CAPITOLO 8 - GESTIONE DELLA MEMORIA

La memoria principale si colloca in vetta tra le più importanti risorse che un sistema operativo gestisce perché di fondamentale importanza per l'esecuzione dei programmi. Il sistema operativo tiene traccia della locazione e dimensione dello spazio libero allocanto su domanda, e lo recupera non appena i processi vanno completandosi. In molti sistemi, dove la richiesta di memoria in un dato momento eccede il totale disponibile, il sistema operativo deve suddividere la memoria fisica tra processi in attesa di usarla. La suddivisione della memoria può gestirsi tramite *swapping*, dove interi processi sono scritti in memoria secondaria quando non stanno usando la CPU, oppure usando la tecnica di *paginazione*. Un sistema paginato divide ogni programma in piccole parti, chiamate pagine, aventi una fissata dimensione; tutte le pagine tranne le più recenti riferite sono mantenute invece in memoria secondaria. Il programmatore non deve preoccuparsi dello swapping o della paginazione perché il sistema esegue queste attività in modo trasparente per l'esecuzione dei programmi.

Spesso, i sistemi paginati sono più efficienti del suddividere la memoria tra processi - essi forniscono ad ogni processo un indipendente spazio di indizzamento. Il cosiddetto *spazio di indirizzamento virtuale* può essere più grande della memoria fisica montata sulla macchina; un sistema paginato ha il compito di mantenere su disco una immagine virtuale, muovendo in memoria principale solo un piccolo sottoinsieme di pagine riferite. Quando un processo si riferisce ad una locazione *i* nel suo spazio di indirizzamento, l'hardware consulta la tabella delle pagine per determinare se essa risiede correntemente in memoria. In caso contrario, il sistema sospende il processo e carica la pagina, scrivendone qualche altra nella memoria secondaria (se necessario) per fare spazio alla nuova. Finalmente, quando la pagina è stata caricata, il processo riprende l'esecuzione.

Per essere efficiente, la gestione della memoria (in particolare quella con paginazione) richiede un supporto hardware. Se un sistema per la gestione della memoria è ben progettato, il sistema operativo può usarlo per partizionare la memoria in modo tale che l'hardware prevenga che un processo legga o scriva la memoria allocata ad un altro processo. Un certo tipo di protezione è essenziale in ambienti dove i processi sono in competizione o dove la sicurezza è importante; è conveniente in quasi ogni ambiente perché aiuta ad individuare gli errori nei programmi.

8.1 Gestione della memoria nell'8088

Sfortunatamente l'hardware dell'8088 gestisce poveramente spazi di indirizzamenti multipli e non può proteggere i processi dagli altri. Di conseguenza lo spazio di sistema di Xinu e tutti i processi occupano porzioni dello stesso spazio di indirizzamento per il testo e i dati, disposti come mostrato in figura 8.1:

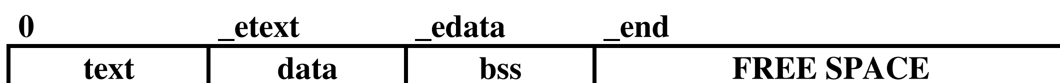

Figura 8.1 - Struttura di memorizzazione all'avvio di PC-Xinu

Il testo del programma occupa la parte più bassa della memoria, seguita dalle variabili globali nel segmento *data*. All'avvio del sistema PC-Xinu inizializza la variabile globale *maxaddr* alla locazione di memoria più alta indirizzabile nel segmento dati, in modo da poter determinare la dimensione dello spazio libero in un programma C.

Mantenere i processi utilizzando un singolo spazio dati è certamente non ideale, ma esso comporta alcuni vantaggi. Da una parte, i processi possono passare puntatori al sistema operativo facilmente, perché l'interpretazione di un indirizzo non dipende dal contesto del processo. L'abilità a condividere dati è un altro vantaggio: poichè i processi condividono variabili globali, essi possono scambiare grandi quantità di dati senza copiarli. Infine, un singolo spazio di indirizzamento per i dati rende le routine di gestione della memoria più semplici da implementare rispetto a quelle presenti in altri sistemi.

