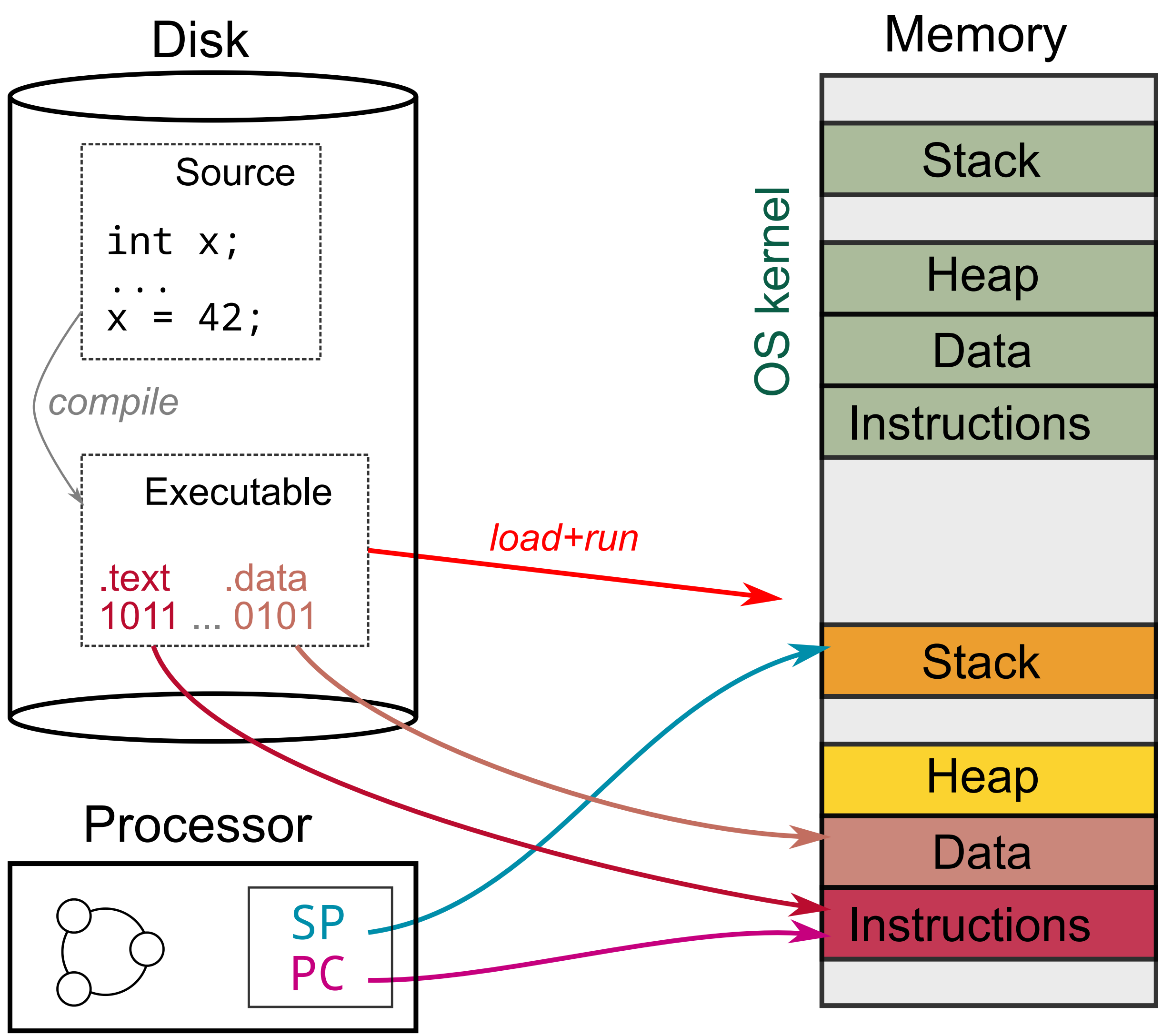
The Kernel Abstraction

Process abstraction

Process definition

A process is a program in execution



Process abstraction

Lack of protection

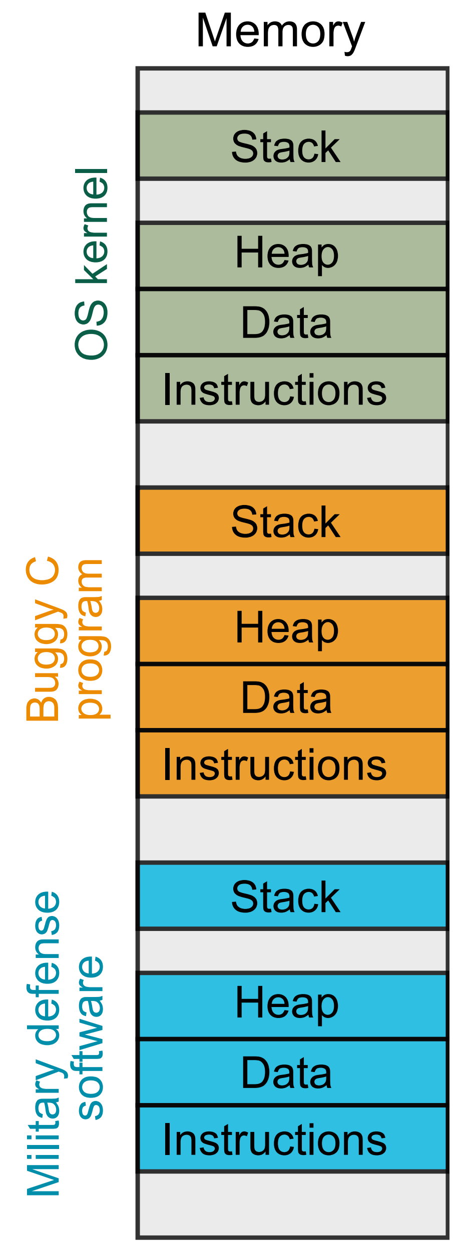
Multiple processes can be loaded in memory and run concurrently

