CAPITOLO 3 - LA MANIPOLAZIONE DI LISTE E CODE

L'elaborazione di liste concatenate è fondamentale in un sistema operativo - sembra pervadere ogni componente. Questo capitolo introduce un insieme di procedure formanti la spina dorsale del trattamento delle liste concatenate nel gestore dei processi di PC-Xinu. Le routine descritte di seguito sono usate per mantenere code ordinate in base al tempo di inserimento e code ordinate per priorità. Esse eseguono azioni del tipo: inserire un elemento nella in fondo a una lista, inserire un elemento in una lista ordinata, rimuovere un elemento dalla testa della lista, allocare una nuova lista.

Le routine dedicate alle liste concatenate, inserite in questo capitolo, forniscono una buona introduzione al linguaggio di programmazione C e alle convenzioni di programmazione usate nel resto di questo libro. Iniziare con questi programmi è di aiuto perché trattano argomenti familiari ed inoltre un solo processo può eseguirli in un dato momento. In questo modo il lettore può pensare il codice inerente come se facesse parte di un programma sequenziale - non c'è bisogno di preoccuparsi dell'eventuale interferenza che potrebbe apportare la presenza di altri processi che vengono eseguiti in concorrenza.

3.1 Liste concatenate di processi

Il gestore dei processi tratta oggetti chiamati *processi*, muovendoli da/a svariate liste frequentemente. Le voci attualmente memorizzate in queste liste sono dei piccoli, non negativi, interi chiamati identificatori di processi (o process ids); useremo il termine "processo", "identificatore di processo", e "process id" in modo intercambiabile per tutto questo capitolo. La costante *NPROC* dà la portata massima di identificatori di processi. Se può aiutare, assumiamo *NPROC* avente valore 30 e la voce che deve essere memorizzata fa parte dell'intervallo di interi da 0 a 29.

Un'architettura precedente usava liste di processi aventi strutture dati differenti. Alcune di queste erano code di tipo *FIFO* (primo entrato, primo uscito), altre erano ordinate in base a una chiave; alcune usavano una singola concatenazione altre doppia concatenazione. Tali requisiti che abbiamo formulato hanno indotto i progettisti a centralizzare l'elaborazione delle liste concatenate in un'unica struttura dati; in questo modo il codice viene snellito dal controllo di ogni singolo caso.

Per adattare tutti i requisiti, abbiamo scelto una rappresentazione nella quale: tutte liste hanno una doppia concatenazione (ogni nodo punta al suo predecessore e successore), ogni nodo contiene una chiave (sebbene tale valore non sia usato nalle liste di tipo *FIFO*) e ogni lista possiede la rispettiva testa e coda. Fondamentalmente, esse hanno la forma mostrata in figura 3.1.

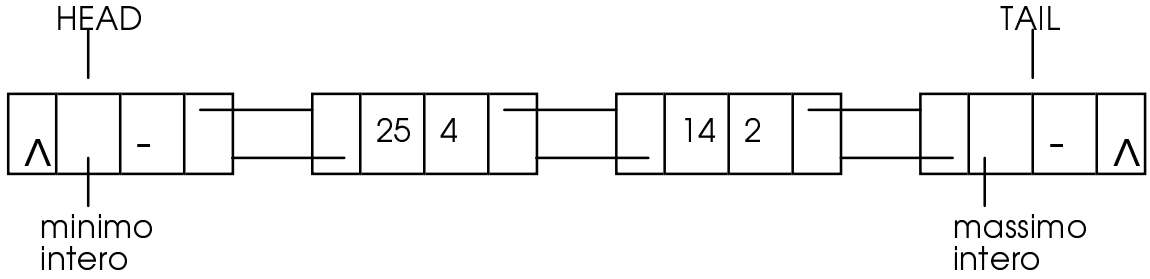

Figura 3.1 - Una lista con doppia concatenazione contenente 4 (chiave=25) e 2 (chiave=14).

Il campo chiave nella testa contiene il minimo intero possibile; quello della coda contiene invece il massimo intero possibile. Come previsto, il successore della coda e il predecessore della testa sono puntatori nulli. Quando una lista è vuota, il successore della testa è la coda, e il predecessore della coda è la testa.

Il diagramma mostrato sopra è solamente di tipo logico. Nella pratica, la richiesta di memoria è stata ridotta memorizzando i dati in modo implicito. Tale ottimizzazione è possibile grazie alla seguente proprietà:

In un dato momento, un processo appare al massimo su una singola lista.

