University of Nottingham
UK | CHINA | MALAYSIA

Operating Systems and Concurrency

Lecture 3: Process

University of Nottingham, Ningbo China

2024

Recap
Last Lecture

- The **operating systems provides abstractions** (e.g., it hides hardware details) and **manages resources**
- Computer Hardware
  - Interrupts, context switching
- OS structures/implementation

Goals for Today

Overview

- Introduction to **processes** and their **implementation**
- Process **states** and state **transitions**
- Operating Control Structures
- **System calls** for process management

Processes
Definition

- Process vs. Program:
  - A program:
    - A collection of sub-routines that cooperate to perform a particular task.
    - is **passive** entity and “sits” on a disk, not doing anything.
  - Process:
    - The simplified definition: “a process is a running instance of a program”
    - A program that is currently running
- Before the concept of process was introduced in OS, early computer system used simpler and more primitive models for program execution.
  - These early models were typically single-tasking or batch processing systems, where only one program could run at a time, and there was no concept of multiple concurrent activities.

Definition: Concepts before process

- Serial Processing (Single-tasking):
  - No Operating System or Minimal Control Program: Early computers did not have sophisticated operating systems. Instead, programs were loaded manually, often through physical switches or punched cards. These programs were executed one at a time in a sequential or serial manner.
  - Manual Scheduling: a human operator would load a program into memory, initiate its execution, and wait for it to finish before starting the next program. There was no automation or management of tasks by the machine itself.
  - No Memory Protection or Isolation: Programs had direct access to the entire machine, including memory and hardware. This meant that there was no isolation between different programs, leading to potential instability if one program malfunctioned.

Definition: Concepts before process

- **Batch Processing:** Jobs (tasks) were collected into a "batch" and executed sequentially, one after the other.
  - There was no interaction between the user and the computer during the execution of these jobs.
  - The system would process the entire batch without interruptions or multitasking.
  - **Job Control Language (JCL):**
    - Early batch systems used a JCL to specify instructions for running jobs in a batch.
    - The JCL described how to load, execute, and handle I/O for each job.
  - **No Concurrent Execution:**
    - Only one program could be running at any given time.
    - There was no multitasking or overlapping of jobs.
    - While one job was being executed, others were queued.

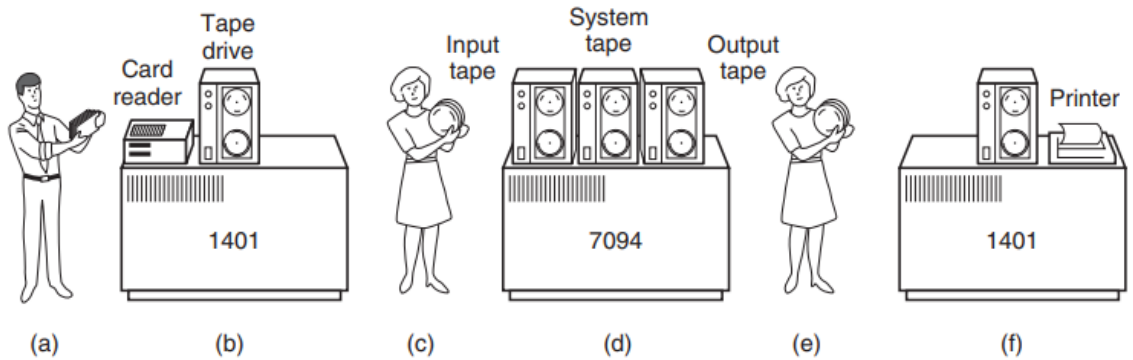


Figure 1-3. An early batch system. (a) Programmers bring cards to 1401. (b) 1401 reads batch of jobs onto tape. (c) Operator carries input tape to 7094. (d) 7094 does computing. (e) Operator carries output tape to 1401. (f) 1401 prints output.

Tanenbaum 4th ed.

Processes
Definition

- Attributes of a Process:
  - A process has **control structures** associated with it, state (may be **active)**, and may have **resources** assigned to it (e.g. I/O devices, memory, processor) .
  - It has a program, input, and output.
- Example:
  - Program: The Notepad application on your disk (notepad.exe), which is static and not running.
  - Process: The active instance of Notepad that appears when you open the application, with memory, CPU time, and other resources assigned to it. It can interact with your operating system and other resources (e.g., files you open or save, text input from your keyboard).

Processes

Process memory image

- The **process memory image** is the **actual content in memory** that the process uses when it is running.
- It consists of different segments that store code, data, and other resources the process needs during its execution.
  - A **text** segment: Contains **the executable code** of the program. It holds the compiled instructions that the CPU executes.
  - A **static/data** segment: **Stores global variables, constants, and static variables** that the program uses.

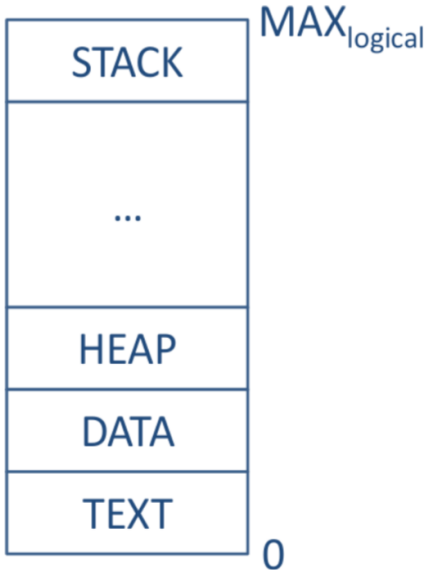


Figure: Representation of a process in memory

Processes

Process memory image

- A **heap** segment: used for dynamic memory allocation. Memory in the heap is allocated and freed as needed during the execution of the program, typically using functions like malloc or new.
  - E.g. If your program dynamically allocates an array of integers using int* arr = malloc(100 * sizeof(int));, this memory is allocated from the heap.
- A **stack** segment: The stack segment is used to manage function calls, local variables, and control flow within the process. It grows and shrinks as functions are called and return. The stack also holds function parameters, return addresses, and local variables.
  - E.g. When a function is called, its local variables and the return address are pushed onto the stack. When the function returns, these are popped off the stack.

