Operating Systems

Inter-process Communication (IPC)

Xinu Message Passing

1

Xinu Messages

- Xinu supports completely synchronous and partially asynchronous message functionality
  - Also illustrates direct vs indirect operation with point to point exchange or rendezvous
  - Asynchronous, indirect case discussed later
- For the synchronous case, the system is designed to ensure that processes do not block and that waiting messages don't grow to consume undue memory
  - Primitive operation for resource-constrained embedded systems

2

Message Passing Design

- Limited message size
  - The system limits each message to a small, fixed size
  - In the basic implementation, it is one word (int)
- No message queues
  - The system is permitted to store only one unreceived message – per process – at any time
- First message semantics
  - If several messages are sent to a process, only the first is stored and the subsequent senders do not block
  - This is useful for determining which of a set of events completes first

3

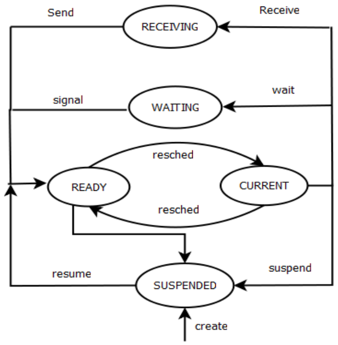
Xinu MP Functions

- Three system calls: *send*, *receive* and *recvclr*
- *send* takes a message and a PID
- *receive* takes no arguments, waits until a message arrives, and returns it
  - Or returns with a message immediately
- *recvclr* is a non-blocking version of receive
  - If a message has arrived, return the message immediately
  - If no message has arrived, return value *OK* immediately
  - Also used to clear an old message if one exists

4

Xinu MP Functions

- Process state for messaging: PR_RECV



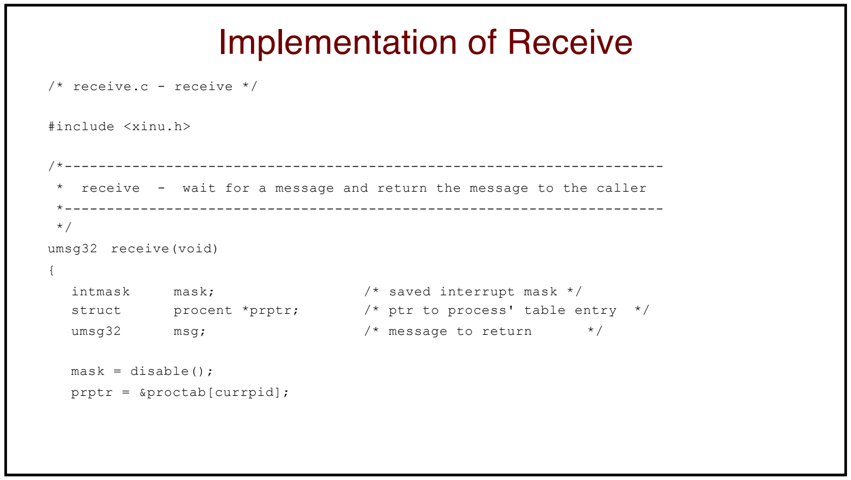
Implementation of Send

- Requires agreement / coordination between senders and receivers
- Sender must store the message somewhere
  - Can't be in the sender's memory, since it might exit
  - Can't be in the receiver's memory since writing into another process's memory is problematic for security, coordination
- Message size and pending limitation addresses this for Xinu
- Space is allocated in the process table entry for a message destined to that process

Implementation of Send

```
/* send.c - send */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  send  -  pass a message to a process and start recipient if waiting
 *------------------------------------------------------------------------
 */
syscall send(
     pid32             pid,          /* ID of recipient process    */
     umsg32        msg               /* contents of message        */
   )
{
   intmask     mask;                 /* saved interrupt mask       */
   struct      procent *prptr;       /* ptr to process' table entry*/
```

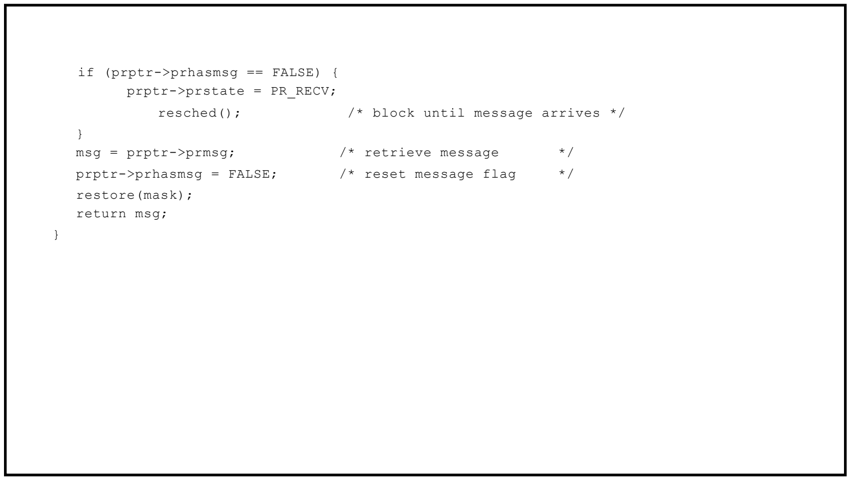
```
   mask = disable();
   if (isbadpid(pid)) {
        restore(mask);
        return SYSERR;
   }
   prptr = &proctab[pid];
   if ((prptr->prstate == PR_FREE) || prptr->prhasmsg) {
        restore(mask);
        return SYSERR;
   }
   prptr->prmsg = msg;             /* deliver message        */
   prptr->prhasmsg = TRUE;    /* indicate message is waiting  */

   /* If recipient waiting or in timed-wait make it ready */
   if (prptr->prstate == PR_RECV) {
        ready(pid, RESCHED_YES);
   } else if (prptr->prstate == PR_RECTIM) {
        unsleep(pid);
        ready(pid, RESCHED_YES);
   }
   restore(mask);             /* restore interrupts */
   return OK;
}
```



