Operating Systems

Limited Direct Execution + Memory Virtualisation

Nipun Batra

Direct Execution

OS

Program

Direct Execution

OS	Program
1. Create entry for process	

Direct Execution

OS	Program
1. Create entry for process 2. Allocate memory for process	

Direct Execution

OS	Program
1. Create entry for process 2. Allocate memory for process 3. Load program into memory	

Direct Execution

OS

Program

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack

Direct Execution

OS | Program

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Program

1. Run main()

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Program

1. Run main()
2. Execute return from main

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

1. Free memory

Program

1. Run main()
2. Execute return from main

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Program

1. Run main()
2. Execute return from main

OS

1. Free memory
2. Remove process from process list

Direct Execution Challenges

Direct Execution Challenges

1. How would OS stop the current process and run another

Direct Execution Challenges

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn’t make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

- Do we stop accessing I/O and network?

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

- Do we stop accessing I/O and network?

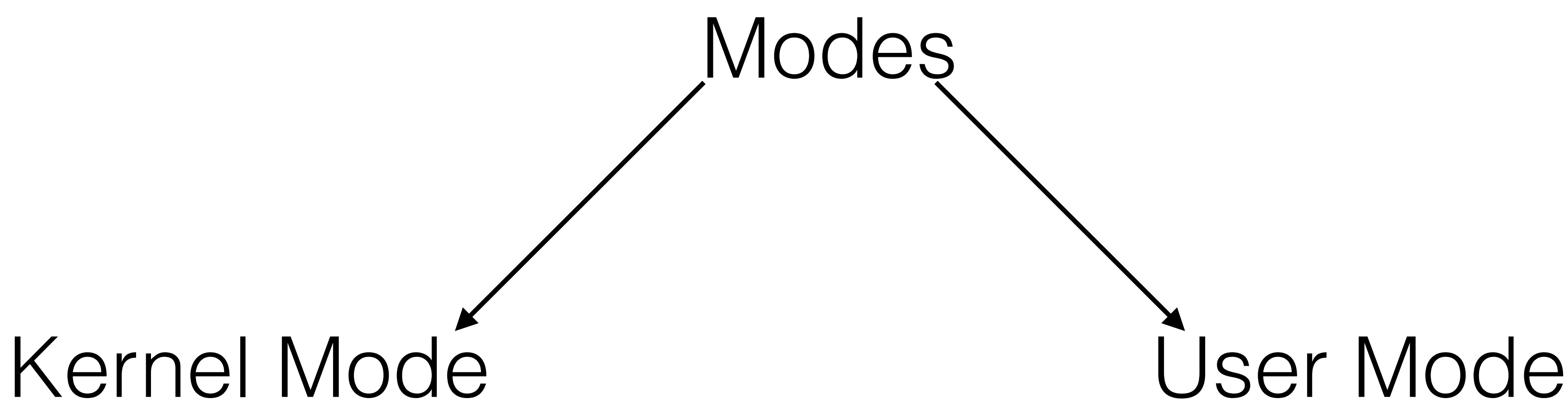
Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

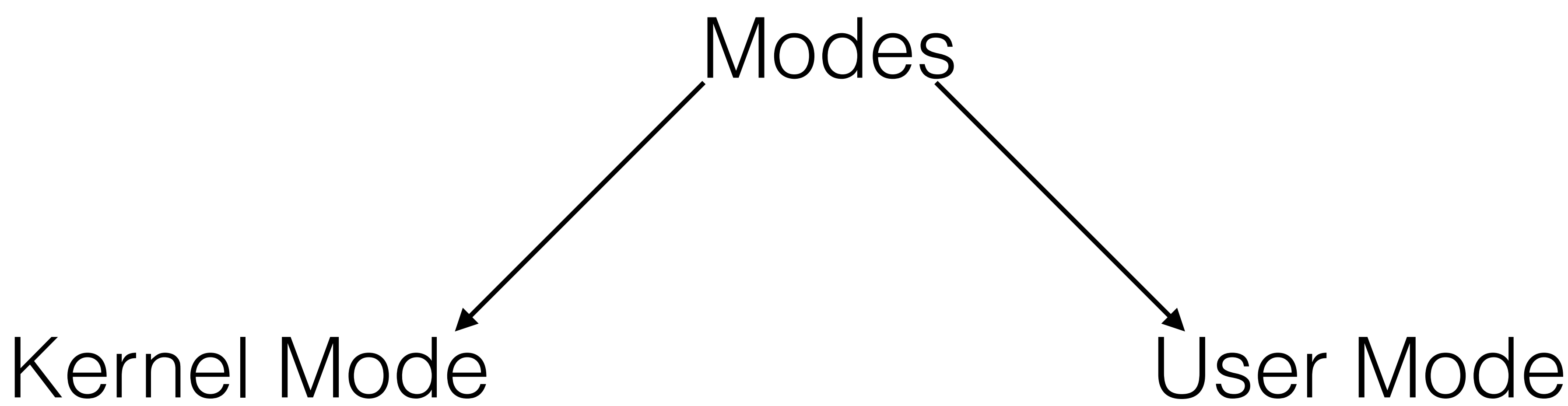
- Do we stop accessing I/O and network?

- Goal: A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.

Restricted Operations



Restricted Operations



1. Restricted mode - can not issue IO

Restricted Operations

Modes

Kernel Mode

User Mode

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

Restricted Operations

Modes

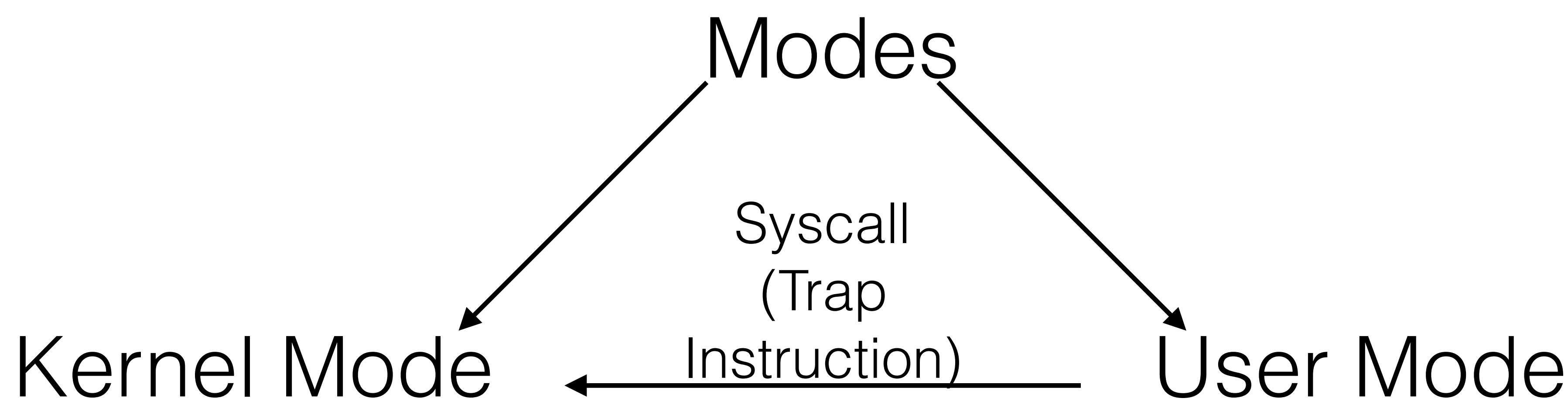
Kernel Mode

Code can issue IO.
Mode that OS runs in.

User Mode

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

Restricted Operations

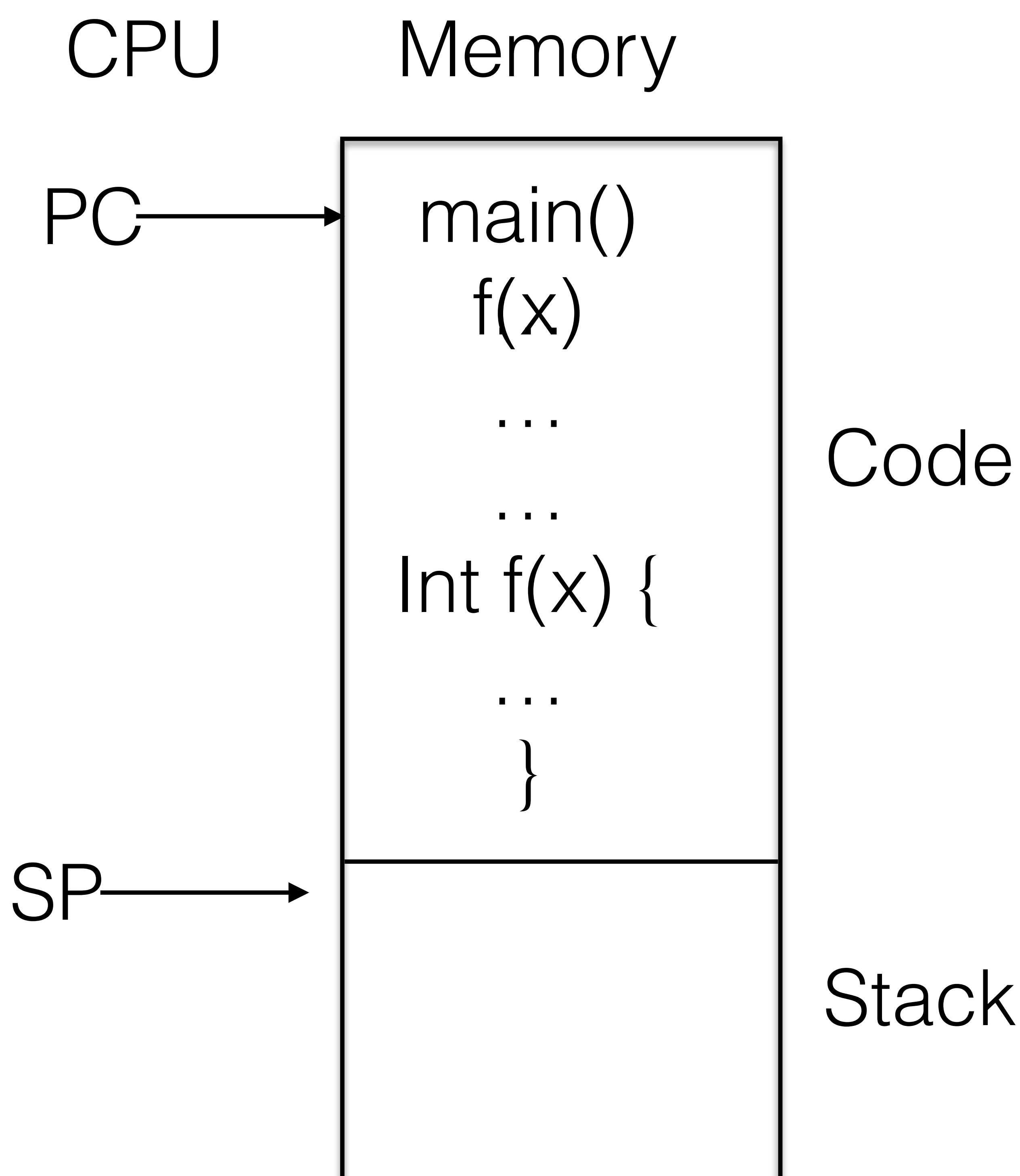


Code can issue IO.
Mode that OS runs in.

