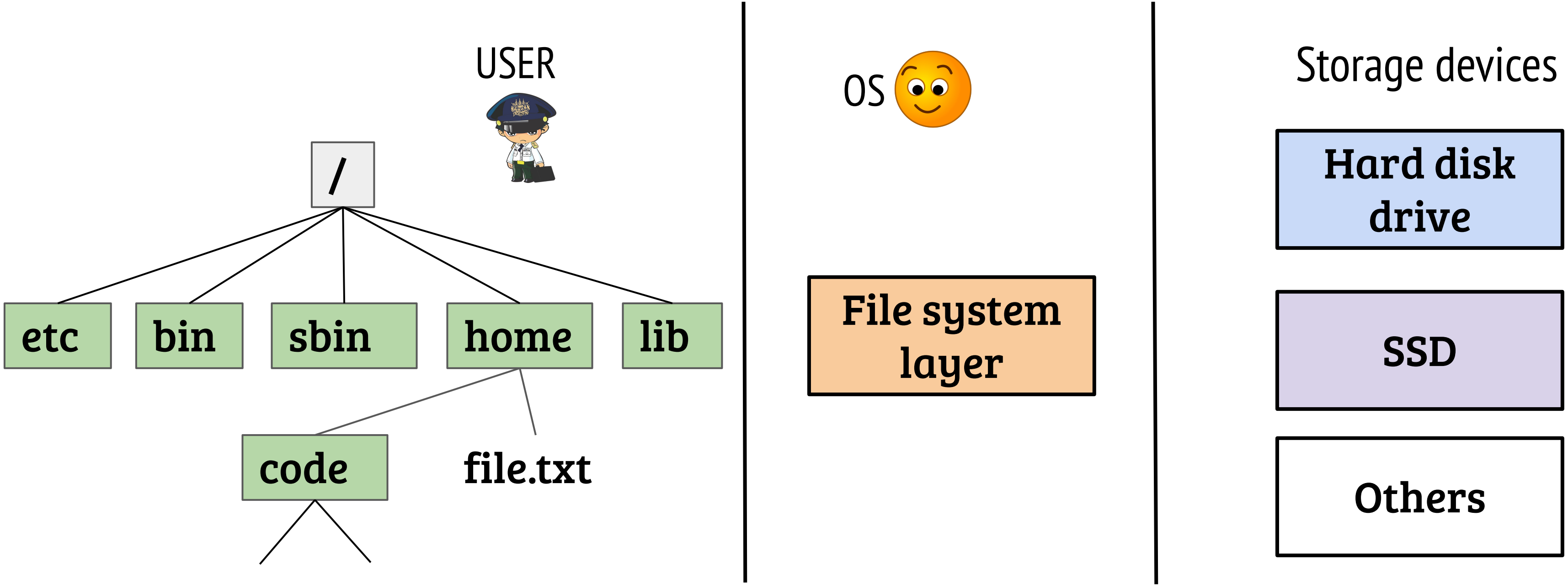
- Entries for user VA remain in both PTs
- Kernel mode page table is just like it was w/o PT

CS614: Linux Kernel Programming

File System Overview

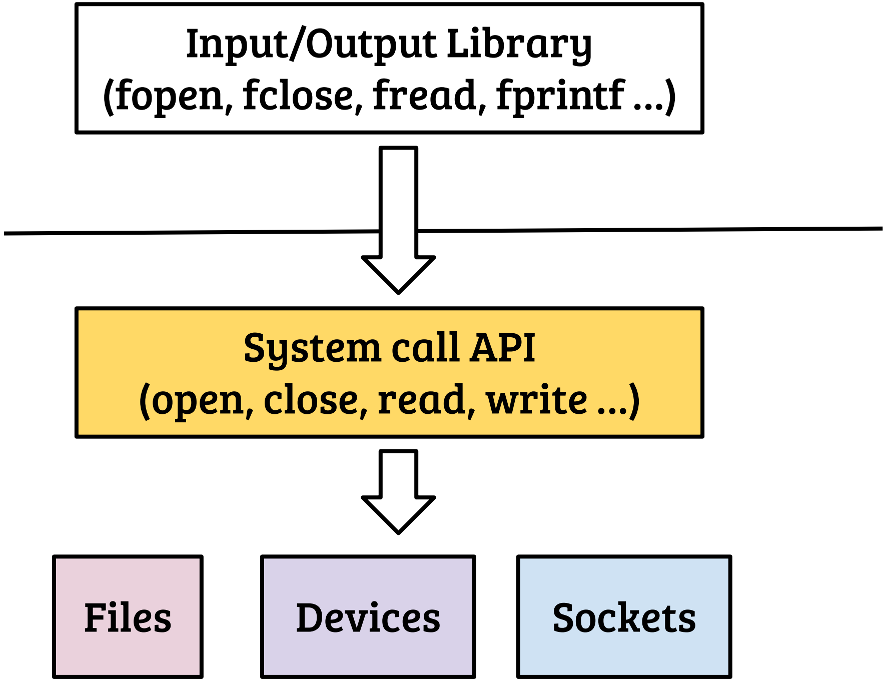
Debadatta Mishra, CSE, IIT Kanpur

Recap: file system



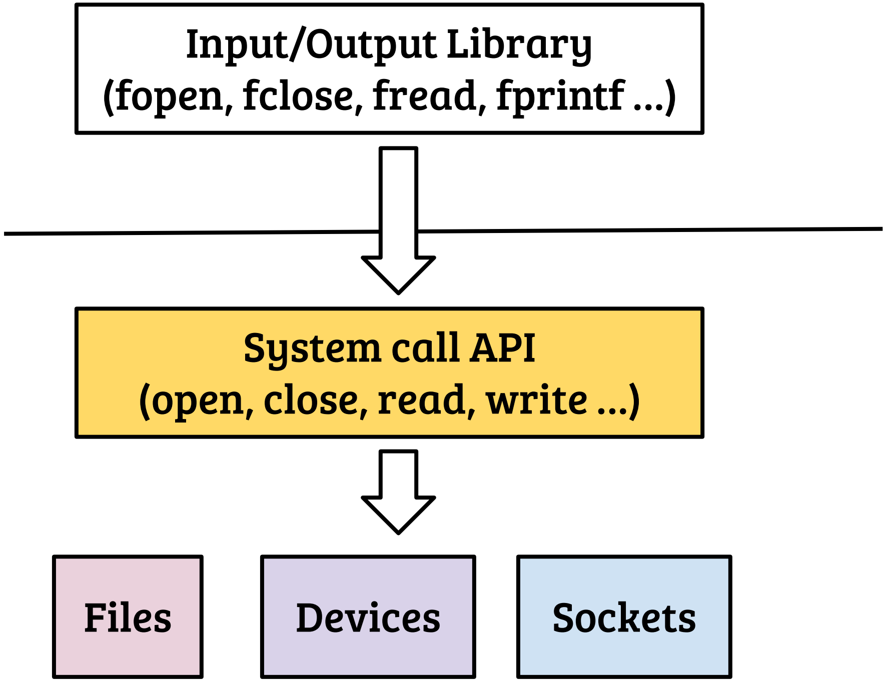
- File system is an important OS subsystem
    - Provides abstractions like files and directories
    - Hides the complexity of underlying storage devices

File system interfacing


Input/Output Library
(fopen, fclose, fread, fprintf ...)
System call API
(open, close, read, write ...)
Files
Devices
Sockets

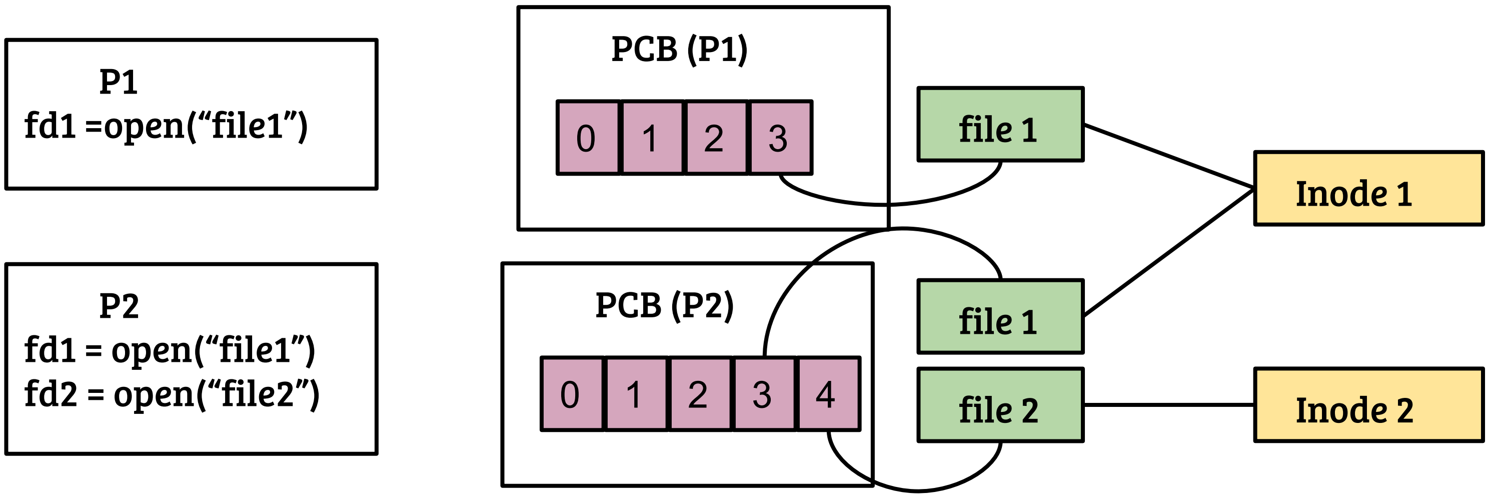
- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.
- Important file related system calls?

File system interfacing


Input/Output Library
(fopen, fclose, fread, fprintf ...)
System call API
(open, close, read, write ...)
Files
Devices
Sockets

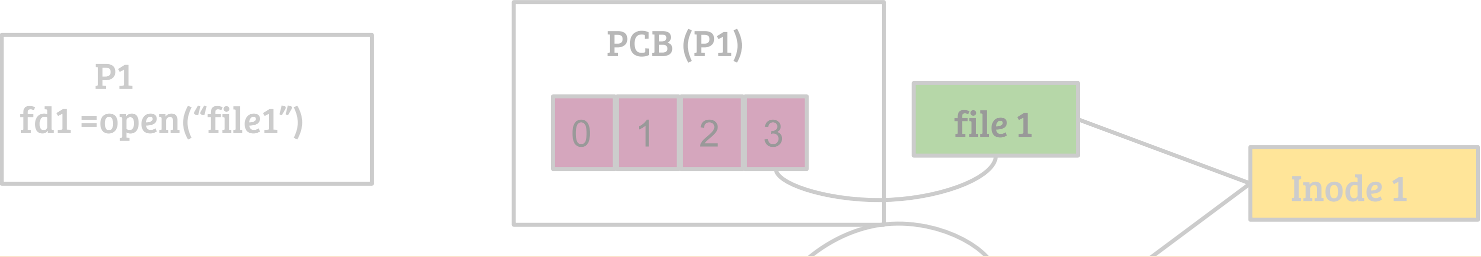
- User process identify files through a file handle a.k.a. file descriptors
- In UNIX, the POSIX file API is used to access files, devices, sockets etc.
- Important file related system calls: open, close, read, write, lseek, dup, stat, select, poll ...