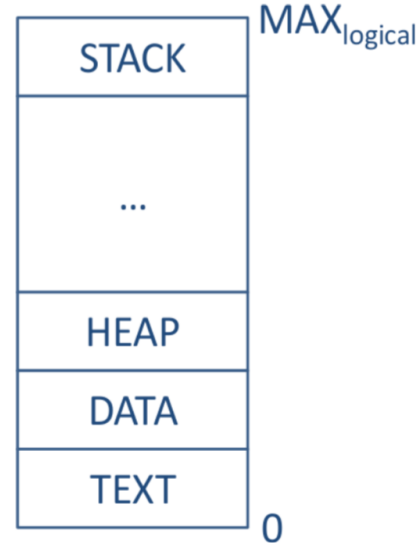


Figure: Representation of a process in memory

Process
Process memory image: example

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

// Global variable (initialized)
int global_var = 100;

// Uninitialized global variable (BSS)
int uninitialized_global_var;

void my_function() {
    // Local variable (stack)
    int local_var = 10;

    // Static local variable (data segment - initialized static)
    static int static_var = 20;

    // Dynamically allocated memory (heap)
    int *dynamic_var = (int *)malloc(sizeof(int));
    *dynamic_var = 30;

    // Printing the addresses of different variables
    printf("Address of local_var (stack): %p\n", (void *)&local_var);
    printf("Address of static_var (static - data segment): %p\n", (void *)&static_var);
    printf("Address of dynamic_var (heap): %p\n", (void *)dynamic_var);

    // Free the dynamically allocated memory
    free(dynamic_var);
}

int main() {
    // Call the function to demonstrate memory usage
    my_function();

    // Print addresses of global variables
    printf("Address of global_var (data segment): %p\n", (void *)&global_var);
    printf("Address of uninitialized_global_var (BSS): %p\n", (void *)&uninitialized_global_var);

    return 0;
}
```

main(), my_function(), malloc(), printf()	Code
Global_var	Data
uninitialized_global_var	
static_var	
dynamic_var (value)	Heap
local_var	Stack

