Operating Systems COMS(3010A) Kernels and Processes 2

Branden Ingram

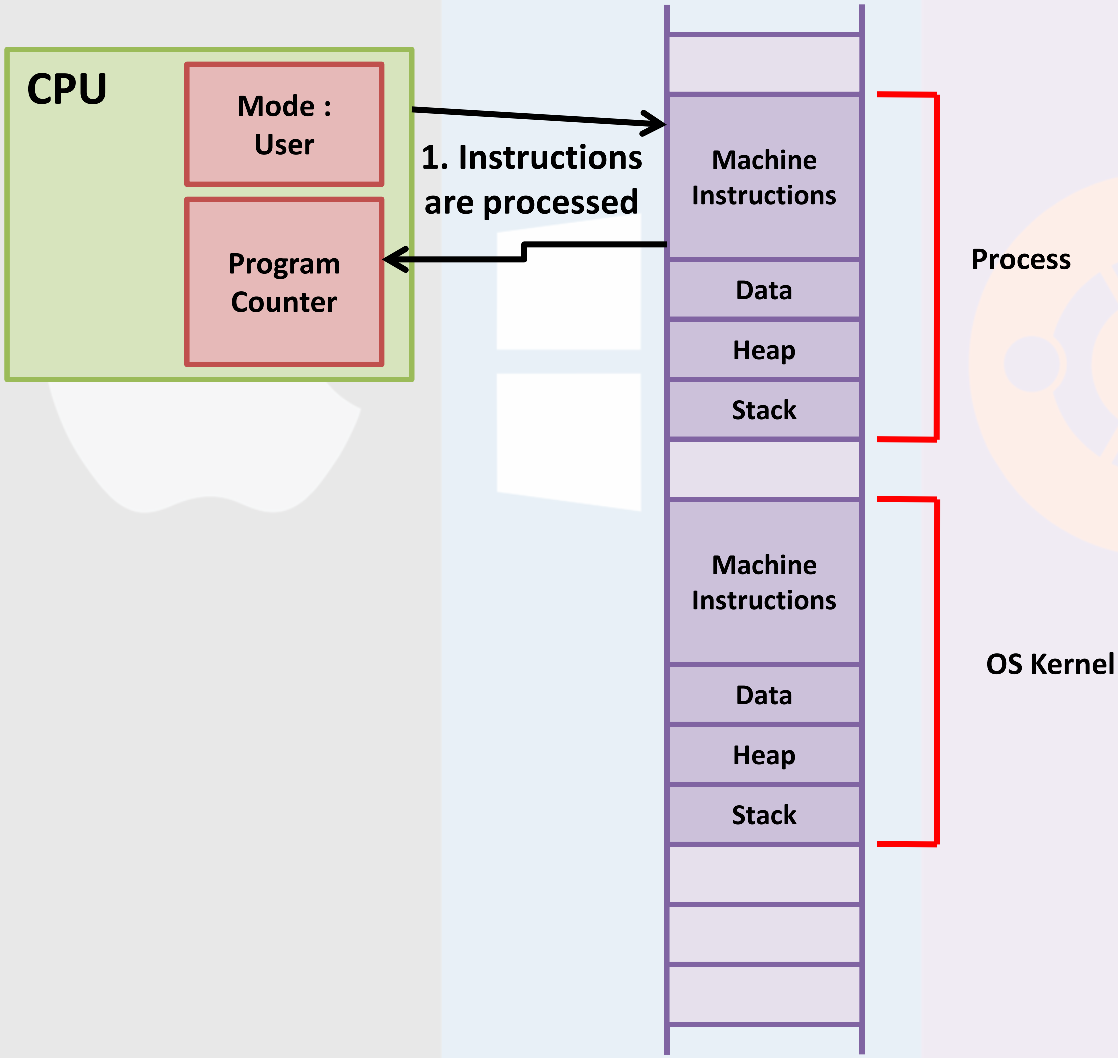
branden.ingram@wits.ac.za

Office Number : ???

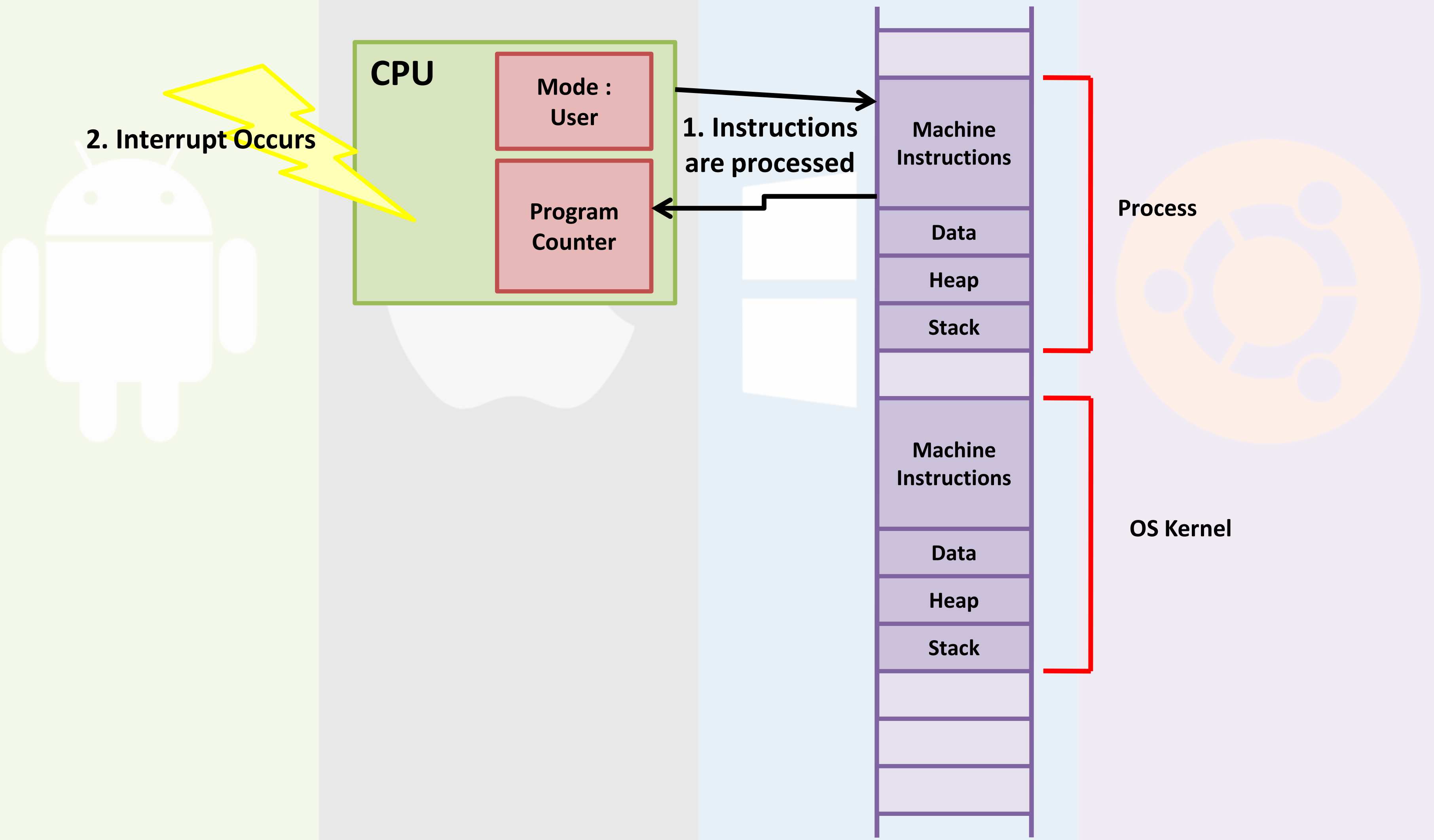
Recap

- Kernel
- Process Abstraction
- Process Control Block
- Dual-Mode Operation
- Mode Transfer

Summary

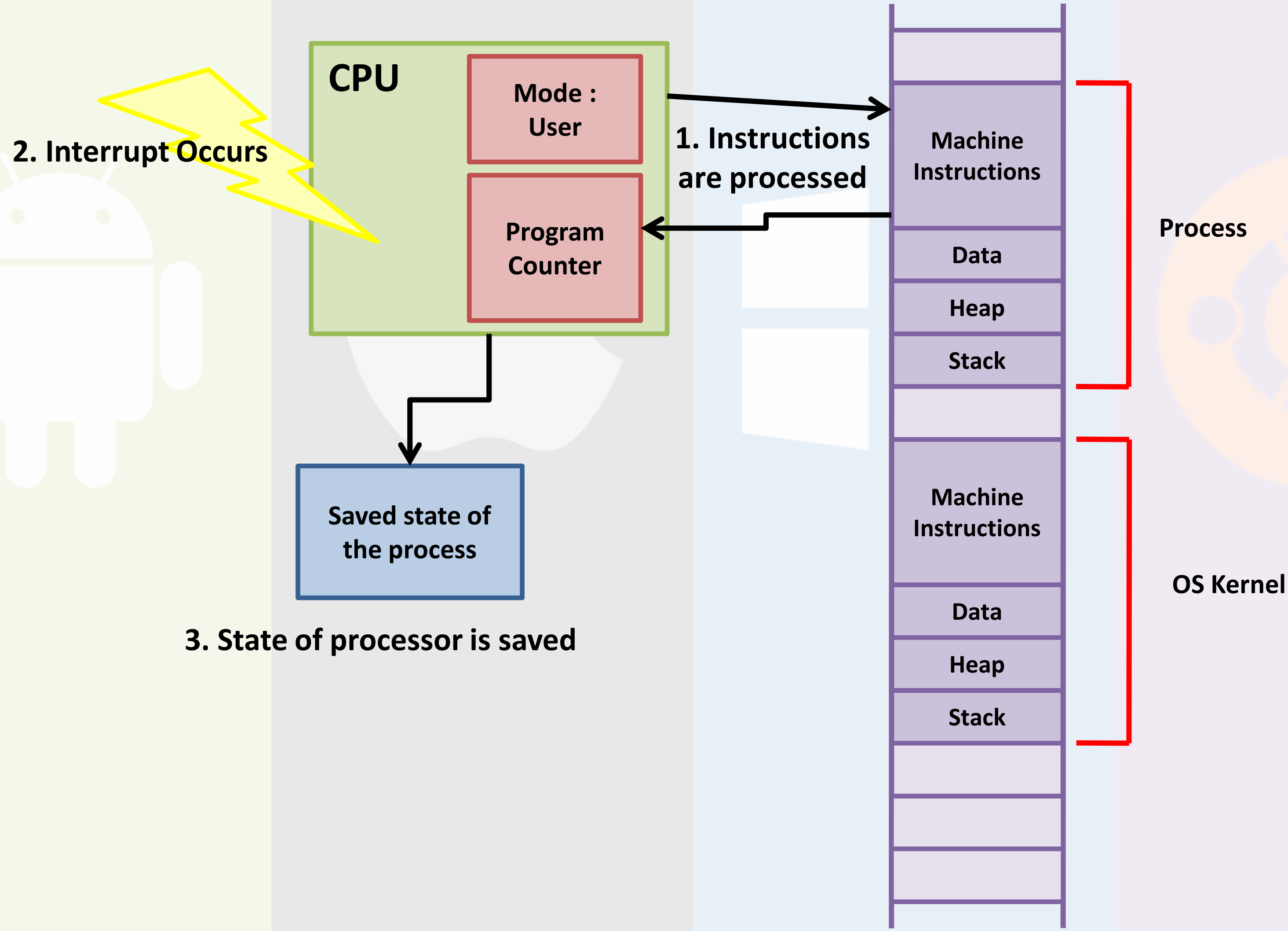

CPU
Mode : User
Program Counter
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel

Summary



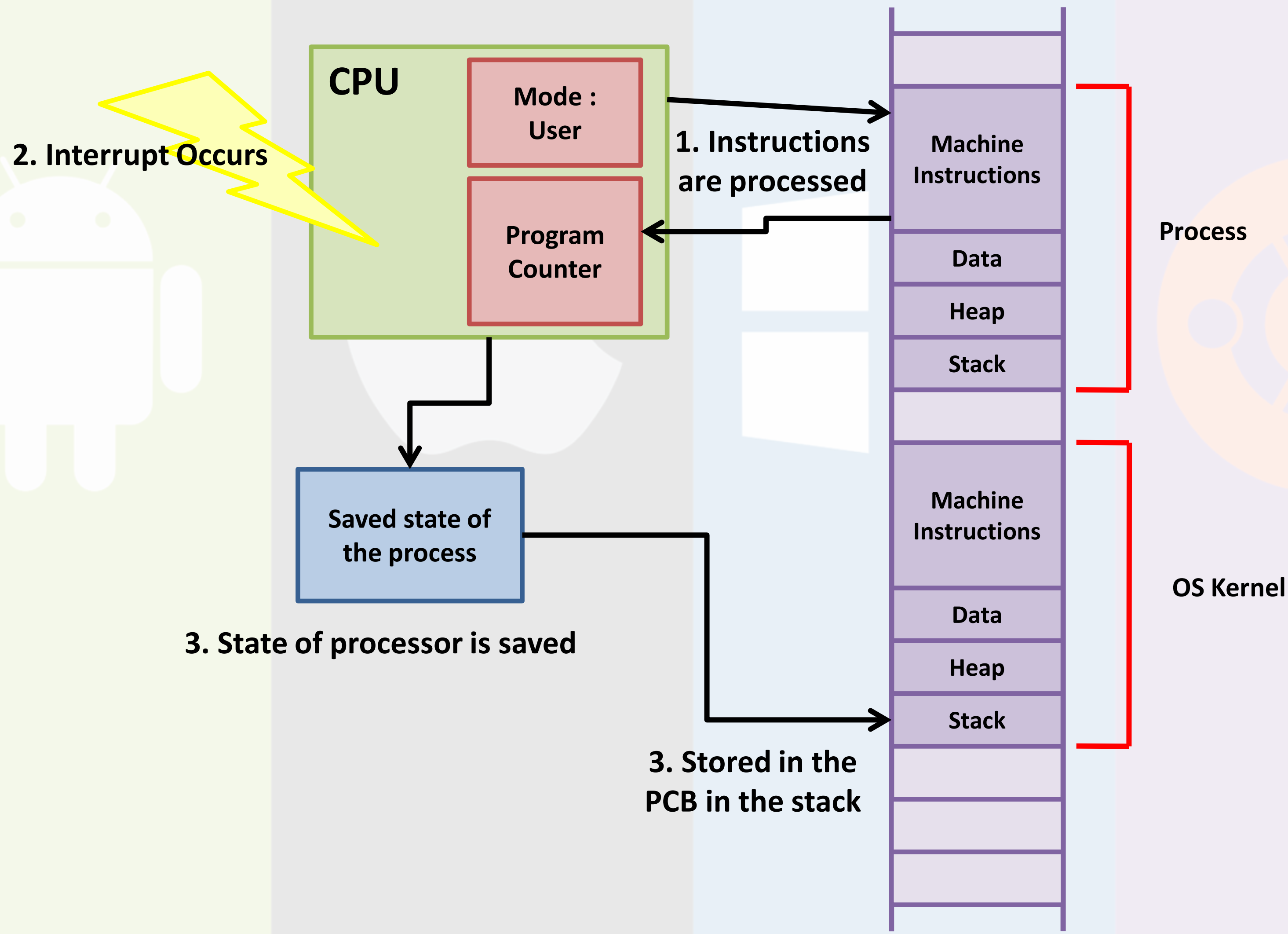
CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel

Summary



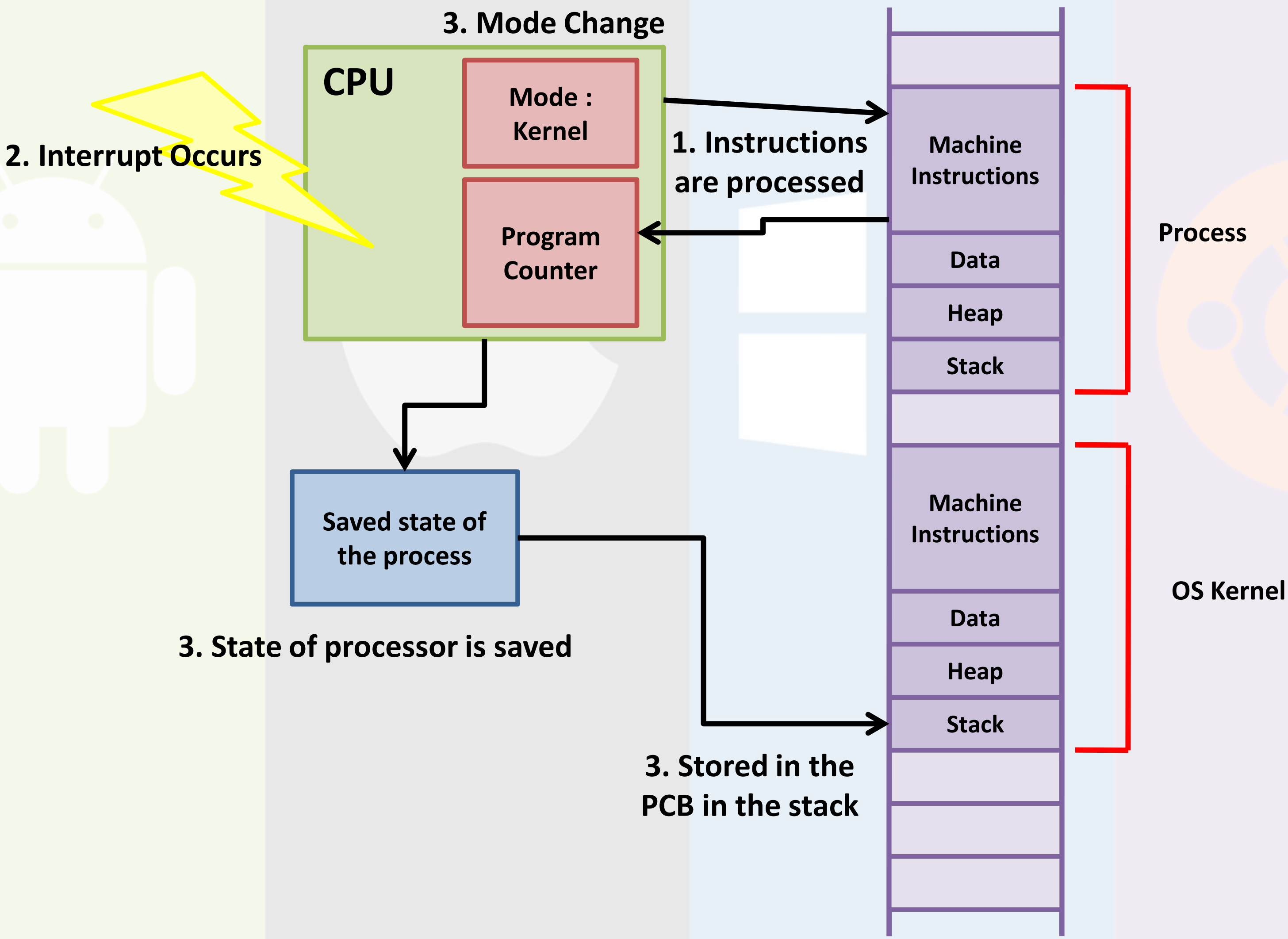
CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
Saved state of the process
3. State of processor is saved

Summary

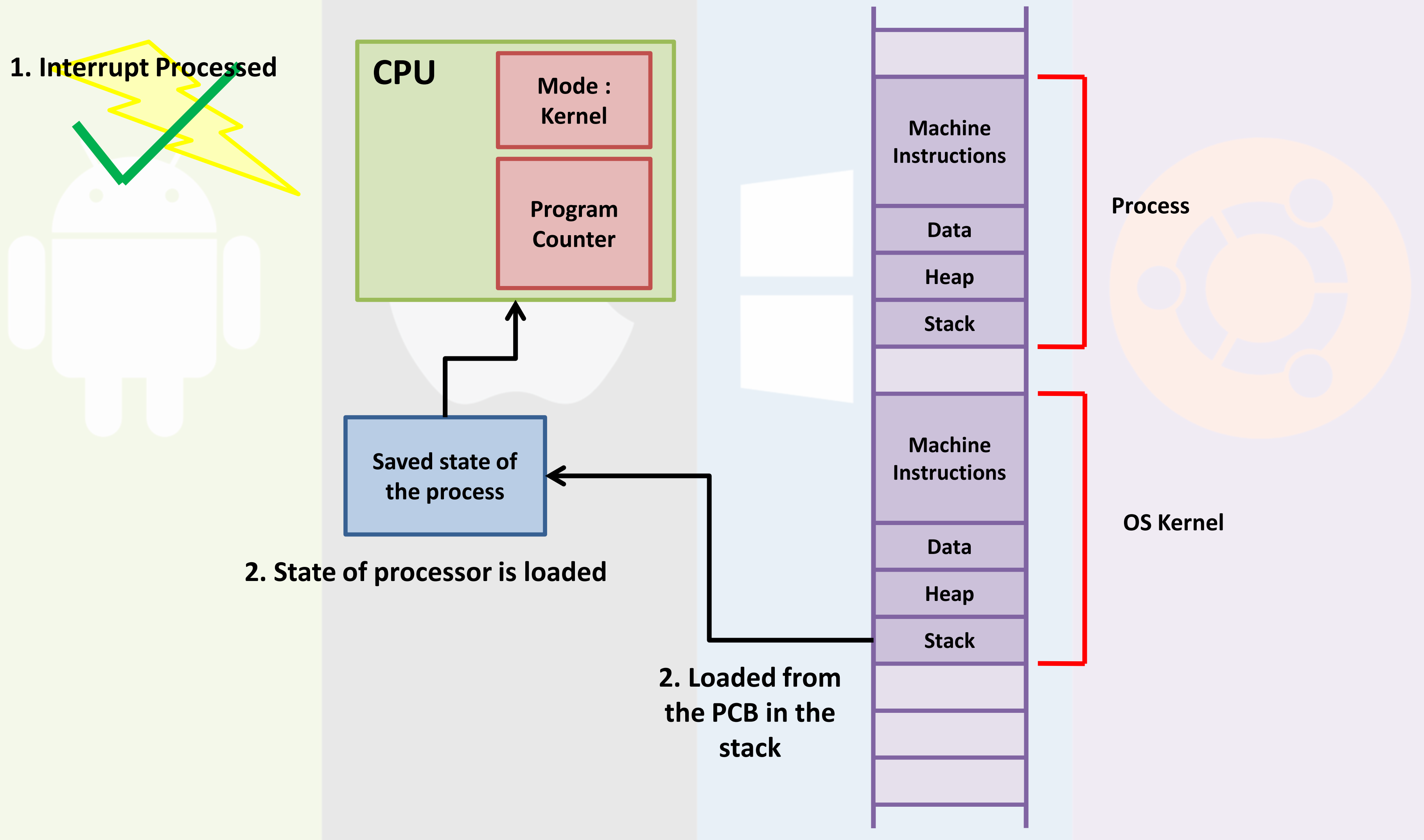


CPU
Mode : User
Program Counter
2. Interrupt Occurs
1. Instructions are processed
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
Saved state of the process
3. State of processor is saved
3. Stored in the PCB in the stack

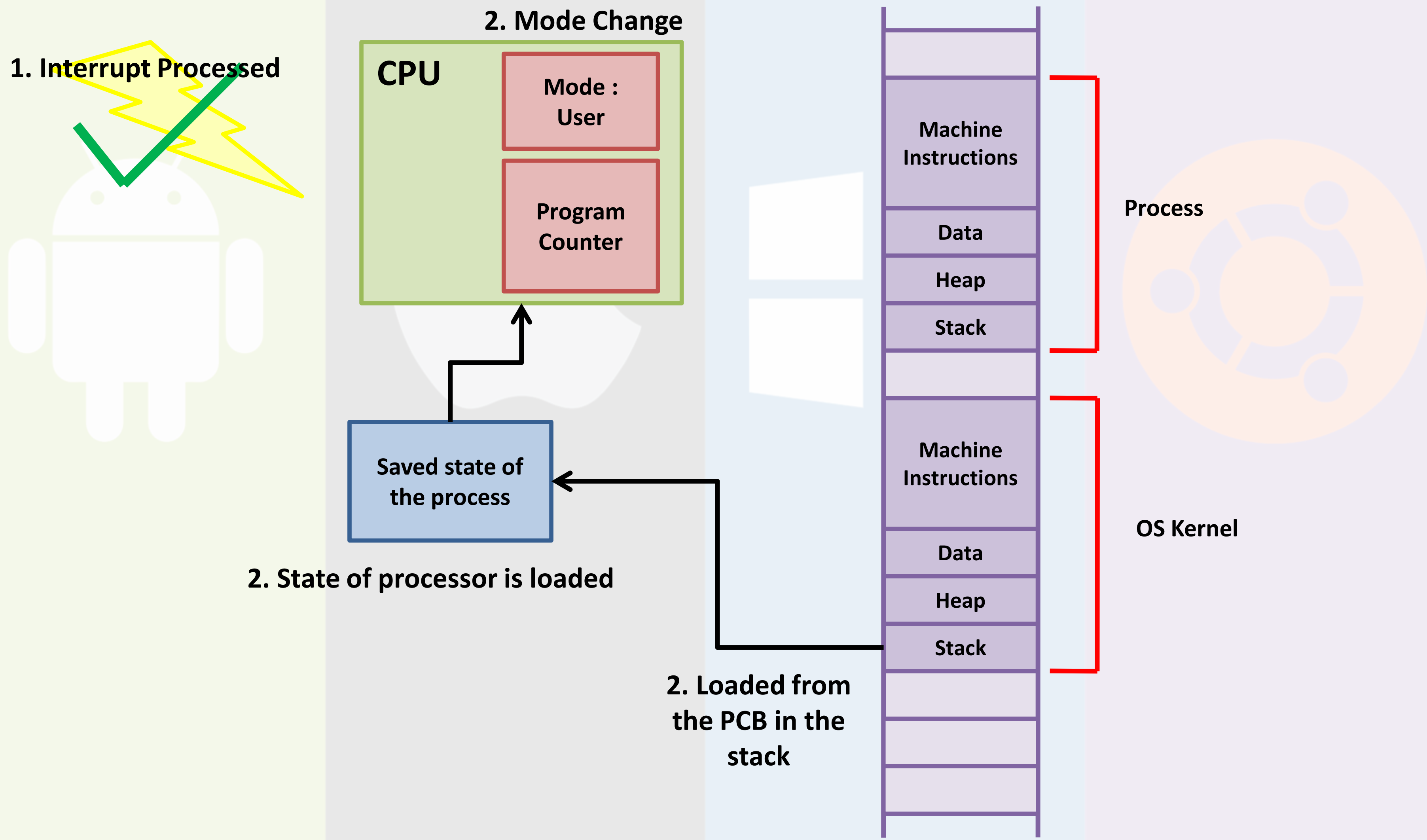
Summary


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

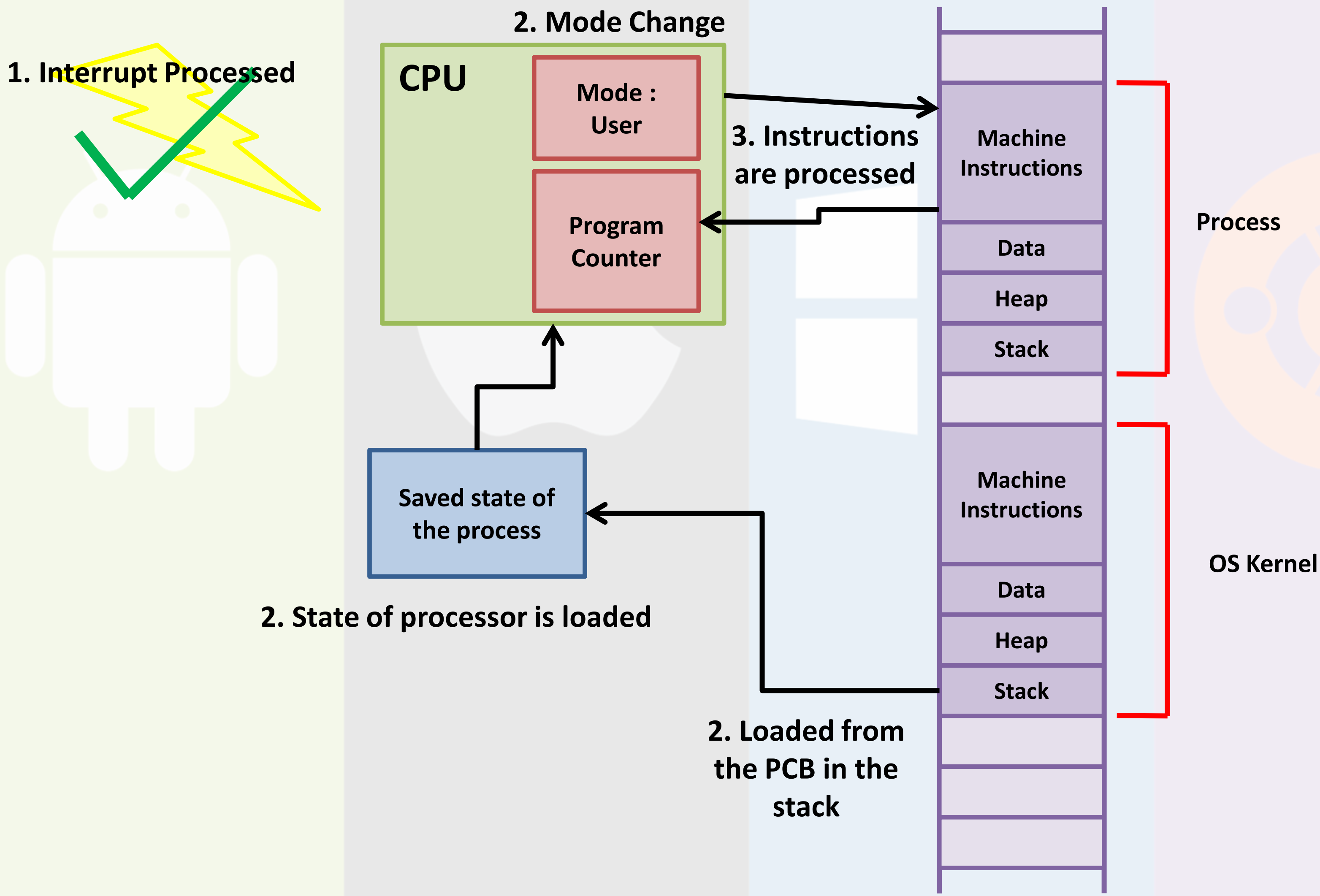

1. Interrupt Processed
CPU
Mode : Kernel
Program Counter
Saved state of the process
2. State of processor is loaded
Machine Instructions
Data
Heap
Stack
Process
Machine Instructions
Data
Heap
Stack
OS Kernel
2. Loaded from the PCB in the stack

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
Saved state of
the process
2. State of processor is loaded
2. Loaded from
the PCB in the
stack
Machine
Instructions
Data
Heap
Stack
Process
Machine
Instructions
Data
Heap
Stack
OS Kernel

Implementing Safe Mode Transfer

- **Care needs to be taken when implementing mode transfer to ensure malicious programs cannot gain access to the kernel**

Implementing Safe Mode Transfer

- Care needs to be taken when implementing mode transfer to ensure malicious programs cannot gain access to the kernel

- Limited entry
  - To transfer control to the kernel, the hardware must ensure that the entry point to the kernel is one setup by the kernel

Implementing Safe Mode Transfer

- Care needs to be taken when implementing mode transfer to ensure malicious programs cannot gain access to the kernel

- Atomic changes to processor state
    - Changing of program counter, mode, stack and memory protection are all changed at the same time

Implementing Safe Mode Transfer

- Care needs to be taken when implementing mode transfer to ensure malicious programs cannot gain access to the kernel

- Transparent, restartable execution
    - Interrupts should be invisible to the user processes

```c
#include <stdio.h>

int main()
{

    int i = 0;
    i = i/0;
    printf("Hello World");

    return 0;
}
```

