5 Experiment 5: Process Management using System Calls

5.1 Objective

By the end of this experiment, students will learn about different system commands used to manage processes. They will also gain an understanding of orphan and zombie processes.

5.2 Reading Material[1][3][2]

5.2.1 Process management related system calls

System Call	Description	Programming Syntax
fork()	Creates a new process by duplicating the calling process.	$\text{pid}_t fork(void)$;
exec()	Replaces the current process with a new program.	int exec(const char *pathname, char *const argv[]);
wait()	Causes the parent process to wait until one of its child processes terminates.	$\text{pid}_t wait(int * status)$;
exit()	Terminates the calling process and returns an exit status to the operating system.	void exit(int status);
getpid()	Retrieves the process ID (PID) of the calling process.	$\text{pid}_t getpid(void)$;
getppid()	Retrieves the parent process ID (PPID) of the calling process.	$\text{pid}_t getppid(void)$;
kill()	Sends a signal to a specified process or group of processes.	int kill($\text{pid}_t pid, int sig$);
nice()	Modifies the priority of a process.	int nice(int incr);
sleep()	Causes the calling process to sleep for a specified number of seconds.	unsigned int sleep(unsigned int seconds);

Table 4: Process management related system calls

5.2.2 Orphan and Zombie Processes

Orphan Process: An orphan process is a child process whose parent process has terminated or finished before the child process completes. When the parent process exits or is terminated unexpectedly without properly waiting for the child to finish, the operating system reassigns the orphaned child process to the init process (PID 1 in Linux). The init process adopts and manages orphan processes until they complete execution.

Zombie Process: A zombie process is a terminated process that has completed its execution but still has an entry in the process table. After a process completes, it sends an exit status to its parent process and becomes a zombie waiting for the parent to retrieve the exit status using the `wait()` system call. If the parent fails to fetch the exit status of the terminated child (due to neglect or termination), the zombie process remains in the process table as an inactive process entry.

%begincenter

6 Experiment 6: Creation of Multithreaded Processes using Pthread Library

6.1 Objective

Introduce the operations on threads, which include initialization, creation, join and exit functions of thread using pthread library.

6.2 Reading Material[1][3][2]

6.2.1 Commonly used library functions related to POSIX threads (pthread)

Function	Description	Programming Syntax
pthread_create	Create a new thread	int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
pthread_join	Wait for termination of a specific thread	int pthread_join(pthread_t thread, void **retval);
pthread_exit	Terminate calling thread	void pthread_exit(void *retval);
pthread_cancel	Request cancellation of a thread	int pthread_cancel(pthread_t thread);

Table 5: Commonly used library functions related to POSIX threads (pthread)

6.3 Sample Programs

1. A C program using the pthread library to create a thread with NULL attributes.

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

void *thread_function(void *arg) {
    printf("Inside the new thread!\n");
    return NULL;
}

int main() {
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thread_function, NULL);
    pthread_join(thread_id, NULL);
    return 0;
}
```

2. A C program that creates a thread and passes a message from the main function to the thread.

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

void *thread_function(void *message) {
    printf("Message received in thread: %s\n", (char *)message);
    return NULL;
}

int main() {
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thread_function, "Hello from the main thread!");
    pthread_join(thread_id, NULL);
    return 0;
}
```

3. A C program where a thread returns a value to the main function using pointers.

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

#define NUM_THREADS 1

void *thread_function(void *arg) {
    int *returnValue = malloc(sizeof(int));
    *returnValue = 143; // Set the return value
    pthread_exit(returnValue);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int *thread_return;

    pthread_create(&threads[0], NULL, thread_function, NULL);
    pthread_join(threads[0], (void **)&thread_return);

    printf("Value returned from thread: %d\n", *thread_return);

    free(thread_return); // Free allocated memory for return value
    return 0;
}
```

pthread_t tid;
int num1 = 5, num2 = 8;
pthread_create(&tid, NULL, calculate_statistics, (void *)(&num1));
pthread_join(tid, NULL);
return 0;
}

Video Reference:

https://youtube.com/playlist?list=PLWjmN065fOfGdAZrlP6316HVHh8jlve_

References

[1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.

[2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.

[3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

6.4 Lab Exercises: [Attempt any 3 within the designated lab hours]

Exercise 1: Develop a program using pthread to concatenate multiple strings passed to the thread function.

Exercise 2: Create a pthread program to find the length of strings passed to the thread function.

Exercise 3: Implement a program that performs statistical operations (calculating average, maximum, and minimum) for a set of numbers. Utilize three threads, where each thread performs its respective operation.

Exercise 4: Write a multithreaded program where a globally passed array of integers is divided into two smaller lists and given as input to two threads. Each thread sorts their half of the list and then passes the sorted lists to a third thread, which merges and sorts them. The final sorted list is printed by the parent thread.

Exercise 5: Create a program using `pthread_create` to generate multiple threads. Each thread should display its unique ID and execution sequence.

Exercise 6: Create a threaded application that demonstrates graceful thread termination using `pthread_exit` for resource cleanup compared to abrupt termination via `pthread_cancel`.

6.5 Sample Viva Questions

Question 1: Discuss the steps involved in creating a thread using the `pthread_create` function.

Question 2: What parameters does the `pthread_create` function take, and what are their purposes?

Question 3: Explain the role and usage of the `pthread_join` function in managing threads.