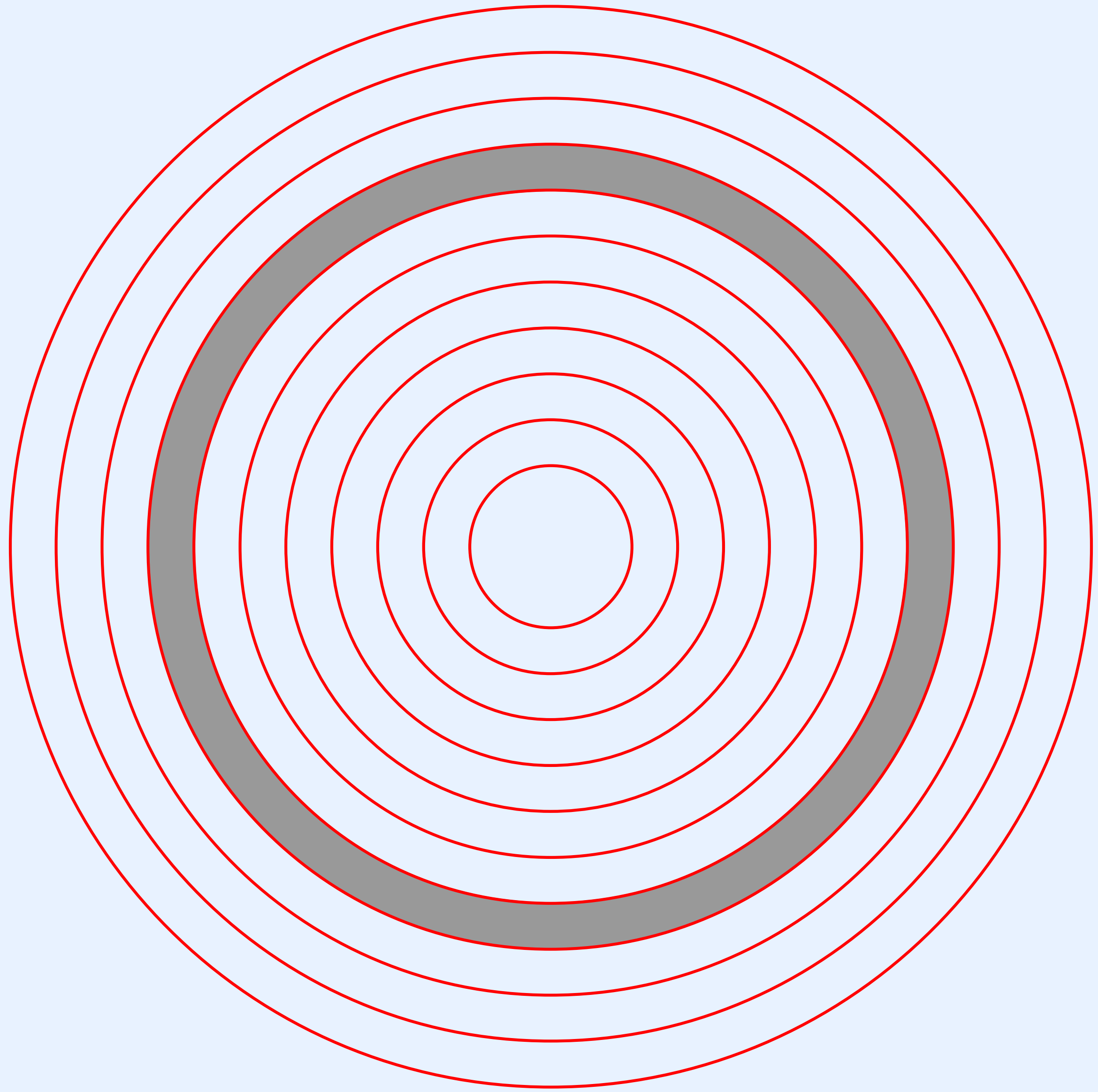
Module XI

High-Level Synchronous Message Passing

Location Of Synchronous Message Passing In The Hierarchy



A Review Of Xinu’s Low-Level Message Passing Facility

- A message is always sent from one process directly to another
- Each process has a one-message message buffer
- Transmission is asynchronous (non-blocking)
- Reception is synchronous (blocking)
- An asynchronous function can be used to clear the message buffer