Implementation of Receive

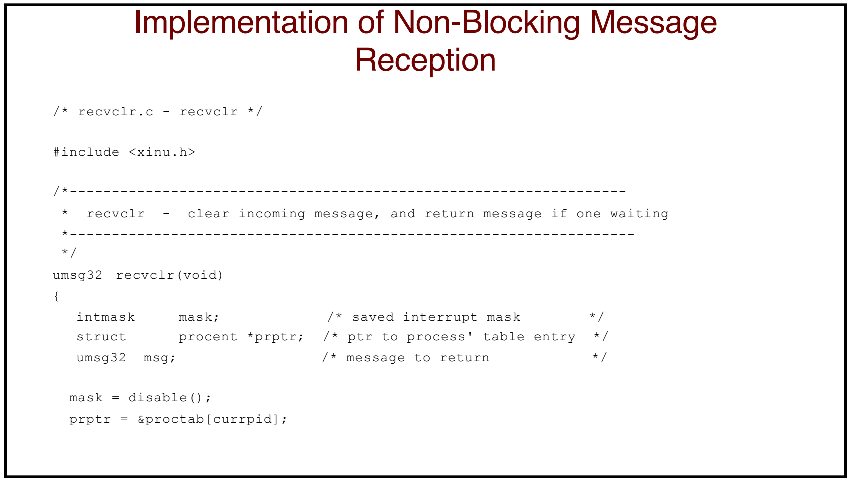
```
/* receive.c - receive */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  receive  -  wait for a message and return the message to the caller
 *------------------------------------------------------------------------
 */
umsg32  receive(void)
{
   intmask     mask;                   /* saved interrupt mask */
   struct      procent *prptr;        /* ptr to process' table entry  */
   umsg32      msg;                   /* message to return      */

   mask = disable();
   prptr = &proctab[currpid];
```

9



```
    if (prptr->prhasmsg == FALSE) {
          prptr->prstate = PR_RECV;
              resched();             /* block until message arrives */
    }
    msg = prptr->prmsg;             /* retrieve message       */
    prptr->prhasmsg = FALSE;        /* reset message flag     */
    restore(mask);
    return msg;
}
```

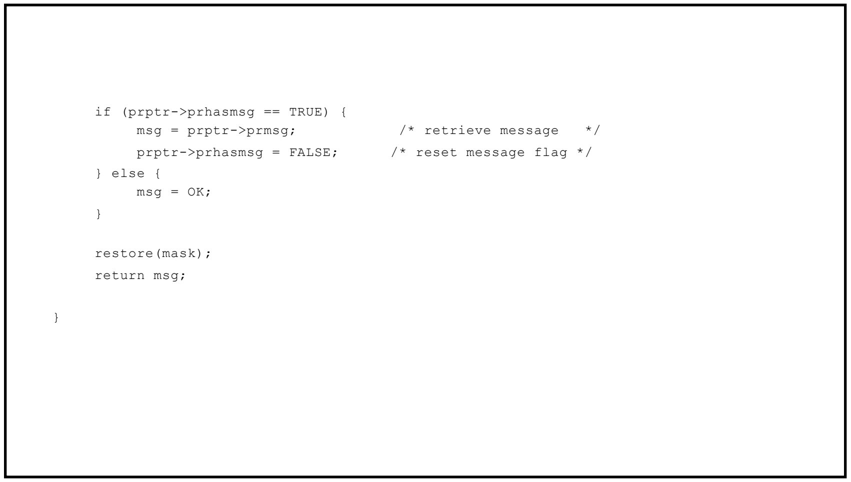
10



Implementation of Non-Blocking Message Reception

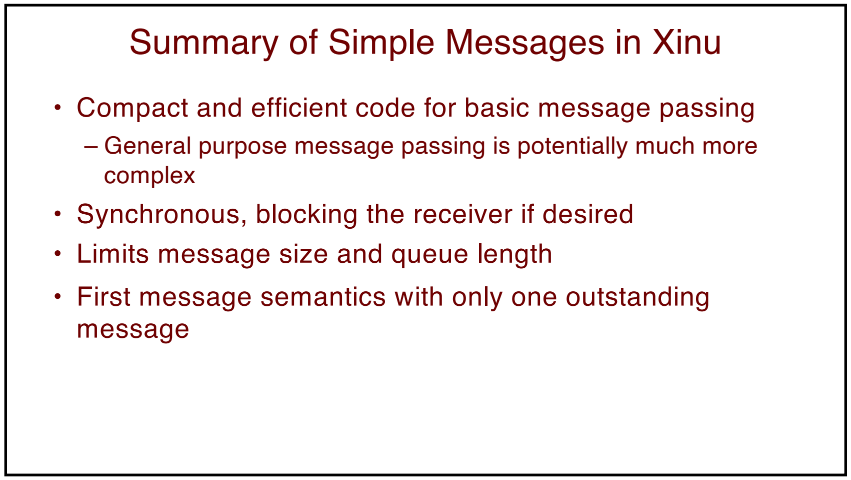
```
/* recvclr.c - recvclr */

#include <xinu.h>

/*------------------------------------------------------------------
 *  recvclr  -  clear incoming message, and return message if one waiting
 *------------------------------------------------------------------
 */
umsg32  recvclr(void)
{
   intmask     mask;             /* saved interrupt mask        */
   struct      procent *prptr;  /* ptr to process' table entry  */
   umsg32   msg;                /* message to return            */

  mask = disable();
  prptr = &proctab[currpid];
```

11



```
    if (prptr->prhasmsg == TRUE) {
         msg = prptr->prmsg;            /* retrieve message   */
         prptr->prhasmsg = FALSE;      /* reset message flag */
    } else {
         msg = OK;
    }

    restore(mask);
    return msg;

}
```

