Operating Systems COMS(3010A) Kernels and Processes

Branden Ingram

branden.ingram@wits.ac.za

Office Number : ???

Recap

- What an OS is
- The Roles it plays
- Basic OS functionality
- The importance of OS
- OS Design similarities

Recap

- Responsible for
  - Making it easy to run programs
  - Allowing programs to share memory
  - Enabling programs to interact with devices

OS is in charge of making sure the system operates correctly and efficiently.

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

- **Protection was required to achieve some of the goals of OS**
    - **Reliability – ** preventing bugs from one program from affecting another

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

- **Protection was required to achieve some of the goals of OS**
  - **Reliability –** preventing bugs from one program from affecting another
  - **Security –** preventing malicious software or users from infecting your system

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

- **Protection was required to achieve some of the goals of OS**
    - **Reliability** – preventing bugs from one program from affecting another
    - **Security –** preventing malicious software or users from infecting your system
    - **Privacy –** preventing users from accessing unauthorized data

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

- **Protection was required to achieve some of the goals of OS**
    - **Reliability** – preventing bugs from one program from affecting another
    - **Security –** preventing malicious software or users from infecting your system
    - **Privacy –** preventing users from accessing unauthorized data
    - **Fair resource allocation –** preventing applications from hogging resources

The Kernel

- Protection – The isolation of potentially misbehaving applications and users so that they do not corrupt other applications or the OS itself.

- Protection was required to achieve some of the goals of OS
    - Reliability – preventing bugs from one program from affecting another
    - Security – preventing malicious software or users from infecting your system
    - Privacy – preventing users from accessing unauthorized data
    - Fair resource allocation – preventing applications from hogging resources

- **Implementing protection is the job of an OS's Kernel**

The Kernel

- The kernel is the central **module** of an operating system (OS) responsible for **memory management**, **process** and **task management**, and **disk management**.

The Kernel

• The kernel is the central module of an operating system (OS) responsible for memory management, process and task management, and disk management.

• **It is the part of the operating system that loads first, and it remains in main memory.**

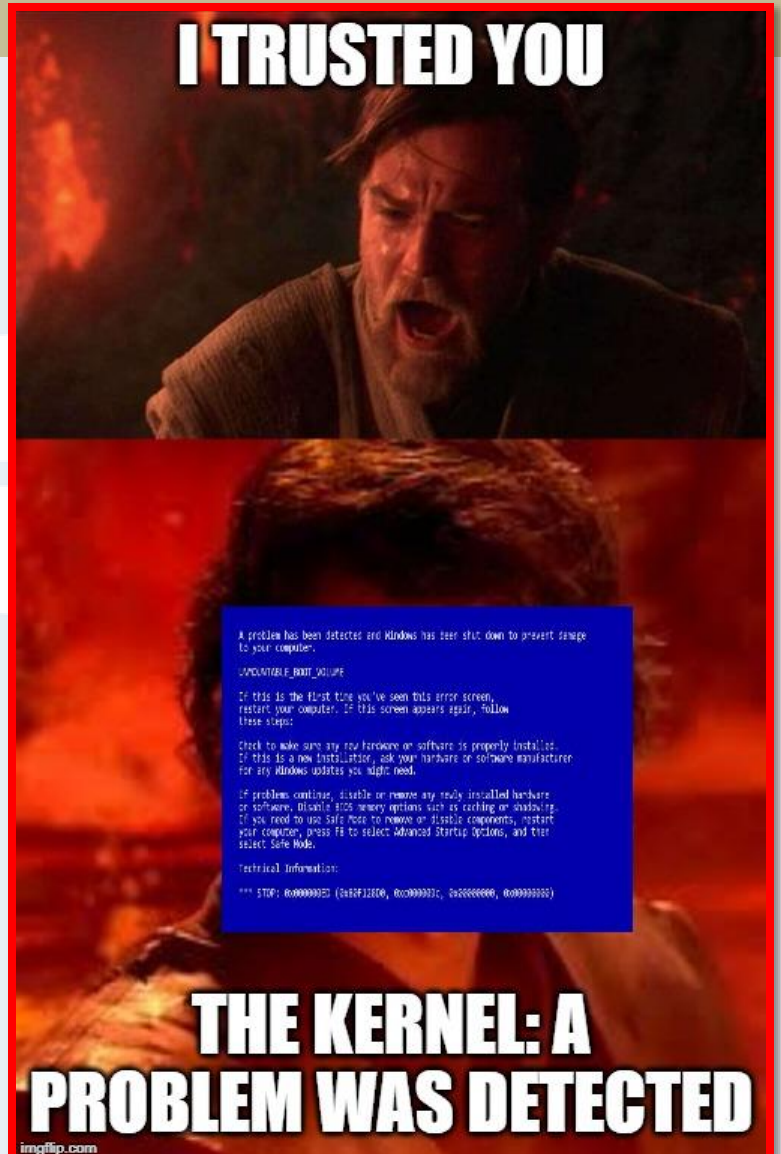
The Kernel

- The kernel is the central module of an operating system (OS) responsible for memory management, process and task management, and disk management.

- **It is the part of the operating system that loads first, and it remains in main memory.**

The Kernel

- The Kernel is the **lowest level** of software running on the system
- Has **full access** to all of the machine hardware
- "**Trusted**" to do anything with the hardware



The Kernel

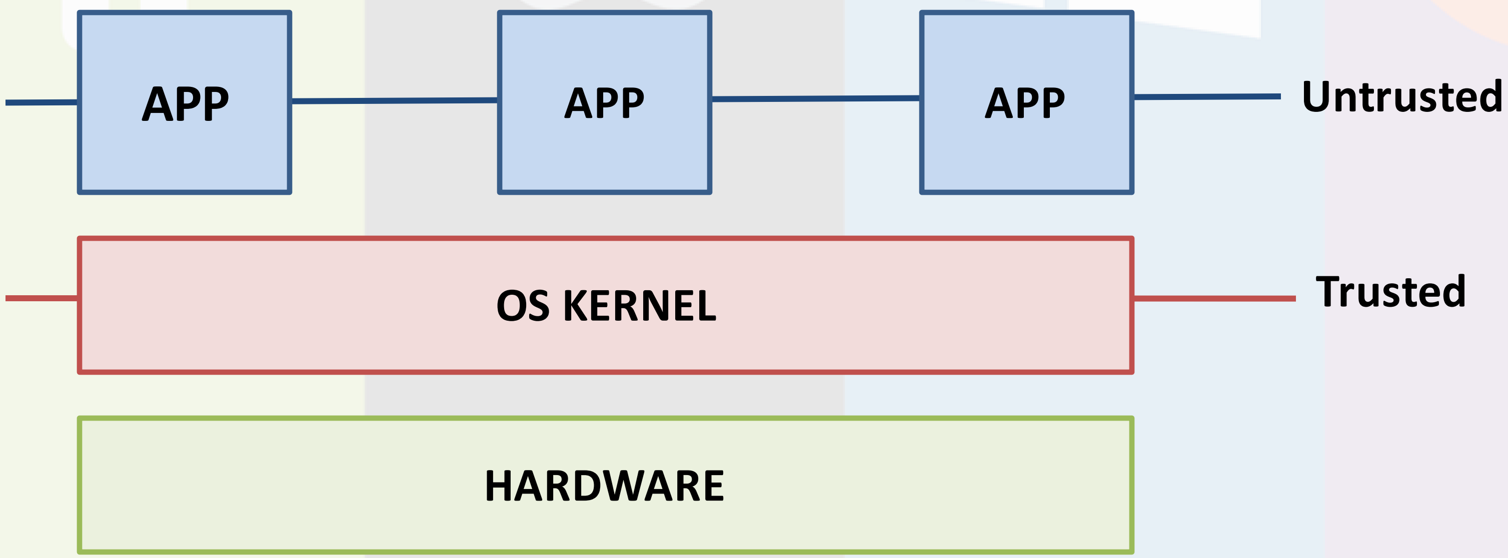
- The Kernel is the lowest level of software running on the system
- Has full access to all of the machine hardware
- "Trusted" to do anything with the hardware

- **Everything else – considered "Untrusted"**
    - **This software is run in a restricted environment with less than complete access to the full power of the hardware**

The Kernel

- The Kernel is the lowest level of software running on the system
- Has full access to all of the machine hardware
- "Trusted" to do anything with the hardware

- **Everything else – considered "Untrusted"**
  - **This software is run in a restricted environment with less than complete access to the full power of the hardware**



Do applications need to implement protection?

Do applications need to implement protection?

- Of course

Do applications need to implement protection?

- Of course

- **Just like the OS needs to safely execute untrusted software, so too does an application**
- **Web browser executing embedded JavaScript to draw a webpage**

Do applications need to implement protection?

- Of course

- Just like the OS needs to safely execute untrusted software, so too does an application
- Web browser executing embedded JavaScript to draw a webpage

- **Without protection an embedded virus could be forwarding keystrokes' to an attacker**

Virtualization

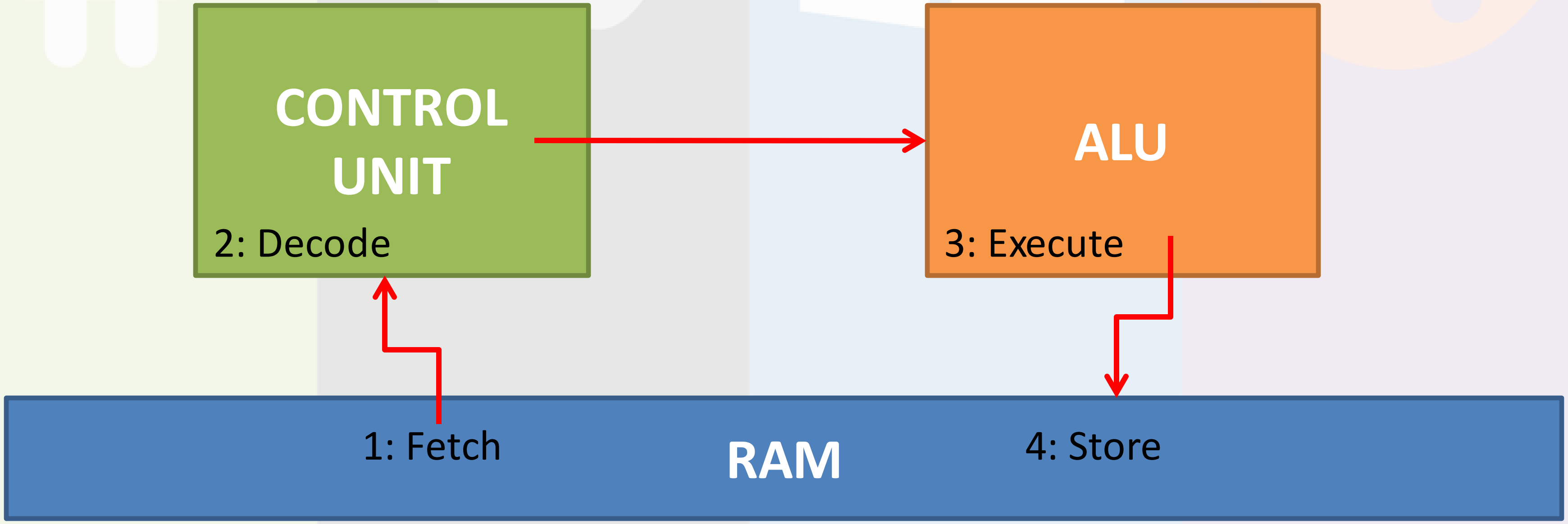
- The OS takes a physical resource and transforms it into a virtual form of itself.
  - Physical resource: Processor, Memory, Disk …

Virtualization

- The OS takes a physical resource and transforms it into a virtual form of itself.
    - Physical resource: Processor, Memory, Disk …
- The virtual form is more general, powerful and easy-to-use.
- Sometimes, we refer to the OS as a virtual machine.

Running a Program

- **A running program executes instructions.**
  - The processor **fetches** an instruction from memory.
  - **Decode**: Figure out which instruction this is
  - **Execute**: i.e., add two numbers, access memory, check a condition, jump to function, and so forth.
  - The processor moves on to the **next instruction** and so on.



