Operating System

BCSE303L

Module 2

Dr. Naveenkumar Jayakumar
Associate Professor
Department of Computational Intelligence
PRP 217-4
naveenkumar.jk@vit.ac.in

System Calls

- ❑ An operating system (OS) acts as the fundamental intermediary between computer hardware and the software that runs on it.

- ❑ At the heart of this interaction lies a crucial mechanism known as a **system call**.

- ❑ A system call is a **programmatic way** for a running program to **request a service** from the operating system's **kernel**

System Calls

- An operating system provides a **layer of abstraction**, **managing the computer's resources** and offering a consistent **interface to applications**.

- For security and stability, modern operating systems employ a dual-mode operation: **user mode and kernel mode**.

System Calls

User Mode

- This is the non-privileged mode where most applications run.
- In this mode, a program has restricted access to system resources and cannot directly interact with hardware.

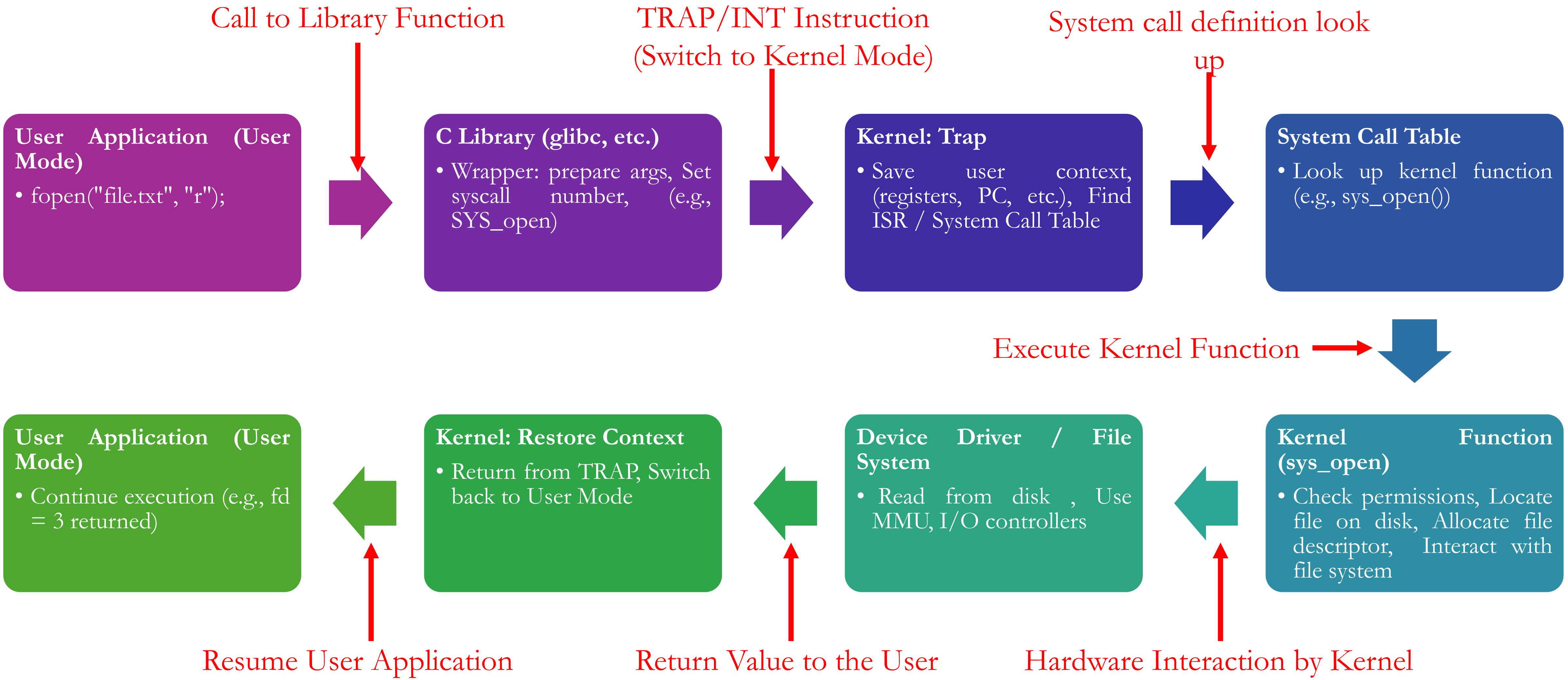
Kernel Mode

- This is the privileged mode where the operating system kernel executes.
- In this mode, the code has complete access to all hardware and system resources.

System Calls

- System calls provide a controlled and secure bridge between these two modes.
- When an application needs to perform a privileged action, such as reading from a file or opening a network connection, it must request the operating system's kernel to perform the task on its behalf.
- This request is a **system call**.

System Calls - Working



Call to Library Function

TRAP/INT Instruction (Switch to Kernel Mode)

System call definition look up

User Application (User Mode)
- fopen("file.txt", "r");

C Library (glibc, etc.)
- Wrapper: prepare args, Set syscall number, (e.g., SYS_open)

Kernel: Trap
- Save user context, (registers, PC, etc.), Find ISR / System Call Table

System Call Table
- Look up kernel function (e.g., sys_open())

Execute Kernel Function

Kernel Function (sys_open)
- Check permissions, Locate file on disk, Allocate file descriptor, Interact with file system

Device Driver / File System
- Read from disk , Use MMU, I/O controllers

Kernel: Restore Context
- Return from TRAP, Switch back to User Mode

User Application (User Mode)
- Continue execution (e.g., fd = 3 returned)

Hardware Interaction by Kernel

Return Value to the User

Resume User Application

System Calls - Working

- ❑ The execution of a system call involves a well-defined sequence of steps that facilitates the transition from user mode to kernel mode and back.

- ❑ **Application Initiates the Call**: A user program, written in a high-level language like C++ or Python, makes a call to a library function (e.g., fopen() to open a file). This library function is part of the Application Programming Interface (API) provided by the system.

System Calls - Working

- ❑The Library Function Invokes the System Call:
  - ❑The library function is a wrapper that contains the necessary code to prepare for the actual system call.
  - ❑This preparation involves placing the system call number (a unique integer identifying the requested service) and its arguments (e.g., the filename and the mode of access) into specific registers or a stack in memory.

System Calls - Working

- **The TRAP Instruction:** The library function then executes a special instruction often called a TRAP or INT (interrupt).
- This instruction causes a software interrupt, which is a signal to the processor to switch from user mode to kernel mode.

System Calls - Working

- ❑ **The Kernel Takes Over:** Upon receiving the trap, the processor saves the current state of the user program (including the program counter and registers) and jumps to a specific location in the kernel's memory.
- ❑ This location is the starting address of the Interrupt Service Routine (ISR) or System Call Handler.

System Calls - Working

- **Executing the System Call**: The system call handler uses the system call number to look up the corresponding kernel function in a system call table.
- This table maps each system call number to the address of the kernel code that implements that service.

System Calls - Working

- **Performing the Operation**: The kernel then executes the requested operation.
- For instance, if the system call was for opening a file, the kernel would check file permissions, locate the file on the storage device, and create an entry in the system-wide open file table.

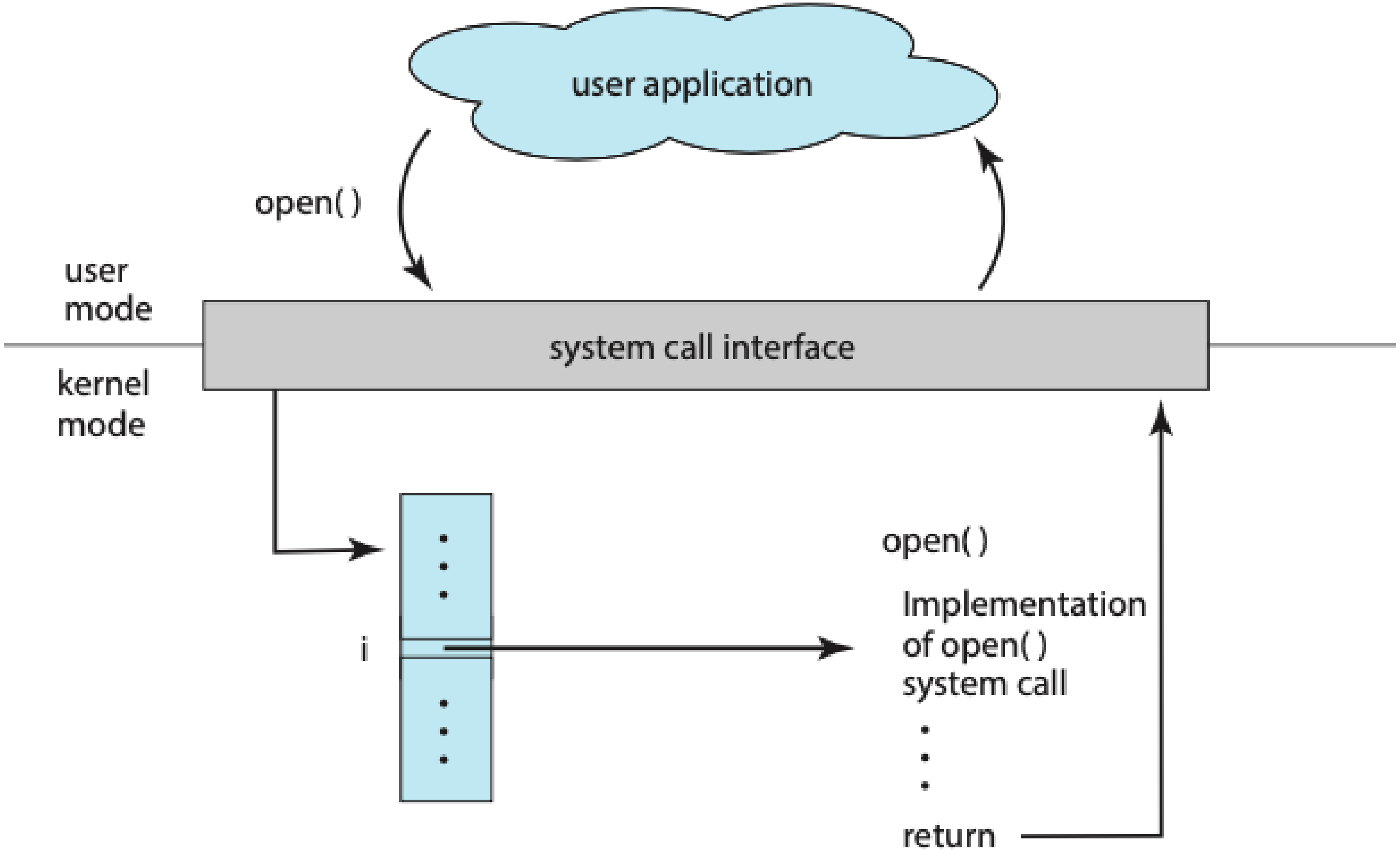
System Calls - Working

- ❑ **Returning to User Mode**: Once the kernel has completed the operation, it places the return value (e.g., a file descriptor or an error code) in a designated register.
- ❑ It then restores the saved state of the user program and executes a special return-from-interrupt instruction.

System Calls - Working

- ❑ **Resuming Application Execution:** This instruction switches the processor back from kernel mode to user mode, and the execution of the user program resumes from where it left off, now with the result of the system call available.

System Calls - Working



System Calls Interface / API

- ❑A set of functions, system calls and protocols that allows applications to interact with the OS and other software components.
- ❑It defines the way that applications request services from the OS
- ❑The API provides a level of abstraction between the application and the underlying OS, simplifying development and ensuring compatibility and security.

System Calls Interface / API

- A number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers.
- The system-call interface then invokes the intended system call in the operating-system kernel and returns the status of the system call.
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
- Managed by run-time support library (set of functions built into libraries included with compiler)

System Calls - Importance

Hardware Abstraction

- System calls provide a standardized way for programs to interact with hardware, abstracting the complexities and ensuring hardware-independent operation.

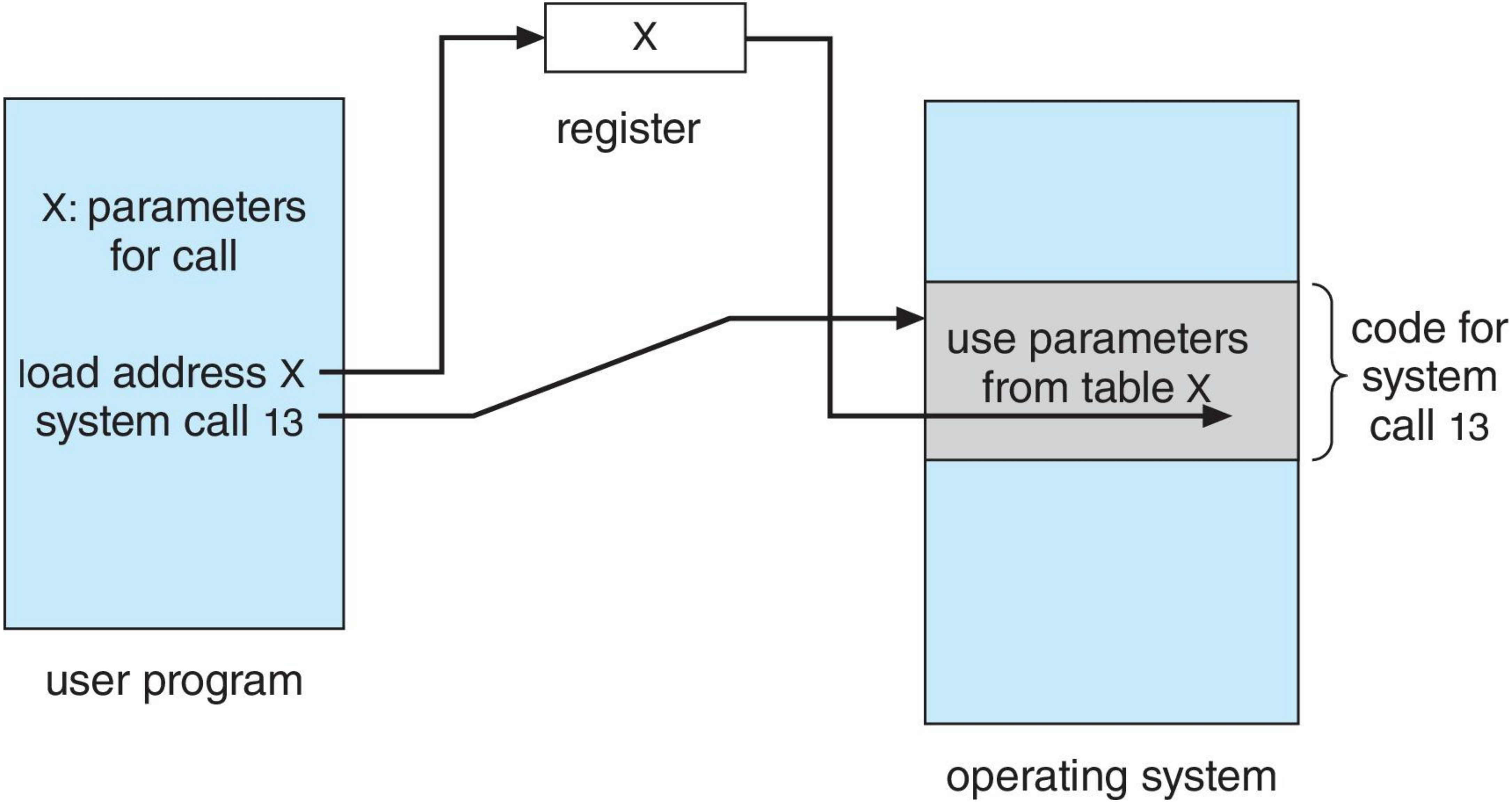
Resource Management

- They enable efficient management and sharing of hardware resources like CPU, memory, and I/O devices.

