ADVANCED C PROGRAMMING – MODULE 3

1. **Which signals are triggered, when the following actions are performed?**
    a. user press ctrl+C - **SIGINT**
    When a user presses Ctrl+C, it typically sends the **SIGINT** signal to the foreground process running in the terminal. This signal is used to interrupt the process and typically prompts it to terminate.

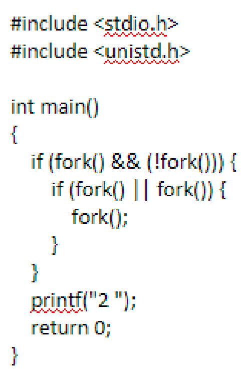
    b. kill() system call is invoked - **SIGTERM** (By default)
    The kill() system call is used to send signals to processes. When a process receives a kill() call, it can be instructed to perform different actions based on the signal sent as an argument.
    - If a process receives the **SIGTERM** signal, it typically terminates gracefully.
    - If a process receives the **SIGKILL** signal, it is immediately terminated without any chance for cleanup.
    - If a process receives the **SIGSTOP** signal, it is paused.
    - If a process receives the **SIGCONT** signal, it is resumed if it was previously paused.

    c. CPU tried to execute an illegal instruction – **SIGILL**
    When the CPU attempts to execute an illegal instruction, it typically generates a hardware exception or trap. This usually results in the operating system handling the exception and sending a **SIGILL** (illegal instruction) signal to the offending process.

    d. When the program access the unassigned memory – **SIGSEGV**
    When a program accesses memory that it doesn't have permission to access, it typically results in a segmentation fault. This generates a **SIGSEGV** (segmentation violation) signal, which is sent to the offending process. The process then typically terminates due to receiving this signal.

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

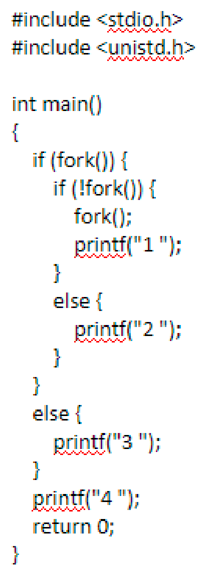
3. **Guess the output for the following program.**



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

int main()
{
   if (fork() && (!fork())) {
      if (fork() || fork()) {
         fork();
      }
   }
   printf("2 ");
   return 0;
}
```

Output: 2222222

4. Guess the output for the following program.

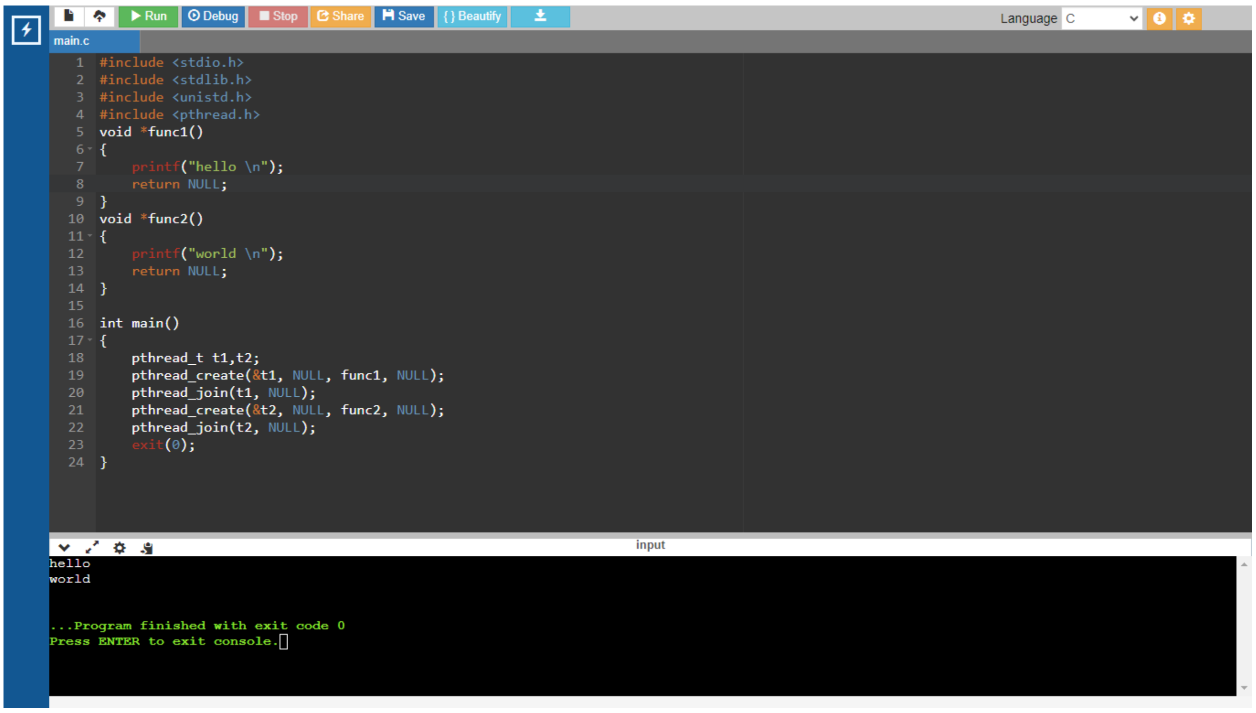