Per comprendere come i dati possano essere memorizzati implicitamente vediamo la figura 3.2. Essa mostra un array chiamato "struttura Q", dove ogni elemento contiene tra campi: una chiave, un puntatore al prossimo elemento e uno al precedente. Le posizioni da 0 a *NPROC - 1* corrispondono agli identificatori di processo inseriti nella lista; le posizioni a partire da *NPROC* vengono sfruttate da teste e code delle liste. E' possibile riservare soltanto un nodo per ogni elemento perché l'intervallo dei valori è piccolo (tipicamente *NPROC* = 30) e nessun valore può apparire su più liste simultaneamente. Una vista più da vicino al codice dovrebbe chiarire queste operazioni.

	key	**next**	**prev**
0			
1			
2	**14**	**33**	**4**
3			
4	**25**	**2**	**32**
5			
		. . .	
NPROC - 1			
NPROC			
		. . .	
Head at 32	**MININT**	**4**	**-1**
Tail at 33	**MAXINT**	**-1**	**2**
		. . .	

Figura 3.2 - La lista della figura 3.1 memorizzata nell'array q

3.2 Implementazione della struttura Q

In C, la figura sopra descritta è una array *q* di strutture *qent*. Il file *q.h* contiene le dichiarazioni di *q* e *qent*.

```
/* q.h - firstid, firstkey, isempty, lastkey, nonempty */
```

```
/* dichiarazione della struttura q, costanti, e procedure in linea */

#ifndef     NQENT
#define     NQENT NPROC + NSEM + NSEM + 4
#endif

struct      qent  {          /* una per processo più due per lista */
      int   qkey;            /* chiave in base alla quale la coda è ordinata*/
      int   qnext;           /* puntatore al prossimo processo o alla coda */
      int   qprev;           /* puntatore al precedente processo o alla testa*/
      };

extern      struct      qent q[];   /* array q */
extern      int   nextqueue;

/* procedure per la manipolazione di liste */

#define isempty(list)   (q[(list)].qnext >= NPROC)    /* 1 se lista vuota */
#define nonempty(list)  (q[(list)].qnext < NPROC)     /* 1 se lista non vuota */
#define firstkey(list)  (q[q[(list)].qnext].qkey)     /* ritorna prima chiave */
#define lastkey(tail)   (q[q[(tail)].qprev].qkey)     /* ritorna ultima chiave */
#define firstid(list)   (q[(list)].qnext)             /* ritorna l'identificatore
del primo processo in lista */

#define EMPTY       (-1)      /* puntatore NULL */

extern char *deq(int);
extern char *seeq(int);
```

Ogni elemento dell'array corrisponde a una testa o una coda di una lista oppure un elemento che deve essere inserito. Per il momento ricordiamo che i valori memorizzati in una lista sono gli identificatori di processo (interi) appartenenti all'intervallo 0, NPROC-1. Assumiamo implicito il fatto che in tutto il codice gli elementi da *q[0]* a *q[NPROC-1]* corrispondano ai loro process ids, mentre da *q[NPROC]* a *q[NQENT-1]* corrispandono le teste e le code delle liste.

La costante simbolica *NQENT* definisce il numero di elementi dell'array *q*; il valore "*NPROC+NSEM+NSEM+4*" allocherà lo spazio sufficiente nella struttura *q* per: *NPROC* processi, (teste e code per) *NSEM* code di semafori, una lista per i pronti, una per quelli sospesi.

Il contenuto degli elementi nell'array *q* è definito dalla struttura *qent*. (Questo file contiene la dichiarazione della forma degli elementi nell'array *q*; vedremo la definizione del suo contenuto nel capitolo 13). Il campo *qnext* punta in avanti, *qprev* punta indietro mentre *qkey* contiene una chiave intera per il nodo. Quando i puntatori non contengono indici validi, viene assegnato il valore *EMPTY*.

3.2.1 Funzioni Q in linea

Le funzioni *isempty* e *nonempty* sono funzioni booleane che, ricevendo come argomento l'indice della testa di una lista, testano se essa è vuota o meno. *Isempty* determina se una lista è vuota controllando il primo nodo; esso sarà un processo o la coda della lista; *nonempty* effettua il test opposto. Ricordiamo che un elemento è un processo se e solo se il suo indice è minore di *NPROC*.

Le altre funzioni sono altrettanto facili da comprendere. Le funzioni *firstkey*, *lastkey* e *firstid* ritornano rispettivamente la chiave del primo processo in una lista, la chiave dell'ultimo processo in una lista, o l'indice q del primo processo in una lista. Generalmente queste funzioni sono usate su liste non vuote, ma non danno problemi se lo sono perché *qkey* è sempre inizializzato.

