PART 6

Inter-Process Communication
Location Of Inter-process Communication In The Hierarchy
Inter-process Communication

- Used for
  - Exchange of (nonshared) data
  - Coordination

- General technique: *message passing*
Two Approaches To Message Passing

• Approach #1
  – Message passing is one of many services
  – Messages are separate from I/O and process synchronization services
  – Implemented using lower-level mechanisms, such as semaphores

• Approach #2
  – The entire operating system is message-based
  – Messages, not function calls, provide the fundamental building block
  – Messages, not semaphores, used for process synchronization
Design Of A Message Passing Facility

- To understand the issue, begin with a trivial message passing facility
- Allow a process to send a message directly to another process
- In principle, the design should be straightforward
- In practice, many design decisions arise
Message Passing Design Decisions

- Are messages fixed or variable size?
- How many messages outstanding at a given time?
- Where are messages stored?
- How is a recipient specified?
- Does a receiver know the sender’s identity?
- Are replies supported?
- Is the interface synchronous or asynchronous?
Synchronous vs. Asynchronous Interface

- Synchronous interface
  - Blocks until operation performed
  - Easy to understand / program
  - Extra processes can be used to obtain asynchrony
Synchronous vs. Asynchronous Interface (continued)

• Asynchronous interface
  – Starts an operation
  – Allows initiating process to continue execution
  – Notification
    * Arrives when operation completes
    * May entail abnormal control (e.g., software interrupt)
  – Polling can be used to determine status
Why Is A Message Passing Facility Difficult To Design?

- Interacts with
  - Process coordination subsystem
  - Memory management subsystem
- Affects user’s perception of system
Xinu Inter-process Message Passing

- Simple, low-level mechanism
- Direct process-to-process communication
- One-word messages
- One-message buffer
- Synchronous, buffered reception
- Asynchronous transmission and “reset” operation
Xinu Inter-process Message Passing (continued)

- Three functions
  
  ```c
  send(msg, pid);
  
  msg = receive();
  
  msg = recvclr();
  ```

- Only `receive` blocks

- Message stored in receiver’s process table
Xinu Inter-process Message Passing (continued)

- First-message semantics
  - First message sent to a process is stored until it has been received
  - Subsequent attempts to send fail

- Typical idiom

```c
recvclr();  /* prepare to receive a message */
...
/* allow other processes to send messages */
msg = receive();
```

- *Receive* returns first message that was sent
New Process State Needed

- While receiving a message, a process is not
  - Executing
  - Ready
  - Suspended
  - Waiting on a semaphore
- Therefore, a new state is needed for message passing
- Named RECEIVING
- Entered when receive called
State Transitions With Message Passing
/* 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];
    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;
}
Xinu Code For Message Transmission (part 1)

/* send.c - send */

#include <xinu.h>

/*-----------------------------------------------
 * send - pass a message to a process and start recipient if waiting
 *-----------------------------------------------
 */
system 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;
    }
Xinu Code For Message Transmission (part 2)

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;
Xinu Code For Clearing Messages

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

    restore(mask);
    return msg;
}
Summary

- Inter-process communication
  - Implemented by message passing
  - Can be synchronous or asynchronous
- Synchronous interface is the simplest
- Xinu uses synchronous reception and asynchronous transmission