Stop and Continue at the same point

Implementing Safe Mode Transfer

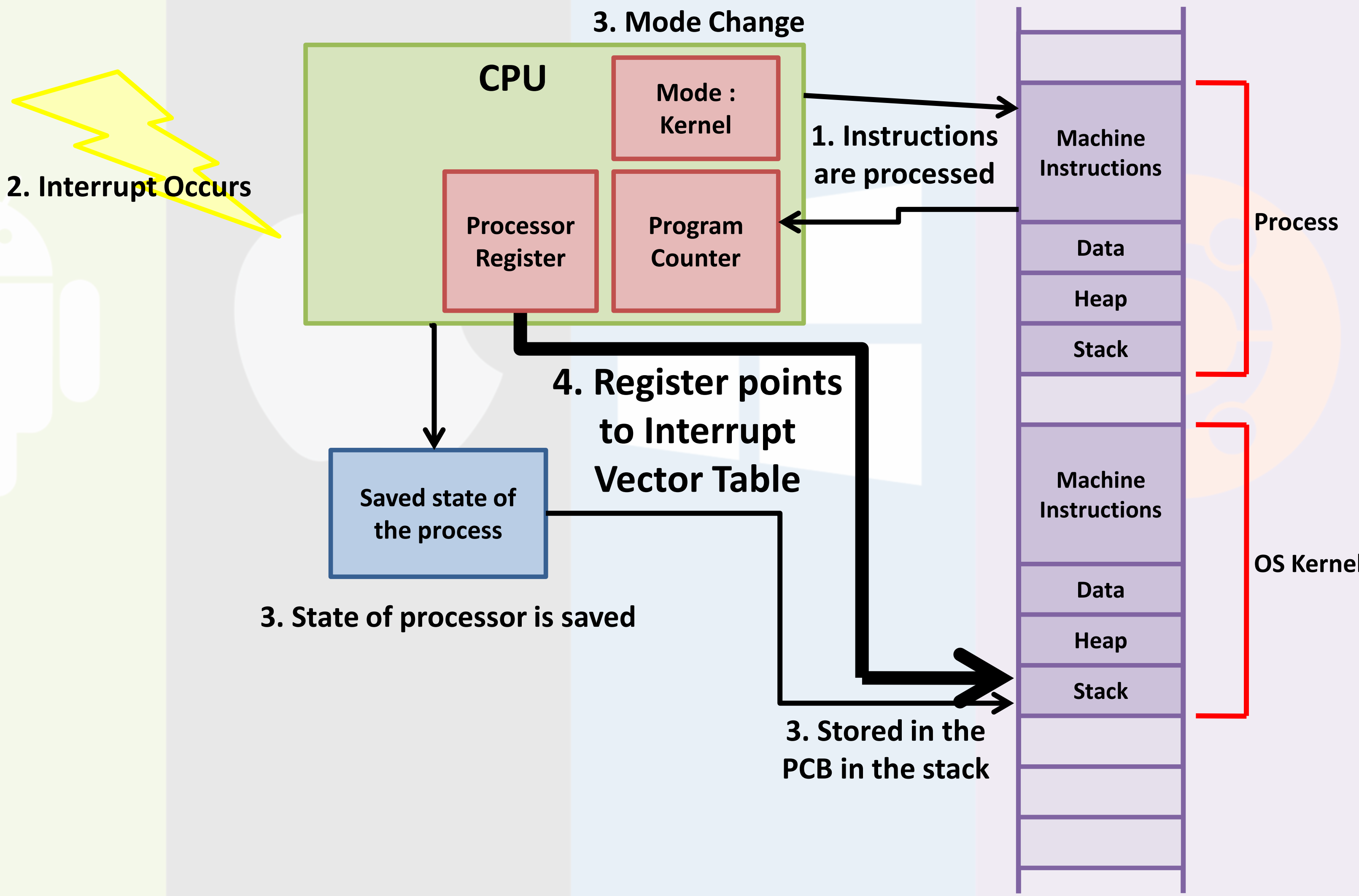
- On an **interrupt** the **processor saves** its **current state** to memory, changes to kernel mode and jumps to the exception/interrupt handler

System Calls

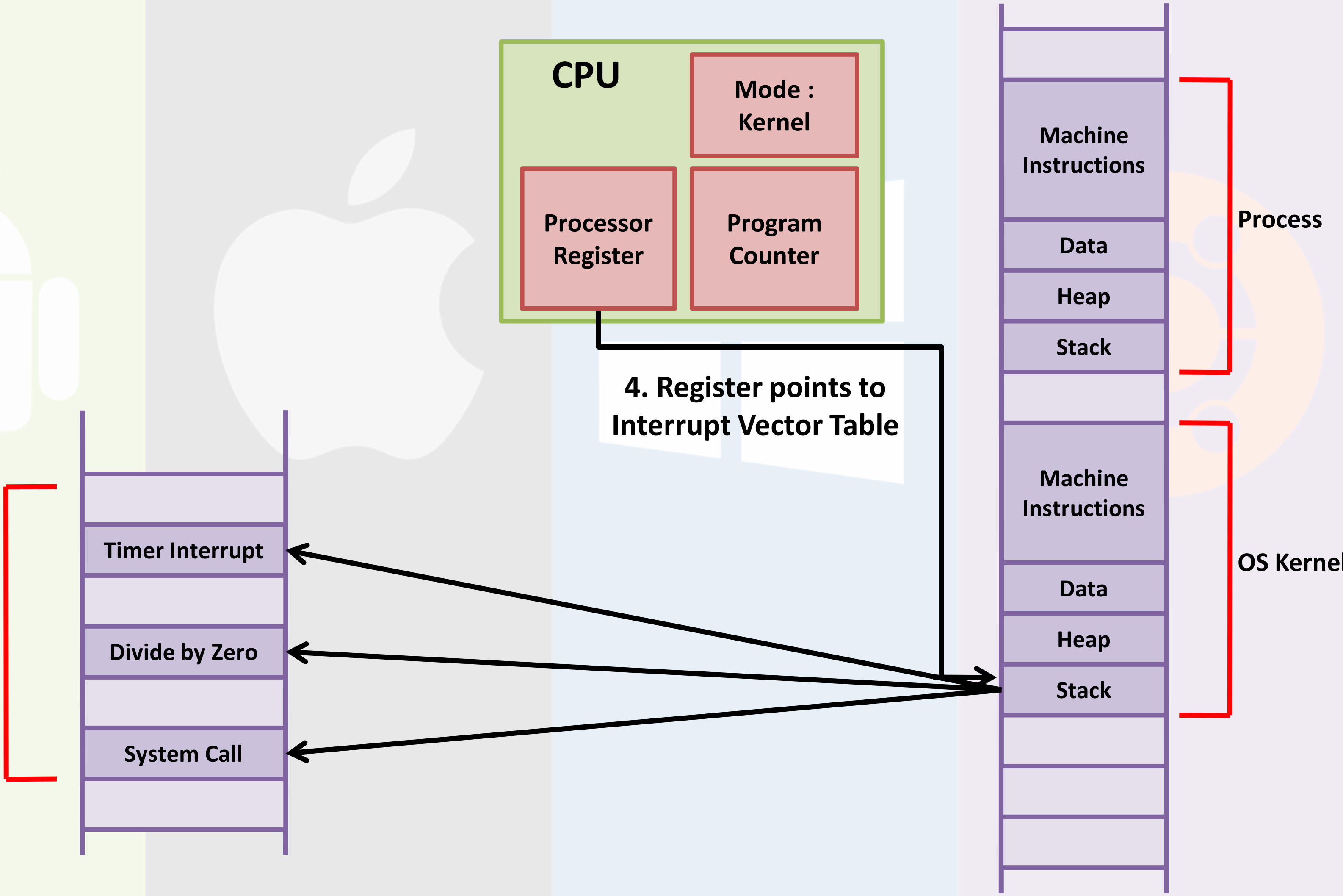
OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of … syscall handler	

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

Implementing Safe Mode Transfer



Implementing Safe Mode Transfer



<u>Interrupt Vector Table</u>

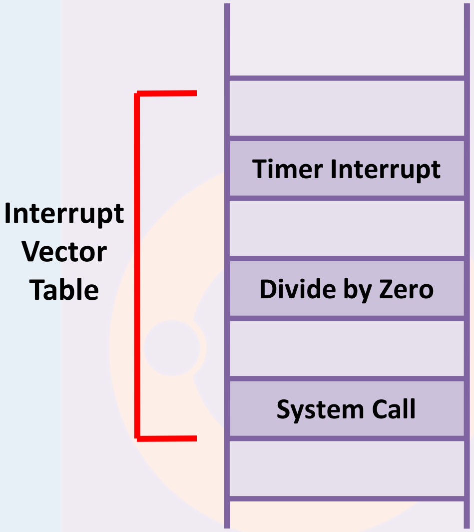
- The processor has a **special register** that in the event of an interrupt **points** to the corresponding **interrupt handler**.

- The handlers are **found** in a special part of a **kernels stack** memory called the **<u>Interrupt vector table</u>**

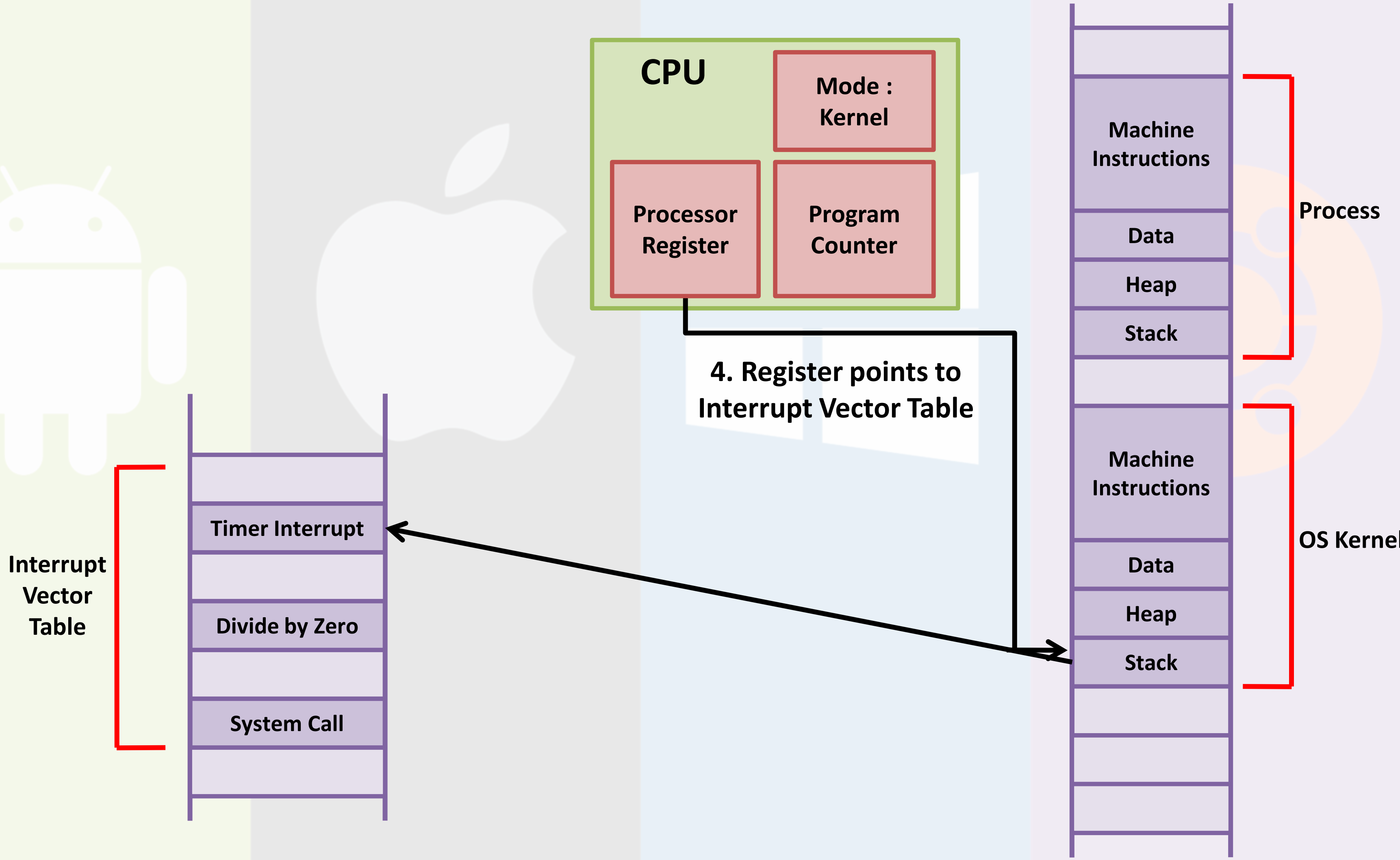
- The format of the vector is processor specific

- x86 Architecture
  - 0 - 31 exception handlers
  - 32 – 255 interrupt handlers

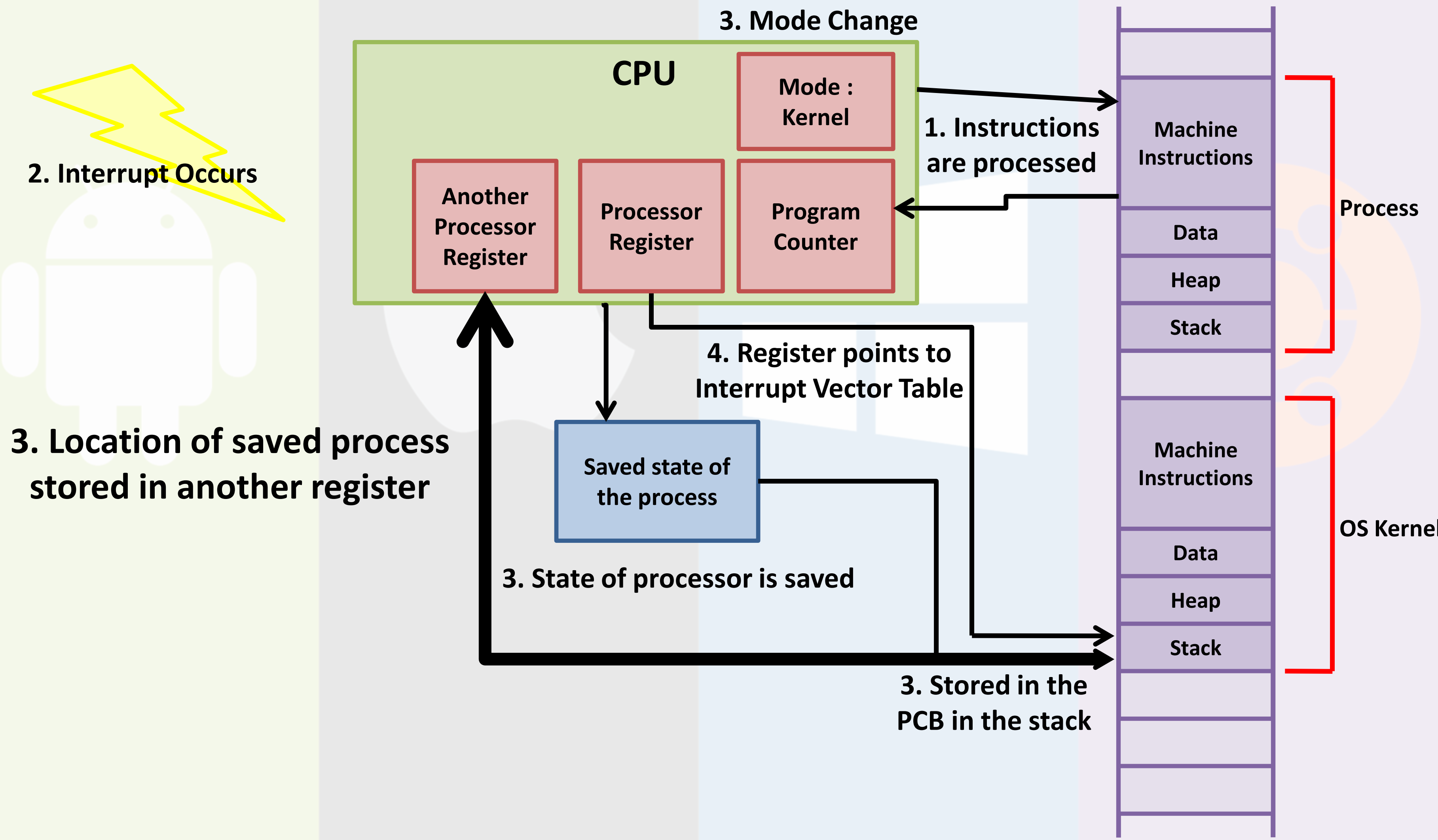
- **Since the hardware determines which device caused the exception the hardware will be able to use the correct handler**