3.2.2 La manipolazione di code FIFO

Per produrre una coda *FIFO* basta inserire gli elementi sempre in coda alla lista e rimuoverli dalla testa. Le procedure *enqueue* e *dequeue* del file *queue.c*, eseguono le operazioni *FIFO* in PC-Xinu. Il codice è coerente una volta aver capito come operano i puntatori. Le variabili *tptr* e *mptr* sono puntatori alla struttura *qent*. Le prime due istruzioni eseguibili in *enqueue* assegnano questi puntatori agli indirizzi di elementi di *q*; uno punterà alla coda della lista mentre l'altro al processo che deve essere inserito. Non appena l'indirizzo di un elemento è stato registrato, i singoli campi di una struttura sono riferiti tramite l'operatore "->".

```
/* queue.c - dequeue, enqueue */

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

/*------------------------------------------------------------------------
 *  enqueue  --  inserisce un elemento in coda alla lista
 *------------------------------------------------------------------------
 */
int enqueue(item, tail)
int item;                   /* elemento da inserire nella lista */
int tail;                   /* indice in q della coda della lista */
{
      struct      qent  *tptr;           /* punta alla coda della lista */
      struct      qent  *mptr;           /* punta al nuovo elemento */

      tptr = &q[tail];
      mptr = &q[item];
      mptr->qnext = tail;
      mptr->qprev = tptr->qprev;
      q[tptr->qprev].qnext = item;
      tptr->qprev = item;
      return(item);
}


/*------------------------------------------------------------------------
 *  dequeue  --  rimuove un elemento dalla lista e lo ritorna
 *------------------------------------------------------------------------
 */
int dequeue(item)
int item;
{
      struct      qent  *mptr;      /* puntatore all'elemento q della voce */

      mptr = &q[item];
      q[mptr->qprev].qnext = mptr->qnext;
      q[mptr->qnext].qprev = mptr->qprev;
      return(item);
}
```

Il file *queue.c* include tre altri file: *conf.h, kernel.h e q.h.* Il file *q.h* è necessario perché entrambe le procedure *enqueue* e *dequeue* fanno riferimento alla struttura q. Ma per quale motivo sono inclusi

gli altri due? Sembra che il codice non li necessiti. Tuttavia, *queue.c* include il file *q.h* che fa riferimento a costanti come *NPROC* e *NSEM* che sono definiti in *kernel.h* e *conf.h*. Come regola generale, le costanti più importanti sono stati collocate in questi due file perciò la maggior parte delle routine includono tali file.

3.3 Manipolazione di code con priorità

Il gestore dei processi spesso ha la necessità di selezionare, tra un insieme di processi, quello che ha la priorità maggiore, perciò le routine delle liste concatenate devono essere abilitate a mantenere insiemi di processi con una priorità associata. (Le priorità in PC-Xinu sono facili da comprendere: per adesso pensiamole come valori interi assegnati ai processi.) In generale, il lavoro di selezione del processo con priorità più alta è eseguito frequentemente rispetto all'inserimento o la cancellazione; così l'idea è di progettare una struttura dati che rendi la selezione efficiente rispetto all'inserimento.

Una varietà di strutture dati sono state concepite per memorizzare insiemi in cui la priorità è importante. Alcune strutture dati sono chiamate *priority queues* (code con priorità). Sebbene non tutte le "priority queues" usano *code* il termine descrive accuratamente l'implementazione di PC-Xinu - le *code con priorità* di Xinu sono soltanto liste concatenate nella quale i processi sono ordinati per priorità. Il processo con più alta priorità può essere trovato sempre in coda alla lista. Naturalmente, l'inserimento in una *coda a priorità* è più costoso di un tipo FIFO perché la lista deve essere scandita per determinare in quale posto deve essere allocata la nuova voce.

Quando appaiono molte voci o le inserzioni sono più frequenti rispetto al numero di volte in cui gli elementi sono estratti, usare liste lineari sarebbe non efficiente. Tuttavia, in un piccolo sistema come PC-Xinu, dove attendiamo 2 o 3 elementi in una *coda a priorità*, le semplici liste sono sufficienti.

Le procedure che mantengono ordinate le liste sono spiegate di seguito. *Insert* prende come argomento un identificatore di processo, un intero che indica la testa della lista nella struttura *q* e una priorità; essa inserisce il processo nella posizione corretta della lista. Il campo *qkey* di un processo è usato per memorizzare il processo con tale priorità. Per trovare la corretta locazione nella lista, *insert* cerca un elemento esistente con una chiave maggiore o uguale della chiave dell'elemento che deve essere inserito. Durante la ricerca l'intero *next* si muove lungo la lista. Ad ogni modo il ciclo termina perché la chiave della coda contiene l'intero più grande. Dopo aver trovato la corretta locazione, *insert* cambia i puntatori per unire il nuovo nodo alla lista.

```
/* insert.c - insert */

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

/*------------------------------------------------------------------------
 *  insert  --  inserisce un processo in una lista q ordinata per chiave
 *------------------------------------------------------------------------
 */
int insert(proc, head, key)
int proc;                       /* processo da inserire */
int head;                       /* indice q della testa della lista */
int key;                        /* chiave da usare per il processo */
{
      int   next;               /* percorre la lista */
      int   prev;

    next = q[head].qnext;
      while (q[next].qkey < key)    /* la coda ha come chiave MAXINT */
```

```
            next = q[next].qnext;
      q[proc].qnext = next;
      q[proc].qprev = prev = q[next].qprev;
      q[proc].qkey  = key;
      q[prev].qnext = proc;
      q[next].qprev = proc;
      return(OK);
}
```

