Let's start with the basics of an **Operating System (OS)** in a very simple way:

What is an Operating System?

Imagine your computer is a factory. The **Operating System** is like the manager of the factory, organizing and coordinating all activities. It helps run programs (like games or browsers), manage resources (like memory and CPU), and lets you interact with the computer.

Without an OS, you wouldn't be able to use the computer effectively because it's the one handling all the complex stuff behind the scenes, like when you open a file, run a program, or save something.

Key Functions of an Operating System:

1. **Process Management**: The OS decides which programs (processes) run and when. Imagine you are running a game and a music player at the same time—the OS manages how these programs share the computer's resources (like the CPU).

2. **Memory Management**: It keeps track of how much memory each program is using and allocates memory when needed. For example, if you're running multiple apps, the OS ensures they all get the memory they need.

3. **File Management**: It helps you create, save, and manage files on your computer. If you open a document or save a photo, the OS organizes where these files are stored.

4. **Device Management**: The OS helps your computer communicate with hardware like your keyboard, mouse, printer, and monitor.

5. **Security**: It keeps your computer safe by managing access to files and programs. The OS prevents unauthorized access and manages user accounts.

Now, let's break down each of your codes in simple words:

1. **Process Scheduling (First Code)**

This code deals with **process scheduling**, which is how the OS decides which task (or process) gets to use the CPU and for how long.

It defines a structure `Process` which has:

- **pid**: Process ID, to identify the process.

- **bt**: Burst time, how long the process takes to complete.

- **art**: Arrival time, when the process starts.

The functions calculate:

- **Waiting Time**: How long a process waits before it gets executed.

- **Turnaround Time**: Total time taken by a process from its arrival to its completion.

Example:

- If you are cooking, and two dishes need the stove (CPU), the OS decides which dish (process) gets the stove first, for how long (burst time), and when it can start (arrival time).

The code simulates the Shortest Remaining Time First (SRTF) scheduling, where the process with the shortest remaining time is selected next.

---

2. **Fork and Process IDs (Second Set of Codes)**

These codes demonstrate how **forking** works in an operating system.

- **Fork** is used to create a new process. A process is a running program, and when a process is created, it gets a unique **Process ID (PID)**.

Example:

Imagine you are copying a file, and while copying, you open a game. The OS will use a fork to create a new "child" process for the game while the file copy "parent" process continues.

Code Breakdown:

- `fork()` creates a new process. If the return value is **0**, it's the child process, and if it's positive, it's the parent process. Each process has its own unique **Process ID (PID)**.

Example for Code 1:

- The parent says: "My process ID is 2001, and I just created a child process with ID 3001."

- The child process says: "I am the child process, and my parent's ID is 2001."

---

3. **Process and Command Execution (Third Set of Codes)**

In these codes, you are learning about how processes can run **commands** on the system.

- **exec()** functions replace the current process with a new program. This is useful when a process starts a new task.

Example:

You have a music player open, and it suddenly opens your music library (list of songs). The music player replaces itself with the library program.

Code Breakdown:

- `execlp("/bin/ls", "ls", NULL);` runs the **ls** command, which lists files in a directory. It's like saying, "Hey OS, I want to stop this program and start listing files instead."

---

4. **File Handling (Fourth Set of Codes)**

This code shows how the OS handles **files**. It opens, reads, writes, and closes files using system calls like:

- `open()` to open a file.
- `write()` to write data into a file.
- `read()` to read data from a file.
- `close()` to close the file.

Example:

Imagine writing a note in a notebook (file). You open the notebook, write the note, then close it.

Code Breakdown:

- It opens two files, writes details into them, and checks if the city "New York" is mentioned in both files.

---

5. **Scheduling Algorithms (Fifth Code)**

Here you are seeing different **CPU Scheduling Algorithms** that decide which process gets the CPU and for how long:

1. **FCFS (First Come First Serve)**: Processes are executed in the order they arrive.

   - **Example**: If you stand in a line at a food stall, the first person gets served first.

