Αντωνης Καλαμακης SDI1800056

to run the program we need to compile it with the help of a Makefile.
after that, we will need 3 shells.
one for ./P1, one for ./P2, one for ./CHAN
please execute processes with that order.
P1 is first sending and P2 is first receiving.

Documentation:
we have 5 different processes based on a client-server-client technique

the clients is p1, p2
and the server is chan

also we have the processes enc1 and enc2 which are "hidden" processes, started by p1 and p2 respectively with the help of fork to duplicate the process and then execvp to run encX's code.
p1 and p2 have very similar source code, also enc1 and enc2 has very similar source code the only difference is that for the semaphores and the shared memory we have different labels.
Also one main difference between p1 and p2 is that at p1 at first we send and at p2 at first we receive.
For every source code provided, the first segment of the code is the initialization part in which we create the semaphore and shared memory variables and we check if they return any type of error, if they do there is a problem with the order that we run the processes or the machine wont allow us to use semget or shmget.
Because of the fact that in this specific implementation we create semaphores in the pX and encX processes, first we have to run ./P1 and ./P2 and then run ./CHAN.
the processes pX initialize shared semaphores and memory with encX (before encX is run) and the processes encX initialize shared memory and semaphores for the chan process.
In my implementation i use steps to see in which state my programs are.

For PX.c:
i have 2 states (0,1) in my switch-case inside the infinite loop (so the program runs infinitely).
the case 0 of the switch-case is when we send a message into encX to be transmitted into the other p process the case 1 of the switch-case is when we wait for the other process to send a message and to receive it so it can get printed.
For organization purposes I've used functions send_message and receive_message for that purposes.

its good to mention that inside lib.c/lib.h i have some functions and structs that help me to make things quicker in terms of up/down the semaphores and returning multiple values from the functions struct rv (returned_value) is used to pass to the main the next step, the input given from the user and from enc, the confirmation number(don't need it in process pX) and the hash value(don't need it in process pX).

generally in my implementation i use a flag-type semaphore to sync who will use the semaphore first and then i use a semaphore to lock other processes out of critical section of the code.
that's why I am using flags as variable names inside the functions.
for example if i need p1 to use the critical section ALWAYS first and then enc1 to use it, i simply block the enc1 with a down-semaphore_flag then i down-normal_semaphore inside p1 and then i up-flag_semaphore to release the enc1 from waiting the flag and start waiting for the normal_semaphore to go up for each function returns (inside a struct or just as an int) a step value to give into main, this is used to continue to the next step.

inside send_message:
i simply wait for user to give output and then release the normal_semaphore so encX would wait for user input too I've used fgets because i couldn't get scanf to easily input multiple words with '\n' at the same time. then copy the data to the shared memory and up the normal_semaphore.

inside receive_message:
is the same technique but now the pX process is waiting for the flag to go up and then will try to make down the normal_semaphore(so encX will always use the critical section first).
After that if a user sends a TERM message(sending or receiving either way) after each step it checks if its TERM to finish the loop and wait for the kid to finish so it can detach and RMID the shared memory and semaphores if no TERM message is given, program continues to next loop and goes to next step.

For ENCX.c:
because of the fact that we have initialized every semaphore/shared memory used in pX we just need to get access to it so we don't need to IPC_CREAT the semaphores and shared Memory
but for the shared components that we use into chan we have to initialize them so chan will find them and use them to.
now we have 6 steps inside our switch-case, one for each situation we might will be.
as said before the difference between enc1 and enc2 switch-case is that in enc1 the first ever step is to receive from p but in enc2 the first ever step is to receive from chan, that is done with the help of changing the step value when initializing int step before the while loop.
To check for TERM message, if we sending the TERM we wait for confirmation(irrelevant, if the program sees TERM it always have correct confirmation) and if we are receiving the TERM we check it after we send the message to the pX.

For receive_from_p:
i simply use the technique mentioned before, at pX, for the flag-semaphore and simply store the input data to a global variable so i can access them inside my other functions easily.

For send_to_chan:
Here i use a info_struct structure so i can store the message and its hash value easily.
at first i create the hash value of the message and then try to occupy the semaphore to use the critical section of the function. Then i simply copy the data and then release the semaphore.

For wait_confirmation_from_chan:
Im waiting the chan to have ready the confirmation message and then when i receive it i check if the message transferred successfully or not. if yes i continue with the next step, if not i go back to send_to_chan step so it will re-transmit the message(its stored inside input_from_p so no need to re-request the message from user).

For receive_from_chan:
here we waiting for chan to have the data ready for us to receive it and to store it inside our global variable.

For confirm_to_chan:
first of all we create a hash value from the input given from chan, then we let know chan that we are at the critical section so it will wait for us to stop using it, then we need to check if the message is a TERM message, if yes no need to check the hash values, just return a correct confirmation message and go to next step if its not a TERM message we have to memcmp the two hash values if they are the same, if yes no type of corruption has happen so set as confirmation message the correct one

and return the program to go to the next step if not we have to request re-transition so we do that and go again to the receive_from_chan step.

For send_to_p:
finally if every thing is ok with the message we occupy the semaphores and set the input message to the shared memory with pX.

For CHAN.c:
First of all here we have to receive (if provided) an argument for the probability of the corruption. the provided value must be from 1 to 99 if not the program returns an error message, if more than one arguments are provided it also returns an error message. If there are no arguments, default's probability value is 2.
To give the correct probability value you must know this:
to corrupt a message (each and every one of it's letters), the random value that the rand() function will give must be lower than the probability given from the user(or by default).

After that we connect to the semaphores and shared memory that the other processes shared with us by creating and initializing them first and then we start the "main" section of the program, the switch-case inside the while loop.

here its great to mention that if you define DEBUG at the beginning of the program some debug-friendly messages will get printed so it will make easier to see in which step inside our program we are.

As said before we have a similar with every other process type of implementation. Each case inside our switch-case is a step that we need to go through a retrieve-transmit scenario. because of the fact that P1 is the first user that will transmit a message, we start with receive_from_enc1 then we send_to_enc2, then receive confirmation and pass it to enc1 and then do the same thing with data provided from P2.

For receive_from_enc1:
here we wait for enc1 to provide a message(by waiting for the flag to go down and then wait for critical section of enc1 to be done and use it) and when we do we copy the message to a temp variable and print the message(for debug purposes, servers shouldn't view client's messages :)) after that we set the next step value and we return temp to use it in the later functions.

For send_to_enc2:
here before everything we have to 'corrupt' the message, if its not TERM(no need to do that if its TERM), we go to every letter of the message(everything but the ENDL) and check if the rand() (from 0-100) is lower than the probability given. if yes we change the letter to a random letter so its 'corrupted'. after that we let know the enc2 that we are ready to transmit the message, occupy the semaphores and set the message to the shared memory and finally continue with the next step.

For wait_confirmation_from_enc2:
After sending it we need to wait for confirmation from enc2 so we can pass the data to the next step.

For send_confirmation_to_enc1:
here we send the confirmation to enc1 and check if the confirmation is correct or not, if not we need to change the flow of our program so we return a the correct step value to repeat the process.

After that the next functions are repeats of the other functions explained before, but now is with enc2 as sender and enc1 as receiver.