Computer Networks Lab
Assignment 5

Design Report

Group Members:
20CS10085 Pranav Mehrotra
20CS30065 Saransh Sharma

Report

- **Data structures for send and receive tables:** send and receive tables have been implemented in the form of queues (using a linked list). Two data structures have been used to implement the queue using a linked list:
  - **queue**: data structure queue represents one node of the linked list.

```
typedef struct _queue
{
   struct _queue *next;
   char *data;
   int len;
}queue;
```

    next is a pointer to the next node of the linked list. **Note: next == NULL** indicates the end of the linked list.
    data is a char* variable that would store the message that needs to be sent/received.
    len variable would be used to store the length of the message stored in the data field of the node.

    Therefore, one node of the send/ receive queue would store the pointer to the next node of the list, data that needs to be sent/received along with the length of the message stored in this node.

  - **queue_head**: queue_head data structure would store pointers to the beginning and end of the queue.

```
typedef struct _queue_head
{
   queue *front;
   queue *rear;
```

```
 int curr_size;
 int max_size;
}queue_head;
```

The **front** is the pointer to the first node of the linked list.
The **rear** is the pointer to the last node of the linked list.
curr_size filed stores the current number of messages/nodes in the queue excluding the dummy node.
max_size denotes the maximum number of nodes that are allowed to be present in the queue. (max_size is set to 10 in the question).

If the front and rear point to the same node that indicates that the queue is empty.
The queue will always be initialized by a dummy node, i.e. front and rear point to a dummy node and front->next and rear->next are NULL pointers. Add and remove nodes are explained later.

- **Global Variables:**
  - queue_head ***send_queue**: send queue is declared a global variable to share amongst threads.
  - queue_head ***receive_queue**: receive queue is declared a global variable to share amongst threads.
  - int **my_type** = 0: my_type is 1 when the socket is of type SOCK_MyTCP else is set to 0 by default.
  - int **curr_sockfd** = -1: curr_sockfd would be a global variable used by threads to send/receive data from the socket of the server/client.
  - pthread_t **R_thread**: R thread
  - pthread_t **S_thread**: S thread

- pthread_attr_t **R_attr**: attribute variable for R Thread
- pthread_attr_t **S_attr**: attribute variable for S Thread
- pthread_mutex_t **send_mutex**: lock to access send queue
- pthread_mutex_t **receive_mutex**: lock to access receive queue

- **Functions:**
  - **Helper functions:**
    - **min:** The function returns the minimum of the two parameters passed to it as the argument.
    - **recv_num:** The function is used to receive 4 (= sizeof(int)) bytes of data (i.e. One integer) in chunks from socket. The exact use of this function will be explained in detail in the recieve_expr function.
      `int recv_num(int newsockfd);`

  - **Queue helper functions:**
    - **push:**
      `int push(queue_head *head, char *s, int len){}`
      The push function is used to push a node to a queue i.e. add a node to the queue.
      A new node is created, and data and len fields are set. The new node is added at the rear end of the queue i.e. new_node would be the new rear end of the queue. Therefore new_node->next = NULL and the next pointer of the queue's rear node would now point to the new node instead of NULL. The rear pointer of the queue and the current size of the queue is updated. Push function returns 0 if the node could be successfully pushed else returns 1 in case the node could not be pushed due to no space available i.e. curr_size == max_size.

- **pop:**
  `char *pop(queue_head *head, int *len){}`
  The pop function is used to remove a node from the queue. The first node after the dummy node i.e. front->next would be popped out, front pointers would be updated and the data stored in the node is returned. The len variable passed as a pointer argument to the function is updated to the length of the data stored in the node. In case no node was present to be popped out, the function returns NULL.

- **Socket programming functions:**
  - **my_bind**: wrapper around bind call. Binds the socket with some address port using the bind call.
    `int my_bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

  - **my_accept**: wrapper around accept call. Accepts a connection on MyTCP socket by making a TCP connect call. Global variable curr_sockfd is set to the sockfd passed as a parameter to the my_accept call.
    `int my_accept(int sockfd, struct sockaddr *restrict addr, socklen_t *restrict addrlen);`

  - **my_connect**: wrapper around connect call. Opens a connection through the MyTCP socket using connect call on the TCP socket. Global variable curr_sockfd is set to the sockfd returned by connect call.
    `int my_connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);`

- **my_listen**: wrapper around listen call.
  int my_listen(int sockfd, int backlog);

- **my_socket**: if the function is called with type SOCK_MyTCP, a TCP socket is created, and send and receive queues are created (i.e. dummy nodes, curr_size, front, and rear pointers set appropriately).

  int my_socket(int domain, int type, int protocol);
  send_mutex (lock to access send queue) and receive_mutex (lock to access receive queue) are initialized, and attributes for R and S threads are initialized. Since, we want my_close() to be non-blocking, hence the threads are created using detached state attribute, hence no thread needs to wait for the threads to join. The socket returned by socket call is returned.