A Process

- The execution of an application program with restricted rights
  - The abstraction for protected execution provided by the OS

A Process

- The execution of an application program with restricted rights
  - The abstraction for protected execution provided by the OS

- **The process requires permission from the OS kernel to:**
  - access memory of other processes
  - read and write to disk
  - change hardware settings

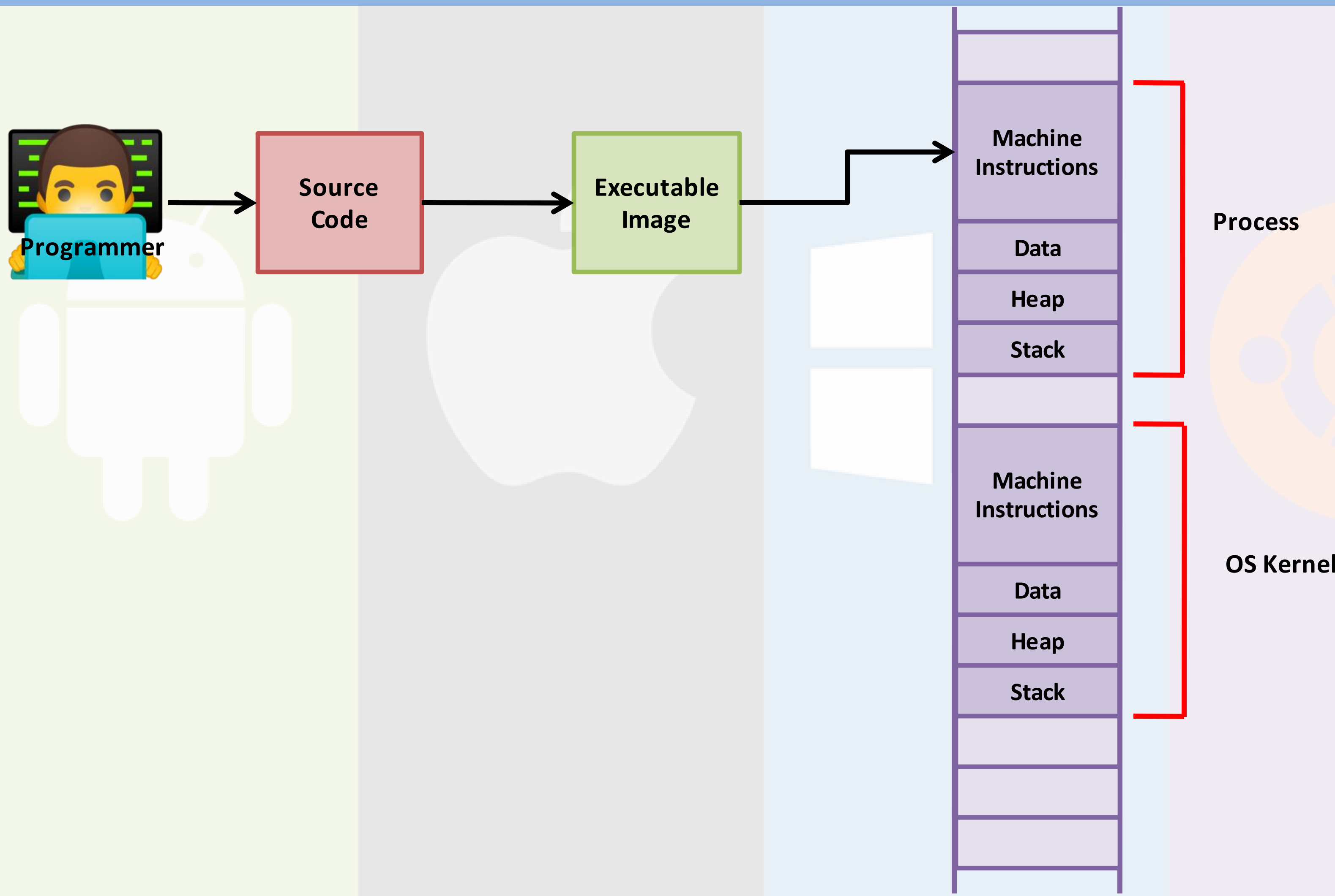
A Process

- The execution of an application program with restricted rights
  - The abstraction for protected execution provided by the OS

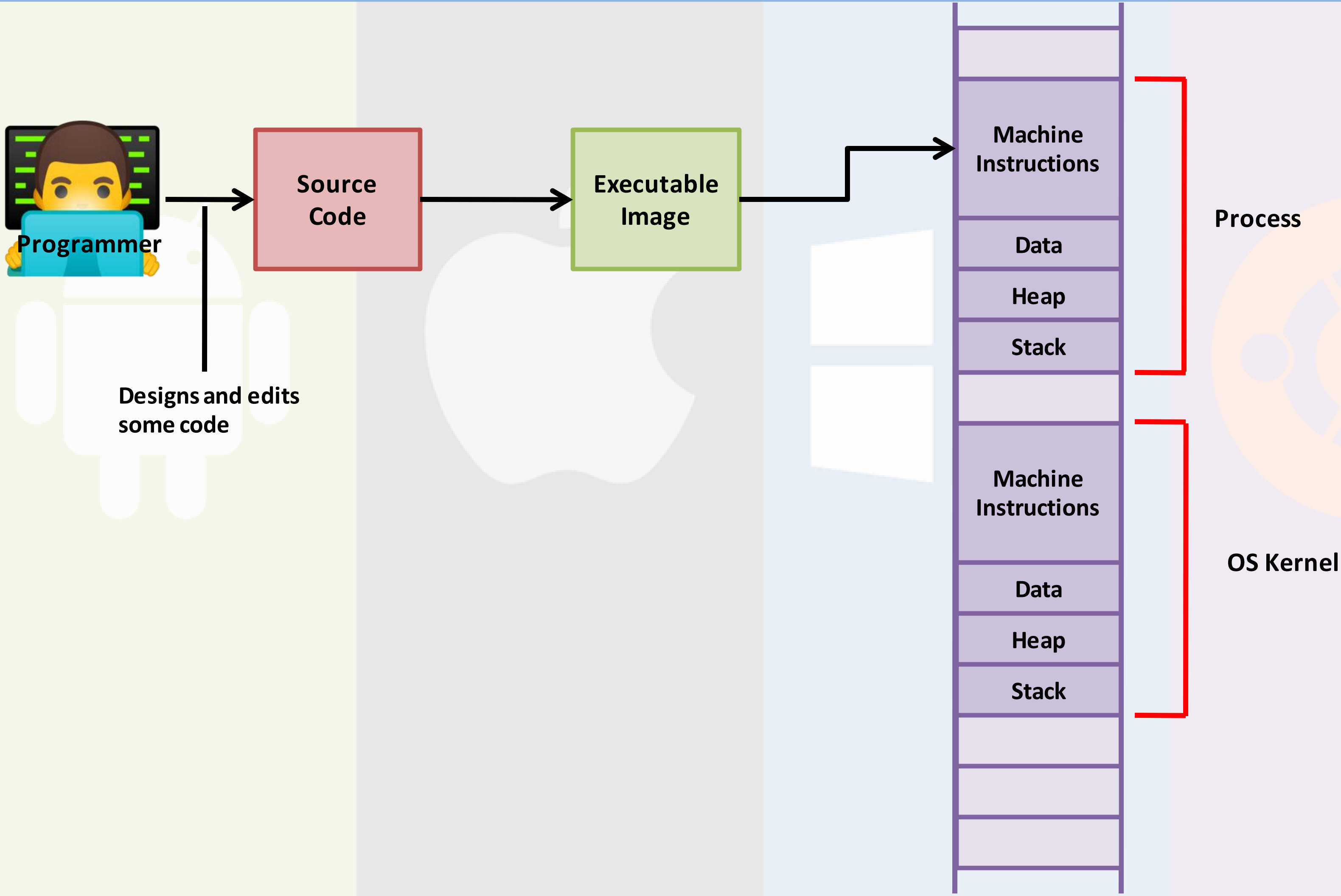
- The process requires permission from the OS kernel to:
  - access memory of other processes
  - read and write to disk
  - change hardware settings

- **Once again it's the idea of the OS kernel mediating and checking a processes access to hardware**