2. **SJF (Shortest Job First)**: The process with the shortest burst time (smallest job) gets executed first.

   - **Example**: If you have a list of tasks, and you do the smallest, quickest ones first.

3. **SJF Preemptive (Shortest Job First with Preemption)**: If a shorter job arrives while another is running, the OS interrupts the current job and switches to the shorter one.

   - **Example**: You start cooking, but then realize there's a quicker task, like boiling water, which you can finish first before continuing.

Code Breakdown:

- The code simulates all three scheduling algorithms and prints the **Gantt Chart**, which is a graphical representation of the order in which processes run.

I hope this gives you a good foundation. Let me know if you have any more questions or need more details!

Scheduling Algorithms: Detailed Explanation

CPU Scheduling Algorithms are strategies used by the operating system to determine the order in which processes (programs) will be executed by the CPU. These decisions impact how efficiently the CPU is utilized and how fast programs run.

Let's dive deeper into **FCFS (First Come First Serve)** and **SJF (Shortest Job First)** and understand how they work along with how the code simulates these algorithms.

---

1. **FCFS (First Come First Serve)**

Concept:

- **FCFS** is one of the simplest CPU scheduling algorithms.

- It works like a queue system: the process that arrives first gets executed first.

- No preemption occurs, meaning once a process starts executing, it runs until completion, even if other processes arrive during its execution.

Example (Real-Life Scenario):

Imagine a line at a food stall. Whoever stands first in line gets served first, regardless of how long it will take to serve them. If someone behind in the line has a simpler or faster order, they still have to wait until the person in front is served.

Code Behavior (FCFS):

- **Arrival Time**: When a process arrives in the queue.

- **Burst Time**: The time required by the process to complete its execution (the time it needs the CPU for).

The FCFS algorithm works by:

1. Sorting processes by their **arrival time**.

2. Running the first process that arrives until it finishes.

3. Picking the next process in line and running it, repeating this until all processes are done.

Steps:

1. Sort processes based on **arrival time**.

2. For each process:

  - Start the process when the CPU is free.

  - Calculate the **Waiting Time** (how long a process waited before starting) and **Turnaround Time** (total time from process arrival to completion).

Example of FCFS:

| Process | Arrival Time | Burst Time | Start Time | End Time | Waiting Time | Turnaround Time |

|---------|--------------|------------|------------|----------|--------------|-----------------|

| P1 | 0 | 5 | 0 | 5 | 0 | 5 |

| P2 | 1 | 3 | 5 | 8 | 4 | 7 |

| P3 | 2 | 1 | 8 | 9 | 6 | 7 |

- **Waiting Time** = Start Time - Arrival Time.

- **Turnaround Time** = End Time - Arrival Time.

---

2. **SJF (Shortest Job First)**

Concept:

- **SJF** schedules processes in the order of their **burst time**, meaning the process with the **shortest burst time** is executed first.

- There are two types:

  1. **Non-preemptive SJF**: Once a process starts, it cannot be interrupted, and it runs to completion (similar to FCFS but with a different order).

  2. **Preemptive SJF** (also called **Shortest Remaining Time First, SRTF**): If a new process arrives with a shorter burst time than the current running process, the current process is paused, and the shorter one runs.

Example (Real-Life Scenario):

At the same food stall, instead of serving customers based on their arrival time, the stall owner serves the person with the **quickest order** first. If someone arrives with a simpler and faster order, they get served before a person who has a more complicated order, regardless of when they arrived.

Code Behavior (SJF):

- **Burst Time**: The main factor determining which process to run first.

- The process with the shortest burst time is selected, regardless of its **arrival time**.

