

4. Synchronization Problem: Message Passing [12 points]

Message passing implementations come in many forms. In one variant, called a *rendezvous*, both sending and receiving a message are blocking operations. That is, a thread calling the `msg_send()` function will block until the recipient calls `msg_rcv()`. Similarly, if a thread calls `msg_rcv()`, it must block until some other thread calls `msg_send()`. When sending a message, the intended destination must be specified, however, messages can be received from any source, with the id of the sender returned along with the message.

To simplify the problem, we assume that messages are of a fixed length, `MSG_LEN`, and that thread ids are chosen from a small range of integer values, `[0..TID_MAX-1]`. We define an array of *mailboxes*, one per thread, indexed by thread id, to help implement the functions.

```
struct mailbox {
    char *dest_buf;      /* set by receiver */
    int sender_id;      /* set by sender */
    struct semaphore mutex; /* initialized to 1 */
    struct semaphore recvr_ready; /* initialized to 0 */
    struct semaphore sender_done; /* initialized to 0 */
};

struct mailbox mailboxes[TID_MAX];
```

(i) [8 points] Complete the implementation of the message passing functions `msg_send()` and `msg_rcv()` on the following page. Remember that a thread may receive messages from multiple senders. [You should at least read the partial implementation before answering the following two questions.]

(ii) [2 points] What is the main *advantage* of using blocking sends and receives in this way?

No intermediate storage is needed to hold sent messages until the receiver is ready to receive them.

Also acceptable: The sender knows the receiver has the message when it returns from send (or else it has any error immediately, although the functions on the following page don't return errors), which may simplify programming since we don't need to check separately that the receiver got the message.

(iii) [2 points] What is the main *disadvantage* of using blocking sends and receives?

The sender must wait for the receiver before the message can be sent, instead of going on to other work. The receiver also waits, but this may be less of a problem if we assume the receiver needs to get the message before it can continue with its work anyway.

```
/* Add the necessary calls to P() and V() on the mailbox semaphores.  
You do not need to add code to every blank space. */
```

```
void msg_send(char *msg, int recvr_id) {  
    struct mailbox *mbox = &mailboxes[recvr_id];  
  
    /* lock out other senders to same receiver */  
    P(mbox->mutex);  
  
    /* wait for the receiver to set dest_buf */  
    P(mbox->recvr_ready);  
  
    memcpy(mbox->dest_buf, msg, MSG_LEN);  
  
  
  
    mbox->sender_id = get_current_thread_id();  
  
    /* signal that sender is done copying message and setting id */  
    V(mbox->sender_done);  
  
    /* allow other senders to send their messages to same receiver */  
    V(mbox->mutex);  
}  
  
void msg_rcv(char *msg, int *sender_id) {  
    struct mailbox *mbox = &mailboxes[get_current_thread_id()];  
  
  
  
  
  
  
    mbox->dest_buf = msg;  
  
    /* Signal sender that receiver is ready to get message */  
    V(mbox->recvr_ready);  
  
    /* Wait for sender to send message */  
    P(mbox->sender_done);  
  
    *sender_id = mbox->sender_id;  
  
}
```