CAPITOLO 7 - SCAMBIO DI MESSAGGI

Lo scambio di messaggi è una forma di comunicazione nel quale un processo richiede al sistema operativo di mandare dei dati direttamente ad un altro processo. In alcuni sistemi, i processi depositano e recuperano i messaggi da cosiddetti “pickup points”; in altri, ogni messaggio deve essere indirizzato direttamente al processo. Tale tecnica è sia conveniente che potente, e molti sistemi la usano come base per tutte le comunicazioni. Per esempio, operazioni come inviare dati a un terminale o attraverso la rete ad un’altra macchina possono essere sviluppate usando primitive dello scambio di messaggi.

I messaggi forniscono anche una forma di sincronizzazione tra processi perché il ricevente può sospendersi fino all’arrivo del prossimo messaggio. La principale differenza tra la sincronizzazione con l’uso dei messaggi e quella con i semafori consiste nel fatto che questi ultimi necessitano un preciso cooridnamento tra processi che effettuano operazioni di *wait* e quelli che usano invece la *signal*, perché ci deve essere una chiamata a *wait(s)* per ogni chiamata a *signal(s).* Differentemente, lo scambio di messaggi può essere effettuato anche senza alcuna sincronizzazione. Quest’ultima è una tecnica più semplice da usare se un processo non può conoscere il numero di messaggi che riceverà, quando gli saranno inviati, o quali processi glieli spediranno. Per esempio, un processo che guida il display del video usa messaggi mandati da altri processi per sapere quando un carattere è disponibile per essere visualizzato.

7.1 Scambio di messaggi in PC-Xinu

PC-Xinu supporta due forme di scambio di messaggi. Questo capitolo tratta della prima forma: messaggi passati da un processo direttamente a un altro. Il Capitolo 15 discute invece la seconda: messaggi lasciati in punti di incontro. Separare in due classi i messaggi ha il vantaggio di renderne lo scambio più efficiente, ma richiede all’utente la conoscenza della destinazione dei messaggi quando scrive i programmi. (I lettori con un interesse speciale potrebbero pensare i potenziali benefici e responsabilità nell’unificare lo scambio dei messaggi dopo aver letto questo materiale.)

Lo scambio di messaggi tra processi è stato progettato con cura per garantire che i processi non si blocchino (ad es. sospendersi) mentre inviano messaggi, e i messaggi sospesi non consumino tutta la memoria. Per garantire ciò, ogni messaggio è stato limitato a una word (la dimensione di un intero o un puntatore) ed è permesso solo l’invio di un messaggio per processo a volta. L’implementazione di queste restrizioni è stata ben definita: se parecchi messaggi sono inviati ad un processo prima che esso provi a riceverne uno di essi, solo il primo sarà ricevuto. Così, un processo può usare lo scambio di messaggi per determinare quale tra gli eventi viene completato per primo, spedendo un messaggio dopo il completamento.

Cinque chiamata di sistema manipolano i messaggi in PC-Xinu: *receive, recvclr, send, sendf and sendn. Send* prende un messaggio e un identificatore di processo come argomenti e spedisce il messaggio ad un processo specificato. *Receive* attende l’arrivo di un messaggio che poi viene ritornato al suo chiamante; essa non richiede argomenti. *Recvclr* è l’analoga *receive* asincrona; essa non attende mai che arrivi un messaggio. Se un processo ha un messaggio pendente quando chiama *recvclr*, la chiamata ritorna il messaggio esattamente come una *receive*. Ma se nessun messaggio è in attesa, *recvclr* ritorna il valore OK al suo chiamante senza sospendersi per l’arrivo di un messaggio. Come il nome indica *recvclr* è spesso usata per pulire i vecchi messaggi che dovrebbero essere in attesa. *Sendf* e *sendn* sono simili a *send*; *sendf* forza l’invio del messaggio anche se ce ne sono altri sospesi, (distruggendo ogni messaggio esistente), mentre *sendn* si comporta esattamente

come *send* ma non forza la schedulazione. *Sendf* è usata per i messaggi urgenti che non dovrebbero essere ignorati. *Sendn* è usata nelle routine di servizio di interrupt quando la schedulazione può non essere appropriata.

Una questione sorge: “in quale stato dovrebbe essere un processo che attende un messaggio?” Poiché l’attesa di un messaggio differisce da quella per un semaforo o della CPU, differisce dallo stato sospeso e in esecuzione, nessuno di essi ci soddisfa. Perciò è arrivato il momento di aggiungere un nuovo stato al nostro progetto. Il nuovo stato “attesa per la ricezione di un messaggio” è riferito nel software dalla costante simbolica PRRECV. Aggiungendola agli altri si produce il diagramma di transizione mostrato in figura 7.1

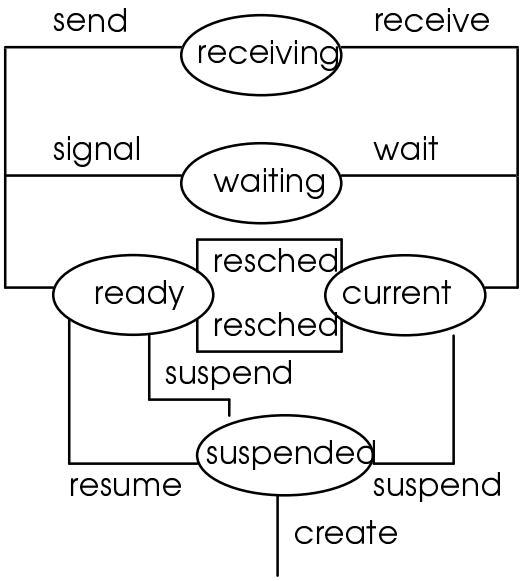

Figura 7.1 Transizione di stato per una “receiving”

7.2 Implementazione di Send

Send deve memorizzare i messaggi in un luogo dove il destinatario possa riceverli. Essi non possono essere tenuti nella memoria del mittente perché tale processo deve uscire prima che il messaggio sia ricevuto. Inoltre non possono essere tenuti in nella memoria del destinatario perché sorgerebbero problemi di sicurezza. Abbiamo risolto il problema allocando spazio per i messaggi nell’elemento della tabella dei processi.