Implementing Safe Mode Transfer



Interrupt Stack



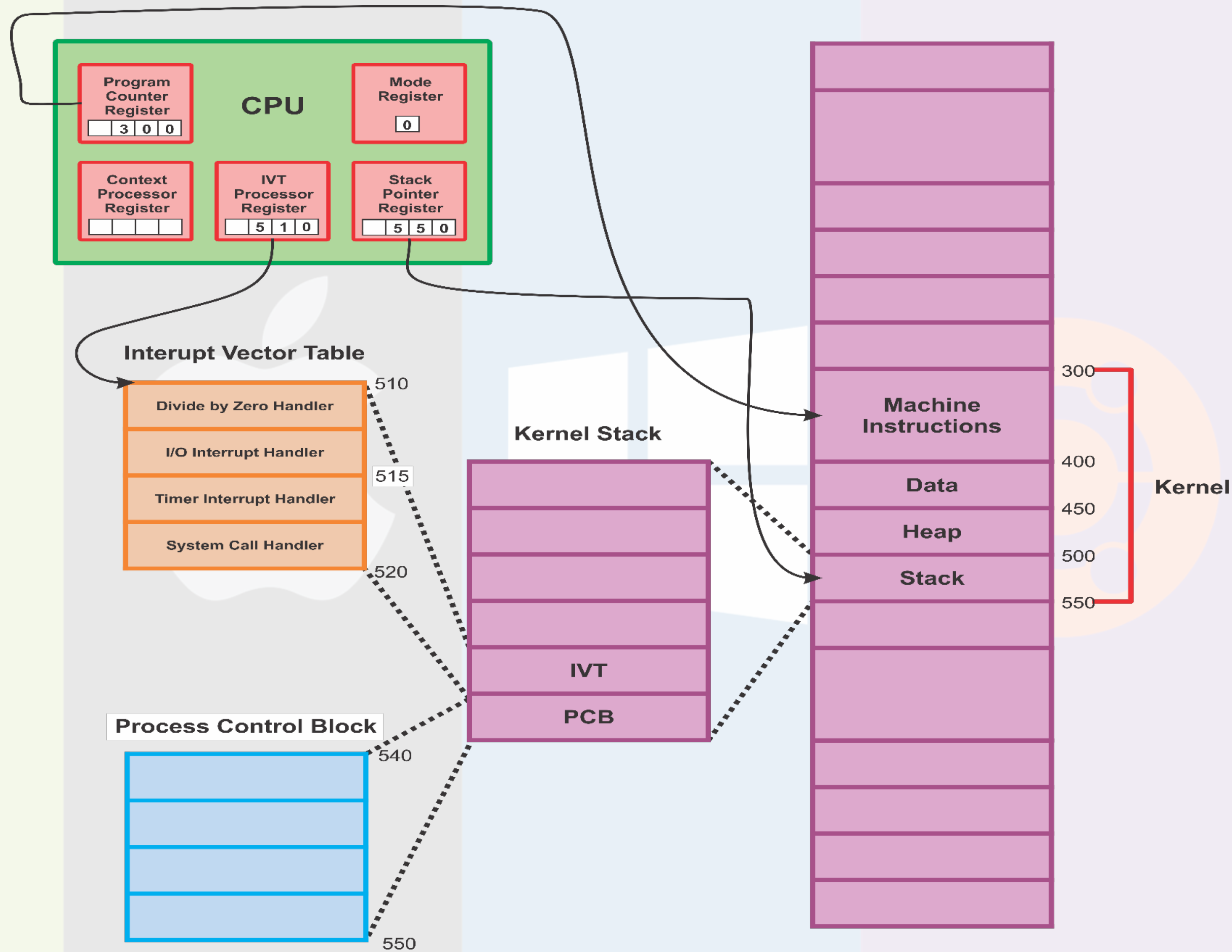
Interrupt Stack

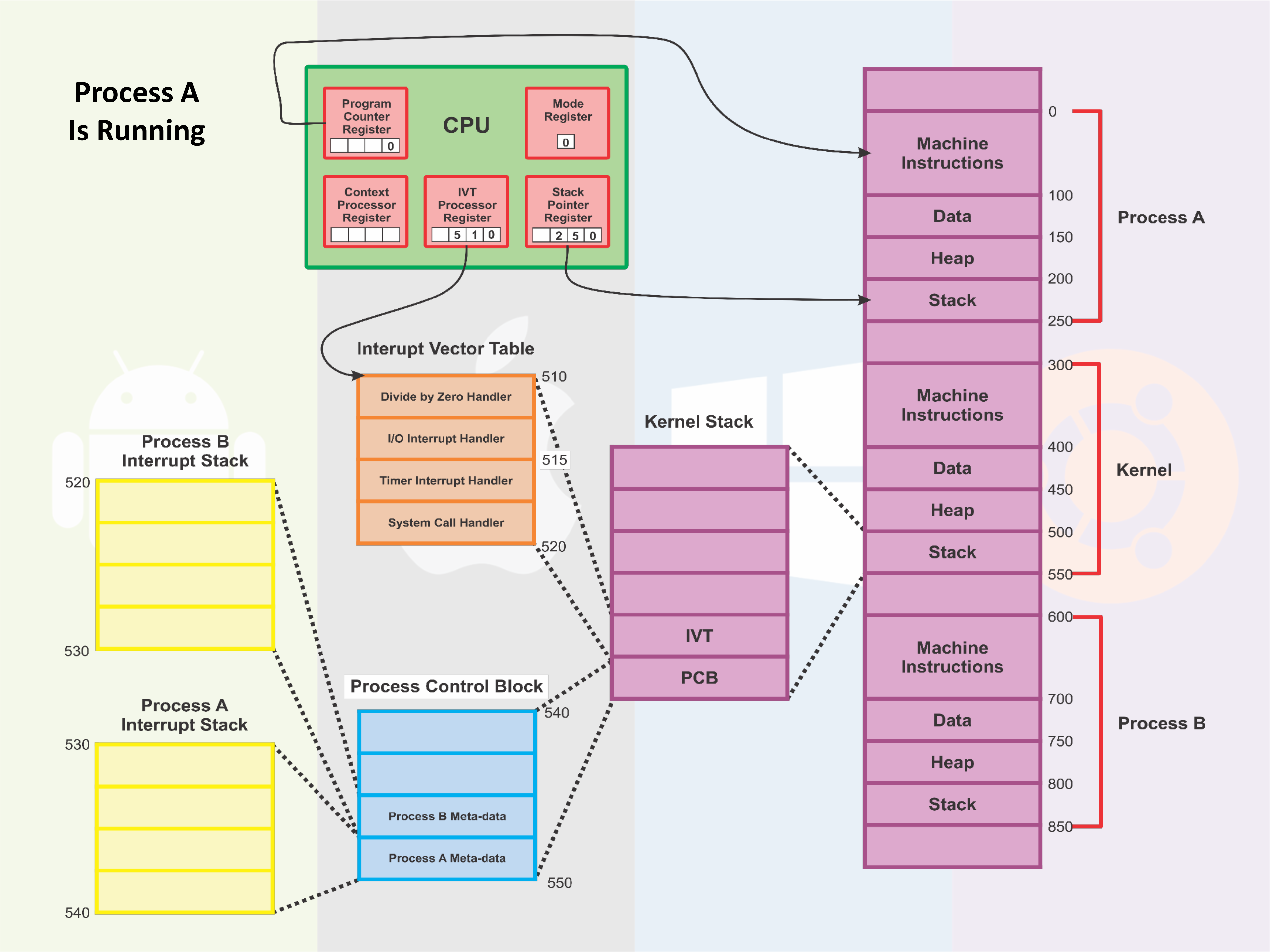
- Interrupt/Exception/System Call occurs
- Process halted
- Stack pointer set to base of kernel stack
- Hardware saves(pushes) process's registers to interrupt stack in kernel
- Control switched to kernel

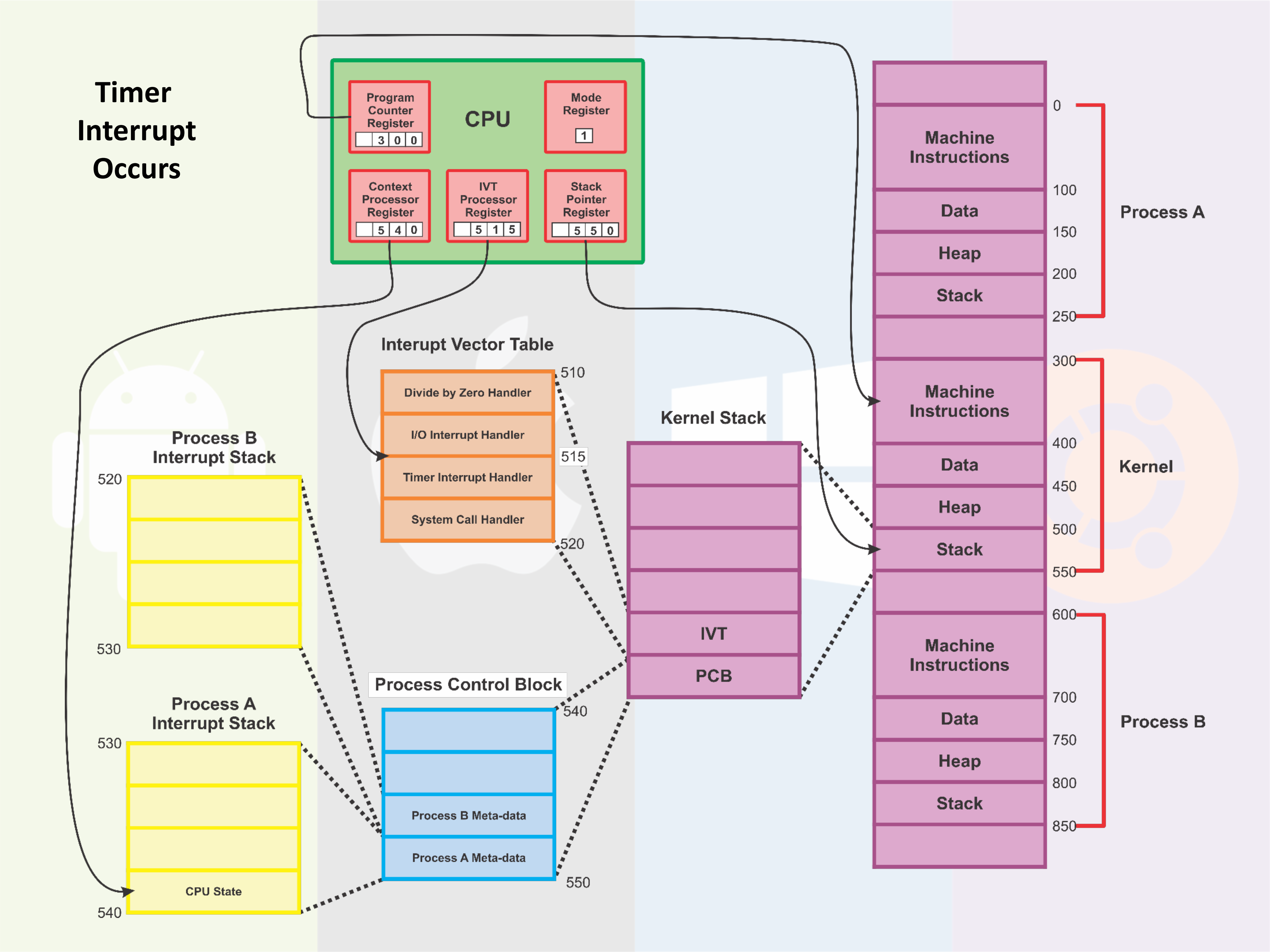
- Kernel does its job and handle event

- When returning from Interrupt/Exception/System Call
- The saved registers are loaded(popped) using the address stored in the CPU
- Process continues from where it left off

On Boot


CPU
Program Counter Register
3 0 0
Mode Register
0
Context Processor Register
IVT Processor Register
5 1 0
Stack Pointer Register
5 5 0
Interupt Vector Table
510
Divide by Zero Handler
I/O Interrupt Handler
515
Timer Interrupt Handler
System Call Handler
520
Process Control Block
540
550
Kernel Stack
IVT
PCB
Machine Instructions
Data
Heap
Stack
300
400
450
500
550
Kernel


Process A
Is Running
Program Counter Register
0
CPU
Mode Register
0
Context Processor Register
IVT Processor Register
5 1 0
Stack Pointer Register
2 5 0
Interupt Vector Table
510
Divide by Zero Handler
I/O Interrupt Handler
515
Timer Interrupt Handler
System Call Handler
520
Process B
Interrupt Stack
520
530
Process A
Interrupt Stack
530
540
Process Control Block
540
Process B Meta-data
Process A Meta-data
550
Kernel Stack
IVT
PCB
Machine Instructions
Data
Heap
Stack
0
100
150
200
250
300
400
450
500
550
600
700
750
800
850
Process A
Kernel
Process B


Timer Interrupt Occurs
CPU
Program Counter Register
3 0 0
Mode Register
1
Context Processor Register
5 4 0
IVT Processor Register
5 1 5
Stack Pointer Register
5 5 0
Interupt Vector Table
510
Divide by Zero Handler
I/O Interrupt Handler
515
Timer Interrupt Handler
System Call Handler
520
Process B Interrupt Stack
520
530
Process A Interrupt Stack
530
CPU State
540
Process Control Block
540
Process B Meta-data
Process A Meta-data
550
Kernel Stack
IVT
PCB
Machine Instructions
Data
Heap
Stack
0
100
150
200
250
Process A
300
400
450
500
550
Kernel
600
700
750
800
850
Process B

Why not just store it on the process stack?

- **Two reasons**

Why not just store it on the process stack?