Security and Protection

- System calls allow the OS to enforce security policies, ensuring that programs operate within their allocated permissions and preventing unauthorized access to critical resources.

Parameter Passing between Programs & Kernel



Parameter Passing between Programs & Kernel

- **Using Registers (Limited Space)**
  - This is the simplest method.
  - The program puts the information directly into special locations in the CPU called registers.
  - However, there are only a few registers, so this might not work if there's a lot of information to send.

Parameter Passing between Programs & Kernel

- ❑Packing Information (Like a Block)
  - ❑ If there's too much information for registers, the program can create a block of memory like a box and store all the details there.
  - ❑ Then, the program sends the address of this "block" to the OS using a register.
  - ❑ This way, the OS knows where to find all the information it needs. This approach is used by operating systems like Linux and Solaris.

Parameter Passing between Programs & Kernel

- ❑ **Stack It Up**
  - ❑ Another way is for the program to put the information one piece at a time onto a special area of memory called the stack.
  - ❑ This is like stacking dishes. Then, the OS can access the information by taking it off the stack, one piece at a time, in the reverse order it was added.
  - ❑ This method, along with the block method, allows for sending a lot of information without limitations.

Interrupts

- An interrupt in an operating system is a **signal that prompts the OS to temporarily halt its current activities** and **execute a specific function**, often referred to as an interrupt handler or interrupt service routine (ISR).

- The CPU hardware has a wire called the interrupt-request line that the CPU senses after executing every instruction.

Interrupts - Interrupt Handling Process - Example

I/O Request Initiated

- The CPU issues an I/O command (e.g., read/write) to a device via its device driver (part of the OS).

I/O Controller Takes Over

- The I/O controller (hardware) receives the command and begins the data transfer between the device and memory.

CPU Continues Working

- While I/O is in progress, the CPU continues executing other instructions (non-blocking operation).

I/O Operation Completes

- When the I/O finishes (or an error occurs), the I/O controller sends an interrupt signal to the CPU.

Interrupt Detected

- The CPU detects the interrupt, pauses current execution, and saves the current task state (context switching).

Interrupt Handler Invoked

- The OS calls the appropriate interrupt handler (a small routine), usually part of the device driver.

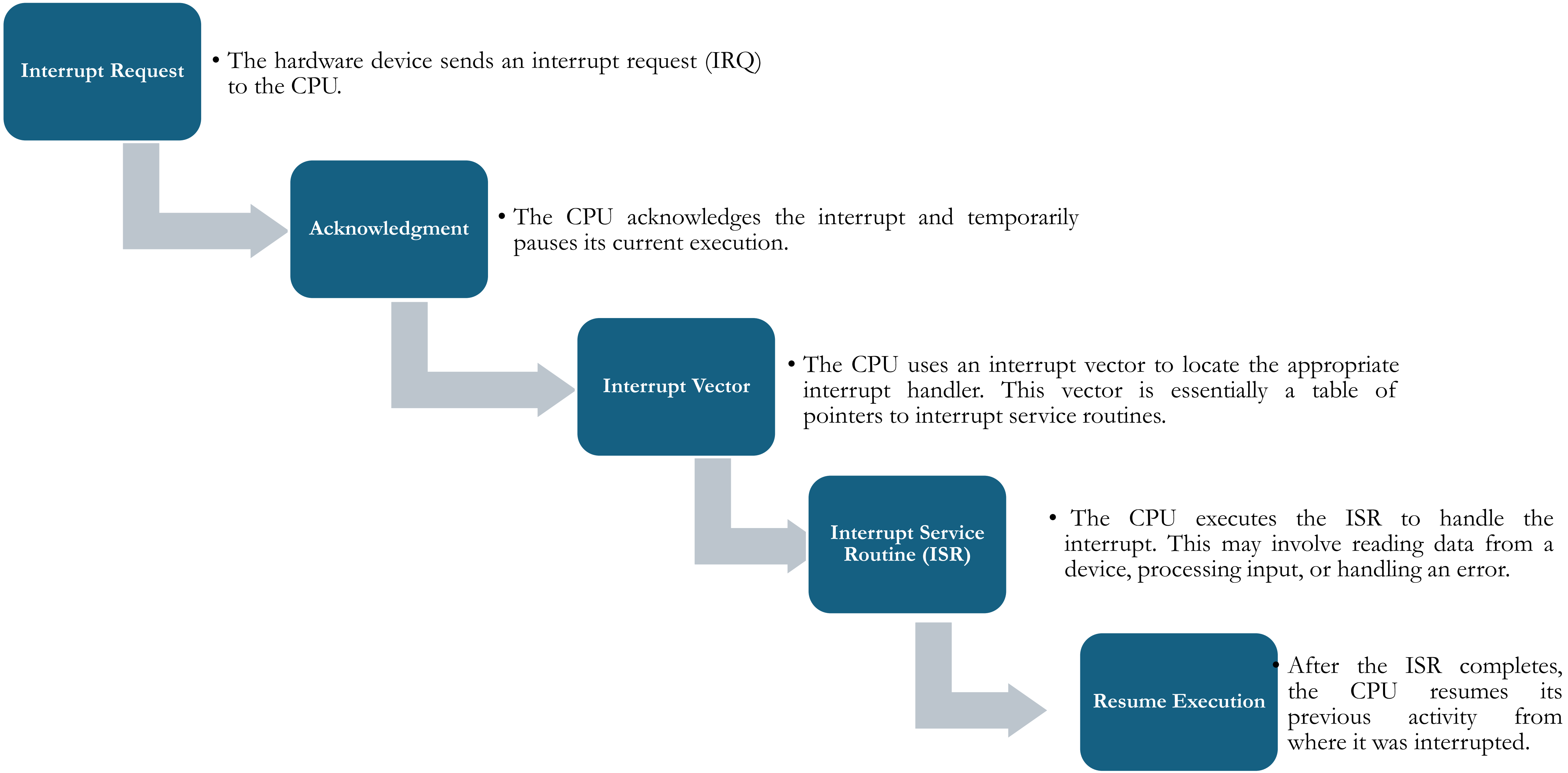
Interrupt Processed

- The handler processes the event (e.g., read input data, check for errors, signal completion).

Return to Previous Task

- Once complete, the CPU restores the previous task's state and resumes execution as if it was never interrupted.

Interrupts - Interrupt Handling Process



Interrupt Request

- The hardware device sends an interrupt request (IRQ) to the CPU.

Acknowledgment

- The CPU acknowledges the interrupt and temporarily pauses its current execution.

Interrupt Vector

- The CPU uses an interrupt vector to locate the appropriate interrupt handler. This vector is essentially a table of pointers to interrupt service routines.

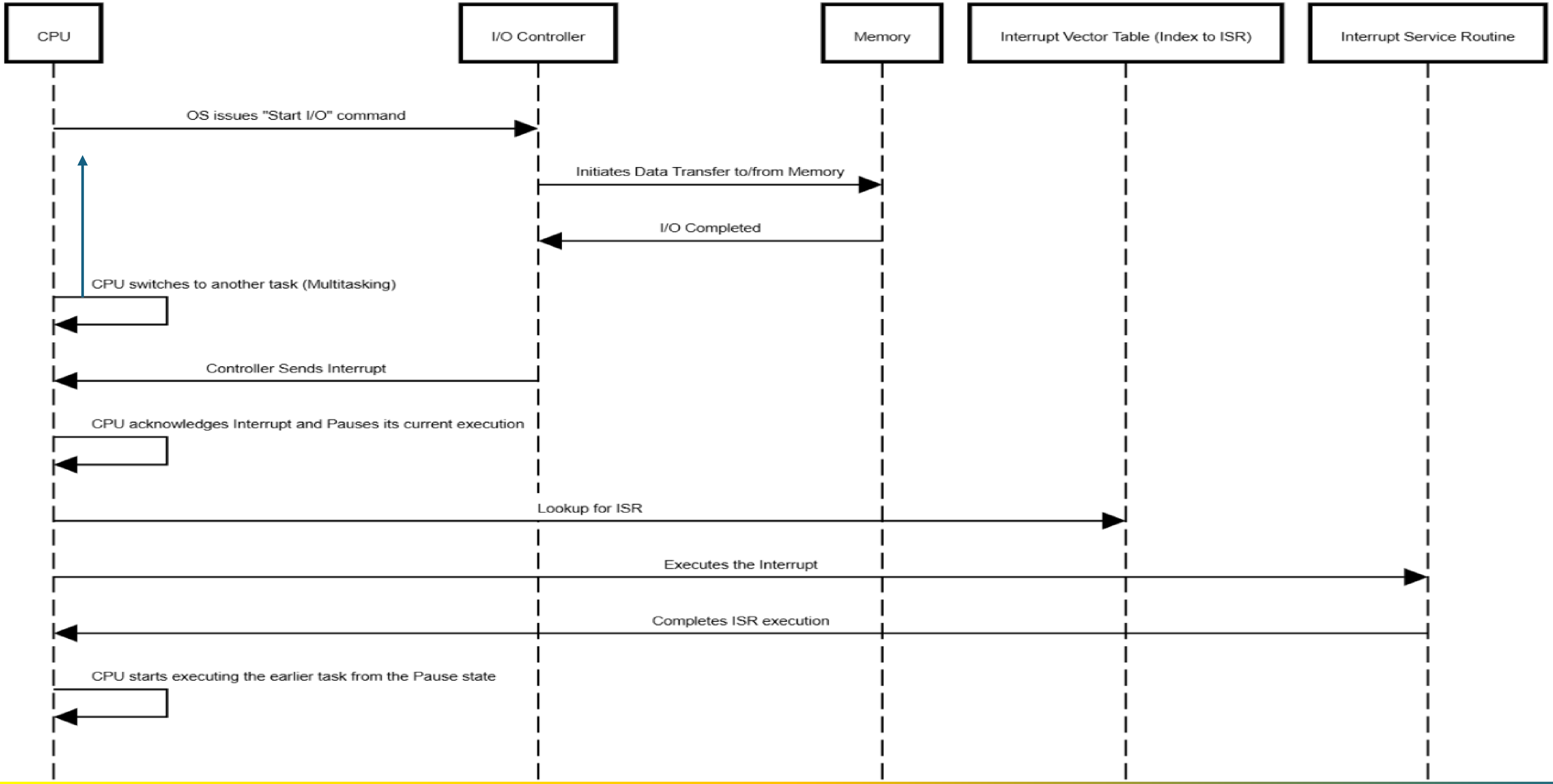
Interrupt Service Routine (ISR)

- The CPU executes the ISR to handle the interrupt. This may involve reading data from a device, processing input, or handling an error.

Resume Execution

- After the ISR completes, the CPU resumes its previous activity from where it was interrupted.

Interrupts - Interrupt Handling Process


CPU
I/O Controller
Memory
Interrupt Vector Table (Index to ISR)
Interrupt Service Routine
OS issues "Start I/O" command
Initiates Data Transfer to/from Memory
I/O Completed
CPU switches to another task (Multitasking)
Controller Sends Interrupt
CPU acknowledges Interrupt and Pauses its current execution
Lookup for ISR
Executes the Interrupt
Completes ISR execution
CPU starts executing the earlier task from the Pause state

Process

- ❑ Essentially a program that's actively running

- ❑ It is the foundation for all the computations that happen on your computer

- ❑ A process is unit of work that OS manages.

Process

- ❑ A process is an active instance of a program that includes:
  - ❑ The **program's machine code**.
  - ❑ Its **current activity** (what it's doing right now).
  - ❑ **Resources allocated** to it by the OS, such as memory, CPU time, and open files.

Process – Memory Layout

- ❑ When the operating system loads a program to run it (creating a process), it doesn't just dump the code into memory randomly.