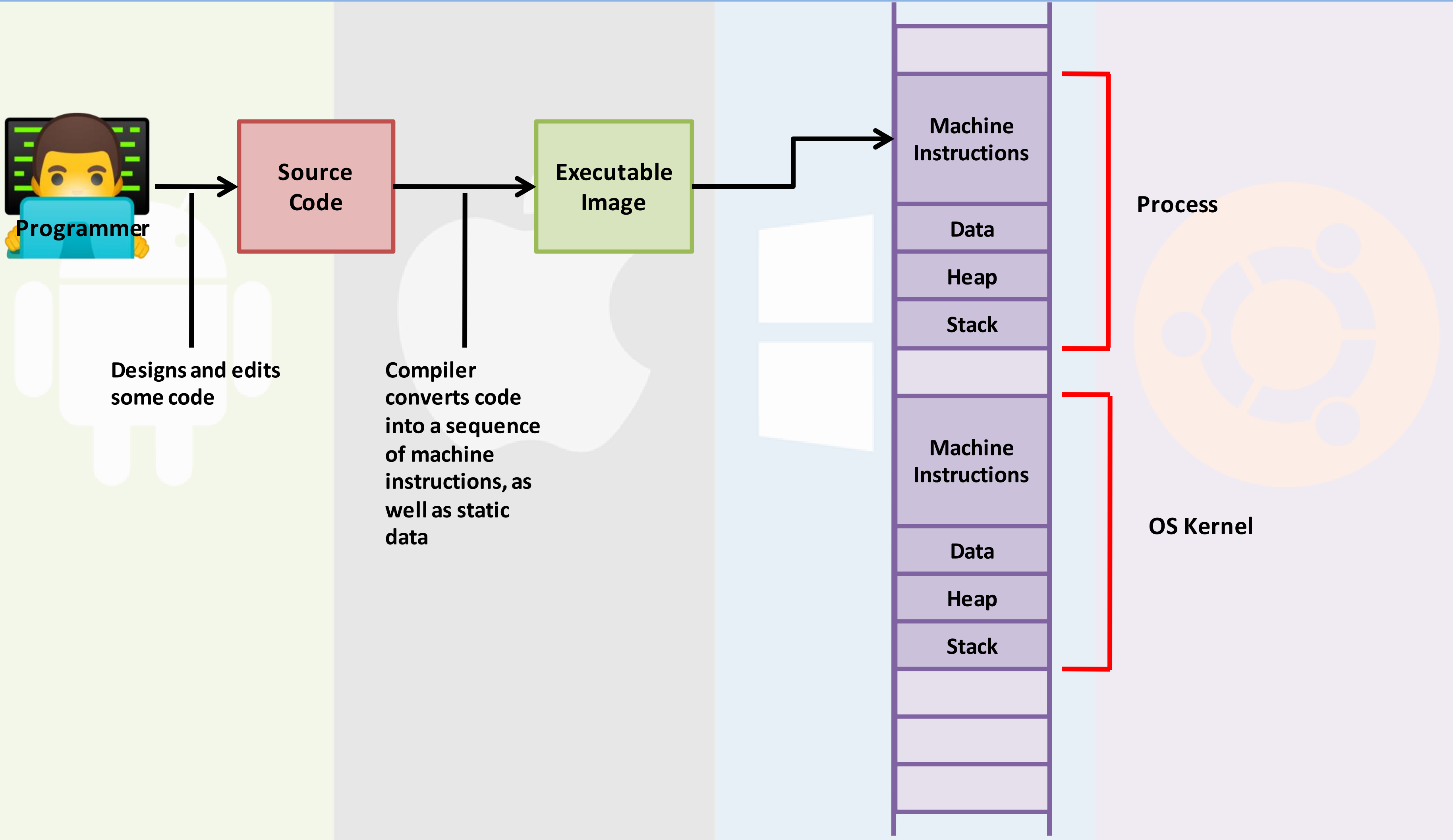
The Process Abstraction



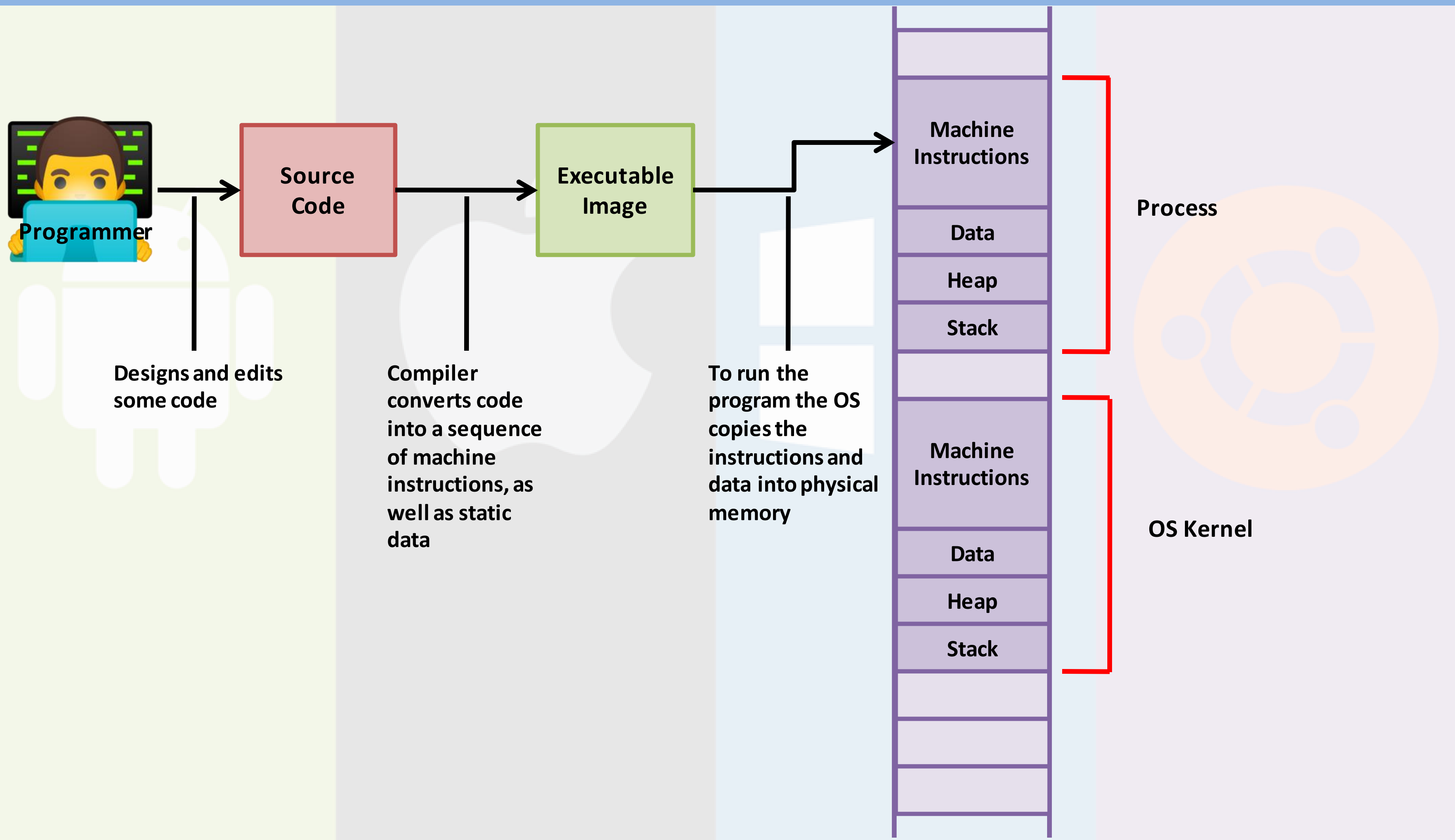
The Process Abstraction



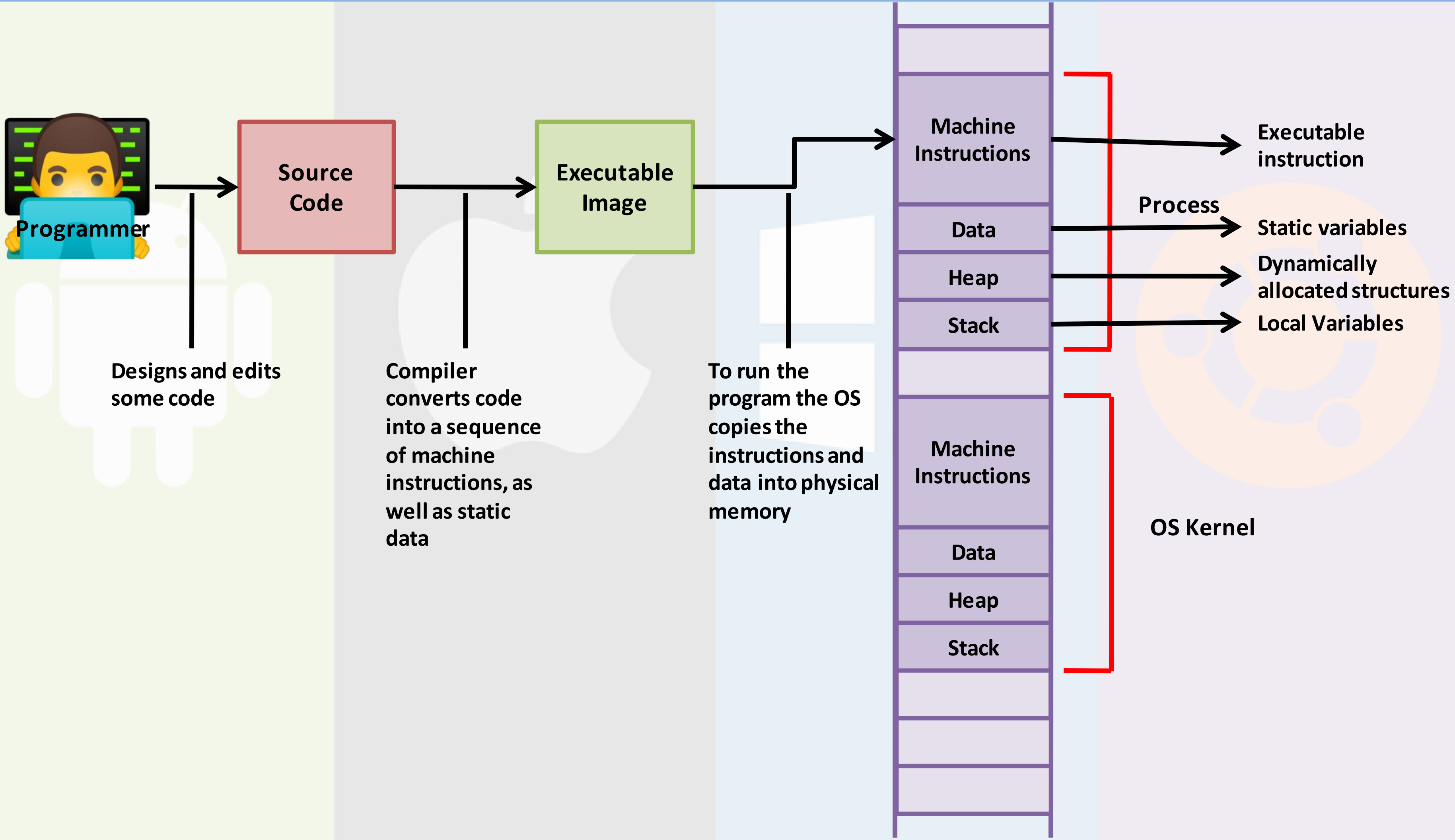
The Process Abstraction



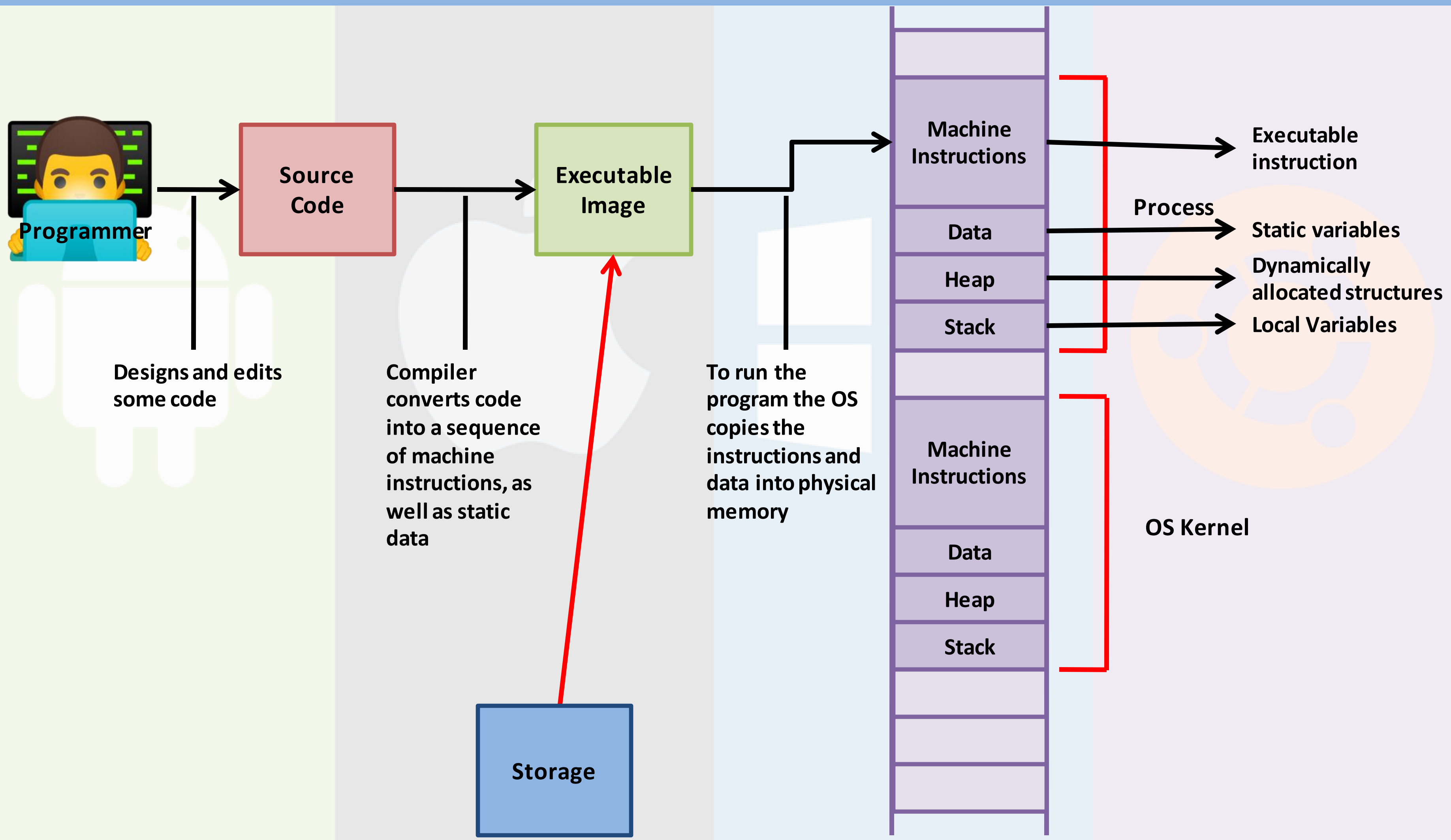
The Process Abstraction



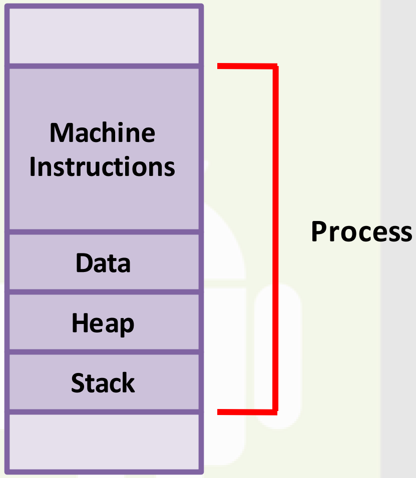
The Process Abstraction



The Process Abstraction



The Process Abstraction



- Process is made up of :
  - Instructions
  - Data
  - Heap
  - Stack

How do we run a program?

- The OS provides an API for which we can use to create processes.

- An API is an Application Programming Interface, which is a software intermediary that allows two applications to talk to each other

Process API

- These APIs are available on any modern OS.
  - **Create**
    Create a new process to run a program
  - **Destroy**
    Halt a runaway process
  - **Wait**
    Wait for a process to stop running
  - **Miscellaneous Control**
    Some kind of method to suspend a process and then resume it
  - **Status**
    Get some status info about a process

Process API - Process Creation

1. **Load** a program code into memory, into the address space of the process. Programs initially reside on disk in *executable format*.

   OS perform the loading process **lazily**.

   Loading pieces of code or data only as they are needed during program execution.

Process API - Process Creation

1. **Load** a program code into memory, into the address space of the process.
   Programs initially reside on disk in *executable format*.

   OS perform the loading process lazily.

   Loading pieces of code or data only as they are needed during program execution.