Steps:

1. When a new process arrives, check the burst time of all the processes waiting in the queue.

2. Select the process with the **shortest burst time**.

3. Calculate the **Waiting Time** and **Turnaround Time** similarly as in FCFS.

Example of SJF (Non-Preemptive):

| Process | Arrival Time | Burst Time | Start Time | End Time | Waiting Time | Turnaround Time |

|---------|--------------|------------|------------|----------|--------------|----------------|

| P1 | 0 | 5 | 0 | 5 | 0 | 5 |

| P2 | 1 | 3 | 5 | 8 | 4 | 7 |

| P3 | 2 | 1 | 8 | 9 | 6 | 7 |

Notice in this case, the processes are still executed in FCFS order because P1 has the shortest burst time. However, with more varied burst times, the execution order would change.

How the Code Works for FCFS and SJF:

Here's a general breakdown of how a code simulates both FCFS and SJF scheduling algorithms.

1. **FCFS Scheduling Code**:

```c
#include<stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int at[]) {
   int service_time[n];
   service_time[0] = at[0];
   wt[0] = 0;

   for (int i = 1; i < n ; i++) {
```

```
        service_time[i] = service_time[i-1] + bt[i-1];
        wt[i] = service_time[i] - at[i];
        if (wt[i] < 0) {
            wt[i] = 0;
        }
    }
}

void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
    for (int i = 0; i < n ; i++) {
        tat[i] = bt[i] + wt[i];
    }
}

void findavgTime(int processes[], int n, int bt[], int at[]) {
    int wt[n], tat[n];
    findWaitingTime(processes, n, bt, wt, at);
    findTurnAroundTime(processes, n, bt, wt, tat);

    printf("Processes   Burst Time   Waiting Time   Turn-Around Time\n");
    for (int i = 0; i < n; i++) {
        printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
    }
}
```

Key Parts of the Code:

- **findWaitingTime()**: This function calculates how long each process has to wait before starting execution. It calculates waiting time based on the arrival time and the burst time of previous processes.

- **findTurnAroundTime()**: This function calculates the **Turnaround Time**, which is the total time a process takes from arriving to finishing execution.

- **findavgTime()**: This function handles the overall scheduling, calling the other two functions and printing out the results for each process.

---

2. **SJF Scheduling Code (Non-Preemptive)**:

```c
#include<stdio.h>

void findWaitingTime(int processes[], int n, int bt[], int wt[], int at[]) {
   int rt[n];
   for (int i = 0; i < n; i++)
      rt[i] = bt[i];

   int complete = 0, t = 0, minm = INT_MAX;
   int shortest = 0, finish_time;
   int check = 0;

   while (complete != n) {
      for (int j = 0; j < n; j++) {
         if ((at[j] <= t) && (rt[j] < minm) && rt[j] > 0) {
```

```
            minm = rt[j];
            shortest = j;
            check = 1;
        }
    }

    if (check == 0) {
        t++;
        continue;
    }

    rt[shortest]--;
    minm = rt[shortest];
    if (minm == 0)
        minm = INT_MAX;

    if (rt[shortest] == 0) {
        complete++;
        finish_time = t + 1;
        wt[shortest] = finish_time - bt[shortest] - at[shortest];

        if (wt[shortest] < 0)
            wt[shortest] = 0;
    }
    t++;
  }
}
```

```
void findTurnAroundTime(int processes[], int n, int bt[], int wt[], int tat[]) {
   for (int i = 0; i < n ; i++) {
      tat[i] = bt[i] + wt[i];
   }
}

void findavgTime(int processes[], int n, int bt[], int at[]) {
   int wt[n], tat[n];
   findWaitingTime(processes, n, bt, wt, at);
   findTurnAroundTime(processes, n, bt, wt, tat);

   printf("Processes   Burst Time   Waiting Time   Turn-Around Time\n");
   for (int i = 0; i < n; i++) {
      printf("%d\t\t%d\t\t%d\t\t%d\n", processes[i], bt[i], wt[i], tat[i]);
   }
}
```

Key Parts of the Code:

- This version of SJF keeps track of the **remaining time** for each process and checks if a shorter job arrives while the CPU is busy. It then switches to the process with the shortest time remaining.

- **findWaitingTime()** in SJF finds the process with the smallest burst time at each time step and reduces its remaining time. Once it's finished, the function moves on to the next shortest process.