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

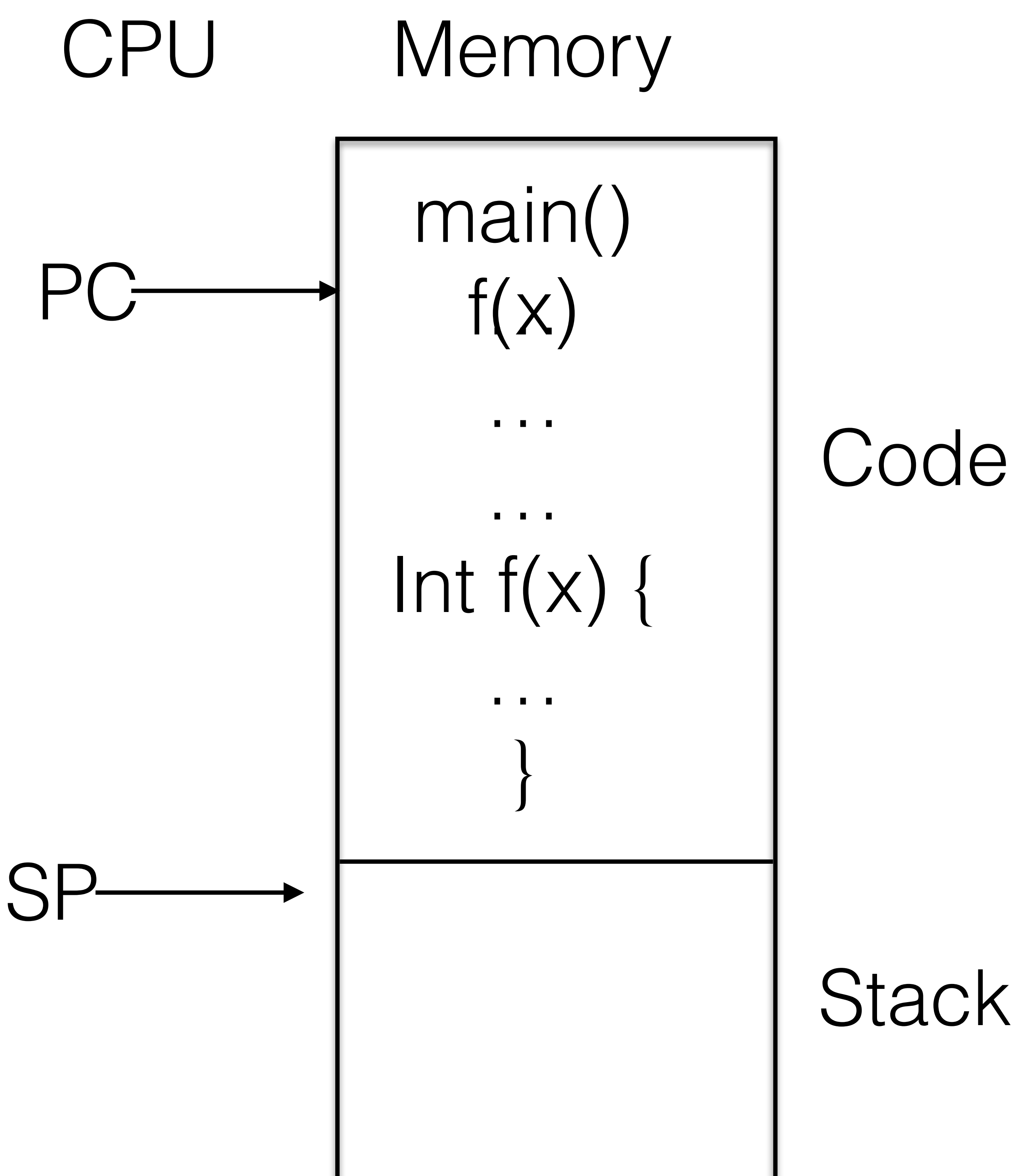
Traps v/s Function Calls

Function call



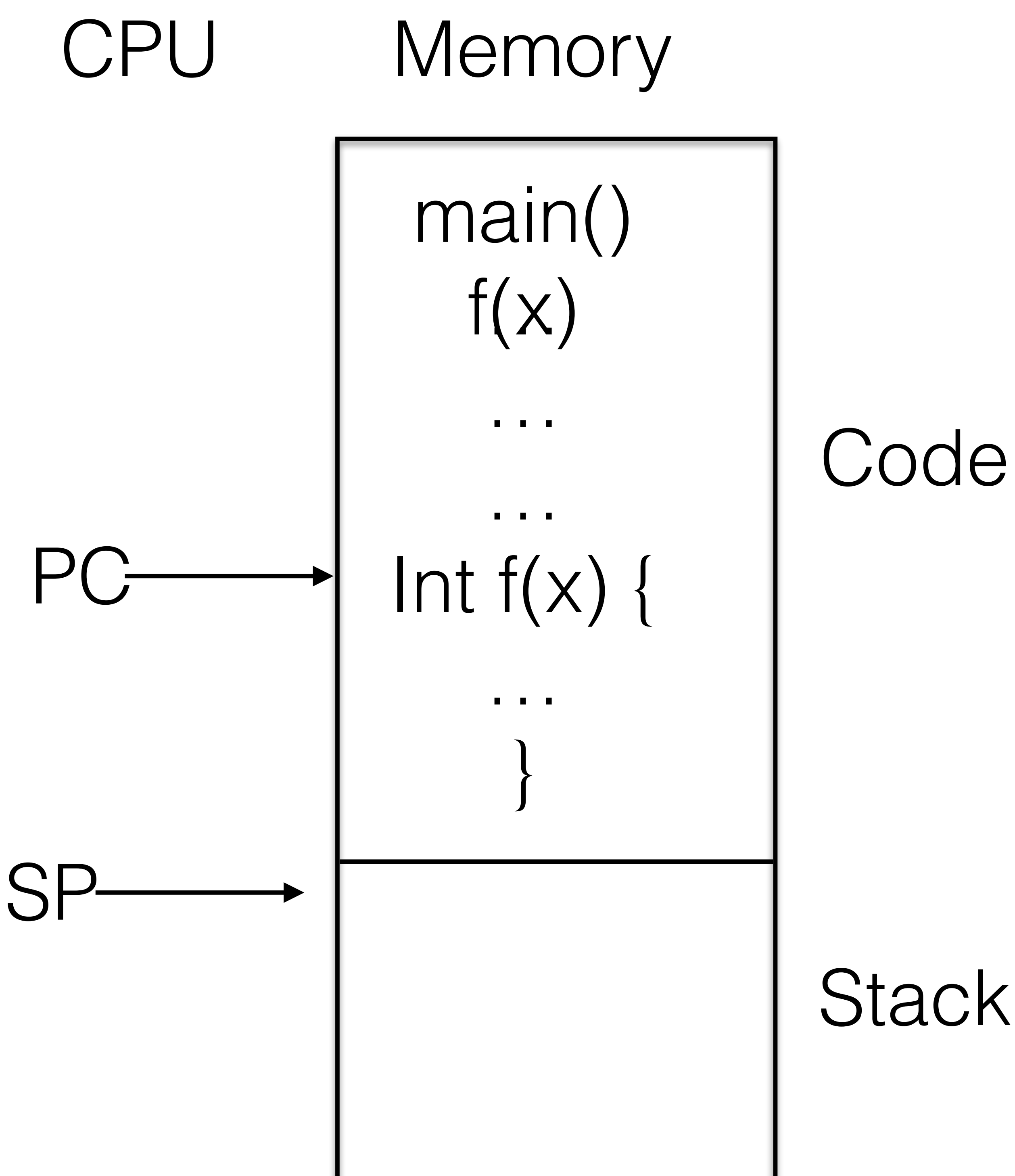
Traps v/s Function Calls

Function call



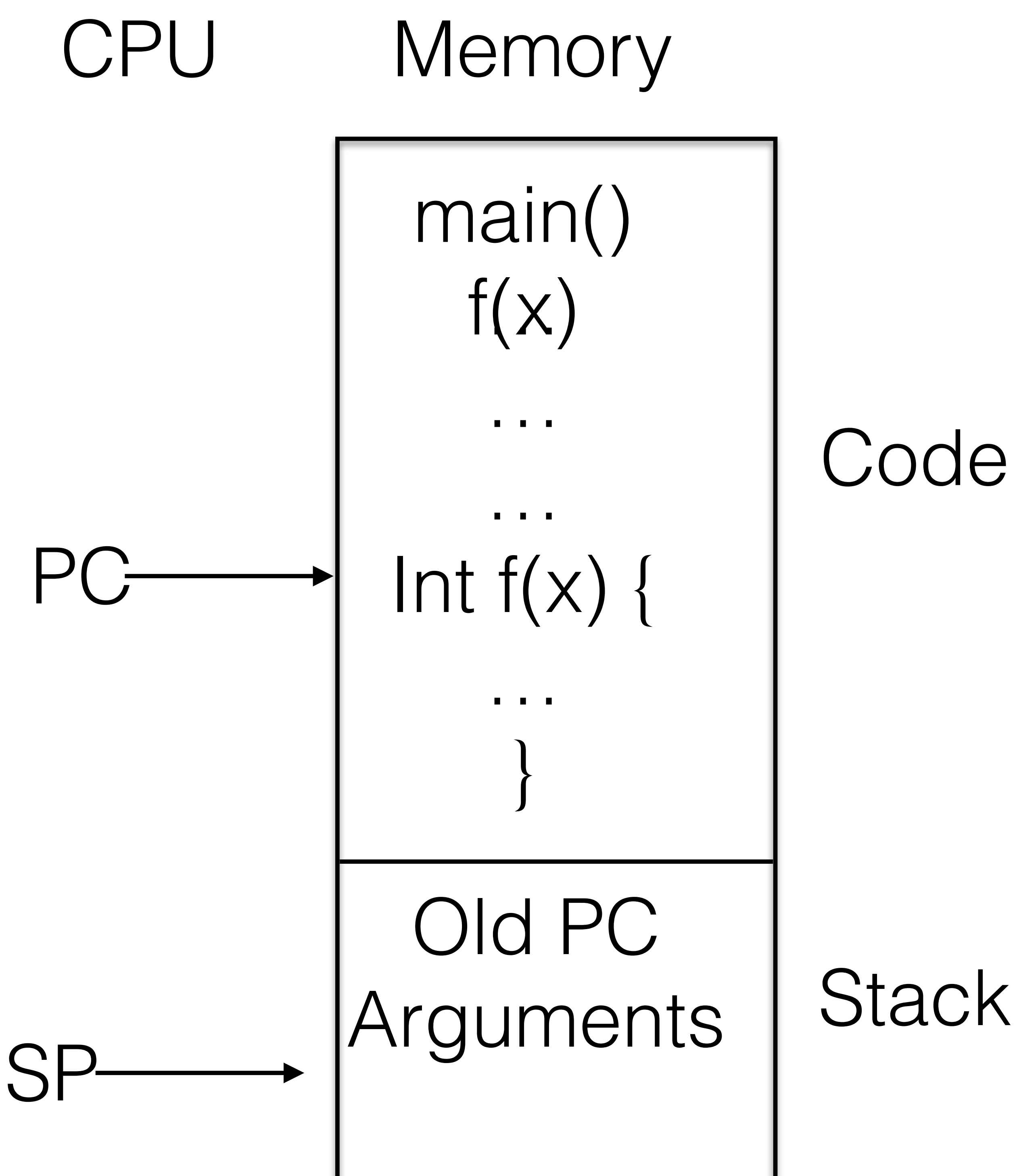
Traps v/s Function Calls

Function call



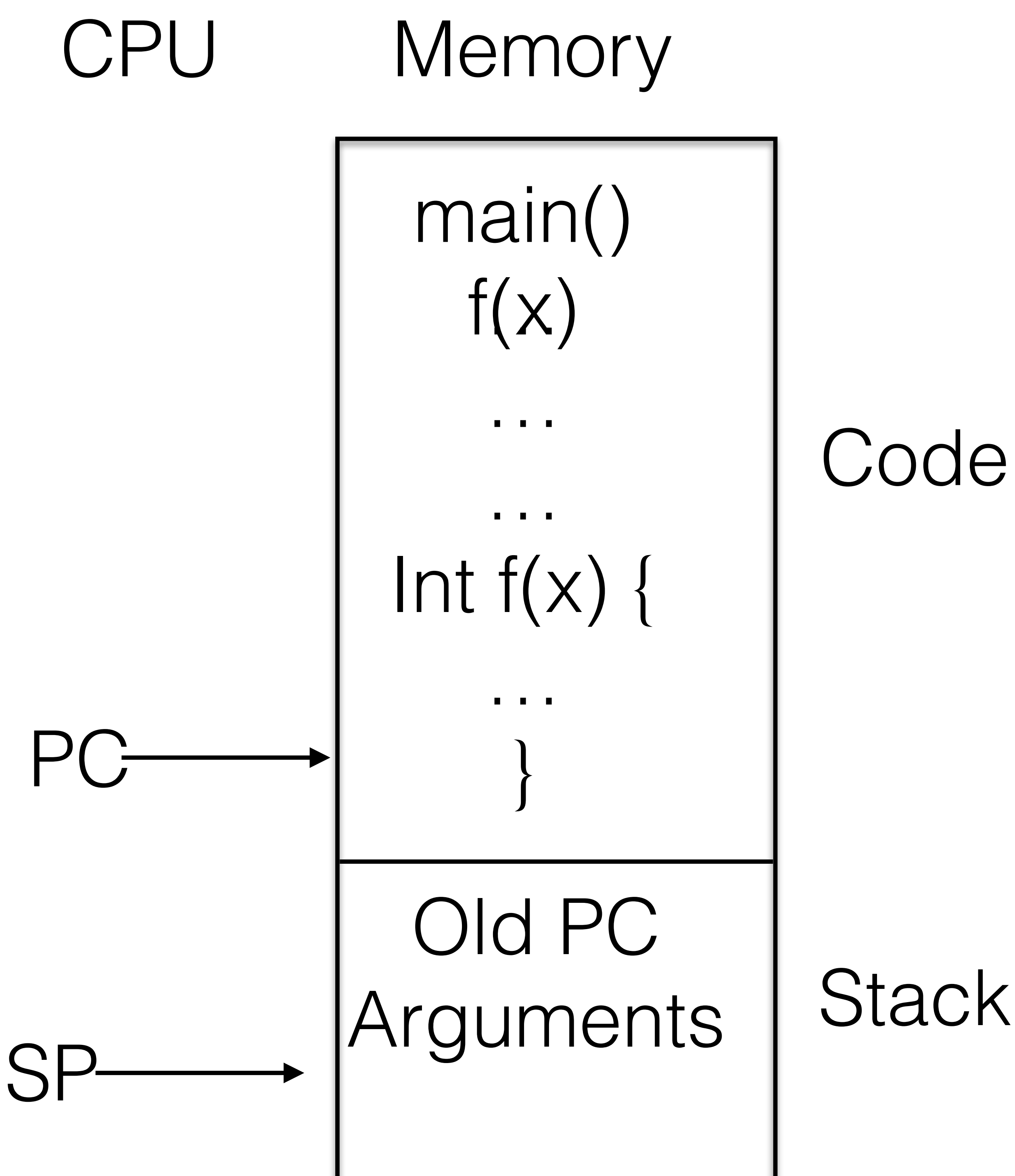
Traps v/s Function Calls

Function call

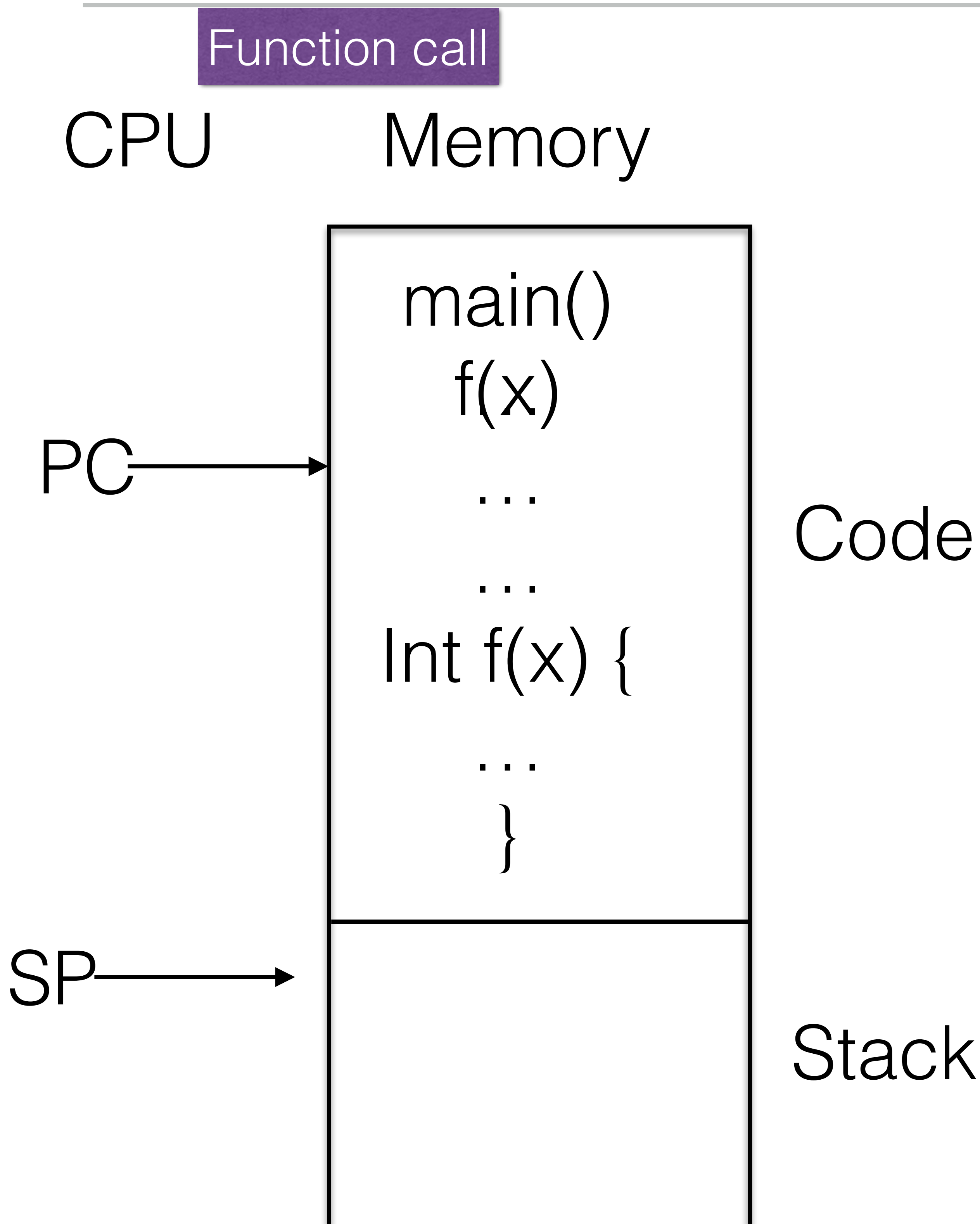


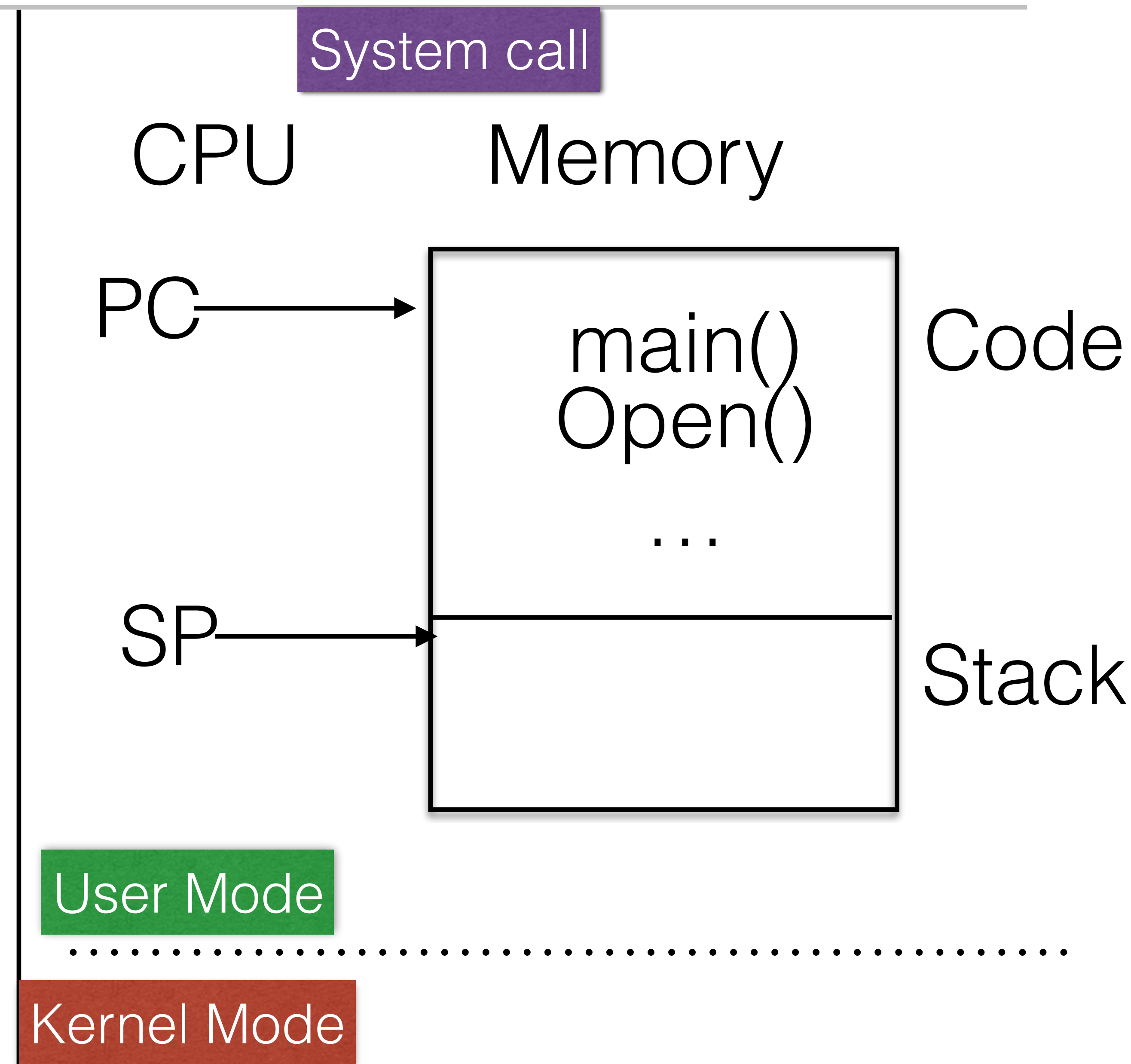
Traps v/s Function Calls

Function call

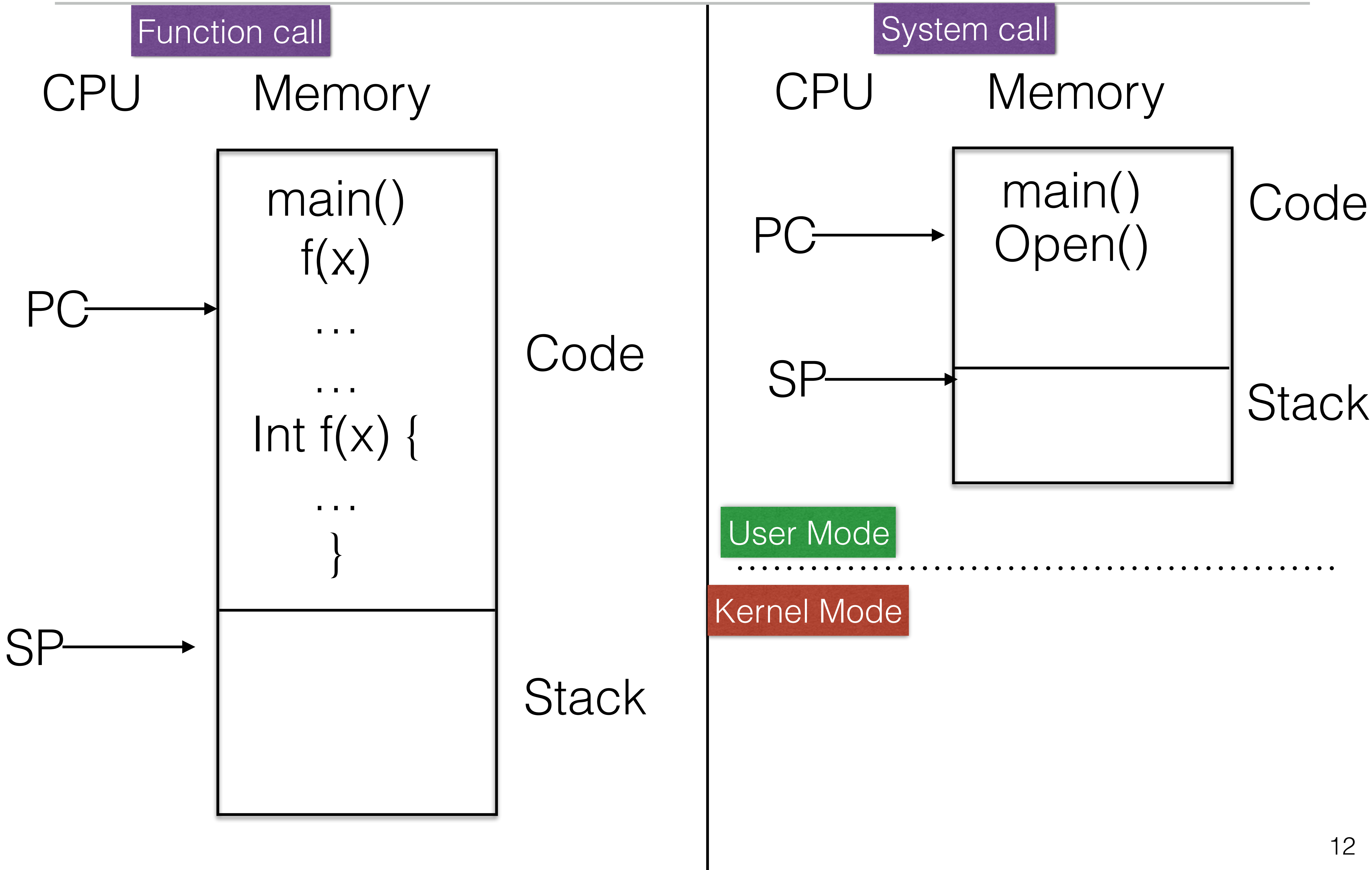


Traps (System) v/s Function Calls

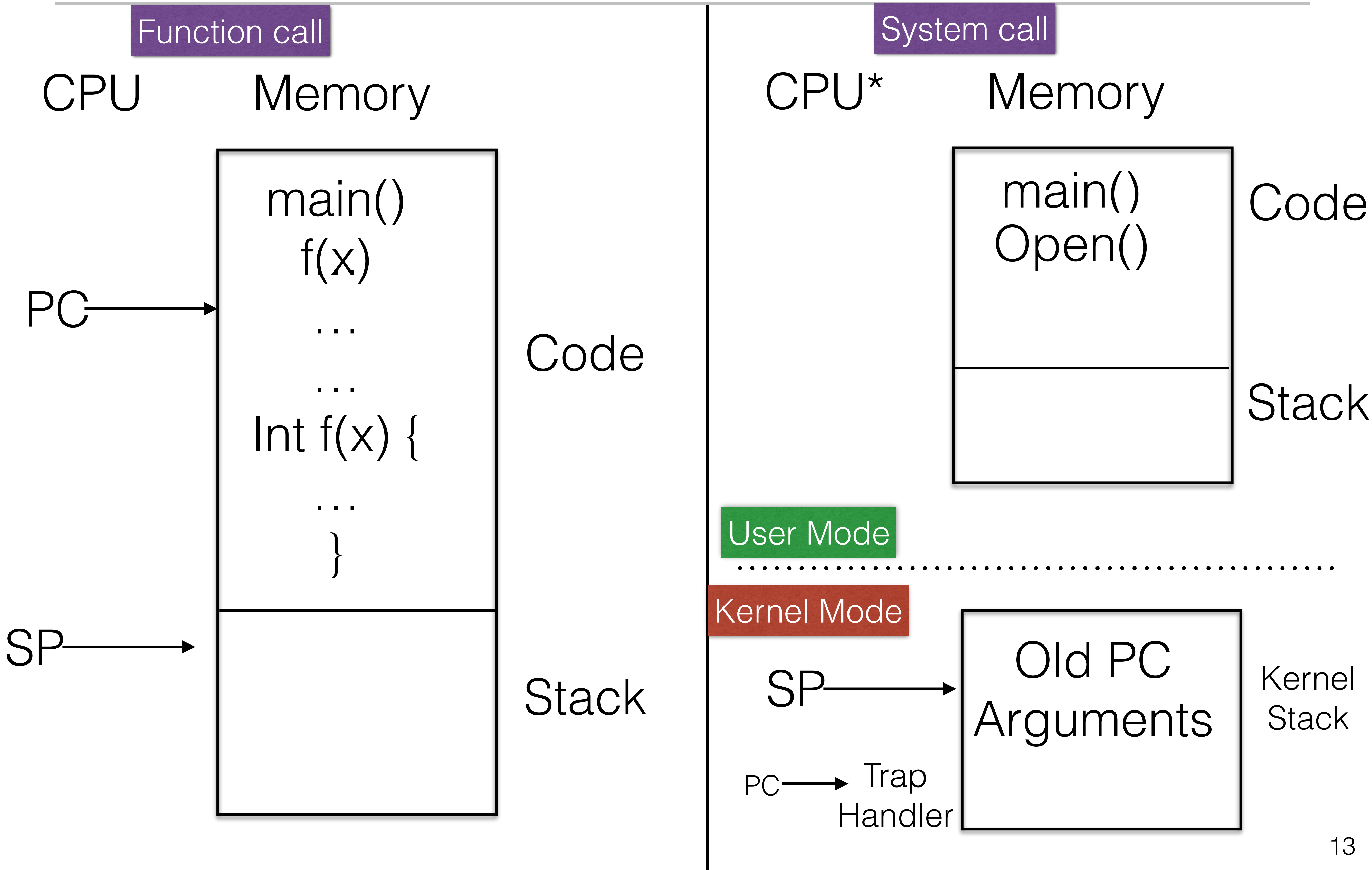




Traps (System) v/s Function Calls



Traps (System) v/s Function Calls



CPU boots in kernel mode, with full access to the system hardware. It then proceeds to load and start the operating system running.

CPU boots in kernel mode, with full access to the system hardware. It then proceeds to load and start the operating system running.

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

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	

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 call **trap** into OS
	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		

Summary

https://minnie.tuhs.org/CompArch/Lectures/week05.html

Switching Between Processes

Switching Between Processes

- Is the OS running on CPU when program is running?

Switching Between Processes