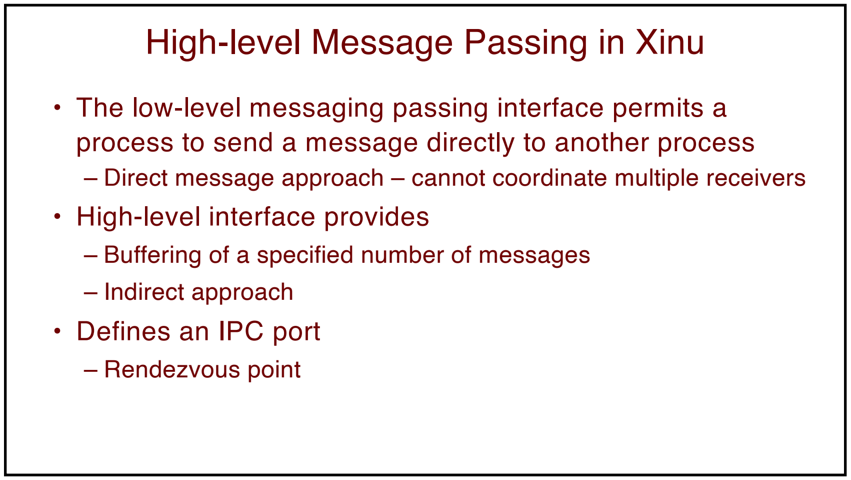
12



Summary of Simple Messages in Xinu

- Compact and efficient code for basic message passing
  - General purpose message passing is potentially much more complex
- Synchronous, blocking the receiver if desired
- Limits message size and queue length
- First message semantics with only one outstanding message

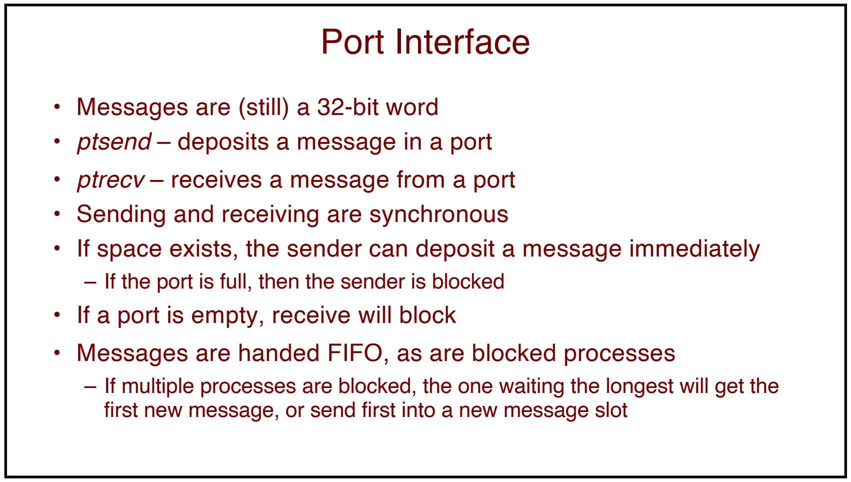
13



High-level Message Passing in Xinu

- The low-level messaging passing interface permits a process to send a message directly to another process
  - Direct message approach – cannot coordinate multiple receivers
- High-level interface provides
  - Buffering of a specified number of messages
  - Indirect approach
- Defines an IPC port
  - Rendezvous point

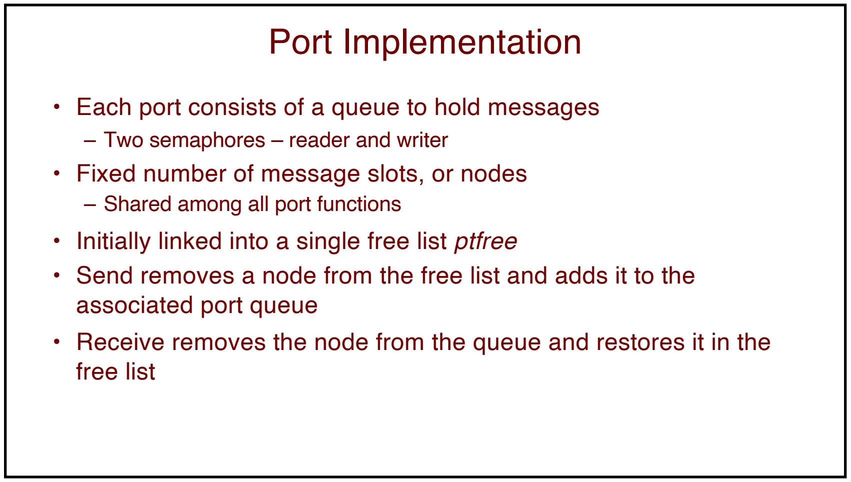
14



Port Interface

- Messages are (still) a 32-bit word
- *ptsend* – deposits a message in a port
- *ptrecv* – receives a message from a port
- Sending and receiving are synchronous
- If space exists, the sender can deposit a message immediately
  - If the port is full, then the sender is blocked
- If a port is empty, receive will block
- Messages are handed FIFO, as are blocked processes
  - If multiple processes are blocked, the one waiting the longest will get the first new message, or send first into a new message slot

