PUNJAB UNIVERSITY COLLEGE OF INFORMATION AND TECHNOLOGY

PROGRAMMING ASSIGNMENT # 1

Course: Operating Systems

Submitted to: Sir Ahmad

Submitted by: Mariam Qadeem

Roll no: Bitf22m006

QUESTION#1

In operating systems like Linux, the struct task_struct represents a *Process Control Block* (PCB), which contains various details about a process, such as its state, ID, memory usage, and scheduling information. Each member in task_struct is specifically tailored to store an aspect of the process, making it easier to access and manage process information in a structured manner.

Core Members of `task_struct`

1. **pid**
    - **Code Representation**: `pid_t pid`
    - **Description**: The unique identifier assigned to each process.
2. **state**
    - **Code Representation**: `long state`
    - **Description**: Represents the process's current state (e.g., running, sleeping, stopped, zombie).
3. **flags**
    - **Code Representation**: `unsigned int flags`
    - **Description**: Holds flags that provide metadata on the process, including its privileges or special conditions.
4. **parent**
    - **Code Representation**: `struct task_struct *parent`
    - **Description**: A pointer to the `task_struct` of the parent process, enabling hierarchy and tracking.
5. **children**
    - **Code Representation**: `struct list_head children`
    - **Description**: Manages a linked list of child processes, facilitating process tree structure.

6. sibling

- **Code Representation**: `struct list_head sibling`
- **Description**: A linked list node for navigating between sibling processes, representing other children of the same parent.

7. comm

- **Code Representation**: `char comm[TASK_COMM_LEN]`
- **Description**: Contains the name of the process's executable, used in various monitoring tools.

8. mm

- **Code Representation**: `struct mm_struct *mm`
- **Description**: Points to the memory management structure (`mm_struct`), managing the process's memory allocations.

`mm_struct` Members (Referenced by `mm`)

- **pgd**
  - **Code Representation**: `pgd_t *pgd`
  - **Description**: Points to the Page Global Directory, which is essential for managing page tables and virtual memory.
- **mmap**
  - **Code Representation**: `struct vm_area_struct *mmap`
  - **Description**: Pointer to a list of memory regions (`vm_area_struct`) allocated to the process.
- **total_vm**
  - **Code Representation**: `unsigned long total_vm`
  - **Description**: Tracks the total number of virtual memory pages the process has allocated.
- **rss**
  - **Code Representation**: `unsigned long rss`
  - **Description**: The Resident Set Size (RSS), indicating how many physical memory pages are currently held by the process.
- **stack_vm**
  - **Code Representation**: `unsigned long stack_vm`
  - **Description**: Amount of virtual memory allocated specifically for the process's stack.
- **data_vm**
  - **Code Representation**: `unsigned long data_vm`

- **Description**: The virtual memory size of the data section, which includes initialized and uninitialized data segments.

9. cred

- **Code Representation**: `const struct cred *cred`
- **Description**: Points to the credential structure, managing the user and group IDs, and security context.

10. files

- **Code Representation**: `struct files_struct *files`
- **Description**: Points to a structure that manages the open file descriptors for the process.

`files_struct` Members (Referenced by `files`)

- **fdt**
  - **Code Representation**: `struct fdtable *fdt`
  - **Description**: The file descriptor table, managing open files for the process.

- **next_fd**
  - **Code Representation**: `int next_fd`
  - **Description**: The next available file descriptor in the process's file descriptor table.

11. fs

- **Code Representation**: `struct fs_struct *fs`
- **Description**: Points to filesystem information, including the current and root directories.

12. signal

- **Code Representation**: `struct signal_struct *signal`
- **Description**: Manages signal handling for the process, storing pending signals and signal masks.

`signal_struct` Members (Referenced by `signal`)

- **shared_pending**
  - **Code Representation**: `struct sigpending shared_pending`

- **Description**: Stores signals that are pending and shared across threads.

➢ **rlim**

- **Code Representation**: `struct rlimit rlim[RLIM_NLIMITS]`
- **Description**: Array of resource limits, covering aspects like CPU time and memory usage limits.