- Is the OS running on CPU when program is running?

Switching Between Processes

- Is the OS running on CPU when program is running?
- NO!

Switching Between Processes

- Is the OS running on CPU when program is running?
- NO!

Switching Between Processes

- Is the OS running on CPU when program is running?
- NO!
- How does OS get back in control?

Switching Between Processes - Cooperative

Switching Between Processes - Cooperative

- OS trusts processes

Switching Between Processes - Cooperative

- OS trusts processes

Switching Between Processes - Cooperative

- OS trusts processes
- Long running processes periodically give up CPU

Switching Between Processes - Cooperative

- OS trusts processes
- Long running processes periodically give up CPU
  - Via system call

Switching Between Processes - Cooperative

- OS trusts processes
- Long running processes periodically give up CPU
  - Via system call
    - But what if we don't need a system call?

Switching Between Processes - Cooperative

- OS trusts processes
- Long running processes periodically give up CPU
  - Via system call
    - But what if we don't need a system call?
    - Explicit system calls!

Switching Between Processes - Cooperative

- OS trusts processes
- Long running processes periodically give up CPU
  - Via system call
    - But what if we don't need a system call?
    - Explicit system calls!

Switching Between Processes - Non-Cooperative

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don’t need cooperative approach

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don't need cooperative approach
  - Raise every x milliseconds

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don’t need cooperative approach
  - Raise every x milliseconds
  - What to execute when interrupt occurs?

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don't need cooperative approach
  - Raise every x milliseconds
  - What to execute when interrupt occurs?
    - OS sets up interrupt service routine

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don't need cooperative approach
  - Raise every x milliseconds
  - What to execute when interrupt occurs?
    - OS sets up interrupt service routine
    - OS starts timer at the boot time

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
  - What if there is a bug?
    - Restart!
- Timer interrupt
  - Don’t need cooperative approach
  - Raise every x milliseconds
  - What to execute when interrupt occurs?
    - OS sets up interrupt service routine
    - OS starts timer at the boot time

Switching Between Processes - Non-Cooperative

- Process refuses to make system calls?
    - What if there is a bug?
        - Restart!
- Timer interrupt
    - Don't need cooperative approach
    - Raise every x milliseconds
    - What to execute when interrupt occurs?