2. The program's **run-time stack** is allocated.
   Use the stack for *local variables*, *function parameters*, and *return address*.
   Initialize the stack with arguments → `argc` and the `argv` array of `main()` function

Process API - Process Creation

3. The program's **heap** is created.
   Used for explicitly requested dynamically allocated data.
   Program request such space by calling `malloc()` and free it by calling `free()`.

Process API - Process Creation

3. The program's **heap** is created.
   Used for explicitly requested dynamically allocated data.
   Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other initialization tasks.
   input/output (I/O) setup
      Each process by default has three open file descriptors.
      Standard input, output and error

Process API - Process Creation

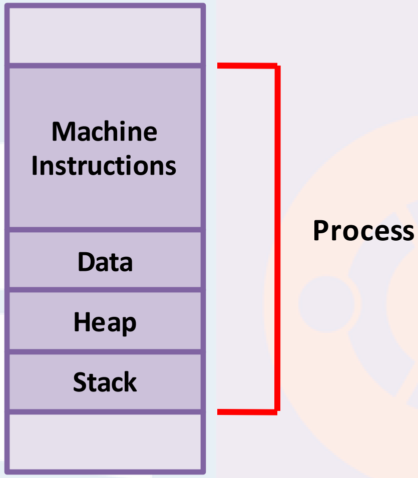
3. The program's **heap** is created.
   Used for explicitly requested dynamically allocated data.
   Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other initialization tasks.
   input/output (I/O) setup
      Each process by default has three open file descriptors.
      Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`.
   The OS *transfers control* of the CPU to the newly-created process.

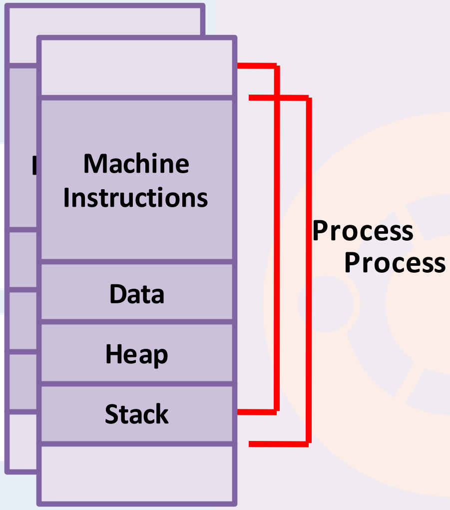
How do we run multiple copies of the same program?

- The OS can make multiple copies of the programs'
  - instructions
  - data
  - stack
  - heap



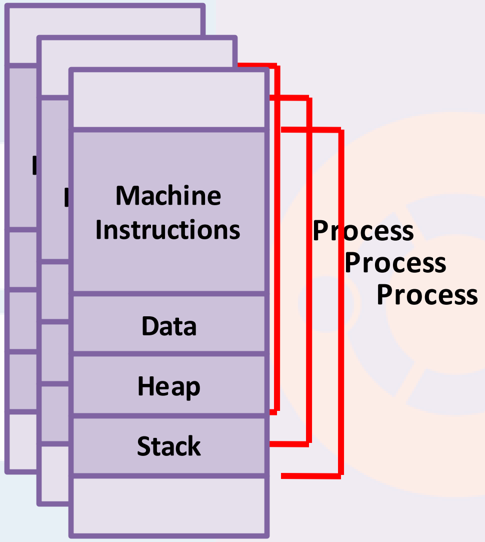
How do we run multiple copies of the same program?

- The OS can make multiple copies of the programs'
  - instructions
  - data
  - stack
  - heap



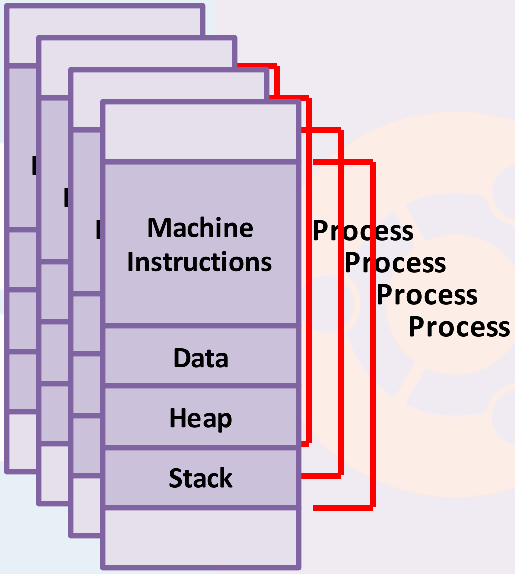
How do we run multiple copies of the same program?

- The OS can make multiple copies of the programs'
  - instructions
  - data
  - stack
  - heap



How do we run multiple copies of the same program?

- The OS can make multiple copies of the programs'
  - instructions
  - data
  - stack
  - heap



How do we run multiple copies of the same program?

- The OS can make multiple copies of the programs'
  - instructions
  - data
  - stack
  - heap

- Better way is to reuse memory where possible
  - just store one instance of instructions
  - This will be discussed in later lectures

Machine Instructions	~~Machine Instructions~~	~~Machine Instructions~~	~~Machine Instructions~~
Data	Data	Data	Data
Heap	Heap	Heap	Heap
Stack	Stack	Stack	Stack

What is the difference between a process and a program?

- Process is an instance of a program
    - Just like an object is an instance of a class

What is the difference between a process and a program?

- Process is an instance of a program
  - Just like an object is an instance of a class

- This means for each instance of a program there is a process in memory with its own copy of the program

How do we keep track of all these instances?

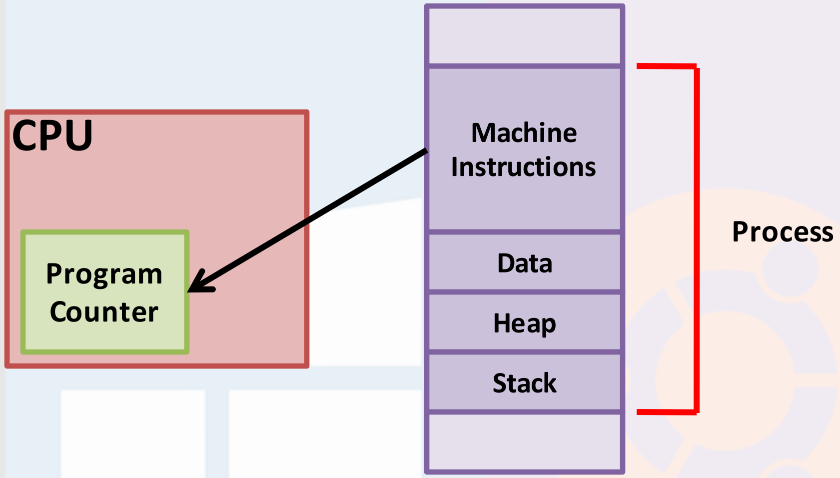
- The OS uses a data structure called a process control block (PCB) to keep track of the various processes

- The PCB stores :
  - Where a process is stored in memory
  - Where the executable image is on the disk
  - Which user asked to execute the process
  - Which privileges the process has