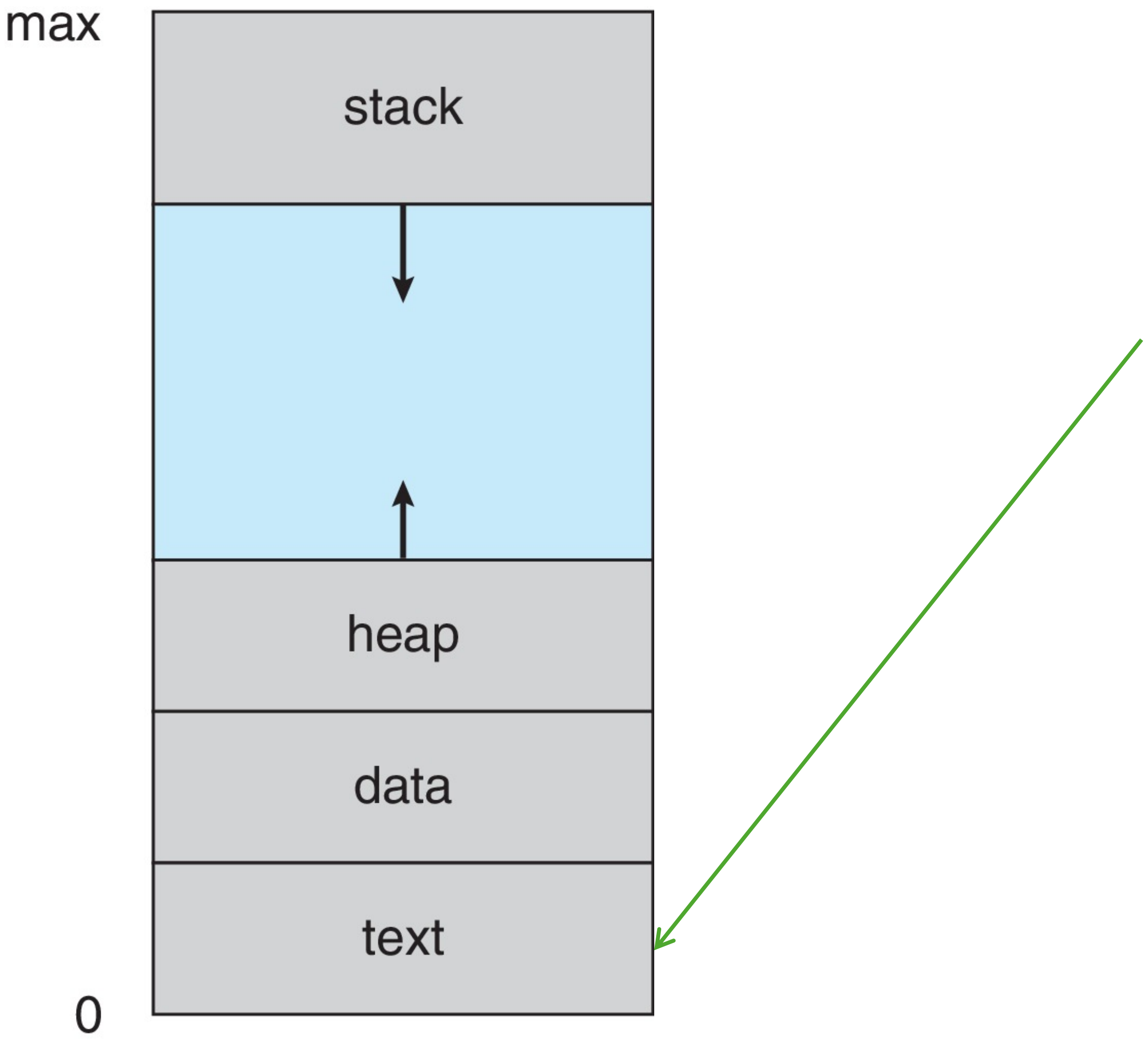
- ❑ It organizes the process's memory into a logical, standardized structure called the process address space

Process – How stored in Memory

- ❑A process in an operating system has its own virtual address space, separate from other processes.

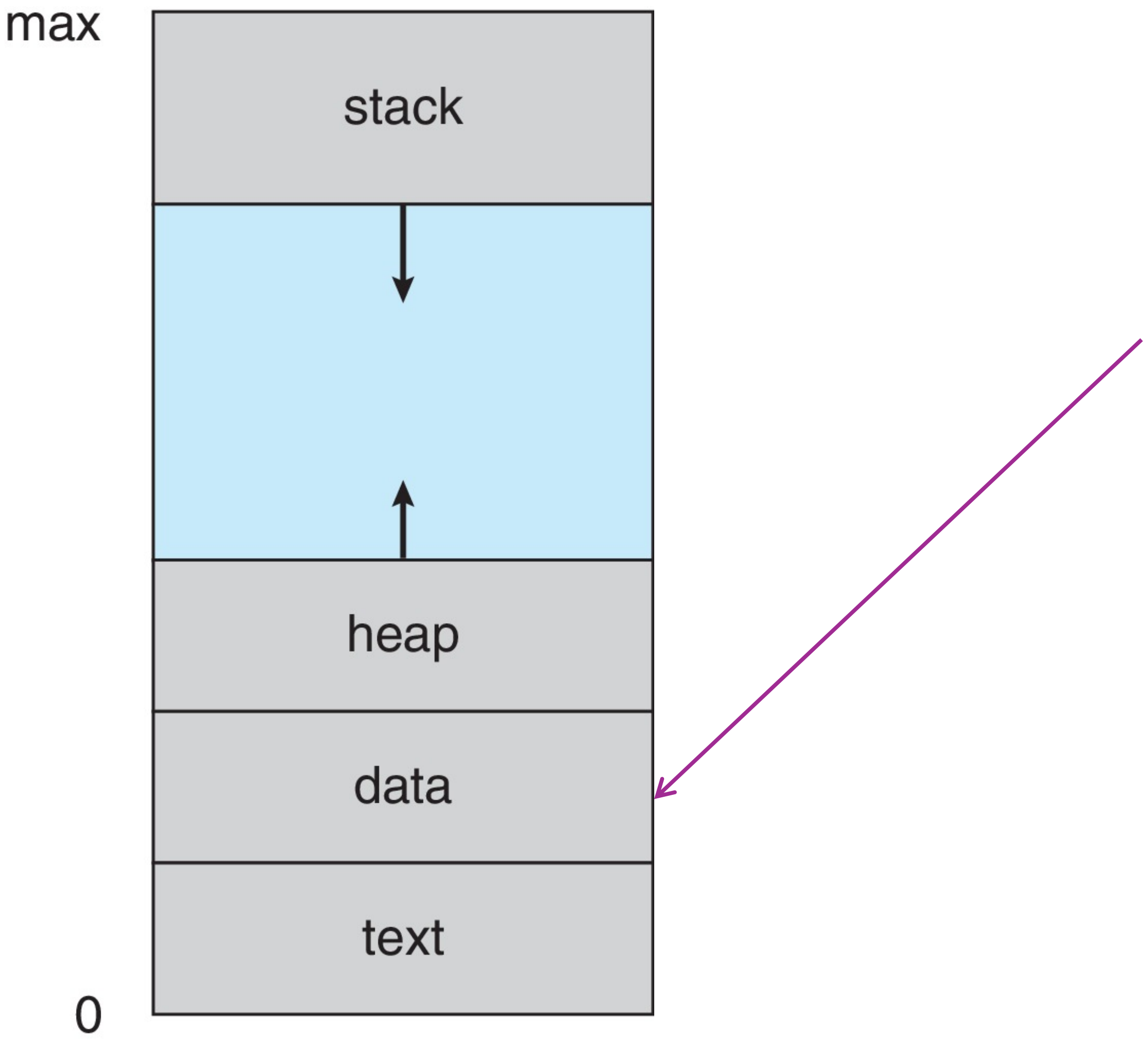
- ❑This virtual space is typically divided into distinct regions:
  - ❑ Text
  - ❑ Data
  - ❑ Heap
  - ❑ Stack

Process – How stored in Memory



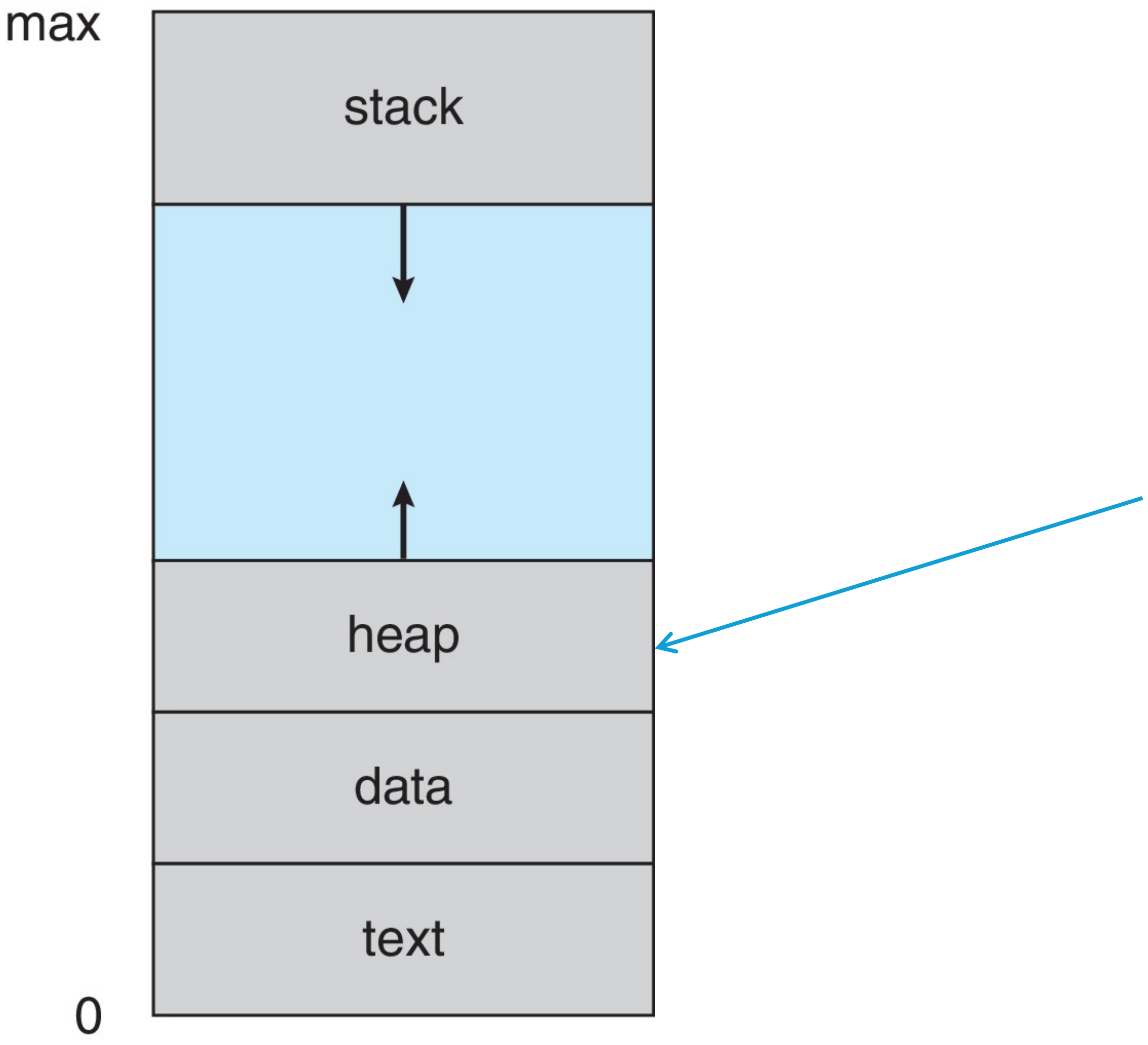
- ❑ Contains the compiled machine code of the program.
- ❑ This segment is often marked as read-only so that the process cannot accidentally modify its own instructions.
- ❑ It has a fixed size.

Process – How stored in Memory



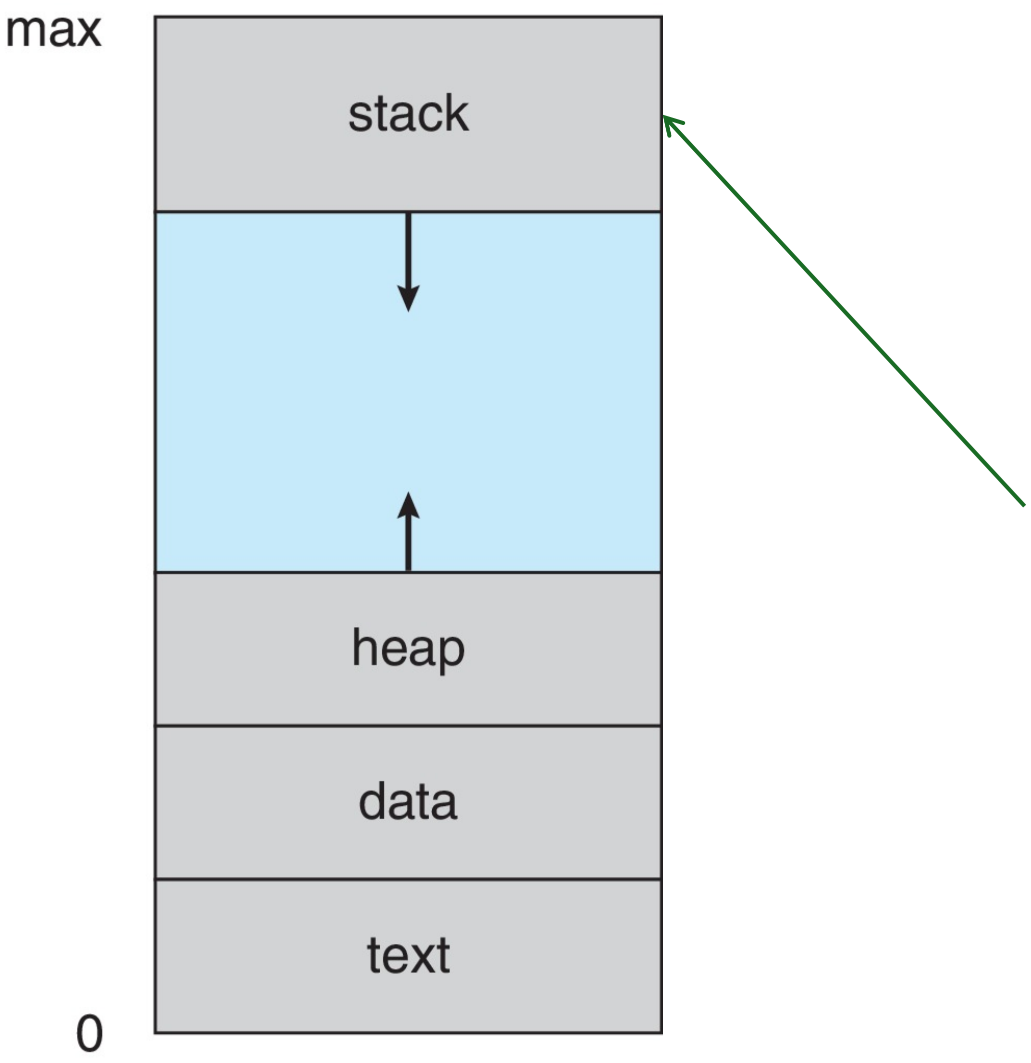
- ❑ Contains global and static variables that are uninitialized or initialized by the programmer.
- ❑ For example, int max_users = 100;.
- ❑ The size of this segment is also fixed, as it is known at compile time.
- ❑ For example, static int counter;.