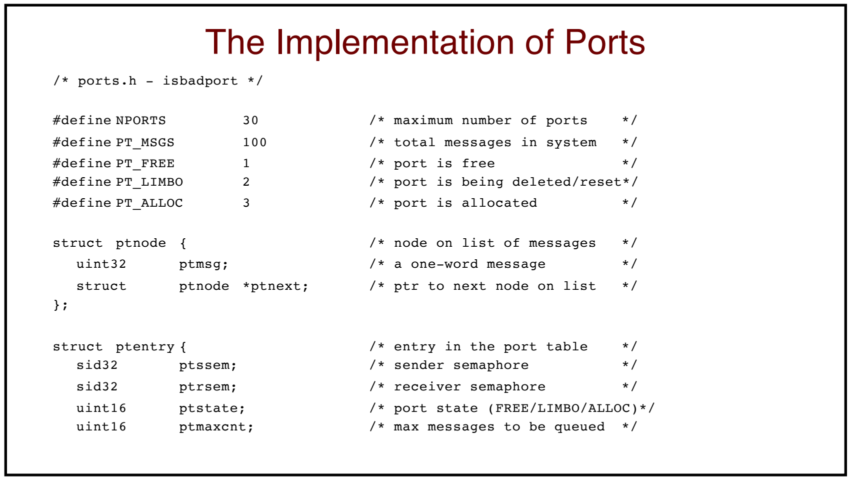
15



Port Implementation

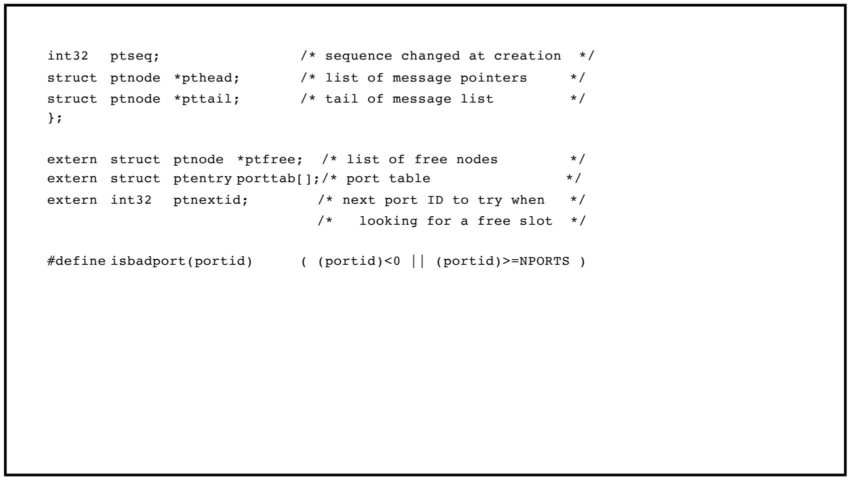
- Each port consists of a queue to hold messages
  - Two semaphores – reader and writer
- Fixed number of message slots, or nodes
  - Shared among all port functions
- Initially linked into a single free list *ptfree*
- Send removes a node from the free list and adds it to the associated port queue
- Receive removes the node from the queue and restores it in the free list

16

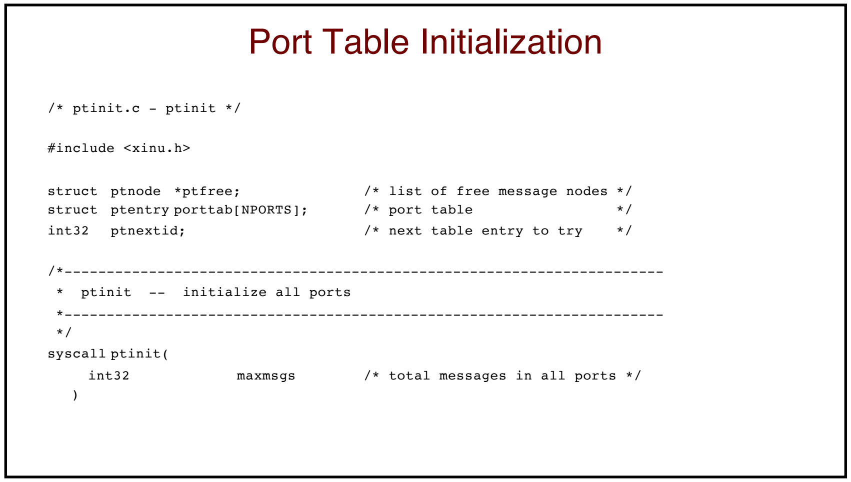


The Implementation of Ports

```
/* ports.h - isbadport */

#define NPORTS          30              /* maximum number of ports    */
#define PT_MSGS         100             /* total messages in system   */
#define PT_FREE         1               /* port is free               */
#define PT_LIMBO        2               /* port is being deleted/reset*/
#define PT_ALLOC        3               /* port is allocated          */

struct  ptnode  {                       /* node on list of messages   */
   uint32      ptmsg;                   /* a one-word message         */
   struct      ptnode  *ptnext;         /* ptr to next node on list   */
};

struct  ptentry {                       /* entry in the port table    */
   sid32       ptssem;                  /* sender semaphore           */
   sid32       ptrsem;                  /* receiver semaphore         */
   uint16      ptstate;                 /* port state (FREE/LIMBO/ALLOC)*/
   uint16      ptmaxcnt;                /* max messages to be queued  */
```

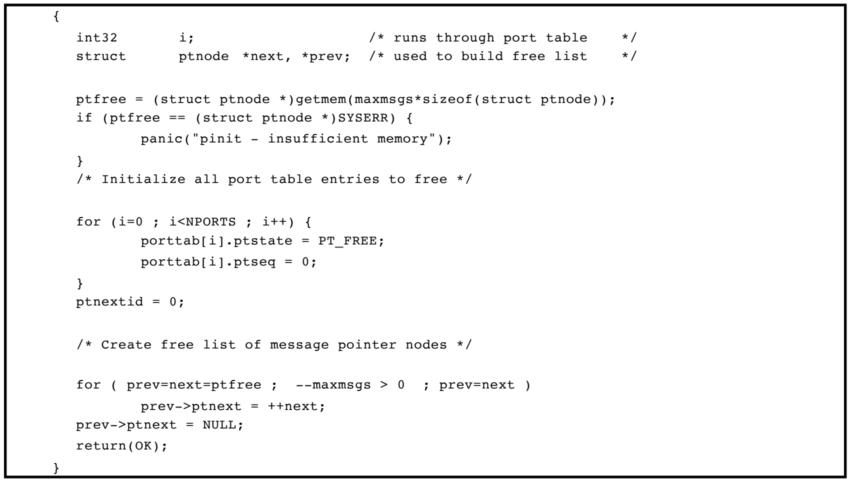


```
int32   ptseq;                  /* sequence changed at creation  */
struct  ptnode  *pthead;        /* list of message pointers     */
struct  ptnode  *pttail;        /* tail of message list         */
};

extern  struct  ptnode  *ptfree;  /* list of free nodes         */
extern  struct  ptentry porttab[];/* port table                */
extern  int32   ptnextid;         /* next port ID to try when   */
                                  /*   looking for a free slot  */

#define isbadport(portid)      ( (portid)<0 || (portid)>=NPORTS )
```