- **my_close**: if the socket was of type SOCK_MyTCP, the function sleeps for 5 seconds to prevent race between threads, after sleep, threads R and S are killed using pthread_cancel(). The threads call cleanup_R and cleanup_S functions before cancellation. These functions are explained later.
  int my_close(int fd)

Thread functions:

- **R:** R thread upon creation will start execution of this function. The cleanup function, to be called when the R

thread is killed, is registered. The thread can be canceled only when curr_sockfd is set to -1 (i.e. either my_close has been called or before the accept/connect call) or no message is received i.e. timeout. The thread waits for the incoming message using a poll, In case of timeout, check for pending cancellation requests and continue polling. In case of an incoming message, receive the expression in chunks using recieve_expr, if the length is non-zero, acquire the lock for receive queue, and push the message to the queue. In case of no space available in receive queue, wait for my_recv to delete an existing message in the queue (i.e. reattempt after a sleep of 10,000 microseconds).
void *R (void *arg)

- **S:** S thread upon creation will start execution of this function. The cleanup function of S is registered. The thread sleeps for 10,000 microseconds (T = 10,000), checks for cancellation requests, wakes up, and checks for pending messages in send queue by acquiring a lock on send queue and popping the message out of the queue. The message is then sent using the send_expr function.
  void *S (void *arg)

- **cleanup_R:** The function is called when a cancellation request to the R thread has been processed by the R thread. The receive queue is emptied i.e. nodes are freed and the recieve_lock is destroyed. This function is called when my_close calls to kill the R thread.
  void cleanup_R (void *arg)

- **cleanup_S:** The function is called when a cancellation request to the S thread has been processed by The S thread. The send queue is emptied i.e. nodes are free and the send_lock is destroyed. This function is called when my_close calls to kill the S thread.
  void cleanup_S (void *arg)

- **Helper functions for Send/receive:**
  - **send_expr:** this function is used to send a message in chunks. To inform the receiver end about the end of the message or boundary of the message, the first 4 bytes of the message to be sent using send calls are reserved for the length of the message i.e. every message to be sent using my_send() function will be preceded by message length at the beginning.
    void send_expr(int newsockfd, char *expr, int size){}
    send_expr first sends the length of expr (stored in the size variable) and then expr is sent in chunks.

  - **recieve_expr:** The function first calls recv_num to receive the length (recv_size) of the message sender has sent. Then the message is received in chunks using recv call until the number of bytes received equals to recv_size. The received expression is then returned and the parameter len_recieved is updated to the length of the message received.
    char *recieve_expr(int newsockfd, int *len_recieved)

- **my_send:**

  ssize_t my_send(int sockfd, const void *buf, size_t len, int flags)

- The function takes in a buffer buf which stores the content to be sent, len which contains the length of the message to be sent, and flags as the argument.
- If the socket is not of type SOCK_MyTCP, the parameters will be passed to send function.
- If the socket is of type SOCK_MyTCP, the send function makes a local copy of the buffer. It then acquires a lock (**pthread_mutex_lock(&send_mutex)**) for send queue and tries to push (**push()**) the message along with the length in the send queue. The lock is then released (**pthread_mutex_unlock(&send_mutex)**).
- Upon successfully being able to insert the message in send queue the function returns the length i.e. number of bytes occupied by the message.
- In case of no available space in send queue, the function sleeps for 10,000 microseconds and then tries again until it can successfully able to push the message to the send queue.
- **send_queue, a** queue globally declared for storing messages to be sent, is accessed by the function. Thread S routinely checks for pending messages and sends the message if any from send_queue.

- **my_recv:**

  ssize_t my_recv(int sockfd, void *buf, size_t len, int flags)

- The function takes as input buffer buf wherein the received message would be saved along with the len

variable that stores the length of the buf buffer as arguments.

- If the socket is not of type SOCK_MyTCP, the parameters will be passed to recv function.
- In case the type of socket is SOCK_MyTCP, the function acquires a lock on receive queue (**pthread_mutex_lock(&receive_mutex)**), and tries to pop the message from receive queue. On success, the function releases the lock (**pthread_mutex_unlock(&recieve_mutex)**) the message is copied into the buffer and the length of the message is returned by the function.
- In case of no message pending in receive queue, the function sleeps for 10,000 microseconds and then tries again till it finds a pending message in receive queue.
- **receieve_queue**, a globally declared queue to store pending messages to be received is accessed by the function.