Process view of file



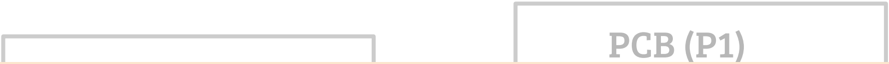
- Per-process file descriptor table with pointer to a "file" object
- file object ⟶ inode is many-to-one

Process view of file



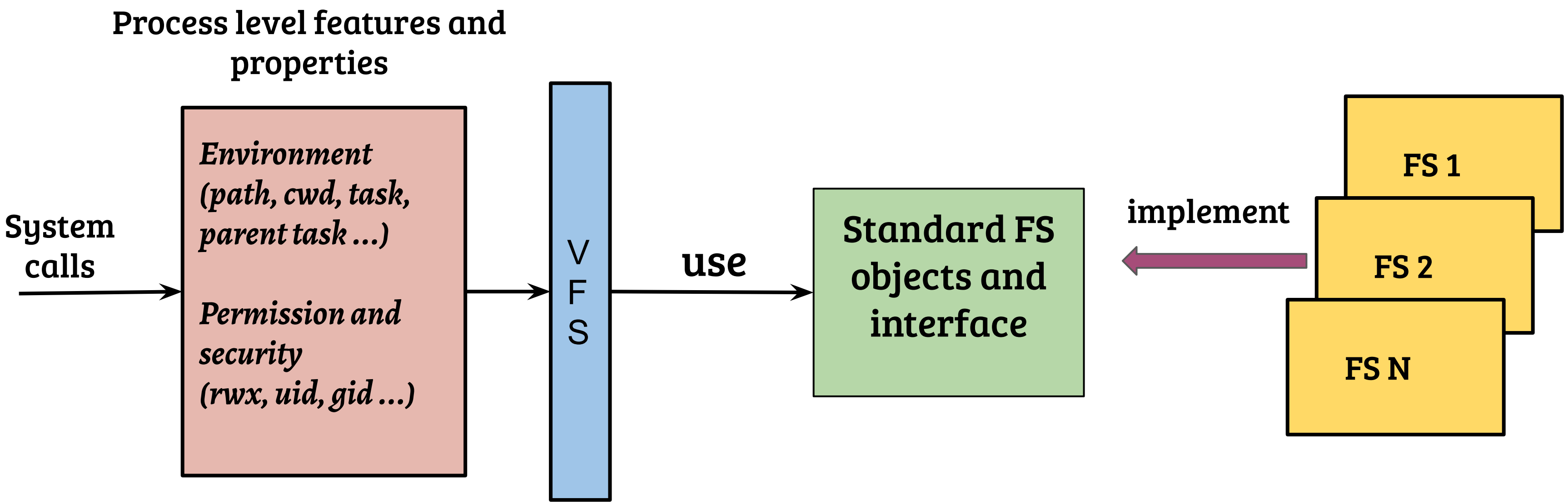
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- Can multiple FDs point to the same file object?

Process view of file



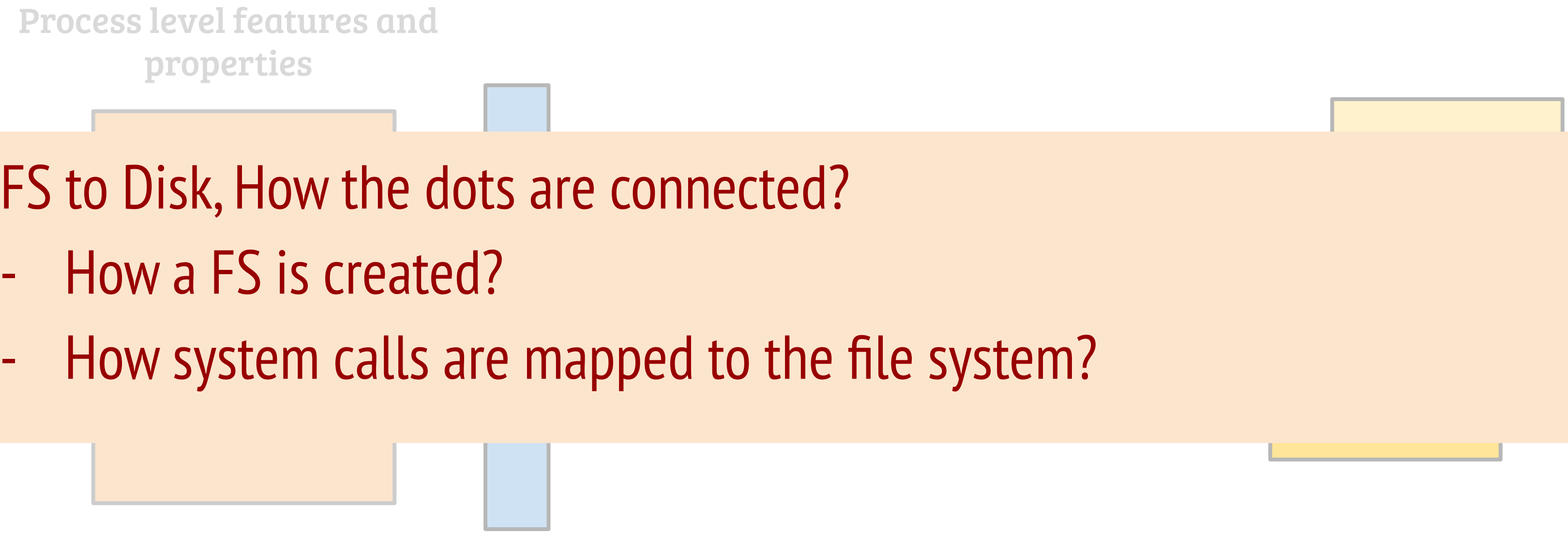
- What happens to the FD table and the file objects across fork( )?
  - What happens in exec( )?
- The FD table is copied across fork( ) ⇒ File objects are shared
- On exec, open files remain shared by default
- Can multiple FDs point to the same file object?
- Yes, duped FDs share the same file object (within a process)

Linux virtual file system (VFS)


Process level features and properties
System calls
Environment (path, cwd, task, parent task ...)
Permission and security (rwx, uid, gid ...)
V F S
use
Standard FS objects and interface
implement
FS 1
FS 2
FS N

- Object and interface choices guided by API requirement (mostly)
- Sometimes standards (e.g., POSIX) determines the interfacing
- Implementation can be different for different file systems

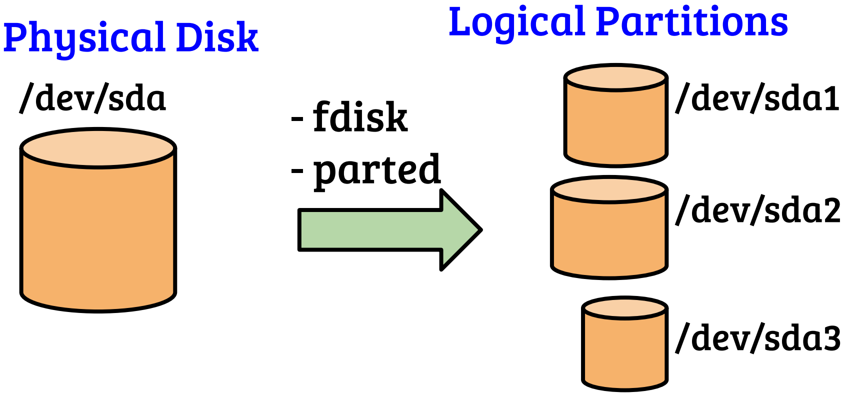
Linux virtual file system (VFS)



- VFS to Disk, How the dots are connected?
    - How a FS is created?
    - How system calls are mapped to the file system?

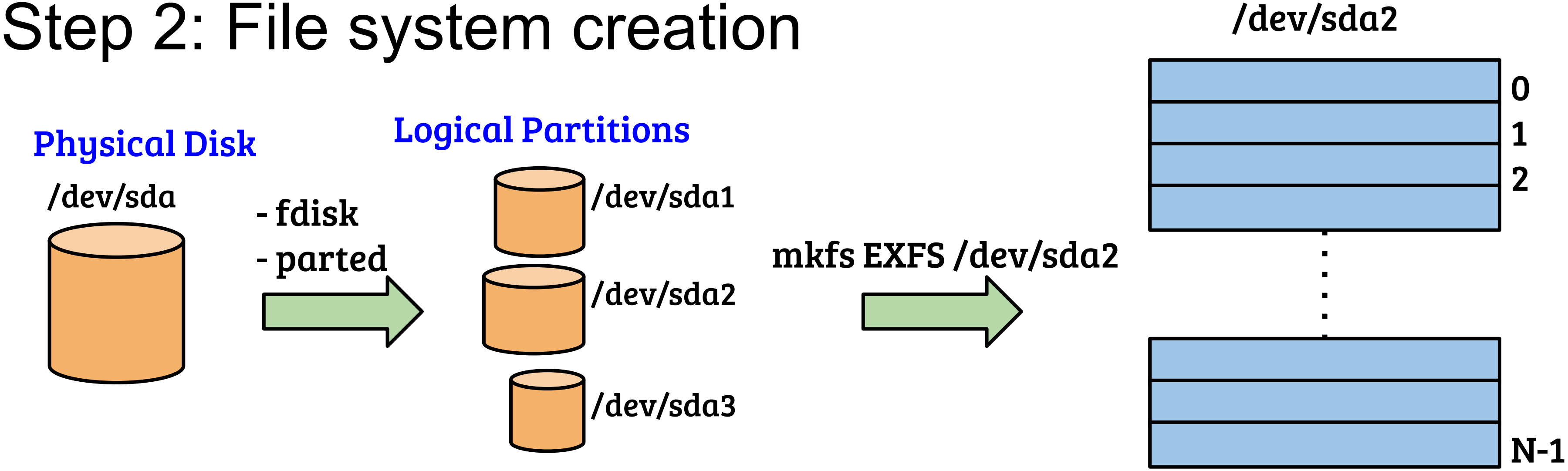
- Object and interface choices guided by API requirement (mostly)
- Sometimes standards (e.g., POSIX) determines the interfacing
- Implementation can be different for different file systems

Step-1: Disk device partitioning



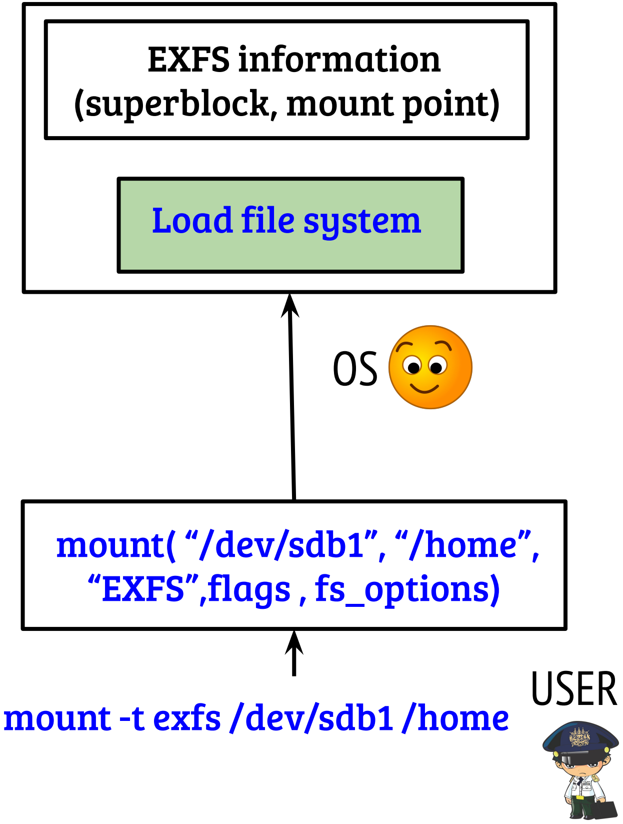
- Partition information is stored in the boot sector of the disk
- Creation of partition is the first step
  - It does not create a file system
- A file system is created on a partition to manage the physical device and present the logical view
- All file systems provide utilities to initialize file system on the partition (e.g., MKFS)

Step 2: File system creation



- MKFS creates initial structures in the logical partition
    - Creates the entry point to the filesystem (known as the super block)
    - At this point the file system is ready to be mounted

Step 3: File system mounting



EXFS information
(superblock, mount point)