Process Control Block

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                  // Start of process memory
    uint sz;                    // Size of process memory
    char *kstack;               // Bottom of kernel stack
                                // for this process
    enum proc_state state;            // Process state
    int pid;                    // Process ID
    struct proc *parent;        // Parent process
    void *chan;                 // If non-zero, sleeping on chan
    int killed;                 // If non-zero, have been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;          // Current directory
    struct context context;           // Switch here to run
process
    struct trapframe *tf;       // Trap frame for the
                                // current interrupt
};
```

How to handle the processing with a level of control ?



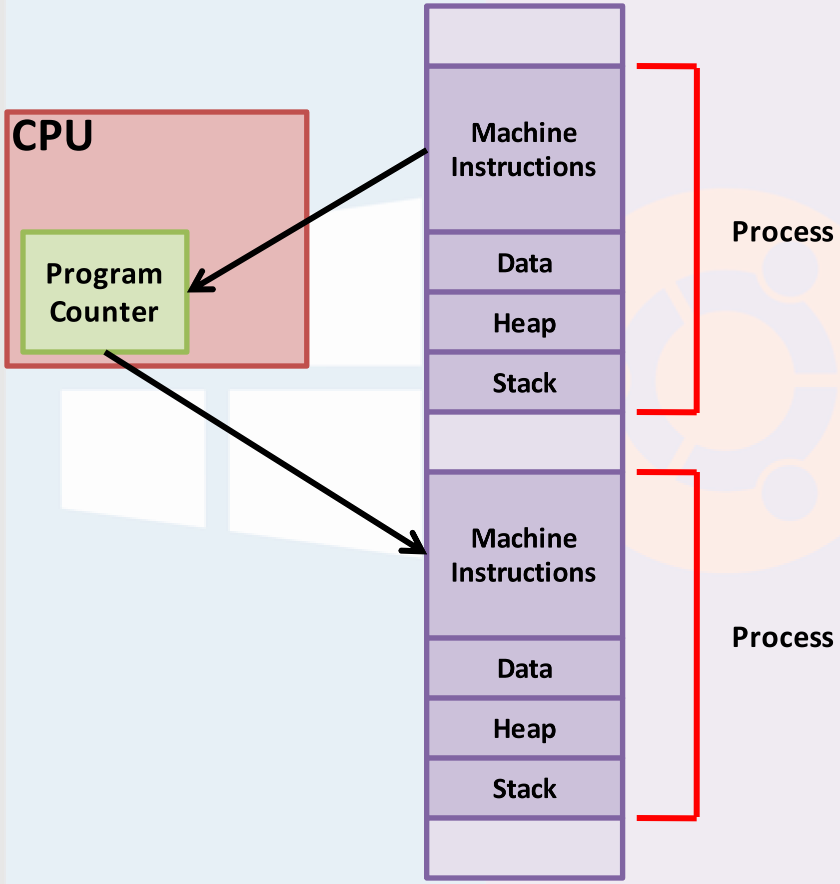
Direct Execution

- **Just run the program directly on the CPU.**

OS	**Program**
1. Create entry for process list 2. Allocate memory for program 3. Load program into memory 4. Set up stack with `argc` / `argv` 5. Clear registers 6. Execute call `main()`	
	7. Run `main()` 8. Execute `return` from `main()`
9. Free memory of process 10. Remove from process list	

How do we prevent a process accessing another?

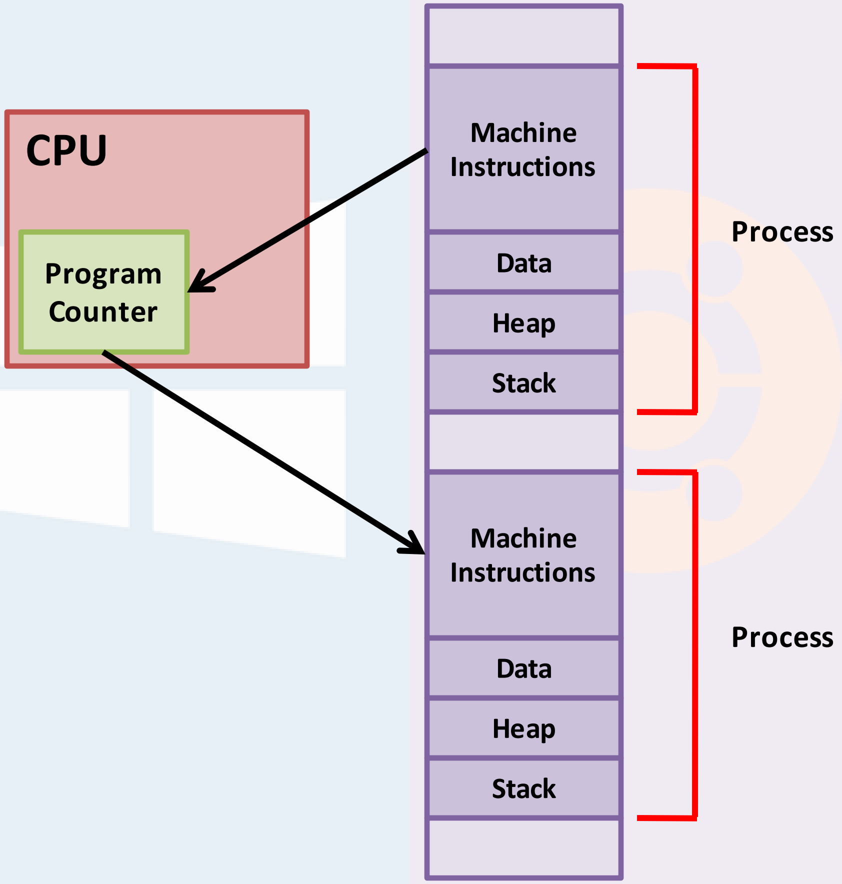
- OpCode
  - BR
  - LOAD
  - ADDA



How do we prevent a process accessing another?

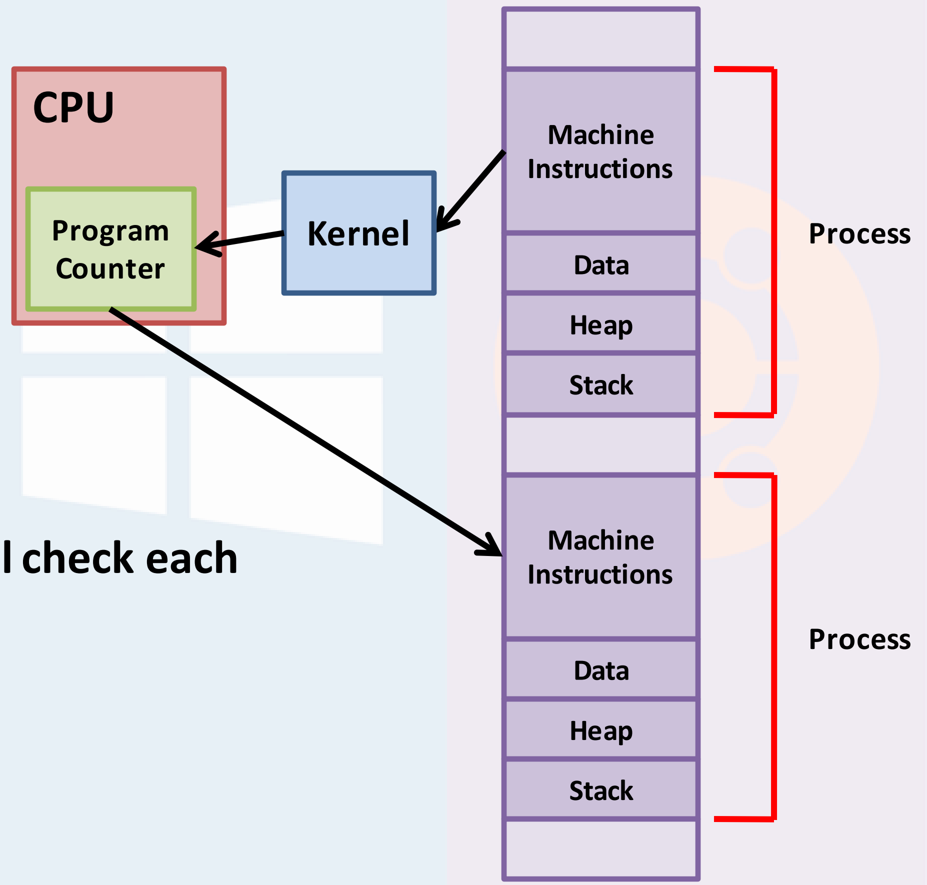
- **OpCode**
  - **BR –** branch to another instruction
  - **LOAD –** load a value into a register
  - **ADDA –** add a value to the accumulator

- **How do we prevent a process from manipulating another.**



Kernel Checks

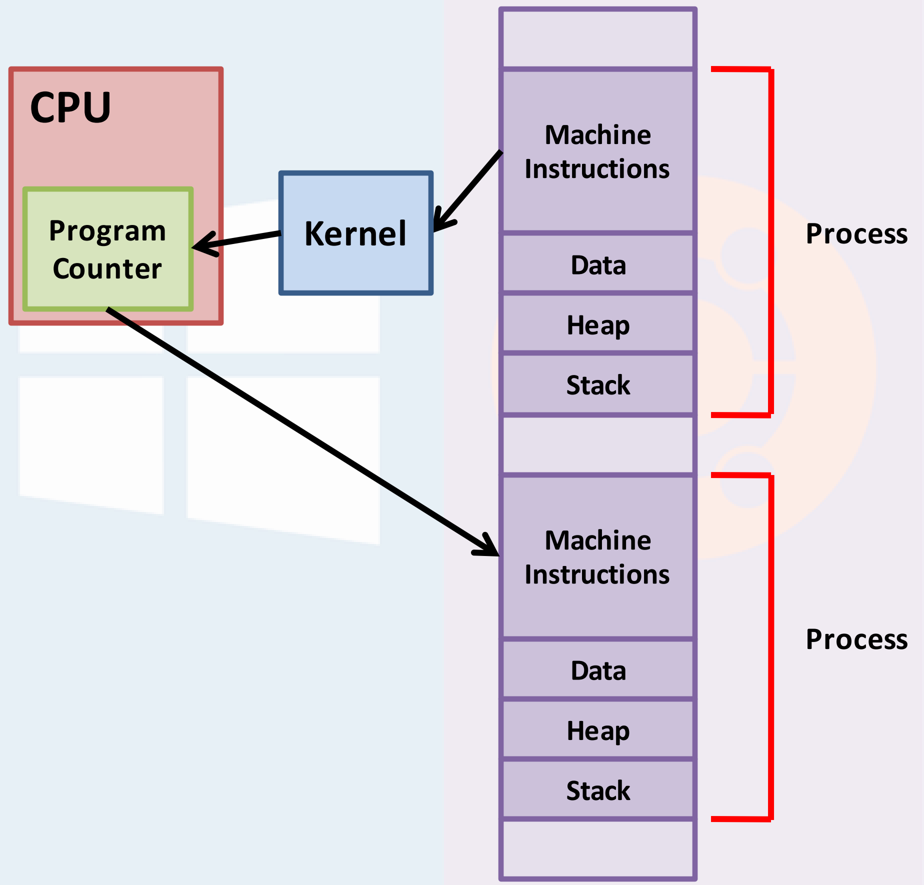
- **OpCode**
  - **BR –** branch to another instruction
  - **LOAD –** load a value into a register
  - **ADDA –** add a value to the accumulator



- How do we prevent a process from manipulating another.

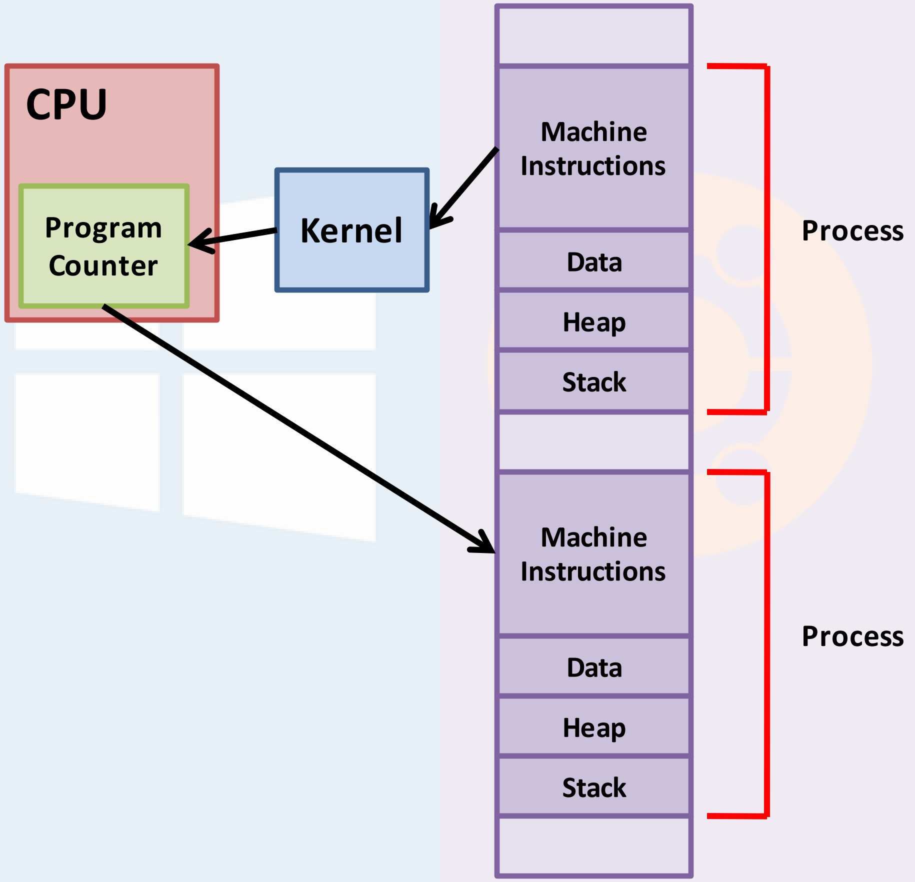
- **Simple approach would be to have the kernel check each instruction to see if it had permission.**

How can we speed up this approach?



How can we speed up this approach?

- Most instructions are perfectly safe
- So we only have to check those instructions we could pose risk



How can we speed up this approach?

- Most instructions are perfectly safe
- So we only have to check those instructions we could pose risk

- **To accomplish this we implement the same checks on the hardware level**

How can we speed up this approach?

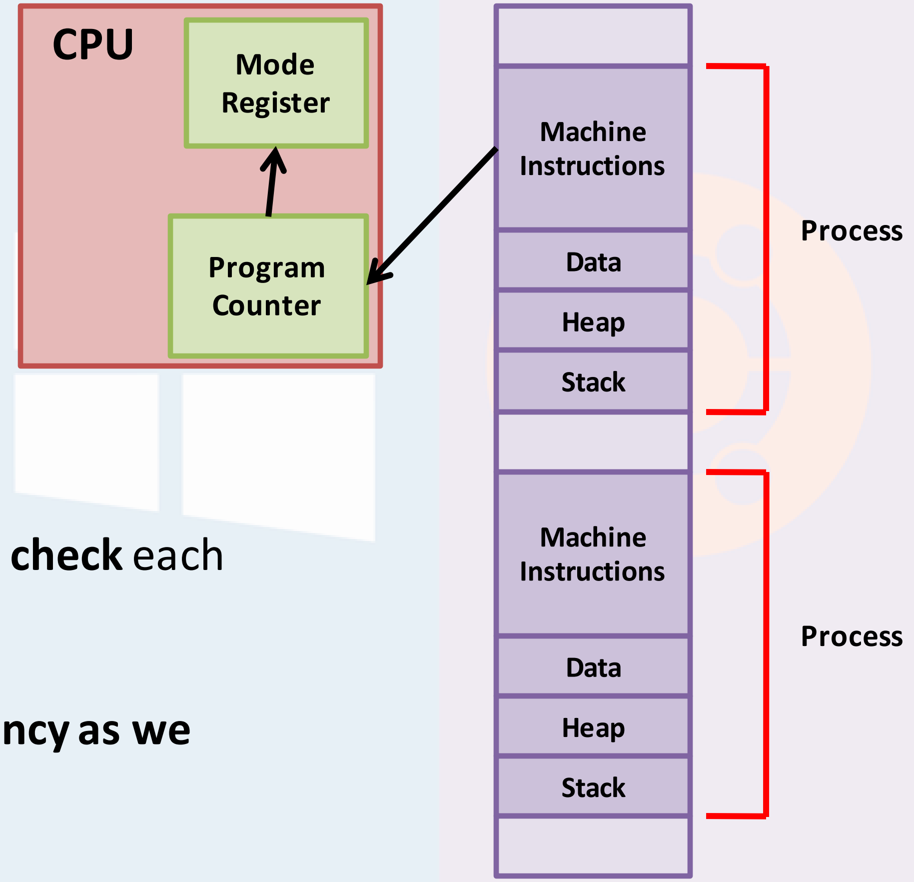
- Most instructions are perfectly safe
- So we only have to check those instructions we could pose risk

- To accomplish this we implement the same checks on the hardware level

- **This is called dual-mode operation, represented by a single bit in the processor which represents its' status**

Dual Mode Operation

- In User Mode
  - The processor checks instruction
- In Kernel Mode
  - The processor executes the instructions



- **Simple approach** would be to have the **kernel check** each instruction to see if it had permission.

- **Dual-Mode approach allows increased efficiency as we only check when we do not trust the process**

What instructions can't a process execute?

What instructions can't a process execute?