Per depositare un messaggio, send prima controlla che il processo destinatario specificato esista. Esso verifica che il destinatario non abbia messaggi straordinari esaminando il campo *phasmsg* nel corrispondente elemento della tabella dei processi. Se non ve ne sono, *send* deposita il nuovo messaggio nel campo *pmsg* e mette un valore diverso da zero nel campo *phasmsg* per indicare che un messaggio è in attesa. Se il processo stava aspettando l’arrivo di un messaggio, *send* lo rimette nella lista dei processi pronti, gli permette l’accesso al messaggio e il proseguimento dell’esecuzione.

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

#include <conf.h>
```

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

/* ------------------------------------------------------------------------
 *  send -- spedisce un messaggio a un altro processo
 * ------------------------------------------------------------------------
 */
SYSCALL send(pid, msg)
int  pid;
int  msg;
{
     struct pentry   *pptr;          /* indirizzo della tabella dei */
                                     /* processi del ricevente */
     int  ps;

     disable(ps);
     if (isbadpid(pid) || ((pptr=&proctab[pid]->pstate==PRFREE)
     || pptr->phasmsg!=0) {
          restore(ps);
          return(SYSERR);
     }
     pptr->pmsg = msg;                    /* deposita il messaggio */
     pptr->phasmsg++;
     if (pptr->pstate == PRRECV) {        /* se il ricevente era in */
               ready(pid);                /* attesa, lo inizializza */
          resched();
     }
     restore(ps);
     return(OK);
}
```

L’implementazione di sendf e sendn sono delle modifiche di send.

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

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

/* ------------------------------------------------------------------------
 * sendf --  spedisce un messaggio a un processo forzando la consegna
 * ------------------------------------------------------------------------
 */
SYSCALL sendf(pid, msg)
int  pid;
int  msg;
{
     struct pentry   *pptr;
     int   ps;

     disable(ps);
     if (isbadpid(pid)||(pptr=&proctab[pid])->pstate==PRFREE) {
     /* manca il controllo su phasmsg */
     restore(ps);
     return(SYSERR);
     }
     pptr->pmsg = msg;
```

```
	if (pptr->pstate == PRRECV) {
		ready(pid);
		resched();
	}
	restore(ps);
	return(OK);
}


/* sendn.c - sendn /*

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

/* ------------------------------------------------------------------------
 * sendn -- spedisce un messaggio a un altro processo, ma non effettua
 * la schedulazione
 * ------------------------------------------------------------------------
 */
SYSCALL sendn(pid, msg)
int	pid;
int	msg;
{
	struct pentry	*pptr;		/* indirizzo del ricevente nella */
					/* tabella dei processi */
	int	ps;

	disable(ps);
	if (isbadpid(pid) || ((pptr=&proctab[pid]->pstate==PRFREE)
	|| pptr->phasmag != 0) {
		restore(ps);
		return(SYSERR);
	}
	pptr->pmsg = msg;		/* deposita il messaggio */
	pptr->phasmsg++;
	if (pptr->pstate == PRRECV)	/* se il ricevente attende */
		ready(pid);		/* il messaggio, lo inizializza */
	restore(ps);
	return(OK);
}
```

7.3 Implementazione di Receive

Un processo P chiama *receive* (o *recvclr*) per ottenere un messaggio che gli è stato mandato. *Receive* esamina il campo *phasmsg* nell'elemento della tabella dei processi per determinare se c'è un messaggio inviato al processo. Se non ve ne sono, mette P nello stato di receiving e chiama *resched*, permettendo l'esecuzione di altri processi. Eventualmente, quando un altro processo Q spedisce un messaggio a P, *send* lo rimette nella lista dei processi pronti. Quando P esegue (la chiamata di ritorno da resched) *receive* prende il messaggio e lo ritorna al suo chiamante.

```
SYSCALL receive()
{
	struct pentry	*pptr;
	int	msg;
	int	ps;
```

```
      disable(ps);
      pptr = &proctab[currpid];
      if (pptr->phasmsg == 0) {       /* se non vi sono messaggi, */
            pptr->pstate = PRRECV;    /* attendine uno */
            resched();
      }
      msg = pptr->pmsg;               /* riottiene il messaggio */
      pptr->phasmsg = 0;
      restore(ps);
      return(msg);
}
```

Recvclr opera come *receive* tranne per il fatto che essa ritorna immediatamente. Eccone l'implemetazione:

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

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

/* ------------------------------------------------------------------------
 *  recvclr  --  riceve un messaggio, se ve ne sono, e ritorna
 * ------------------------------------------------------------------------
 */
SYSCALL recvclr()
{
      int   ps;
      int   msg;

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