7

Message Passing

Message passing refers to a form of inter-process communication in which one process requests that the operating system send data directly to another. In some systems, processes deposit and retrieve messages from named ‘‘pickup points’’; in others, each message must be addressed directly to a process. Message passing is both convenient and powerful, and many systems use it as the basis for all communication. For example, operations like sending data to a terminal or across a network to another machine can all be designed on top of message passing primitives.

Messages also provide a form of process coordination because the receiver can delay until the arrival of the next message. The chief difference between coordinating with messages and semaphores is that semaphores require precise synchronization between the processes that *wait* and those that *signal*, because there must be a call of *wait*(*s*) for every call to *signal*(*s*). By contrast, message passing can be unsynchronized. Unsynchronized messages are easier to use if a process does not know how many messages it will receive, when they will be sent, or which processes will send them. For example, a process driving the video display uses messages sent from other processes to inform it when there are characters available to display.

7.1 Message Passing In PC-Xinu

PC-Xinu supports two forms of message passing that serve to demonstrate two design approaches. This chapter deals with the first form, messages passed from one process directly to another. Chapter 15 discusses the second form, messages left at rendezvous points. Separating messages into two classes has the advantage of making process-to-process messages more efficient, but it has the disadvantage of requiring the user to know the destination of messages when writing programs. (Readers with special interest in message passing facilities should think about the potential benefits and liabilities of unifying all message passing as they read this material.)

Process-to-process message passing has been carefully designed to ensure that processes do not block (i.e., delay) while sending messages, and waiting messages do not consume all of memory. To make these guarantees, the message passing facility limits each message to one word (the size of an integer or pointer) and permits only one unreceived message per process at any time. The implementation of these restrictions is well-defined: if several messages are sent to a process before it attempts to receive any of them, only the *first* message will be received. Thus, a process can use message passing to determine which of several events completed first, by having them each send a unique message upon completion.

Five PC-Xinu system calls manipulate messages: *receive*, *recvclr*, *send*, *sendf* and *sendn*. *Send* takes a message and a process id as arguments and delivers the message to the specified process. *Receive* waits for a message to arrive and then returns that message to its caller; it requires no arguments. *Recvclr* is the asynchronous analog of *receive*; it never waits for a message to arrive. If the process has a message when it calls *recvclr*, the call returns the message exactly like *receive*. But if no message is waiting, *recvclr* returns the value *OK* to its caller without delaying to wait for a message to arrive. As the name implies, *recvclr* is often used to clear away any old messages that might be waiting. *Sendf* and *sendn* are similar to *send*; *sendf* forces delivery of the message even if there are other messages pending (by destroying any existing messages), while *sendn* behaves exactly like *send* but does not force a reschedule. *Sendf* is used to send an urgent message which must not be ignored. *Sendn* is used in interrupt service routines where a reschedule may not be appropriate.

Again, the question arises: ‘‘in what state should a process be while waiting for a message?’’ Because waiting for a message differs from waiting for a semaphore, waiting for the CPU, suspended animation, or currently executing, none of the existing states exactly solves the problem. So, it is time to add another state to our design. The new state, ‘‘waiting to receive a message,’’ is referenced in the software with the symbolic constant *PRRECV*. Adding it to the other states produces the transition diagram shown in Figure 7.1.

7.2 Implementation Of Send

Send must store messages where the recipient can *receive* them. They cannot be kept in the sender’s memory because the sending process might exit before the message was received. They cannot be kept in the recipient’s memory because allowing the sender to write into it poses a security threat. We have solved the problem by allocating space for messages in the process table entry.

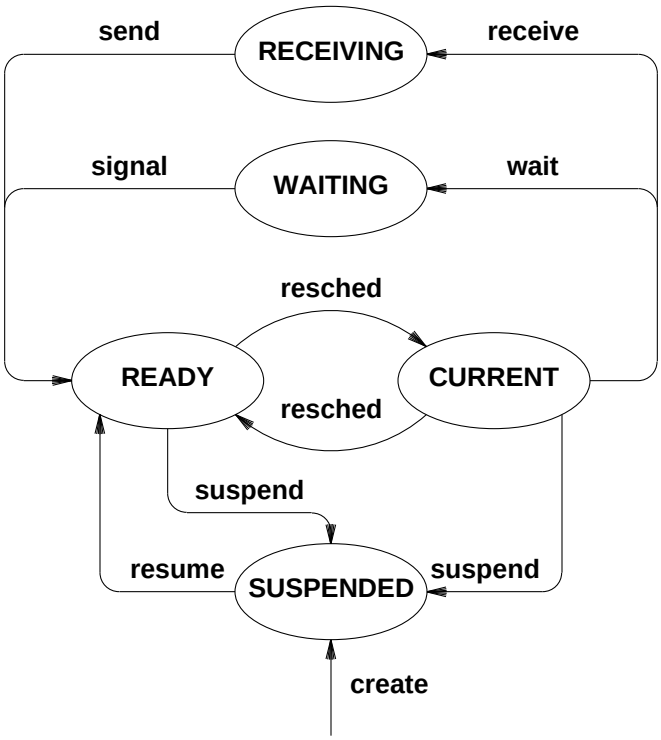

Figure 7.1 Process state transitions for the 'receiving' state

To deposit a message, *send* first checks that the specified recipient process exists. It verifies that the recipient does not have a message outstanding by examining the *phasmsg* field of its process table entry. If the process has no outstanding messages, *send* deposits the new message in the *pmsg* field and makes the *phasmsg* field nonzero to indicate that a message is waiting. Finally, if the process is waiting the arrival of a message, *send* moves it to the ready list, enabling it to access the message and continue execution.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  send  --  send a message to another process
 *------------------------------------------------------------------------
 */
SYSCALL send(pid, msg)
int     pid;
int     msg;
{
        struct  pentry  *pptr;          /* receiver's proc. table addr. */
        int     ps;

        disable(ps);
        if (isbadpid(pid) || ( (pptr = &proctab[pid])->pstate == PRFREE)
           || pptr->phasmsg != 0) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;               /* deposit message              */
        pptr->phasmsg++;
        if (pptr->pstate == PRRECV) {   /* if receiver waits, start it  */
                ready(pid);
                resched();
        }
        restore(ps);
        return(OK);
}
```

The implementations of *sendf* and *sendn* are straightforward modifications of *send*.