```
Address of local_var (stack): 0x7ffee6a59b4c
Address of static_var (static - data segment): 0x561f734da028
Address of dynamic_var (heap): 0x561f734da3a0
Address of global_var (data segment): 0x561f734da010
Address of uninitialized_global_var (BSS): 0x561f734da014
```

Process States and Transitions

State

- 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:
  - 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.

Note: These names are arbitrary, and they vary across operating systems.

Process States and Transitions

Transitions

- State transitions include:
  - **New** → **ready**: admit the process and commit to execution
  - **Ready** → **running**: the process is selected by the **process scheduler**
  - **Running** → **waiting**: e.g. process is waiting for input or carried out a system call

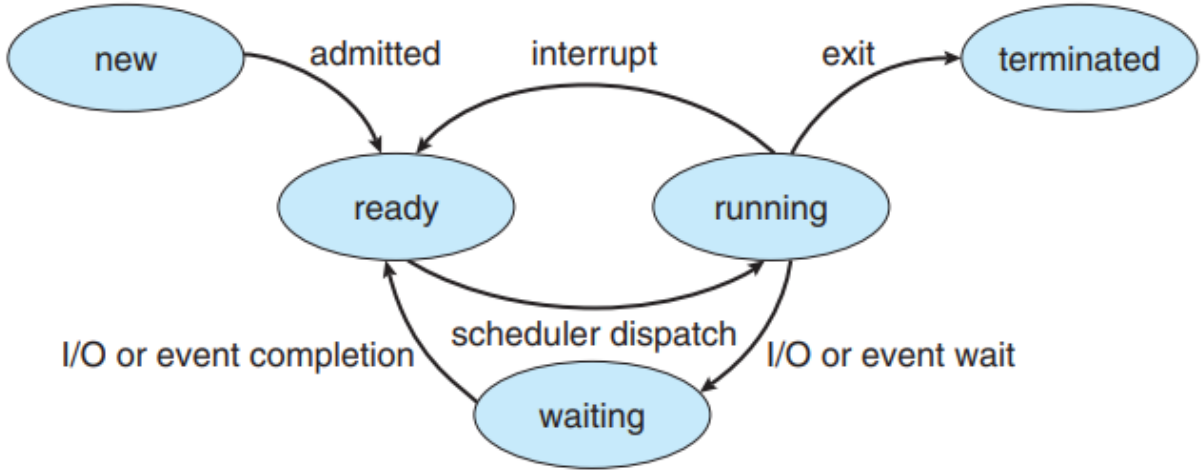


Figure: Diagram of process state.

- **waiting** → **ready**: event happens, e.g. I/O operation has finished.
- **Running** → **ready**: the process is preempted, e.g., by a **timer interrupt.**
- **Running** → **exit**: process has finished, e.g. program ended or exception encountered.

Process States and Transitions (con't)

Transitions

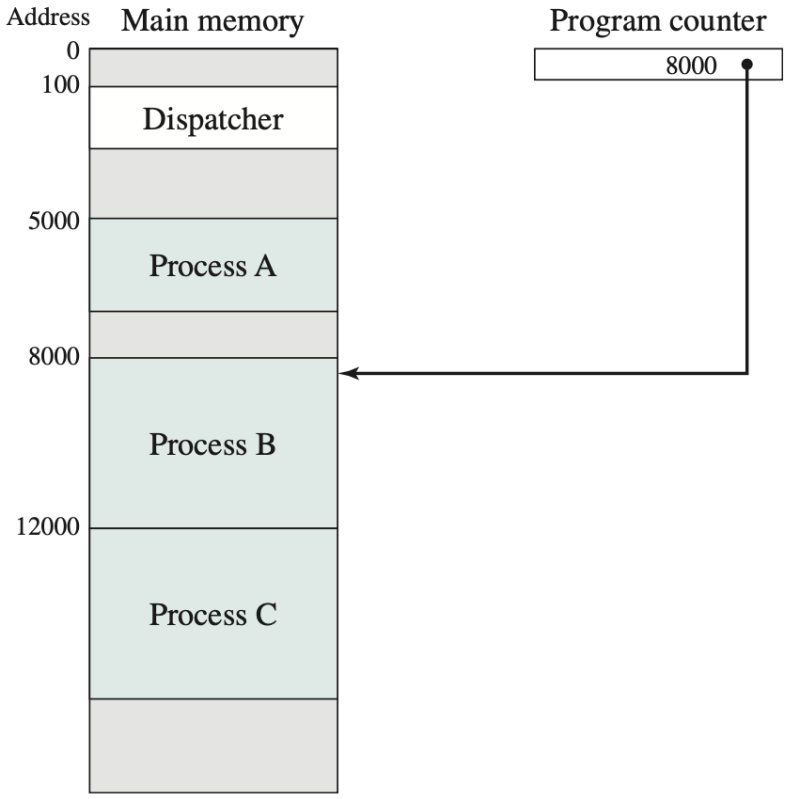


Figure. Memory Example of multi-program execution

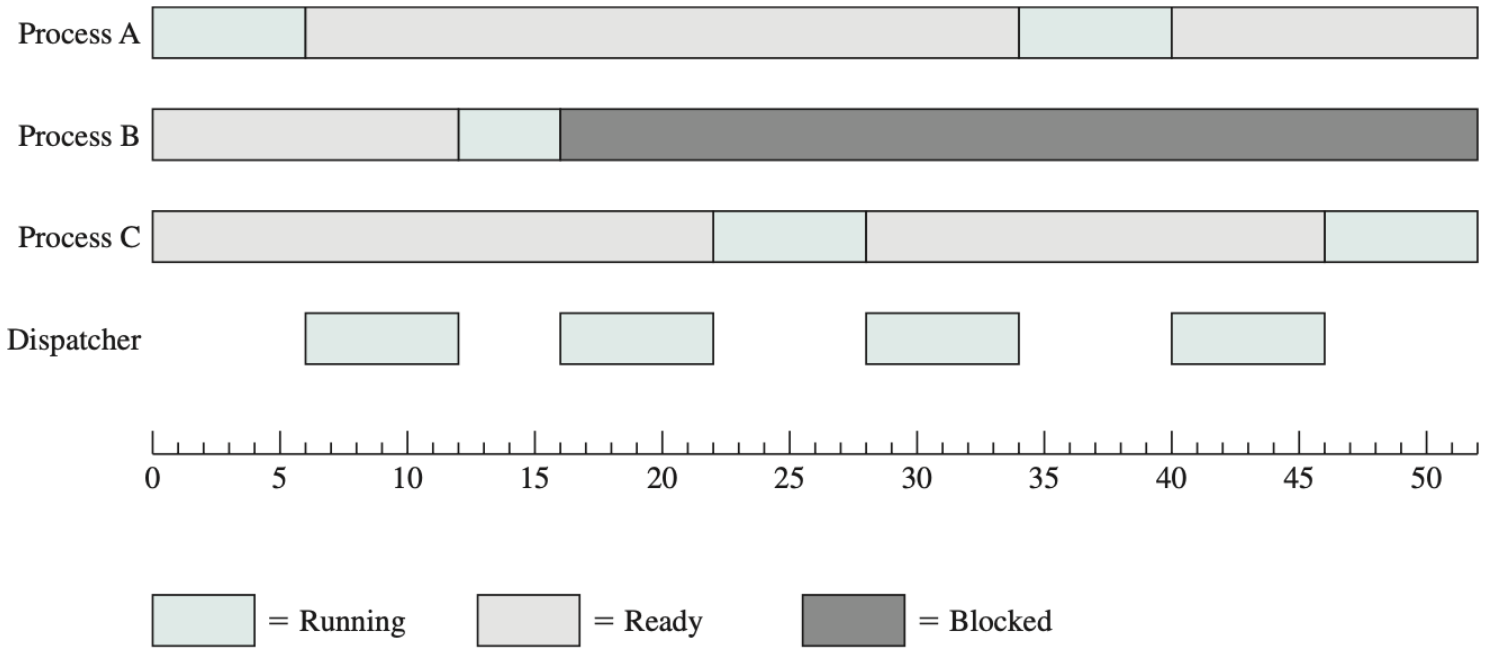


Figure. Process states of multi-program execution

Processes Description

Operating Control Structures

- Operating control structures in an operating system (OS) are data structures and mechanisms that the OS uses to manage and control the execution of processes, resource allocation, and overall system operations.
- Each process, during its **execution**, needs access to certain system **resources.**
  - P1: running (occupying processor)
  - P2: blocked (waiting for I/O)
  - Pn: swapped out (not in main memory)

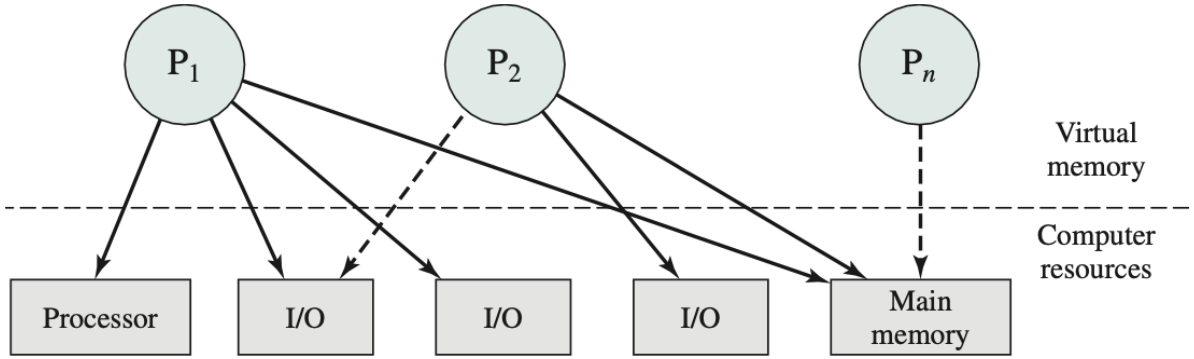


Figure. Processes interacting with resources: Solid lines indicate active utilization of resources, while dashed lines indicate processes that are either utilizing virtual memory or are in a blocked/waiting state.

Processes Description

Operating Control Structures

- An operating system **maintains information** about the status of "resources" in **tables**
  - **Process tables** (process control blocks): is collection of **Process Control Blocks (PCBs**), where each PCB stores detailed information about an individual process.