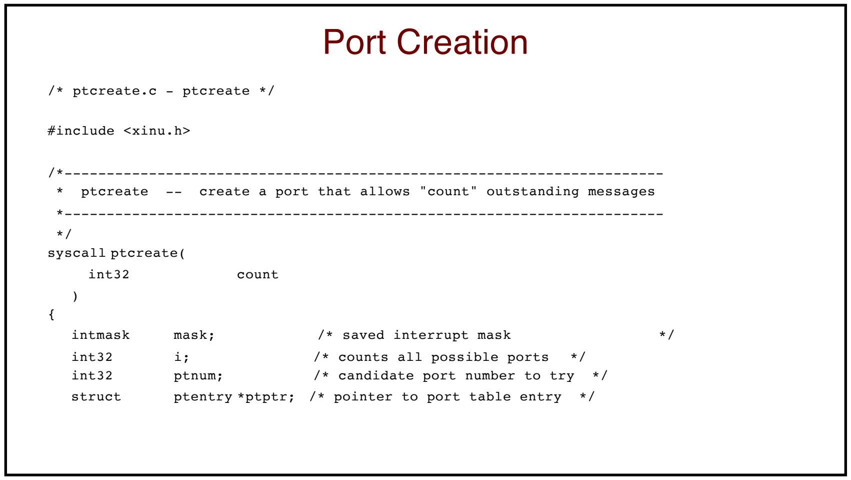


Port Table Initialization

```
/* ptinit.c - ptinit */

#include <xinu.h>

struct  ptnode  *ptfree;              /* list of free message nodes */
struct  ptentry porttab[NPORTS];      /* port table                 */
int32   ptnextid;                     /* next table entry to try    */

/*------------------------------------------------------------------------
 *  ptinit  --  initialize all ports
 *------------------------------------------------------------------------
 */
syscall ptinit(
     int32             maxmsgs        /* total messages in all ports */
   )
```



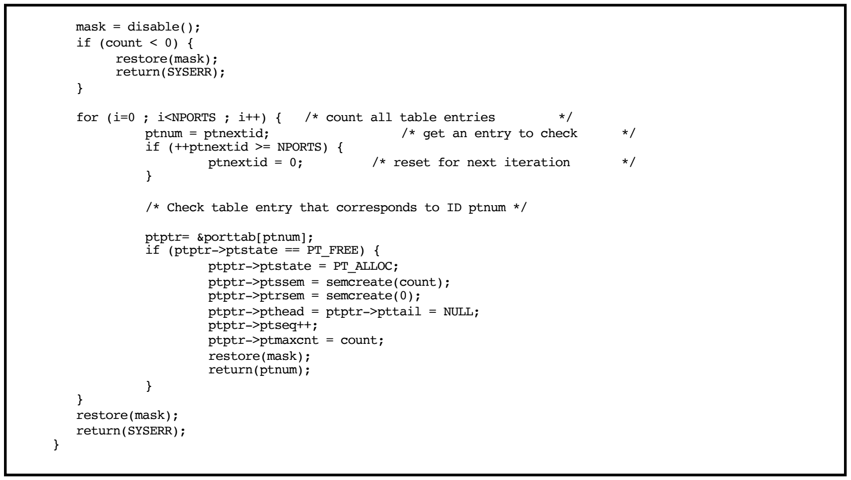
```
{
   int32       i;                     /* runs through port table    */
   struct      ptnode  *next, *prev;  /* used to build free list    */

   ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
   if (ptfree == (struct ptnode *)SYSERR) {
           panic("pinit - insufficient memory");
   }
   /* Initialize all port table entries to free */

   for (i=0 ; i<NPORTS ; i++) {
           porttab[i].ptstate = PT_FREE;
           porttab[i].ptseq = 0;
   }
   ptnextid = 0;

   /* Create free list of message pointer nodes */

   for ( prev=next=ptfree ;  --maxmsgs > 0  ; prev=next )
           prev->ptnext = ++next;
   prev->ptnext = NULL;
   return(OK);
}
```



Port Creation

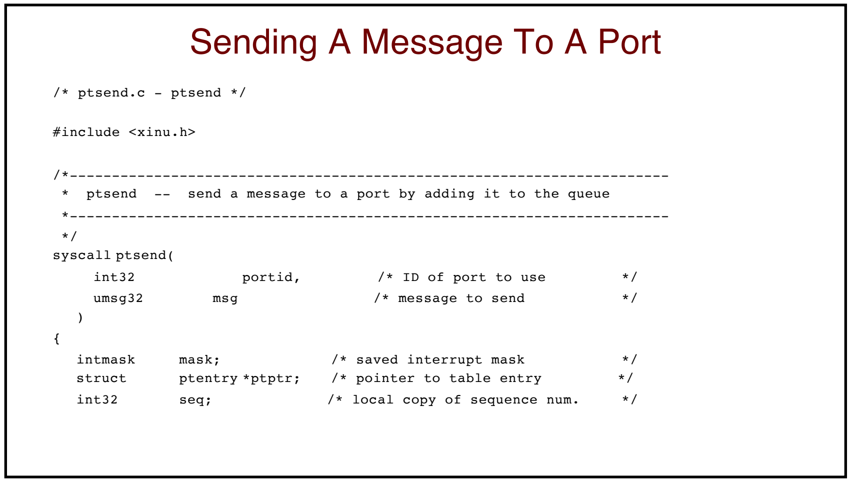
```
/* ptcreate.c - ptcreate */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptcreate  --  create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
     int32            count
   )
{
   intmask     mask;             /* saved interrupt mask                 */
   int32       i;               /* counts all possible ports   */
   int32       ptnum;           /* candidate port number to try  */
   struct      ptentry *ptptr;  /* pointer to port table entry  */
```

21



```
   mask = disable();
   if (count < 0) {
        restore(mask);
        return(SYSERR);
   }

   for (i=0 ; i<NPORTS ; i++) {   /* count all table entries        */
           ptnum = ptnextid;                 /* get an entry to check      */
           if (++ptnextid >= NPORTS) {
                   ptnextid = 0;         /* reset for next iteration       */
           }

           /* Check table entry that corresponds to ID ptnum */

           ptptr= &porttab[ptnum];
           if (ptptr->ptstate == PT_FREE) {
                   ptptr->ptstate = PT_ALLOC;
                   ptptr->ptssem = semcreate(count);
                   ptptr->ptrsem = semcreate(0);
                   ptptr->pthead = ptptr->pttail = NULL;
                   ptptr->ptseq++;
                   ptptr->ptmaxcnt = count;
                   restore(mask);
                   return(ptnum);
           }
   }
   restore(mask);
   return(SYSERR);
}
```