Load file system

OS

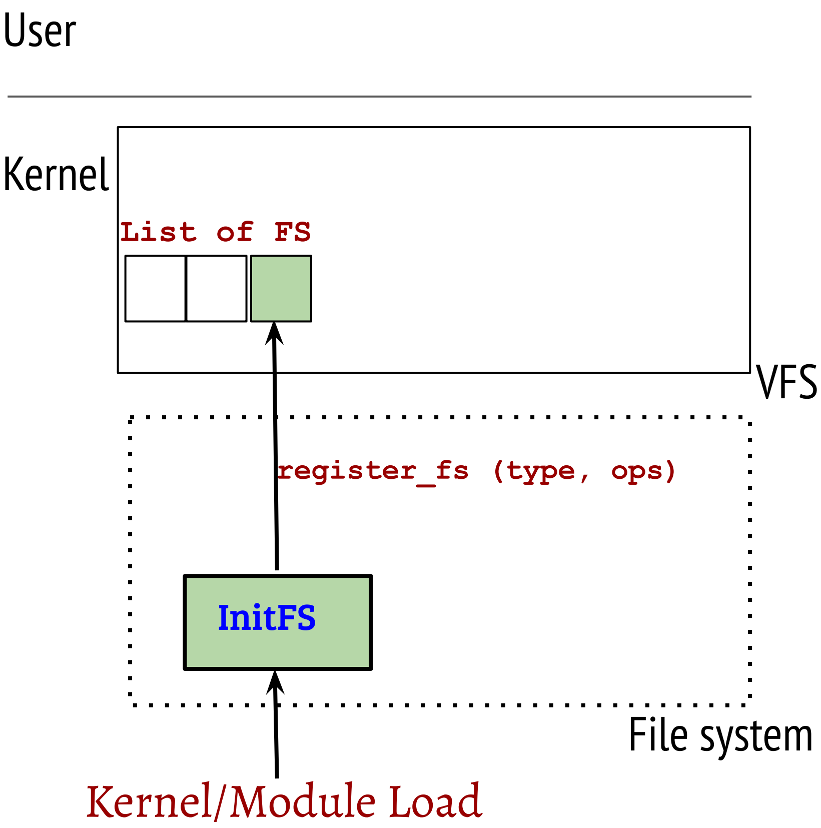
mount(“/dev/sdb1”, “/home”,
“EXFS”,flags , fs_options)

mount -t exfs /dev/sdb1 /home

USER

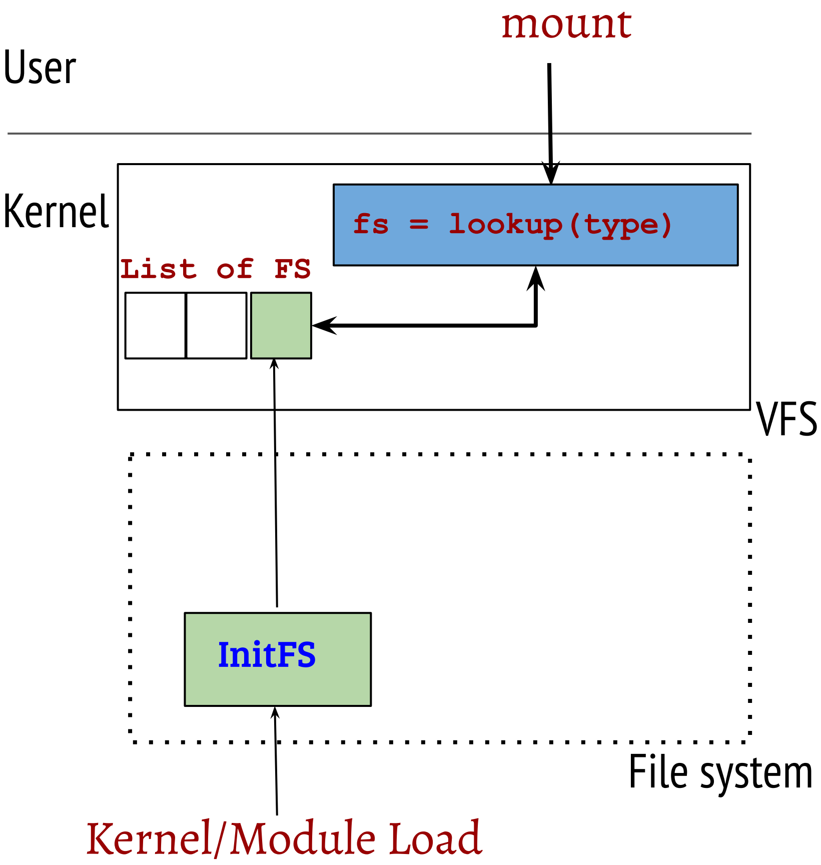
- *mount( )* associates a superblock with the file system mount point
- Example: The OS will use the superblock associated with the mount point “/home” to reach any file/dir under “/home”
- Superblock is a copy of the on-disk superblock along with other information

Details of FS mount in Linux (simplified)



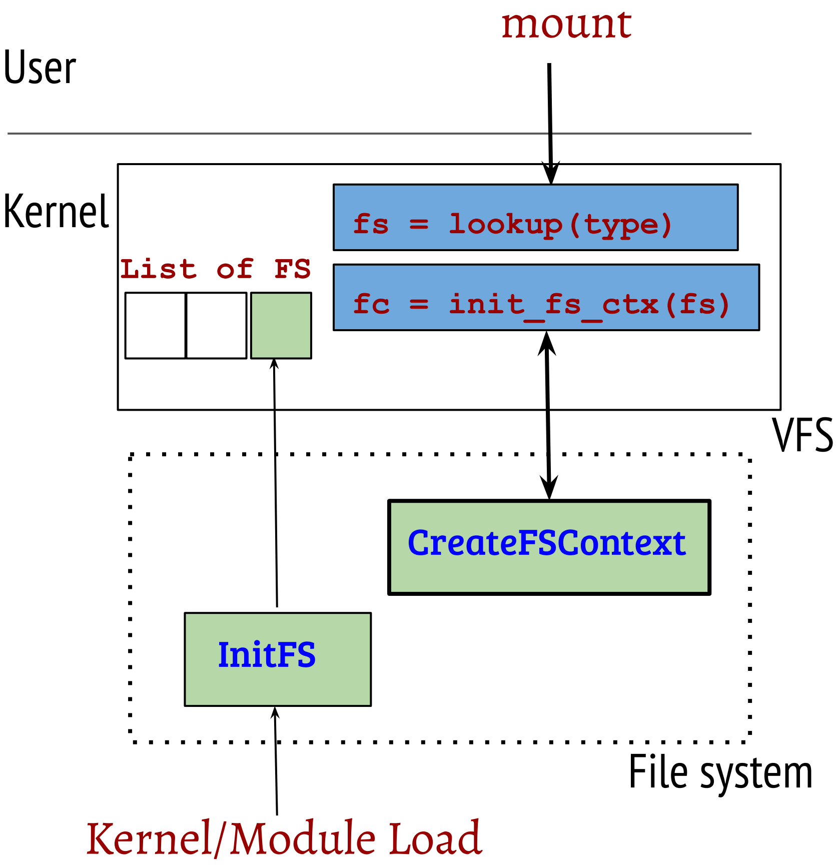
- File system registers itself with the VFS layer during initialization
  - “type” is the identity of the file system (e.g., ext4)
  - “ops” contains the callbacks for different events such as context initialization and mount
- VFS layer maintains a list of registered file system types

Details of FS mount in Linux (simplified)



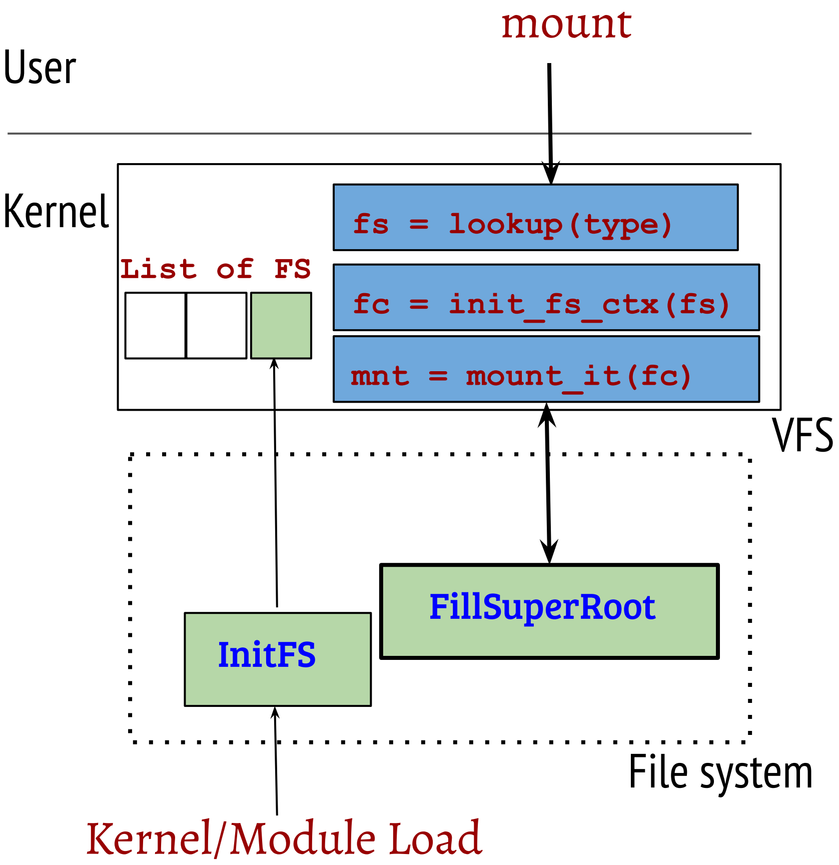
- System call handler for mount looks up the FS type

Details of FS mount in Linux (simplified)



- System call handler for mount looks up the FS type
- Creates a context – an instance of the FS for a given mount point

Details of FS mount in Linux (simplified)



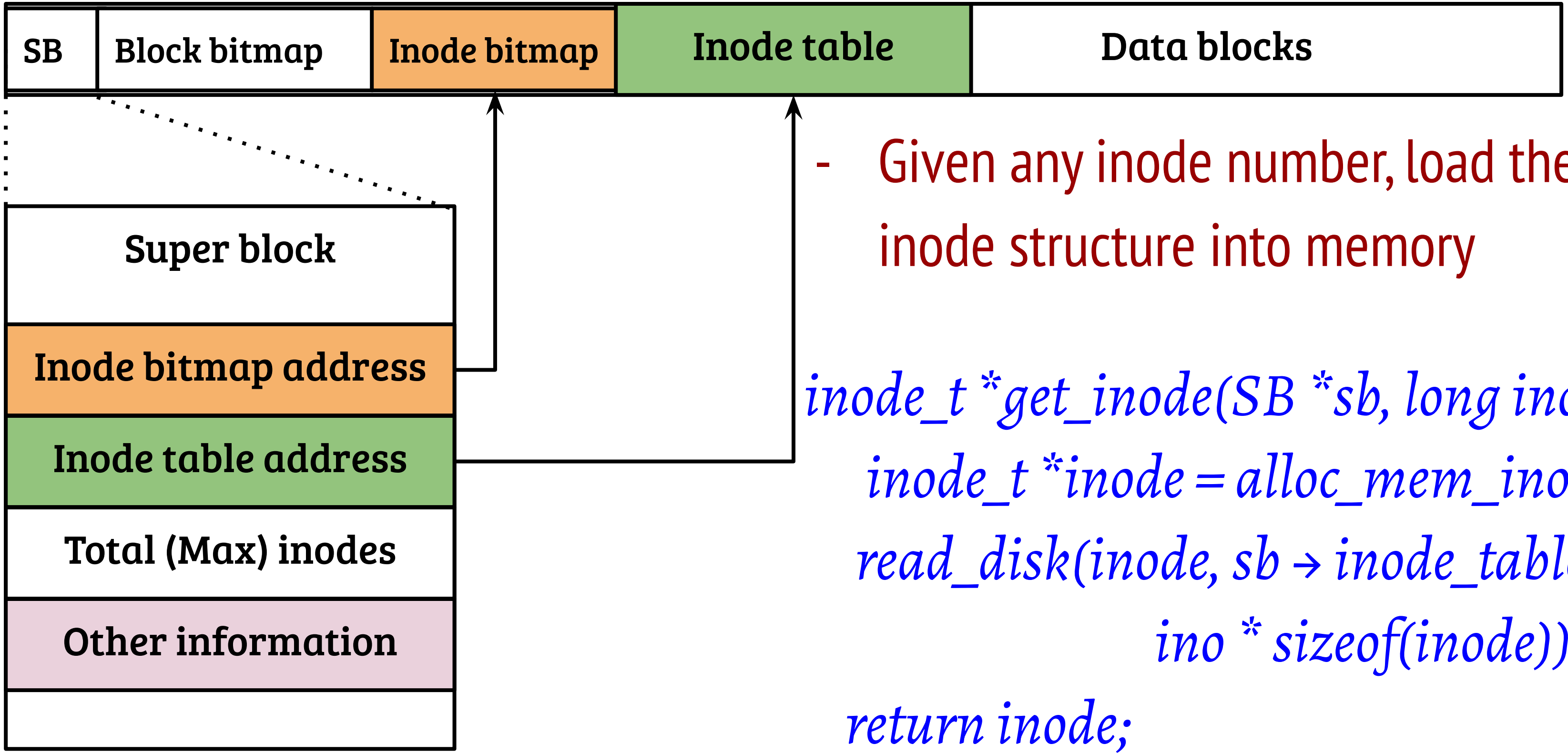
- System call handler for mount looks up the FS type
- Creates a context – an instance of the FS for a given mount point
- The FS fills superblock and root inode information (by performing disk block I/O)
- A new mount point is created at the VFS layer for future use. What kind of use?

