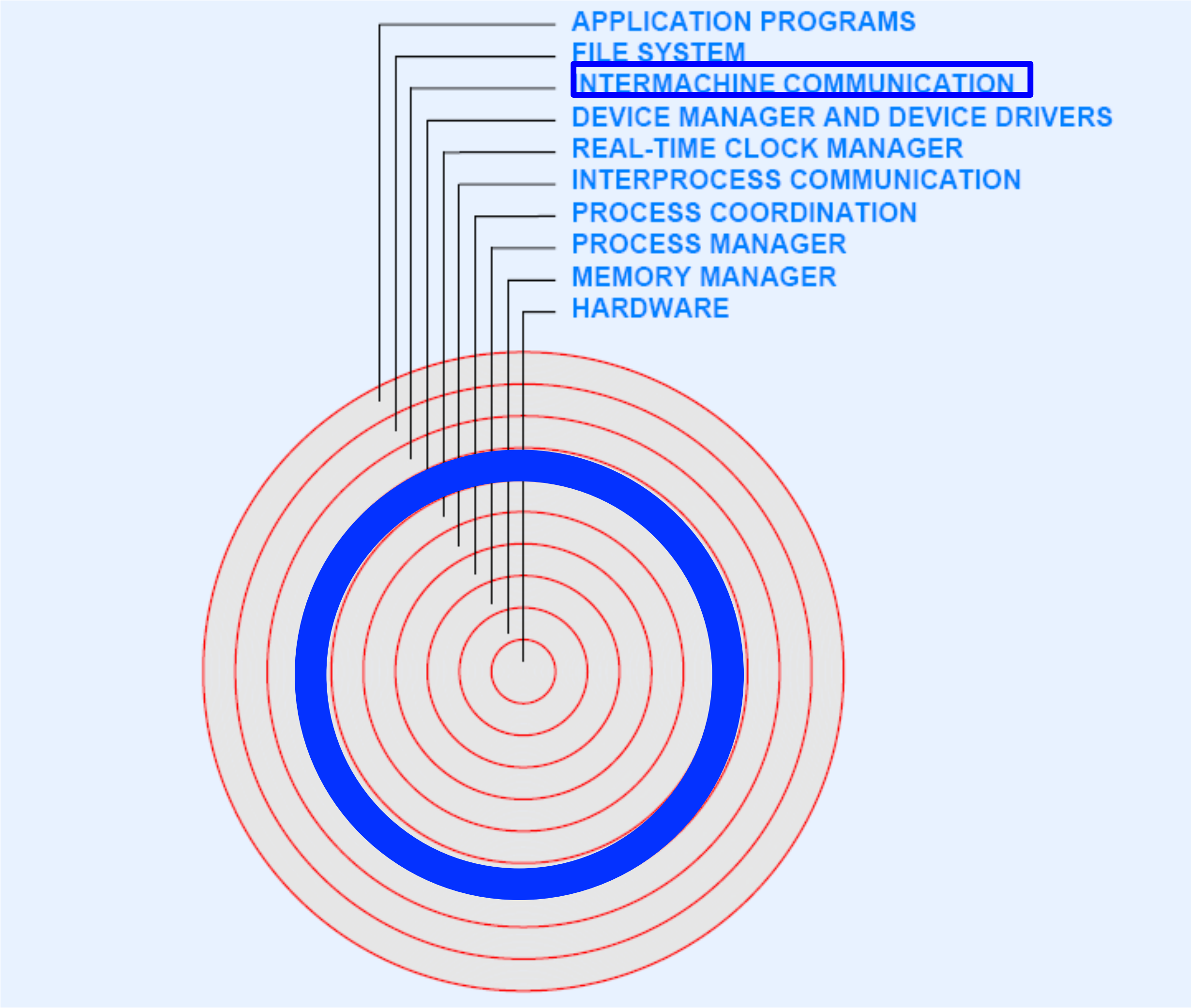
CSCI 8530
Advanced Operating Systems

Part 11

High-level Synchronous Message Passing

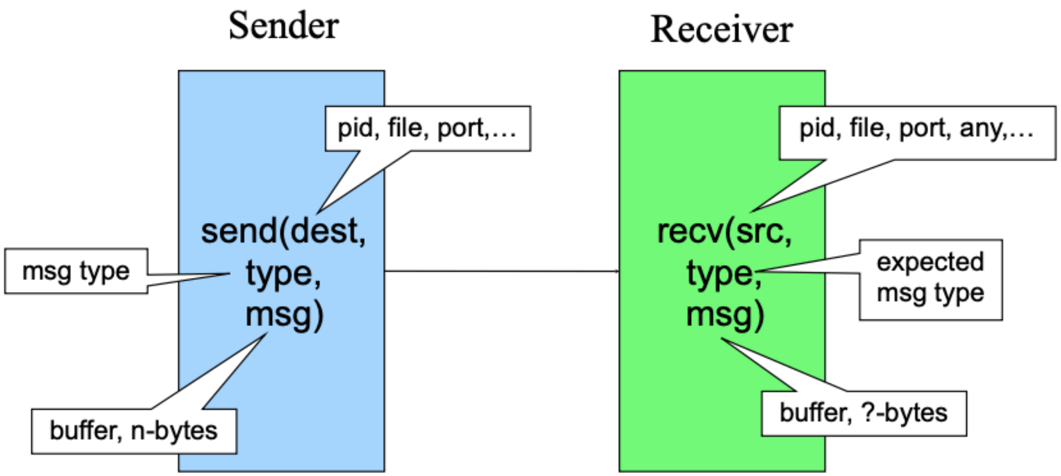
Location of Synchronous Message Passing in the Hierarchy



Review of Message Passing Choices

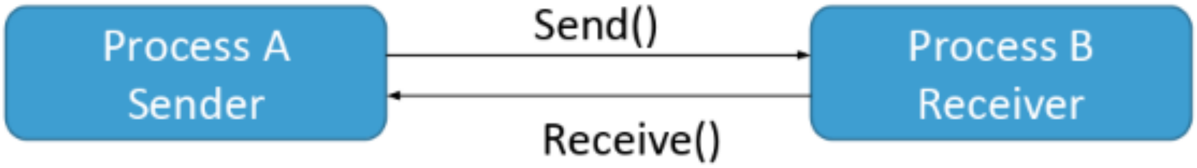
- Potential synchronization
  - Sender blocks
  - Receiver blocks
  - Neither blocks
  - Both block

- Messages outstanding at a given time
  - Arbitrary number
  - Small, fixed number



Review of Message Passing Choices (continued)

- Use two system call
  - send (destination, &message)
  - receive (source, &message)
- Message storage
  - Associated with sender
  - Associated with receiver
  - Independent of sender and receiver
- Destination
  - A specific process
  - Intermediate pickup point accessible to multiple processes



More Review: Xinu Low-level Message Passing

- Asynchronous (non-blocking) transmission
- Synchronous (blocking) reception
- Asynchronous message clear
- Message buffer holds one message
- Destination is a specific process

Motivations for High-level Message Passing

- Permit synchronous message transfer
  - Block sender until receiver ready
  - Block receiver until sender ready
  - Example: data pipeline
- Make a message available to any process in a set
- Example
  - Concurrent server
  - Set of processes that can handle requests
  - Next process in set handles each incoming request
  - Allows short requests to be serviced quickly

Xinu High-Level Message Passing Mechanism

- Use inter-process communication port (a *port* mechanism) to refer to a rendezvous point
- Separate abstraction, unrelated to low-level message passing
- Part of kernel
- Independent from processes
- Allows arbitrary process to
  - Send messages
  - Receive messages

Port Details

- Port
  - Created dynamically
  - Provides a synchronous interface using a producer and consumer semaphore per port
    - Receiver blocks when port empty
    - Sender blocks when port full
    - Requests are also handled in FIFO order
- At port creation
  - Maximum number of messages is fixed and storage is allocated
  - Semaphores are created
    - Producers and Consumers

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 )
```

Xinu Port Functions

- *Ptinit*
  - Called once at startup
  - Initializes port system
- *Ptcreate*
  - Creates a new port
  - Argument specifies number of messages
- *Ptsend*
  - Sends a message to a port
- *Ptrecv*
  - Retrieves a message from a port

Xinu Port Functions
(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

Xinu Ptinit (part 1)

1. Make each port free
2. Form the linked list of free nodes
3. Initialize *ptnextid*

```
/* 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("pinit - insufficient memory");
        }
```

allocate a block of 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;
}
```

The search will start when a new port is needed.