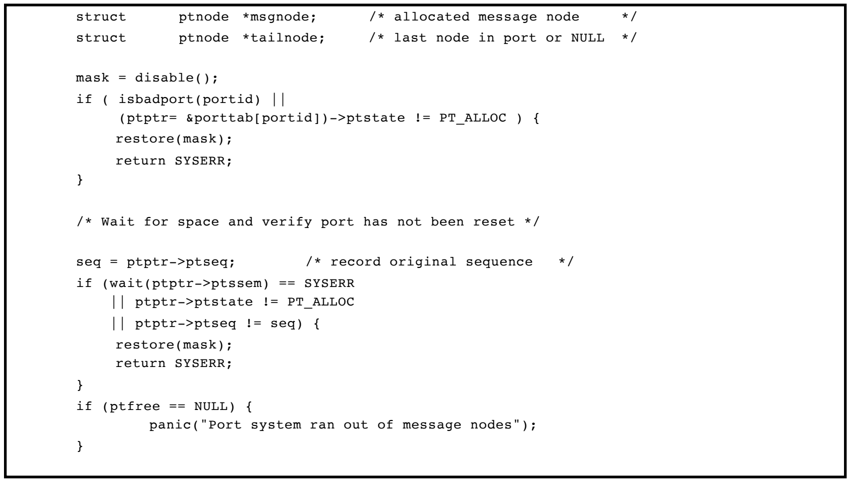
22



Sending A Message To A Port

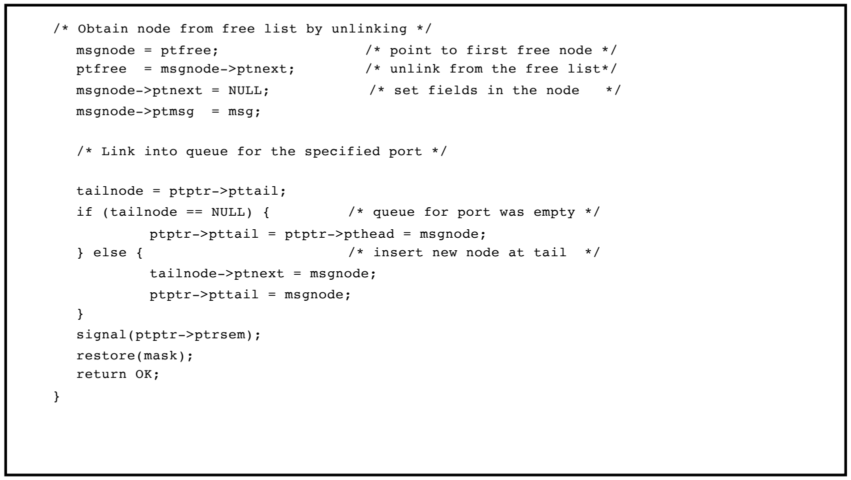
```
/* ptsend.c - ptsend */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptsend  --  send a message to a port by adding it to the queue
 *------------------------------------------------------------------------
 */
syscall ptsend(
     int32            portid,         /* ID of port to use          */
     umsg32        msg                /* message to send            */
   )
{
   intmask     mask;              /* saved interrupt mask            */
   struct      ptentry *ptptr;    /* pointer to table entry          */
   int32       seq;              /* local copy of sequence num.     */
```

23



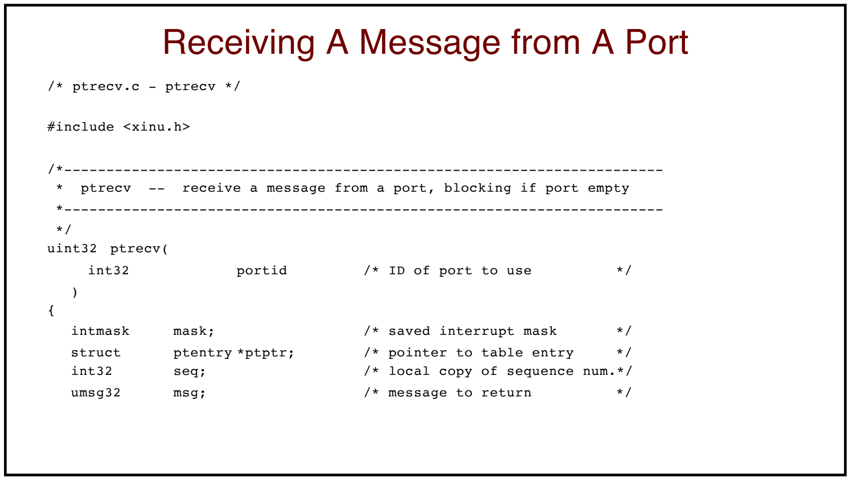
```
   struct      ptnode  *msgnode;      /* allocated message node      */
   struct      ptnode  *tailnode;     /* last node in port or NULL  */

   mask = disable();
   if ( isbadport(portid) ||
        (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return SYSERR;
   }

   /* Wait for space and verify port has not been reset */

   seq = ptptr->ptseq;          /* record original sequence   */
   if (wait(ptptr->ptssem) == SYSERR
       || ptptr->ptstate != PT_ALLOC
       || ptptr->ptseq != seq) {
        restore(mask);
        return SYSERR;
   }
   if (ptfree == NULL) {
           panic("Port system ran out of message nodes");
   }
```

24



```
/* Obtain node from free list by unlinking */
   msgnode = ptfree;                  /* point to first free node */
   ptfree  = msgnode->ptnext;         /* unlink from the free list*/
   msgnode->ptnext = NULL;             /* set fields in the node   */
   msgnode->ptmsg  = msg;

   /* Link into queue for the specified port */

   tailnode = ptptr->pttail;
   if (tailnode == NULL) {          /* queue for port was empty */
            ptptr->pttail = ptptr->pthead = msgnode;
   } else {                         /* insert new node at tail  */
            tailnode->ptnext = msgnode;
            ptptr->pttail = msgnode;
   }
   signal(ptptr->ptrsem);
   restore(mask);
   return OK;
}
```