- **Privileged Instructions**
  - **Instructions available in kernel mode but not in user mode**
    - Change privilege levels
    - Access memory
    - Disable/Enable interrupts

Types of Mode Transfer

- **The next question is how to safely transfer to and from our different modes**
  - These transitions are not rare
  - Safe, Fast and Efficient

User to Kernel Mode

- 3 reasons for the kernel to take control
  - Interrupts
  - Processor Exceptions
  - System Calls

User to Kernel Mode

- **3 reasons for the kernel to take control**
  - **Interrupts –** Asynchronous event
  - **Processor Exceptions –** Synchronous event
  - **System Calls –** Synchronous event

- **Asynchronous event =** triggered by external events
- **Synchronous event =** triggered by process execution

User to Kernel Mode

- **3 reasons for the kernel to take control**
  - **Interrupts –** Asynchronous event
  - **Processor Exceptions –** Synchronous event
  - **System Calls –** Synchronous event

- Asynchronous event = triggered by external events
- Synchronous event = triggered by process execution

- **We use the term trap to refer to any synchronous transfer of control from user to kernel (less privileged to more)**

Interrupts

- Asynchronous signal to the processor indicating some event occurred that the processor should look at

<u>Interrupts</u>

- Asynchronous signal to the processor indicating some event occurred that the processor should look at

- **As the process executes instructions it will check if an interrupt has occurred**
    - If Yes = completes or stalls processing current instruction, saves current execution state then starts executing a <u>interrupt handler</u> in the kernel

    - If No = continues with current instruction processing

- **For different interrupts we have different handlers**

<u>How does a kernel regain control from a runaway process?</u>

- **<u>Timer Interrupts</u>**

- Since through process isolation we give the process the illusion of being fully in control, we need a way to regain control

- For example when a program becomes non responsive and a user chooses to close it

How does a kernel regain control from a runaway process?

- **Timer Interrupts**

- Since through process isolation we give the process the illusion of being fully in control, we need a way to regain control of the processor

- For example when a program becomes non responsive and a user chooses to close it

- **Additionally, the OS needs to regain control in normal operation as well**

- **For example if you are typing, listening to music and downloading a file**
    - The OS needs to be able to switch between tasks smoothly
    - This is handled by a device called a hardware timer

- **The hardware timer is used to interrupt the processor after a certain delay**
    - After a specified delay, the CPU transfers control from the user process to the kernel running in kernel mode

Different Interrupts

- **Timer Interrupt**
  - Checks if current process is being responsive to user input, used to detect infinite loops
  - Switches between processes to ensure that each process gets a turn

Different Interrupts

- **Timer Interrupt**

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	

Different Interrupts

- **Timer Interrupt**

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