Gli elementi di una coda *FIFO* possono essere estratti rimuovendoli dalla testa; in una *coda a priorità* possono pure essere estratti sia dalla testa che dalla coda. Le procedure *getfirst* e *getlast* provvedono a rimuovere elementi dalla coda. *Getfirst* prende come argomento l'indice della testa di una lista mentre *getlast* l'indice della coda. Se la lista è una coda a priorità, *getfirst* rimuove un elemento con la chiave minore e *getlast* quello con la chiave più grande. Per le code *FIFO*, *getfirst* rimuove l'elemento più vecchio dalla lista. Entrambe le routine ritornano l'indice dell'elemento rimosso. Questi valori ritornati sono identificatori di processo o il valore *EMPTY*.

```
/* getitem.c - getfirst, getlast */

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

/*------------------------------------------------------------------------
 *  getfirst  --  rimuove e ritorna il primo processo in una lista
 *------------------------------------------------------------------------
 */
int   getfirst(head)
      int   head;             /* indice q della testa della lista */
{
      int   proc;             /* primo processo nella lista */

      if ((proc=q[head].qnext) < NPROC)
            return( dequeue(proc) );
      else
            return(EMPTY);
}



/*------------------------------------------------------------------------
 *  getlast  --  rimuove e ritorna l'ultimo processo in una lista
 *------------------------------------------------------------------------
 */
int   getlast(tail)
      int   tail;             /* indice q alla coda di una lista  */
{
      int   proc;             /* ultimo processo nella lista */

      if ((proc=q[tail].qprev) < NPROC)
            return( dequeue(proc) );
      else
            return(EMPTY);
}
```

3.4 Inizializzazione di una lista

Sebbene le liste possano essere vuote, le procedure descritte assumono i nodi della testa e della coda come già inizializzati. Adesso consideriamo come creare una lista vuota. E' appropriato

l'inserimento di questo materiale proprio alla fine del capitolo perché mette in risalto un importante punto sulla progettazione dei processi:

L'inizializzazione è il passo finale nella progettazione.

Ciò può suonare strano perché non è possibile pensare di rimandare tale fase completamente, ma il punto è semplice: la progettazione di strutture dati necessitava in primis che il sistema fosse in esecuzione per delle prove e poi venivano fuori le modalità di inizializzazione. Partizionare la parte dello "stato stabile" da quella dello "stato transitorio" in un sistema aiuta ad evitare la tentazione a sacrificare un buon progetto per una semplice modalità di creazione.

Tale fase è eseguita sulla struttura *q* quando arriva una nuova richiesta. I programmi in esecuzione chiamano *newqueue* per creare una nuova lista. *Newqueue* alloca un numero pari di posizioni adiacenti nell'array *q* che vengono usate per i nodi della testa e della coda. Esso avvia la lista facendo puntare il successore della testa alla coda e il predecessore della coda alla testa. Altri puntatori contengono il valore EMPTY. Quando inizializza la testa e la coda *newqueue* setta anche i rispettivi campi chiave al più piccolo e grande intero possibile; in tal modo possono essere usati con una lista ordinata. Finalmente, *newqueue* restituisce l'indice della testa della lista al suo chiamante.

```
/* newqueue.c - newqueue */

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

/*------------------------------------------------------------------------
 *  newqueue  --  crea una nuova lista nella struttura q
 *------------------------------------------------------------------------
 */
int   newqueue()
{
      struct      qent  *hptr;           /*  indirizzo  della  testa  della  nuova
lista */
      struct      qent  *tptr;           /*  indirizzo  della  coda  della  nuova
lista */
      int   hindex, tindex;              /* indice di testa e coda */

      hptr = &q[ hindex=nextqueue++ ];   /* nextqueue è una variabile globale */
      tptr = &q[ tindex=nextqueue++ ];   /* che da la prossima pos. usata di q*/
      hptr->qnext = tindex;
      hptr->qprev = EMPTY;
      hptr->qkey  = MININT;
      tptr->qnext = EMPTY;
      tptr->qprev = hindex;
      tptr->qkey  = MAXINT;
      return(hindex);
}
```