  - **Memory tables:** keep track of memory allocation, protection, and management of virtual memory.
  - **I/O tables:** keep track of the status of input/output devices, their availability, and ongoing I/O operations.
  - **File tables:** maintain information about files that are currently open, their locations, and their status.

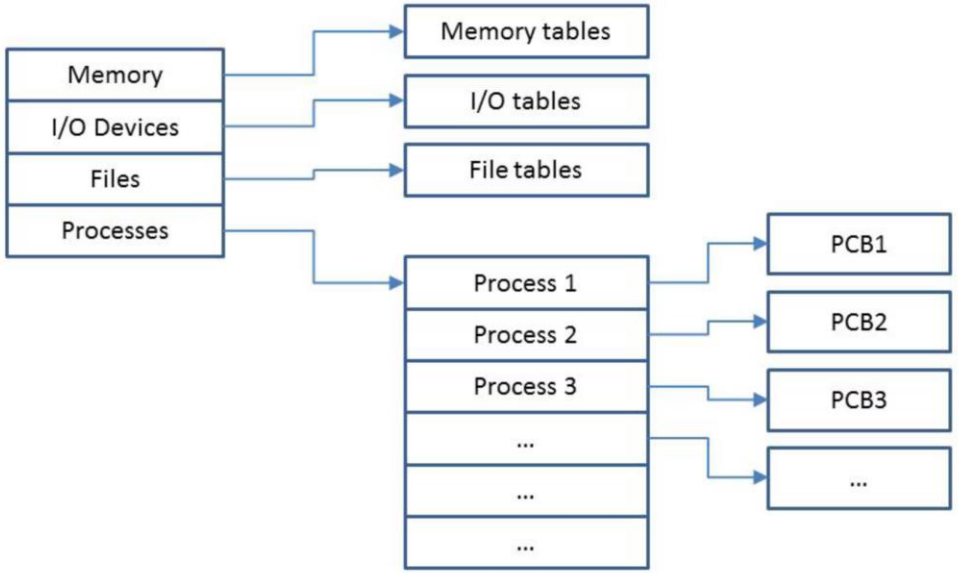


Figure. General structure of OS control tables

Processes Description
Process Control Block

- The OS uses **process control blocks** and a **process table** to manage processes and maintain their information.
- **Process control blocks** are **kernel data structures**, i.e. they are **protected** and only accessible in **kernel mode**!
  - The **operating system manages** them on the user's behalf through **system calls**
- A **process control block (PCB)** contains three types of **attributes**: Process identification, Process state information & Process control information.

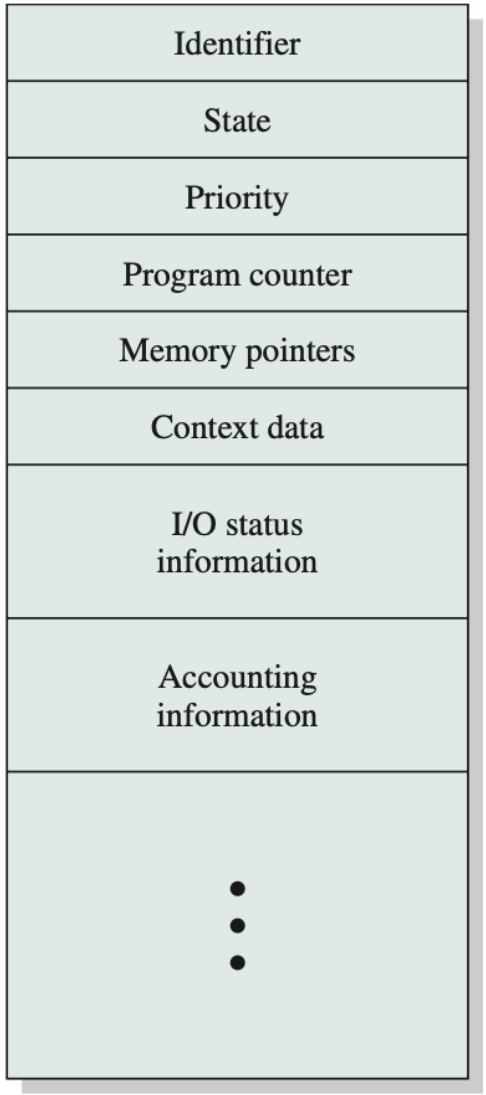


Figure. Process Control Block

Processes Description

Process Control Block

- **Process identification:** This part of the PCB contains identifiers used to uniquely identify the process and manage its relationships with other processes.
  - Process ID (PID):
    - A unique identifier assigned to each process.
    - It helps the OS distinguish between different processes.
  - User ID (UID):
    - The identifier of the user who owns the process.
    - This is important for determining the process's access rights and privileges.
  - Parent Process ID (PPID):
    - The identifier of the parent process, which is the process that created (or spawned) this process.
    - This helps in managing process hierarchies.

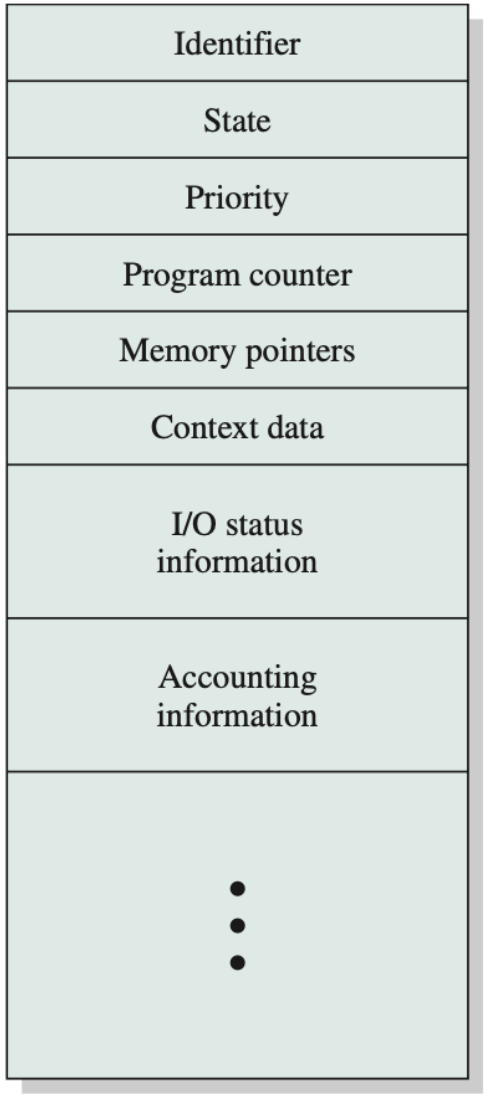


Figure. Process Control Block

Processes Description

Process Control Block

- **Process state information:** Contains information about the current status or condition of the process. It helps the OS manage the process's execution.
  - Process State: This represents the current state of the process (e.g., ready, running, waiting, blocked, or terminated).
  - Program Counter (PC):Holds the address of the next instruction to be executed. It helps the OS resume execution after a context switch.