Process – How stored in Memory



- ❑ This is the area for dynamic memory allocation.
- ❑ When your program needs more memory while it's running—for example, when you use malloc() in C or the new operator in C++—the memory is allocated from the heap.
- ❑ The heap starts at the end of the data or BSS segment and grows upwards towards higher memory addresses as more memory is requested.

Process – How stored in Memory



- ❑ Used for static, local memory allocation.
- ❑ It stores local variables, function parameters, and return addresses for function calls.
- ❑ Every time a function is called, a "stack frame" containing its local variables and context is pushed onto the stack.
- ❑ When the function returns, the frame is popped off.
- ❑ The stack is located at the top of the address space and grows downwards towards lower memory addresses.

Process Memory Layout - Example

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 10;             // Global variable
void my_function() {           // Function with local variables

    int local_var = 20;
    char *ptr = malloc(10);
    printf("Inside function:\n");
    printf("Global variable: %d\n", global_var);
    printf("Local variable: %d\n", local_var);
    printf("Malloced memory: %s\n", ptr);
    free(ptr);
}
int main() {
    my_function();
    printf("Outside function:\n");
    printf("Global variable: %d\n", global_var);
    return 0;
}
```

Process Memory Layout - Example

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 10;                // Global variable
void my_function()                 // Function with local variables
{
            int local_var = 20;
            char *ptr = malloc(10);
            printf("Inside function:\n");
            printf("Global variable: %d\n", global_var);
            printf("Local variable: %d\n", local_var);
            printf("Malloced memory: %s\n", ptr);
            free(ptr);
}
int main()
{
            my_function();
            printf("Outside function:\n");
            printf("Global variable: %d\n", global_var);
            return 0;
}
```

Program converted to Machine Code and this machine code is stored in **text section**

Process Memory Layout - Example

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 10;            // Global variable
void my_function()              // Function with local variables
{
        int local_var = 20;
        char *ptr = malloc(10);
        printf("Inside function:\n");
        printf("Global variable: %d\n", global_var);
        printf("Local variable: %d\n", local_var);
        printf("Malloced memory: %s\n", ptr);
        free(ptr);
}
int main()
{
        my_function();
        printf("Outside function:\n");
        printf("Global variable: %d\n", global_var);
        return 0;
}
```

Global variable which is initialized and will be stored in Data section and no uninitialized variable available so (block starting symbol) BSS section will be empty.

Process Memory Layout - Example

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 10;            // Global variable
void my_function()              // Function with local variables
{
            int local_var = 20;
            char *ptr = malloc(10);
            printf("Inside function:\n");
            printf("Global variable: %d\n", global_var);
            printf("Local variable: %d\n", local_var);
            printf("Malloced memory: %s\n", ptr);
            free(ptr);
}
int main()
{
            my_function();
            printf("Outside function:\n");
            printf("Global variable: %d\n", global_var);
            return 0;
}
```

My_function () and its local variables local_var will be stored in Stack.

In case of char *ptr variable **only the variable and return address** will be stored in the stack.

(local variables, Function parameters, return address)

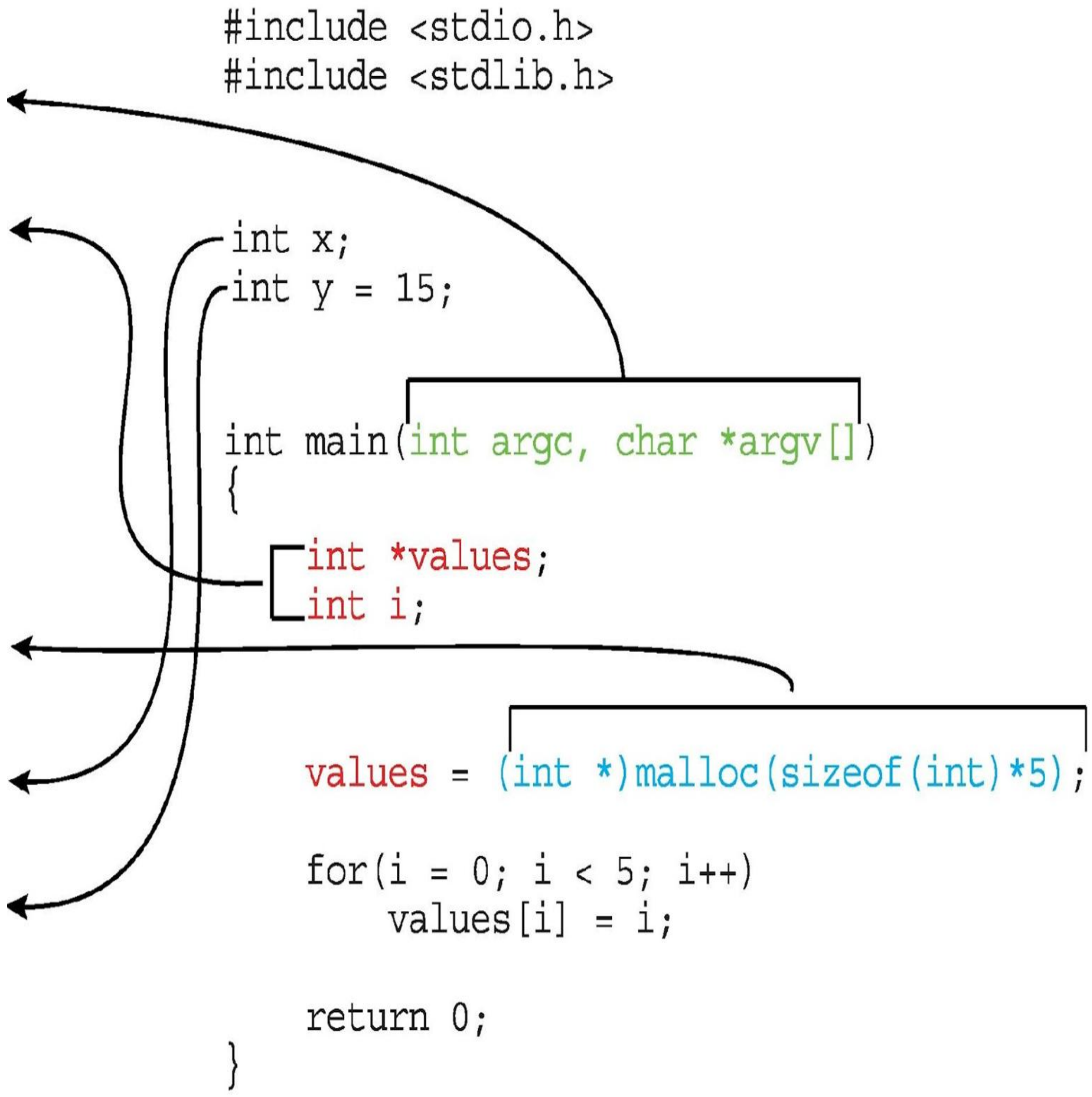
Int main() will be stored in Stack.

Process Memory Layout - Example

```
#include <stdio.h>
#include <stdlib.h>
int global_var = 10;               // Global variable
void my_function()                // Function with local variables
{
            int local_var = 20;
            char *ptr = malloc(10);
            printf("Inside function:\n");
            printf("Global variable: %d\n", global_var);
            printf("Local variable: %d\n", local_var);
            printf("Malloced memory: %s\n", ptr);
            free(ptr);
}
int main()
{
            my_function();
            printf("Outside function:\n");
            printf("Global variable: %d\n", global_var);
            return 0;
}
```

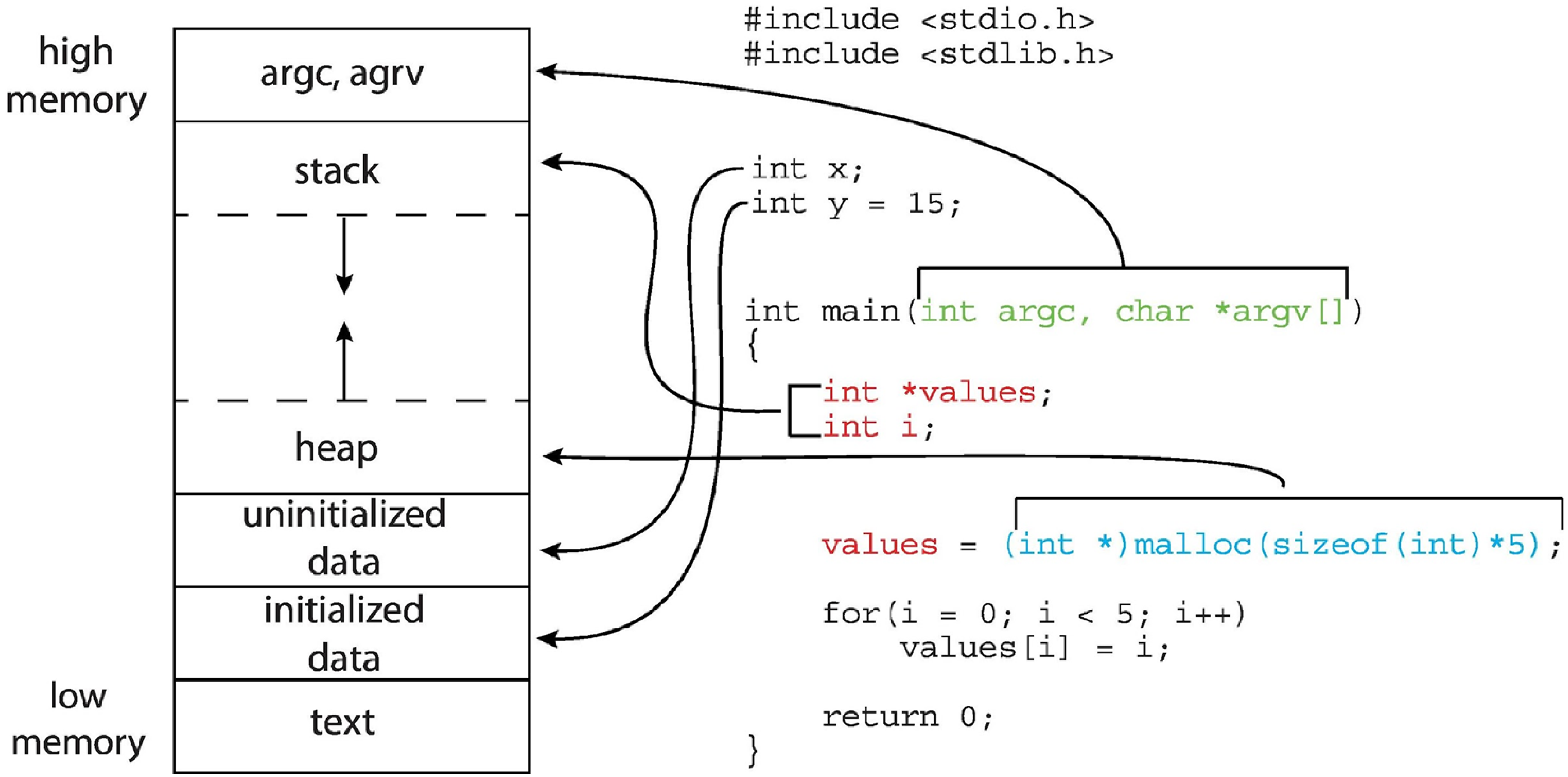
The program is asking for 10 bytes of memory while in executing status so heap segment will provide memory in runtime and all the intermediary data will be stored in heap segment

Process Memory Layout – Example 2



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
   int *values;
   int i;

   values = (int *)malloc(sizeof(int)*5);

   for(i = 0; i < 5; i++)
      values[i] = i;

   return 0;
}
```

Process Memory Layout – Example 2



```
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

Process

- ❑The operating system's memory management unit (MMU) translates virtual addresses used by the process into physical memory addresses.

Process Control Block

- ❑ A Data structure containing all the information about a process
- ❑ The PCB contains pointers to the process's memory layout.
- ❑ Each process is represented in the operating system by a process control block (PCB)—also called a task control block.
- ❑ The PCB acts as the handle for the OS to manage the process's memory layout, scheduling and other resources.

Process Control Block

Process ID
Process State
Process Priority
Program Counter
CPU registers
Heap & Stack Memory Limits
I/O Permissions & Status
CPU Scheduling
Accounting Information
List of Open files
………..
Pointers

Process Control Block

Process state.

- The state may be new, ready, running, waiting, halted, and so on.

Program counter.

- The counter indicates the address of the next instruction to be executed for that process.

CPU registers.

- The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward when it is rescheduled to run.

Process Control Block

CPU-scheduling information

- This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.

Memory-management information

- This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.