Issues

- Buggy process
  - Crash other processes
  - Crash the OS
  - Hog all the resources
- Malicious process

Solution

- Redefine process abstraction
- Include notion of protection



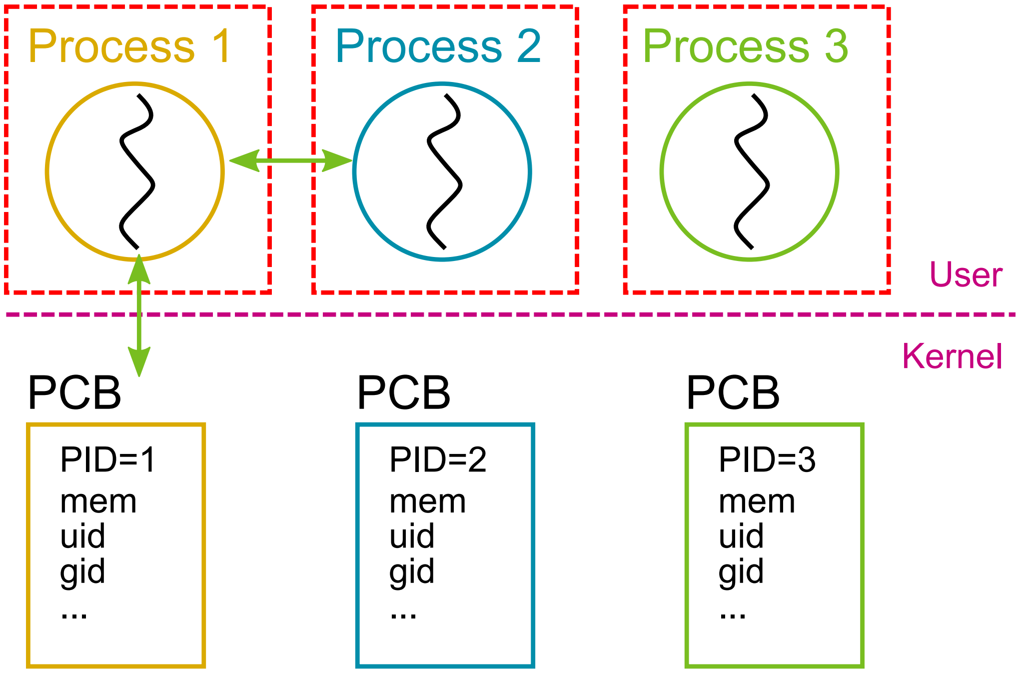
Process abstraction

Process RE-definition

A process is a program in execution, **running with limited rights**

Protected execution

- Memory segments process can access
- Other permissions process has
  - E.g., what files it can access
  - Based on process' user ID, group ID



But efficient

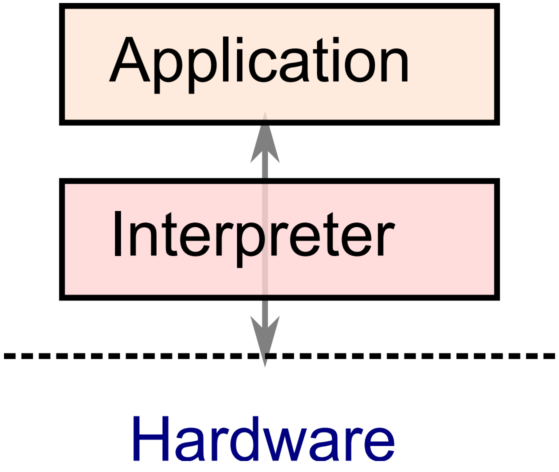
- Restricting rights must not hinder functionality
- Efficient use of hardware
- Communication with OS and between processes is safe

Process abstraction

Limited privilege execution

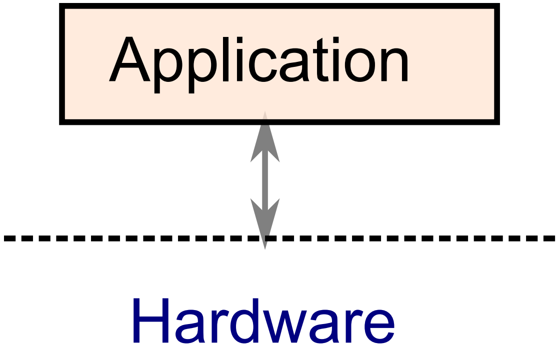
Interpreted execution

- Basic model in interpreted languages
  - Javascript, Python, etc.
- Emulate each program instruction
  - If instruction is permitted, then perform it
  - Otherwise, stop process
- But execution quite slow...



Native execution

- Run unprivileged code directly on the CPU
  - Very fast execution
- But safe execution needs specific hardware support...



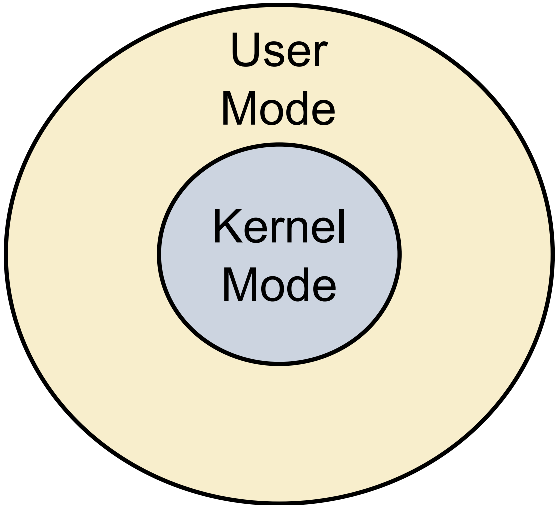
Dual-mode operation

Concept

- Distinct execution modes supported directly in hardware
  - Indicated by a bit in processor status register (e.g., 0 or 1)
  - Can be more than one mode on some processor architectures

Kernel mode

- Execution with full privileges on the hardware
  - Read/write to any memory location
  - Access to any I/O device
  - Read/write to any disk sector
  - Send/receive any packet
  - Etc.