```
#include <stdio.h>
#include <unistd.h>

int main()
{
   if (fork()) {
      if (!fork()) {
         fork();
         printf("1 ");
      }
      else {
         printf("2 ");
      }
   }
   else {
      printf("3 ");
   }
   printf("4 ");
   return 0;
}
```

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 <unistd.h>
#include <pthread.h>
void *func1()
{
   printf("hello \n");
   return NULL;
}
void *func2()
{
   printf("world \n");
   return NULL;
}

int main()
{
   pthread_t t1,t2;
   pthread_create(&t1, NULL, func1, NULL);
   pthread_join(t1, NULL);
   pthread_create(&t2, NULL, func2, NULL);
   pthread_join(t2, NULL);
   exit(0);
}
```



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
void *func1()
{
    printf("hello \n");
    return NULL;
}
void *func2()
{
    printf("world \n");
    return NULL;
}

int main()
{
    pthread_t t1,t2;
    pthread_create(&t1, NULL, func1, NULL);
    pthread_join(t1, NULL);
    pthread_create(&t2, NULL, func2, NULL);
    pthread_join(t2, NULL);
    exit(0);
}
```

```
hello
world


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

6. **How to avoid Race Conditions and Deadlocks?**

- To avoid race conditions we can use synchronization mechanisms like locks or semaphores to control access to shared resources. Only one thread can access a critical section at a time. Also prefer immutable data structures or thread-local storage to reduce shared data. Immutable data cannot be changed after creation, eliminating the possibility of race conditions.

- To avoid deadlock we can implement deadlock detection mechanisms or design systems to avoid deadlocks. We can also use deadlock avoidance algorithm (Bankers' Algorithm). Always release acquired resources when they're no longer needed to avoid resource starvation and potential deadlocks.

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

fork()	**exec()**
Create a new process by duplicating the current process.	Replace the current process image with a new one.
Creates two separate processes: the parent and the child.	Replaces the current process with a new program.
Child process receives a return value of 0. Parent process receives the child's PID.	Does not return to the calling process.
Both parent and child processes have identical copies of memory at the time of the fork call.	Memory of the current process is replaced by the new program.
Both parent and child processes share the same resources initially.	No resource sharing between the old and new program.