Accounting information

- This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

I/O status information

- This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

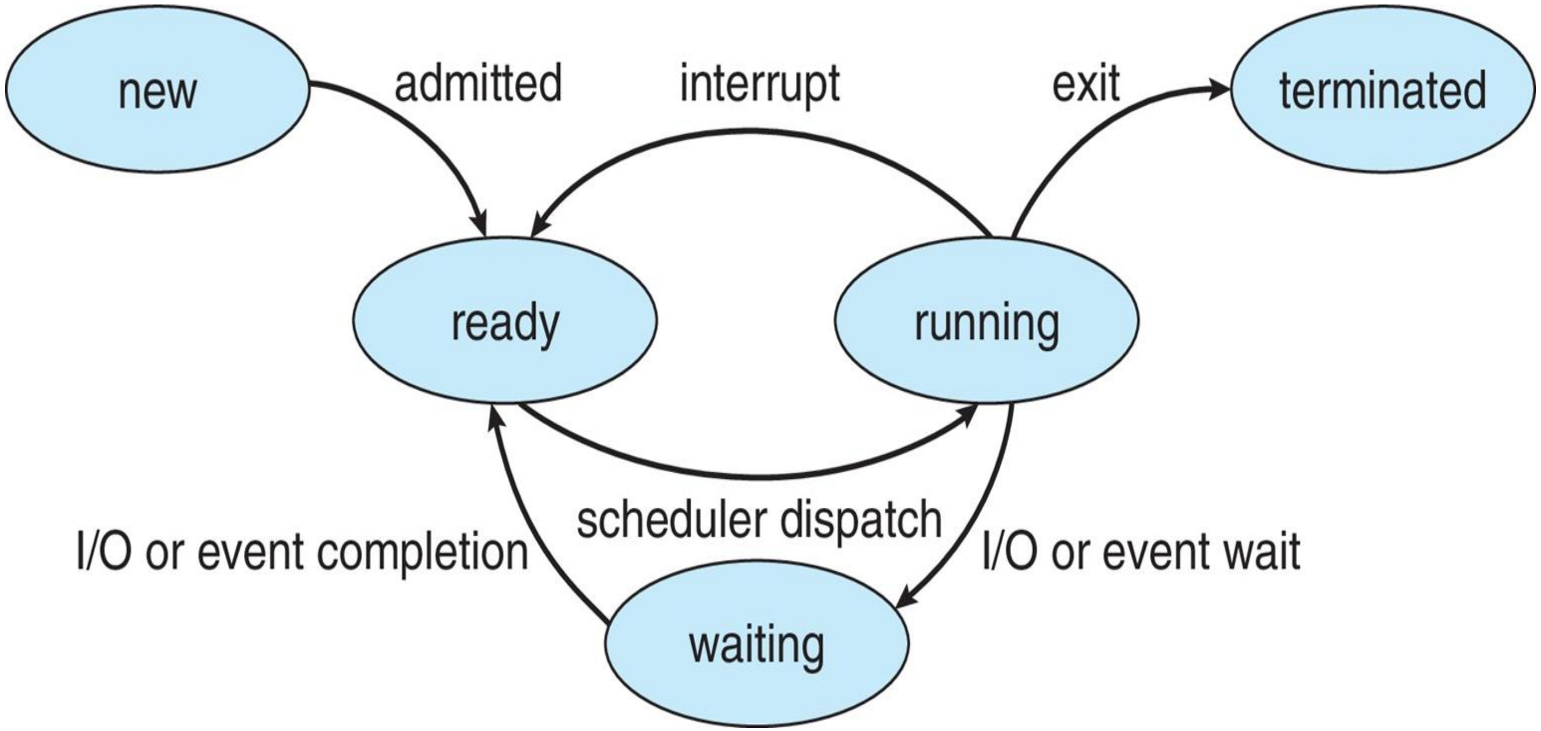
Process Tables

- ❑ A data structure maintained by the operating system to store information about all active processes.
- ❑ It's a table or array of process control blocks (PCBs).
- ❑ Each entry in the process table represents a process and contains a pointer to its corresponding PCB.

Process States

- ❑ As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

- ❑ A process may be in one of the following states

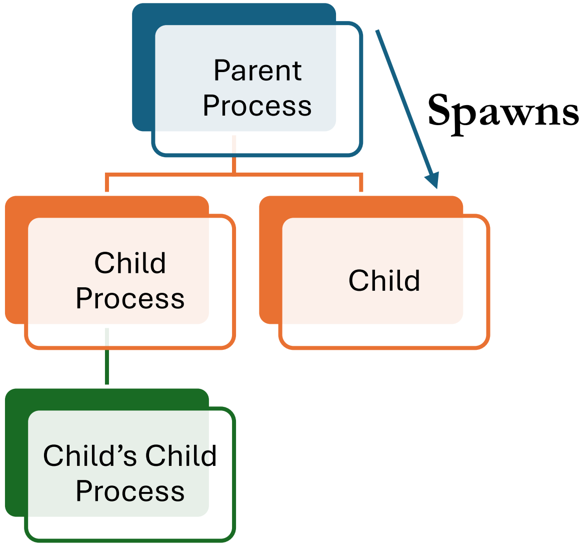


Process States

New	• The process is being created.
Running	• Instructions are being executed.
Waiting	• The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
Ready	• The process is waiting to be assigned to a processor.
Terminated	• The process has finished execution.

Process Creation

- ❑A process in operating systems is an active instance of a program in execution, encompassing its code, data, resources, and execution state.
- ❑ Creation allows the system to generate new processes, enabling multitasking, resource sharing, and efficient program execution.



Process Creation

- ❑ Process creation typically occurs when an existing process (parent) spawns a new one (child), forming a hierarchical tree structure.
- ❑ This hierarchy helps manage dependencies and resource allocation.
- ❑ The operating system oversees creation by allocating resources like memory, CPU time, and I/O devices.
- ❑ Each **new process** receives a **unique Process Identifier (PID)** and a **Process Control Block (PCB)** to track its state, program counter, and registers

Process Creation

- ❑How Resource Allocation, Resource Sharing, Address Space and Execution happens when child processes are created?
  - ❑ When a child process is created, it needs resources like CPU time, memory, files, and I/O devices to execute.
    - ❑ The child process can **obtain resources directly from the OS**
    - ❑ The child process can **inherit a subset of resources from** the **parent** process. (Resources are limited)
      - ❑ The **parent process may need to partition its resources among its children** (Resources are limited)

Process Execution

❑ After process creation,

  ❑ The **parent can run concurrently with the child** or

  ❑ **Wait for its one or all child's execution completion**.

❑ When a new (child) process is created, there are two possibilities for its address space

Duplicate Address Space

- The child process has the same program and data as the parent process

New Address Space

- The child process has a new program loaded into it, with its own separate address space.

Process Execution - Mechanisms

Unix-like Systems

The fork() system call creates a child as an exact duplicate of the parent, returning 0 to the child and the child's PID to the parent. Following **fork()**, the **exec()** system call can replace the child's image with a new program

Windows Systems

The **CreateProcess()** function combines creation and program loading, initializing memory and loading the specified executable

Other Methods

Alternatives include **vfork()** for shared memory until exec() or posix_spawn() for attribute-specified creation

Process Execution – Mechanisms – Unix Example

fork()

- This system call **creates a new child process** by **duplicating the parent process**. The child is an exact copy of the parent, including memory and state, but with a **unique process ID** (PID).
- It returns 0 to the child and the child's PID to the parent

exec()

- After forking, the child often uses exec() (or variants like execv(), execl()) to **replace its process image with a new program**. This loads and runs a different executable in the child's address space, without creating a new process

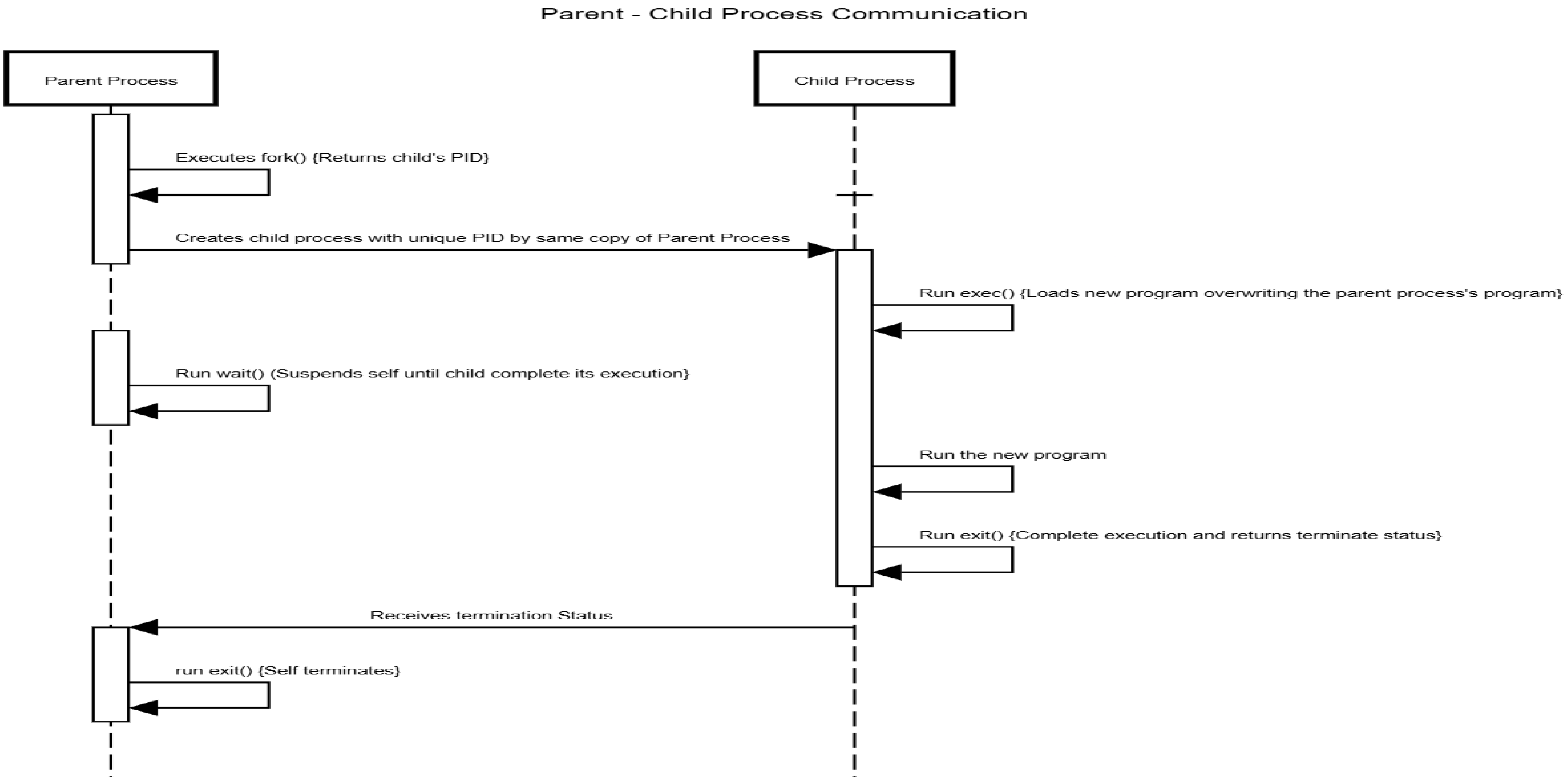
wait()

- The parent process can call wait() (or waitpid()) to **suspend its execution until the child terminates**. This **allows the parent to collect the child's exit status** and ensures proper cleanup

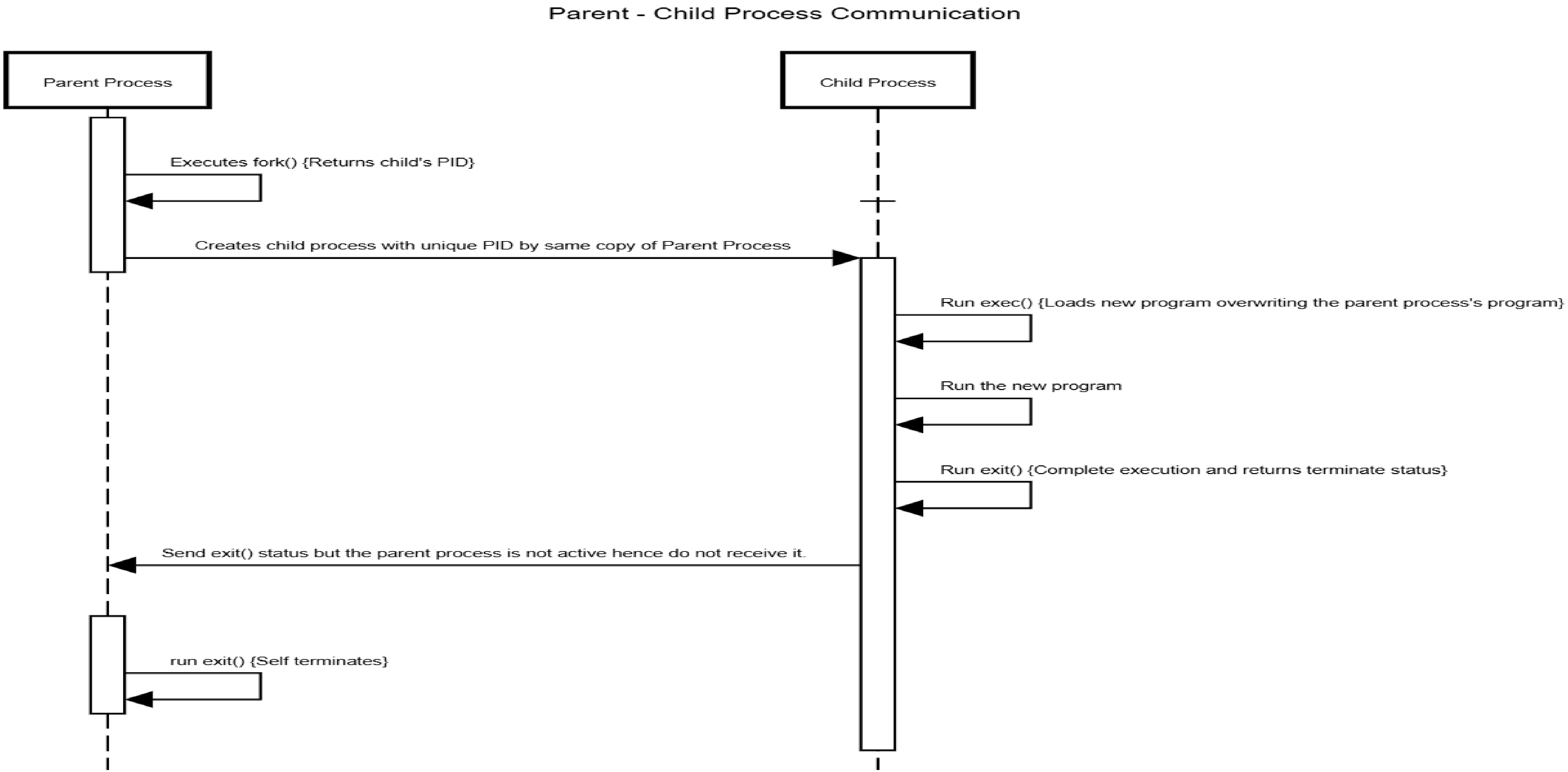
exit()

- This **terminates the process**, passing an exit status to the parent. It **decrements the process count and releases resources**. The child typically calls exit() after completing its task