```
/* sendf.c - sendf */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  sendf  --  sendf a message to another process, forcing delivery
 *------------------------------------------------------------------------
 */
SYSCALL sendf(pid, msg)
int     pid;
int     msg;
{
        struct  pentry  *pptr;
        int     ps;

        disable(ps);
        if (isbadpid(pid)||((pptr= &proctab[pid])->pstate == PRFREE)) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;
        pptr->phasmsg++;
        if (pptr->pstate == PRRECV) {
                ready(pid);
                resched();
        }
        restore(ps);
        return(OK);
}
```

```
/* sendn.c - sendn */

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  sendn  --  send a message to another process, but do not reschedule
 *------------------------------------------------------------------------
 */
SYSCALL sendn(pid, msg)
int     pid;
int     msg;
{
        struct  pentry  *pptr;          /* receiver's proc. table addr. */
        int     ps;

        disable(ps);
        if (isbadpid(pid) || ( (pptr= &proctab[pid])->pstate == PRFREE)
           || pptr->phasmsg != 0) {
                restore(ps);
                return(SYSERR);
        }
        pptr->pmsg = msg;               /* deposit message              */
        pptr->phasmsg++;
        if (pptr->pstate == PRRECV)     /* if receiver waits, start it  */
                ready(pid);
        restore(ps);
        return(OK);
}
```

7.3 Implementation Of Receive

A process, *P*, calls *receive* (or *recvclr*) to obtain a message that has been sent to it. *Receive* examines the *phasmsg* field of its process table entry to determine if there is a message waiting. If not, it changes *P* to the receiving state and calls *resched*, allowing other processes to run. Eventually, when another process, *Q*, sends *P* a message, *send* places *P* back on the ready list. When *P* executes, the call to *resched* returns, allowing *receive* to pick up the message and return it to the caller.

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  receive  --  wait for a message and return it
 *------------------------------------------------------------------------
 */
SYSCALL receive()
{
        struct  pentry  *pptr;
        int     msg;
        int     ps;

        disable(ps);
        pptr = &proctab[currpid];
        if (pptr->phasmsg == 0) {       /* if no message, wait for one  */
                pptr->pstate = PRRECV;
                resched();
        }
        msg = pptr->pmsg;               /* retrieve message             */
        pptr->phasmsg = 0;
        restore(ps);
        return(msg);
}
```

Recvclr operates much like *receive* except that it always returns immediately. The implementation is straightforward:

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

#include <conf.h>
#include <kernel.h>
#include <proc.h>

/*------------------------------------------------------------------------
 *  recvclr  --  clear messages, returning waiting message (if any)
 *------------------------------------------------------------------------
 */
SYSCALL recvclr()
{
        int     ps;
        int     msg;

        disable(ps);
        if (proctab[currpid].phasmsg) {         /* existing message?    */
                proctab[currpid].phasmsg = 0;
                msg = proctab[currpid].pmsg;
        } else
                msg = OK;
        restore(ps);
        return(msg);
}
```

7.4 Summary

Message passing allows one process to send information to another. This chapter explored a simple form of interprocess message passing that used the central process table as an exchange point. The chief advantages of such an implementation are small size and efficient code; the chief disadvantage is the limitation to one outstanding message per process. Later chapters will explore a generalization of message passing in which processes rendezvous at common message exchange points called ports.

FOR FURTHER STUDY

Brinch Hansen [1970, 1972] introduced the notion of message passing and showed how it can be used in the RC 4000 system. The text by Peterson and Silberschatz [1983] surveys the area, discussing the advantages and disadvantages of allowing multiple messages to be enqueued for a receiver.

EXERCISES

7.1 Set up an environment to test the message-passing primitives described in this chapter. How can you ensure that processes send messages in a particular order for test purposes?

7.2 Write a program that prints a prompt and then loops, printing the prompt again every 8 seconds until someone types a character. (Hint: *sleep*(8) delays the calling process for 8 seconds).

7.3 It is often best to try a new facility before installing it. Assume *send* and *receive* did not exist, and write experimental versions without using the *PRRECV* process state. (Hint: *suspend* and *resume* almost suffice; make sure that the resumption came from *send*).

7.4 PC-Xinu records the first message sent to a process and rejects others. When is message order important?

7.5 Implement versions of *send* and *receive* that record all messages.

7.6 Discuss the use of a message passing scheme in which each process can have at most *k* outstanding messages.

7.7 Discuss a message passing mechanism in which each process is allowed at most *k* outstanding messages, with the added restriction that successive calls to *send* block (e.g., by waiting on a semaphore) until the receiving process makes room for additional messages by receiving some of those that are waiting.

7.8 Investigate systems in which the innermost layer implements message passing instead of context switching. What is their chief liability?

7.9 What facility in PC-Xinu handles the case where a process wants to receive the most recent message sent, instead of the first message sent?