MODULE 3 ASSIGNMENTS

1. Which signals are triggered, when the following actions are performed .
   - **i.** user press ctrl+C - **SIGINT**.
   - **ii.** Kill() system call is invoked – **SIGTERM**.
   - **iii.** CPU tried to execute an illegal instruction – **SIGILL**.
   - **iv.** When program tries to access the unassigned memory – **SEGSEGV**.

2. List the gdb command for the following operations.
   - **i.** To run the current executable file – **run[args].**
   - **ii.** To create breakpoints at – **b** or **break[function_name]** or **break[line_number].**
   - **iii.** To resume execution once after breakpoint – **c** or **continue**.
   - **iv.** To clear break point created for a function - **clear** or **clear[function_name]** or **clear[line_number]**.
   - **v.** Print the parameters of the functions in the backtrace – **bt**.

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;
}
```

ANS : 2 2

4. Guess the output of the 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;
}
```

ANS: 2434 or 24 or 3424

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

main.c

```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;
    }
```

6. How to avoid race condition and deadlocks.
   Race condition : Race condition can be avoided by the concept of synchronization. By using sleep() wait() or circular wait mechanisms.
   Dead locks: Dead locks can be avoided using or using mutex locks and semaphores. By which we let only one process to access the critical region (may it be a global variable or any memory that is common to the execution run time) at a given time.

7. Difference between exec() and fork().
   In simple terms fork() creates an exact image of the parent process and continues to execute both whereas the exec() is used to load a new program or set of instructions different from the parent process.

8. Difference between process and threads.
   - Fundamentally, Process is a runtime instance of program and threads are sub units of a single process. Different processes execute different programs but threads of the same process execute different tasks of the same process.

- Importantly, Different processes do not share memory or data between them and have their own memory space but threads of the same process have a shared memory space and can communicate within themselves directly.

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

main.c

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

pthread_mutex_t mutex;

void *thread_function(void *arg) {
    int thread_id = *((int *)arg);
    pthread_mutex_lock(&mutex);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
int main() {
    pthread_t threads[2];
    int thread_args[2];

    pthread_mutex_init(&mutex, NULL);

    for (int i = 0; i < 2; i++) {
        thread_args[i] = i + 1;
        pthread_create(&threads[i], NULL, thread_function, &thread_args[i]);
    }

    for (int i = 0; i < 2; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex);

    return 0;
}

```