**IMPLEMENTATION DETAILS - AYUSH JAIN [jain207@purdue.edu]**

*Specific questions asked in lab2 for implementation are at the end*

Steps taken for the implementation

1. Two additional states PR_MYRECV and PR_MYRECTIM are introduced as process state number 8 and 9 in process.h
2. Additional data is included in struct procent like :
    - umsg32 buffer[BUFFERSIZE];
    - int32 head;
    - int32 tail;
    - int32 wait_count;

- BUFFERSIZE = 20 as per the lab requirement
- Head and tail are used to maintain the circular buffer which has been implemented as FIFO queue so that messages are received in the order in which they are enqueued when they are sent by single/multiple sender.
- Circular queue is implemented in the same way as lab1
- Wait_count is used to keep track of the total number of messages that the process wants to receive and only put it on the ready queue when it has been sent the correct number of messages.
- Head and tail are initialized to -1 for each process entry in process table including null process in initialize.c

3. For myreceive() function call, the wait count is set to 1 and current size is checked.

4. For myreceiven() function call, the wait count is set to the argument that is passed.
    - If current size of receiver's buffer is greater than wait count, the message is received.
    - Else it's state is changed to MYRECV and blocked by resched() until a message is sent to the receiver. (In both cases)

5. When a message is being sent by the sender using mysend() or mysendn(), it checks the receiver's buffer if it's less than maximum size (i.e. < 20) and message(s) is enqueued. Also, it is checked in mysend/mysendn that the current size of buffer is greater or equal to the wait count of receiver. In case the condition is true, the receiver is unblocked to receive the sent messages.

    Thus, the wait_count helps us to unblock the receiver only when required number of messages are present in the queue instead of continuously checking by receiver in a while loop.

6. The interrupts are disabled at the beginning of system call and restored before returning from the call.(in both successful / error cases). This ensures exclusive access to the global data structures like process table.

7. Error can be due to invalid process id of receiver.or when mysend() tried to send message to a full buffer.

8. The existing systems are not affected as 2 new states are introduced and buffer is entirely disjoint from the existing variables that are used in existing calls like send, receive etc.

**EXTRA-CREDENTIAL EXPLANATION**

The myrecvtime system call blocks and is set to an additional new state PR_MYRECTIM. It returns only if one of the two conditions have been met:

- The number of messages specified in msg_count have been received by the process from either mysend or mysendn. This is ensured by checking the wait_count of the receiver process table entry and comparing it with the current size of queue. If sufficient messages are not present, then the receiver sleeps with a timer of maxwait milisec else it receives all the messages.

- A timeout of maxwait has occurred: In this case, any message that have already been sent (using mysend or mysendn) are returned in the msgs array as insertd call awakens the process and receives as many messages that are present in the buffer even if they are not equal to wait_count.

**TEST CASES** ( Additional explanation in comments for each test case method in main.c)

- There are 17 test cases to test the functionality of all 5 new system calls individually as well as inter-mingled with each other.
- Comments are written along with each test case method in main.c that check all the requirements asked in lab2.

- Test Cases [1-5] involve mysend and myrecieve() calls
- Test Cases [6-14] involve mix of all system calls except myrecvtime() call
- Test Cases [15-17] involves myrecvtime() call as well

**QUESTIONS ASKED IN LAB2**

How does your solution guarantee messages are received in the order in which they were sent?

1. By implementing circular queue. Head and tail are used to maintain the circular buffer which has been implemented as FIFO queue so that messages are received in the order in which they are enqueued when they are sent by single/multiple sender.
2. Buffer_count, head and tail all are added in the process entry structure .

Describe your test cases. How to they ensure that the system calls correctly meet the requirements?

1. Done in the above section as well as comment for each test case method ensuring all the requirements

What modifications would need to be made to allow for a truly unlimited number of messages to be sent to a target process? Are these modifications practical?

1. In order to have unlimited messages be sent to a target process, the buffer size would have to be increased from 20 to a very large size.
2. This modification is not practical as the process table entry for each process would increase by a very high amount. As a result, the size of process table would be very large and we will be unable to allocate the required amount of memory for the process table. That is why, the buffersize has been restricted to 20 in our assignment.