	(Cont.)	
Handle the trap Call switch() routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B …

Different Interrupts

- Timer Interrupt
  - Checks if current process is being responsive to user input, used to detect infinite loops
  - Switches between processes to ensure that each process gets a turn
- **I/O requests**
  - **A mouse triggers an interrupt every time a click is detected**

Processor Exceptions

- **Hardware event caused by a user program behavior that causes a control transfer from user to kernel mode**

- As with interrupts ; completes or stalls processing current instruction, saves current execution state then starts executing a exception handler in the kernel

Processor Exceptions

- Hardware event caused by a user program behavior that causes a control transfer from user to kernel mode

- As with interrupts ; completes or stalls processing current instruction, saves current execution state then starts executing a exception handler in the kernel

- **Examples of exceptions**
  - Process attempts to perform privileged instruction
  - Access memory outside of own memory region
  - Division of integers by zero
  - Writing to read-only memory

- **In these cases the OS simply stops execution of the process and returns an error code**

System Calls

- Lastly, user processes can transition willing into kernel in order to request that the kernel perform an operation on the user's behalf

- **A System call is any procedure provided by the kernel that can be called from user level.**

- As with interrupts/Exceptions ; saves current execution state then starts executing a pre-defined handler in the kernel

System Calls

- Lastly, user processes can transition willing into kernel in order to request that the kernel perform an operation on the user's behalf

- A System call is any procedure provided by the kernel that can be called from user level.

- As with interrupts/Exceptions ; saves current execution state then starts executing a pre-defined handler in the kernel

- **Examples of System Calls**
  - Create (fork) / terminate processes
  - Wait command
  - Create/Delete files
  - Get/Set DateTime
  - Get/Set File permissions

System Calls

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember address of … syscall handler

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from -trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() … Call system **trap** into OS

System Calls

OS @ run (kernel mode)	Hardware	Program (user mode)
	(Cont.)	
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		… return from main trap (via `exit()`)
Free memory of process Remove from process list		

Kernel to User

- **There are several types of transitions from kernel to user**

- New Processes
- Resuming after an Interupt, Exception or System Call
- Context Switching
- User-level Upcalls

New Processes

- **To start a New Process**
  - Kernel copies program into memory
  - Sets program counter to the start of the process
  - Switches to user mode

Resuming

- **To resume a process after the kernel finishes handling the interrupt**
  - Restores the program counter to the instruction of the interrupted program
  - Restores the registers
  - This information is found in the saved state
  - Switches to user mode

Context Switches