  - CPU Registers: Contains the values of CPU registers, such as accumulators, index registers, stack pointers, and general-purpose registers, at the moment of the context switch. This allows the process to continue execution from the exact point where it was stopped.

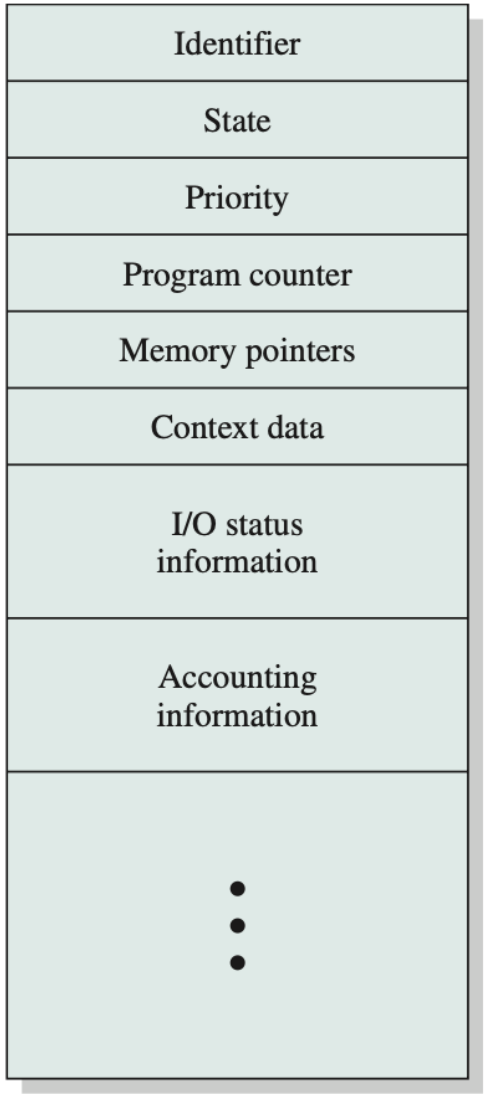


Figure. Process Control Block

Processes Description
Process Control Block

- **Process control information:** This section contains additional information the OS needs to control the process and manage its execution.
  - Scheduling Information: Includes priority level, scheduling queue pointers, and other details necessary for CPU scheduling algorithms (e.g., round-robin, priority scheduling).
  - Memory Management Information: This section includes details about the process's memory allocation, such as:
    - Base and limit registers: Define the address space the process is allowed to use.
    - Page tables: If the system uses virtual memory, this maps logical addresses to physical memory addresses.
    - Segment tables: In segmented memory systems, it defines the segments of the process.

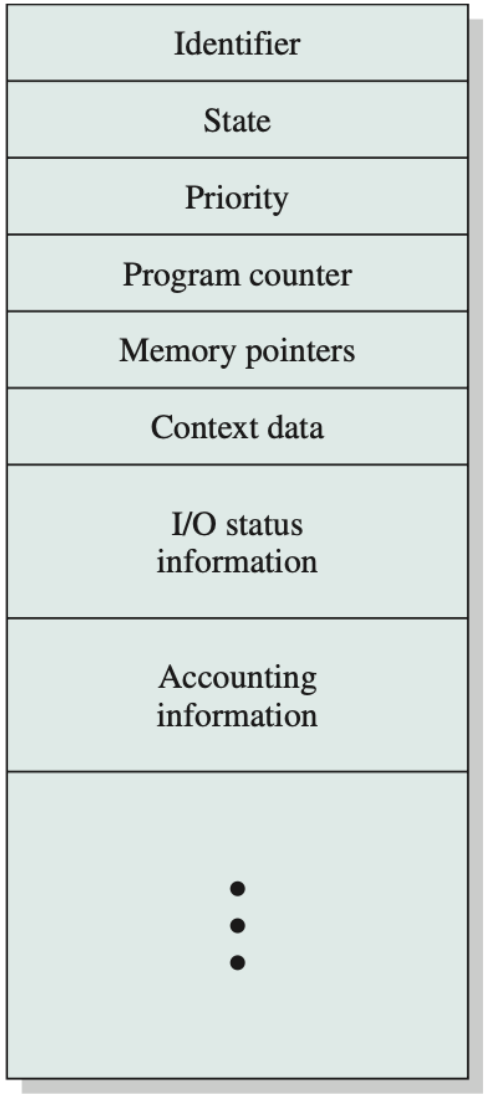


Figure. Process Control Block

Processes Description

Process Control Block

- **Process control information**
  - I/O Status Information: Includes a list of open files, I/O devices assigned to the process, and pointers to I/O buffers.
  - Accounting Information: This section tracks resource usage by the process (e.g., CPU time, memory usage, I/O operations) for performance evaluation or billing purposes.
  - Process Privileges: Defines the rights and privileges of the process, such as what system resources it can access and what operations it is allowed to perform (e.g., reading files, accessing I/O devices).
  - Signals and Flags: Indicates signals or flags that the process has received. This can include interrupts, kill signals, or system alarms.

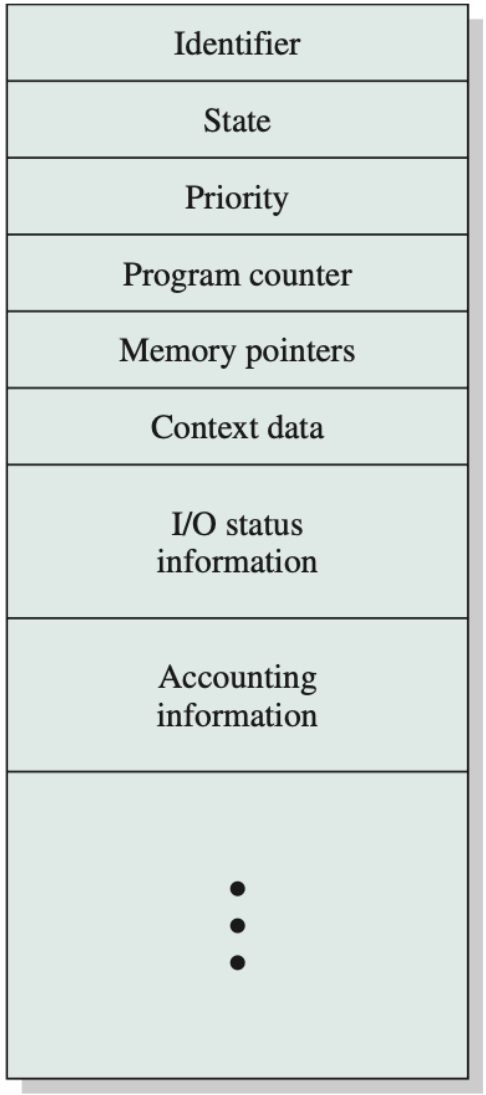


Figure. Process Control Block

Process Management
OS responsibility

- The Operating system is responsible for the following activities in connection with **process management**:
  - Management of **process control blocks**
  - Process **creation** and **termination**
  - Process **switching**
  - Process **scheduling** and **dispatching** (lecture 4)