8.2 Requisiti della memoria dinamica in PC-Xinu

PC-Xinu richiede che il testo del programma e tutti dati globali rimangano residenti in memoria principale in ogni momento. Tuttavia, essi sono riferiti solamente per la parte di spazio richiesta da un processo in esecuzione. Ogni processo richiede spazio anche per lo stack (che mantiene la struttura delle procedure e le variabili locali) e l'heap (per altre variabili allocate dinamicamente). PC-Xinu alloca la memoria dello stack dagli indirizzi piu bassi dello spazio libero, producendo un'allocazione in esecuzione come mostrato in figura 8.2:

Figura 8.2 - Struttura di memorizzazione durante l'esecuzione

Questo capitolo esplora le procedure e strutture dati che gestiscono lo spazio libero, allocano spazio per la memorizzazione di stack e heap, e tengono traccia della memoria che è stata rilasciata. A questo livello, lo spazio libero è trattato come una risorsa esauribile - il sistema lo distribuisce fin quando le richieste possono essere soddisfatte. Un processo che non può ottenere memoria decide per se stesso se riprovare ancora. Un'allocazione esauriente lavora bene quando i processi cooperano a non consumare tutta la memoria. (Il capitolo 15 esplora un insieme di utility di alto livello per la gestione della memoria che ne previene l'esaurimento partizionandola e bloccando le richieste da parte dei processi fin quando non se ne ha a disposizione)

8.3 Procedure di alto livello per la gestione della memoria

Due procedure, *getmem* e *freemem*, ottengono e rilasciano spazio per lo stack del processo. Ricordiamo che *create* usava la procedura *getmem* per allocare lo spazio dello stack quando formava un nuovo processo. *Getstk* ottiene un blocco di memoria e ritorna il suo indirizzo. *Create* registra la dimensione e la locazione dello spazio allocato nell'elemento della tabella dei processi. Successivamente, quando il processo termina, *kill* chiama la procedura *freemem* per restituire alla lista libera la parte allocata per lo stack. Freemem prevede come argomento l'indirizzo del blocco che deve essere rilasciato e la sua dimensione. *Getmem* e *freemem* ottengono e rilasciano spazio per altri scopi di sistema.

Poiché solo *create* e *kill* allocano e liberano lo stack del processo, PC-Xinu garantische che questo spazio allocato ad un processo sarà rilasciato alla sua terminazione. Sfortunatamente, il sistema non garantisce il rilascio di altro spazio allocato da *getmem*, perché non registra l'allocazione di questo. Così, il peso di ritornare la parte di memoria usata da un heap è lasciata all'utente:

Un processo deve rilasciare la memoria che ha alloca dall'heap prima di uscire

Naturalmente, il ritorno dello spazio allocato non garantisce che l'heap sarà mai esaurito. La domanda potrebbe ancora eccedere la memoria a disposizione o lo spazio libero potrebbe essere frammentato in piccoli pezzi discontinui. Ma il rilascio di tale spazio evita un inutile esaurimento.

8.4 La posizione della memoria allocata

E' desiderabile avere stack ed heap allocati in poli opposti dello spazio disponibile. Poiché l'hardware fa crescere gli stack verso il basso, tale tecnica di allocazione è ideale per un singolo processo utente perché tutto la rimanente memoria li separerebbe. Se lo stack sconfina accidentalmente, esso invaderà lo spazio libero tra se stesso e l'heap invece dei dati. Quando più di un processo è esecuzione concorrente, la situazione non è piacevole. L'overflow dello stack di un processo corromperà i dati contenuti nello stack di un altro perché il sistema li alloca contiguamente. Infatti, questo è uno dei problemi più comuni in PC-Xinu. Gli esercizi nel capitolo 4 suggerivano una soluzione - piazzare un insolito valore alla base dello stack di ogni processo; in più resched dovrebbe controllare tale valore e la dimensione dello stack del processo corrente prima che il processo ricominci.

8.5 L'implementazione della gestione della memoria di PC-Xinu

PC-Xinu tiene assieme i blocchi di memoria liberi in una lista; la variabile globale *memlist* punta al primo blocco libero. Una importante invariante mantenuta da tutte queste procedure è:

I blocchi nella lista libera sono ordinati in ordine crescente di indirizzo

Ogni blocco contiene, nelle prime due words, un puntatore al prossimo blocco e la dimensione del blocco corrente. Il record *memlist* è dichiarato in modo tale da avere la stessa forma di tutti i blocchi sulla lista libera. Il file *mem.h* contiene le pertinenti definizioni in C:

```
/* mem.h - roundew, truncew, getstk, freestk */

/* ----------------------------------------------------------------------------
 * roundew, truncew - arrotonda o tronca indirizzi alla prossima word pari
 * ----------------------------------------------------------------------------
 */
#define roundew(x)      ( (3 + (WORD) (x) ) & (~3) )
#define truncew(x)      ( (WORD) (x) ) & (~3) )

#define getstk(n)       getmem(n)
#define freestk(b,s)    freemem(b,s)

struct mblock {
      struct mblock *mnext;
      word  mlen;
      };

#define end endaddr                 // inizio memoria disponibile;
                                    // evita il conflitto con le librerie del C

extern struct     mblock memlist;   /* testa della lista di memoria libera */
extern char       *maxaddr;         /* punta all'indirizzo max indirizzabile
extern char       *end              // indirizzo da cui la memoria è disponibile

#define MMAX      65024             // massima dimensione della memoria
```

```
#define MBLK      512             // dimensione blocco per una alloc globale
#define MMIN      8192            /* minima allocazione di Xinu */
#define MDOS      1024            /* salva qualcosa per MS-DOS */

extern char       *getmem();      /* alloca memoria */
extern int        freemem();      /* rilascia memoria */
```

La struttura *mblock* dà la forma di ogni nodo della lista libera. Il campo *mnext* punta sempre al prossimo blocco (o contiene il valore NULL), mentre *mlen* dà la lunghezza del blocco corrente in bytes, includenti le due word citate.