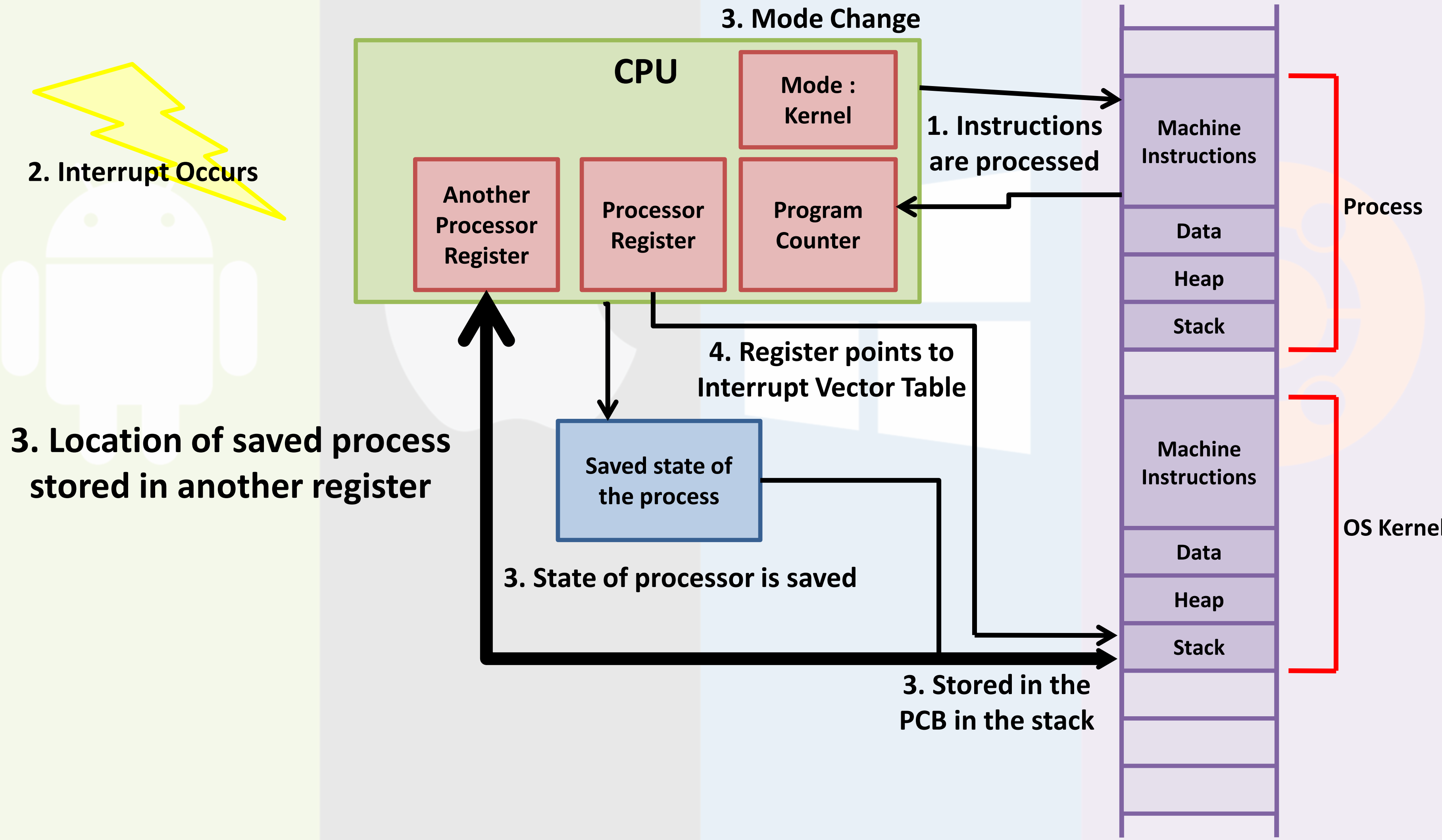
- Two reasons
  - Reliability – The process's user-level stack might be invalid due to some bug

Why not just store it on the user-level stack?

- **Two reasons**
    - Reliability – The process's user-level stack might be invalid due to some bug

    - **Security – Other threads in a multiprocessor running the same process can modify user memory during a system call. The user program might then be able to modify the kernel's return address**

Interrupt Stack



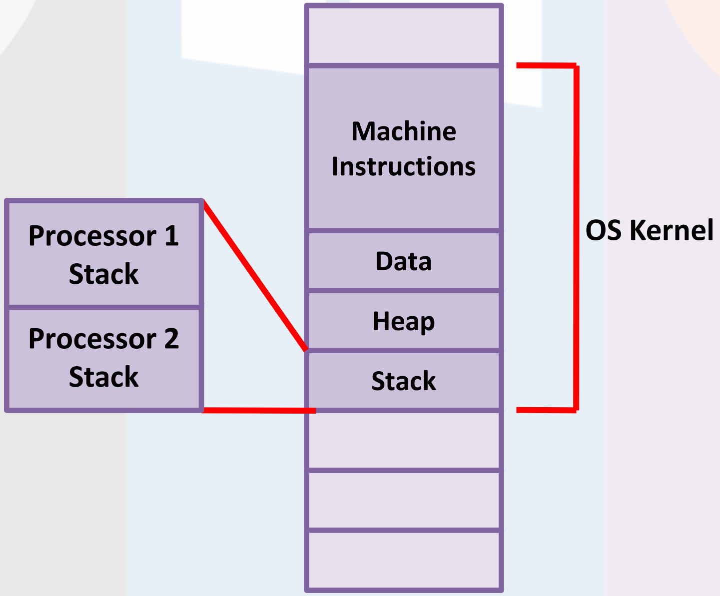
What happens when we have multiple processors?

- How can the kernel handle simultaneous system calls and exceptions across multiple processors?

What happens when we have multiple processors?

- How can the kernel handle simultaneous system calls and exceptions across multiple processors?

- **For each processor the kernel allocates a separate region of memory as that processors interrupt stack.**

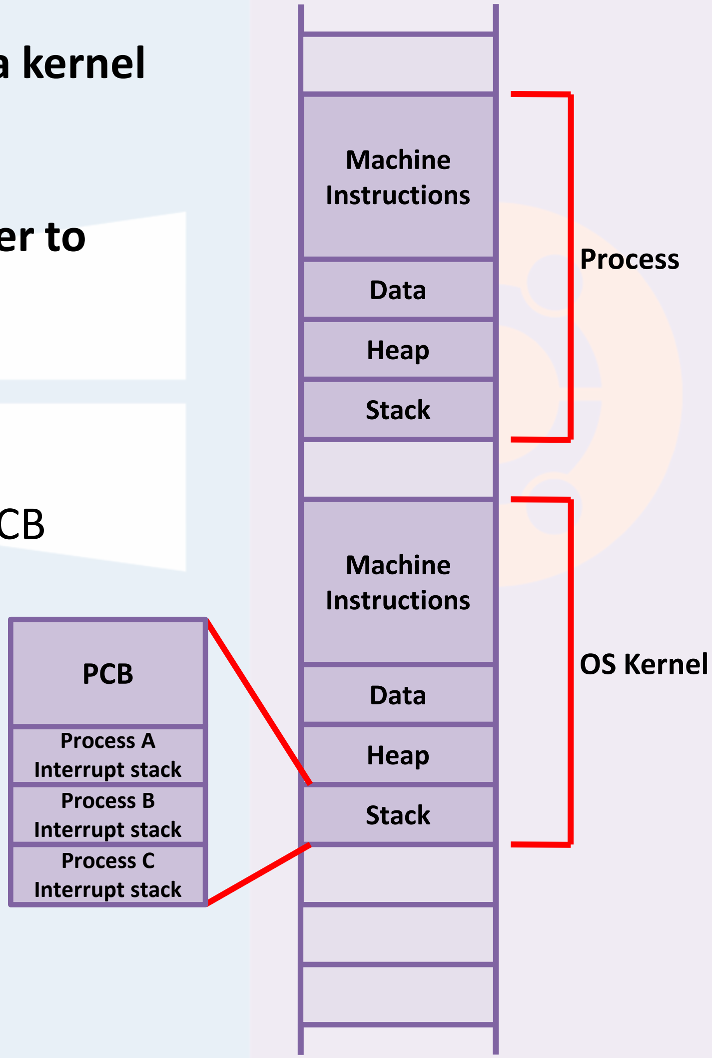


Two Stacks per Process

- **Most OS kernels go one step further and allocate a kernel interrupt stack per user process**

- **Allocating a kernel stack per process makes it easier to switch between processes**

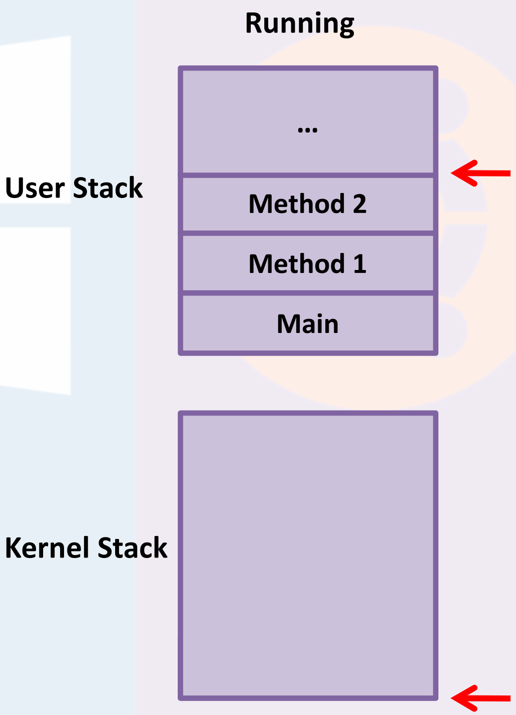
- **Now to switch from process A to B**
    - Stop processing A
    - Store a pointer to process A's kernel stack in PCB
    - Save state of CPU in kernel stack A
    - Clear registers
    - Read process B's kernel stack pointer found in PCB
    - Load state of kernel stack B into CPU
    - Process B's instructions



Two Stacks per Process

- If the process is running on the processor in user mode
  - Kernel stack empty
  - Ready for an interrupt

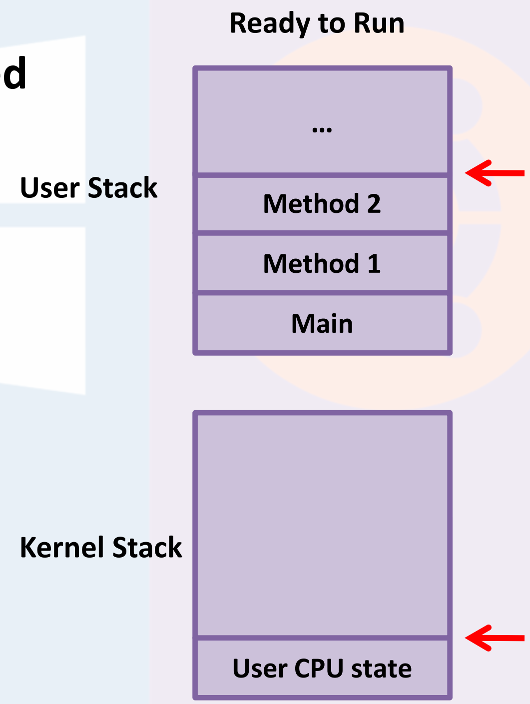
```
#include <stdio.h>
void method1(){
		method2();
}
void method2(){
		Printf("Nothing");
}
 int main() {
		method1();
 return 0;
}
```



Two Stacks per Process

- If the process is ready to run on the processor but is awaiting it turn
    - Kernel stack contains registers and state to be restored

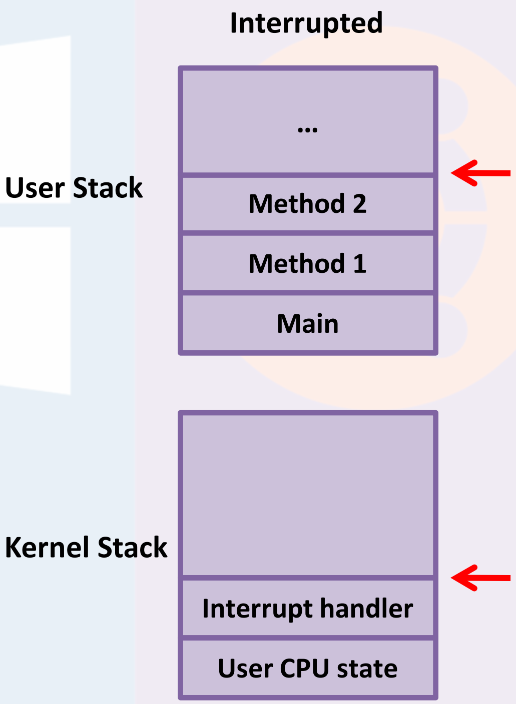
```
#include <stdio.h>
void method1(){
        method2();
}
void method2(){
        Printf("Nothing");
}
 int main() {
        method1();
 return 0;
}
```



Two Stacks per Process

- If the process is running on the processor in kernel mode due to an Interrupt/System call/Exception
    - Kernel stack is in use
    - Saved registers and state of suspended CPU
    - Current state of kernel handler

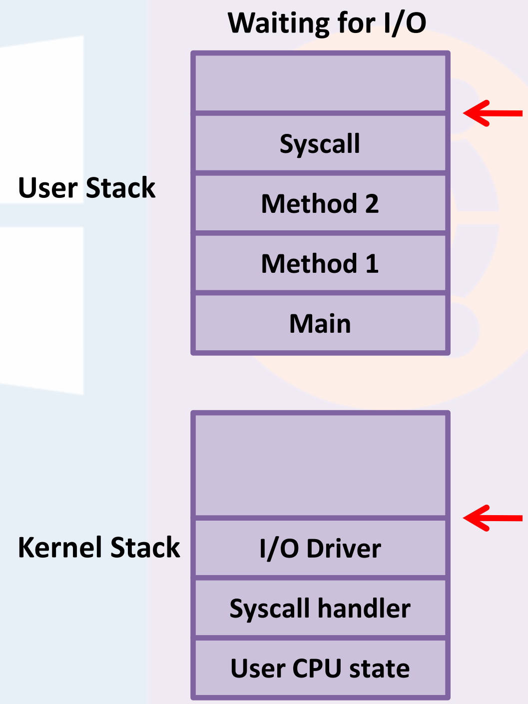
```
#include <stdio.h>
void method1(){
                method2();
}
void method2(){
                Printf("Nothing");
}
 int main() {
                method1();
 return 0;
}
```



Two Stacks per Process

- If the process is waiting for I/O event
  - Saved registers and state of suspended CPU
  - Current state of kernel handler

```c
#include <stdio.h>
void method1(){
		method2();
}
void method2(){
		char c;
		scanf("%c",&c);
}
 int main() {
		method1();
 return 0;
}
```



Interrupt Masking

- Can we interrupt the interrupted?

- Since interrupts happen asynchronously
    - We could get the event of an interrupt while the kernel is processing another
    - This could cause confusion

Interrupt Masking

- Can we interrupt the interrupted?

- Since interrupts happen asynchronously
  - We could get the event of an interrupt while the kernel is processing another
  - This could cause confusion

- **To simplify the kernel design provides a privileged instruction**
  - **This instruction temporarily defers deliveries of interrupts**
  - **on x86 infrastructure = “Disable Interrupts”**

- **Deferred not ignored**
  - **Once the corresponding “Enable Interrupts” instruction is executed, any pending interrupts are delivered**

What happens when multiple interrupts occur when disabled

- Stored in a buffer
- Delivered in turn when interrupts are re-enabled

- Limited buffering for interrupts
  - Some interrupts are lost if disabled too long
  - Generally only buffer one of each type

How do we prevent the same instruction from causing an exception ?

- If the handler returns back the instruction that caused the exception, the exception would immediately reoccur

```c
#include <stdio.h>

int main()
{

    int i = 0;
    i = i/0;
    printf("Hello World");

    return 0;
}
```

Division by 0

How do we prevent the same instruction from causing an exception ?