  - Process **synchronization** and support for interprocess **communication** (concurrency lectures)

Process Management

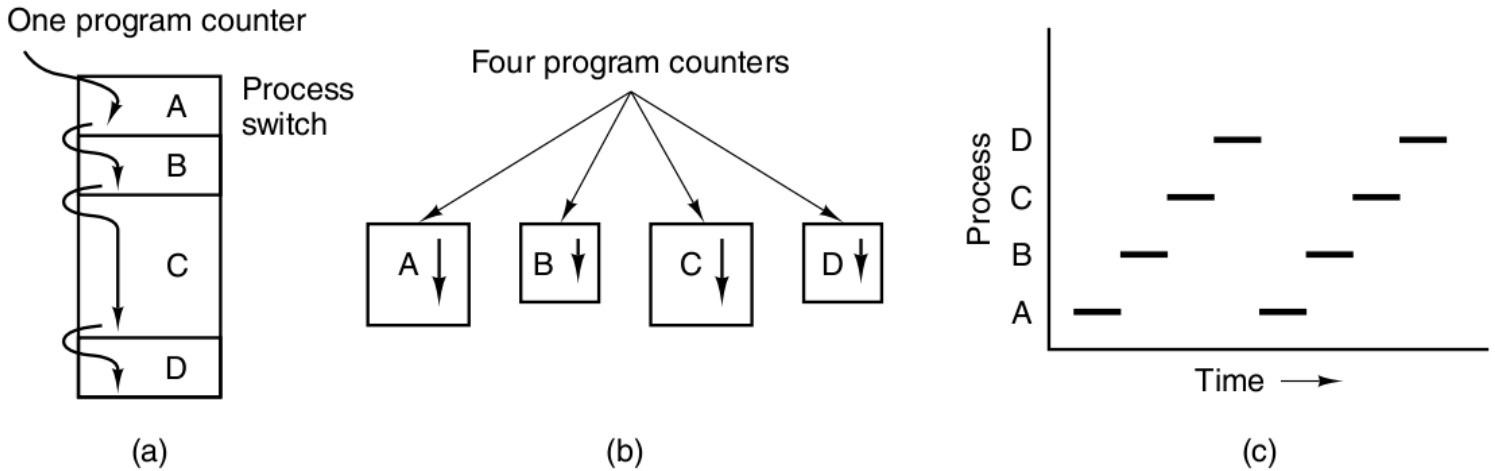
Process Creation and Termination

- Process **creation** conditions:
  - System Initialization: When the operating system starts up.
  - Process Creation System Call: A running process requests to create a new process (e.g., using fork).
  - User Request: A user manually requests to create a new process, often through a command in a shell.
    - E.g. In a command-line interface (CLI), a user might type python myscript.py, which creates a new process to run the Python script.
  - Batch Job Initiation: When a batch job starts execution.

Process Management
Process Creation and Termination

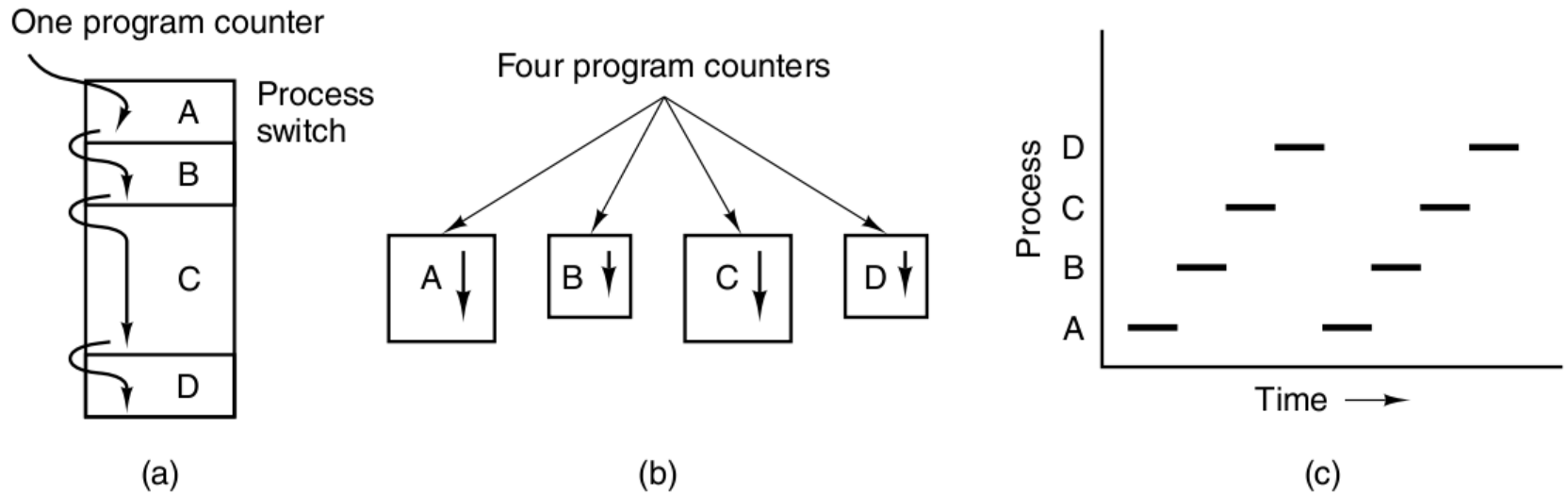
- Process **termination** conditions:
  - Normal Exit (Voluntary): The process ends normally using a system call like exit.
  - Error Exit (Voluntary): The process ends due to an error, such as a program bug or an attempt to access nonexistent memory.
  - Fatal Error (Involuntary): The process is terminated due to a serious error, like referencing a non-existent file or passing bad arguments.
  - Killed by Another Process (Involuntary): The process is forcibly terminated by another process using a system call like kill.

Process Management
Process Switching – What?



- Modern OS are **multiprogramming** systems (why?)
  - Assuming a **single processor system**, the instructions of individual processes are executed **sequentially**
    - CPU's rapid **switching back and forth** from process to process is called **multiprogramming**.
    - Multiprogramming is achieved by **alternating** processes and **context switching**
    - Each process has its own **flow of control** (i.e., its own logical program counter)
- **Efficient Utilization of CPU:** Multiprogramming allows multiple processes to reside in memory at the same time, maximizing CPU utilization by quickly switching between processes. This ensures that the CPU is not idle while waiting for I/O operations to complete.

Process Management
Process Switching – What?



- **True parallelism** requires **multiple processors:**
  - For true parallelism, where processes actually run simultaneously, multiple processors (or cores) are required.
  - Multiprogramming in a single processor system only simulates parallelism by quickly switching between processes.