User mode

- Limited privileges on the hardware
  - As granted by the operating system

Dual-mode operation

Hardware support

Privileged instructions

- Potentially unsafe instructions prohibited when in user mode
- Only available in kernel mode

Memory protection

- Memory accesses outside of process' memory limits prohibited
- Prevent process from overwriting kernel's or other processes' memory

Timer interrupts

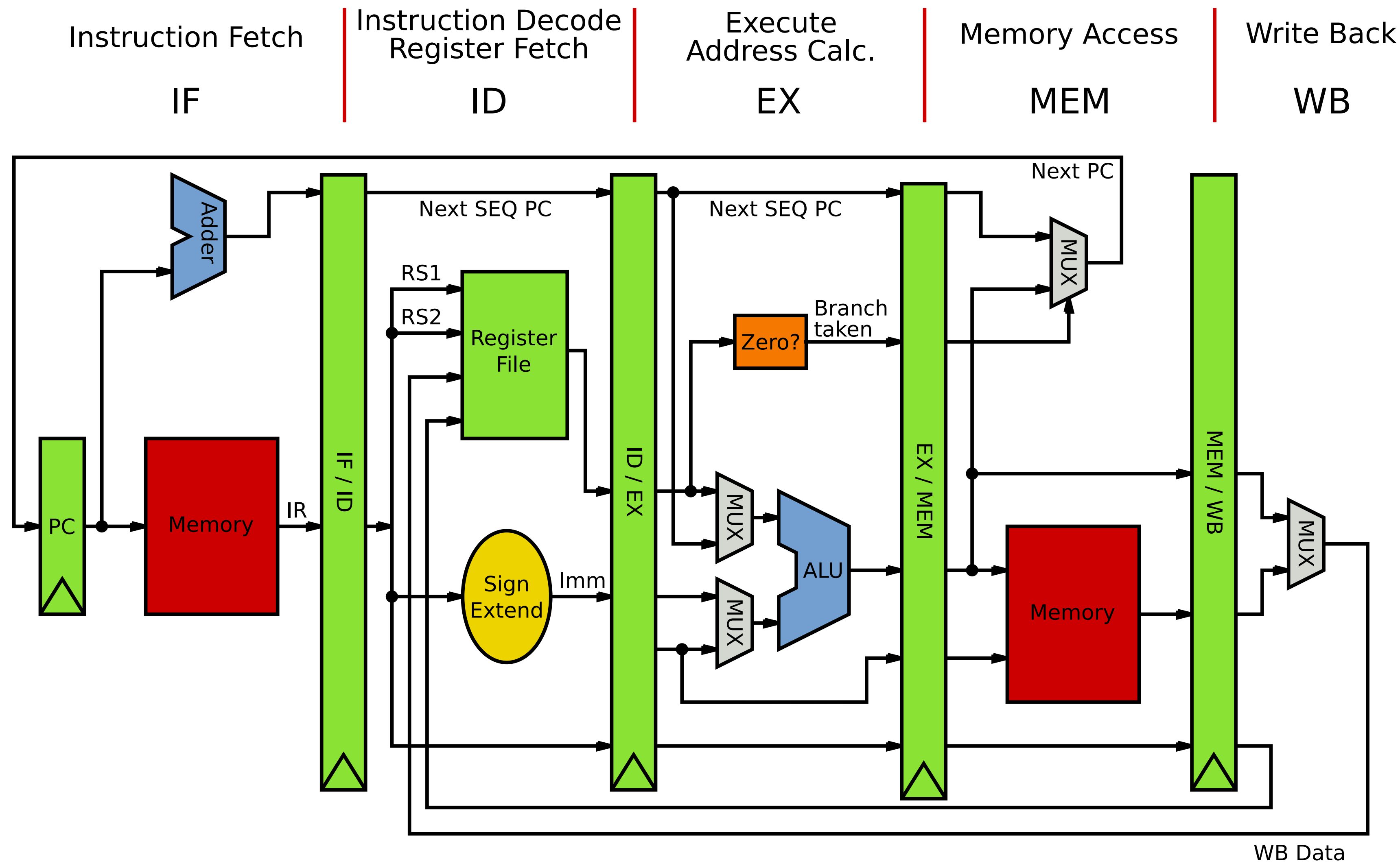
- Kernel periodically regains control on CPU
- Prevent running process from hogging hardware

Mode switch

- Safe and efficient way to switch mode
- From user mode to kernel mode, and vice-versa

