MODULE 3

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

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

Result:

User presses Ctrl+C:

The Ctrl+C keyboard combination sends an interrupt signal, specifically SIGINT (Signal Interrupt), to the currently running process in the terminal. This signal is typically used to request the program to terminate.

kill() system call is invoked:

The kill() system call is used to send a signal to a process. The signal can be specified as an argument to the kill() function.

Common signals include:

SIGTERM (15): Termination signal, allowing the process to perform cleanup operations before exiting.

SIGKILL (9): Immediate termination signal, forcefully terminating the process without allowing it to perform any cleanup.

CPU tries to execute an illegal instruction:

When the CPU attempts to execute an illegal or undefined instruction, it triggers an interrupt or exception. The specific signal generated can depend on the operating system and architecture, but often it is SIGILL (Signal Illegal Instruction).

Program accesses unassigned memory:

When a program tries to access unassigned or protected memory, it results in a segmentation fault. This triggers the SIGSEGV signal (Signal Segmentation Violation), indicating that the program has attempted an invalid memory access.

2. List the gdb command for the following operations

- To run the current executable file
- To create breakpoints at
- To resume execution once after breakpoint
- To clear break point created for a function
- Print the parameters of the function in the backtrace

Result:

To run the current executable file:

→ **run**

To create breakpoints at a specific location:

→ **break <location>**

To resume execution once after a breakpoint:

→ **continue**

To clear a breakpoint created for a function:

→ **clear <function_name>**

Print the parameters of the function in the backtrace:

→ **backtrace**

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;
}
```

Result:

```
[ad-exam001@mepcolinux module3]$cat qn3.c
#include <stdio.h>
#include <unistd.h>
int main() {
        if (fork() && (!fork())) {
                if (fork() || fork()) {
                        fork();
                                }
                }
            printf("2");
                return 0;
}
[ad-exam001@mepcolinux module3]$cc qn3.c
[ad-exam001@mepcolinux module3]$./a.out
2222[ad-exam001@mepcolinux module3]$222
```

4. Guess the output for the following program.

```
#include <stdio.h>
#include <unistd.h>
int main() {
   if (fork()) {
      if (!fork()) {
         fork();
         printf("1");
      } else {
         printf("2");
      }
   } else {
      printf("3");
   }
   printf("4");
   return 0;
}
```

Result:

```
[ad-exam001@mepcolinux ~]$cat qn4.c
#include <stdio.h>
#include <unistd.h>
int main() {
       if (fork()) {
                  if (!fork()) {
                                 fork();
                                                printf("1");
                                                       } else {
                                                                                printf("2");
                                                                                       }
                     } else {
                                 printf("3");
                                     }
           printf("4");
               return 0;
}
[ad-exam001@mepcolinux ~]$cc qn4.c
[ad-exam001@mepcolinux ~]$./a.out
24341414[ad-exam001@mepcolinux ~]$
```

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

Source code:

```
#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) {
      fprintf(stderr, "Error creating thread 1\n");
      return 1;
   }


   pthread_join(thread1, NULL);


   if (pthread_create(&thread2, NULL, printWorld, NULL) != 0) {
      fprintf(stderr, "Error creating thread 2\n");
      return 2;
   }


   pthread_join(thread2, NULL);
```

```
    return 0;
}
```

```
[ad-exam001@mepcolinux ~]$cat qn5.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) {
                     fprintf(stderr, "Error creating thread 1\n");
                             return 1;
                                 }


               pthread_join(thread1, NULL);


                   if (pthread_create(&thread2, NULL, printWorld, NULL) != 0) {
                             fprintf(stderr, "Error creating thread 2\n");
                                     return 2;
                                         }


                        pthread_join(thread2, NULL);