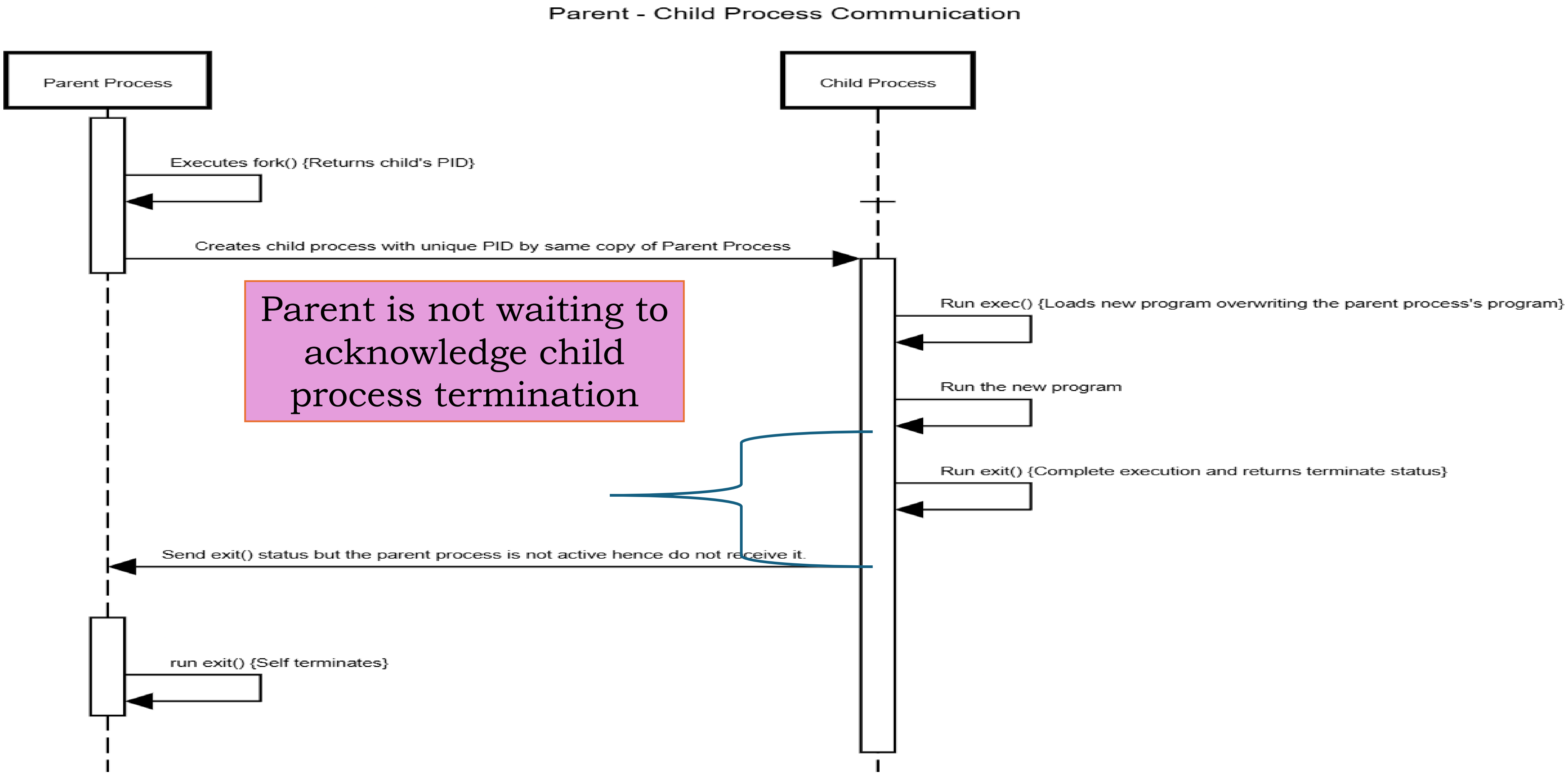
Process Execution – Mechanisms – Unix Example


Parent - Child Process Communication
Parent Process
Child Process
Executes fork() {Returns child's PID}
Creates child process with unique PID by same copy of Parent Process
Run exec() {Loads new program overwriting the parent process's program}
Run wait() (Suspends self until child complete its execution}
Run the new program
Run exit() {Complete execution and returns terminate status}
Receives termination Status
run exit() {Self terminates}

Process Execution – Mechanisms – Zombie Process


Parent - Child Process Communication
Parent Process
Child Process
Executes fork() {Returns child's PID}
Creates child process with unique PID by same copy of Parent Process
Run exec() {Loads new program overwriting the parent process's program}
Run the new program
Run exit() {Complete execution and returns terminate status}
Send exit() status but the parent process is not active hence do not receive it.
run exit() {Self terminates}

Process Execution – Mechanisms – Zombie Process



Parent - Child Process Communication
Parent Process
Child Process
Executes fork() {Returns child's PID}
Creates child process with unique PID by same copy of Parent Process
Run exec() {Loads new program overwriting the parent process's program}
Run the new program
Run exit() {Complete execution and returns terminate status}
Send exit() status but the parent process is not active hence do not receive it.
run exit() {Self terminates}

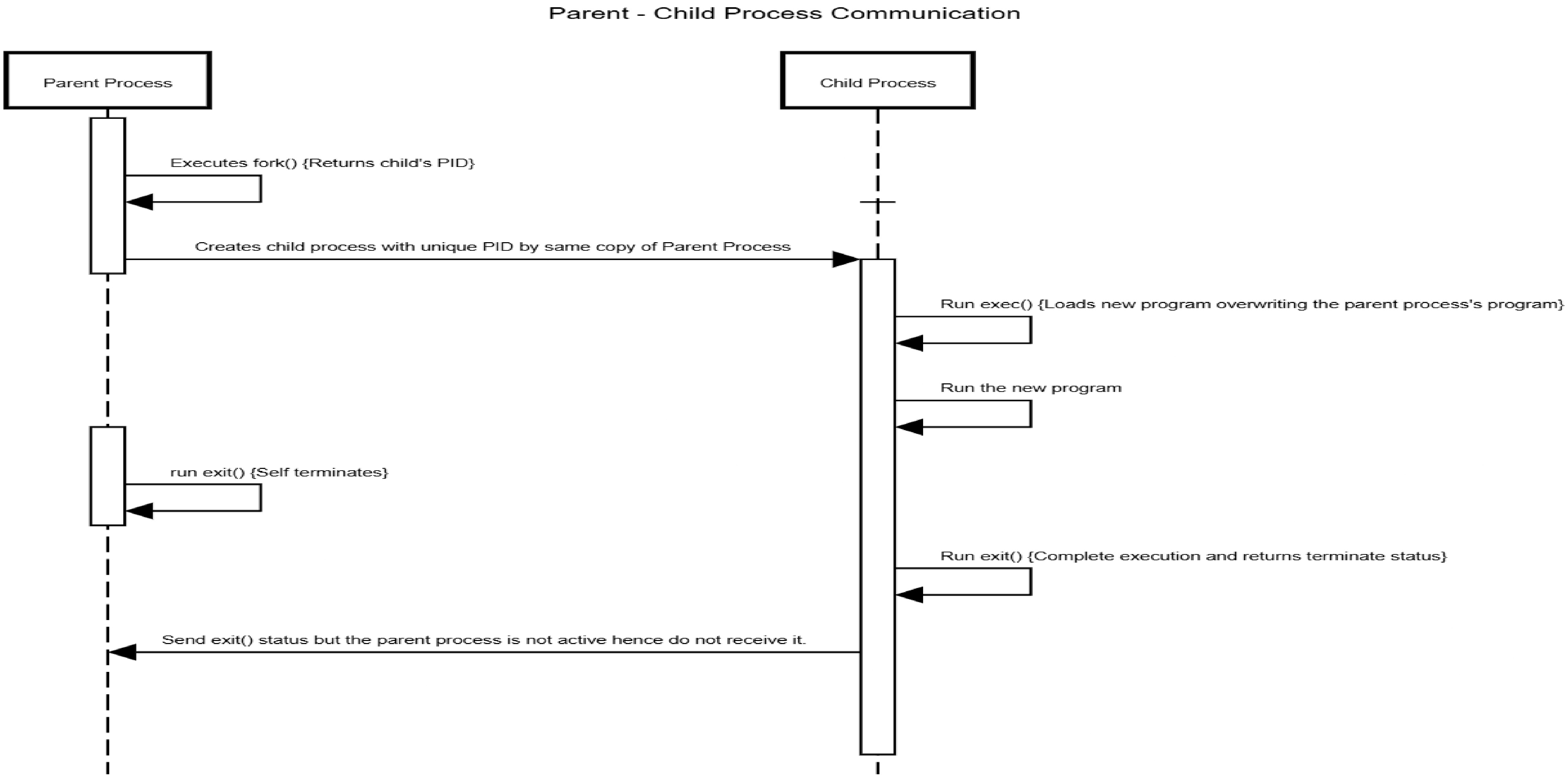
Parent is not waiting to acknowledge child process termination

Process Execution – Mechanisms – Zombie Process

Zombie Process

- ❑ A zombie process occurs when a child process completes execution (calls exit()) but remains in the process table because the parent hasn't called wait() to collect its exit status.
- ❑ The child is "dead" but not fully removed, holding a PID and minimal resources until the parent acknowledges.
- ❑ Resolution - The parent eventually calls wait(), or if the parent terminates, the zombie becomes an orphan and is adopted by the init process (PID 1), which cleans it up

Process Execution – Mechanisms – Orphan Process



Parent - Child Process Communication

Parent Process

Child Process

Executes fork() {Returns child's PID}

Creates child process with unique PID by same copy of Parent Process

Run exec() {Loads new program overwriting the parent process's program}

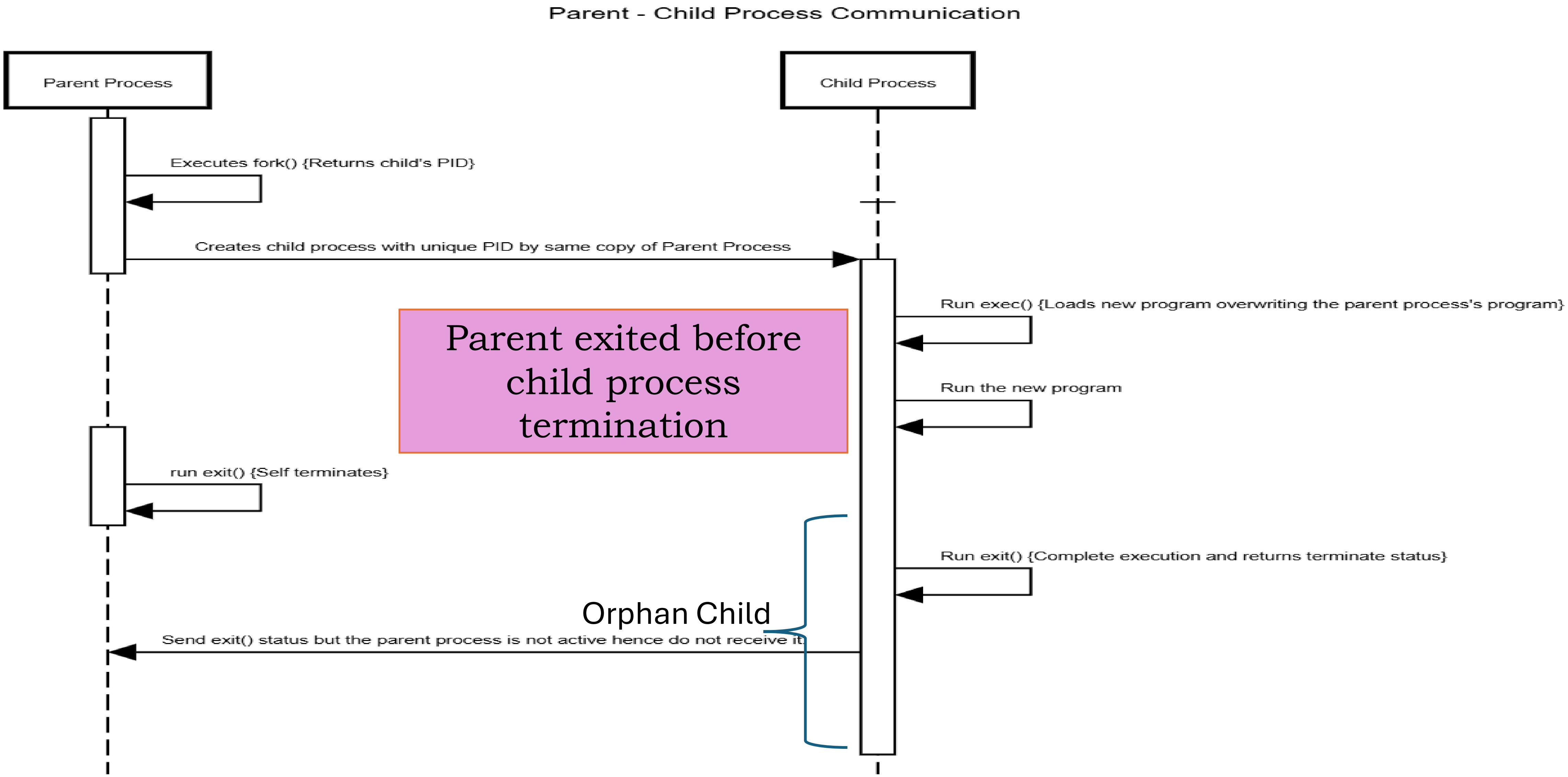
Run the new program

run exit() {Self terminates}

Run exit() {Complete execution and returns terminate status}

Send exit() status but the parent process is not active hence do not receive it.

Process Execution – Mechanisms – Orphan Process



Parent - Child Process Communication

Parent Process

Child Process

Executes fork() {Returns child's PID}

Creates child process with unique PID by same copy of Parent Process

Run exec() {Loads new program overwriting the parent process's program}

Run the new program

Parent exited before child process termination

run exit() {Self terminates}

Run exit() {Complete execution and returns terminate status}

Orphan Child

Send exit() status but the parent process is not active hence do not receive it.

Process Execution – Mechanisms – Orphan Process

Orphan Process

- ❑ An orphan process is a running child process whose parent terminates before it does, leaving the child without a parent.
- ❑ The kernel automatically re-parents it to the init process (PID 1), which handles cleanup.

Thread

- ❑ A thread is the smallest unit of execution within a process, representing a sequence of programmed instructions that the operating system can manage independently.

