1. Which signals are triggered, when the following actions are performed.
    1. user press ctrl+C – SIGINT

    2. kill() system call is invoked – SIGKILL

    3. CPU tried to execute an illegal instruction – SIGILL

    4. When the program access the unassigned memory —SIGSEGV

==========

2. List the gdb command for the following operations

    1. To run the current executable file – run [args] or r[args]

    2. To create breakpoints at – break[function_name] or break[line_number] or b[file_name]:[line_number]

    3. To resume execution once after breakpoint – continue

    4. To clear break point created for a function – clear[function_name]

    5. Print the parameters of the function in the backtrace

       Backtrace info args[frame_no]

==========

3 OUTPUT :

2222222

==========

4 OUTPUT

24341414

==========

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>
void *hello (void *arg) {
        printf("Hello ");
        pthread_exit(NULL);
}
void *world(void *arg) {
        printf("World ");
        pthread_exit(NULL);
}
int main(){
        pthread_t  t1, t2;
        pthread_create(&t1, NULL, hello, NULL);
        pthread_join(t1, NULL);
        pthread_create(&t2, NULL, world, NULL);
        pthread_join(t2, NULL);
        return 0;
}
```

==========

6. How to avoid Race conditions and deadlocks?

To avoid race conditions :

we can use synchronization mechanisms like locks and semaphores to coordinate access to shared resources and ensure that only one process can access the resource at a time.

To avoid deadlock:

There are four necessary conditions for deadlock to occur. They are,

- Mutual exclusion
- No preemption
- Hold and wait
- Circular wait

In order avoid deadlock we can design the system in such a way that atleast one of the necessary conditions for deadlock does not occur.

==========

7. What is the difference between exec and fork ?

FORK:

- The fork system call is used to create a the child process, as a copy of the parent process.
- After a successful fork, both the parent and the child processes continue execution from the point of the fork call.
- child process and parent process have separate memory spaces and can run independently.
- The fork system call returns different values in the parent and child processes, which helps to distinguish between them.

EXEC:

- The exec system call is used to replace the current process with a new process.
- The newly created process replaces the memory space of the previous process.

==========

8. What is the difference between process and threads?

Process:

- A process is a program in execution. It has its own memory space, which includes data and resources.
- Processes are isolated from each other and they cannot directly access each other's memory space.
- Inter-process communication mechanisms such as pipes and shared memory are used for communication between processes.
- Processes are heavyweight in terms of resource consumption.

Thread:

- A thread is a lightweight unit of execution within a process. Multiple threads can exist within a single process and they can share same memory space.
- Threads within the same process can communicate directly with each other through shared memory.
- Threads share same resources of a process which can lead to synchronization problems.
- Threads are more efficient in terms of resource consumption as they share resources.

==========

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

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define NUM 5
pthread_mutex_t mutex;
int shared_variable=0;
void *threadfun (void *threadid) {
    int id=*(int *)threadid;
    pthread_mutex_lock(&mutex);
    printf("\nThread %d is accessing the shared variable.!",id);
    shared_variable++;
    printf("\nThread %d shared_variable = %d", id, shared_variable);
    pthread_mutex_unlock(&mutex);
    pthread_exit(NULL);
}
int main(){
    pthread_t threads [NUM];
    int threadid [NUM];
    int i;
    for(i=0;i<NUM; i++){
        threadid[i]=i;
        pthread_create(&threads[i], NULL, threadfun, (void
*)&threadid[i]);
    }
    for(i=0;i<NUM; i++) {
        pthread_join(threads [i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    return 0;
}
```