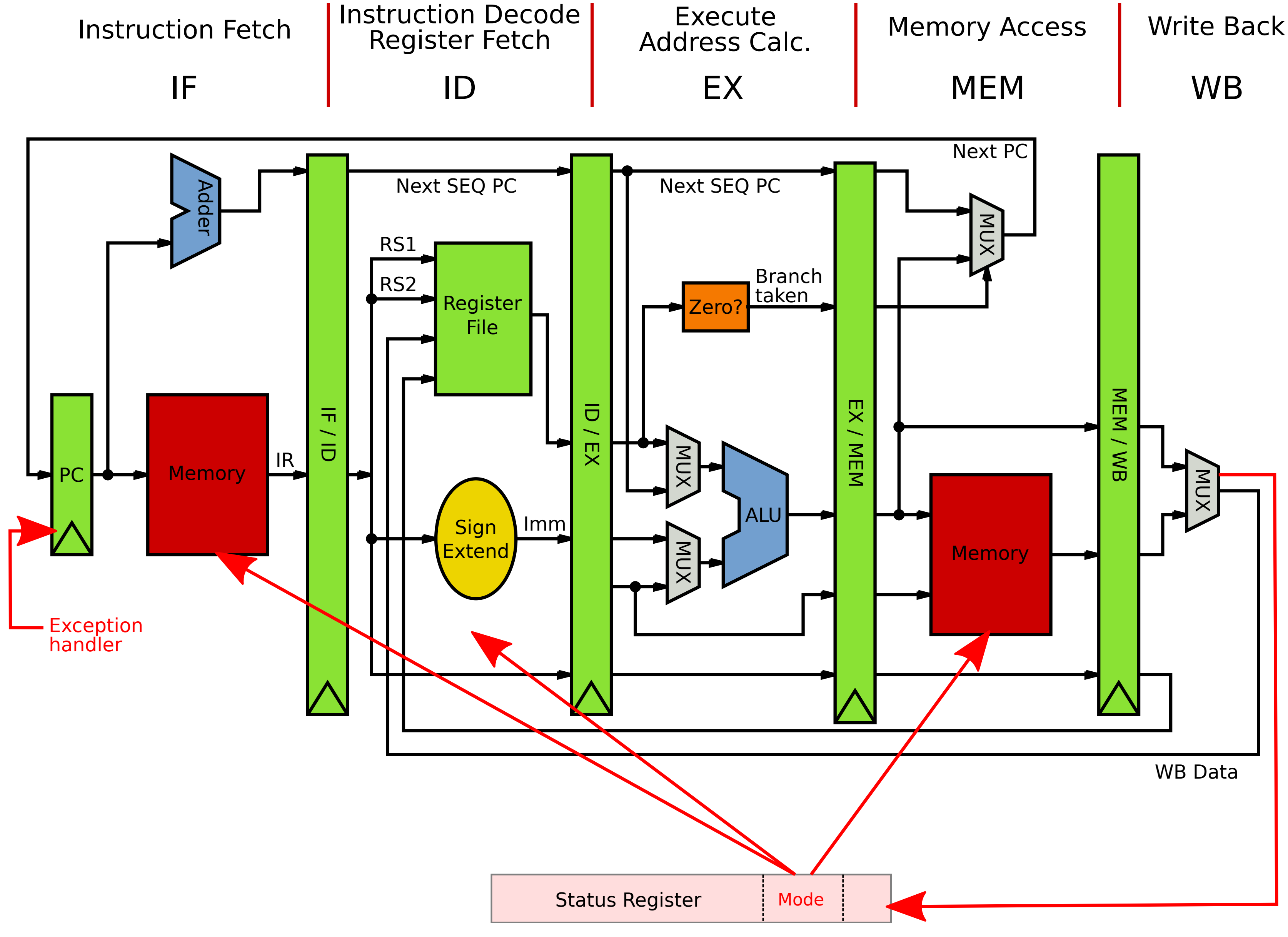
Dual-mode operation

Typical 5-stage pipeline



Dual-mode operation

"Dual-mode" 5-stage pipeline



ECS 150 - The Kernel Abstraction

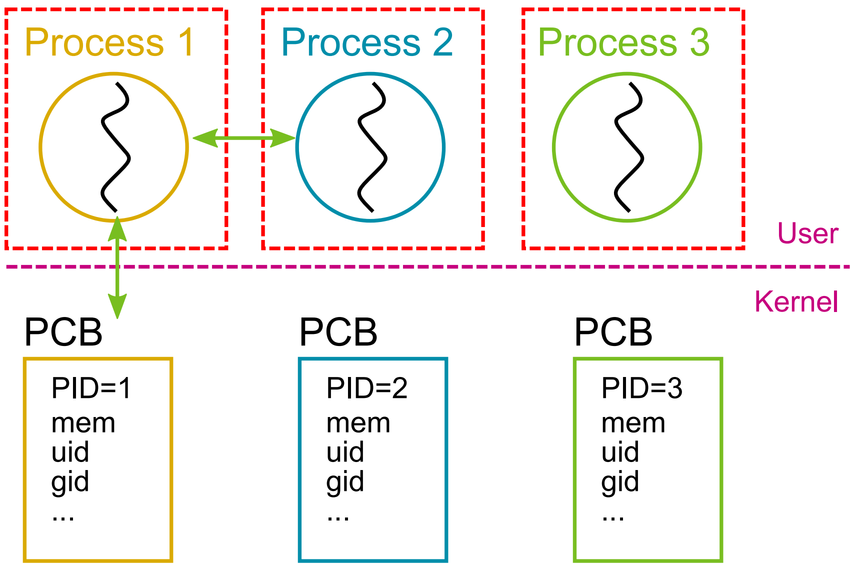
Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

