Directory Structure

- **Assignment_2**
  - **Problem_1** - Implementation of Exec and basic Signal Handling
    - main.c
    - SRHandler.c - Functionality for SR, on compiling will generate the SRHandler Executable for exec for SR
    - STHandler.c - Functionality for ST, on compiling will generate the SRHandler Executable for exec for SR
    - makefile
  - **Problem_2** - Implementation of problem 2 and modification of stack using inline assembly.
    - changes.txt - File Containing the changes made to linux kernel in the text format along with path of files modified.
    - demo.c - A demo program to check the working of the system call.
    - patch - diff file containing the patch for linux-5.14.3
    - makefile
    - linux-5.14.3.tar.gz
    - .config
  - readme.pdf

Problem_1

Input
No Input needs to be given

Output
We will receive a Random Number generated by RDRAND inline assembly
We will also receive the time using the RDTSC inline assembly instruction
Both the instruction will keep on producing the output until the program is exited.

Expected Output

```
❯ make
gcc SRHandler.c -no-pie -o SRHandler
gcc STHandler.c -no-pie -o STHandler
gcc main.c -o main
❯ ./main
Random Number: 1104462900
S1 received a SIGTERM CALL
Random Number: 4236975554
S1 received a SIGTERM CALL
Alarm form ST
Time Since Computer Started 0 Hours 10 Minutes 58 Seconds
S1 received a SIGTERM CALL
Random Number: 2503110769
S1 received a SIGTERM CALL
Alarm form ST
Random Number: 907816180
Time Since Computer Started 0 Hours 11 Minutes 2 Seconds
S1 received a SIGTERM CALL
Random Number: 3921262695
```

Compiling the Program
For compiling the program a makefile is supplied in the program directory.
First run make in a linux terminal inside the program folder **Problem_1**
Then run the **main** file using **./main**.

System Calls Used
- signal - This system call is used to handle a specified signal using our custom signal handler.
  In case of SR and ST , signal is used to handle SIGALRM via alarmHandler() function which performs getting a random number and getting the time elapsed respectively.
  In case of S1 , signal is used to handle SIGTERM via sigtermHandler().
- execl - This system call is used to replace the current process image with the specified executable along with the arguments supplied.
  In SR and ST we use them to replace the process image with SRHandler and STHandler executable respectively
- fork() - This system call is used to create a child process.
  It is used to create S1, ST and SR from the main function.

- Waitpid - It is used to wait execution till the process with specified PID exits.
  We use it to make the main process wait for S1 to exit to prevent main to exit before we terminate the program.

Description of How the program works

- First, we create a child program S1 using fork system call and store the PID of S1 in pidS1 local variable
- Now when we are in S1 then pidS1=0, and we create an infinite loop inside S1 along with a signal system call to handler any SIGTERM signal with our custom signal handler sigtermHandler(). It prevents S1 from exiting till the program is terminated and helps to handle SIGTERM signal again and again.
- When we are inside the main process then the pidS1>0 and we create another child process pidSR using fork and store pid of SR in local variable pidSR.
- If we are in SR then we make an exec system call and launch SRHandler executable, which our makefile generates automatically along with S1id which is the pid of S1 for further use.
- Similarly main process forks another child process SR ,storing its PID in pidST and then again calling exec similar to SR.
- Now we make main process to wait for S1 to exit using waitpid to prevent it from exiting.
- Now inside SR and ST we implement a timer for 2second and 3 seconds respectively using setitimer() system call. We make use of itimerval structure for making an setitimer system call, it_value.tv_sec and it_value.tv_usec are set appropriately.
- We then run an infinite while loop to prevent SRHandler and STHandler from exiting and keep running the timer till user exits the program.
- In case of SRHandler , the SIGALRM signal is handled and then using the inline assembly RDRAND , we move a random number to a variable randomNum. Then we raise a SIGTERM call to S1 using kill which handles and then print the number.
- In case of STHandler , the SIGALRM signal is handled and then using the inline assembly RDTSC we get the clock cycles elapsed since the computer was started. rdtsc uses the CPU timestamp counter introduced on Intel CPU's.
- Now we then divide the elapsed clock cycle by 2400000000 to get the elapsed time assuming the frequency of the computer is 2.4GHz.