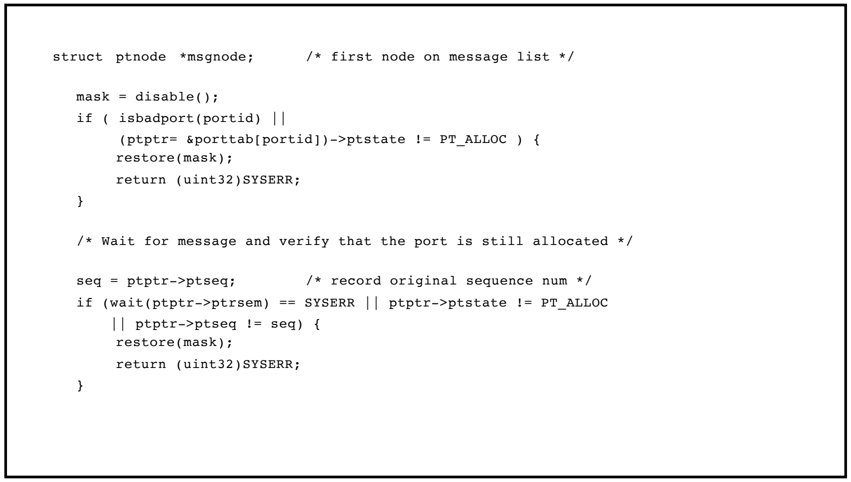
25



Receiving A Message from A Port

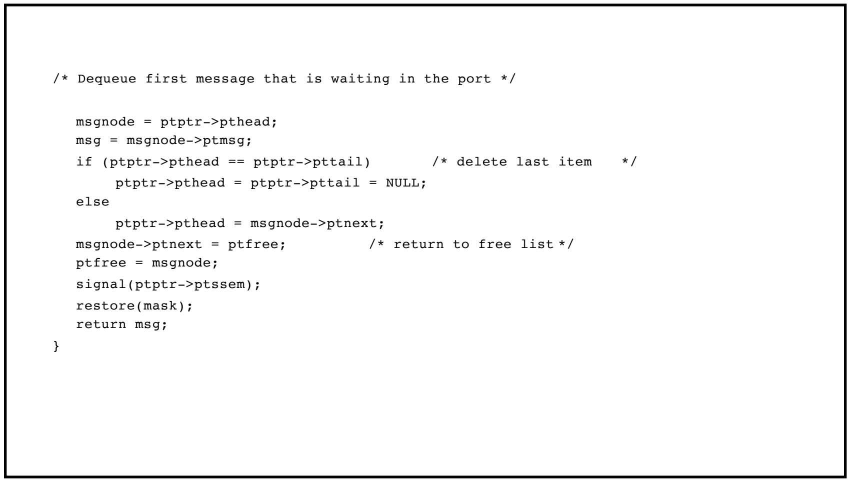
```
/* ptrecv.c - ptrecv */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptrecv  --  receive a message from a port, blocking if port empty
 *------------------------------------------------------------------------
 */
uint32  ptrecv(
     int32            portid         /* ID of port to use          */
   )
{
   intmask     mask;                 /* saved interrupt mask       */
   struct      ptentry *ptptr;       /* pointer to table entry     */
   int32       seq;                  /* local copy of sequence num.*/
   umsg32      msg;                  /* message to return          */
```

26



```
struct  ptnode  *msgnode;      /* first node on message list */

   mask = disable();
   if ( isbadport(portid) ||
        (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return (uint32)SYSERR;
   }

   /* Wait for message and verify that the port is still allocated */

   seq = ptptr->ptseq;        /* record original sequence num */
   if (wait(ptptr->ptrsem) == SYSERR || ptptr->ptstate != PT_ALLOC
       || ptptr->ptseq != seq) {
        restore(mask);
        return (uint32)SYSERR;
   }
```

27



```
/* Dequeue first message that is waiting in the port */

   msgnode = ptptr->pthead;
   msg = msgnode->ptmsg;
   if (ptptr->pthead == ptptr->pttail)       /* delete last item    */
        ptptr->pthead = ptptr->pttail = NULL;
   else
        ptptr->pthead = msgnode->ptnext;
   msgnode->ptnext = ptfree;          /* return to free list*/
   ptfree = msgnode;
   signal(ptptr->ptssem);
   restore(mask);
   return msg;
}
```

28

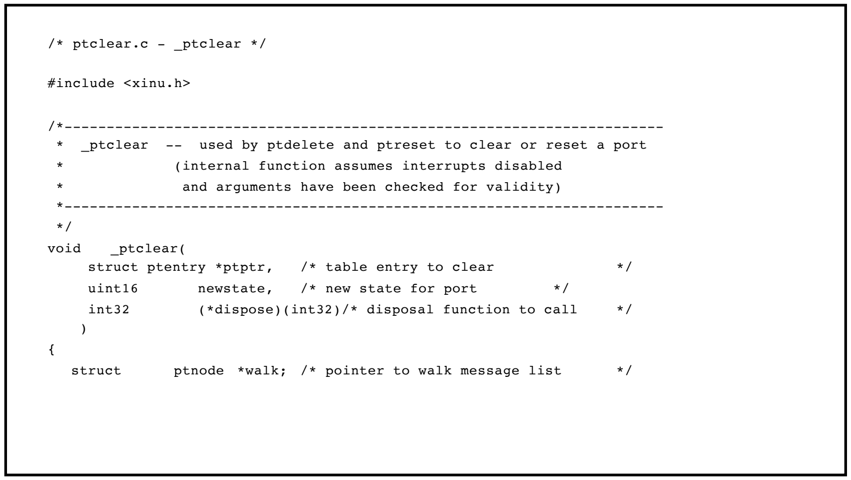
Port Deletion and Reset

```
/* ptdelete.c - ptdelete */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptdelete  --  delete a port, freeing waiting processes and messages
 *------------------------------------------------------------------------
 */
syscall ptdelete(
     int32      portid,             /* ID of port to delete */
     int32      (*dispose)(int32)   /* function to call to dispose */
   )                                /*   of waiting messages       */
{
   intmask     mask;                 /* saved interrupt mask       */
   struct      ptentry *ptptr;       /* pointer to port table entry*/
```

```
    mask = disable();
    if ( isbadport(portid) ||
         (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
         restore(mask);
         return(SYSERR);
    }
    _ptclear(ptptr, PT_FREE, dispose);
    ptnextid = portid;
    restore(mask);
    return(OK);
}
```

