Module 3 Assessment

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

1. **user press ctrl+C**
2. **kill() system call is invoked**
3. **CPU tried to execute an illegal instruction**
4. **When the program access the unassigned memory**

1. When the user presses **Ctrl+C**:
   - Signal: **SIGINT** (Interrupt signal)
   - Description: This signal is typically sent by the terminal to interrupt the running process. It's commonly used to gracefully terminate a program.
2. When the **kill()** system call is invoked:
   - Signal: Specified by the user as the first argument of the **kill()** function.
   - Description: The **kill()** function sends a signal to a specified process or process group. The signal sent depends on the signal number passed as an argument.
3. When the CPU tries to execute an illegal instruction:
   - Signal: **SIGILL** (Illegal instruction signal)
   - Description: This signal is raised when the CPU encounters an illegal or undefined instruction during program execution.
4. When the program accesses unassigned memory:
   - Signal: **SIGSEGV** (Segmentation fault signal)
   - Description: This signal is raised when a program attempts to access memory that is not allowed (such as unassigned memory or memory that is outside of its allocated region). It indicates a memory access violation or segmentation fault.

2. List the gdb command for the following operations

1. **To run the current executable file**
2. **To create breakpoints at**
3. **To resume execution once after breakpoint**
4. **To clear break point created for a function**
5. **Print the parameters of the function in the backtrace**

1. Run the current executable file: **run**
2. Create breakpoints:
    - At a specific line number: **break <line_number>**
    - At a specific function: **break <function_name>**
    - At a specific file and line number: **break <file_name>:<line_number>**
3. Resume execution once after a breakpoint: **continue**
4. Clear a breakpoint created for a function: **clear <function_name>**
5. Print the parameters of the function in the backtrace: **bt full**

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;
}
```

The output of the given program will be "2".

1. The program includes the `<stdio.h>` and `<unistd.h>` header files, which provide the necessary functions and declarations for input/output operations and the `fork()` system call, respectively.

2. In the `main()` function, the first condition `(fork() && (!fork()))` is evaluated.

- The `fork()` system call creates a new process by duplicating the calling process. It returns 0 for the child process and the process ID of the child for the parent process.

- The first `fork()` call creates a child process, and the parent process continues executing.

- The second `!fork()` evaluates to `false` (0) for the child process and `true` (non-zero) for the parent process.

- Therefore, the condition `(fork() && (!fork()))` evaluates to `true` (non-zero) only for the parent process and `false` (0) for the child process.

3. Inside the first `if` statement, another condition `(fork() || fork())` is evaluated.

- For the parent process, the first `fork()` call creates another child process, and the parent process continues executing.

- For the child process (from the first `fork()`), the second `fork()` call creates another child process, and the original child process continues executing.

- Therefore, the condition `(fork() || fork())` evaluates to `true` (non-zero) for both the parent process and the child process (from the first `fork()`).

4. Inside the second `if` statement, the `fork()` system call is executed again, creating another child process.

5. The `printf("2 ");` statement is executed by all the processes created (the parent process and all child processes).

6. The program exits with a return value of 0.

Since the `printf("2 ");` statement is executed by all the processes created, the output will be "2" printed multiple times (depending on the number of processes created). However, due to the nature of the `fork()` system call and the way processes are scheduled, the exact number of times "2" is printed may vary between different runs of the program.

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;
}
```

The output of this program will be "2 1 4".

1. The program includes the `<stdio.h>` and `<unistd.h>` header files, which provide the necessary functions and declarations for input/output operations and the `fork()` system call, respectively.

2. In the `main()` function, the first `if` statement checks the condition `fork()`.

- The `fork()` system call creates a new process by duplicating the calling process. It returns 0 for the child process and the process ID of the child for the parent process.

- If `fork()` is successful (non-zero value), it means the current process is the parent process, and the code inside the `if` block will be executed.

- If `fork()` returns 0, it means the current process is the child process, and the code inside the `else` block associated with the outer `else` statement will be executed.

3. Inside the `if` block, another `if` statement checks the condition `!fork()`.

- If `!fork()` is true (non-zero value), it means the current process is the child process of the child process created in the previous `fork()` call.

- Inside this nested `if` block, another `fork()` call is made, creating a new child process.

- The `printf("1 ");` statement is executed by this child process.

4. If the condition `!fork()` is false (0), it means the current process is the parent process of the child process created in the previous `fork()` call.

- In this case, the `printf("2 ");` statement is executed by the parent process.

5. If the original `fork()` call returns 0, it means the current process is the child process of the original parent process.

- In this case, the code inside the outer `else` block is executed, which prints `"3 "`.

6. After all the nested `if` statements and `else` blocks, the `printf("4 ");` statement is executed by all the processes.