                            return 0;
}



[ad-exam001@mepcolinux ~]$gcc -o thread_program qn5.c -pthread
[ad-exam001@mepcolinux ~]$./thread_program
Hello World
```

6. How to avoid Race conditions and deadlocks?

Result:

Race conditions occur when two or more threads access shared data concurrently, and the final outcome depends on the timing of their execution. To avoid race conditions, synchronization mechanisms should be employed to ensure that critical sections of code are executed by only one thread at a time.

Source code:

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


int sharedVariable = 0;


pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void *incrementSharedVariable(void *arg) {

   pthread_mutex_lock(&mutex);


   sharedVariable++;
   printf("Thread ID %ld: Shared variable = %d\n", pthread_self(), sharedVariable);

   pthread_mutex_unlock(&mutex);

   pthread_exit(NULL);
}

int main() {
   pthread_t thread1, thread2;

   pthread_create(&thread1, NULL, incrementSharedVariable, NULL);
```

```
    pthread_create(&thread2, NULL, incrementSharedVariable, NULL);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    return 0;
}
```

```
[ad-exam001@mepcolinux ~]$cat qn6.c
#include <stdio.h>
#include <pthread.h>


int sharedVariable = 0;


pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;


void *incrementSharedVariable(void *arg) {

       pthread_mutex_lock(&mutex);


           sharedVariable++;
               printf("Thread ID %ld: Shared variable = %d\n", pthread_self(), sharedVariable);

                   pthread_mutex_unlock(&mutex);

                       pthread_exit(NULL);
}

int main() {
       pthread_t thread1, thread2;

           pthread_create(&thread1, NULL, incrementSharedVariable, NULL);
               pthread_create(&thread2, NULL, incrementSharedVariable, NULL);

                   pthread_join(thread1, NULL);
                       pthread_join(thread2, NULL);

                           return 0;
}
[ad-exam001@mepcolinux ~]$gcc -o race_condition qn6.c -pthread
[ad-exam001@mepcolinux ~]$./race_condition
Thread ID 139798797928192: Shared variable = 1
Thread ID 139798789535488: Shared variable = 2
[ad-exam001@mepcolinux ~]$
```

Avoiding Deadlocks:

Deadlocks occur when two or more threads are blocked forever, each waiting for the other to release a resource. To avoid deadlocks, it's essential to follow some

Lock Ordering:

Establish a global order for acquiring locks and ensure that all threads follow the same order when acquiring multiple locks.

Lock Timeout:

Use timeouts when acquiring locks. If a thread cannot acquire a lock within a specified time, it releases all acquired locks and retries, preventing deadlock.

Lock Hierarchies:

Implement a hierarchy for acquiring locks. Ensure that all threads acquire locks in the same order, preventing cyclic dependencies.

Sample code:

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

pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void *arg) {
    pthread_mutex_lock(&mutexA);
    printf("Thread 1 acquired mutexA\n");


    sleep(1);

    pthread_mutex_lock(&mutexB);  // Potential deadlock point
    printf("Thread 1 acquired mutexB\n");

    pthread_mutex_unlock(&mutexB);
```

```
    pthread_mutex_unlock(&mutexA);

    pthread_exit(NULL);
}

void *thread2(void *arg) {
    pthread_mutex_lock(&mutexB);
    printf("Thread 2 acquired mutexB\n");

    pthread_mutex_lock(&mutexA);  // Potential deadlock point
    printf("Thread 2 acquired mutexA\n");

    pthread_mutex_unlock(&mutexA);
    pthread_mutex_unlock(&mutexB);

    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;

    pthread_create(&t1, NULL, thread1, NULL);
    pthread_create(&t2, NULL, thread2, NULL);

    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

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


pthread_mutex_t mutexA = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t mutexB = PTHREAD_MUTEX_INITIALIZER;

void *thread1(void *arg) {
       pthread_mutex_lock(&mutexA);
           printf("Thread 1 acquired mutexA\n");

               sleep(1);

                   pthread_mutex_lock(&mutexB);
                       printf("Thread 1 acquired mutexB\n");

                           pthread_mutex_unlock(&mutexB);
                               pthread_mutex_unlock(&mutexA);

                                   pthread_exit(NULL);
}

void *thread2(void *arg) {
       pthread_mutex_lock(&mutexB);
           printf("Thread 2 acquired mutexB\n");

               pthread_mutex_lock(&mutexA);  // Potential deadlock point
                   printf("Thread 2 acquired mutexA\n");

                       pthread_mutex_unlock(&mutexA);
                           pthread_mutex_unlock(&mutexB);

                               pthread_exit(NULL);
}

int main() {
       pthread_t t1, t2;

           pthread_create(&t1, NULL, thread1, NULL);
               pthread_create(&t2, NULL, thread2, NULL);

                   pthread_join(t1, NULL);
                       pthread_join(t2, NULL);