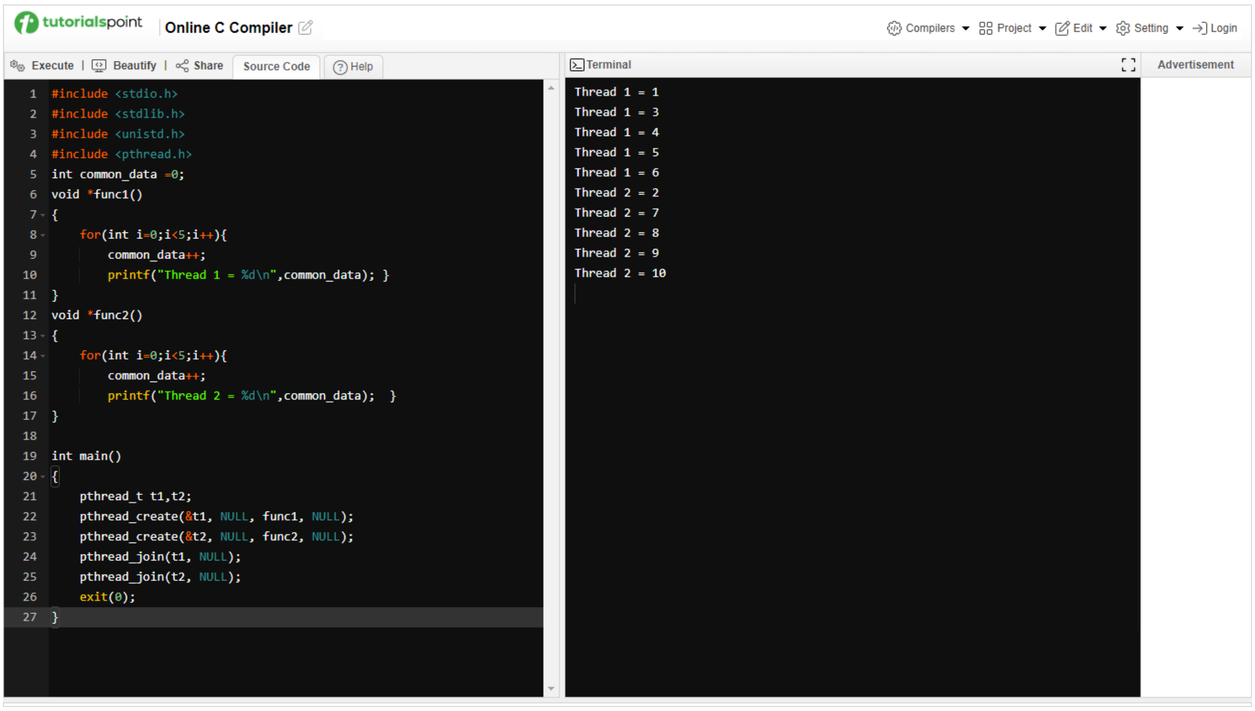
8. **What is the difference between process and threads?**

Process	Thread
An independent program with its own address space, resources, and execution context.	A lightweight unit of execution within a process.
Created by the operating system through system calls like fork() or exec().	Created by the process itself or by the operating system within a process.
Communication between processes usually involves inter-process communication (IPC), like pipes or sockets.	Threads share memory space and can communicate directly through shared variables or synchronization primitives.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mutex;
int common_data =0;
void *func1()
{
  for(int i=0;i<5;i++){
    pthread_mutex_lock(&mutex);
    common_data++;
    printf("Thread 1 = %d\n",common_data);
    pthread_mutex_unlock(&mutex);}
}
void *func2()
{
  for(int i=0;i<5;i++){
    pthread_mutex_lock(&mutex);
    common_data++;
    printf("Thread 2 = %d\n",common_data);
    pthread_mutex_unlock(&mutex);}
}