Process Management
Process Switching – How?

- **When** will the context switch take place? (interrupt)

- Context switch happens, the system **saves** the state of the old process and **loads** the state of the new process (creates **overhead**)
  - **Saved** ⇒ the process control block is **updated**
  - **(Re-)started** ⇒ the process control block **read**

- What happens when a context switch takes place?
  - 1. Save process state (program counter, registers)
  - 2. Update PCB (running -> ready)
  - 3. Move PCB to appropriate queue (ready/blocked)
  - 4. Run scheduler, select new process
  - 5. Update to running state in PCB
  - 6. Update memory structures
  - 7. Restore process

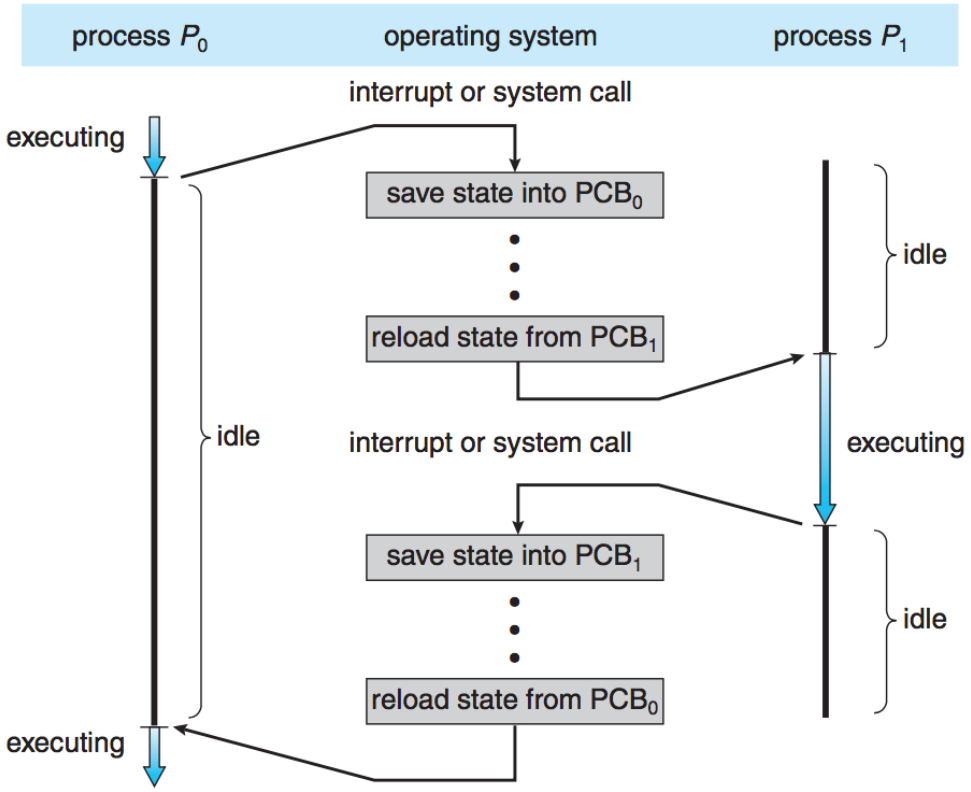
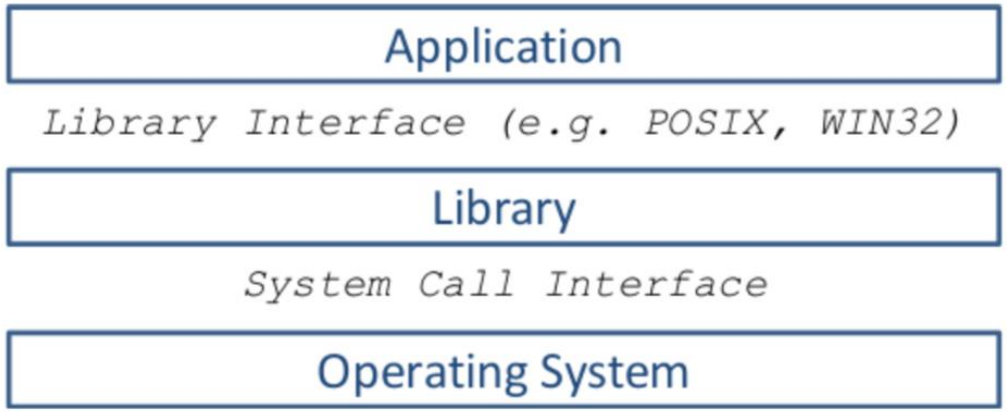


Figure: CPU switch from process to process.

System Calls
Libraries

- A **system call** allows a user process to access and execute operating system **functions inside the kernel**. User programs use system calls to **invoke** operating system services.
- The true system calls are **"wrapped"** in the **OS libraries** (e.g. libc) following a well defined interface (e.g. Portable Operating System Interface (POSIX), WIN32 API)

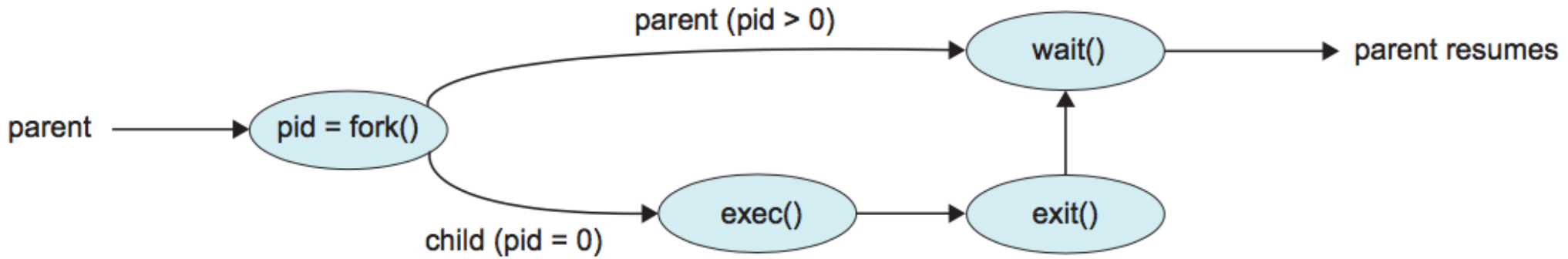


System Calls
Process Creation and Termination

- System calls for **process creation**:
  - Unix:
    - **fork()** generates an exact copy of parent
    - **execve():** This system call replaces the current process image with a new process image (typically a new program)
  - Windows: **NTCreateProcess()**
- System calls are necessary to **notify the OS** that the **process has terminated**
  - Resources must be de-allocated
  - Output must be flushed
  - Process admin may have to be carried out
- A system calls for process termination:
  - UNIX/Linux: **exit()**, kill()
  - Windows: **TerminateProcess()**

System Calls
Fork

- Fork() creates an **exact copy** of the current process's **memory image**
- Fork() returns the **process identifier** of the child process to the **parent process**
- Fork() **return 0** to the **child process**



Often, fork() is followed by one of the exec() family of system calls. The exec() system calls replace the process's memory with a new program, allowing the child process to run a different program than the parent.

