Processes in Linux

What is a Process?

- A process is a **program in execution**.
- It is a **dynamic entity** managed by the operating system.

Components of a Process

A process consists of:

- **Program Counter (PC)**: Tracks the address of the next instruction.
- **CPU Registers**: Includes stack pointer, general-purpose registers, etc.
- **Process Stack**: Stores function parameters, local variables, and return addresses.
- **Memory Regions**: Text, stack, and heap.

Process Memory Layout

Every process in Linux has 3 main memory regions:

1. **Text Region**:
    - Contains the **program code**.
    - **Static** in size.
2. **Stack Region**:
    - Stores local variables, function parameters, and return addresses.
    - Grows **downward**.
3. **Data Region (Heap)**:
    - Stores **program data** that grows dynamically as per user requests.
    - Grows **upward**.

Diagram:

```
mathematica

0x0000
 |   Text Region
 |   Data (Heap)
 |   Unused Free Space
 |   Stack Region
0xffff
```

User Stack vs Kernel Stack

1. **User Stack**:
    - Resides in **user space**.
    - Manipulated by the **process itself**.

2. **Kernel Stack**:
    - Resides in **kernel space**.
    - Used by the OS to handle **system calls** and **interrupts**.
    - Manipulated by the **OS**.

Process Descriptor (`task_struct`)

- A **data structure** that holds all information about a process.
- Contains:
    - **State**: Current state of the process.
    - **PID (Process ID)**: Unique ID for each process.
    - **Program Counter**: Tracks the execution point.
    - **Registers**: CPU state information.
    - **Memory Regions**: Text, stack, heap.
    - **Open Files**: List of open files for the process.
    - **Pointers**: To manage process queues.

Foreground vs Background Processes

1. **Foreground Process**:
    - Requires **user input** to run.
    - Example: Text editors like `vim`.
2. **Background Process**:
    - Runs **continuously** without requiring user input.
    - Example: Download commands like `wget`.

Move Process to Background:

- Add & at the end of the command:

  bash

  ```
  wget http://example.com/file.iso &
  ```

Bring Back to Foreground:

- Use `fg` command:

  bash

  ```
  jobs          # Check background jobs
  fg [job_number]
  ```

Process States

1. **New**: Process is created.
2. **Ready**: Process is ready to execute.
3. **Running**: Process is currently being executed.
4. **Waiting**: Process is waiting for an event/resource.
5. **Terminated (Zombie)**: Process is halted, but its descriptor exists.

Process Management Commands

1. **View Processes**:
   - `ps -ef`: Displays all processes.
   - `top`: Dynamic display of processes.
2. **Kill a Process**:
   - **Find PID**: Use `ps` or `top`.
   - **Kill Command**:

     ```
     bash

     kill [PID]         # Graceful termination
     kill -9 [PID]      # Force termination (SIGKILL)
     ```
3. **Kill Multiple Processes**:
   - Use `killall` to terminate multiple processes:

     ```
     bash

     killall [process_name]
     ```
   - Use `pkill` for specific users:

     ```
     bash

     pkill -u [username]
     ```

Process Queues

- Processes are organized in **queues**:
  - **Ready Queue**: Jobs ready to execute.
  - **Waiting Queue**: Jobs waiting for an event (I/O, signals).

Example:

```
less

Ready Queue: PCB A → PCB B
Printer Queue: PCB X → PCB Y
```

Process State Transitions

- A process changes states based on events:
  - **Running** → **Waiting**: Process is waiting for I/O or resources.
  - **Ready** → **Running**: Scheduler selects the process to execute.
  - **Zombie**: Process ends, but its parent has not collected the exit code.

Process Creation: Forking

- **Forking** creates a new process in Linux.
- The parent process spawns a child process.

Example Code:

```
c

#include <stdio.h>
#include <unistd.h>

void main() {
    int pid;
    pid = fork();  // Creates child process
    if (pid == 0) {
        printf("Child process\n");
    } else {
        printf("Parent process\n");
    }
}
```

Process Termination

Processes terminate in multiple ways:

1. **exit()**: The process calls exit explicitly.
2. **kill()**: Parent process or user sends a kill signal.
3. **SIGKILL**: Force termination with `kill -9`.

Zombie Processes

- **Definition**: A process that has completed execution but still has an entry in the process table.
- **Why?**: Parent process hasn't collected the child's exit status using `wait()`.
- **Solution**: The `init` process cleans up zombie processes.

---

init Process

- The **first process** started by the OS.

- It **adopts orphan processes** and cleans up zombie processes.