Structure of an example superblock

```
struct superblock{
    u16 block_size;
    u64 num_blocks;
    u64 last_mount_time;
    u64 root_inode_num;
    u64 max_inodes;
    disk_off_t inode_table;
    disk_off_t blk_usage_bitmap;
     …
};
```

- Superblock contains information regarding the device and the file system organization in the disk
- Pointers to different metadata related to the file system are also maintained by the superblock
    - Ex: List of free blocks is required before adding data to a new file/directory

Typical file system organization (on-disk)



- Given any inode number, load the inode structure into memory

```
inode_t *get_inode(SB *sb, long ino){
   inode_t *inode = alloc_mem_inode( );
   read_disk(inode, sb → inode_table +
                     ino * sizeof(inode));
   return inode;
}
```

File system organization

SB	Block bitmap	Inode bitmap	Inode table	Data blocks

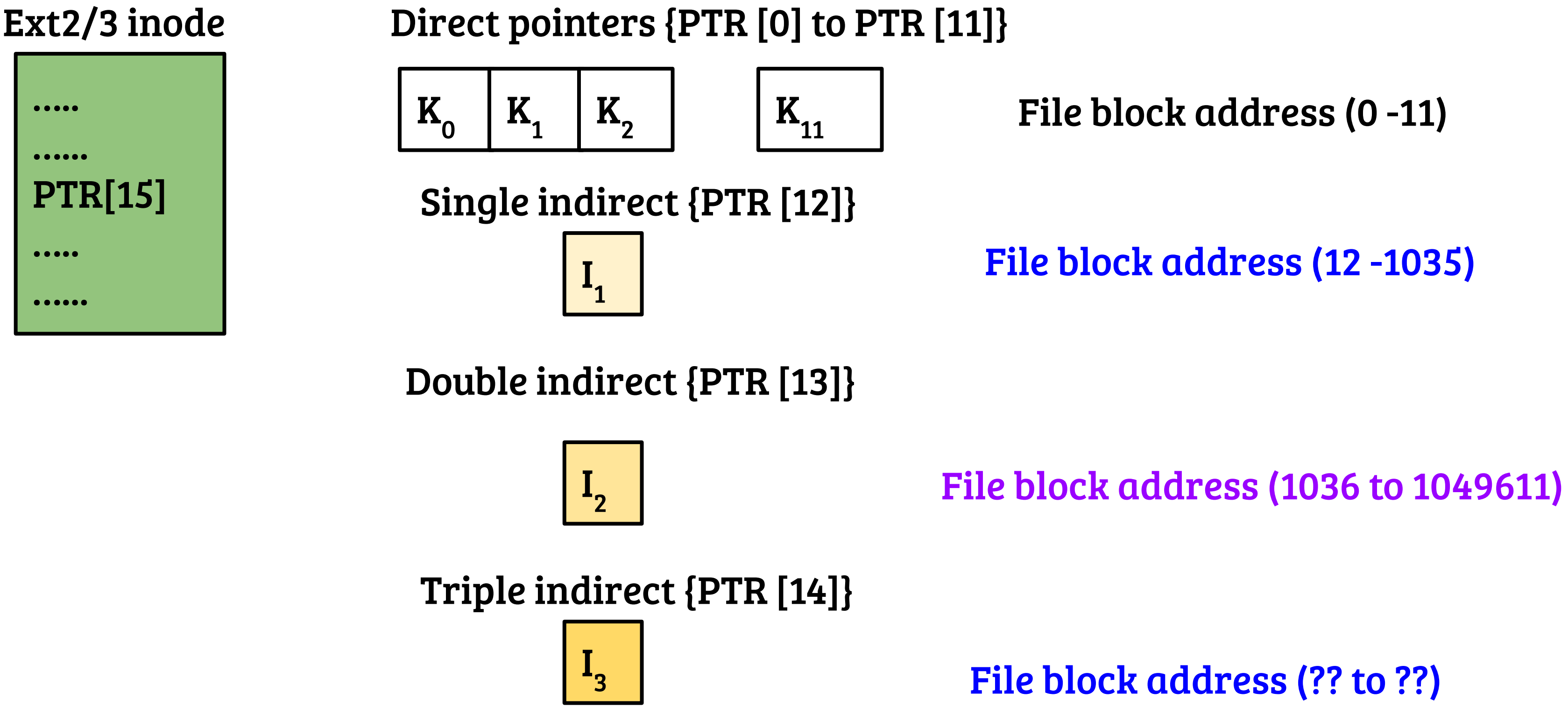
- File system is mounted, the inode number for root of the file system (i.e., the mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Specifically,
  - How to locate the content in disk?
  - How to keep track of size, permissions etc.?

```
return inode;
}
```

Inode

- A on-disk structure containing information regarding files/directories in the unix systems
    - Represented by a unique number in the file system (e.g., in Linux, “ls -i filename” can be used to print the inode)
    - Contains access permissions, access time, file size etc.
    - *Most importantly, inode contains information regarding the file data location on the device*
- Directory inodes also contain information regarding its content, albeit the content is structured (for searching files)

Ext2 file system indexing



Ext2/3 inode

.....

......

PTR[15]

.....

......

Direct pointers {PTR [0] to PTR [11]}

K_0 K_1 K_2 K_{11}

File block address (0 -11)

Single indirect {PTR [12]}

I_1

File block address (12 -1035)

Double indirect {PTR [13]}

I_2

File block address (1036 to 1049611)

Triple indirect {PTR [14]}

I_3

File block address (?? to ??)

File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Specifically,
  - How to locate the content in disk?
  - Index structures in inode are used to map file offset to disk location
  - How to keep track of size, permissions etc.?
  - Inode is used to maintain these information

Organizing the directory content

Fixed size directory entry

```
struct dir_entry{
        inode_t  inode_num;
        char name[FNAME_MAX];
};
```

- Fixed size directory entry is a simple way to organize directory content
- Advantages: avoid fragmentation
- Disadvantages: space wastage

Flat organization of directory entries

Fixed size directory entry

```
struct dir_entry{
        inode_t   inode_num;
        char name[FNAME_MAX];
};
```

Variable size directory entry

```
struct dir_entry{
        inode_t   inode_num;
        u8 entry_len;
        char name[name_len];
};
```

- Variable sized directory entries contain length explicitly
- Advantages: less space wastage (compact)
- Disadvantages: fragmentation issues

File system organization

- File system is mounted, the inode number for root of the file system (mount point) is known, root inode can be accessed. However,
- How to search/lookup files/directories under root inode?
- Read the content of the root inode and search the next level dir/file
- Specifically,
    - How to locate the content in disk?
    - Index structures in inode are used to map file offset to disk location
    - How to keep track of size, permissions etc.?
    - Inode is used to maintain these information

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Examples:
  - Opening a file

```
fd = open(“/home/user/test.c”, O_RDWR);
```

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Examples:
  - Opening a file

```
fd = open("/home/user/test.c", O_RDWR);
```

  - Normal shell operations

```
/home/user$ ls
```

File system and caching

- Accessing data and metadata from disk impacts performance
- Many file operations require multiple block access
- Executables, configuration files, library etc. are accessed frequently
- Many directories containing executables, configuration files are also accessed very frequently. Metadata blocks storing inodes, indirect block pointers are also accessed frequently

/home/user$ ls

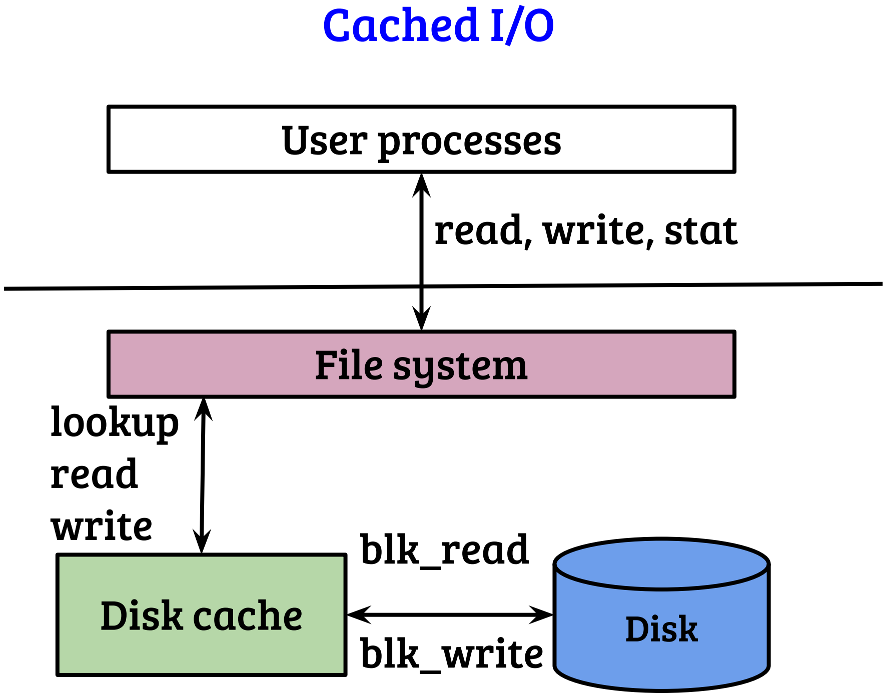
File system and caching

- Accessing data and metadata from disk impacts performance

- Can we store frequently accessed disk data in memory?
  - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
  - Are there any complications because of caching?
  - How the cache managed? What should be the eviction policy?

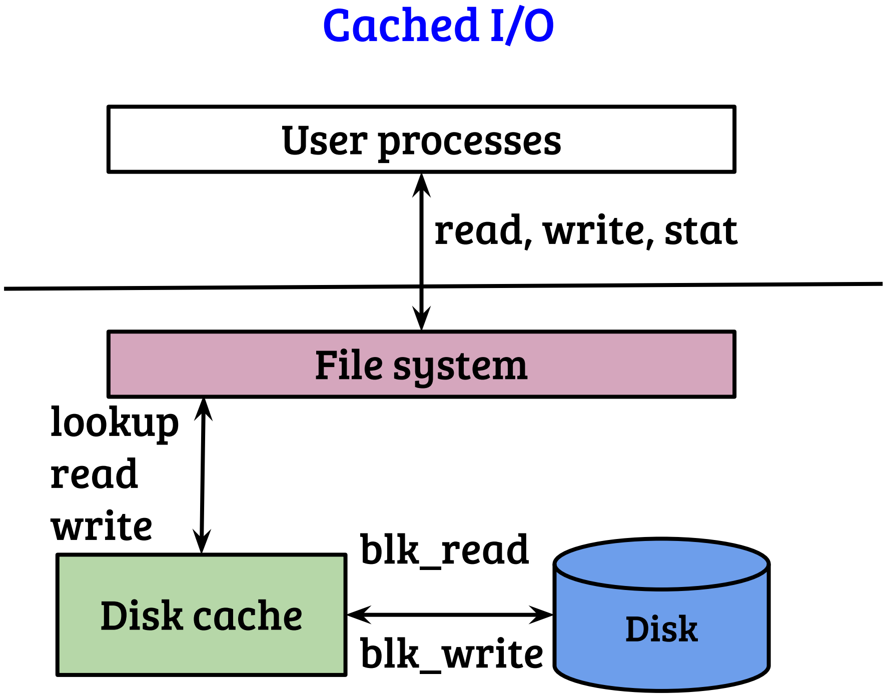
/home/user$ ls

Block layer caching



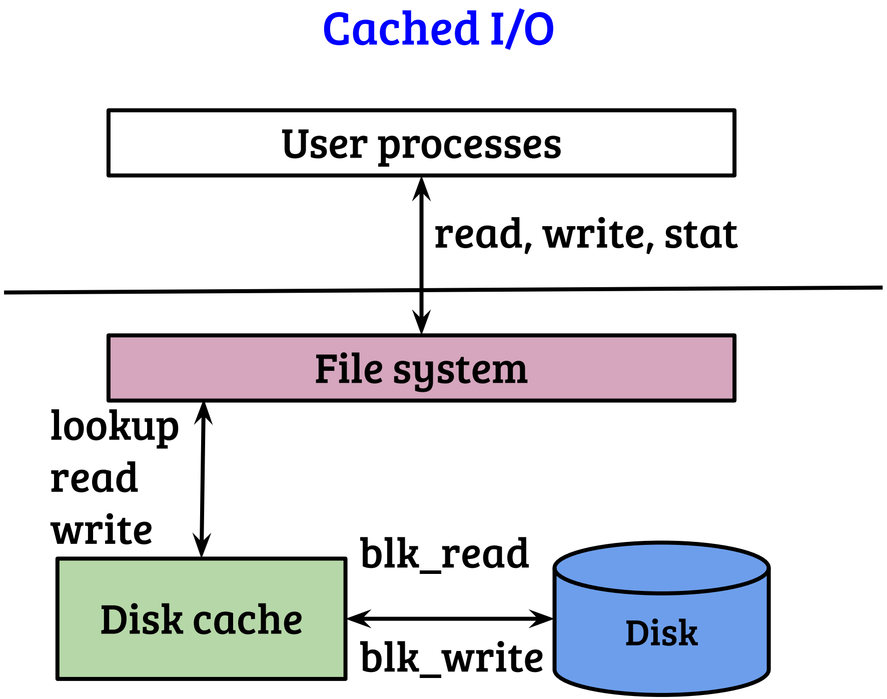
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?