- If the handler returns back the instruction that caused the exception, the exception would immediately reoccur

- **To prevent this infinite loop**
  - **The exception handler modifies the program counter stored at the base of the stack to point to the instruction immediately after the one which caused the exception**

Implementing Secure System Calls

- **Voluntary mode switches from user to kernel**
- **Provide the illusion that the operating system kernel is simply a set of library routines which available to the user program**

- create a new process
- read from keyboard
- read/write from disk

Implementing Secure System Calls

- **To implement these system calls requires defining a calling convention**
  - how to name system calls
  - pass arguments
  - and receive return values across the user-kernel boundary

- **Once the arguments are in the correct format**
  - The user-level program can issue a system call by executing the trap instruction to transfer to the kernel mode

Implementing Secure System Calls

- **To implement these system calls requires defining a calling convention**
  - how to name system calls
  - pass arguments
  - and receive return values across the user-kernel boundary

- **Once the arguments are in the correct format**
  - The user-level program can issue a system call by executing the trap instruction to transfer to the kernel mode

- **Once inside the kernel**
  - A handler handles each system call
  - These handlers are implemented in a way that protects the kernel from any errors or attacks

Implementing Secure System Calls

- **To implement these system calls requires defining a calling convention**
    - how to name system calls
    - pass arguments
    - and receive return values across the user-kernel boundary

- **Once the arguments are in the correct format**
    - The user-level program can issue a system call by executing the trap instruction to transfer to the kernel mode

- **Once inside the kernel**
    - A handler handles each system call
    - These handlers are implemented in a way that protects the kernel from any errors or attacks

- **We bridge the divide between user calling a system call and the kernel implementing the system call with a pair of stubs**
- **A pair of stubs is a pair of procedures which mediate between two environments**

Implementing Secure System Calls

- **Step 1 : The user process makes a normal procedure call to a stub linked**

User-Level Process

```
main(){
    file_open(arg1,arg2)
}
```

1

User Stub

```
File_open(arg1,arg2){
    push #SYSCALL_OPEN
    trap
    return
}
```

Kernel

```
File_open(arg1,arg2){
    //do operation
}
```

Kernel Stub

```
File_open(arg1,arg2){
    // copy arguments from user memory
    // check validity of arguments
    file_open(arg1,arg2)
    //copy return value into memory
    return;
}
```

Implementing Secure System Calls

- **Step 2: The stub executes the trap instruction. This transfers control to the kernel trap handler. The trap handler copies and checks its arguments.**

User-Level Process

```
main(){
    file_open(arg1,arg2
}
```

1

User Stub

```
File_open(arg1,arg2){
    push #SYSCALL_OPEN
    trap
    return
}
```

2 Hardware Trap

Kernel

```
File_open(arg1,arg2){
    //do operation
}
```

Kernel Stub

```
File_open(arg1,arg2){
    // copy arguments from user memory
    // check validity of arguments
    file_open(arg1,arg2)
    //copy return value into memory
    return;
}
```

Implementing Secure System Calls

- **Step 3: Kernel stub calls the kernel implementation of the system call, to do the operation**

User-Level Process

```
main(){
    file_open(arg1,arg2
}
```

1

User Stub

```
File_open(arg1,arg2){
    push #SYSCALL_OPEN
    trap
    return
}
```

2 Hardware Trap

Kernel Stub

```
File_open(arg1,arg2){
    // copy arguments from user memory
    // check validity of arguments
    file_open(arg1,arg2)
    //copy return value into memory
    return;
}
```

3

Kernel

```
File_open(arg1,arg2){
    //do operation
}
```

Implementing Secure System Calls

- **Step 4: The code returns to the trap handler which copies the return value into the user memory**

User-Level Process

```
main(){
    file_open(arg1,arg2
}
```

1

User Stub

```
File_open(arg1,arg2){
    push #SYSCALL_OPEN
    trap
    return
}
```

2 Hardware Trap

Kernel

```
File_open(arg1,arg2){
    //do operation
}
```

3

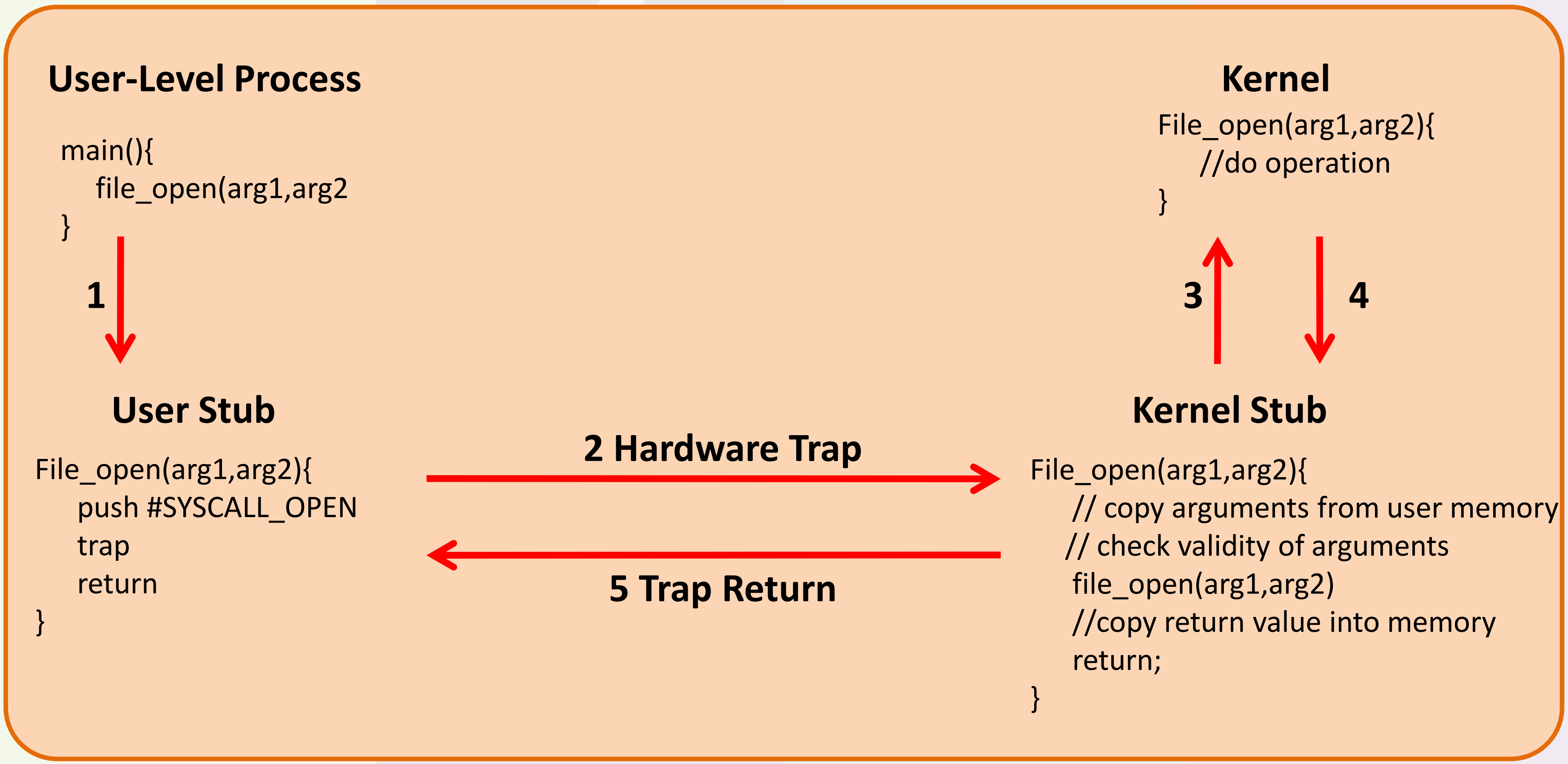
4

Kernel Stub

```
File_open(arg1,arg2){
    // copy arguments from user memory
    // check validity of arguments
    file_open(arg1,arg2)
    //copy return value into memory
    return;
}
```

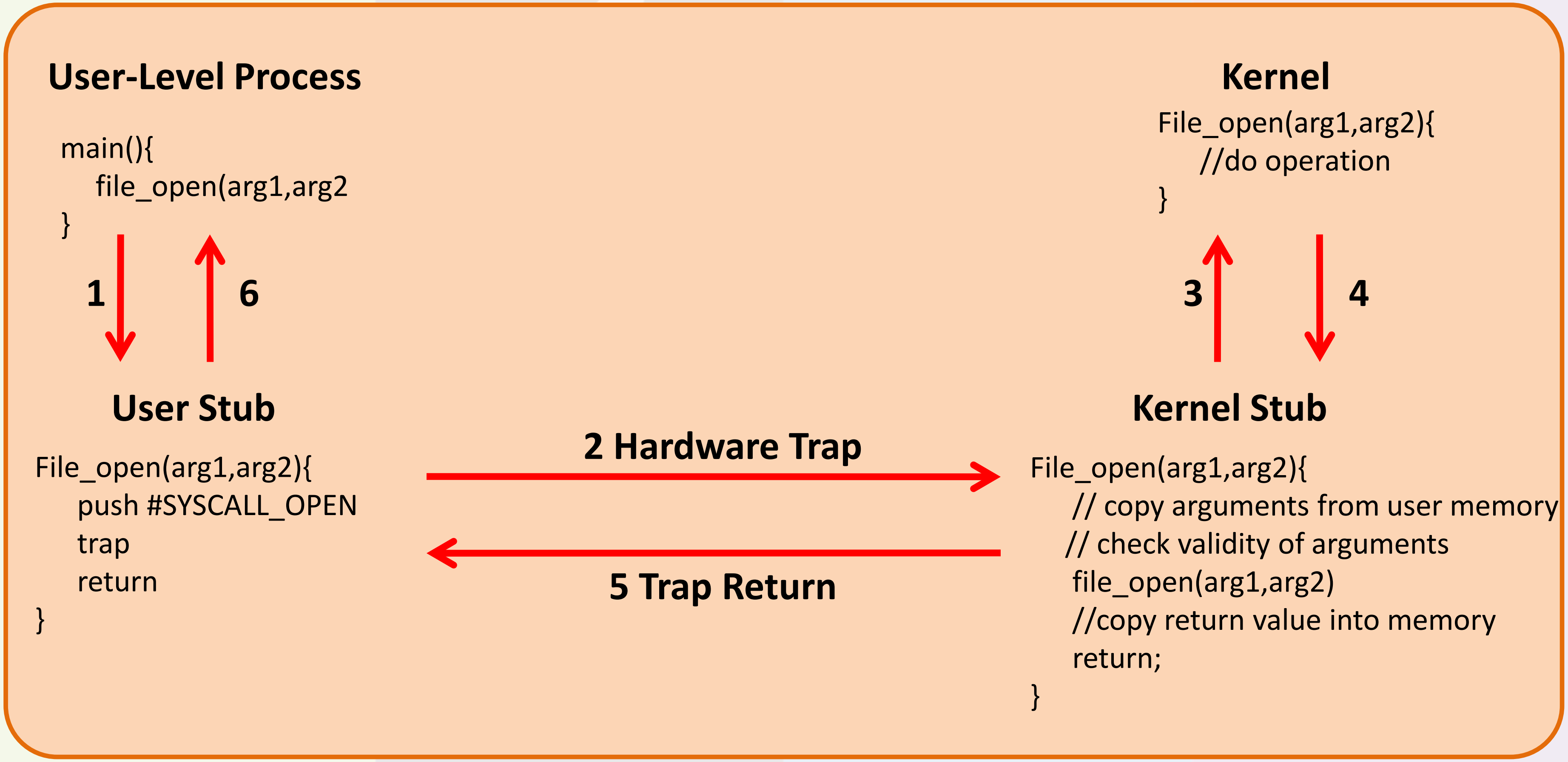
Implementing Secure System Calls

- **Step 5: The handler returns to the user level at the next instruction in the stub**



Implementing Secure System Calls

- **Step 6: The user stub returns to the user-level caller**



Kernel Stub

- The Kernel Stub has four tasks to perform before the kernel can do the operation

Kernel Stub

- **Locate System Call arguments**
    - The arguments are stored in user memory unlike a regular kernel procedure
    - If the system call has a pointer argument
        - The stub must check if the address is within the user domain
    - Stub converts it to a physical address that the kernel can safely use

Kernel Stub

- **Copy Before Check**
  - Kernel copies system call parameters into kernel memory before performing the necessary checks
  - This is to ensure that the application cant modify the parameters after the stub checks but before the parameter is actually used
  - Time of use vs time of check attack (TOCTOU)
  - This happens when multiple processes share memory
    - One process traps into the kernel
    - The other modifies the parameters

Kernel Stub

- **Validate Parameters**
  - The kernel must also protect itself against malicious or accidental errors in format or content of its arguments
  - A filename is typically a zero-terminated string, however, the kernel can't rely on user code to always work correctly
  - The filename might point to regions outside the applications region
  - Half the file might be stored within and the other half might exceed beyond
  - File may not even exist
  - If an error occurs it is returned to the user
  - If not the kernel performs the operation

Kernel Stub

- **Copy Back Any Results**
    - For the user program to access the results of the system call, the stub must copy the result of the kernel back into user memory
    - Again the kernel stub must check the address

Implementing Upcalls

- For many of the same reasons that kernels need interrupt based event delivery, applications can benefit from being told when events that need their immediate attention occur

- We have virtualized a component of the system to provide functionality to user programs, we apply the same technique here

- Virtualized interrupts and exceptions are called “Upcalls”

Uses of Upcalls

- **Preemptive user-level threads**
  - Just as OS runs multiple processes on a single processor so too can an application run multiple tasks or threads in a process
  - A user-level thread package could use upcalls,
    - switch between tasks
    - stop a runaway task
    - e.g. A web browser terminating a nonresponsive embedded script

Uses of Upcalls

- **Asynchoronous I/O Notification**
    - Most system calls wait until the operation is completed, what happens if the process has other things to do in the meantime
    - You can utilise an upcall to poll the kernel for I/O completion
    - An upcall can also be used to send a notification to the application when the I/O completes

Uses of Upcalls

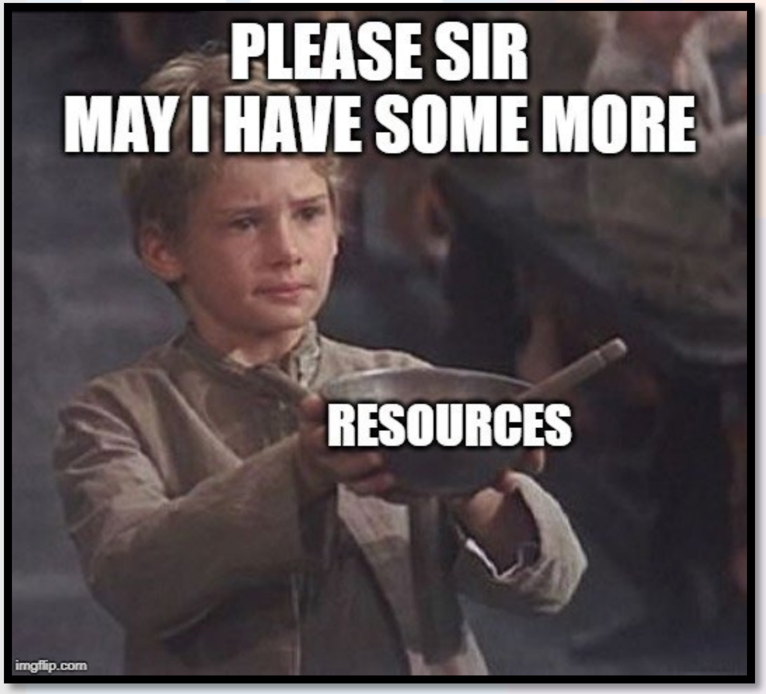
- **Interprocess communication**
    - Most interprocess communication can be handled with system calls
    - However, what happens when an application needs the instant attention of another
    - e.g. A debugger needs to suspend or resume the process
    - Here we can utilise upcalls to notify the process that the debugger wants to suspend

Uses of Upcalls

- **User-Level exception handling**
    - We have described processor exceptions such as divide by zero
    - What happens when the application has their own exception handling routines
    - Upcalls are utilised so that the kernel can inform the application that it must use its own handlers rather than the kernel

Uses of Upcalls

- **User-Level Resource Allocation**
  - OS allocates resources, deciding which process/user gets how much
  - Many applications are also resource adaptive, able to optimize their behaviour based on the resources available
  - Java garbage collector, the more resources available the fewer amount of times the garbage collector is run