Il file *mem.h* introduce due funzioni in linea, *roundew* e *truncew*. La prima arrotonda un indirizzo di frontiera alla word pari in eccesso, mentre la seconda la arrotonda per difetto. La richiesta di memoria farà in modo che essa sia un multiplo di doubleword. Poiché solo blocchi di due word (4 byte) o più possono essere inseriti nella lista, PC-Xinu rifiuta di allocare o liberare quantità di memoria più piccole. Per essere sicuri che la richiesta specifica un corretto ammontare di memoria, le routine di gestione della memoria usano *roundew* e *truncew* per arrotondare o troncare un numero pari di word, creando un numero di byte multiplo di 4.

8.5.1 Allocare memoria

La procedura *getmem* alloca memoria. Il codice di *getmem* appare di seguito nel file *getmem.c*. Esso usa *roundew* per arrotondare la richiesta di memoria e poi cerca la lista libera per trovare il primo blocco abbastanza largo da soddisfare la richiesta (algoritmo detto First Fit). Poiché la lista dei blocchi liberi usa una concatenazione singola, *getmem* usa due puntatori, *p* e *q*, per la ricerca. Quando *p* punta al blocco della dimensione appropriata, *q* punta al suo predecessore (possibilmente, la testa, *memlist*). Se la dimensione del blocco libero combacia esattamente la quantità richiesta, *getmem* cancella soltanto il blocco dalla lista libera e ritorna il suo indirizzo. Se la dimensione del blocco libero invece eccede, *getmem* partiziona il blocco in due parti; una parte di questo (gli *nbytes* richiesti) vengono restituiti al chiamante tramite il suo indirizzo mentre la rimanente viene unita alla lista libera.

Quando un blocco della lista libera deve essere diviso, la variabile *leftover* punta al pezzo che deve essere lasciato nella lista. Calcolare questo indirizzo è concettualmente semplice: il pezzo in avanzo giace *nbytes* al di là dall'inizio del blocco. Tuttavia, l'aggiunta di *nbytes* al puntatore *p* non produce l'effetto desiderato. Per forzare il C a usare l'aritmetica degli interi invece di quella tra puntatori, *p* viene cambiato in un puntatore a carattere tramite un'operazione di casting (ad es. "(char *) p") e poi il risultato viene cambiato in un puntatore a struttura *mblock* con un altro cast.

```
/* getmem.c - getmem */

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

/* ------------------------------------------------------------------------------
 *  getmem  --  allocaca memoria, restituendo l'indirizzo del blocco richiesto
 * ------------------------------------------------------------------------------
 */
```

```
char *getmem(nbytes)
word nbytes;
{
      int   ps;
      struct mblock *p, *q, *leftover;

      disable(ps);
      if (nbytes == 0) {
            restore(ps);
            return(NULL);
      }
      nbytes = roundew(nbytes);

      for (q = &memlist, p = q->mnext;
            (char *)p != NULL;
            q = p, p = p->mnext)
            if (p->mlen == nbytes) {       /* trovato blocco di uguale dimens. */
                  q->mnext = p->mnext;     /* il blocco p non è più in lista */
                  restore(ps);
                  return((char *) p);      /* p punta al blocco di memoria */
            }
            else if (p->mlen > nbytes) {  /* blocco più grande */
                  leftover = (struct mblock *) ((char *)p + nbytes);
                  /* leftover punta ad una distanza di nbytes
                   * dal blocco puntato da p
                   */
                  q->mnext = leftover;     /* q punterà a leftover */
                  leftover->mnext = p->mnext; /* blocco successivo a leftover */
                  leftover->mlen = p->mlen - nbytes;  /* dim. nuovo blocco */
                  restore(ps);
                  return((char *)p);
            }
      restore(ps);
      return(NULL);
}
```

L’allegato in fondo al capitolo tratta il codice sorgente con allineamento per paragrafo.