        - OS sets up interrupt service routine
        - OS starts timer at the boot time

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

OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses 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	
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)**		

System Call handling (save regs(A) to k-stack(A))

User mode registers (save regs(A) to proc-struct(A), restore regs(B) from proc-struct(B))

System Call handling (switch to k-stack(B))

OS @ boot (kernel mode)	**Hardware**
initialize trap table	

	remember addresses 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	

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 ...

Simultaneous Interrupts?

Simultaneous Interrupts?

- What happens if two interrupts (say timer and syscall) occur together?

Simultaneous Interrupts?

- What happens if two interrupts (say timer and syscall) occur together?
- Hard to handle!

Simultaneous Interrupts?

- What happens if two interrupts (say timer and syscall) occur together?
- Hard to handle!
- Simple way of handling : Disable interrupts while handling interrupts

Simultaneous Interrupts?

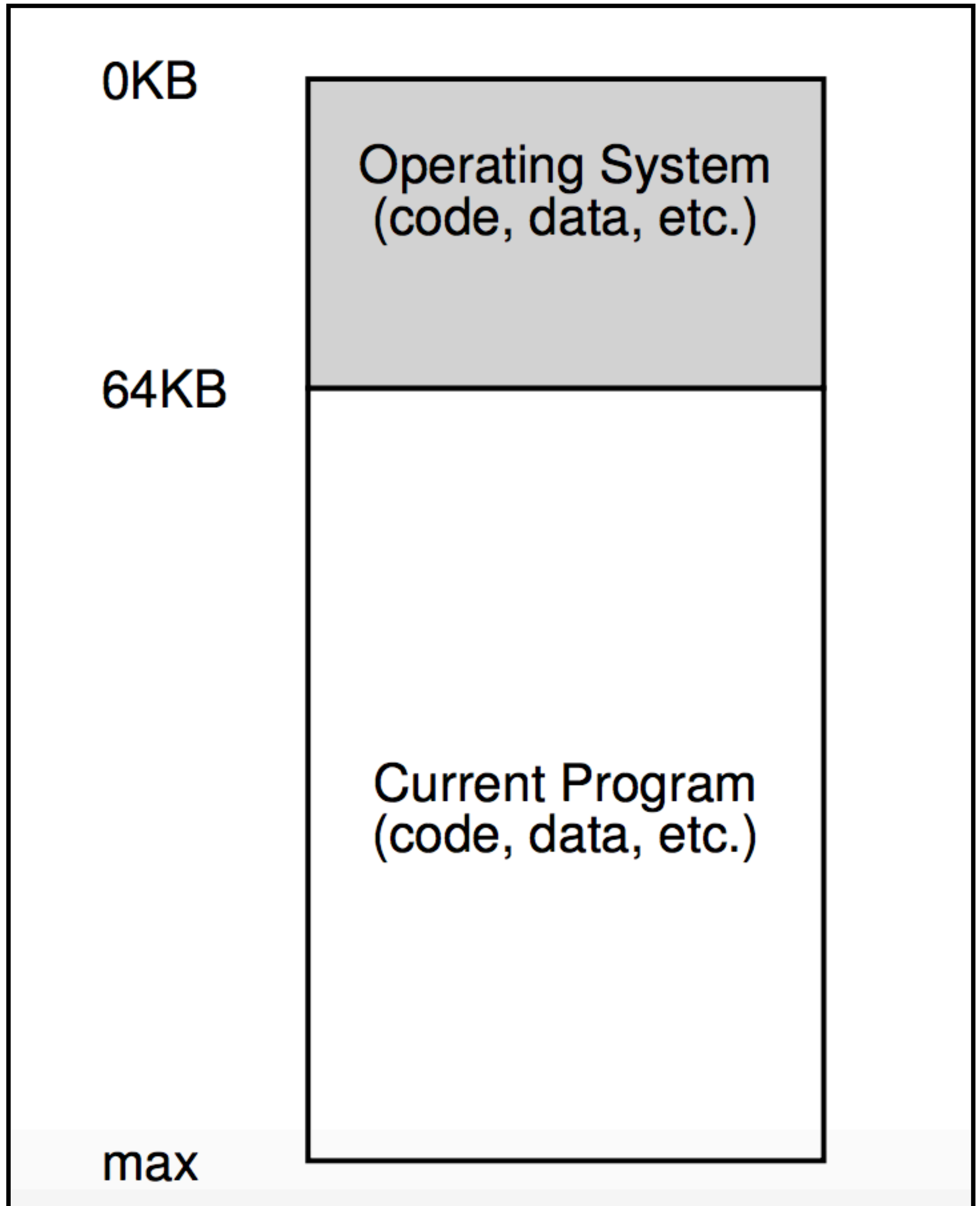
- What happens if two interrupts (say timer and syscall) occur together?
- Hard to handle!
- Simple way of handling : Disable interrupts while handling interrupts
    - How long to disable? -> Lost interrupts?

Simultaneous Interrupts?