13. cpu

- **Code Representation**: `int cpu`
- **Description**: Indicates the CPU on which the process is currently scheduled, aiding in load balancing.

14. start_time

- **Code Representation**: `struct timespec start_time`
- **Description**: Records the start time of the process, useful in calculating the total running time.

15. exit_code

- **Code Representation**: `int exit_code`
- **Description**: Holds the exit code provided by the process upon termination.

16. stack

- **Code Representation**: `void *stack`
- **Description**: Points to the process's kernel stack, utilized during execution for storing function calls and local variables.

QUESTION#2

List of Linux system calls used in process creation and management

1. **fork**:
   Creates a new process by duplicating the calling process. The child process inherits the parent's memory space and gets a unique process ID (PID).

2. **vfork**:
   Similar to `fork`, but it creates a new process without copying the parent's address space until an `exec` call is made, improving performance for quick executions.

3. **exec family** (e.g., execl, execv, execle, execve, etc.):
   Replaces the current process image with a new program. These calls allow a process to execute a different executable, essentially changing its function.

4. **wait**:
   Suspends the calling process until one of its child processes terminates, allowing the parent to retrieve the child's exit status and prevent zombie processes.

5. **waitpid**:
   Similar to `wait`, but it can wait for a specific child process and can be used in a non-blocking manner to avoid being suspended if no children have terminated.

6. **clone**:
   Creates a new process with shared resources, used primarily for threading. This call gives greater control over the shared resources compared to `fork`.

7. **exit**:
   Terminates the calling process and performs necessary cleanup operations. The process can return an exit status to the operating system.

8. **getpid**:
   Returns the process ID of the calling process. This is often used for identifying the current process in various operations.

9. **getppid**:
   Retrieves the parent process ID of the calling process, which can be useful for managing process hierarchies.

10. **setuid**:
    Sets the user ID of the calling process. This changes the process's privileges and can enhance security by limiting access.

11. **setgid**:
    Sets the group ID of the calling process, influencing group permissions and access rights for resources.

12. **kill**:
    Sends a signal to a specified process, which can be used to terminate it or communicate with it. This is crucial for process control and management.

13. **nice**:
    Changes the scheduling priority of a process, which can help manage how much CPU time a process receives based on its priority level.

14. s**ched_yield**:
    Causes the calling process to relinquish the CPU voluntarily, allowing other processes to run. This can improve responsiveness in multitasking environments.

15. **getpriority**:
    Retrieves the scheduling priority of a specified process. This information can help understand the relative importance of processes.

16. **setpriority**:
    Adjusts the scheduling priority of a specified process. This can be used to optimize performance based on process importance.

17. **times**:
    Provides user and system CPU time consumed by the calling process and its children. This information is useful for performance monitoring.

18. **getrlimit**:
    Retrieves resource limits for the calling process, such as maximum memory usage or CPU time. This can help manage resource allocation effectively.

19. **setrlimit**:
    Sets resource limits for the calling process, allowing control over how much system resources can be utilized.

20. **ptrace**:
    Enables one process to observe and control the execution of another, commonly used for debugging. This can facilitate process management and behavior monitoring.

21. **prctl**:
    Provides a way to control the behavior of a process, including setting its name and controlling its child process handling.

22. **sigaction**:
    Allows a process to specify how to handle specific signals. This is crucial for managing interruptions and signals from other processes.

23. **sigprocmask**:
    Examines and changes the signal mask of the calling process, affecting how signals are handled or blocked.

24. **sigpending**:
    Returns the set of signals pending for the calling process. This helps in managing signal delivery and handling.

25. **syscall**:
    Provides a generic interface for making system calls. It can be used to invoke system calls directly with specific parameters.