Features Of The Xinu High-Level Message Passing Mechanism

- Defines a set of message storage facilities called *ports* used for inter-process communication
- When creating a port, an application specifies the number of messages a given port can hold
- The mechanism supports many-to-many communication
  - Allows an arbitrary process to send a message to a port
  - Allows an arbitrary process to receive a message from a port
- Uses a synchronous interface
  - Blocks a sender if a port is full
  - Blocks a receiver until a message arrives at a port
- Handles port deletion and reset

An Example Use Of Ports: A Concurrent Server

- Create a port, *P*
- Think of messages that are sent to the port as requests for some service
- Create a set of server processes that each repeatedly receive a request from *P* and "handle" the request (supply the service)
- An arbitrary process can send a request to *P*; one of the server processes handles the request
- Because server processes run concurrently, a server process can receive a later request and start handling the request while another process continues to handle a previous request
- The advantage: short requests can be serviced quickly

A Few Details

- When the port system is initialized, a global pool of messages is created
  - The maximum number of messages in all ports is specified
  - Memory is allocated for the pool, and messages are linked onto a free list
- An individual port can be created (and later deleted) dynamically
- Semaphores are used to
  - Block a sender if a port is full
  - Block a receiver if a port is empty
- When a port is created
  - An argument specifies the number of messages that can be stored in the port
  - The message count is used to initialize a semaphore

Functions That Operate On Ports

- *Ptinit*
  - Must be called once before ports can be used
  - Initializes the entire port system
- *Ptcreate*
  - Creates a new port
  - An argument specifies maximum number of messages
- *Ptsend*
  - Sends a message to a port
- *Ptrecv*
  - Retrieves a message from a port

Functions That Operate On Ports
(continued)

- *Ptreset*
  - Resets existing port
  - Disposes of existing messages
  - Allows waiting processes to continue
- *Ptdelete*
  - Deletes existing port
  - Disposes of existing messages
  - Allows blocked processes to continue

Programmer's Responsibility

- A programmer must plan ahead
  - Specify the maximum number of messages when calling ptcreate
  - Avoid creating ports that can take more than the total messages available for all ports
- Worst case: ptsend will panic if no message buffers appear on the free list
- Possible improvement: keep a global count of messages, and decrement it each time ptcreate is called and increment it each time ptdelete is called

Port Declarations

```
/* 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;        /* Pointer 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 )
```

An Invariant Used Throughout The Code

Semaphore ptrsem *has a nonnegative count* n *if* n *messages are waiting in the port; it has negative count* –n *if* n *processes are waiting for messages.*

Xinu Ptinit (Part 1)

```
/* 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 the port table  */
        struct  ptnode  *next, *curr;   /* Used to build a free list    */

        /* Allocate memory for all messages on all ports */

        ptfree = (struct ptnode *)getmem(maxmsgs*sizeof(struct ptnode));
        if (ptfree == (struct ptnode *)SYSERR) {
                panic("ptinit - insufficient memory");
        }
```

Xinu Ptinit (Part 2)

```
        /* 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 a free list of message nodes linked together */

        for ( curr=next=ptfree ;  --maxmsgs > 0  ; curr=next ) {
                curr->ptnext = ++next;
        }

        /* Set the pointer in the final node to NULL */

        curr->ptnext = NULL;
        return OK;
}
```

Xinu Ptcreate (Part 1)

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

#include <xinu.h>

/*------------------------------------------------------------------------
 *  ptcreate  -  Create a port that allows "count" outstanding messages
 *------------------------------------------------------------------------
 */
syscall ptcreate(
          int32         count           /* Size of port                 */
        )
{
        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  */

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

Xinu Ptcreate (Part 2)

```
	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;
}
```

Xinu Ptsend (Part 1)

```
/* 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.  */
        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;
        }
```

Xinu Ptsend (Part 2)

```
/* 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");
}

/* 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;
```

Xinu Ptsend (Part 3)

```
        /* 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;
}
```

Xinu Ptrecv (Part 1)

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

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

Xinu Ptrecv (Part 2)