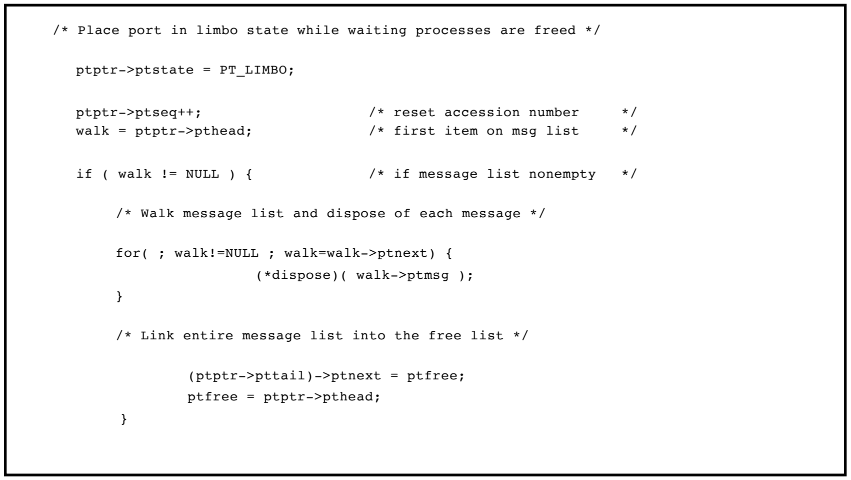
```
/* ptreset.c - ptreset */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptreset  --  reset a port, freeing waiting processes and messages and
               leaving the port ready for further use
 *------------------------------------------------------------------------
 */
syscall ptreset(
     int32      portid,             /* ID of port to reset         */
     int32      (*dispose)(int32)   /* function to call to dispose */
   )                                /*   of waiting messages       */
```

```
{
   intmask     mask;             /* saved interrupt mask    */
   struct      ptentry *ptptr;   /* pointer to port table entry     */

   mask = disable();
   if ( isbadport(portid) ||
        (ptptr= &porttab[portid])->ptstate != PT_ALLOC ) {
        restore(mask);
        return SYSERR;
   }
   _ptclear(ptptr, PT_ALLOC, dispose);
   restore(mask);
   return OK;
}
```



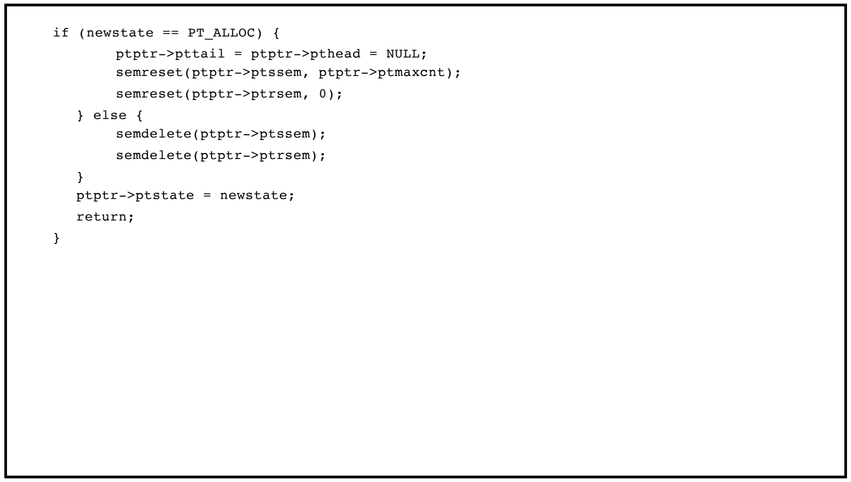
```
/* ptclear.c - _ptclear */

#include <xinu.h>

/*------------------------------------------------------------------------
 *  _ptclear  --  used by ptdelete and ptreset to clear or reset a port
 *             (internal function assumes interrupts disabled
 *              and arguments have been checked for validity)
 *------------------------------------------------------------------------
 */
void    _ptclear(
     struct ptentry *ptptr,   /* table entry to clear               */
     uint16       newstate,   /* new state for port         */
     int32        (*dispose)(int32)/* disposal function to call     */
    )
{
   struct      ptnode  *walk;  /* pointer to walk message list       */
```

33



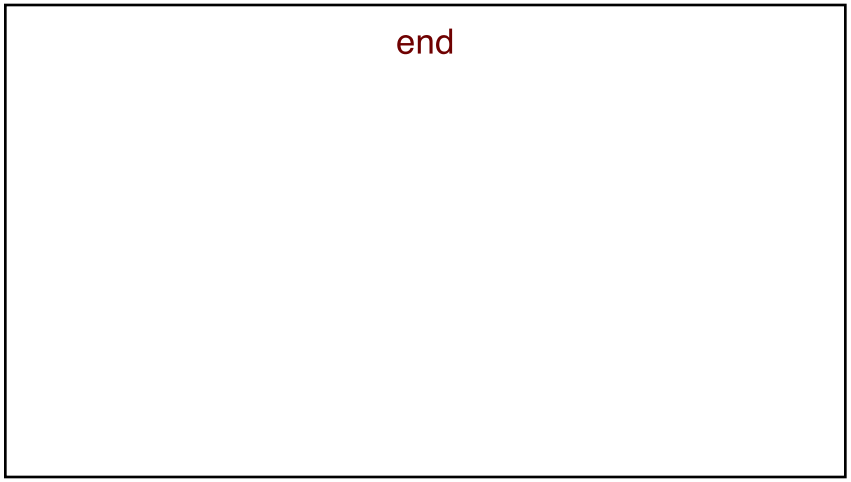
```
/* Place port in limbo state while waiting processes are freed */

   ptptr->ptstate = PT_LIMBO;

   ptptr->ptseq++;                    /* reset accession number     */
   walk = ptptr->pthead;              /* first item on msg list     */

   if ( walk != NULL ) {              /* if message list nonempty   */

        /* Walk message list and dispose of each message */

        for( ; walk!=NULL ; walk=walk->ptnext) {
                        (*dispose)( walk->ptmsg );
        }

        /* Link entire message list into the free list */

                (ptptr->pttail)->ptnext = ptfree;
                ptfree = ptptr->pthead;
        }
```

34



```
if (newstate == PT_ALLOC) {
        ptptr->pttail = ptptr->pthead = NULL;
        semreset(ptptr->ptssem, ptptr->ptmaxcnt);
        semreset(ptptr->ptrsem, 0);
   } else {
        semdelete(ptptr->ptssem);
        semdelete(ptptr->ptrsem);
   }
   ptptr->ptstate = newstate;
   return;
}
```

35



end

36