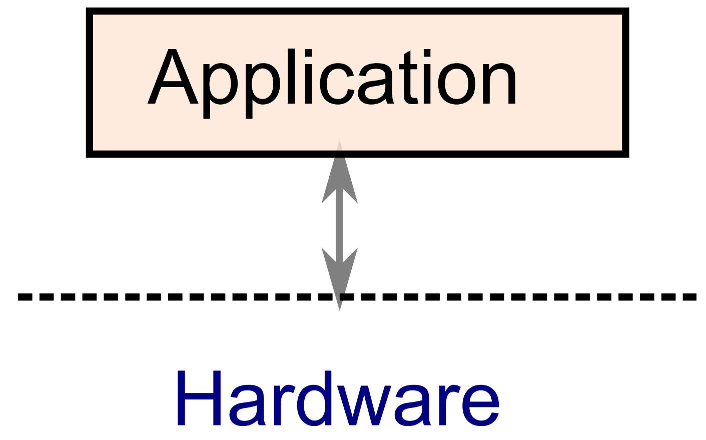
Recap

Process abstraction, v2.0

Protected execution

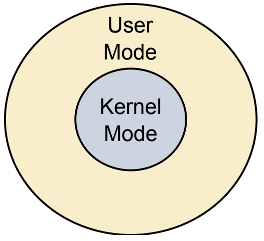


Native execution



Dual-mode operation

User mode vs kernel mode



Hardware support

- Privileged instructions
- Memory protection
- Timer interrupts
- Mode switch

Privileged instructions

Definition

- Instructions only available to code running in kernel mode
- Processor exception if user code tries to execute privileged instruction

Example

```c
int main(void)
{
    printf("Hello!\n");

    /* Try deactivating
     * hardware interrupts */
    asm ("cli" ::: "memory");

    while (1)
        printf("I win?\n");

    return 0;
}
```

x86_cli.c

```
$ ./x86_cli
Hello!
Segmentation fault (core dumped)
```

Illegal instructions are reported as segmentation faults on x86/Linux

And more...

- Toggle processor mode
- Modify memory protection
- Flush/invalidate caches/TLBs
- Halt or reset processor
- Etc.

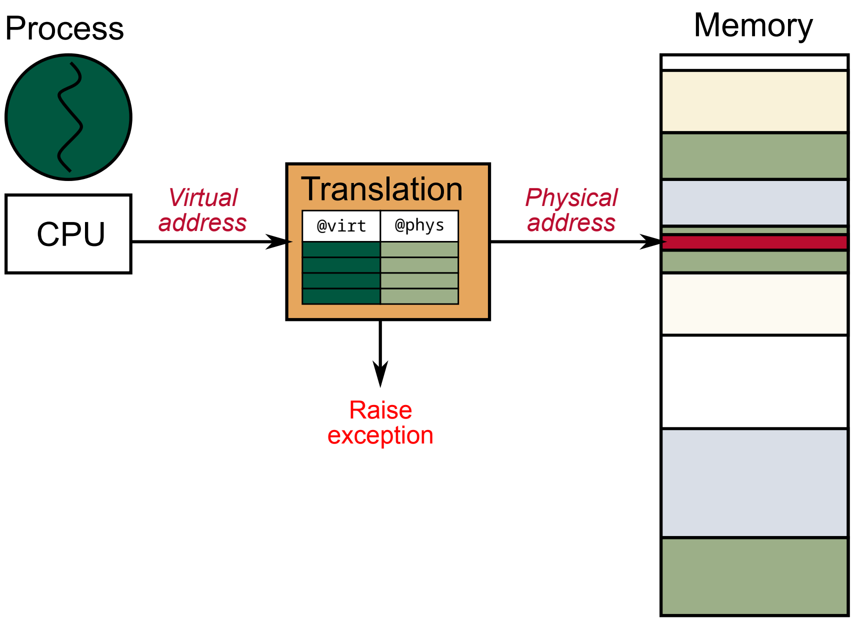
Memory protection

Concept

- Enforce memory boundaries to processes
- Processor exception if code tries to access unauthorized memory