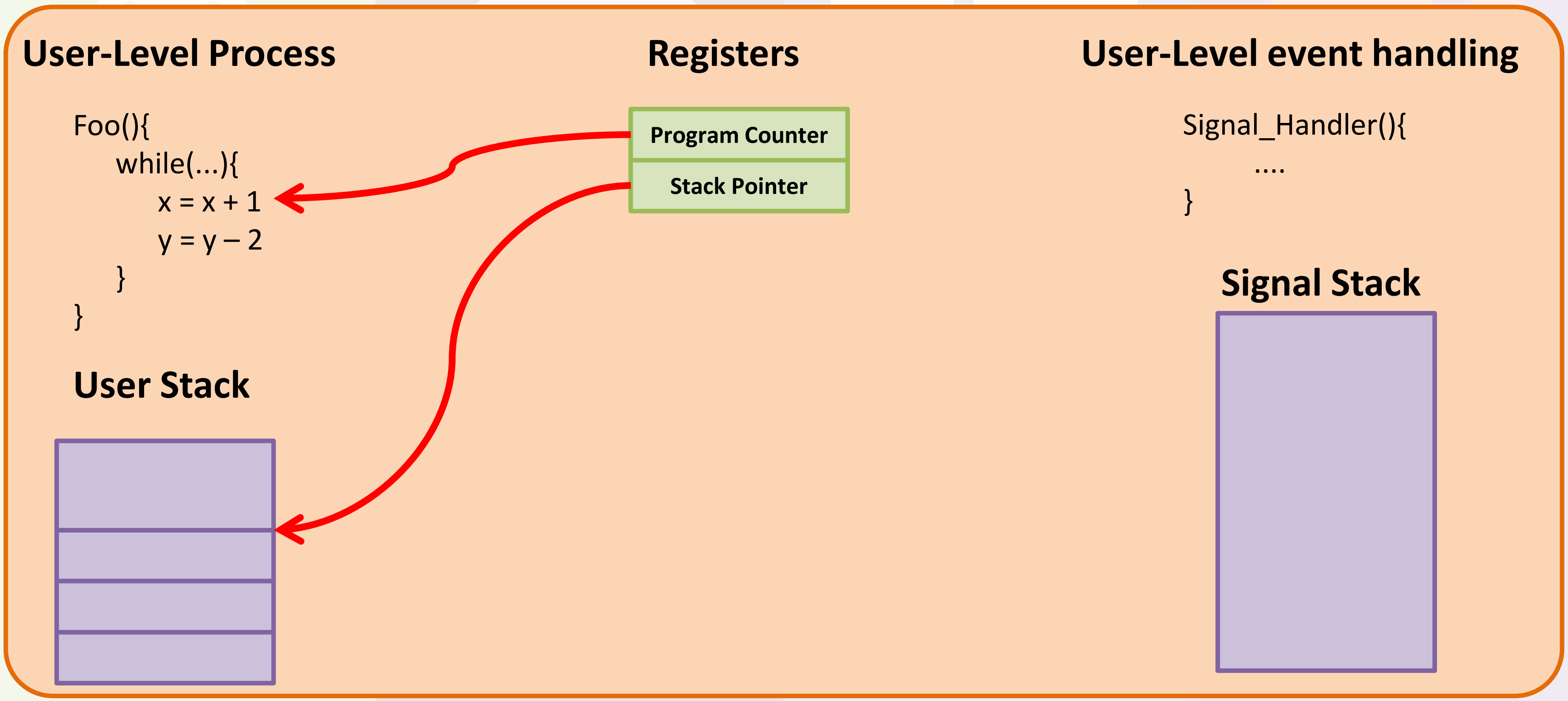


Upcalls

- **The virtualized interrupts share similarities to the hardware interrupts**
  - **Types of signals :** in place of hardware exceptions the kernel defines a limited number of signal types
  - **Handlers :** Each process defines its own handlers for each signal in a similar way as the interrupt vector table works
  - **Signal Stack :** Special stack for event handling is similar to that found in the kernel
  - **Signal Masking :** signals are deferred in the same way as in hardware interrupts by disabling interrupts
  - **Processor State :** The kernel copies onto the signal stack the saved state of the program counter, stack pointer and all other registers

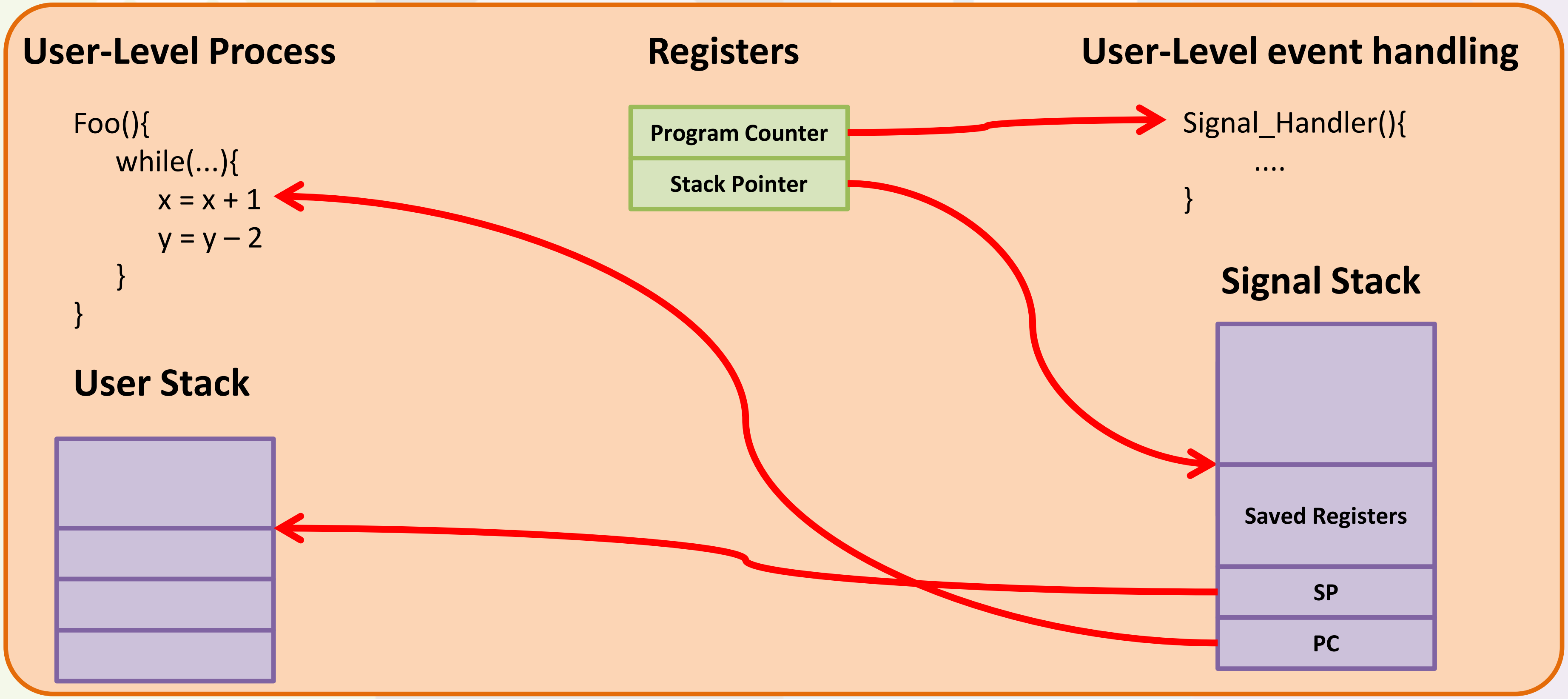
Upcalls

- The state of a user program and signal stack **before a UNIX signal(Upcall)**
- Signals behave analogously to processor exceptions but at user level



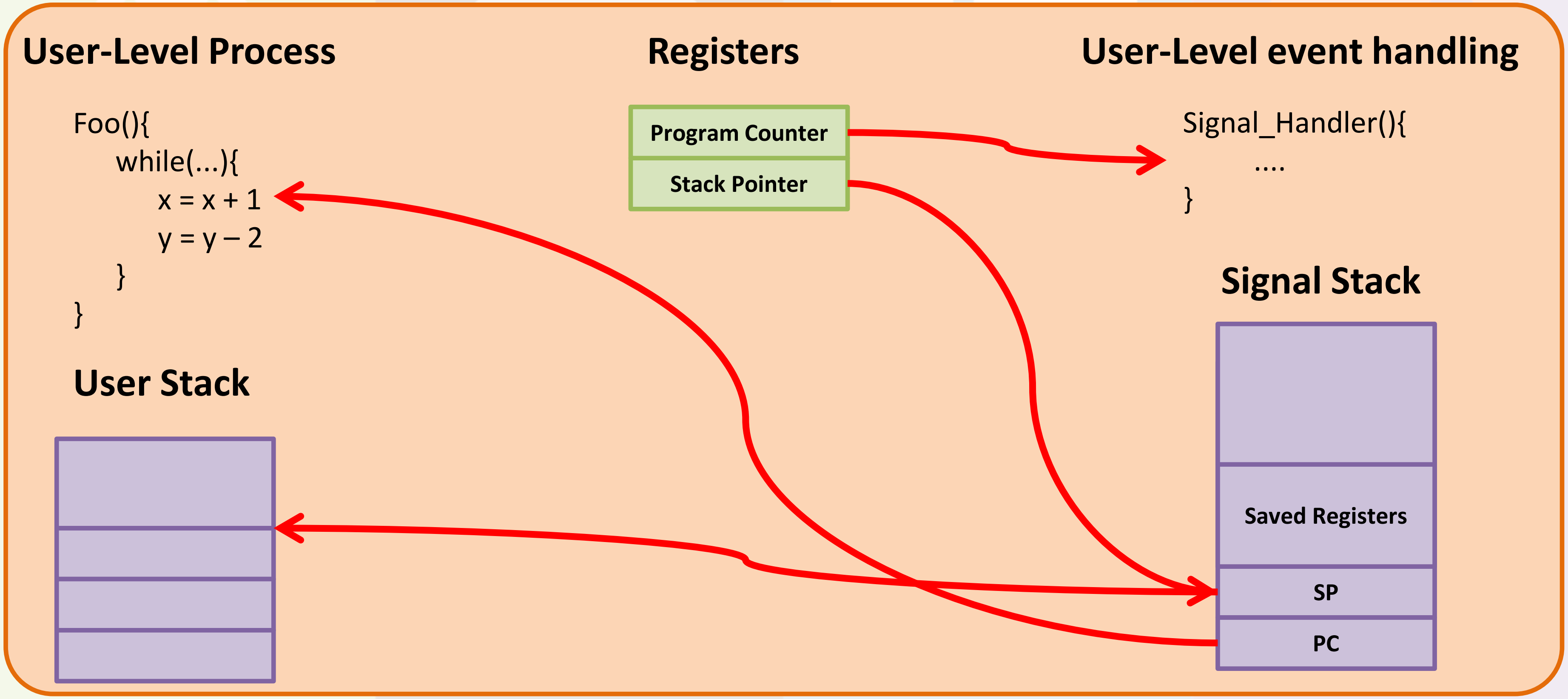
Upcalls

- The state of a user program and signal stack during a UNIX signal(Upcall)
- The signal stack stores the state of the registers at point when process interrupted
- With room for the signal handler to operate on the signal stack



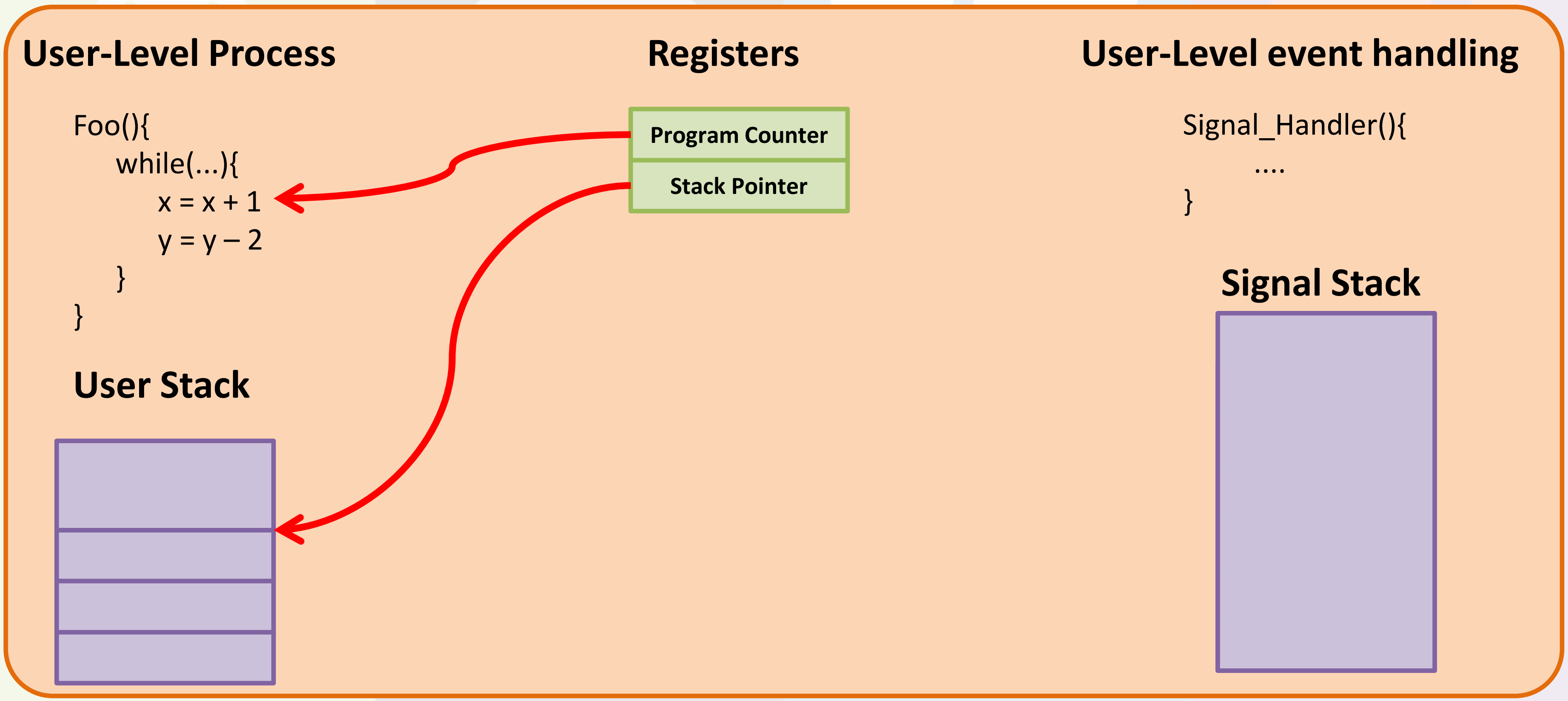
Upcalls

- The state of a user program and signal stack as Signal Processing finishes



Upcalls

- The state of a user program and signal stack after a UNIX signal(Upcall)



UNIX

- Unix is a family of multitasking, multiuser computer operating systems that derive from the original AT&T Unix

- Formed the foundation of Linux/ Mac0S

- We will be looking at a subset of the programming interface of UNIX
    - Process Management
    - I/O Control

Process Management

- Traditionally in the early batch processing systems, the kernel handled process management by necessity

- A different approach that was developed was one that allowed users programs to create and manage their own processes
    - Web browsers managing embedded scripts
    - Window Manages managing various windows

- An early motivation for user-level process management was to allow developers to write there own shell command line interpreters

Shell

- A shell is a job control system
  - Windows and Unix both have a shell

- Many tasks involve a sequence of steps each of which could be there own program

- With a shell you can write down the sequence of steps, as a sequence of programs to run

Shell

- A shell is a job control system
    - Windows and Unix both have a shell

- Many tasks involve a sequence of steps each of which could be there own program

- With a shell you can write down the sequence of steps, as a sequence of programs to run

- For example Makefiles are utilised to compile multiple C programs

- Makefiles are an example of a shell

- The C compiler itself is a shell program
    - The compiler first invokes a process to expand header include files
    - Separate process parses the output
    - Another process to convert to Assembly code
    - Lastly a process to convert Assembly into executable machine instructions

How does Windows handle process management?