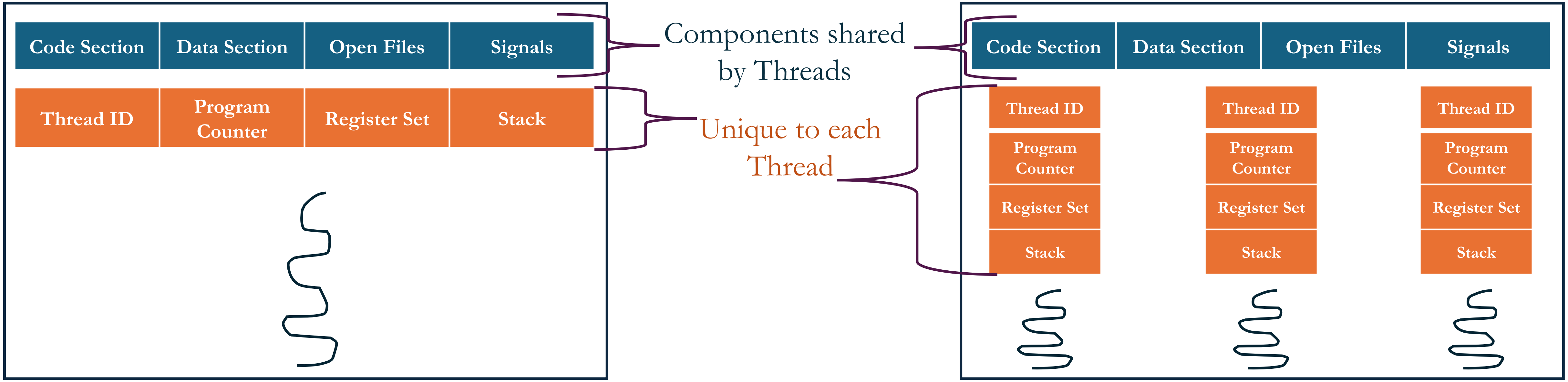
- ❑ It operates in the context of a process, sharing the process's resources like memory and open files, but each thread has its own stack, program counter, and registers.

- ❑ Threads are often called lightweight processes because they enable concurrency without the overhead of creating separate processes.

- ❑ Example - In a web browser, a single process might run multiple threads: one for rendering the user interface, another for handling network requests, and a third for executing JavaScript. This allows the browser to remain responsive while performing background tasks.

Thread

- ❑ A process is an executing program that can contain one or more threads, forming a container for threads to run concurrently.
- ❑ All threads within a process share the same address space, code, data, and resources, but they can execute independently to improve efficiency and responsiveness.



Thread – Single thread

- Only one thread runs at a time, executing tasks sequentially.
- This can lead to inefficiencies, such as waiting for I/O operations, resulting in higher idle time and reduced responsiveness.
- For example, a single-threaded file downloader processes one file at a time, pausing if it encounters delays.

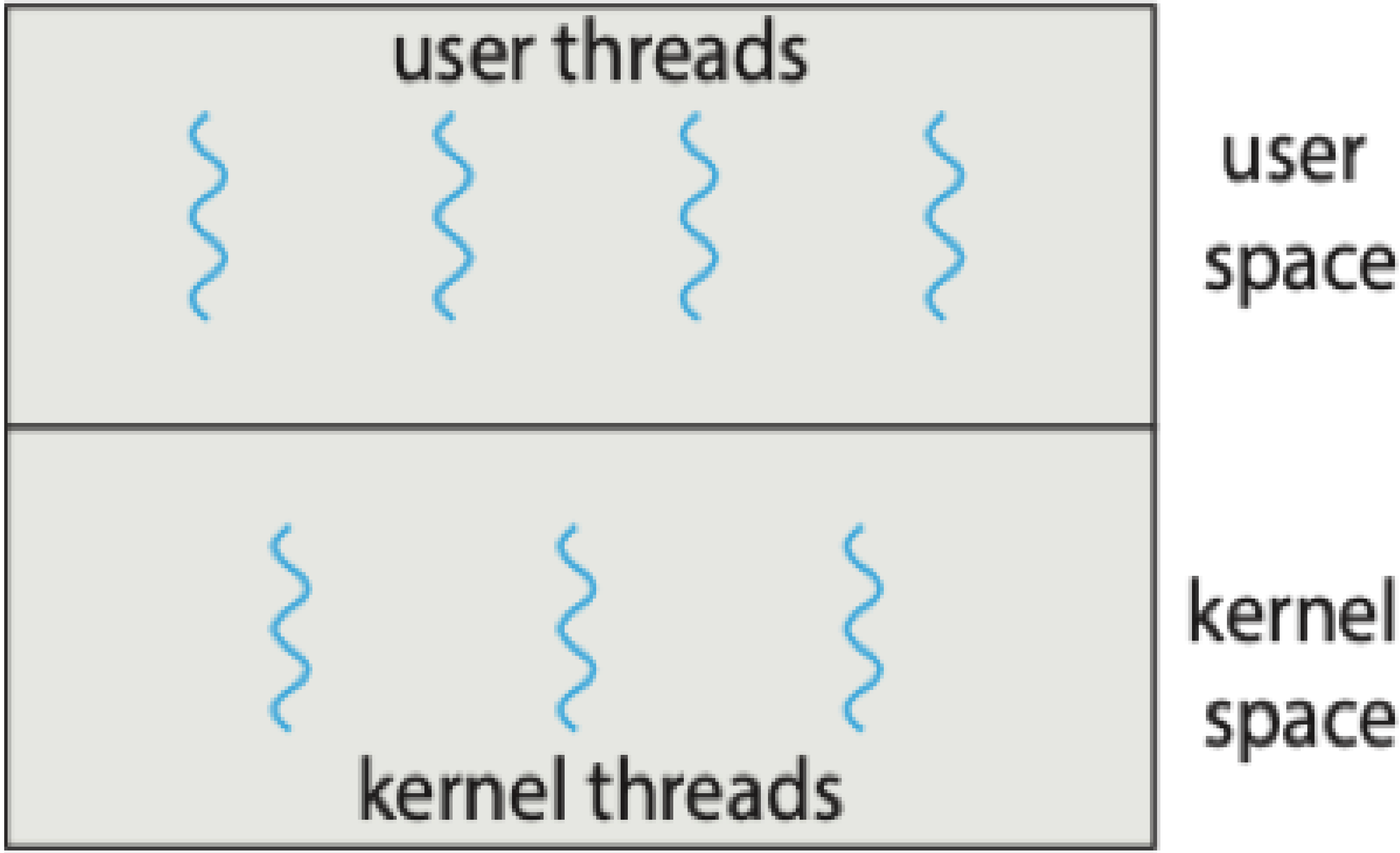
Thread – Multi thread

- ❑ Multiple threads run concurrently within the same process, allowing parallel task execution and better resource utilization, especially on multi-core processors.
- ❑ This minimizes idle time and improves performance, but it introduces complexities like synchronization.

Thread – Single Vs Multi thread

Aspect	Single Thread	Multi-Thread
Execution	Sequential, one task at a time	Concurrent, multiple tasks simultaneously
Efficiency	Higher idle time, less scalable	Minimal idle time, better scalability
Responsiveness	Can freeze during waits	Remains responsive even during delays
Example	Basic calculator app handling one operation	Web server handling multiple client requests
Synchronization	Not Required	Heavily Required

Thread – Multithreading Models



Thread – Multithreading Models

User Threads	Kernel Threads
Managed by user-level libraries: created and managed entirely within a user application	Managed by the operating system: created and managed by the kernel
Lightweight: Context switching between user threads is typically faster as it involves only switching between user-level registers and stacks	Heavier weight: Context switching between kernel threads is slower as it involves saving and restoring kernel-level state as well.
No direct OS support: The kernel is unaware of user threads.	Direct OS support: The kernel is aware of kernel threads and can schedule them independently.
Susceptible to blocking: If one user thread performs a blocking system call, the entire process is blocked, including all other user threads.	Resistant to blocking: If one kernel thread is blocked, other threads within the same process can continue execution.

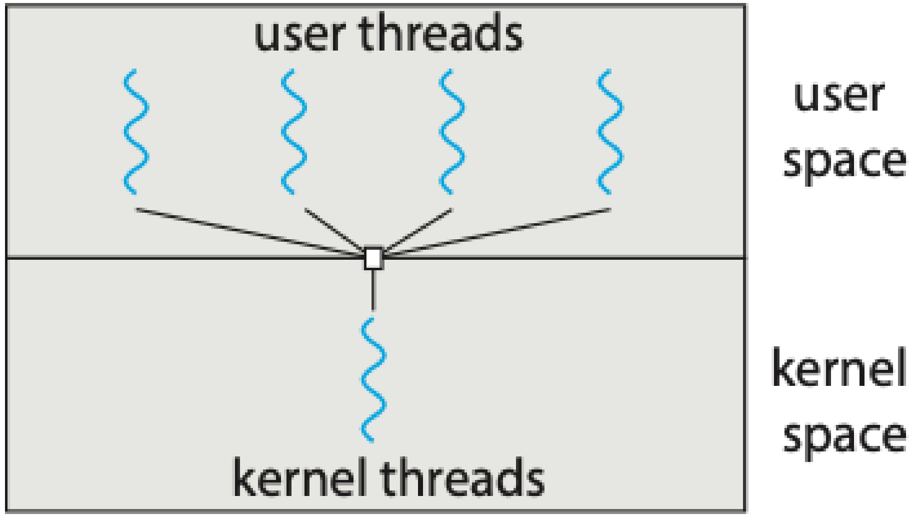
Thread – Multithreading Models

- ❑ Multithreading models define how user-level threads (managed by applications) map to kernel-level threads (managed by the operating system).
- ❑ These models balance concurrency, efficiency, and overhead

Thread – Multithreading Models

❑ **Many-to-One Model**

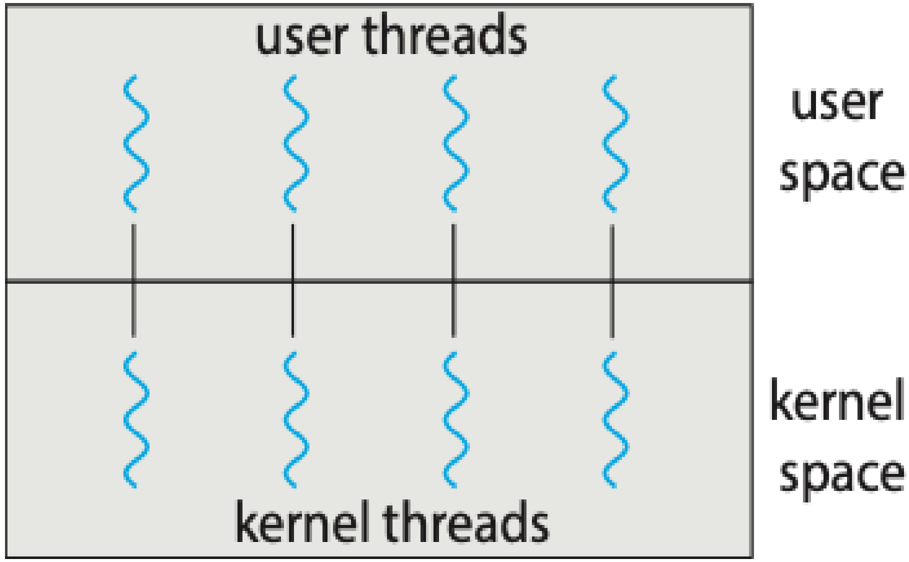
- ❑ Multiple user threads map to a single kernel thread, with management in user space for efficiency.
- ❑ However, a blocking call halts the entire process, limiting parallelism.
- ❑ Efficient for processes that don't perform I/O-bound tasks
- ❑ Example: Green threads in some Java implementations, Early versions of UNIX



Thread – Multithreading Models

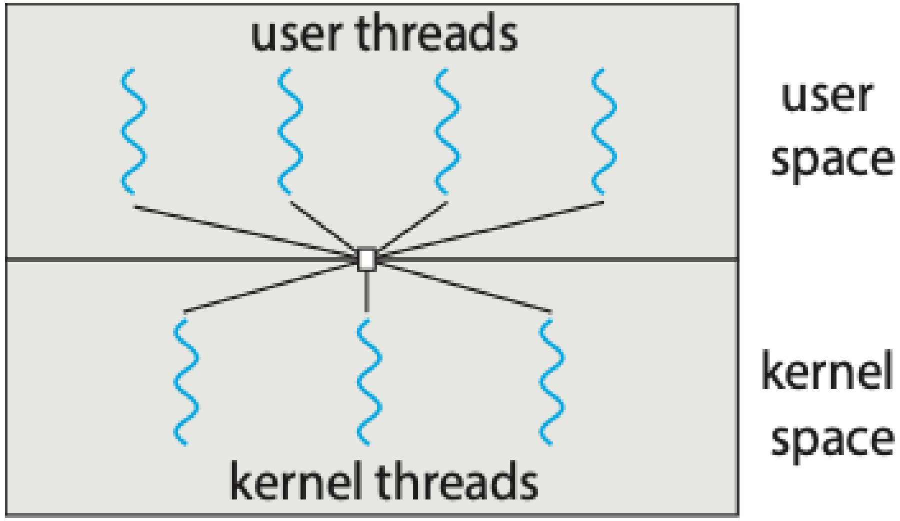
❑ One-to-One Model

- ❑ Each user thread maps to a dedicated kernel thread, enabling true concurrency and parallel execution on multiprocessors.
- ❑ Each user thread can be independently scheduled and blocked
- ❑ Creating a kernel thread for each user thread can be resource-intensive and be overhead.
- ❑ Example: Used in Linux and Windows



Thread – Multithreading Models

- Many-to-Many Model
  - Multiple user threads multiplex onto a smaller or equal number of kernel threads, allowing flexible concurrency without excessive overhead.
  - It supports parallelism and handles blocking calls well.
  - Multiple user threads can continue execution even if one kernel thread is blocked.
  - Requires careful management of thread mapping and scheduling
  - Example: Database servers managing numerous queries across limited kernel threads.