int main()
{
  pthread_t t1,t2;
  pthread_create(&t1, NULL, func1, NULL);
  pthread_create(&t2, NULL, func2, NULL);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  exit(0);
}
```

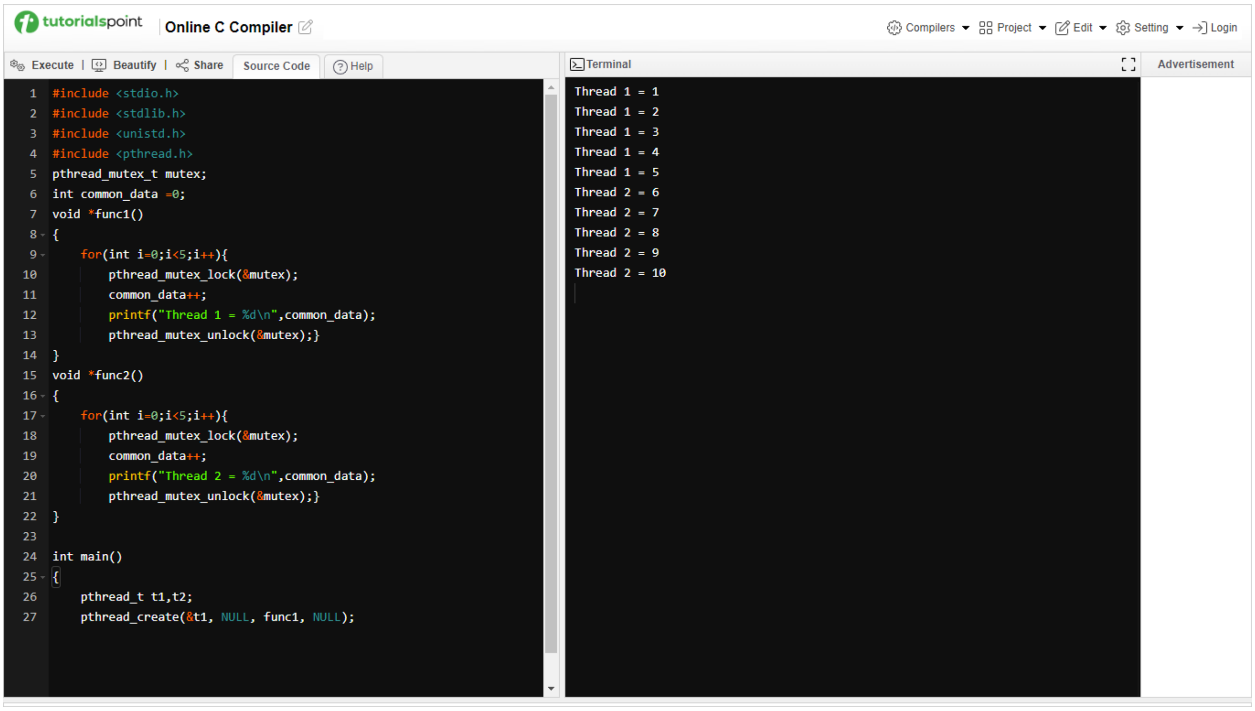
Without Mutex:



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
int common_data =0;
void *func1()
{
    for(int i=0;i<5;i++){
        common_data++;
        printf("Thread 1 = %d\n",common_data); }
}
void *func2()
{
    for(int i=0;i<5;i++){
        common_data++;
        printf("Thread 2 = %d\n",common_data);  }
}

int main()
{
    pthread_t t1,t2;
    pthread_create(&t1, NULL, func1, NULL);
    pthread_create(&t2, NULL, func2, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    exit(0);
}
```

```
Thread 1 = 1
Thread 1 = 3
Thread 1 = 4
Thread 1 = 5
Thread 1 = 6
Thread 2 = 2
Thread 2 = 7
Thread 2 = 8
Thread 2 = 9
Thread 2 = 10
```

With Mutex:



```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
pthread_mutex_t mutex;
int common_data =0;
void *func1()
{
    for(int i=0;i<5;i++){
        pthread_mutex_lock(&mutex);
        common_data++;
        printf("Thread 1 = %d\n",common_data);
        pthread_mutex_unlock(&mutex);}
}
void *func2()
{
    for(int i=0;i<5;i++){
        pthread_mutex_lock(&mutex);
        common_data++;
        printf("Thread 2 = %d\n",common_data);
        pthread_mutex_unlock(&mutex);}
}

int main()
{
    pthread_t t1,t2;
    pthread_create(&t1, NULL, func1, NULL);
```

```
Thread 1 = 1
Thread 1 = 2
Thread 1 = 3
Thread 1 = 4
Thread 1 = 5
Thread 2 = 6
Thread 2 = 7
Thread 2 = 8
Thread 2 = 9
Thread 2 = 10
```