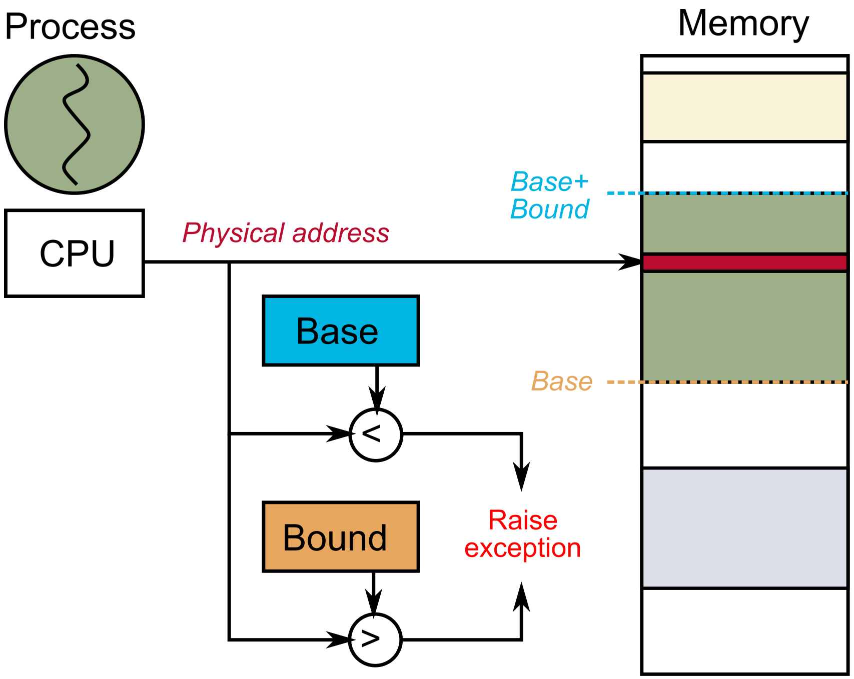
Virtual memory

- Runtime translation between virtual and physical address spaces



Basic segmentation

- Memory area defined by base and bound pair



Timer interrupts

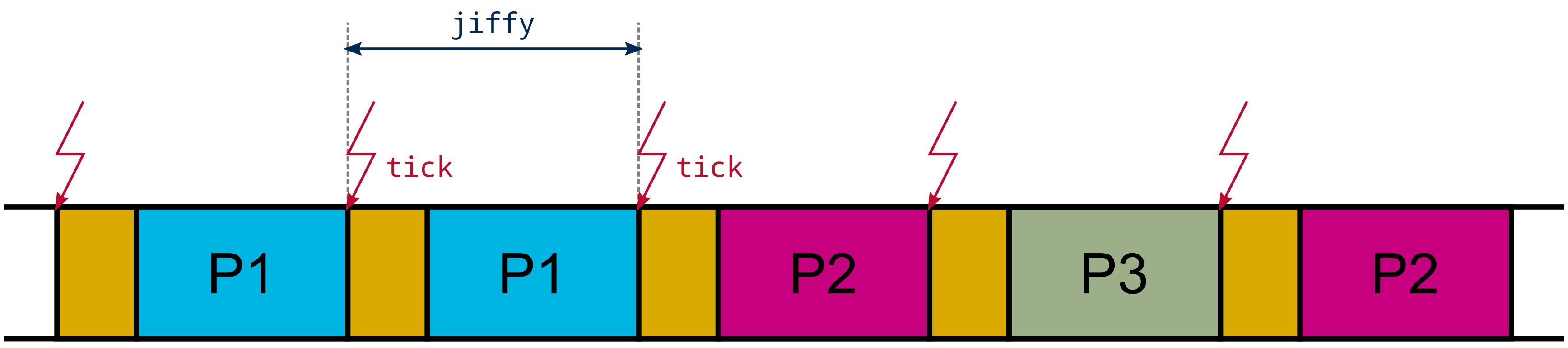
Boot sequence

- Upon powering on the computer
  - Privilege mode set to kernel mode
  - PC set to address of boot code (e.g., BIOS)
- Boot code runs
  - Loads kernel image into memory
  - Jumps to kernel's entry point
- Kernel code runs
  - Machine setup (devices, virtual memory, interrupt vector table, etc.)
  - Chooses the first *user* process to run, loads it, and jumps to it
    - Privilege bit set to user mode
    - PC set to process' entry point
- First process runs
  - Need a way for kernel to re-take control...

Timer interrupts

Hardware timer

- Periodically interrupts the processor
  - Frequency of interruption set by the kernel
  - Returns control to the kernel exception handler
- Also used to maintain accurate and precise time of day



Mode switching

User mode to kernel mode

Exceptions

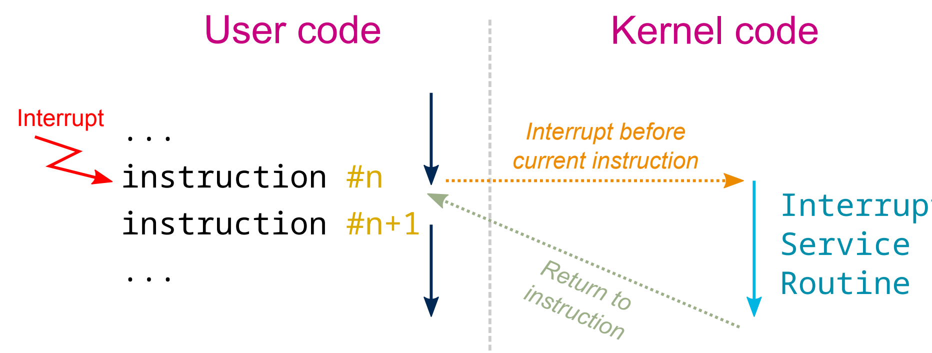
- Triggered by program behavior
- Intentional or unintentional
- Synchronous events

```
asm ("cli" ::: "memory");
```

```
int *a = NULL;
*a = 42;
```

Interrupts

- Triggered by I/O devices
- (Better) alternative to *polling*
- Asynchronous events



System calls

- Request from process for kernel to perform operation on its behalf
- Intentional, synchronous events

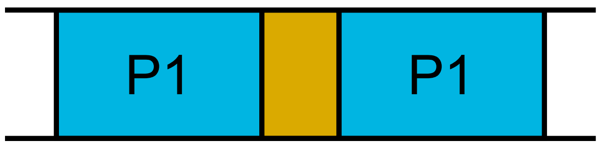
```
read:
    movq $SYS_read, rax
    movq $fd, rdi
    movq $buf, rsi
    ...
    syscall
```

Mode switching

Kernel mode to user mode

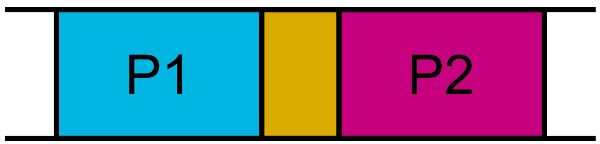
Return from interrupt or system call

- Resume suspended execution



Process context switch

- Resume some other process

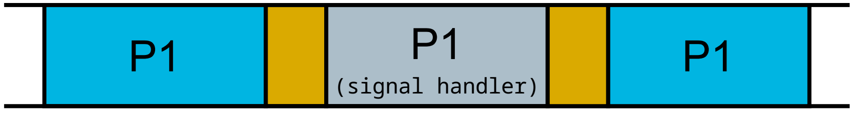


New process start

- Jump to first instruction in program

Signal

- Asynchronous notification
- If signal handler defined



ECS 150 - The Kernel Abstraction

Prof. Joël Porquet-Lupine

UC Davis - 2020/2021

Recap

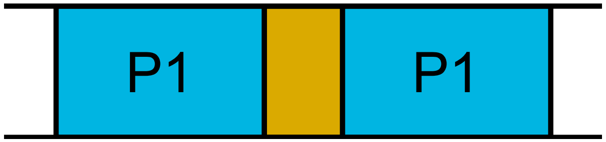
Mode switching

User to kernel

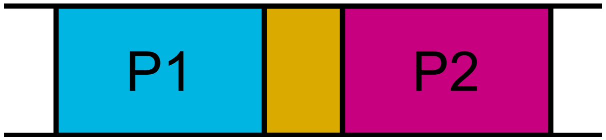
- Exceptions
  - Caused by program behavior
  - Synchronous
- Interrupts
  - Trigged by I/O devices
  - Asynchronous
- System calls
  - Service request to kernel
  - Synchronous