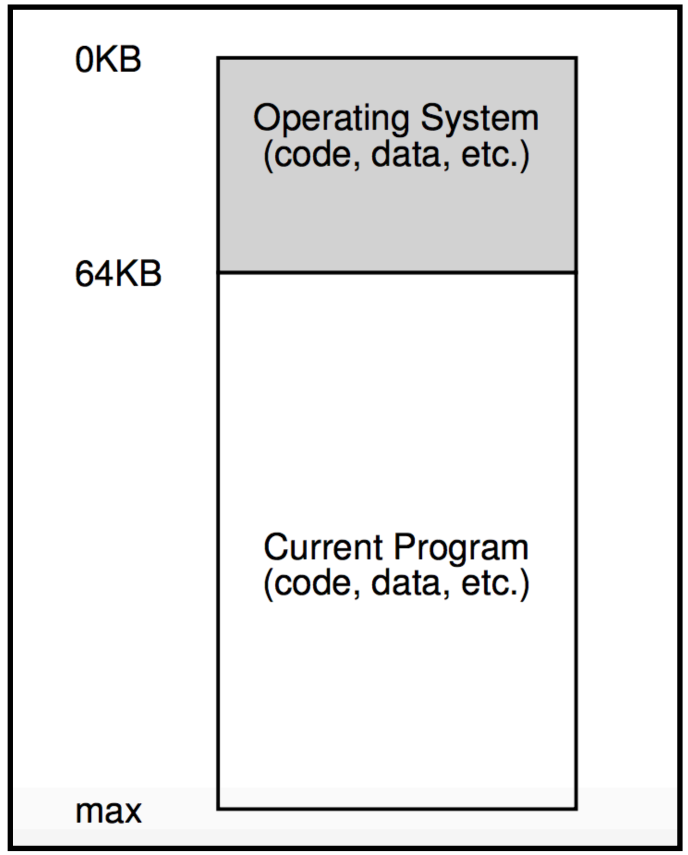
- What happens if two interrupts (say timer and syscall) occur together?
- Hard to handle!
- Simple way of handling : Disable interrupts while handling interrupts
    - How long to disable? -> Lost interrupts?
- More on it when we study concurrency!

Memory Virtualisation

Early days
Single program

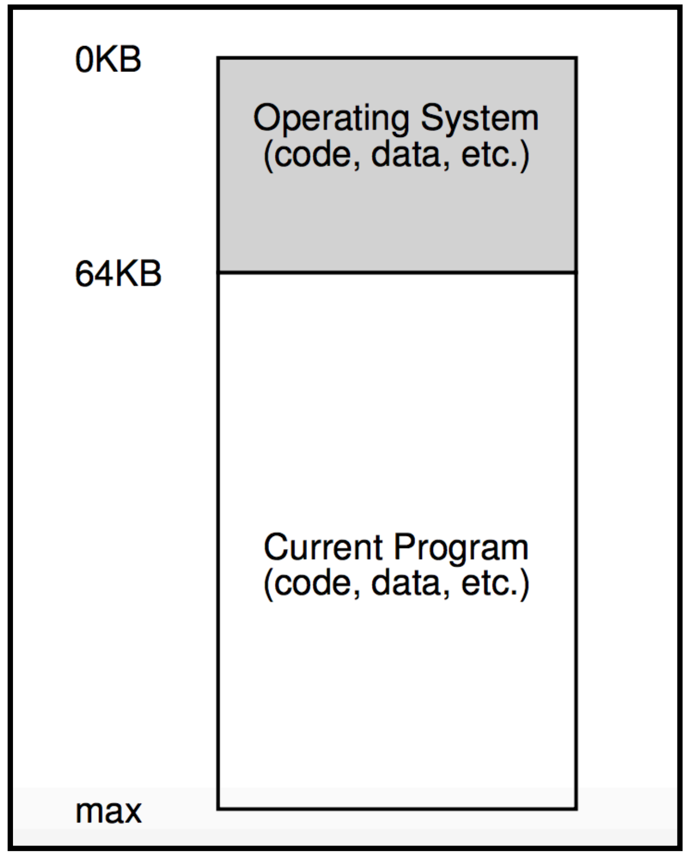


Memory Virtualisation



Early days
Multiprogamming

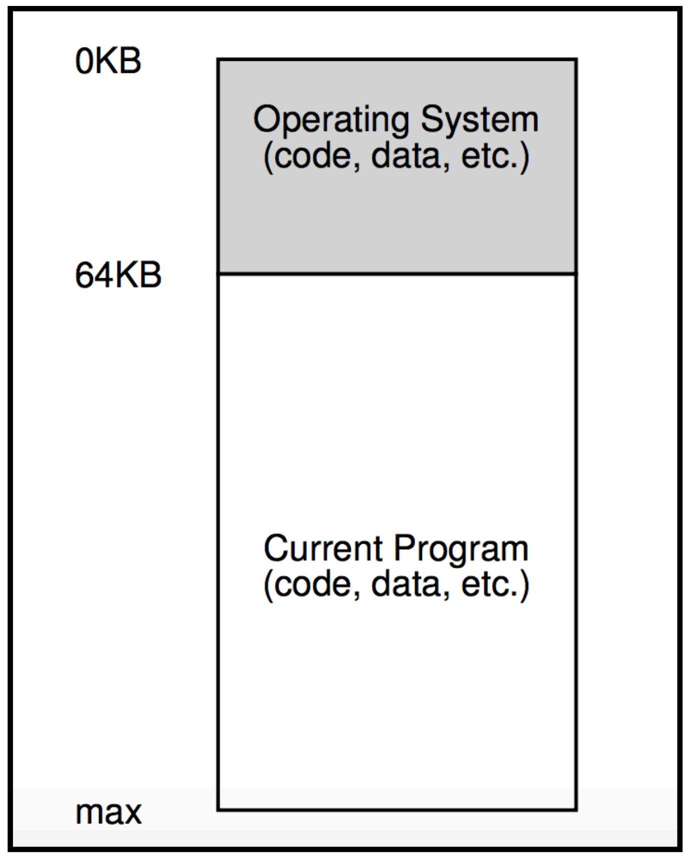
Memory Virtualisation



Early days
Multiprogamming

- Single program takes total memory

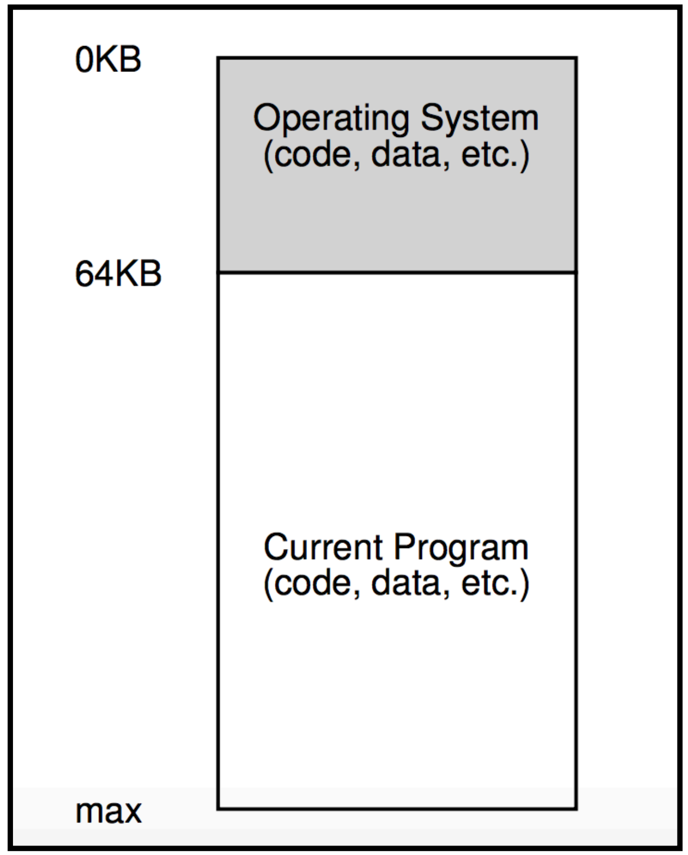
Memory Virtualisation



Early days
Multiprogamming

- Single program takes total memory
- Load another process?

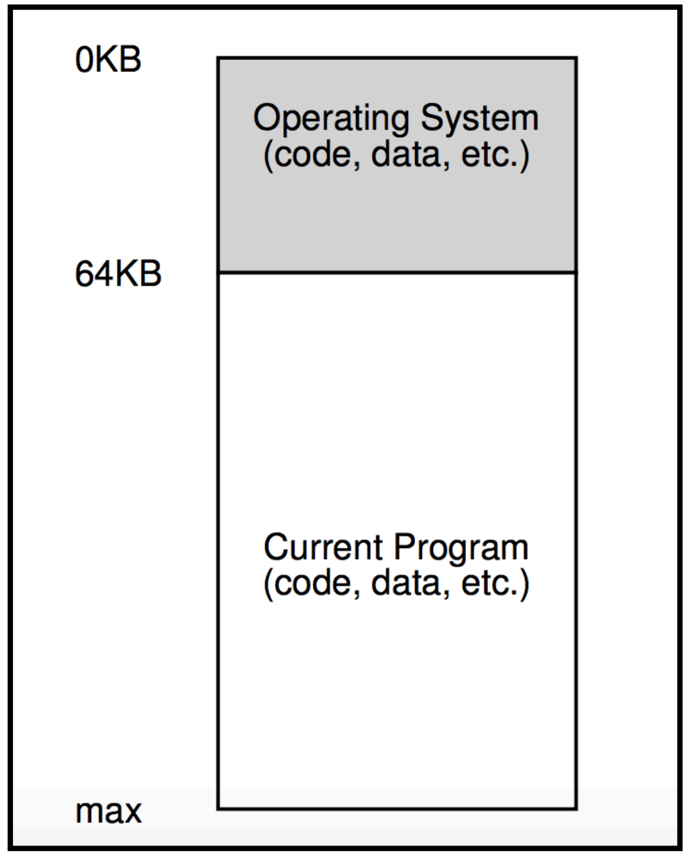
Memory Virtualisation



Early days
Multiprogamming

- Single program takes total memory
- Load another process?
  - Write to disk, read other program from disk