Note: The frequency might be variable for different CPU's and may be different for different times. I have assumed 2.4GHz as it is the Base Clock of my CPU Intel I5-9300H.

- We then convert the seconds time to Hour and Min and seconds respectively.
- We then send a SIGTERM call to S1 and then prints the elapsed time.

Problem_2

Input

No input needs to be given for running the program. As mentioned I have a hardcode 3x3 2d Float array.
The hardcoded array is {{1,2,3},{4,5,6},{7,8,9}}

Output

Firstly the original source array would be printed
Then the original destination array initialised to -1 would be printed.
Then destination array after the system call would be printed.

Expected Output

```
[kern@artixcse231 ~]$ cd build
[kern@artixcse231 build]$ cd -
/home/kern
[kern@artixcse231 ~]$ cd OS
[kern@artixcse231 OS]$ gcc demo.c -o demo
[kern@artixcse231 OS]$ ./demo
The source array is :
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
The destination array before syscall is :
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
-1.000000 -1.000000 -1.000000
The destination array after the system call is :
1.000000 2.000000 3.000000
4.000000 5.000000 6.000000
7.000000 8.000000 9.000000
[kern@artixcse231 OS]$
```

System Calls Used

- syscall - This system call is used to call a particular system call from the syscall table using the syscall number and pass the required arguments to the system call
- kernel 2d memcpy() - This is our custom syscall with syscall number 449. It takes 3 arguments as (void*)source array pointer, (void*)destination array pointer and unsigned long data size.
  It copies the data of the source array pointe to the destination array pointer.

Compiling the Kernel

- First , go to Artix Home Directory and then mkdir Linux-Kernel.
- Then give the command or use the supplied linux-5.14.3 tarball file lynx https://cdn.kernel.org/pub/linux/kernel/v5.x/linux-5.14.3.tar.xz
- Then extract tarball: tar -xvf linux-5.14.3.tar.xz
- Then paste the supplied .config inside the extracted directory.
- Now put the patch file insider Linux-Kernel directory.
- Now go to the linux source by cd linux-5.14.3
- Now apply patch by the following command
- patch -p1 -R <../patch
- The modification for the custom system call has been added to the kernel source.
- Now we need to compile the kernel using following commands:
- make
- sudo make modules_install
- sudo cp arch/x86_64/boot/bzImage /boot/vmlinuz-linux-5.14.3
- sudo cp System.map System.map-5.14.3
- sudo mkinitcpio -k 5.14.3 -g /boot/initramfs-linux-5.14.3.img
- sudo grub-mkconfig -o /boot/grub/grub.cfg
- sudo reboot

Compiling the Program

- After compiling and configuring the linux kernel do the following
- For compiling the program inside the Problem 2 folder give the command make
- Then run the demo executable using ./demo

Description of how the program works

- We need to implement a custom system call.
- So we implement the system call in the sys.c file located at path linux-5.14.3/kernel/sys.c
- We first declare our system call using SYCALL_DEFINE macro and 3 arguments as (void*)source array pointer, (void*)destination array pointer and unsigned long data size with syscall name as kernel_2d_memcpy. The complete instruction is SYSCALL_DEFINE3(kernel_2d_memcpy,void*,src,void*,dst,unsigned long,dataSize)
- Now we make a temp float array to copy data from user space to kernel space as float kernArr[8][8]
- Now we copy the data from user space to kernel space using the unsigned long a =__copy_from_user((void*)kernArr,src,dataSize);

Here we use __copy_from_user kernel space function with source, destination pointers and size of data to copy.

- Now we copy the data form kernel space to user space using the unsigned long b =__copy_to_user(dst,(void*)kernArr,dataSize);
  Here we use __copy_to_user kernel space function with appropriate arguments.
- Hence the data of the source pointer is successfully copied to the destination pointer. We send the pointer to the source and destination array as a void pointer to the system call. Also the temp kernel space array is passed as a void pointer to the __copy_to_user and __copy_from_user.