Kernel to user

- Return from interrupt or system call

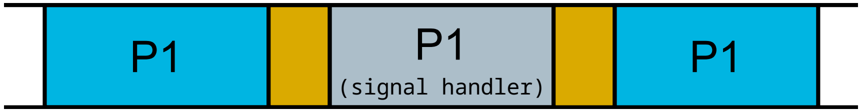


- Context switch between processes



- New process start

- Signal handler



Mode switching

Safe and efficient switching

- Protect from corrupting the kernel
  - Entry door to the kernel for processes
- Reduce overhead of kernel
  - Maximize CPU cycles for processes

Requirements

1. Atomic transfer of control
2. Exception vector
3. Transparent, restartable execution

Mode switching

Atomic transfer of control

- Safe transition between modes must be atomic
  - I.e., in one *unbreakable*, logical step
- CPU mode, PC, stack, memory protection, etc. changed at the same time

User to kernel switch

- Save cause for jump
  - Interrupt, Exception, Syscall?
- Save current PC
- Jump to kernel entry point(s)
- Switch from user to kernel mode
- Change memory protection
- Disable interrupts

Kernel to user switch

- Jump to process (restore PC)
- Switch from kernel to user mode
- Change memory protection
- Restore interrupts

Mode switching

Exception vector

- Provide limited number of entry points into the kernel
- Table set up by kernel: function pointers to exception handlers

Example MIPS processor

```
/* Exception handler init */
exception_handlers[0] = handle_int;
exception_handlers[8] = handle_sys;
exception_handlers[10] = handle_ri;
...
```

Software-managed

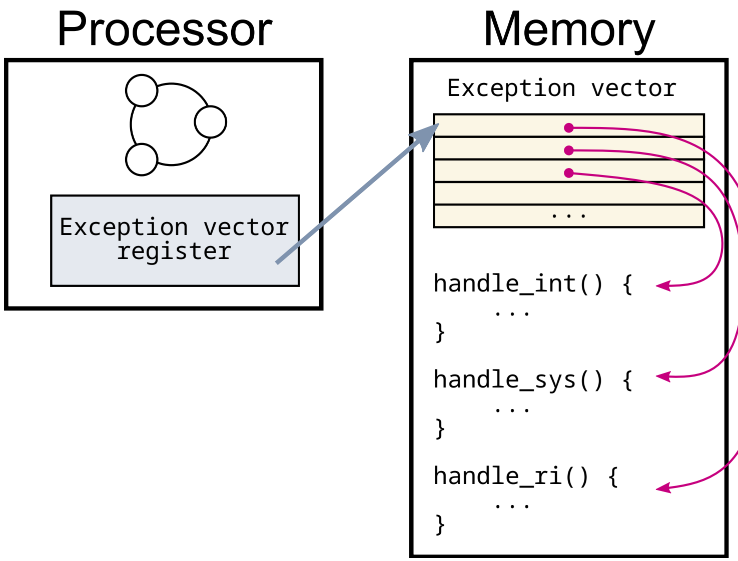
- Single kernel entry point for CPU
  - Fixed or configurable address
- Software dispatch based on exception cause

```
exception_vector:   # Kernel entry point
    # Retrieve cause from system register
    mfc0    k1, CP0_CAUSE
    # Extract exception code
    andi    k1, k1, 0x7c
    # Use code as index in array
    lw      k0, exception_handlers(k1)
    # Jump to proper handler
    jr      k0
```

Example MIPS processor

Hardware-managed

- CPU aware of vector
- Automatic hardware dispatch



Mode switching

Transparent, restartable execution

- Processes should never know when they are interrupted
- Save/restore execution context (processor registers)

Software-managed

```
handle_int:
    # Save all registers
    sw  $1, 4(sp)
    sw  $2, 8(sp)
    sw  $3, 12(sp)
...
    # Jump to C function
    # that processes interrupt requests
    jal do_irq

    # Restore all registers
...
    lw  $3, 12(sp)
    lw  $2, 8(sp)
    lw  $1, 4(sp)

    # Jump back to process
    eret
```