Block layer caching



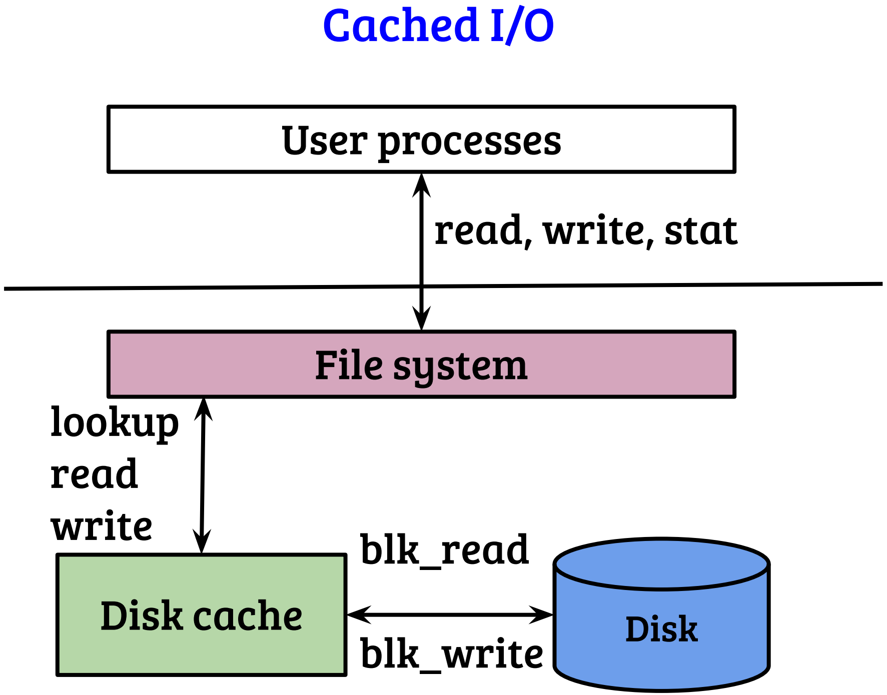
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache

Block layer caching



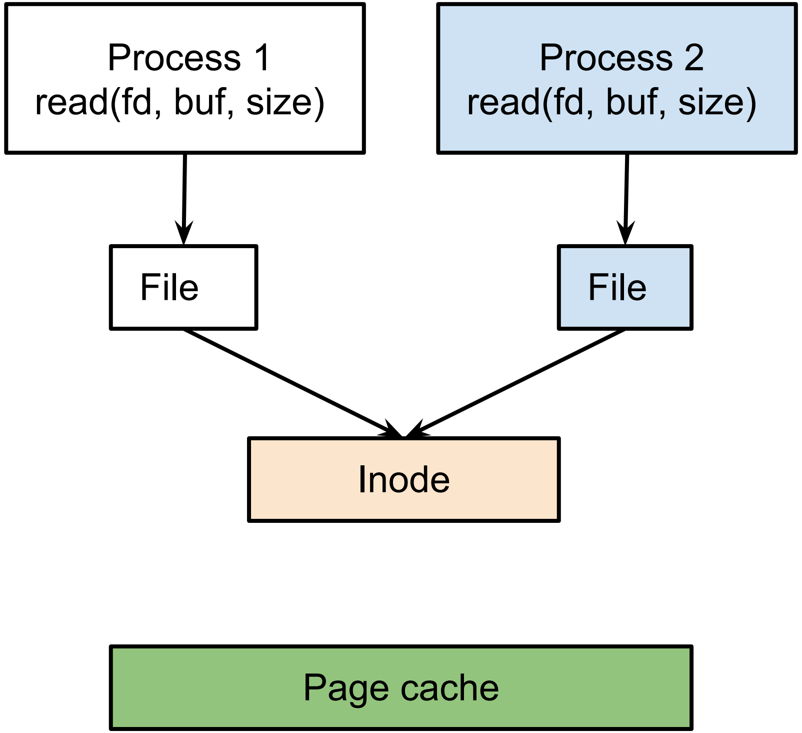
- Lookup memory cache using the block number as the key
- How does the scheme work for data and metadata?
- For data caching, file offset to block address mapping is required before using the cache
- Works fine for metadata as they are addressed using block numbers

File layer caching (Linux page cache)



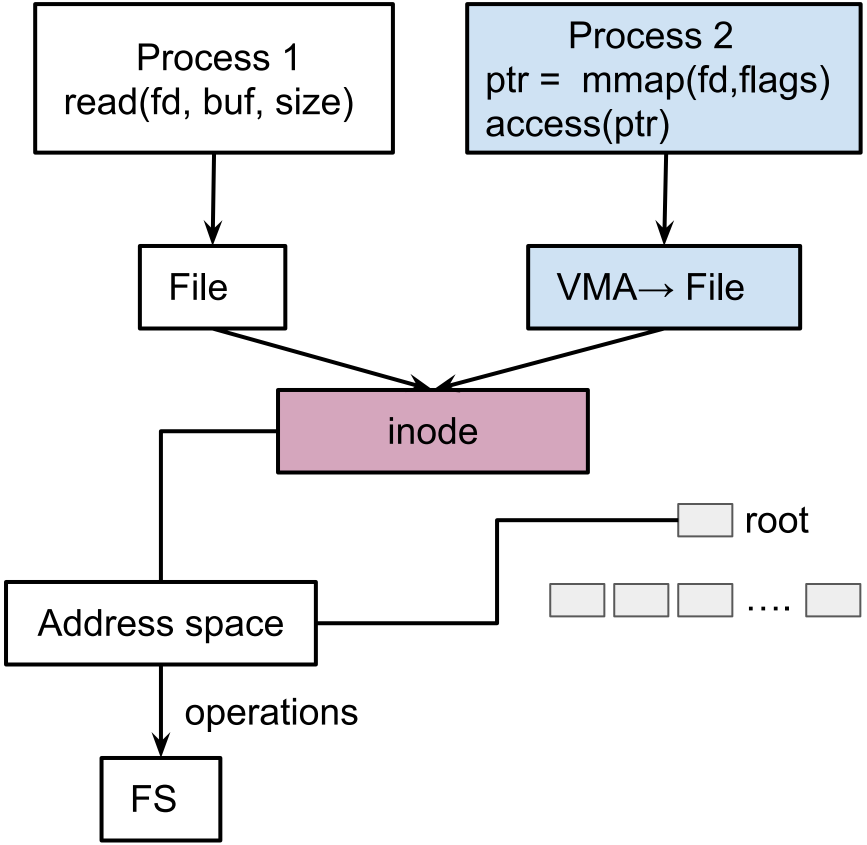
- Store and lookup memory cache using {inode number, file offset} as the key
- For data, index translation is not required for file access
- Metadata may not have a file association, should be handled differently (using a special inode may be!)

Linux page cache: A multi-purpose FS caching layer



- Requirement: File block lookup at different offsets
    - File size can range from very small to huge
- Recall: mmap-ing a file creates a VMA struct
- Should handle both file I/O and page faults

File → Inode → Address spaces → Page Cache



- A per inode cache
  - Lookup, insert, evict, dirty-flush
- Radix tree
  - Root pointed by address space struct
  - Operations at a page size (4K) granularity
- Homework: For a given file, find the number of file blocks cached in PC

File system and caching

- Accessing data and metadata from disk impacts performance
- Can we store frequently accessed disk data in memory?
    - What is the storage and lookup mechanism? Are the data and metadata caching mechanisms same?
    - File layer caching is desirable as it avoids index accesses on hit, special mechanism required for metadata.
    - Are there any complications because of caching?
    - How the cache managed? What should be the eviction policy?

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?

Caching and consistency

- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
  - Example-1: If a write( ) system call is successful, data must be written
  - Example-2: If a file creation is successful then, file is created.
  - Difficult to achieve with asynchronous I/O

Caching and consistency

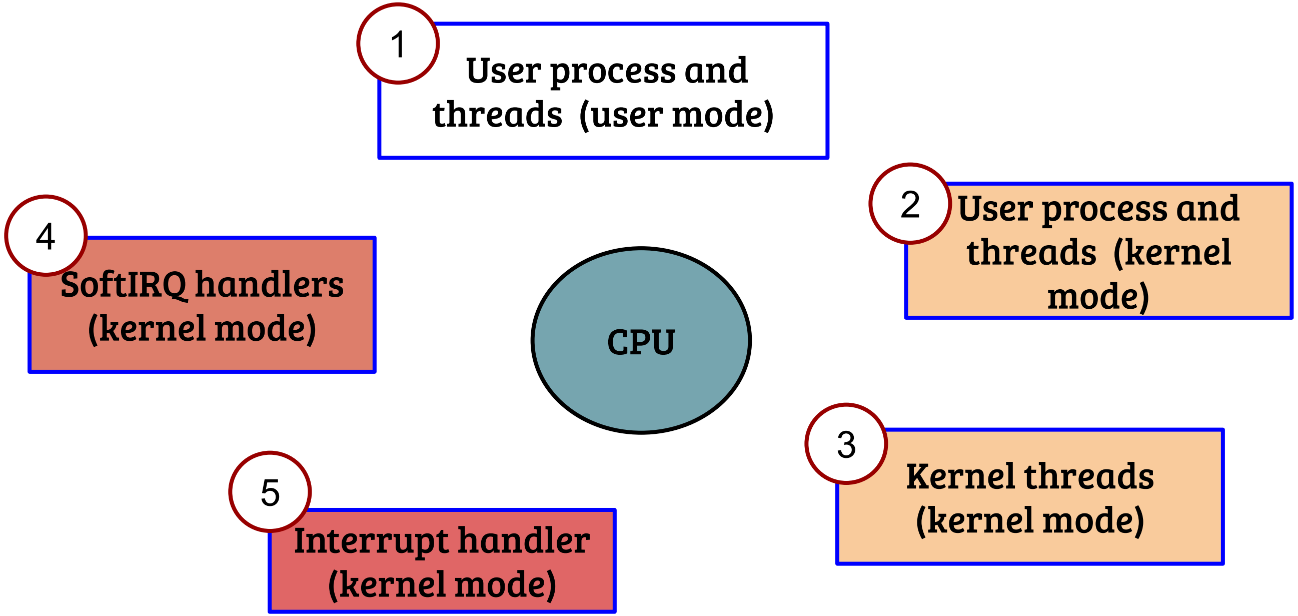
- Caching may result in inconsistency, but what type of consistency?
- System call level guarantees
  - Example-1: If a write( ) system call is successful, data must be written
  - Example-2: If a file creation is successful then, file is created.
  - Difficult to achieve with asynchronous I/O
- Consistency w.r.t. file system invariants
  - Example-1: If a block is pointed to by an inode data pointers then, corresponding block bitmap must be set
  - Example-2: Directory entry contains an inode, inode must be valid
  - Possible, require special techniques

