

MODULE 3

1. Which signals are triggered, when the following actions are performed.

- user press ctrl+C - SIGINT signal
- kill() system call is invoked - SIGTERM
- CPU tried to execute an illegal instruction - SIGILL
- When the program access the unassigned memory - SIGSEGV

2. List the gdb command for the following operations

- To run the current executable file: run or r
- To create breakpoints at: break or b
- To resume execution once after a breakpoint: continue or c
- To clear a breakpoint created for a function: clear
- Print the parameters of the function in the backtrace: info args

3. Guess the output for the following program.

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

```
#include <unistd.h>
```

```
int main() {
```

```
    if (fork() && (!fork())) {
```

```
        if (fork() || fork()) {
```

```
            fork();
```

```
            printf("2");
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

Output

2222222

4. Guess the output for the following program.

```

#include <stdio.h>

#include <unistd.h>

int main() {

    if (fork()) {

        fork();

    } else {

        if (fork()) {

            fork();

            printf("1");

        } else {

            printf("2");

            printf("3");

            printf("4");

        }

        return 0;

    }

}

```

OUTPUT

2 4 1 4 1 4 3 4

5. Create two thread functions to print hello and world separately and create threads for each and execute them one after other in C

```

#include <stdio.h>
#include <pthread.h>
void *printHello(void *arg) {
    printf("Hello ");
    pthread_exit(NULL);
}
void *printWorld(void *arg) {
    printf("World\n");
    pthread_exit(NULL);
}

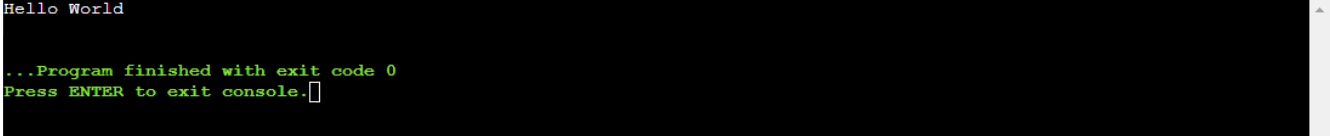
```

```

int main() {
    pthread_t thread1, thread2;
    if (pthread_create(&thread1, NULL, printHello, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }
    if (pthread_join(thread1, NULL) != 0) {
        perror("pthread_join");
        return 1;
    }
    if (pthread_create(&thread2, NULL, printWorld, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }
    if (pthread_join(thread2, NULL) != 0) {
        perror("pthread_join");
        return 1;
    }
    return 0;
}

```

OUTPUT



```

Hello World

...Program finished with exit code 0
Press ENTER to exit console.

```

6. How to avoid Race conditions and deadlocks?

A race condition occurs when the outcome of a program depends on the non-deterministic order of execution of concurrent threads accessing shared resources. Race conditions often lead to unpredictable behavior and incorrect results because threads can interfere with each other's operations.

It can be avoided by using the following strategies

- Use synchronization primitives such as locks, semaphores, or mutexes to control access to shared resources. These mechanisms ensure that only one thread can access a critical section of code at a time.
- Use atomic operations or atomic data types provided by the programming language or library to perform operations that must be executed as a single, indivisible unit.
- Prefer thread-safe data structures, such as mutex-protected queues or hash tables, which internally handle synchronization to ensure safe concurrent access.

A deadlock occurs when two or more threads are blocked indefinitely because each thread holds a resource and waits for another resource held by another thread, resulting in a circular waiting dependency. Deadlocks halt the progress of the program, leading to a state where no thread can proceed.

It can be avoided by using the following strategies

- Define a strict ordering of resource acquisition and always acquire resources in the same order to prevent circular dependencies.
- Use timeouts when acquiring locks or resources. If a thread cannot acquire a resource within a specified time limit, it releases the held resources and retries later to avoid deadlock.
- Establish a lock hierarchy where locks are acquired in a predefined hierarchical order. Threads are allowed to acquire locks only in the order defined by the hierarchy.
- Avoid situations where a thread holds one resource while waiting for another. Instead, acquire all required resources simultaneously, or release held resources before waiting for additional resources.

7. What is the difference between exec and fork ?

EXEC	FORK
It is an operation used in the UNIX OS that lets a process create its copies that do not replace the original process.	It is also an operation used in the UNIX OS, but it creates child processes that replace the parent process or the previous process.
Both parent and child processes exist in the system after making this function call.	After calling the exec(), only the child process exists. No parent process is present since the child process replaces it.
The child process created via the fork() system call is always similar to its parent process. They both coexist.	The child process created via the exec() system call replaces its parent process.
Since the parent and child process coexist after the fork() system call, they reside in different address spaces.	Since the child process replaces the parent process, it also replaces its address space. Thus, they both have the same address space in the system.

8. What is the difference between process and threads.

process	thread
Process means any program is in execution.	Thread means a segment of a process.

The process takes more time to terminate and create.	The thread takes less time to terminate and create.
The process is called the heavyweight process.	A Thread is lightweight as each thread in a process shares code, data, and resources.
The process has its own Process Control Block, Stack, and Address Space.	Thread has Parents' PCB, its own Thread Control Block, and Stack and common Address space.

9. Write a C program to demonstrate the use of Mutexes in threads synchronization

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
pthread_t tid[2];
int counter;
void* trythis(void* arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Job %d has started\n", counter);

    for (i = 0; i < (0xFFFFFFFF); i++) ;
    printf("\n Job %d has finished\n", counter);
    return NULL;
}
int main(void)
{
    int i = 0;
    int error;
    while (i < 2) {
        error = pthread_create(&(tid[i]), NULL, &trythis, NULL);
        if (error != 0)
            printf("\nThread can't be created : [%s]", strerror(error));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    return 0;
}
```

OUTPUT

```
Job 2 has started
```

```
Job 1 has started
```