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

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

        /* 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;
}
```

Port Deletion And Reset

- Illustrate how difficult it can be to delete resources in a concurrent system
- Situations that must be handled
  - If the port is full, processes may be blocked waiting to send messages to the port
  - If the port is empty, processes may be blocked waiting to receive messages from the port
  - If the port contains messages, some processing may be needed for each message
- An example of message processing during deletion
  - Suppose an application allocates heap memory and uses a message to send a pointer to the block of memory
  - When deleting such a port, the appropriate action may be to free the block of memory associated with each message

Disposing Of Messages

- Message disposition is needed during both reset and deletion
- What action should the system take to dispose of a message?
- Key idea: only the applications using the port will know how to dispose of messages
- To accommodate disposition
  - Both *ptreset* and *ptdelete* include an extra argument that specifies a disposition function
  - When a message is removed from the port, the disposition function is called with the message as an argument

How Dynamic Deletion Complicates A Design

- If concurrent processes can create/use/delete a resource, they can interfere
- Consider what happens with ports if
  - Process *A* invokes *ptsend* to send a message to a port
  - The port is full, so process *A* is blocked
  - While process *A* is blocked, process *B* starts to delete the port
  - Once the semaphores are deleted, process *A* will become ready
- If process *B* has lower priority than process *A*, process *A* will run
- How will process *A* know that the port is being deleted?
- A similar situation occurs for senders
- Another surprise: suppose multiple processes attempt to delete and/or reset the port concurrently

Concurrency And Message Disposition

- The function used to dispose of messages during deletion or reset
  - Is specified by user
  - May reschedule allowing other processes to execute
- An example
  - Suppose each message contains a pointer to a buffer from a buffer pool
  - The user's disposition function calls *freebuf* to free the buffer
  - *Freebuf* signals a semaphore, which calls *resched*
- Consequence: we need to handle attempts to use the port concurrently during reset or deletion

Three Possible Ways To Handle Reset/Deletion

- Mechanism 1: Accession Numbers
  - A sequence number is associated with each port
  - The sequence number is incremented when the port is created and when the port is deleted or reset
  - Functions *ptsend* and *ptrecv* record the sequence number when an operation begins and check the sequence number after *wait* returns
  - If the sequence number changed, the port was reset, so the operation must abort

Three Possible Ways To Handle Reset/Deletion
(continued)

- Mechanism 2: A New State For The Port
  - Each port has a *state* variable
  - Many OS objects only need a bit to specify whether the object is in use or free
  - Use an additional state to handle deletion/reset
    - *PTFREE* if the entry for the port is not in use
    - *PTALLOC* if the port is in use
    - *PTLIMBO* if the port is being reset/deleted
  - Functions *ptsend* and *ptrecv* examine the state variable
  - If the state is *PTLIMBO*, the port is currently being reset or deleted and cannot be used

Three Possible Ways To Handle Reset/Deletion
(continued)

- Mechanism 3: Deferred Rescheduling
  - Is not included in the current code
  - The idea: temporarily postpone scheduling decisions during reset
  - To apply deferred rescheduling
    - Call *resched_cntl(DEFER_START)* at the start of reset or delete
    - Call *resched_cntl(DEFER_STOP)* after all operations are performed
  - Note that deferred rescheduling means that message disposition will not start other concurrent processes

Common Code For Reset and Deletion

- We will see that port reset and deletion perform many of the same actions
- To eliminate code duplication
  - Place common code in an internal function, *_ptclear*
  - Have both ptreset and ptdelete call *_ptclear*
- Note: the designation “internal” means that *_ptclear* is *not* a system call — it *must* be called with interrupts disabled

Xinu Ptdelete

```
/* 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         (*disp)(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, disp);
        ptnextid = portid;
        restore(mask);
        return OK;
}
```

Xinu Ptreset

```
/* 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         (*disp)(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, disp);
        restore(mask);
        return OK;
}
```

Xinu _ptclear (Part 1)

```
/* 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 */

        /* 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       */
```

Xinu _ptclear (Part 2)

```
        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;
        }

        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;
}
```

Summary

- Xinu offers a high-level message passing mechanism
- The system uses ports for message storage
- A port can be created dynamically, can have arbitrary senders, and arbitrary receivers
- The interface is completely synchronous — a sender blocks if a port is full, and a receiver blocks if a port is empty
- Port reset / deletion is tricky because
  - Concurrent processes may attempt to use the port while reset or deletion is occurring
  - Senders and receivers must be able to tell that the port changed while they were blocked

Summary
(continued)

- Three techniques can handle transition
  - A sequence number informs waiting processes whether the port was reset or deleted while they were blocked
  - A limbo state prevents new processes from using the port while it is being reset or deleted
  - Processes using ports can defer rescheduling during reset and deletion to guarantee that no other processes execute

Questions?