                           return 0;
}
[ad-exam001@mepcolinux ~]$gcc -o deadlock qn6a.c -pthread
[ad-exam001@mepcolinux ~]$./deadlock
Thread 1 acquired mutexA
Thread 2 acquired mutexB
```

7. What is the difference between exec and fork ?

fork():

- fork() is used to create a new process, which becomes a copy of the calling process (the parent process).
- After a successful fork(), two separate processes are created: the parent process and the child process.
- The child process is an exact copy of the parent process, including its memory, file descriptors, and other attributes.
- fork() returns different values in the parent and child processes: it returns the child's process ID (PID) to the parent and 0 to the child.
- The child process usually continues executing from the point where fork() was called, while the parent process can either continue its execution or wait for the child process to finish using functions like wait() or waitpid().

exec():

- exec() is used to replace the current process image with a new process image. It loads a new program into the current process space, overwriting the previous program.
- Unlike fork(), exec() does not create a new process; instead, it replaces the existing process with a new one.
- There are several variants of the exec() system call, such as execl(), execv(), execle(), execve(), etc., each with different ways of specifying the program to be executed and its arguments.
- After a successful exec(), the new program starts executing, inheriting the process ID (PID) of the original process and any open file descriptors that were not marked as close-on-exec.

8. What is the difference between process and threads.

Process:

- A process is an independent program in execution. It has its own memory space, resources, and state.
- Processes are isolated from each other, meaning they cannot directly access each other's memory.
- Communication between processes typically involves inter-process communication (IPC) mechanisms such as pipes, message queues, or shared files.
- Processes are heavyweight, as they require a separate memory space and resources.

Thread:

- A thread is a lightweight unit of execution within a process. Multiple threads within a process share the same resources, including memory.
- Threads within the same process can communicate more easily through shared memory.
- Threads are more lightweight compared to processes, as they share resources and can be created and terminated more quickly.
- Threads within the same process can be scheduled independently, allowing for parallel execution.

9. Write a C program to demonstrate the use of Mutexes in threads synchronization...explain it and give me one sample program

Source code:

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

#define NUM_THREADS 5

int sharedCounter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void *incrementCounter(void *threadID) {
   long tid = (long)threadID;


   pthread_mutex_lock(&mutex);


   sharedCounter++;
   printf("Thread %ld incremented the counter. Current value: %d\n", tid, sharedCounter);


   pthread_mutex_unlock(&mutex);

   pthread_exit(NULL);
}

int main() {
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;

   for (t = 0; t < NUM_THREADS; t++) {
      printf("Creating Thread %ld\n", t);
      rc = pthread_create(&threads[t], NULL, incrementCounter, (void *)t);

      if (rc) {
         printf("Error: Return code from pthread_create() is %d\n", rc);
         return 1;
      }
   }


   for (t = 0; t < NUM_THREADS; t++) {
      pthread_join(threads[t], NULL);
   }

   printf("All threads have completed. Final counter value: %d\n", sharedCounter);

   return 0;
}
```

Result:

```
#define NUM_THREADS 5

int sharedCounter = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *incrementCounter(void *threadID) {
       long tid = (long)threadID;


           pthread_mutex_lock(&mutex);


               sharedCounter++;
                   printf("Thread %ld incremented the counter. Current value: %d\n", tid, sharedCounter);


                       pthread_mutex_unlock(&mutex);

                           pthread_exit(NULL);
}

int main() {
       pthread_t threads[NUM_THREADS];
           int rc;
               long t;

                   for (t = 0; t < NUM_THREADS; t++) {
                             printf("Creating Thread %ld\n", t);
                                    rc = pthread_create(&threads[t], NULL, incrementCounter, (void *)t);

                                           if (rc) {
                                                        printf("Error: Return code from pthread_create() is %d\n", rc);
                                                                   return 1;
                                                                          }
                                               }


                       for (t = 0; t < NUM_THREADS; t++) {
                                 pthread_join(threads[t], NULL);
                                     }

                           printf("All threads have completed. Final counter value: %d\n", sharedCounter);

                               return 0;
}

[ad-exam001@mepcolinux ~]$gcc -o mutex_threads qn6a.c -pthread
[ad-exam001@mepcolinux ~]$./mutex_threads
Thread 1 acquired mutexA
Thread 2 acquired mutexB
```