CS614: Linux Kernel Programming

Process, Thread, Kernel Threads

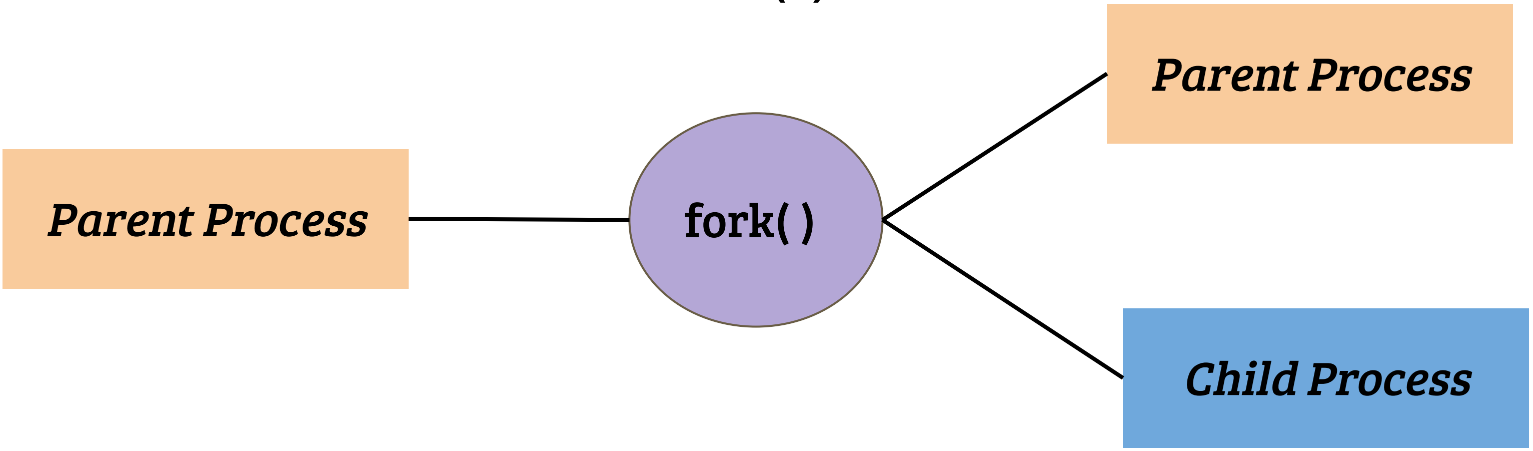
Debadatta Mishra, CSE, IIT Kanpur

Recap: Execution contexts in Linux


1
User process and threads (user mode)
2
User process and threads (kernel mode)
4
SoftIRQ handlers (kernel mode)
CPU
3
Kernel threads (kernel mode)
5
Interrupt handler (kernel mode)

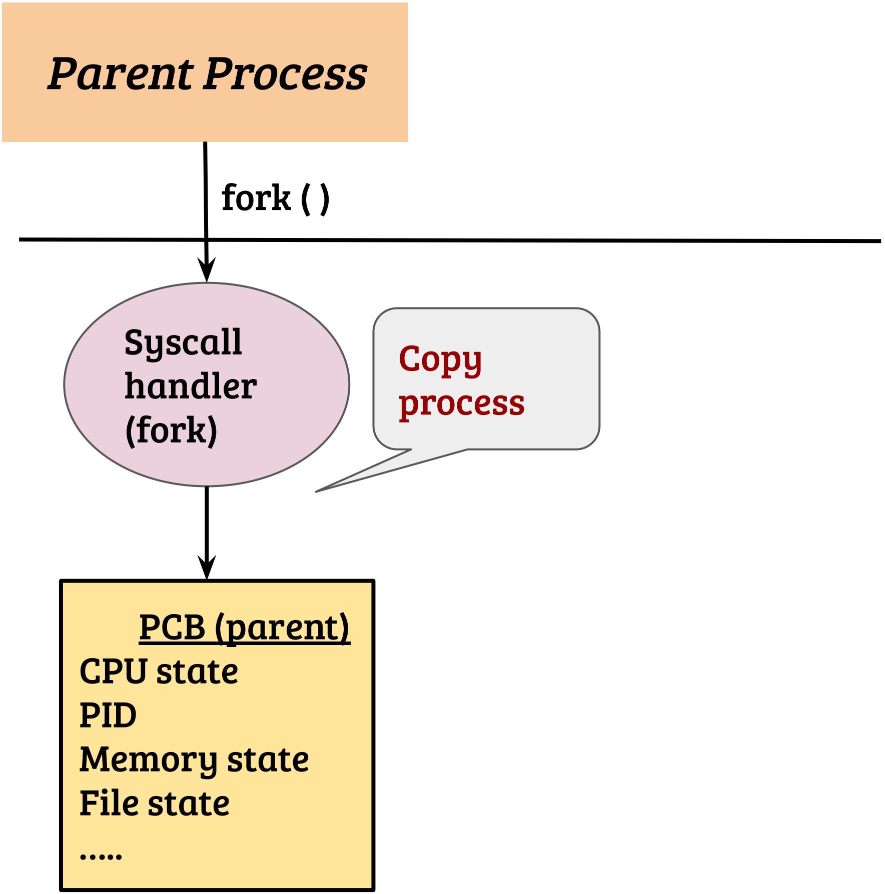
- In a linux system, the CPU can be executing in one of the above contexts
- For (3), (4) and (5), the context is not associated with any user process

Process creation - fork( )



- fork( ) system call is weird; not a typical “privileged” function call
- fork( ) creates a new process; a *duplicate* of calling process
- On success, fork
  - Returns PID of child process to the caller (parent)
  - Returns 0 to the child

Typical implementation of fork



Parent Process

fork ( )

Syscall handler (fork)

Copy process

PCB (parent)

CPU state

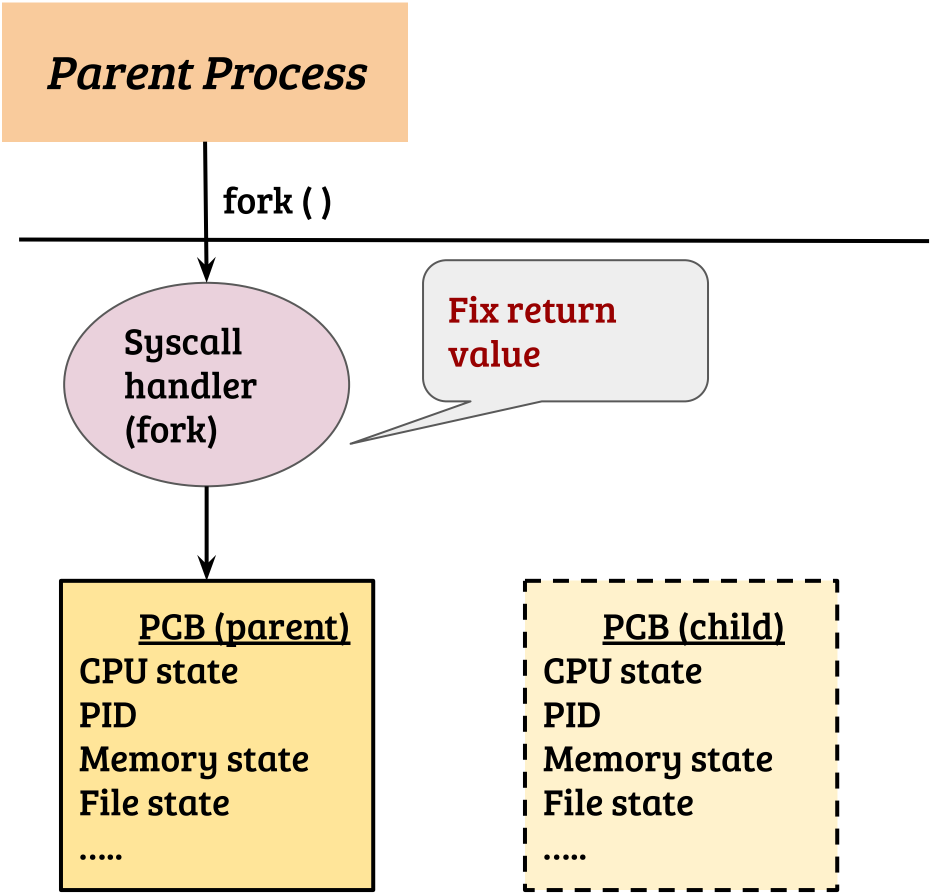
PID

Memory state

File state

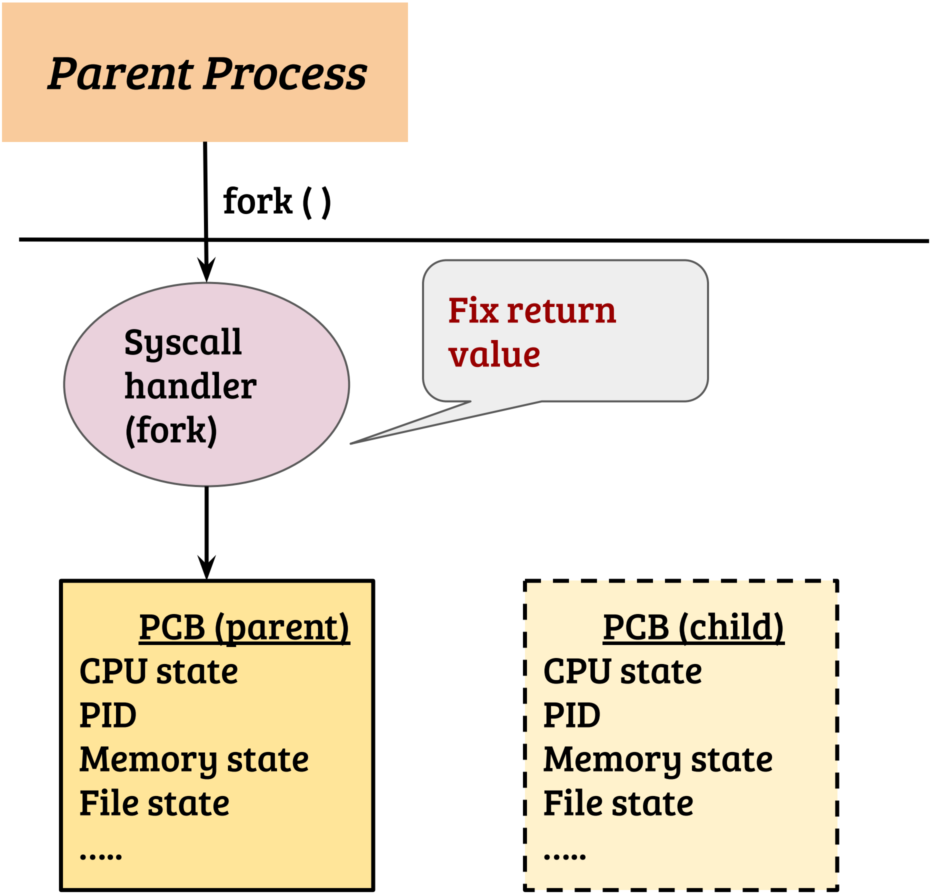
.....

Typical implementation of fork



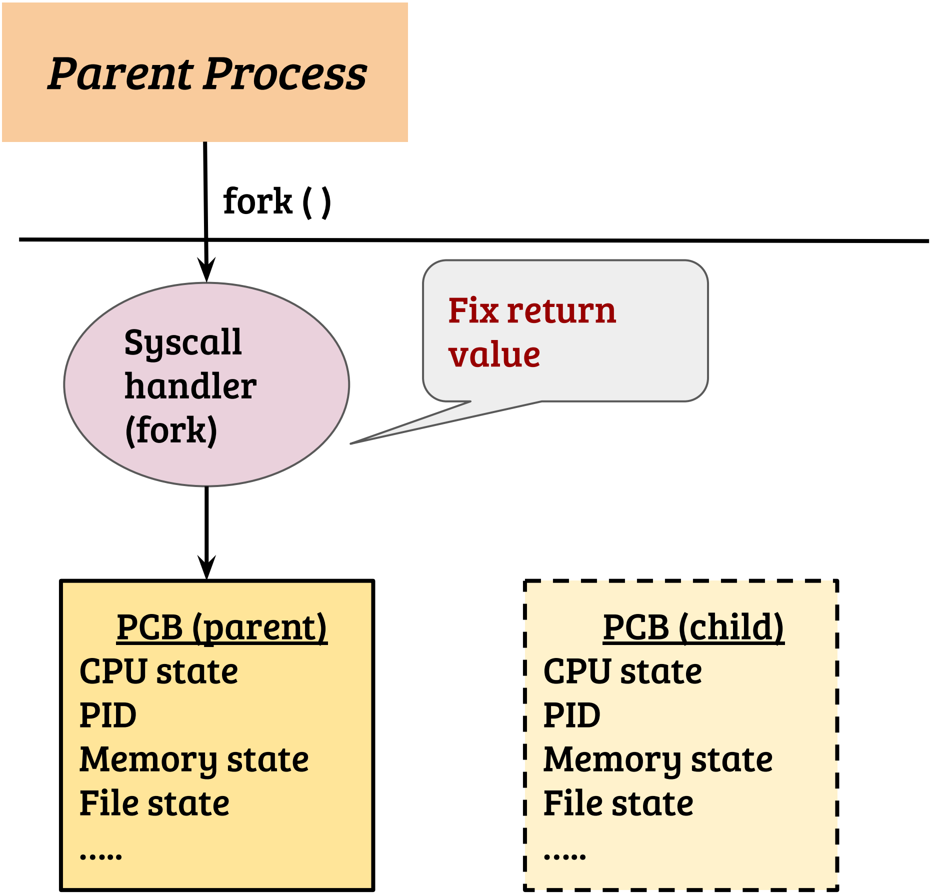
- Child should get ‘0’ and parent gets PID of child as return value. How?

