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

Answer:

1. When user pressed ctrl+c then SIGINT() will be triggered which terminates the process.
2. When kill() system call is invoked with process ID & signal number, then the specified signal will be sent to the process
3. When CPU tried to execute an illegal instruction, then the SIGILL Signal gets generated.
4. When the program accesses the unassigned memory then SIGSEGV(Signal Segmentation Violation) will get generated.

2. List the gdb command for the following operations
    i) To run the current executable file
    ii) To create breakpoints at
    iii) To resume execution once after breakpoint
    iv) To clear break point created for a function
    v) Print the parameters of the function in the backtrace

Answer:

i) Run
ii) break <line_number> , break <function_name>, break <file_name>:<line_number>
iii) continue
iv) clear <function_name>
v) 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;
}
```

Answer: 2 2 2
2 2 2
2 2 2

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;
}
```

Answer: 1 4 2 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

Answer:

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

// Thread function to print "Hello"
void *printHello(void *arg) {
    printf("Hello ");
    pthread_exit(NULL);
}

// Thread function to print "World"
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;}
```

6. How to avoid Race conditions and deadlocks?

To avoid Race conditions, we can use synchronization mechanisms, thread safety, avoid shared mutable state,critical section protection using mutex.

To avoid deadlocks, we can use lock ordering, avoid nested locks, resource allocation hierarchies, avoidance of hold and wait, detection & recovery.

7. What is the difference between exec and fork ?

Fork() is used to create a new process called the child process which is an exact copy of the calling process/parent process. The child process has own address space but shares the same code, data space. Fork() in parent process returns the process ID, Whereas in child process it returns 0.

Exec() is used to replace the current memory space with a new program. It loads the new executable in the current process address space, overwriting the existing program and data.

8. What is the difference between process and threads.

1. **Process:**
    - A process is an instance of a running program. It consists of the program code, associated data, resources (such as memory and file

handles), and execution context (such as program counter, registers, and stack).

- Each process has its own separate memory space, which means processes do not share memory with each other by default.
- Processes are isolated from each other, meaning one process cannot directly access the memory or resources of another process without explicit inter-process communication mechanisms.
- Processes are heavyweight entities in terms of system resources because they require their own address space, which includes memory allocation, file descriptors, and other system resources.
- Processes are managed by the operating system's process scheduler, and each process runs independently and concurrently with other processes.

2. **Thread:**
    - A thread is a lightweight unit of execution within a process. Threads share the same memory space and resources of the process to which they belong.
    - Threads within the same process share the same code segment, data segment, and file descriptors. They can directly access shared memory and resources without the need for explicit communication mechanisms.
    - Threads within the same process can communicate with each other more efficiently compared to inter-process communication because they share the same memory space.
    - Threads are more lightweight than processes because they share resources with other threads within the same process. Creating a thread is typically faster and consumes fewer system resources than creating a new process.
    - Threads within the same process are scheduled by the operating system's thread scheduler, and they can execute concurrently with other threads within the same process.

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 declaration
void *thread_function(void *arg) {
    pthread_mutex_lock(&mutex);
    printf("Thread %ld is accessing the shared resource.\n", (long)arg);
    usleep(100000); // Sleep for 100 milliseconds
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int i;
    if (pthread_mutex_init(&mutex, NULL) != 0) {
        perror("pthread_mutex_init");
        return 1;
    }
    for (i = 0; i < NUM_THREADS; i++) {
        if (pthread_create(&threads[i], NULL, thread_function, (void *)(long)i) !=
0) {
            perror("pthread_create");
            return 1;
```

```
        }}
    for (i = 0; i < NUM_THREADS; i++) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join");
            return 1; } }
    pthread_mutex_destroy(&mutex);
    return 0;
}
```