Give the index in array

Xinu Ptcreate (part 1)

Specify the maximum count of outstanding messages that the port will allow

```
/* 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;
}
```

Allocate an entry in the port table from

Return a port identifier (port ID) to its caller

Xinu Ptsend (part 1)

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

#include <xinu.h>
```

1. Enqueue the message
2. Signal the receiver semaphore
3. Return

```
/*------------------------------------------------------------------------
 * ptsend - Send a message to a port by adding it to the Waiting queue
 *------------------------------------------------------------------------
 */
```

Passing a port ID as an argument

```
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;
        }
```

Specify a valid port ID

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;
```

A local copy of the sequence number

1. Wait sender semaphore
2. Verify that the port is still allocated
3. The sequence number agrees.

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;
}
```

Enqueues messages in FIFO order

Point to a new node after the node has been added to the list

Signal the receiver semaphore after added a new message

Xinu Ptrecv (part 1)

1. Remove a message from a specified port
2. Return the message to its caller

```
/* 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;
        }
```

1. Wait until a msg is available
2. Verify that the port has not been deleted or reused

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

Record the value of the message

```
        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 Reset and Deletion

- *Ptreset* or *ptdelete*
  - Disposes of existing messages, if the port contains any
  - Unblocks processes that are waiting
    - To send
    - To receive
- Semaphores are either reset or deleted
- Processes are informed that an abnormal termination occurred

How should the port system dispose of waiting messages?

Disposing of Messages

- Needed during reset and deletion
- Alternatives
  - Fixed set of choices
  - Allow arbitrary processing
- Arbitrary processing: Permit a caller to specify how to dispose of messages
  - More general
  - Must allow caller to specify disposition function as argument to *ptreset* or *ptdelete*
  - Disposition function is called for each existing message

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;
}
```

- Perform the work of clearing messages and waiting processes
- Place the port in a “limbo” state (PT_LIMBO).

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       */
```

Waiting processes can tell that the port has changed when they awaken

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;
}
```

Delete or reset the semaphores given by its second argument

Concurrency and Message Disposition

- Disposition routine
  - Is specified by user
  - May reschedule
- Example
  - Message to be deleted contains a pointer to a buffer
  - Disposition routine calls *freebuf to* release a buffer back to the pool.

Consequence: concurrency problems may arise

Semaphore Reset and Deletion

- Resetting or deleting a port will reset or delete the semaphores
- If processes are blocked on the semaphore, they will become ready
- The rescheduling invariant means a higher priority process may execute
- Additional processes may attempt to use the port

Consequence: we need to handle attempts to use the port concurrently during reset or deletion

Three Mechanisms for Handling Reset (1/3)

- Accession numbers: assign a sequence number to a port
  - Increment sequence
    - When port created and
    - When port deleted or reset
  - Have *ptsend* and *ptrecv* record sequence number when operation begins and check sequence number after *wait* returns
  - If sequence number changed, port was reset, so operation should abort

Three Mechanisms for Handling Reset (2/3)

- New state for the port: assign each port a *state* variable
  - Values of the state variable
    - *PTFREE* if not in use
    - *PTALLOC* if in use
    - *PTLIMBO* if in transition
  - Have *ptsend* and *ptrecv* examine state variable
  - If state is *PTLIMBO* port is being reset or deleted and cannot be used

Three Mechanisms for Handling Reset (3/3)

- Deferred rescheduling: temporarily postpone scheduling decisions
  - Call *resched_cntl(DEFER_START)* at start of reset or delete
  - Call *resched_cntl(DEFER_START)* after all operations are performed

Summary (1/2)

- Xinu offers a low-level message passing mechanism
  - Process-to-process
  - Only one message outstanding
- Xinu offers a high-level message passing mechanism
  - Dynamically created ports
  - Number of messages and message size fixed when port created
  - Arbitrary senders and receivers
  - Synchronous interface

Summary (2/2)

- Port reset /deletion is tricky because
  - Unblocked processes may execute
  - New processes may attempt to use the port
  - Three techniques can handle transition
    - Sequence number informs waiting processes that port is being reset or deleted
    - Limbo state prevents new processes from using the port while it is being reset or deleted
    - Deferred rescheduling