- **To switch to another process after receiving an interrupt**
    - Kernel saves current state of execution of the current process
    - Kernel then resumes another process by loading its current state
    - Switches to user mode

Context Switches

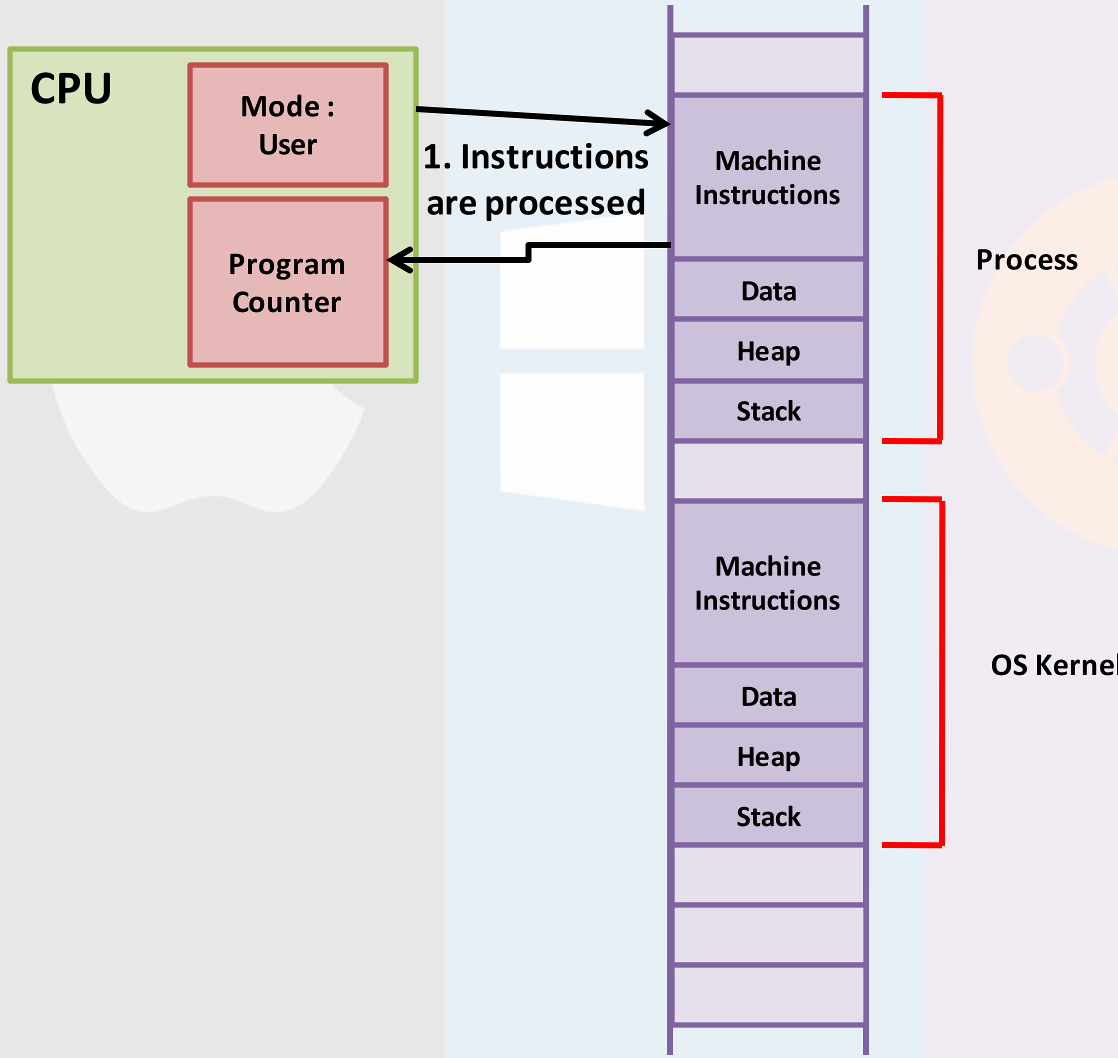
```
# void swtch(struct context **old, struct context *new);
#
# Save current register context in old
# and then load register context from new.
.globl swtch
swtch:
        # Save old registers
        movl 4(%esp), %eax              # put old ptr into eax
        popl 0(%eax)                    # save the old IP
        movl %esp, 4(%eax)              # and stack
        movl %ebx, 8(%eax)              # and other registers
        movl %ecx, 12(%eax)
        movl %edx, 16(%eax)
        movl %esi, 20(%eax)
        movl %edi, 24(%eax)
        movl %ebp, 28(%eax)
        # Load new registers
        movl 4(%esp), %eax              # put new ptr into eax
        movl 28(%eax), %ebp             # restore other registers
        movl 24(%eax), %edi
        movl 20(%eax), %esi
        movl 16(%eax), %edx
        movl 12(%eax), %ecx
        movl 8(%eax), %ebx
        movl 4(%eax), %esp              # stack is switched here
        pushl 0(%eax)                   # return addr put in place
        ret                             # finally return into new ctxt
```

User-level Upcalls

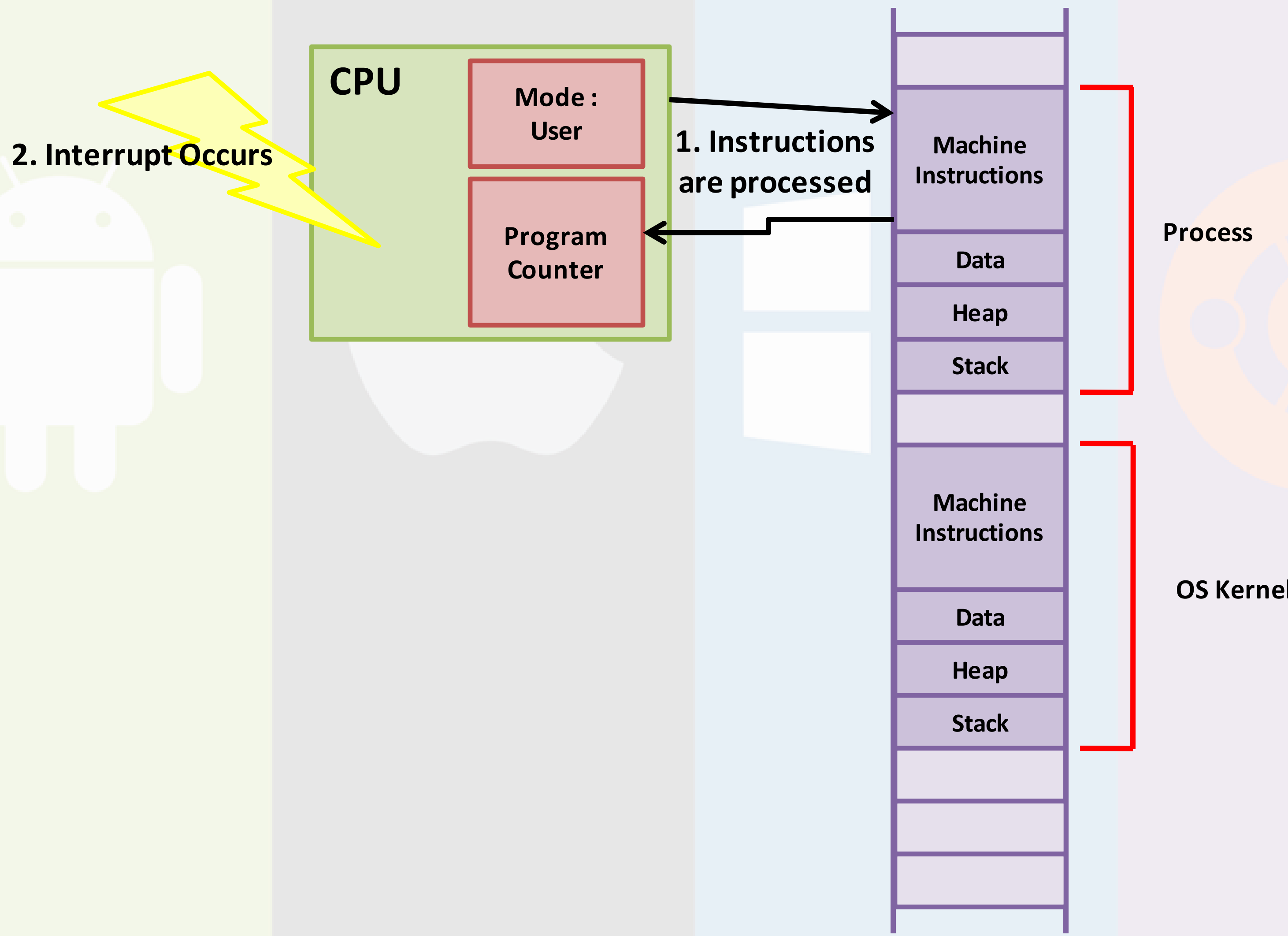
- Upcalls are virtualized interrupts and exceptions which allows user programs to receive asynchronous notifications of events

- It is Kernel Interrupt handling except at user level

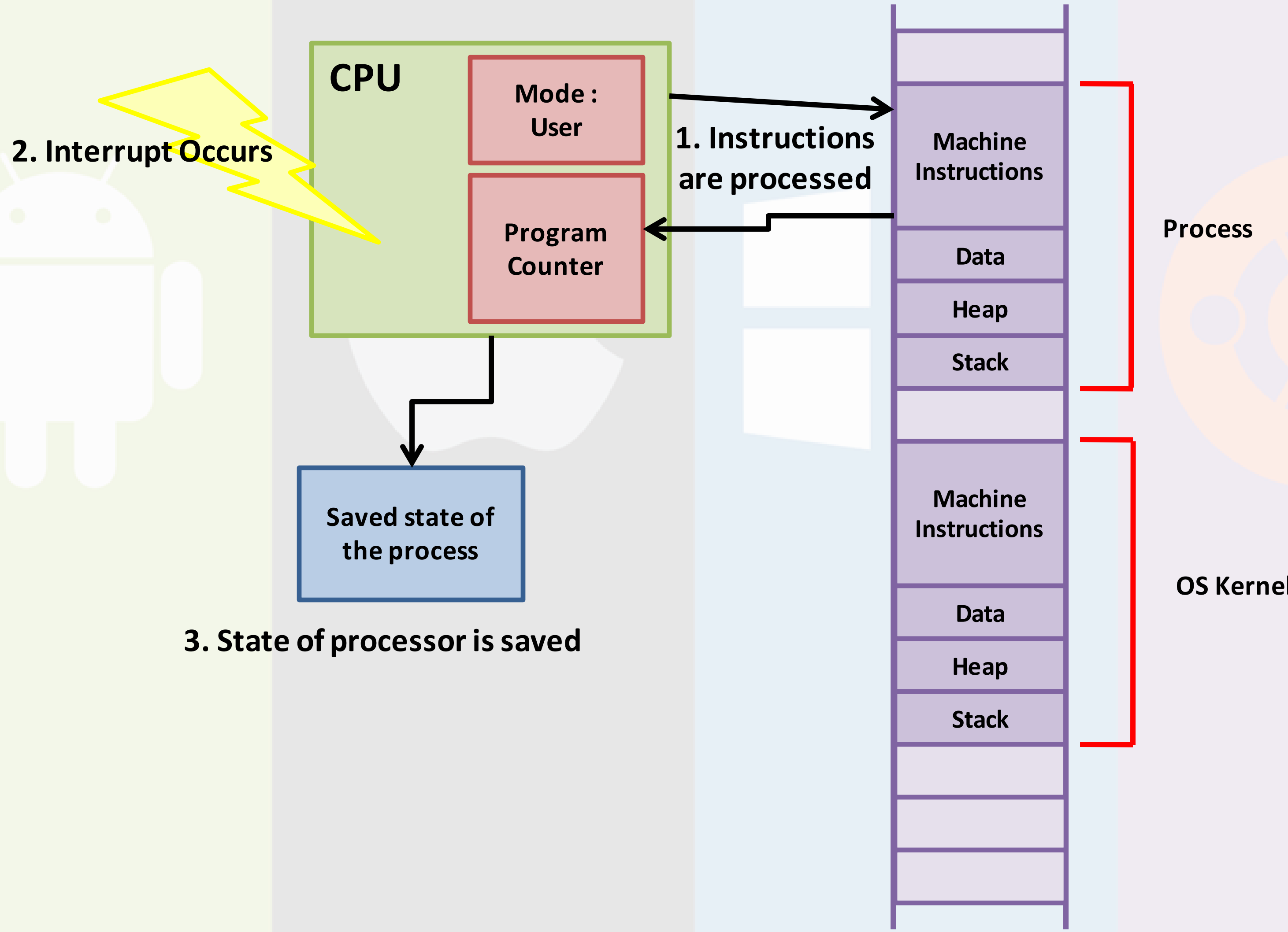
Summary


CPU
Mode : User
Program Counter
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel

Summary

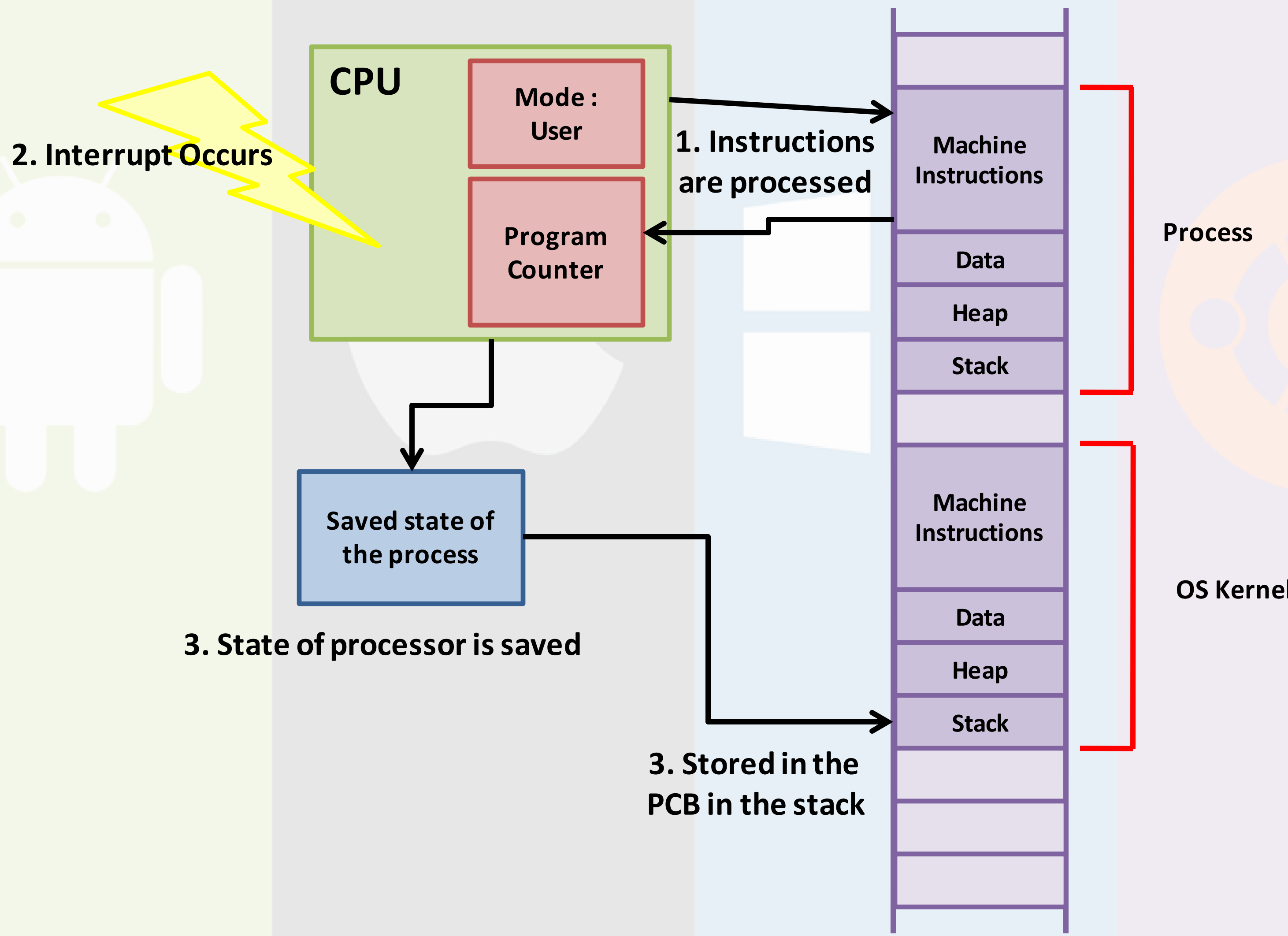

CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel

Summary

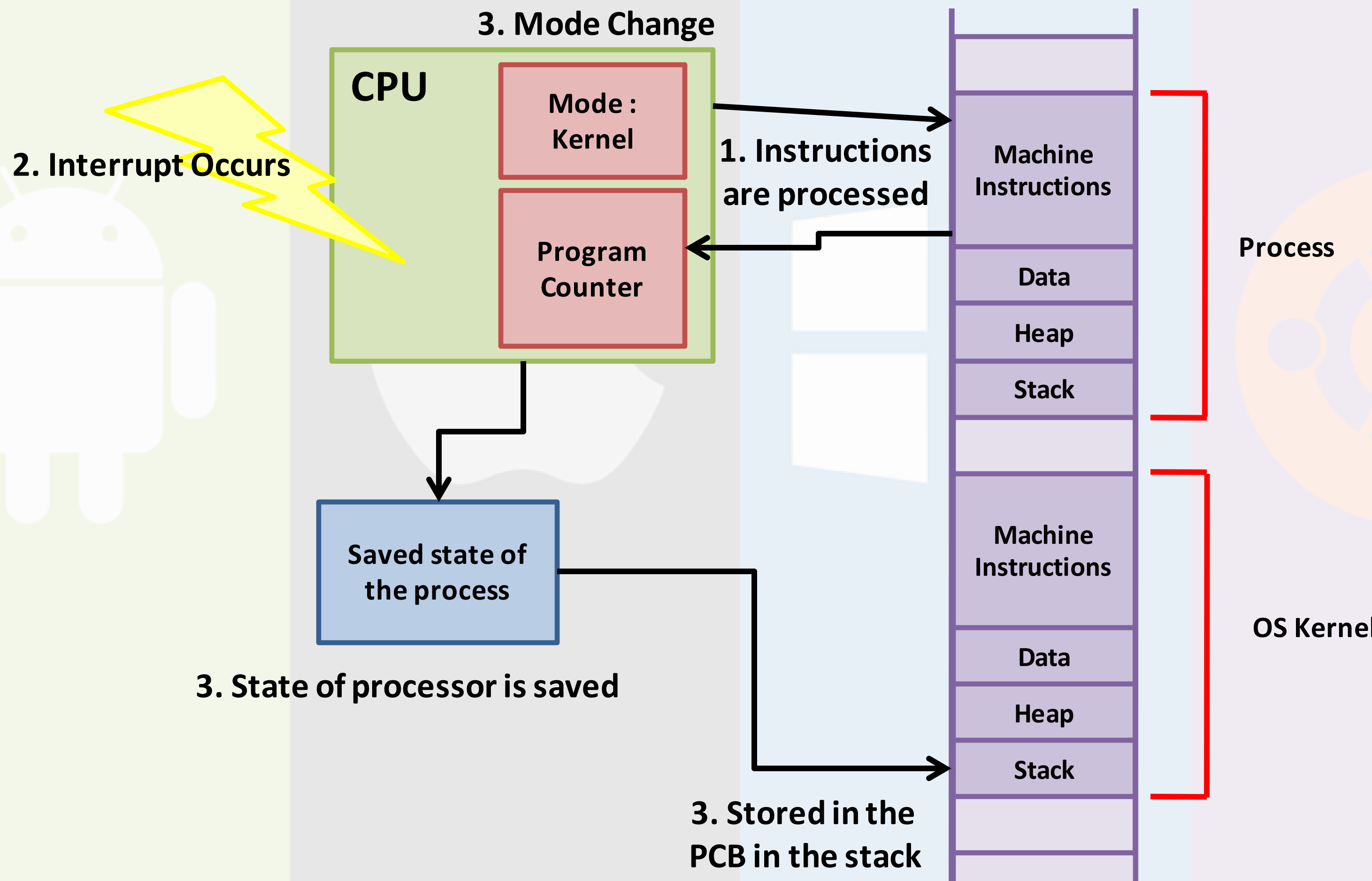


CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Saved state of the process
3. State of processor is saved
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel

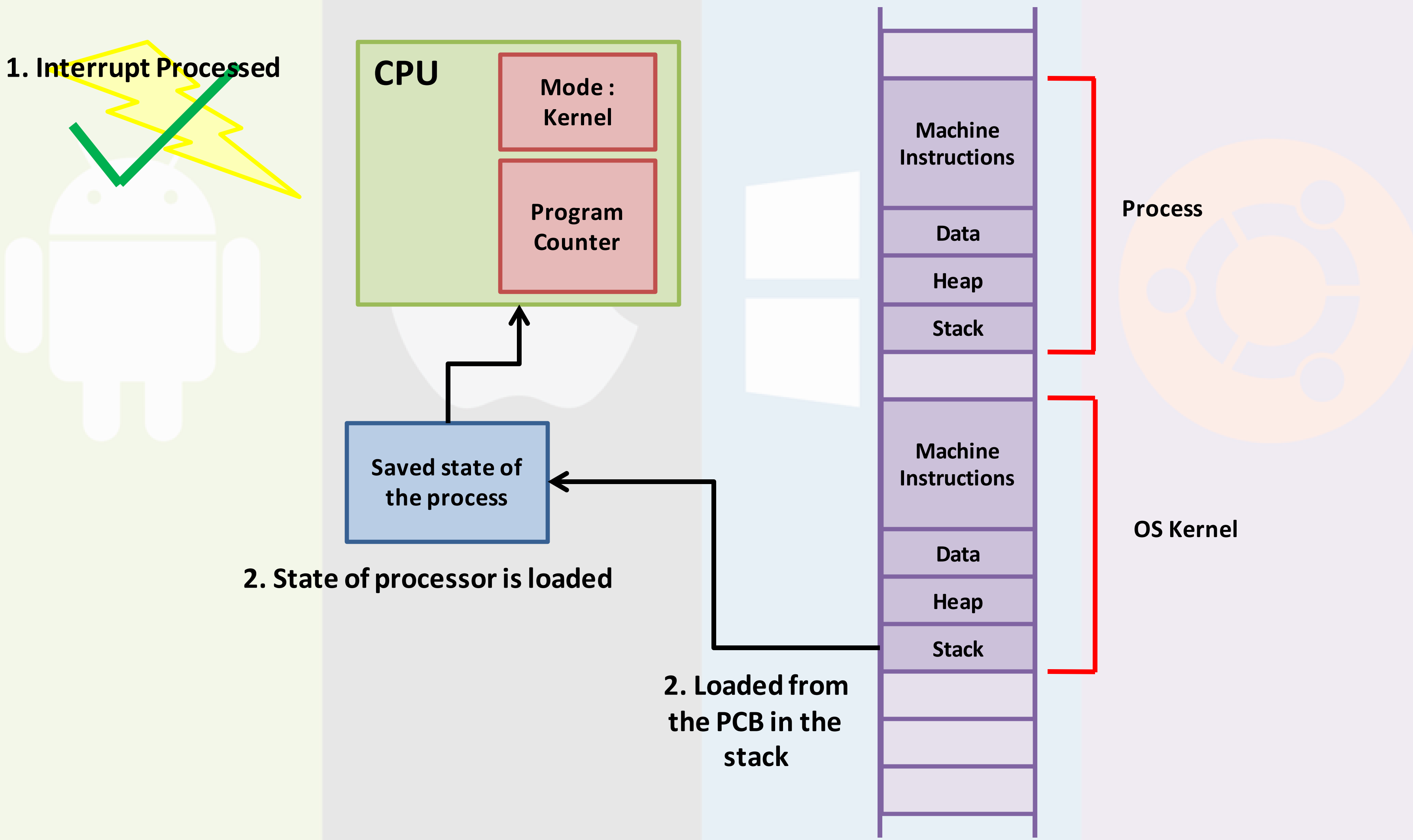
Summary


CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Saved state of the process
3. State of processor is saved
Machine Instructions
Data
Heap
Stack
OS Kernel
3. Stored in the PCB in the stack

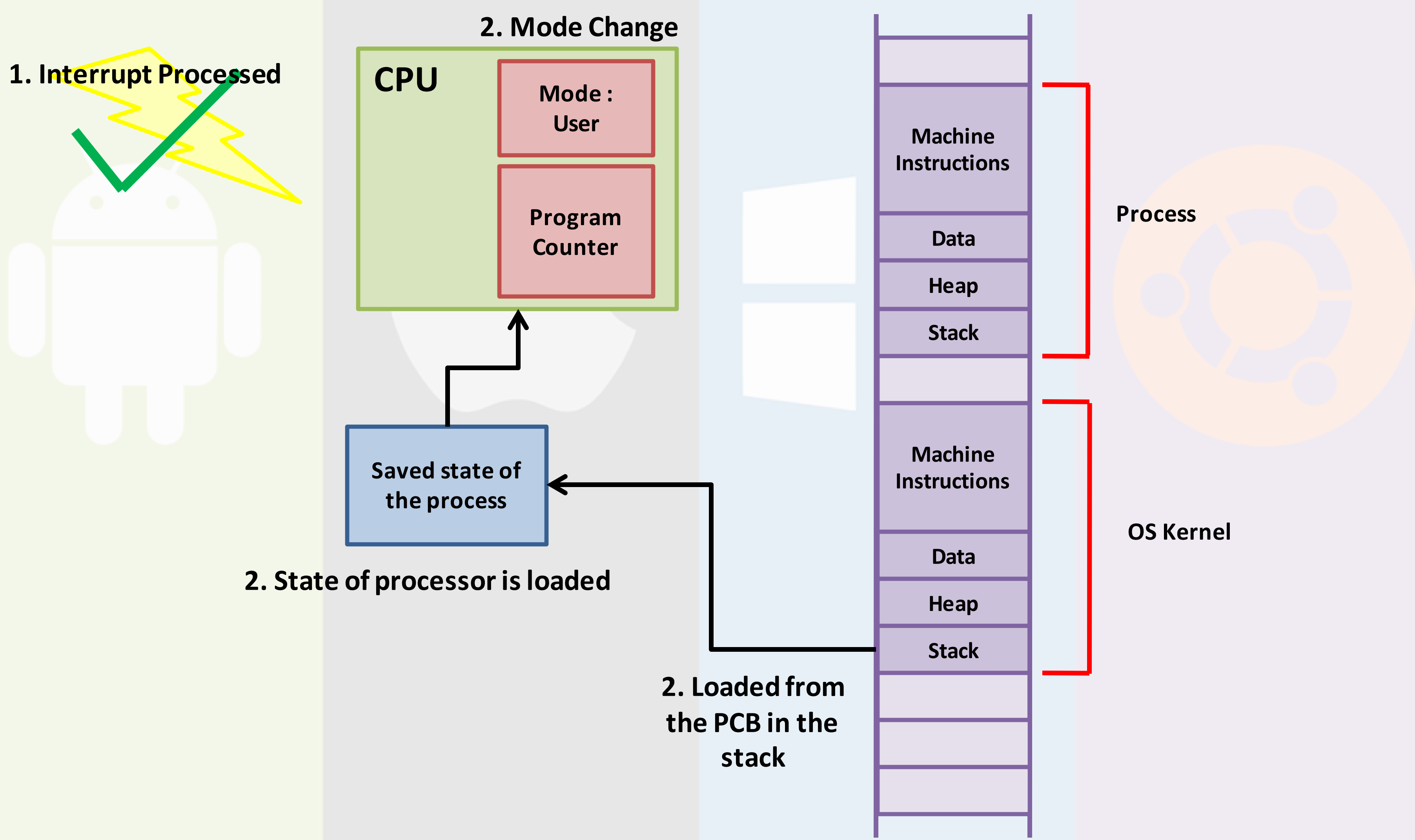
Summary


3. Mode Change
CPU
Mode : Kernel
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
Saved state of the process
3. State of processor is saved
3. Stored in the PCB in the stack

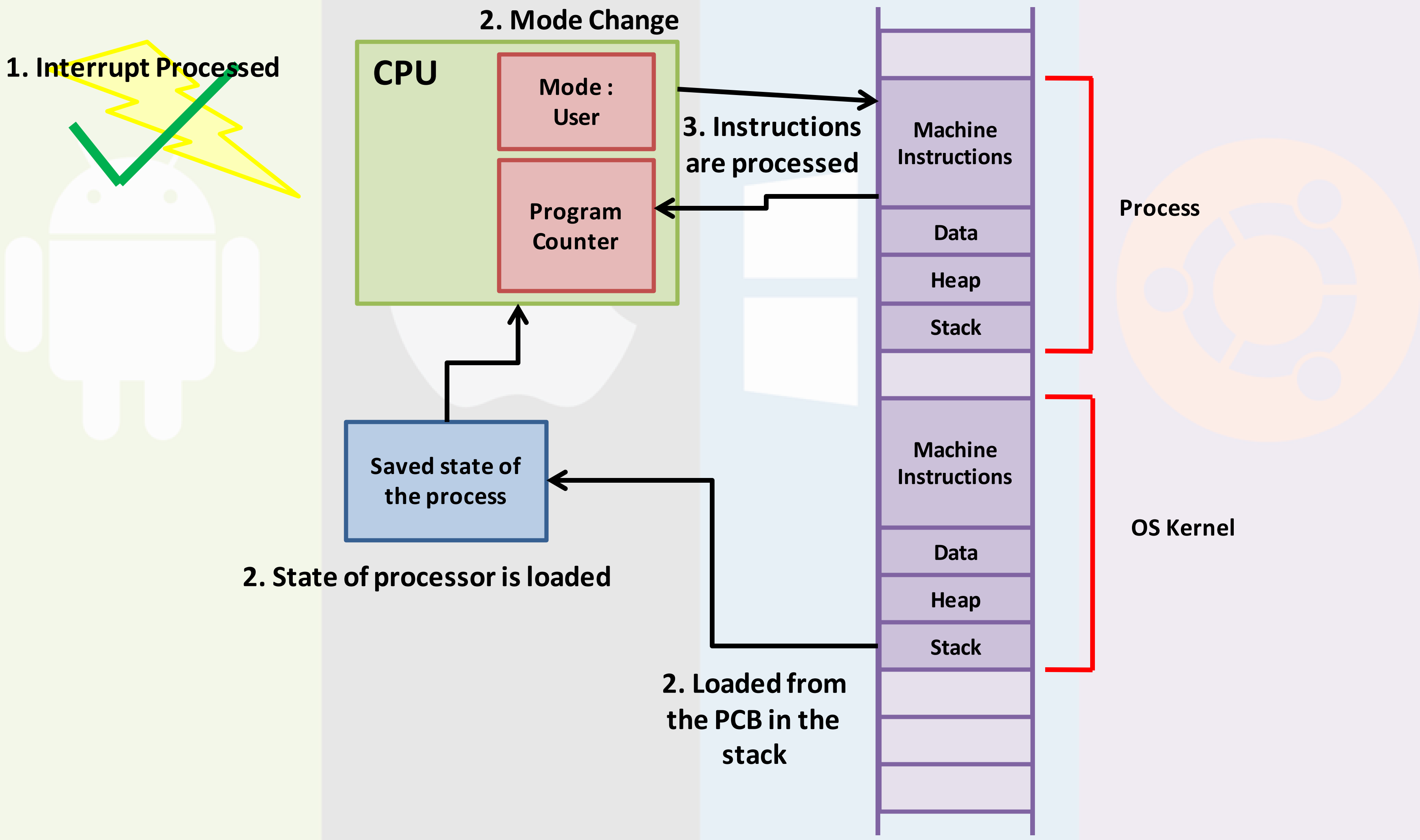
Summary


1. Interrupt Processed
CPU
Mode : Kernel
Program Counter
Saved state of the process
2. State of processor is loaded
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
2. Loaded from the PCB in the stack

Summary


1. Interrupt Processed
2. Mode Change
CPU
Mode : User
Program Counter
Saved state of the process
2. State of processor is loaded
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
2. Loaded from the PCB in the stack

Summary



1. Interrupt Processed
2. Mode Change
CPU
Mode : User
Program Counter
3. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Saved state of the process
2. State of processor is loaded
Machine Instructions
Data
Heap
Stack
OS Kernel
2. Loaded from the PCB in the stack