Thread – Multithreading Models

Aspect	Many to One	One to One	Many to Many
Mapping	Multiple user threads to one kernel thread	One user thread per kernel thread	Multiple user threads to multiple kernel threads
Concurrency	Limited; no true parallelism on multiprocessors	High; supports parallel execution	Balanced; enables parallelism without excessive overhead
Blocking Behavior	One thread's block halts the process	Only the blocking thread is affected	Kernel schedules around blocks
Overhead	Low; user-space management	High; kernel thread creation for each user thread	Moderate; flexible thread allocation
Scalability	High for simple tasks but poor on multicore systems	Limited by kernel thread limits	High; supports many threads efficiently
Best Use Cases	Lightweight, non-parallel applications	CPU-bound tasks needing concurrency	Complex applications with variable workloads

Thread – Multithreading Models

Aspect	Many to One	One to One	Many to Many
Advantage	Efficiency and low overhead, as thread creation avoids kernel involvement	Strong concurrency and multiprocessor utilization	Scalability and balanced resource use, avoiding full process blocks
Disadvantage	Lack of multiprocessor support and process-wide blocking	Performance burdens from excessive kernel threads	Added complexity in implementation

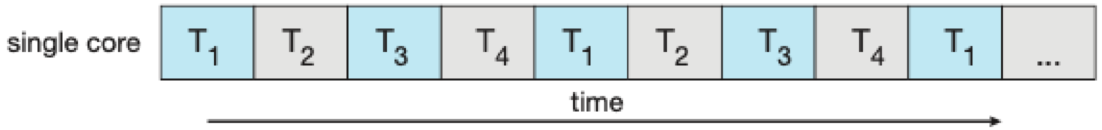
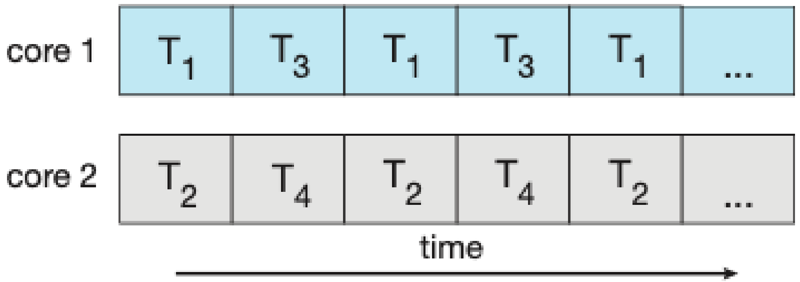
Multicore Programming

- ❑ A multicore processor is a single physical processor that contains multiple processing cores.
- ❑ Each core can execute instructions independently, allowing for multiple tasks to be performed simultaneously.

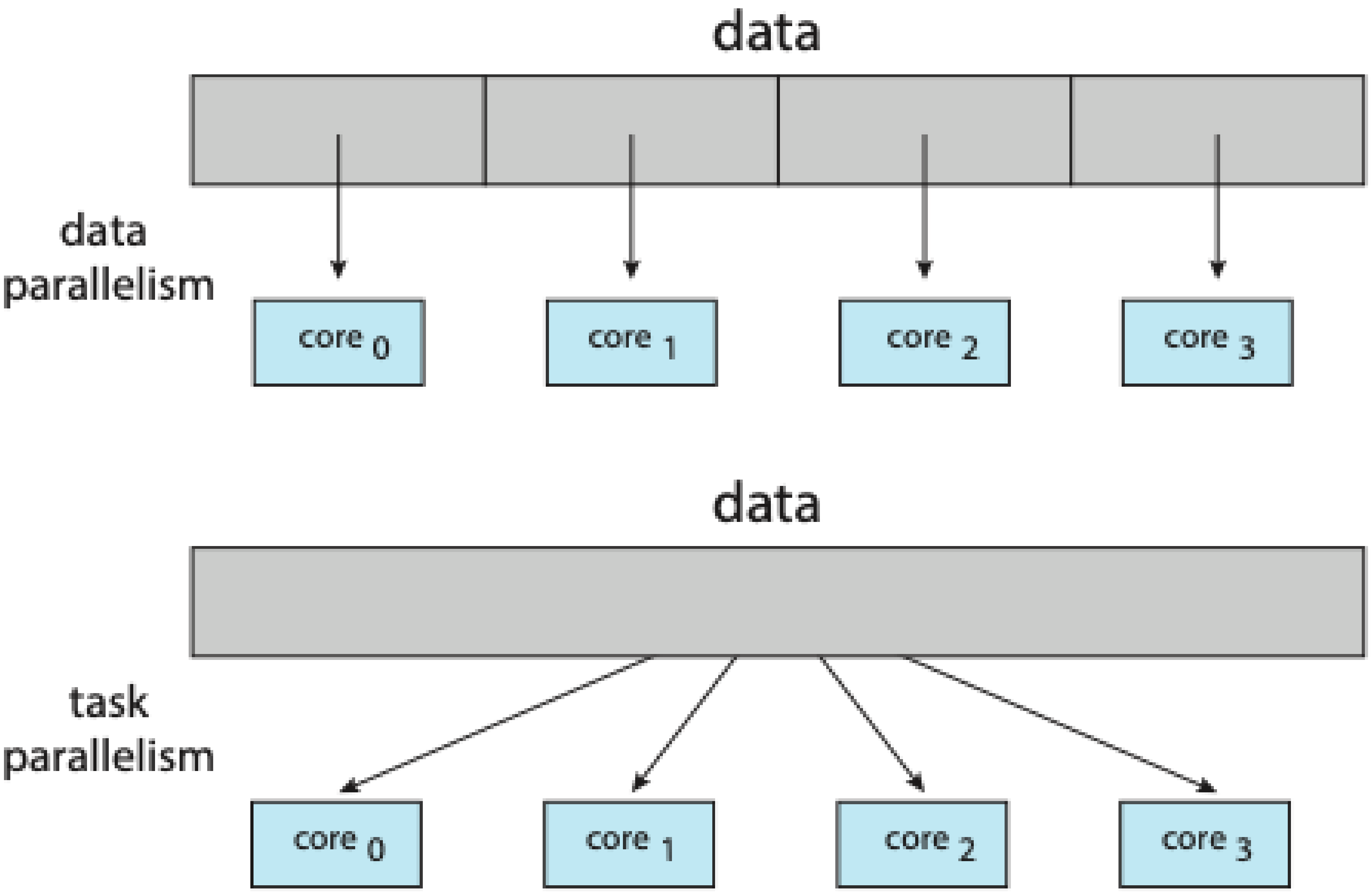
Multicore Programming - Execution

Parallel Execution	Concurrent Execution
Parallel execution refers to the simultaneous execution of multiple tasks or threads on multiple processing cores.	Concurrency execution refers to the ability of a program to perform multiple tasks or threads simultaneously, but not necessarily at the same instant. Concurrency is achieved through **context switching**, where the processor switches between tasks quickly.

Multicore Programming - Execution

Single core System	Multicore System
Concurrency merely means that the execution of the threads will be interleaved over time because the processing core can execute only one thread at a time.	**Concurrency** means that some threads can run in **parallel**, because the system can assign a separate thread to each core.
 single core T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 ... time	 core 1 T_1 T_3 T_1 T_3 T_1 ... core 2 T_2 T_4 T_2 T_4 T_2 ... time
A concurrent system supports more than one task by allowing all the tasks to make progress. .(concurrency without parallelism)	A parallel system can perform more than one task simultaneously.

Multicore Programming – Parallelism Type



Multicore Programming – Parallelism Type

- ❑Parallelism is a technique to achieve faster execution by dividing a task into smaller sub-tasks that can be executed concurrently.

- ❑ Data Parallelism
  - ❑ Focuses on distributing a large dataset across multiple processor cores and having each core perform the exact same operation on its subset of the data.
  - ❑ **One operation, many pieces of data**
  - ❑ Imagine you have a massive image and you want to increase the brightness of every pixel. Instead of one core processing all pixels one by one, you can split the image into four sections. You then assign each section to a different core, and all four cores run the "increase brightness" function simultaneously.

Multicore Programming – Parallelism Type

❑Task Parallelism

- ❑ Focuses on distributing different, independent tasks across multiple cores to be performed concurrently. The tasks can operate on the same or different data
- ❑ **Different tasks, running at the same time**
- ❑ In a video game engine, one core might be dedicated to rendering graphics, a second core could be handling the game's physics calculations, a third could be running the AI for computer-controlled characters, and a fourth could be managing audio processing.

Multicore Programming – Parallelism Type

Data Parallelism	Task Parallelism
Used when Dealing with large datasets	Performing different operations on different data elements
Performing identical operations on each data element	Executing independent tasks concurrently
Using SIMD (Single Instruction, Multiple Data) instructions	Using MIMD (Multiple Instruction, Multiple Data) instructions
Examples • Matrix multiplication • Image processing • Scientific simulations	Examples ▪ Web server handling multiple requests ▪ Compiling multiple source files ▪ Scientific simulations with different parameters

Tutorial Questions

- ❑ Consider a web server application designed to handle many simultaneous client requests on a multi-core processor. If one of these requests requires a slow I/O operation (e.g., reading a large file from a disk), why might the entire application freeze and become unresponsive if it uses User-Level Threads, while an identical application using Kernel-Level Threads would continue to handle other requests smoothly?

Tutorial Questions

- ❑ The fundamental reason for this difference in behavior lies in whether the operating system's kernel is aware of the individual threads. With User-Level Threads, the kernel is not aware of them and sees the entire application as a single task; with Kernel-Level Threads, the kernel manages each thread independently
- ❑ Scenario with User-Level Threads (Application Freezes)
  - ❑ In this model, thread management is handled by a library in the user space, not by the operating system. The kernel only sees one entity: the main process itself. The kernel maps all the user-level threads of the application to a single kernel-level thread or process. To the kernel, your entire multi-threaded application is just one unit of execution. When one user thread makes a "blocking" system call (like reading a file from a slow disk), the kernel sees that its single unit of execution has requested a blocking operation. Consequently, the kernel puts the entire process to sleep until the I/O operation is complete. Since the kernel is unaware of the other user threads, it cannot schedule them to run. The result is that all other threads—even those that are ready to process other client requests—are paused, causing the entire application to freeze

Tutorial Questions

- Scenario with Kernel-Level Threads (Application Stays Responsive)
  - In this model, the operating system kernel directly manages every thread. Each thread is a distinct unit that the kernel can schedule independently. The kernel is aware of every single thread within the application. It sees them as separate, schedulable tasks. When one kernel thread makes a blocking I/O call, the kernel identifies that only that specific thread cannot proceed. The kernel marks just the one blocking thread as "waiting" and removes it from the queue of runnable threads. The OS scheduler is then free to assign the CPU cores to other, ready-to-run threads from the same application. This allows other client requests to be processed concurrently, and the application remains responsive

Tutorial Questions

- ❑ A software development team is designing a new high-performance application that needs to execute numerous tasks concurrently on a multi-core server. The team is evaluating different multi-threading models to determine the most suitable one. If the application chooses the Many-to-One multi-threading model, and one of its user-level threads initiates a complex database query that takes a significant amount of time to complete, why would this choice likely lead to performance bottlenecks and an unresponsive application, even if other CPU cores are idle? Conversely, explain which multi-threading model would be better suited for maintaining application responsiveness and utilizing available CPU resources in the same scenario and why?

Tutorial Questions

- ❑ In the Many-to-One model, multiple user-level threads are mapped to a single kernel-level thread . The operating system (kernel) is only aware of this single kernel thread, not the individual user threads within the application. The kernel views the entire multi-threaded application as a single unit of execution because all user threads share one kernel thread. When one user-level thread executes a blocking system call (like a slow database query), the associated single kernel thread also blocks. Since the kernel perceives only that one kernel thread as blocked, it puts the entire process to sleep, regardless of whether other user-level threads are ready to run or if other CPU cores are available . The kernel cannot schedule any other user threads because it is unaware of their existence as separate entities, leading to the application becoming unresponsive.

Tutorial Questions

- ❑Scenario with Many-to-Many Model (Responsiveness and Resource Utilization)
- ❑The kernel is aware of multiple kernel threads for the application, and user-level threads are mapped to these available kernel threads. When one user-level thread makes a blocking system call, only the specific kernel thread it's currently using might block. Crucially, the operating system can then schedule other ready user-level threads onto other available kernel threads within the same process, or even create new kernel threads if needed . This allows other tasks within the application to continue running concurrently, preventing the entire application from freezing . This model effectively utilizes multi-core processors, enabling true parallelism and maintaining responsiveness even when some threads are blocked .

Tutorial Questions

❑ Imagine a multi-tasking operating system running two independent processes: Process A, a video editor currently rendering a file, and Process B, a simple command-line shell waiting for user input. Suddenly, a hardware interrupt occurs (e.g., a timer ticks), and the operating system's scheduler decides to perform a context switch from the CPU-intensive Process A to the idle Process B. Explain the precise, step-by-step role of the *Process Table, the Process Control Block (PCB), and the concept of a Process Memory Layout in this context switch*. Specifically, detail how the operating system ensures that Process A's complex memory state is safely preserved, so it can resume rendering flawlessly later, while correctly activating the simple, waiting state of Process B.

Tutorial Questions

- ❑ The context switch is initiated by a hardware interrupt. This immediately causes the CPU to stop executing Process A's code and switch from user mode to kernel mode. This gives the operating system's kernel full control of the hardware.
- ❑The kernel consults the Process Table, which is an array or linked list containing entries for every active process. It uses the ID of the currently running process (Process A) to quickly locate the memory address of its Process Control Block (PCB).
- ❑The kernel saves the current state of the CPU's registers directly into designated fields within Process A's PCB.
- ❑This is where the PCB saves the pointers and metadata that define Process A's virtual address space. This information, stored in the PCB, is essential for reconstructing its memory environment later.
- ❑Finally, the kernel updates the process state field in Process A's PCB from "Running" to "Ready," indicating it is ready to run but not currently using the CPU.

Tutorial Questions

- ❑ Now, the scheduler selects Process B to run. The kernel prepares to load its context.
- ❑ The scheduler picks Process B from the queue of ready processes. The kernel again uses the Process Table to find the memory location of Process B's PCB.
- ❑ The kernel does the reverse of Step 2. It loads the values from Process B's PCB into the CPU's registers. Since Process B was idle and waiting, its saved Program Counter points to the instruction that checks for user input.
- ❑ The kernel uses the memory management information stored in Process B's PCB to reconfigure the Memory Management Unit (MMU). The MMU's registers are updated to point to Process B's page tables. This instantly makes Process B's memory space (its code, data, stack, and heap) the active one, effectively hiding Process A's memory from view.
- ❑ The state of Process B in its PCB is changed from "Ready" to "Running".
- ❑ The kernel loads the saved Program Counter from Process B's PCB into the CPU's PC register. It then switches the CPU back from kernel mode to user mode. Execution seamlessly resumes within Process B, which now runs as if it had always been in control.