7. Finally, the program exits with a return value of 0.

So, the order of execution and the output will be:

1. The parent process executes `printf("2 ");`.

2. The child process of the parent process executes `printf("3 ");`.

3. The child process of the child process executes `printf("1 ");`.

4. All processes execute `printf("4 ");`.

Therefore, the output of the program will be "2 1 4" (the order may vary slightly due to the scheduling of processes, but the output will always contain these three numbers).

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>


// Function to print "Hello"

void *printHello(void *arg) {

   printf("Hello ");

   pthread_exit(NULL);
```

```
}

// Function to print "World"
void *printWorld(void *arg) {
    printf("World\n");
    pthread_exit(NULL);
}

int main() {
    pthread_t thread1, thread2;

    // Create thread for printing "Hello"
    if (pthread_create(&thread1, NULL, printHello, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }

    // Wait for thread1 to finish before creating thread for printing "World"
    if (pthread_join(thread1, NULL) != 0) {
        perror("pthread_join");
        return 1;
    }

    // Create thread for printing "World"
    if (pthread_create(&thread2, NULL, printWorld, NULL) != 0) {
        perror("pthread_create");
        return 1;
    }

    // Wait for thread2 to finish
```

```
    if (pthread_join(thread2, NULL) != 0) {
        perror("pthread_join");
        return 1;
    }

    return 0;
}
```

6. **How to avoid Race conditions and deadlocks?**

To avoid race conditions:

- Use synchronization mechanisms like mutexes to control access to shared resources.
- Minimize critical sections to reduce the window for race conditions.
- Limit shared data between threads and use thread-safe data structures.
- Employ atomic operations for shared data access.
- Ensure thread-safe design and synchronization for shared data.

To avoid deadlocks:

- Ensure resources are acquired and released in a consistent order to avoid circular dependencies.
- Implement timeout mechanisms to prevent threads from waiting indefinitely.
- Minimize nested locks and establish lock hierarchies to prevent deadlock situations.
- Implement deadlock detection mechanisms to identify and recover from deadlocks.
- Ensure proper resource allocation and release to prevent resource contention and deadlock scenarios.

7. **What is the difference between exec and fork ?**

Aspect	**fork**	**exec**
Purpose	Creates a new process as a copy of the parent process	Replaces the current process image with a new program
System call	fork()	exec() and its variants (e.g., execl, execv, execvp)
Result	Creates a new process with its own PID and memory space, identical to the parent process	Replaces the current process image with a new program

Aspect	fork	exec
Process flow	Both parent and child processes continue execution from the point of the fork call	The current process is terminated, and the new program starts executing from its entry point
Memory	Child process inherits copies of the parent's memory, file descriptors, and other attributes	Memory of the current process is replaced with the memory of the new program
Usage	Typically used for process creation and concurrency	Used to execute a different program within the same process

8. What is the difference between process and threads.

Aspect	Process	Thread
Definition	An instance of a running program	A lightweight sub-process, also known as a "lightweight process"
Creation	Created by the operating system	Created within a process by the program or the operating system
Resource consumption	Each process has its own memory space, file descriptors, and other resources	Threads within the same process share the same memory space and resources
Communication	Inter-process communication (IPC) mechanisms are required for communication between processes	Threads within the same process can communicate directly through shared memory and variables
Context switching	Context switching between processes is more expensive as it involves switching between different memory spaces	Context switching between threads is less expensive as they share the same memory space
Scalability	Processes are heavyweight and have higher overhead, making them less scalable	Threads are lightweight and have lower overhead, making them more scalable
Fault tolerance	Processes are more isolated, so if one process crashes, it does not affect others	Threads within the same process share the same memory space, so if one thread crashes, it can affect others

Aspect	Process	Thread
Parallelism	Processes can run in parallel on multi-core systems	Threads can run concurrently within the same process, but true parallelism depends on the underlying hardware and scheduling

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

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

#define NUM_THREADS 5

pthread_mutex_t mutex;  // Mutex variable

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

   // Lock the mutex before accessing shared resource
   pthread_mutex_lock(&mutex);

   // Critical section
   printf("Hello from thread #%ld\n", tid);

   // Unlock the mutex after accessing shared resource
   pthread_mutex_unlock(&mutex);

   pthread_exit(NULL);
}

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

   // Initialize the mutex
   pthread_mutex_init(&mutex, NULL);

   // Create multiple threads
   for (t = 0; t < NUM_THREADS; t++) {
      printf("Creating thread #%ld\n", t);
      rc = pthread_create(&threads[t], NULL, printHello, (void*)t);
      if (rc) {
         printf("ERROR: pthread_create() failed with code %d\n", rc);
         return 1;
```

```
        }
    }

    // Wait for all threads to complete
    for (t = 0; t < NUM_THREADS; t++) {
        pthread_join(threads[t], NULL);
    }

    // Destroy the mutex
    pthread_mutex_destroy(&mutex);

    return 0;
}
```