Process Representation: task_struct

In Linux, a process is represented by the **task_struct** structure. It contains:

- **State**: Current status of the process.
- **Scheduling Information**: Priority and scheduling queues.
- **Identifiers**: PID, user ID, group ID.
- **IPC**: Signals, pipes, semaphores.
- **File System**: Open files, root, and working directories.
- **Timers**: Process-specific timers.
- **Virtual Memory**: Memory mappings.

Important Commands Summary

Command	Description
`ps`	Displays processes running on the system.
`top`	Shows dynamic list of processes.
`kill [PID]`	Terminates a specific process.
`killall [name]`	Kills all processes by name.
`fg`	Brings a background process to the foreground.
`jobs`	Lists background jobs.
`pstree`	Displays process hierarchy.

Fork and Zombie Processes in Linux

What is a Process?

- A **process** is a program in **execution**.
- It consists of:
  - **Registers**: CPU state information.
  - **Memory**: Code, data, stack, and heap regions.
  - **Process Context**: Includes open files, stack pointers, etc.

Process Management

1. **Multiprocessing**: Simultaneous execution of multiple processes.
   - On a **single CPU**: The OS rapidly switches between processes using **context switching**.
   - On a **multicore CPU**: Multiple cores can execute different processes in parallel.
2. **Context Switching**:
   - The OS saves the current process state (registers, program counter, etc.) and loads the next process.
   - Provides the **illusion** of parallel execution.

Creating New Processes

1. fork() System Call

- **Definition**: `fork()` creates a **child process** by duplicating the **parent process**.
- The child is an **exact copy** of the parent but has a **different Process ID (PID)**.
- Returns:
  - `0` to the **child process**.
  - Child's PID to the **parent process**.

Example Code:

```
c

#include <stdio.h>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Hello from Child\n");
    } else {
        printf("Hello from Parent\n");
    }

    return 0;
}
```

Output

- "Hello from Parent" and "Hello from Child" can appear in **any order** due to concurrent execution.

---

2. exec() System Call

- **Definition**: `exec()` replaces the **current process's code** and address space with a **new program**.
- `fork()` → creates a process, then `exec()` → loads a new program.

Types of `exec()`:

- `execv`, `execl`, `execve`, `execle`, `execvp`, `execlp`.

Example:

```
c

#include <stdio.h>
#include <unistd.h>

int main() {
    char *args[] = {"/bin/ls", "-l", NULL};
    execv("/bin/ls", args);  // Replaces current process with ls command
    return 0;
}
```

---

3. wait() System Call

- **Definition**: `wait()` makes the parent process **wait** until one of its child processes terminates.
- **Why?**: To **reap** terminated child processes and prevent **zombies**.

Example:

```
c

#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        printf("Child Process\n");
        return 0;
    } else {
        wait(NULL);
        printf("Parent: Child has terminated\n");
    }
    return 0;
}
```

Zombie Processes

What is a Zombie Process?

- A **zombie process** is a process that has **terminated**, but its **process descriptor** (entry in the process table) is still present.
- **Why?**: The parent process hasn't yet **collected the child's exit status** using `wait()` or `waitpid()`.

Why Do Zombies Exist?

- **Exit status** must be passed to the parent process.
- The OS keeps the process descriptor until the parent **reaps** it.

How to Detect Zombies

- Use the `ps` command to view processes. Zombie processes appear as **<defunct>**.

Example:

`bash`

```
ps -ef
```

Output:

`less`

```
PID   TTY    TIME CMD
1234  pts/1  0:00 a.out <defunct>
```

How to Handle Zombies

1. **Reap Zombies Using `wait()`**:
    - The parent process should call `wait()` to clean up the terminated child.
2. **Orphaned Zombies**:
    - If a parent process terminates without reaping its child, the **init** process (PID = 1) adopts the child and cleans it up.
3. **Kill the Parent Process**:
    - Use `kill` to terminate the parent. The zombie will then be reaped by `init`.

Example to Create a Zombie:

```
c

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

int main() {
    if (fork() == 0) {
        printf("Child Process: PID = %d\n", getpid());
        exit(0);  // Child terminates
    } else {
        printf("Parent Process: PID = %d\n", getpid());
        while (1);  // Parent process is stuck in an infinite loop
    }
    return 0;
}
```

Output in `ps`:
The child will appear as **<defunct>** until the parent process terminates or reaps it.

Process Management Summary

1. **fork()**: Creates a new child process.
2. **exec()**: Replaces the current process with a new program.
3. **wait()**: Makes the parent wait for the child to terminate.
4. **Zombie**: Process that has terminated but whose entry remains in the process table.

Key Commands

Command	Description
`ps`	Displays all running processes.
`top`	Monitors processes dynamically.
`kill [PID]`	Terminates a process with the given PID.
`wait`	Makes the parent wait for child termination.
`pstree`	Displays the process hierarchy.