Typical implementation of fork



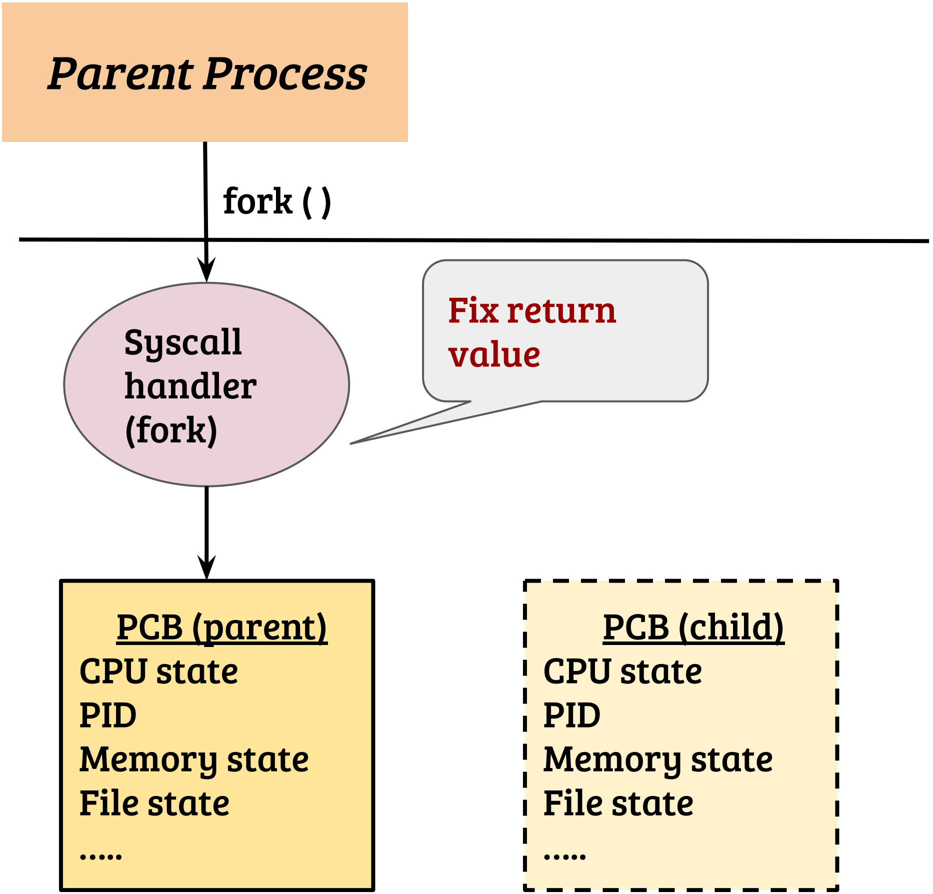
- Child should get ‘0’ and parent gets PID of child as return value. How?
- OS returns different values for parent and child

Typical implementation of fork



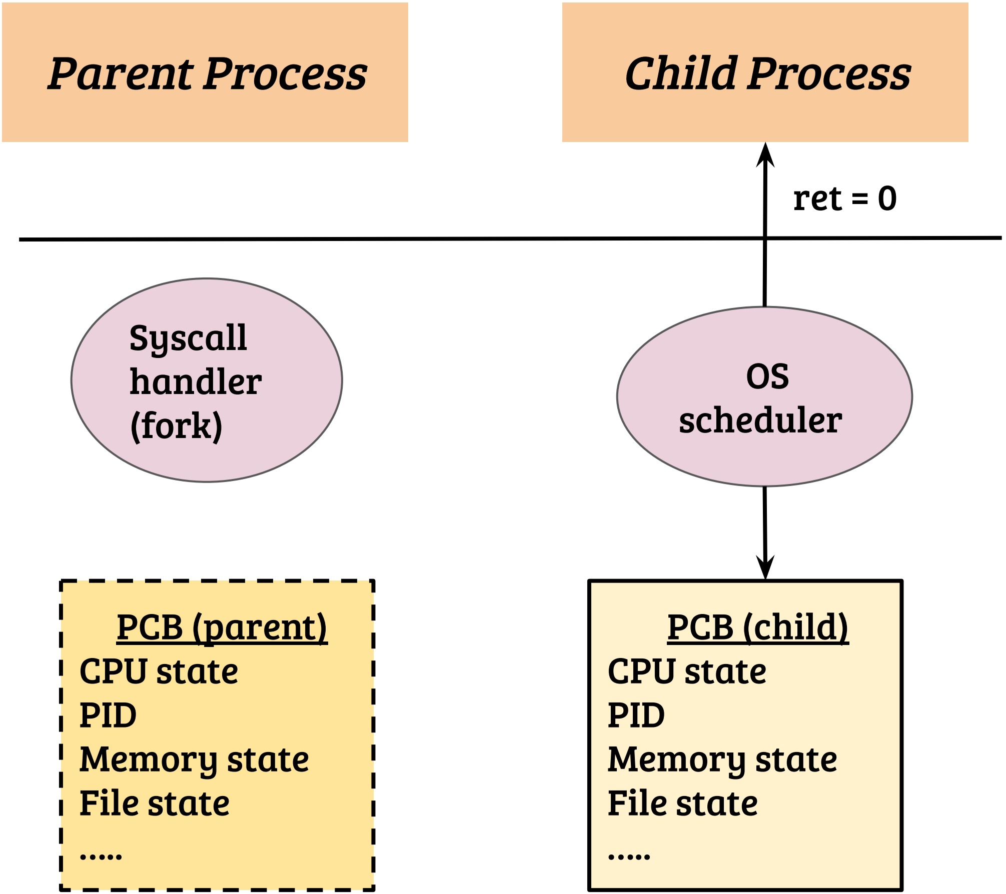
- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?

Typical implementation of fork



- Child should get '0' and parent gets PID of child as return value. How?
- OS returns different values for parent and child
- When does child execute?
- When OS schedules the child process

Typical implementation of fork



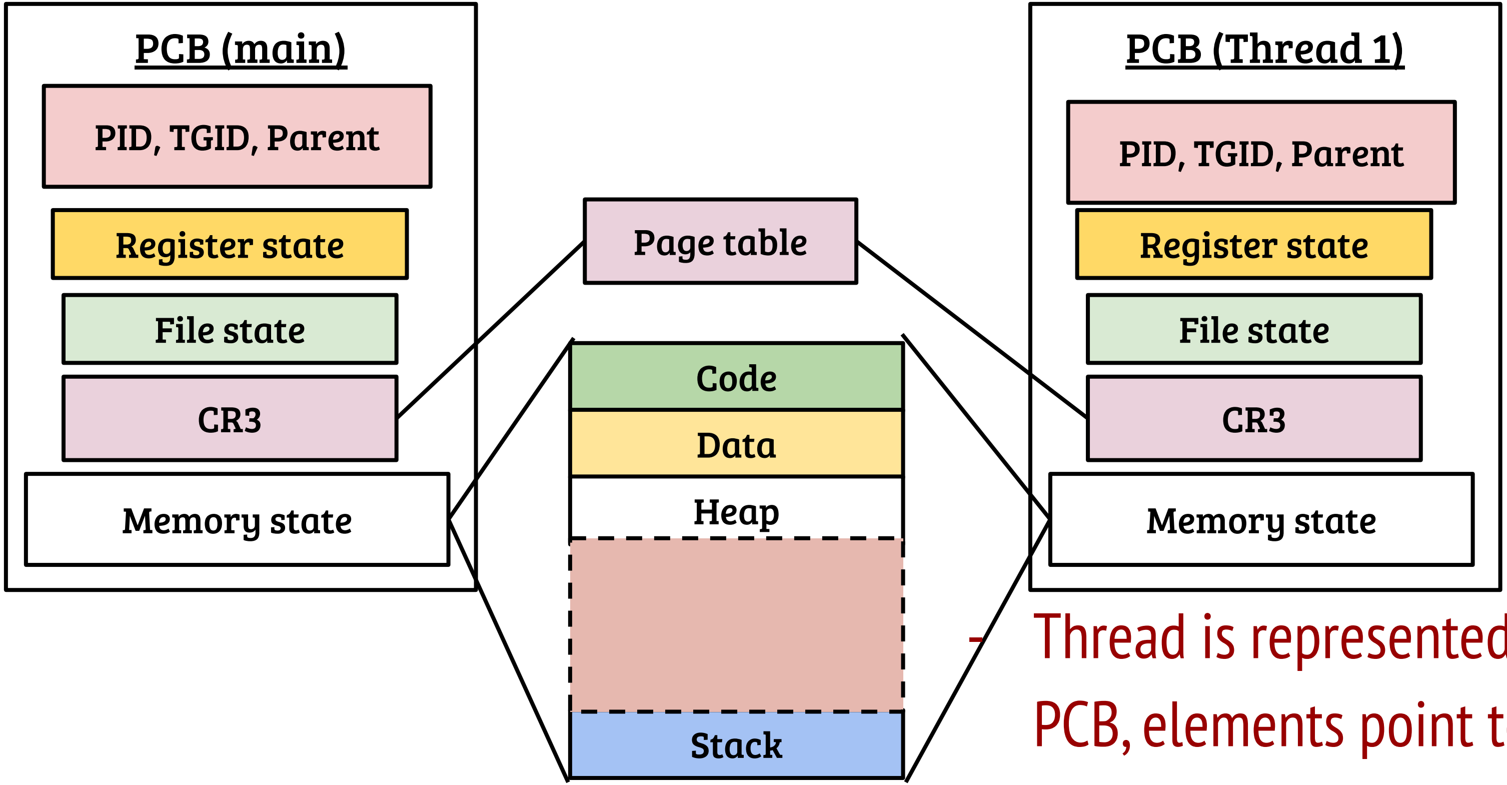
- PC is next instruction after fork( ) syscall, for both parent and child
- Child memory is an exact copy of parent
- Parent and child diverge from this point

User threads using posix thread API

```
int pthread_create( pthead_t *tid, pthread_attr_t *attr,
                    void * (*thfunc) (void*), void *arg);
```

- Creates a thread with “tid” as its handle and the thread starts executing the function pointed to by the “thfunc” argument
- A single argument (of type void *) can be passed to the thread
- Thread attribute can be used to control the thread behavior e.g., stack size, stack address etc. Passing NULL sets the defaults
- Returns 0 on success.
- Thread termination: return from thfunc, pthread_exit( ) or pthread_cancel( )
- In Linux, pthread_create and fork implemented using clone( ) system call

PCB of a multithreaded process (Linux)



- Thread is represented by a separate PCB, elements point to the structure containing subsystem level info.

The clone system call

int clone(int (*fn)(void *), void *child_stack, int flags, void *arg, ...)

- Parent can control the execution of new process (execution and stack)
- Provides flexibility to the parent to share parts of its execution context in a selective manner
- Examples flags:
  - CLONE_FILES: Share files between parent and new process
  - CLONE_VM: Share the address space
  - CLONE_VFORK: Execution of parent process is suspended

Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?

Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?
  - Syscall Handler ⟶ Kernel clone ⟶ Copy process
  - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?

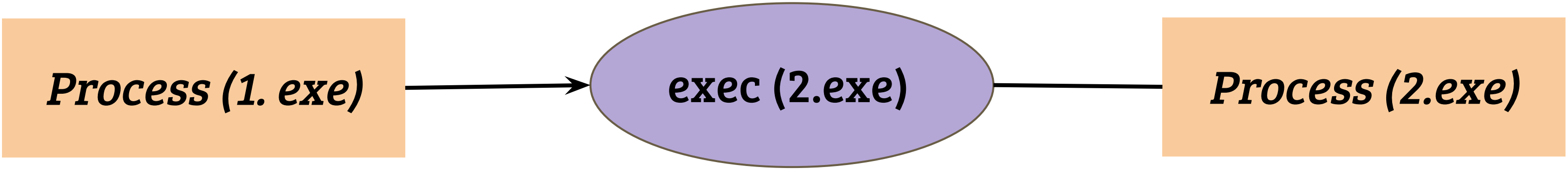
Clone: Implementation in Linux kernel

- Syscall handler for clone should provide flexible sharing. Implementation?
  - Syscall Handler ⟶ Kernel clone ⟶ Copy process
  - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?
  - For pthreads, the SP and RIP need to be changed
  - Change the register states during CPU thread copy
- Changes to the kernel space of newly created execution context required. Why? How implemented?