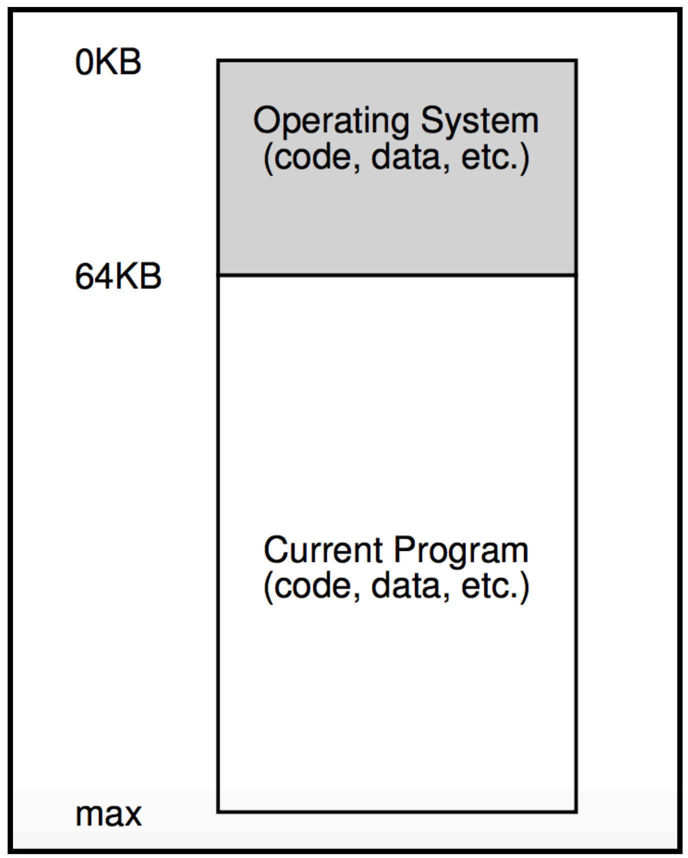
Memory Virtualisation



Early days
Multiprogamming

- Single program takes total memory
- Load another process?
  - Write to disk, read other program from disk
  - Slow?

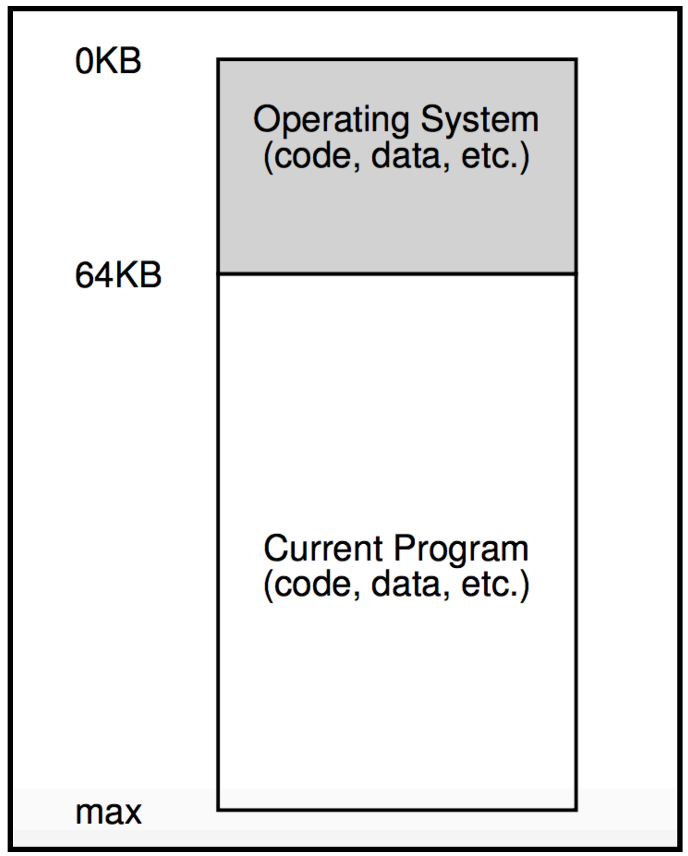
Memory Virtualisation



Early days
Multiprogamming

- Single program takes total memory
- Load another process?
  - Write to disk, read other program from disk
  - Slow?
    - HDD v/s RAM