- Windows simply has system calls to handle process operations

- For example their is specific system call to handle the creation of processes
    - CreateProcess(char *prog, char *args)

- Simple in Theory
- Complex in Practice

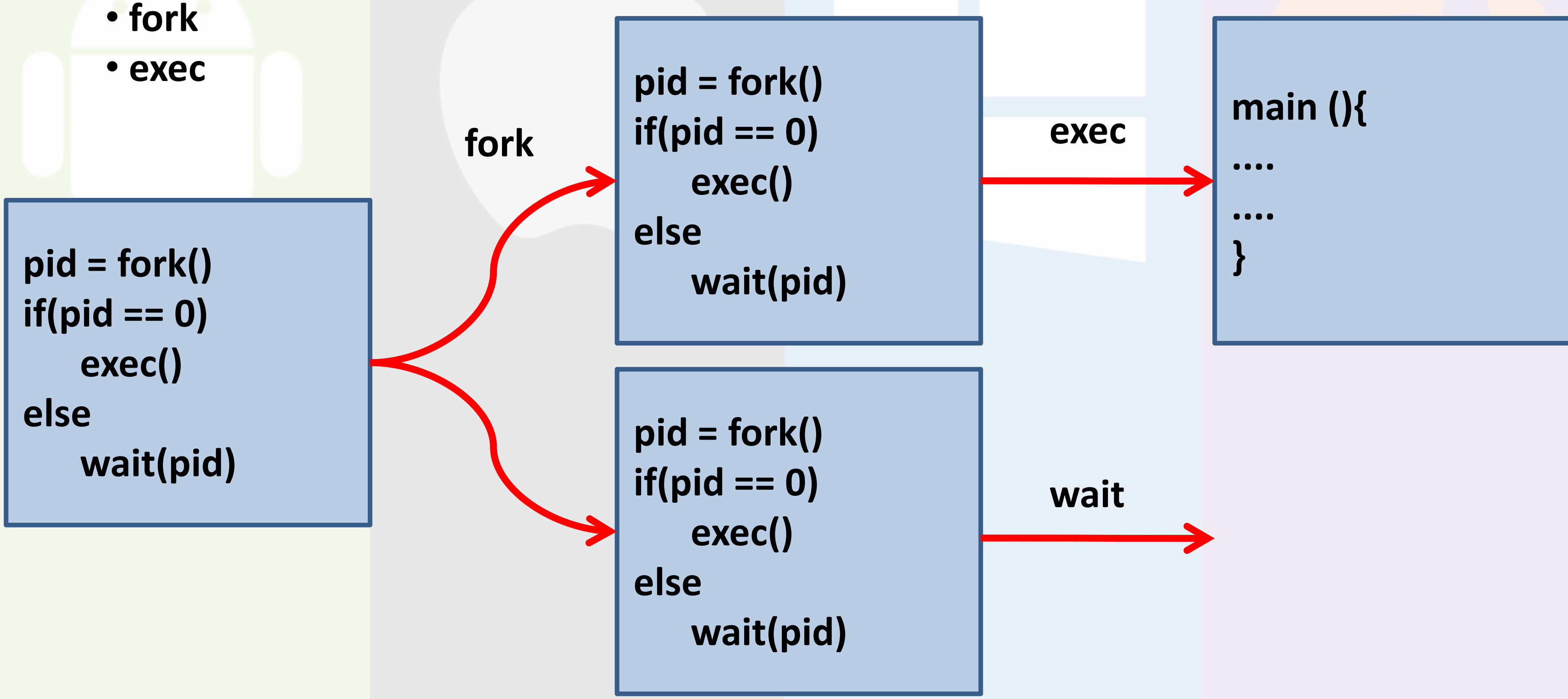
What steps does CreateProcess() take

- Create and initialise PCB in the kernel
- Create and initialise a new address space
- Load the program into the address space
- Copy the arguments into the memory in the address space
- Initialise the hardware context to start execution at the start of the program
- Inform the scheduler that a new process is ready

How does UNIX handle process management?

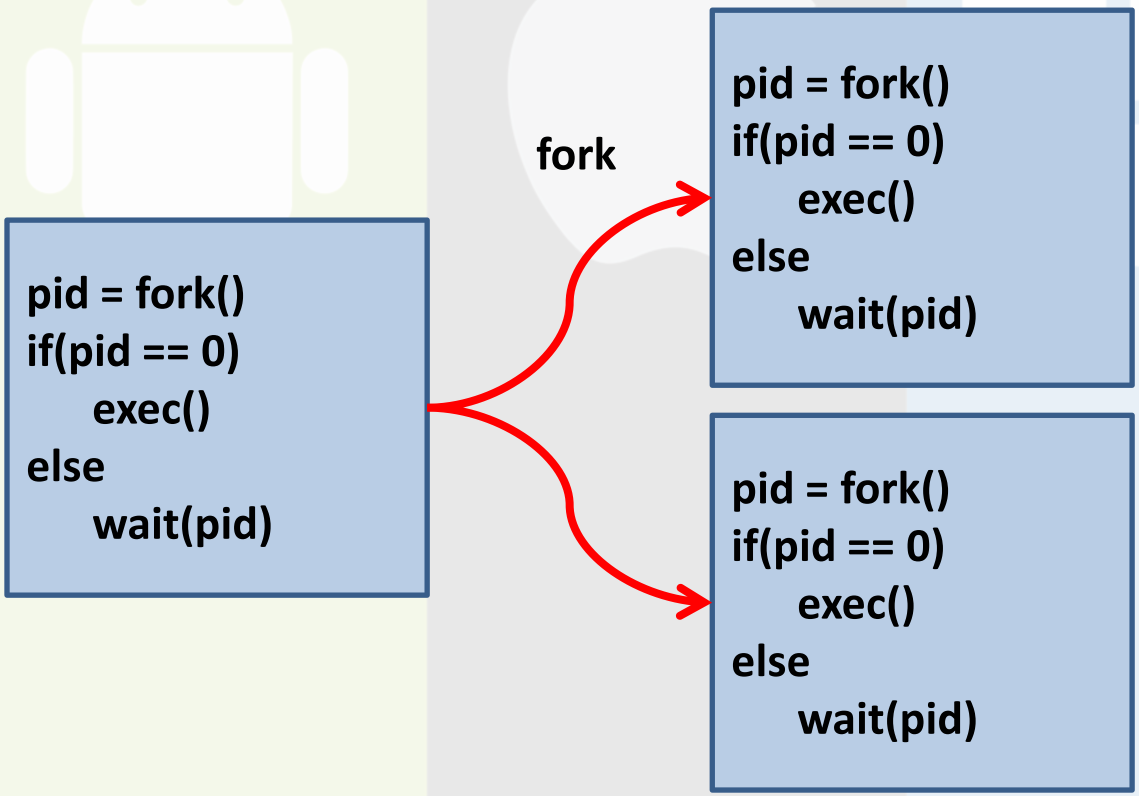
- Complex in Theory
- Simple in Practice

- UNIX splits CreateProcess into two steps called
  - fork
  - exec



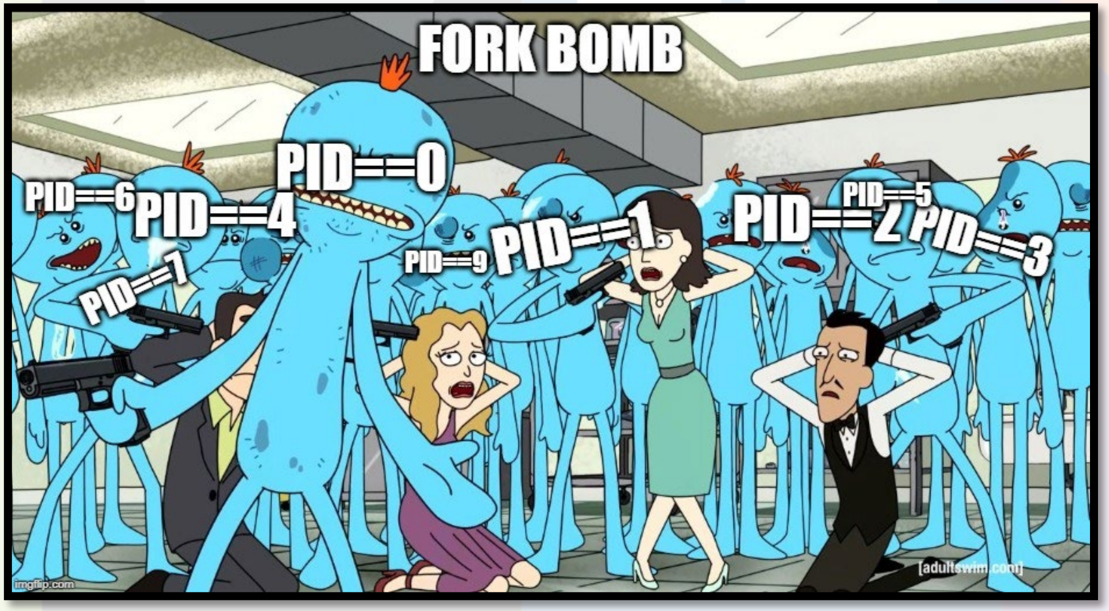
How does UNIX handle process management?

- UNIX fork
    - Creates a complete copy of the parent process
    - The only difference is the id which is used to identify it
    - Child process sets up the same privileges, priorities and I/O the parent would



What steps does UNIX fork take

- Create and initialise the PCB in the kernel
- Create a new address space
- Initialise the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent
    - if any files have been opened
- Inform the scheduler that the new process is ready to run



What steps does UNIX fork take

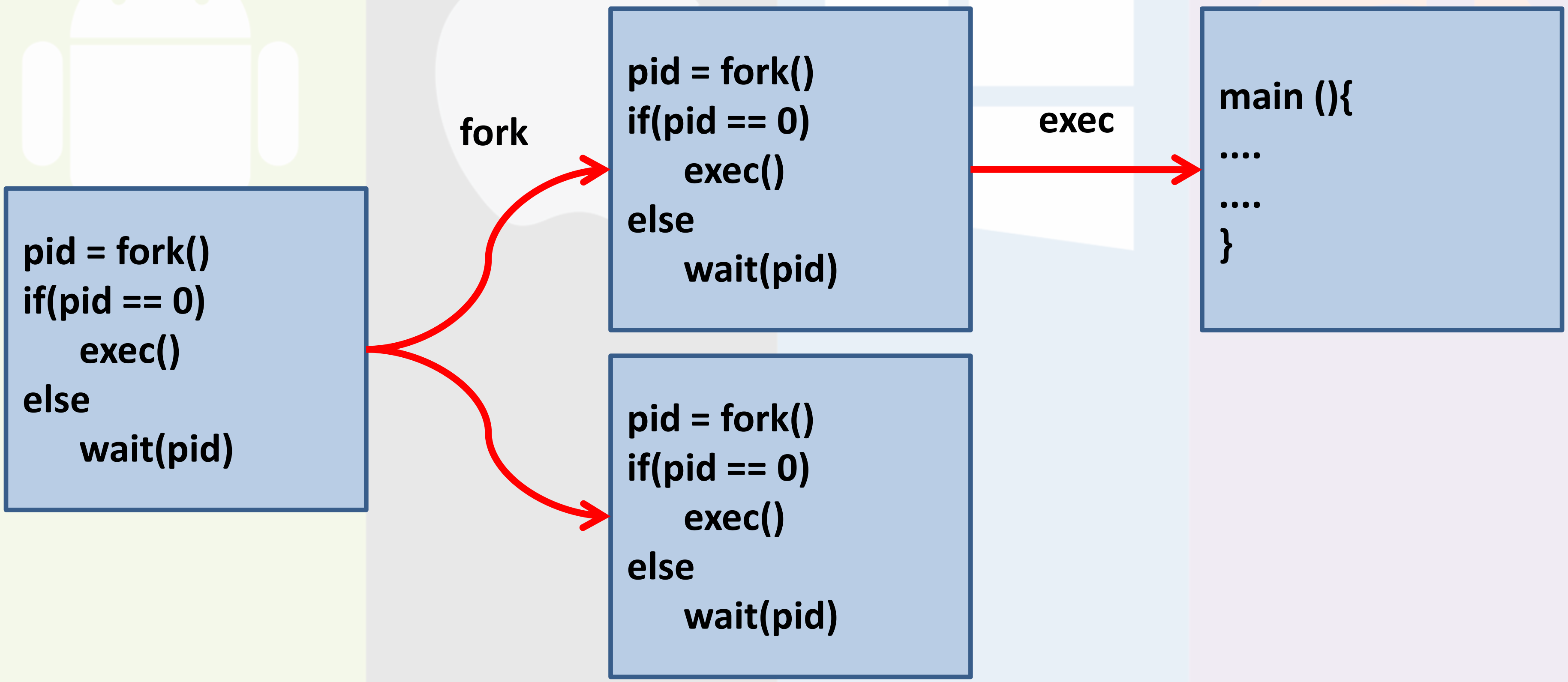
- Create and initialise the PCB in the kernel
- Create a new address space
- Initialise the address space with a copy of the entire contents of the address space of the parent
- Inherit the execution context of the parent
    - if any files have been opened
- Inform the scheduler that the new process is ready to run

- A strange aspect of Unix fork is that the system call returns twice once for the parent and once for the child

- Parent receives the process ID of the child
- Child receives 0 indicating success

How does UNIX handle process management?

- UNIX exec
  - Once the context is set the child process calles UNIX exec
  - Brings the new executable image into memory and runs it
  - exec takes in as parameters the name of the program and the arguments for it

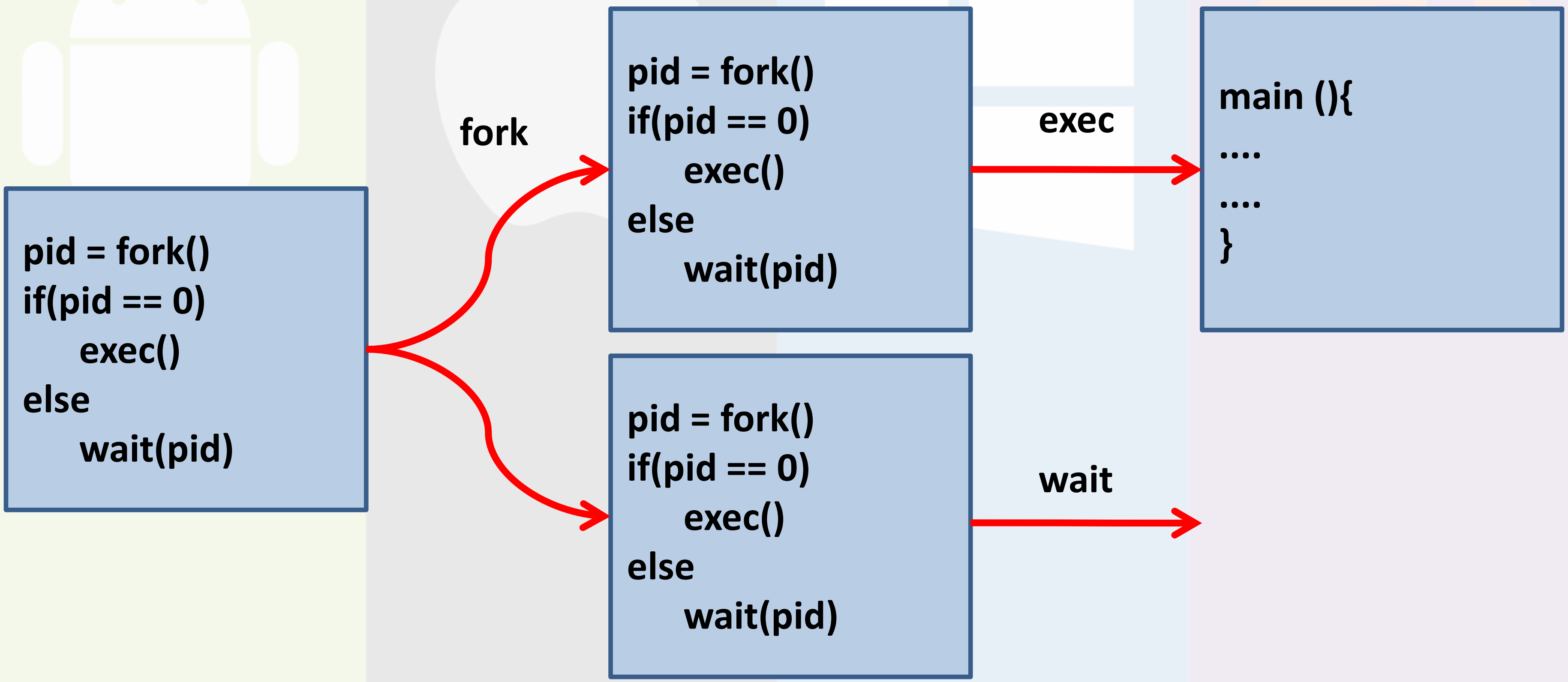


What steps does UNIX exec take

- Loads the program specified into the current address space
- Copy the arguments into memory in the address space
- Initialise the hardware context to start the execution at the start

How does UNIX handle process management?

- UNIX wait
  - Often parent processes need to pause for child processes for this we use wait
  - Pauses the parent process until the child process is finished
  - UNIX wait takes in the process id for which the parent must wait for



How does UNIX handle process management?

- UNIX signal
  - UNIX provides a facility for one process to send another an instant notification (Upcall)
  - This notification is sent using signal