Clone: Implementation in Linux kernel

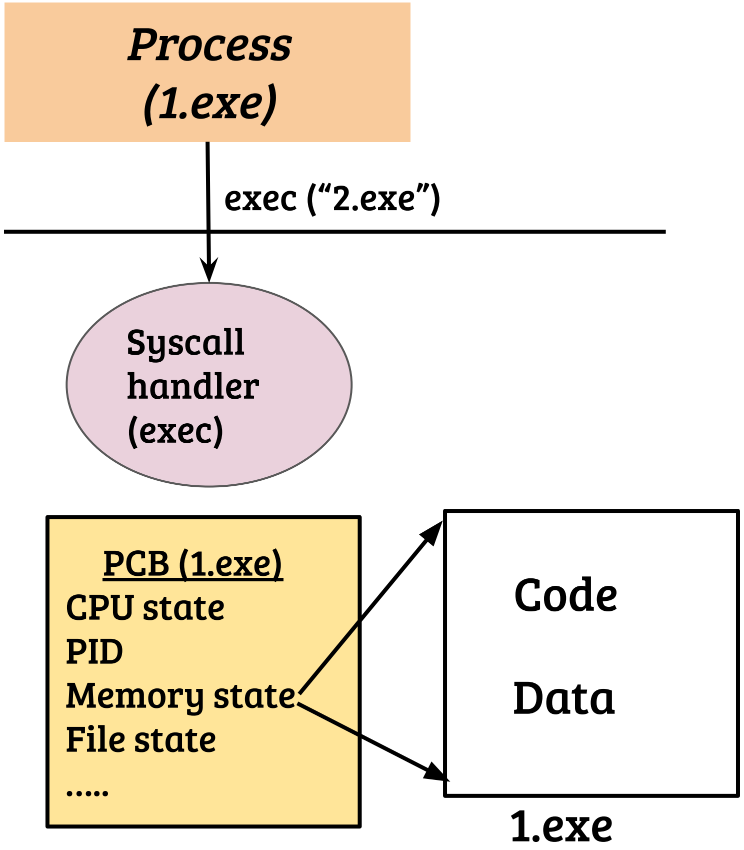
- Syscall handler for clone should provide flexible sharing. Implementation?
    - Syscall Handler → Kernel clone → Copy process
    - Depending on flags, different subsystems are copied or shared
- Depending on the usage, the saved user state is required to be changed. Why? How implemented?
    - For pthreads, the SP and RIP need to be changed
    - Change the register states during CPU thread copy
- Changes to the kernel space of newly created execution context required. Why? How implemented?
    - Child can not return in the same path, returns through a special stub

Load a new binary - exec( )



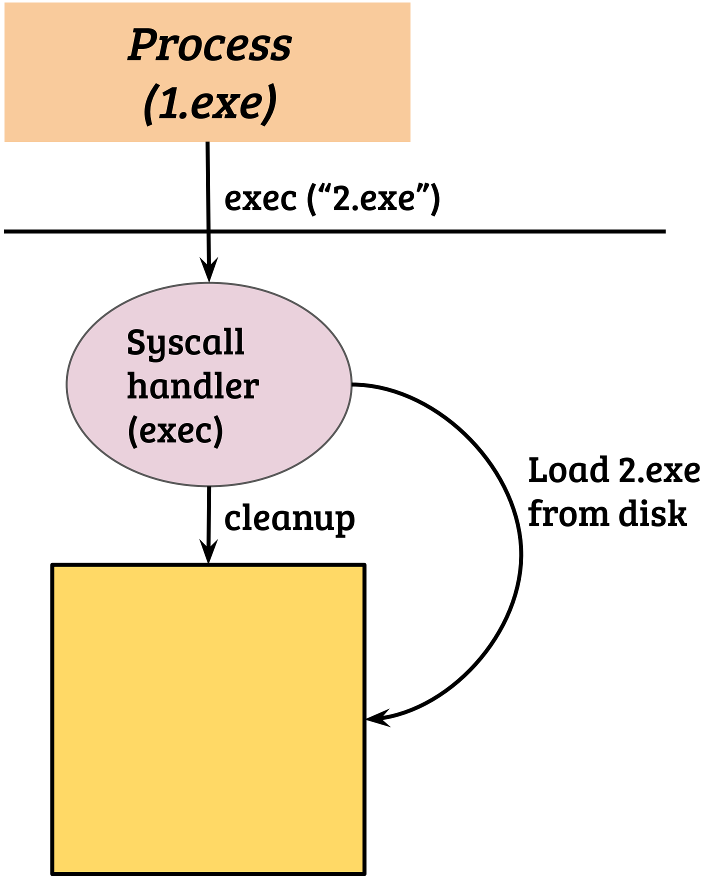
- Replace the calling process by a new executable
  - Code, data etc. are replaced by the new process
  - Usually, open files remain open

Typical implementation of exec



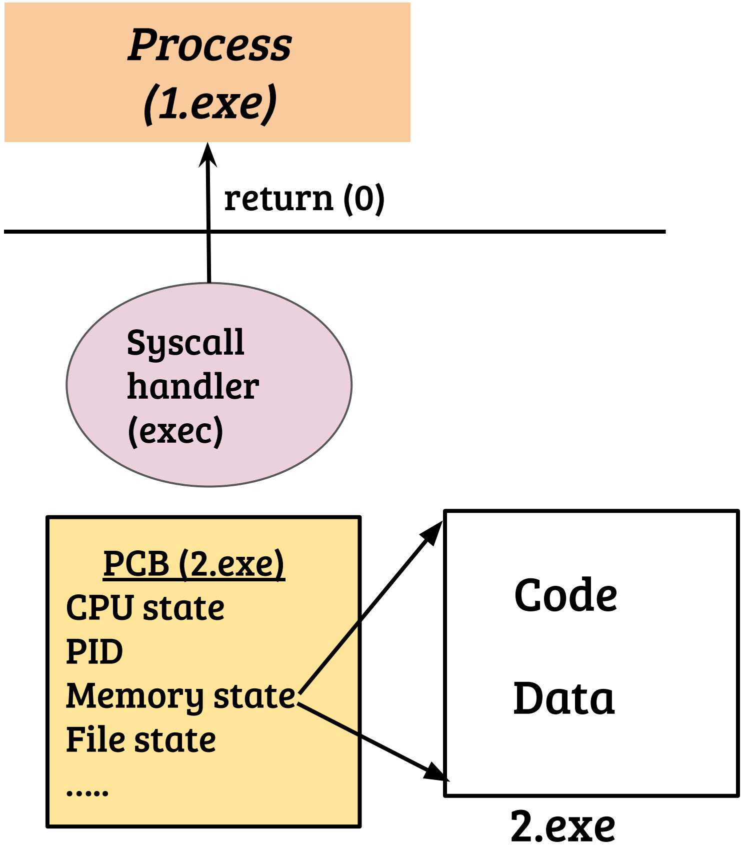
- The calling process commits self destruction! (almost)

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same

Typical implementation of exec



- The calling process commits self destruction! (almost)
- The calling process is cleaned up and replaced by the new executable
- PID remains the same
- On return, new executable starts execution
- PC is loaded with the starting address of the newly loaded binary

Exec: Implementation in Linux kernel

- When should the self destruction of address space take place? What are the design choices?

Exec: Implementation in Linux kernel

- When should the self destruction of address space take place? What are the design choices?
    - Can not destroy until validity is checked; validity check not complete until the binary/arguments are examined
    - Duplicated processing vs. working with a fresh (discardable) space
    - There would be a point of no return, delayed is better
- How does the kernel parse the binary (and deduce entry address)? What about command line arguments?

Exec: Implementation in Linux kernel

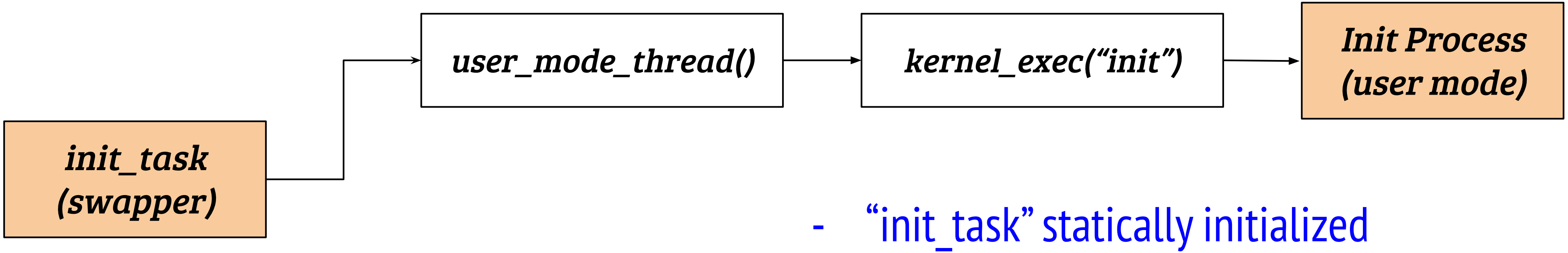
- When should the self destruction of address space take place? What are the design choices?
  - Can not destroy until validity is checked; validity check not complete until the binary/arguments are examined
  - Duplicated processing vs. working with a fresh (discardable) space
  - There would be a point of no return, delayed is better
- How does the kernel parse the binary (and deduce entry address)? What about command line arguments?
  - Basic binary parsing for ELF (and other types) e.g., load_elf_binary ( )
  - Command line arguments are placed in the stack

The first process

- What is the first execution entity in Linux?

The first process

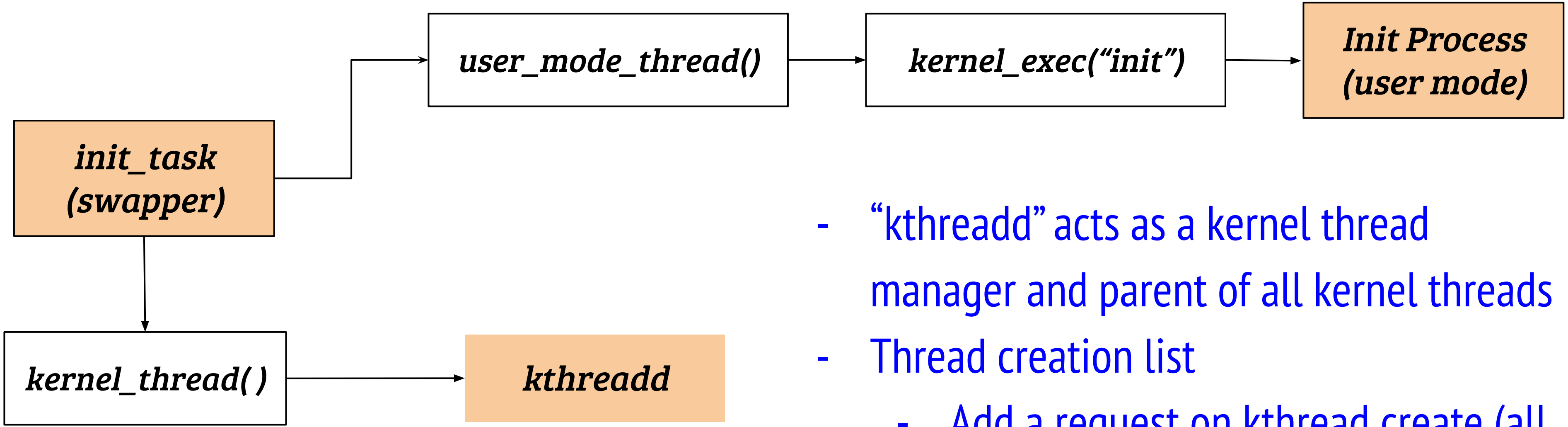
- What is the first execution entity in Linux?



- "init_task" statically initialized
- A special "clone" call from the kernel mode to create a thread of execution in kernel till actual init is executed
- Executes user space init based on configuration and default paths

The first process

- What is the first execution entity in Linux?



- "kthreadd" acts as a kernel thread manager and parent of all kernel threads
- Thread creation list
  - Add a request on kthread create (all types of kernel threads)
  - Wakeup kthreadd
  - Kthreadd → kernel_thread( )