Example MIPS processor

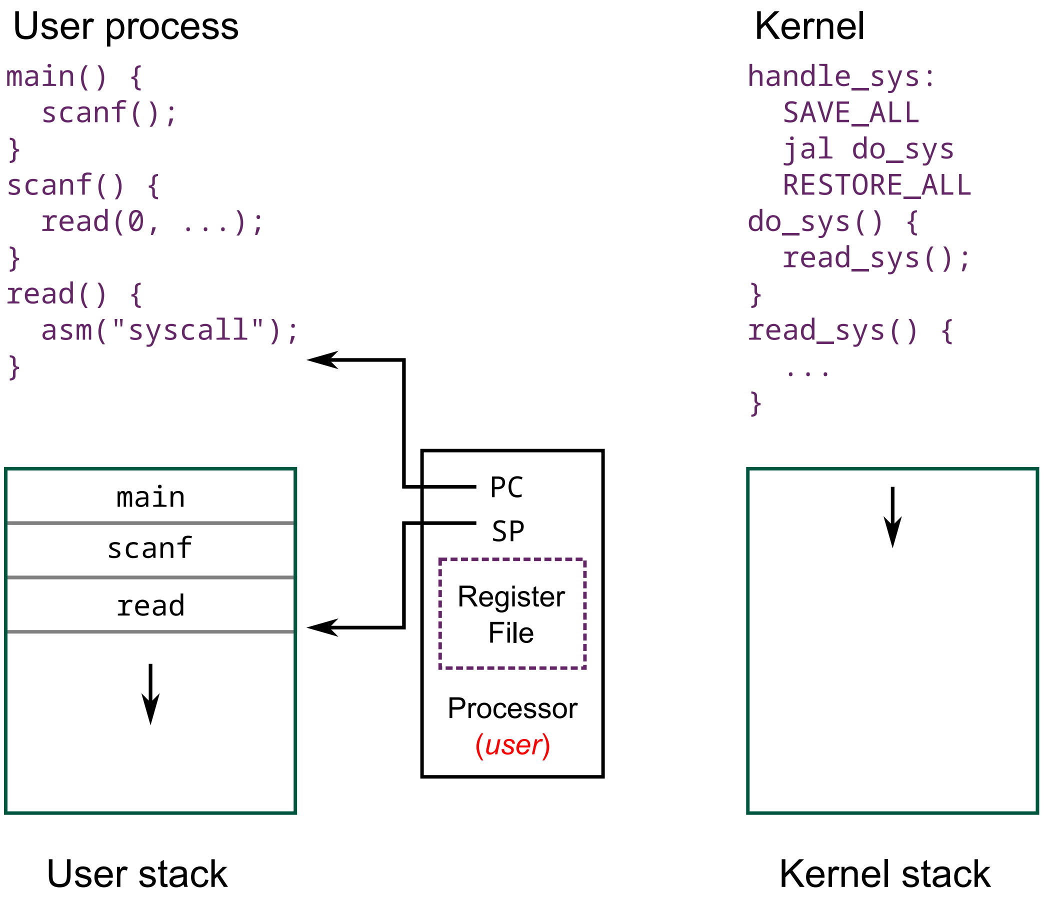
Hardware-managed

- Processor saves all the registers in a provided memory region
  - Task state segment (TSS) on x86 processors
- Rarely used in practice
  - E.g., not used by Linux or Windows

Kernel stack

Definition

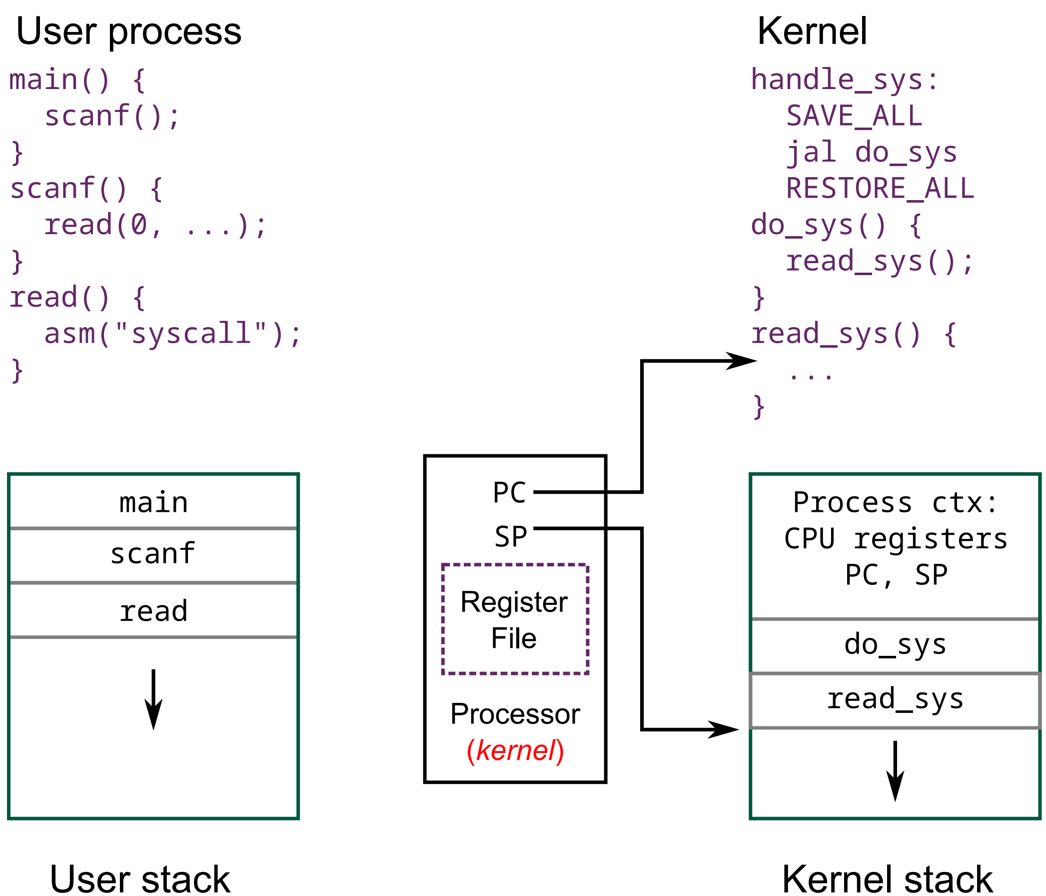
- Kernel has its own stack, located in kernel memory
- Different from process' stack



Kernel stack

Context saving

- Kernel stack is used to save associated process context



- Not a good idea to reuse process's stack pointer
  - Reliability: no guarantee user stack is valid
  - Security: kernel data shouldn't leak to user space

Kernel stack

One kernel stack per process

- Kernel saves its own state when switching between two processes