```
while (TRUE) {                                  /* repeat forever /*/
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, params);          /* read input line from keyboard */

        pid = fork( );                          /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0)      /* child does the work */
        }
}
```

Figure: Use of the fork() and exec() system calls (Tanenbaum)

Example 1: C code, fork()

```c
#include <stdio.h>
#include <unistd.h>   // For fork(), getpid()
#include <sys/types.h> // For pid_t

int main() {
    // Create a child process
    pid_t pid = fork();

    if (pid == 0) {
        // This is the child process
        printf("This is the child process. Child PID: %d\n", getpid());
    } else if (pid > 0) {
        // This is the parent process
        printf("This is the parent process. Parent PID: %d\n", getpid());
        printf("Child PID from parent perspective: %d\n", pid);
    } else {
        // Fork failed
        printf("Fork failed.\n");
    }

    return 0;
}
```

```
This is the parent process. Parent PID: 12345
Child PID from parent perspective: 12346
This is the child process. Child PID: 12346
```

University of Nottingham
UK | CHINA | MALAYSIA

Example 2: C Code, execve()

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   // For getpid(), execve()
#include <sys/types.h>

int main() {
    // Get the process ID of the current process
    pid_t pid = getpid();
    printf("Current Process ID: %d\n", pid);

    // The new program to execute: "/bin/ls"
    char *program = "/bin/ls";

    // Arguments for execve (first argument should be the program name itself)
    char *args[] = { "/bin/ls", "-l", NULL };

    // Environment variables (passing the current environment variables)
    extern char **environ;

    // Replace the current process with "ls -l" command
    if (execve(program, args, environ) == -1) {
        perror("execve failed");  // If execve fails, print an error
        exit(EXIT_FAILURE);
    }

    // This line will never be reached if execve is successful
    printf("This line will not be printed.\n");

    return 0;
}
```

```
Current Process ID: 12345
total 8
-rwxr-xr-x 1 user user  1234 Oct  3 12:34 execve_example
-rw-r--r-- 1 user user   567 Oct  3 12:33 execve_example.c
```

Recap
Take-home Message

- **Definition of a process** and their **implementation** in operating systems
- **States**, state **transitions** of processes
- **Kernel structures** for processes and process management
- **System calls** for process management