Question 1 Documentation

Exec Family System Call and Basic IPC using Signals API

Program Logic & Flow:

1) Logic of main.c

In this program, the main.c is first executed. It performs a fork() to create the child S1. Now, in the body of S1, I have registered the SIGTERM handler for S1 using sigaction with the flags set as SA_SIGINFO. This is to signify that signal information would be received with the signal. The handler is named sig_handler(). In the handler, we retrieve the pid of the sender using siginfo->si_pid, and according to this pid, we can perform the necessary actions of displaying the random number or time string, depending on whether E1 or E2 had sent the SIGTERM signal. To be able to differentiate this, I stored the pids of SR and ST (when they were available after forking) into shared global variables child_sr_pid and child_st_pid, which were created using mmap(), which could be used to compare with the pid of process that sent the signal.

In the parent body, after forking S1, I stored the pid of S1 in the variable pid_s1.

In this body, now we perform another fork() to produce SR child. In the child SR body, I stored the pid of this child in the shared global variable pointer, child_st_pid. We would transfer the rest of the flow in E1 by using the execv() system call and passing the pid of S1 as the argument (by converting it into a string using snprintf() and later resolving it back as integer in E1).

Similarly, in the parent body I performed yet another fork() to produce ST child, and in this body, I stored the pid of this child in the shared global variable pointer, child_sr_pid. We would transfer the rest of the flow in E2 by using the execv() system call and passing the pid of S1 as the argument (by converting it into a string using snprintf() and later resolving it back as integer in E2).

After this we keep running an infinite loop in S1.

2) Logic of E1.c

In the main body, we resolve the pid that is received as a string from the execv() call and convert it to int and store it into a global variable pid_s1. We set-up to catch the SIGALRM signal and handle it using sig_handler() using the signal() system call. Then, we set the value and interval as 1 second and use the setitimer() system call with the parameter ITIMER_REAL, so that SIGALRM signal is sent at every interval of 1 second.

In the sig_handler(), we generate the random number by calling our rdrand() function passing the address of variable that should store the random value, and the function generates a random number using the inline assembly instruction RDRAND. We then assign the sival_int attribute of the sigval union to this random number. Finally, we call the sigqueue() system call that sends the SIGTERM signal to the pid denoted by pid_s1 (S1 process' pid) with the union 'sigval' as the data to be passed with sigqueue().

3) Logic of E2.c

The main body of E2 also does exactly the same as described above for E1.c.

In the sig_handler(), we call our rdtsc() function. This function gets the No of Clock cycles passed since the CPU reset using inline assembly and the command RDTSC. The value gets stored in EDX:EAX registered so we get the value appropriately by left shifting 'hi' by 32 and ORing this value with 'lo'. The function then returns this value and it gets assigned to 'tsc' variable. To get a time in human readable format, I divided the number of clocks by my CPU frequency (obtained in WSL using lscpu), which gives us the time passed (seconds) since the CPU reset. This can be converted to days:hrs:mins:seconds format with an easy computation as present in the code. I have stored this human readable string in str, which is a variable that can be shared across processes, and stored also the 'str' pointer in the union 'sigval' inside 'sival_ptr'. Finally, we call the sigqueue() system call that sends the SIGTERM signal to the pid denoted by pid_s1 (S1 process' pid) with the union 'sigval' as the data to be passed with sigqueue().

MakeFile:

To generate output executable & run the program, the command is 'make'
To clear the generated .o files, the command is 'make clean'
To kill the running processes, the command is 'make kill'