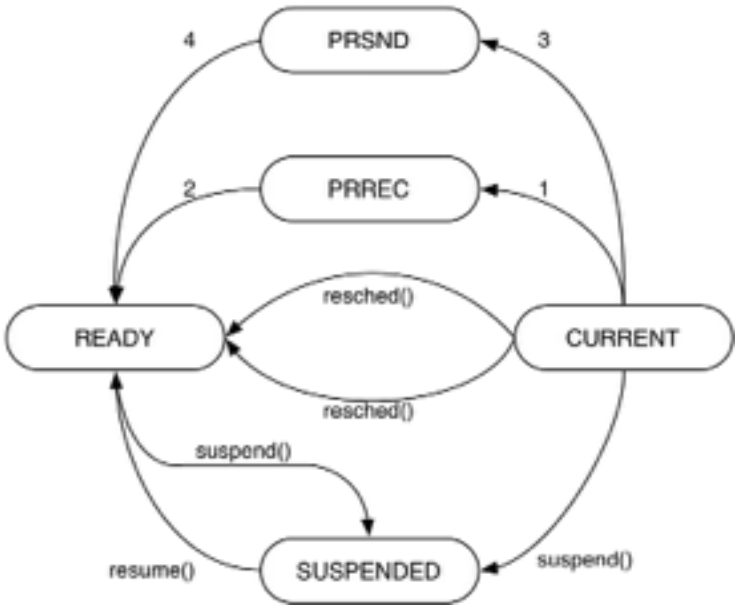
I, Blocking Message Send & Receive

⇛ Data Structure Modified:

- New process states:
  - PRSND // blocked sender, recorded in receivers "blocked sender queue"
  - PRREC // blocked receiver
- New properties in PCB:
  - umsg32 prmsgb[3]; // 3 1-word message
  - uint16 prhasmsgb; // number of valid message in prmsgb
  - qid16 tobe_send; // blocked sender queue, if any
- Additional queue entries:
  - #define NQENT (NPROC + 4 + NSEM + NSEM + NPROC + NPROC)
  - // Assume every process wants to receive messages.
  - // Thus at most 2*NPROC more queuetab entries are needed, to record blocked senders.

⇛ State Transform Diagram:



⇛ Function Added/Modified to Support above Features:

initialize.c	// initialize prhasmsgb
	// create blocked sender queue for each process, record in tobe_send
create.c	// clear prhasmsgb and tobe_send
sendb.c	// block current process, if process with pid has 3 in prmsgb
	// send message to process with pid
	// wake up process with pid, if it is blocked
receiveb.c	// block current process, if it has 0 in prmsgb
	// receive message
	// wake up queueing sender if < 3 prmsgb
kill.c	// if any blocked sender, put blocked processes back to readylist

II, Asynchronous Message Passing

⇛ Data Structure Modified:

- New properties in PCB:

```
bool8   prasync;        // indicate asynchronous message valid or not
```

- New struct:

```
typedef struct {
        umsg32  async_msg;          // 1 word message
        int     (*callback_func)(void);  // fucntion pointer
} cbent;                            // struct for callback entry
```

- New global variable:

```
extern char *callback_ent[];        //pointer to callback entry, 1 for each process
```

⇛ Function Added/Modified:

```
initialize.c      // allocate memory for each callback entry
                  // initialize async_msg and callback_func for each entry
registerrecv.c    // update properties for callback entry of current process
                  // update prasync of current process
send.c            // check prasync first
                  // if TRUE, callback immediately
```

⇛ Design:

Rather than put additional properties to each PCB entry, a global array is used to record pointers to callback entries. Though both solution yield the same result, the first one involves visiting memory space belongs to another process. This is not desired for memory protection in some sense. Moreover, the latter solution is compatible with Xinu's design strategy, which is using relative pointer to indicate callback entry for each process.

⇛ Discussion on Bonus Problem:

Above implementation executes user registered function within kernel. An alternative solution to avoid such situation may be as follow:

> Create another process with user registered function.
> Pass in message as an argument.
> Set its priority to the highest among all the processes.
> Put the process in readylist, with RESCHED_NO flag.

In this way, once a context switch occurs, user registered function gets executed. Even there may be bugs in user supplied code, the process will not affect other processes or corrupt system.

III, Garbage Collection Support

⇒ Data structure modified:

- New struct:

```
typedef struct mb_ent {
        uint32  mb_size;            // current heap variable size
        struct  mb_ent  *mb_next;   // point to next heap variable
} mb_ent;
```

- New properties in PCB:

```
bool8   prhasmem;           // indicate whether additional heap variable is allocated
mb_ent  *prmemlist;         // allocated heap variable list
```

⇒ Design:

The key idea for garbage collection is remembering allocated variables per process, and deallocate them on killing of the process. As in prmemlist, heap variables are tracked using link list within the process, who has allocated them.

In Xinu, stack operation is mostly used for process creation and killing, which are handled pretty well in the original design. Thus, this implementation only includes getmem and freemem, which operates on heap variables, and needs manual deallocation in the original design.

⇒ Function Added/Modified:

```
initialize.c        // initialize prhasmem and prmemlist for each entry
create.c            // clear prhasmem and prmemlist
kill.c              // check prhasmem to see whether heap variable is allocated
                    // if Yes, free all in prmemlist
getmemgb.c          // almost the same as getmem, except:
                    // 1, request additional space to record allocated heap variables
                    // 2, link new "tracker" in prmemlist to keep tracking
freememgb.c         // almost the same as freemem, except:
                    // 1, check if the heap variable is allocated in this process
                    // 2, if Yes, remove from prmemlist. Else, return error
```