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

**User presses ctrl+C:**

The interrupt signal SIGINT is generated when a user types the interrupt character (normally ctrl+C). This signal is sent to all processes in the foreground process group of the controlling terminal. By default, this causes the process to terminate.

**kill() system call is invoked:**

The kill() system call can send any signal to any process group or process. The exact signal sent depends on the arguments provided with the call. The most common use case is sending a SIGTERM signal to request the graceful termination of a process. If a specific signal is not specified, SIGTERM is the default.

**CPU tried to execute an illegal instruction:**

When the CPU detects that a process is trying to execute an illegal or undefined machine language instruction, it generates a SIGILL signal. Unless handled, this signal will usually result in the process being terminated and a core dump being created to allow the issue to be diagnosed.

**When the program accesses unassigned memory:**

If a program tries to read or write to an unassigned area of memory, it will receive the SIGSEGV signal, commonly known as a segmentation fault. The default behaviour when handling SIGSEGV is to terminate the process and create a core dump.

2) List the gdb command for the following operations

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

3) Every process (parent, children, grandchildren, etc.) will print "2 ". To determine the total number of "2 "s printed, we need to count the number of processes created. This program's control flow creates a total of 6 new processes (including the original parent process). Each of these processes will print "2 " once.

Therefore, the expected output will be "2 " printed 7 times (once by each of the 6 child processes and once by the original parent process). The actual order in which the "2 "s appear can vary depending on how the operating system schedules the processes.

4) Given this forking behaviour, the output will consist of multiple "1 ", "2 ", "3 ", and "4 ", but the order may vary because process scheduling is unpredictable. However, we can predict the number of times each number will be printed:

"1 " will be printed once.

"2 " will be printed once.

"3 " will be printed once.

"4 " will be printed by every process. There are 4 processes in total, so "4 " will be printed four times.

So, the output will have one "1 ", one "2 ", one "3 ", and four "4 "s, but the exact order will depend on the OS scheduler. For example, one possible output could be: 3 4 2 4 1 4 4 .

5)

Code:

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

void* printHello(void* arg) {
    printf("Hello ");
    return NULL;
}

void* printWorld(void* arg) {
    printf("World\n");
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    pthread_create(&thread1, NULL, printHello, NULL);
    pthread_join(thread1, NULL);
    pthread_create(&thread2, NULL, printWorld, NULL);
    pthread_join(thread2, NULL);
    return 0;
}
```

6)

## Avoiding Race Conditions:

Use synchronization mechanisms such as mutexes, semaphores, monitors, or atomic operations to ensure that only one thread can access the resource at a time.

Design the program in a way that access to shared resources is minimized or eliminated.

Employ thread-safe libraries that ensure internal synchronization.

## Avoiding Deadlocks:

Acquire resources in a fixed order to prevent circular wait conditions.

Use a try-lock mechanism instead of a lock, and if the resource is busy, release the previously acquired resources and try again.

Set a timeout for lock acquisition.

Use deadlock detection algorithms to pre-emptively avoid deadlocks.

7)

Code for example:

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        printf("This is the child process. PID = %d\n", getpid());
    } else if (pid > 0) {
        // Parent process
        printf("This is the parent process. PID = %d, Child PID = %d\n", getpid(), pid);
    } else {
        // Failed to fork
        printf("Failed to fork.\n");
        return 1;
    }

    return 0;
}
```

When fork() is called, the current process (parent) is duplicated into a new process (child). Both processes will start their execution right after the fork() call. The child process receives a PID of 0 from the fork(), and the parent receives the PID of the child process.

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // Child process
        printf("In child process before exec call. PID = %d\n", getpid());
        execlp("ls", "ls", "-l", (char *)NULL); // Replace child process with a new process image
        // The 'execlp' system call replaces the current process image with a new process image.
        // After the 'execlp' call, the following lines will not be executed in the child process
        // unless 'execlp' fails.
    } else if (pid > 0) {
        // Parent process
        printf("This is the parent process. PID = %d, Child PID = %d\n", getpid(), pid);
        wait(NULL); // Wait for the child process to complete
```

```
    } else {
        // Failed to fork
        printf("Failed to fork.\n");
        return 1;
    }

    return 0;
}
```

In this case, the exec() family of functions (in the example, execlp()) replaces the current process image with a new process image. That is, the child process stops running the current program and instead starts running the ls command. The original child process effectively becomes the ls command, and once the ls command completes, the child process terminates.

8)

## Process example:

Imagine you have two different applications running on your computer, such as a web browser and a text editor. Each application is a separate process with its own dedicated memory space, and they operate independently of each other.

## Thread example:

Within the web browser, you might have multiple tabs open. Each tab might represent a thread within the web browser process. These threads can share memory space and resources within the single web browser process, and they execute independently but can communicate with each other more efficiently than separate processes.

Here's a metaphorical example:

Consider a factory (process) where you have multiple assembly lines (threads). Each assembly line can create products using shared resources (memory). If one assembly line encounters an issue and has to stop, the other assembly lines can continue. However, if the entire factory has to shut down (process terminates), all assembly lines (threads) stop as well.

9)

Code:

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

pthread_mutex_t lock;

void* threadFunction(void* arg) {
    pthread_mutex_lock(&lock);
    printf("%s", (char*)arg)
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main() {
    pthread_t thread1, thread2;
    char* message1 = "Hello ";
```

```
    char* message2 = "World\n";
    pthread_mutex_init(&lock, NULL);

    pthread_create(&thread1, NULL, threadFunction, (void*)message1);
    pthread_create(&thread2, NULL, threadFunction, (void*)message2);

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

    pthread_mutex_destroy(&lock);
    return 0;
}
```