QUESTION#3

This program creates a binary tree of processes, where each process creates two child processes, forming a tree structure of specified depth. Each process prints its own PID (Process ID) and PPID (Parent Process ID). The program also demonstrates zombie processes by killing a last-level parent process using kill(), after which the orphaned child processes print a message indicating they are "zombie" processes.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>


// Function to print indentation based on the depth level of the process in the tree
void print_indentation(int depth) {
    for (int i = 0; i < depth - 1; i++) {
        printf("    "); // Each level has 4 spaces of indentation
    }
}

// Function to simulate zombie processes if their parent is killed
void check_for_zombie_status() {
    printf("I am a Zombie process (pid %d, ppid %d)\n", getpid(), getppid());
}

// Recursive function to create the binary tree of processes
void create_binary_tree(int depth, int position, int max_depth) {

    print_indentation(depth);
    printf("[%d] pid %d, ppid %d\n", position, getpid(), getppid());

    // Base case: if depth is 1, stop creating further child processes

    if (depth == 1) return;


    pid_t left_pid, right_pid; // Process IDs for left and right child processes
    int left_status, right_status; // Variables to store exit statuses of children

    // Fork to create the left child process

    left_pid = fork();

    if (left_pid < 0) {
        perror("Failed to fork left child");
        exit(1);

    } else if (left_pid == 0) {
        // Inside the left child process
        create_binary_tree(depth - 1, 2 * position, max_depth);  // Recursive call for left child
        exit(position);

    } else {
```

```
        // Back in the parent process, print the creation message for left child
        print_indentation(depth);
        printf("[%d] pid %d created left child with pid %d\n", position, getpid(), left_pid);

        // Wait for the left child to finish before creating the right child
        waitpid(left_pid, &left_status, 0);
        print_indentation(depth);

        printf("[%d] left child %d of %d exited with status %d\n", position, left_pid, getpid(),
WEXITSTATUS(left_status));
    }

    // Fork to create the right child process

    right_pid = fork();

    if (right_pid < 0) {
        perror("Failed to fork right child");
        exit(1);

    } else if (right_pid == 0)
        // Inside the right child process
        create_binary_tree(depth - 1, 2 * position + 1, max_depth);
        exit(position);

    } else {

        // Back in the parent process, print the creation message for right child
        print_indentation(depth);
        printf("[%d] pid %d created right child with pid %d\n", position, getpid(), right_pid);

        // If this is the last level parent (other than main), kill it to create zombie children
        if (depth == max_depth - 1) {
            kill(getpid(), SIGKILL); // Kill this parent process
        }
        waitpid(right_pid, &right_status, 0);
        print_indentation(depth);

        printf("[%d] right child %d of %d exited with status %d\n", position, right_pid, getpid(),
WEXITSTATUS(right_status));
    }

}
```

```c
int main(int argc, char *argv[]) {
    // Check if the user provided a depth as a command-line argument
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <depth>\n", argv[0]); // Error message if no depth is given
        return 1;
    }
    int depth = atoi(argv[1]);
    if (depth < 1) {
        fprintf(stderr, "Depth must be at least 1.\n");
        return 1;
    }
    printf("Starting process tree with depth %d:\n", depth);
    create_binary_tree(depth, 1, depth);

    // If any child finds its parent killed, it will print the zombie message
    check_for_zombie_status();
    return 0;

}
```

OUTPUT

```
mariam@mariam-VirtualBox:~/Desktop$ gcc process.c -o process
mariam@mariam-VirtualBox:~/Desktop$ ./process 3
Starting process tree with depth 3:
        [1] pid 11884, ppid 8042
        [1] pid 11884 created left child with pid 11885
    [2] pid 11885, ppid 11884
    [2] pid 11885 created left child with pid 11886
[4] pid 11886, ppid 11885
    [2] left child 11886 of 11885 exited with status 2
    [2] pid 11885 created right child with pid 11887
[5] pid 11887, ppid 11885
        [1] left child 11885 of 11884 exited with status 0
        [1] pid 11884 created right child with pid 11888
    [3] pid 11888, ppid 11884
    [3] pid 11888 created left child with pid 11889
[6] pid 11889, ppid 11888
    [3] left child 11889 of 11888 exited with status 3
    [3] pid 11888 created right child with pid 11890
[7] pid 11890, ppid 1491
        [1] right child 11888 of 11884 exited with status 0
I am a Zombie process (pid 11884, ppid 8042)
mariam@mariam-VirtualBox:~/Desktop$
```