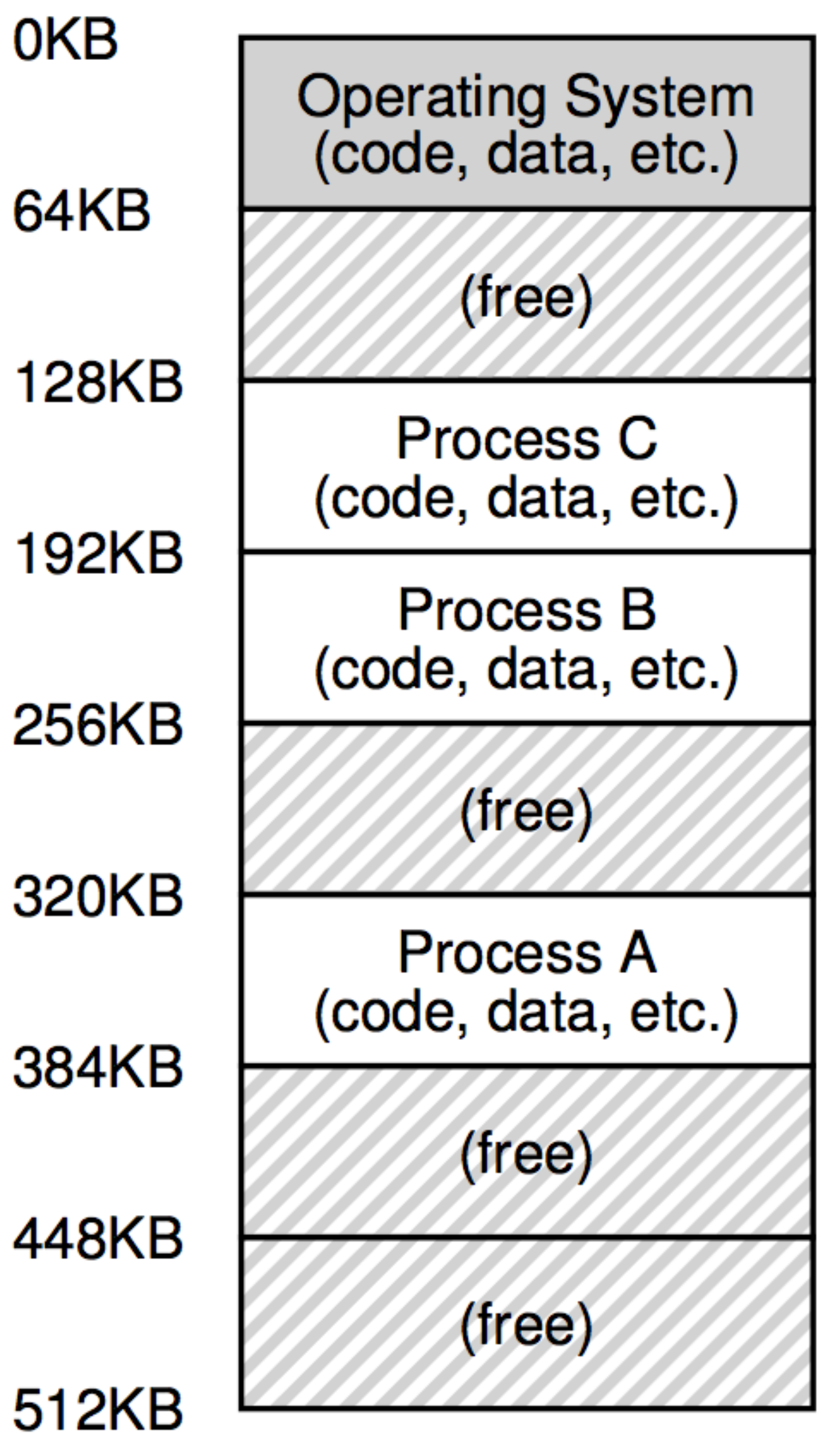
Memory Virtualisation



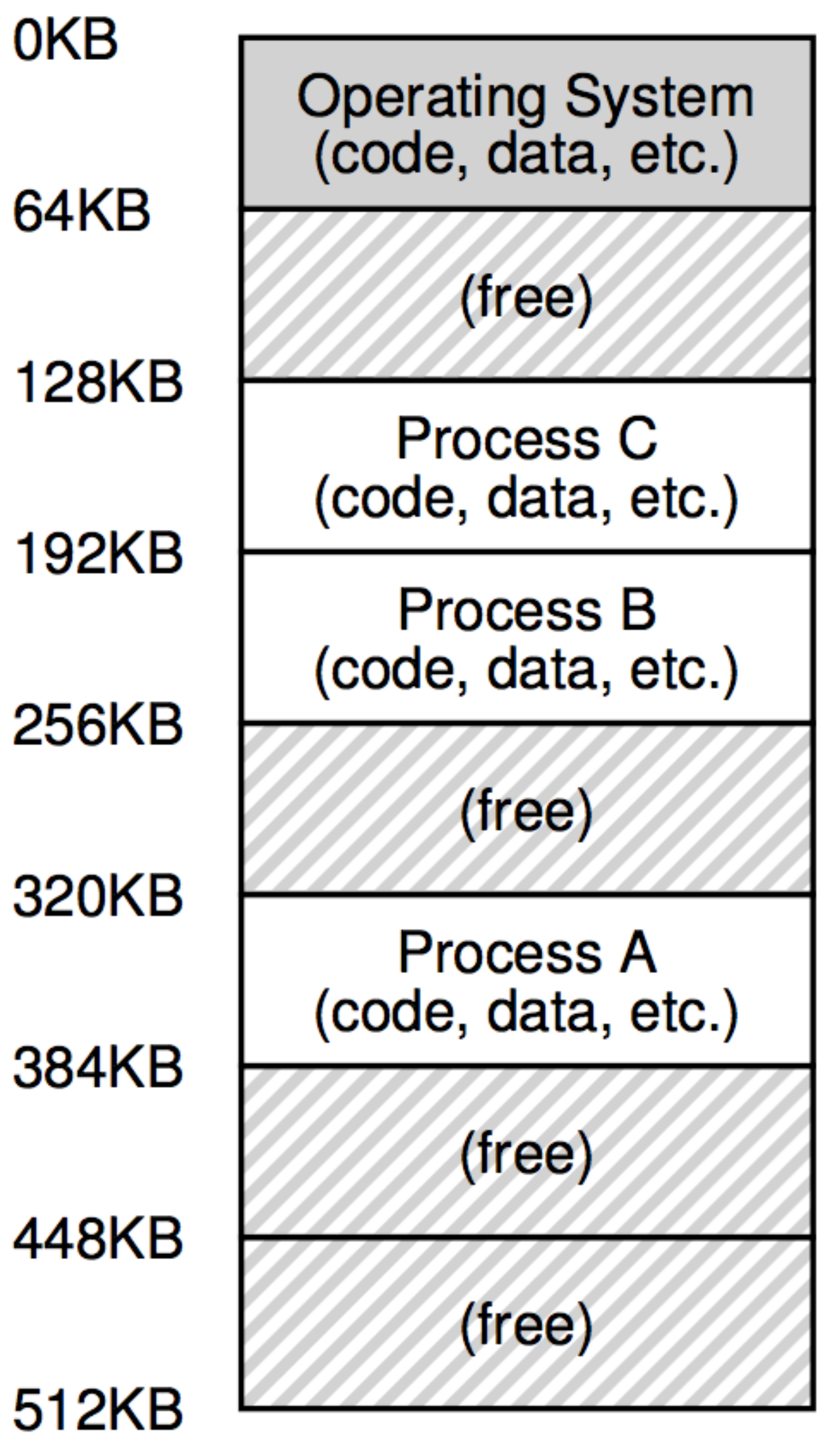
Early days
Multiprogamming

- Single program takes total memory
- Load another process?
  - Write to disk, read other program from disk
  - Slow?
    - HDD v/s RAM
    - SSD v/s RAM

Shared Memory

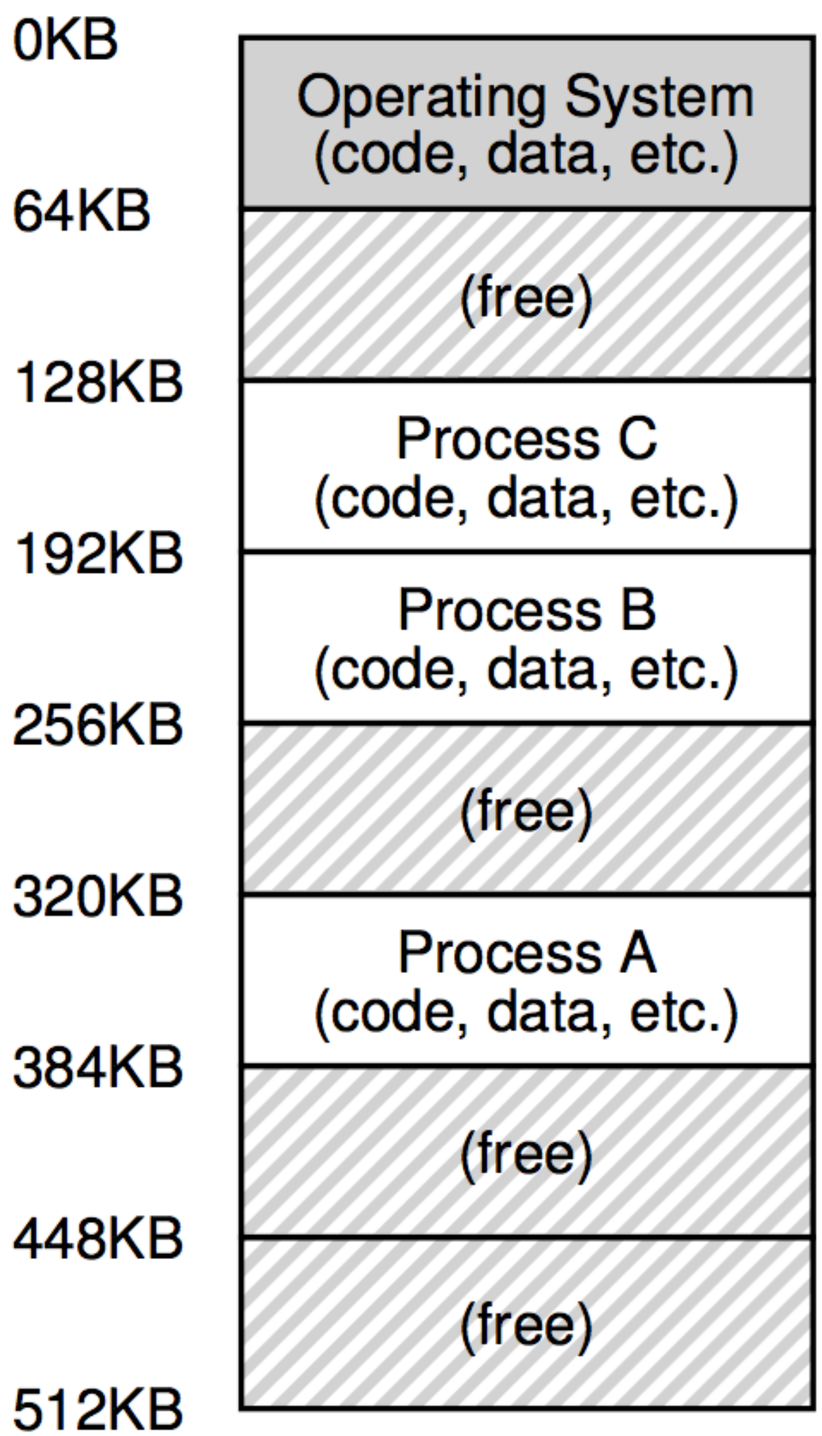


Shared Memory



Risk

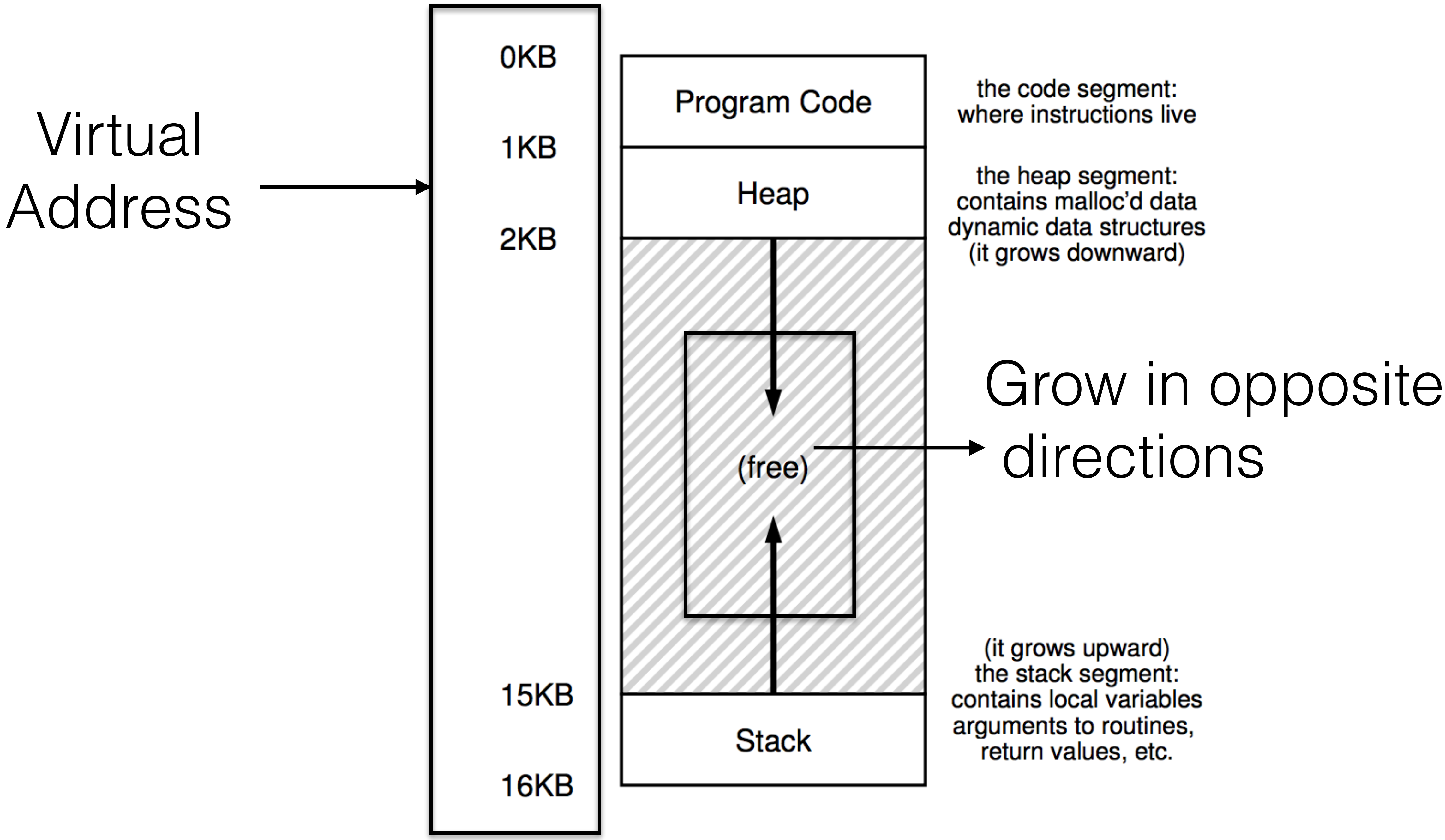
Shared Memory



Risk

- Programs accessing others' memory

Address Space



Goals of OS for Memory Virtualisation

1. Transparency
   1. Virtual memory is invisible to user program
   2. Program thinks it has own private large memory
2. Efficiency
   1. Not taking very long
   2. Not taking too much space
3. Protection/Isolation
   1. Protect processes from each other