8.5.2 Rilasciare memoria

I processi ritornano la memoria precedentemente allocata alla lista dei blocchi liberi quando finiranno di usarla; così potra essere sfruttata ancora. La system call *freemem* ritorna un blocco di memoria inserendolo nell’appropriata locazione della lista libera ed eventualmente unendolo con ogni blocco libero adiacente. Come in *getmem*, i puntatori *p* e *q* scorrono la lista dei blocchi liberi. Dal momento che la lista è tenuta in ordine di indirizzo del blocco, *freemem* arresta la ricerca non appena l’indirizzo del blocco che deve essere restituito giace tra *p* e *q*.

Casi speciali complicano il codice che unisce il blocco ritornato alla lista libera. Il nuovo blocco può giacere adiacentemente a blocchi liberi sopra, sotto o entrambi. (Fallire ciò frammenterebbe la lista libera in piccoli pezzi.)

La trappola può essere evitata ricordando che il nuovo blocco può essere adiacente a blocchi liberi in entrambi i lati. Come mostrato nel file *freemem.c*, il codice controlla sempre se il nuovo blocco è adiacente a quello seguente; ciò avviene anche se è stato appena unito con uno precedente.

```
/* freemem.c - freemem */

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

/* ------------------------------------------------------------------------------
 * freemem  --  libera un blocco di memoria, ritornandolo alla lista libera
 * ------------------------------------------------------------------------------
 */

SYSCALL freemem(block, size)
char *block;
word size;
{
      int   ps;
      struct mblock *p, *q;
      char  *top;

      size = roundew(size);         /* dimensione multipla di 4 */
      block = (char*) truncew((word)block);     // punta all'inizio del blocco
      if (size==0 || block > maxaddr || (maxaddr-block)<size || block < end)
            return(SYSERR);         /* condizioni anomale */
      disable(ps);
      (char *)q = NULL;
      for (p=memlist.mnext;
           (char *)p != NULL && (char *)p < block;
            q = p, p = p->mnext)    /* trova la locazione ordinata */
                        ;           /* in base all'indirizzo crescente */
      if ((char *)q != NULL && (top=(char *)q+q->mlen) > block
          || (char *)p != NULL && (block+size) > (char *)p) {
                  restore(ps);            /* condizioni anomale */
                  return(SYSERR);
      }
      if ((char *)q != NULL && top == block)
            q->mlen += size;        /* viene fuso con il blocco precedente */
      else {                        /* nuovo blocco in testa alla lista */
            ((struct mblock *)block)->mlen = size;    /* dimensione blocco */
            ((struct mblock *)block)->mnext = p;      /* puntatore al prossimo */
            if ((char *) q != NULL)
                  q→mnext = (struct mblock *) block;
            else
                  memlist.mnext = (struct mblock *)block;
            (char *)q = block;
      }
      /* nota che q è diverso da NULL qui */
      if ((char *)p != NULL && ((char *)q + q->mlen) == (char *)p) {
            q->mlen += p->mlen;           /* blocco fuso col successivo */
            q->mnext = p->mnext;          /* punta al next di p */
      }
      restore(ps);
      return(OK);
}
```

Allegato

Il codice sorgente di Xinu usa puntatori a paragrafi a 16 bit per poter indirizzare in tutto 1MB di memoria. I puntatori li chiameremo numeri di paragrafo. Ricordiamo che la struttura mblock usa il campo mnext come intero corto senza segno diversamente da quanto visto in precedenza. Per puntare ai blocchi della lista libera si usa concettualmente una coppia (puntatore far, puntatore come n° di paragrafo); per. es. qp e q, pp e p, leftp e left, etc. Nel codice si vede che quando viene aggiornato un elemento della coppia, viene poi aggiornato anche l'altro - p. es. le 4 righe seguenti il commento /* si deve suddividere il blocco */, dove, in particolare, la macro memp(x) trasforma il n° di paragrafo x nell'indirizzo far del paragrafo x. Di seguito mostriamo il codice di *getmem.c* tratto da PC-Xinu.

```
/* getmem.c - getmem & normal */

#include <conf.h>
#include <kernel.h>
#include <mem.h>
#include <dos.h>

/*------------------------------------------------------------------------------
 *  getmem  --  alloca memoria heap, ritornando l'indirizzo a intero del blocco
 *------------------------------------------------------------------------------
 */
char *getmem(nbytes)
word nbytes;
{
      int   ps;
      char  *pp, *qp, *leftp;       /* pp, qp e leftp sono puntatori far */
      para  p, q, left;             /*  para  (dati  a  16  bit)  ovvero  numeri  di
puntatori a paragrafi */
      word  alloc, nalloc;          /* interi senza segno */

      alloc = roundp(nbytes) >> 4;  /*  arrotonda  nbyte  al  prossimo  paragrafo  e
divide per 16 per ottenere il n° di paragrafo */
      if ( alloc == 0 )
            return(NULLPTR);
      disable(ps);

      qp = (char *) &memlist;
      for (;;) {
            p = memnext(qp);              /* paragrafo iniziale del blocco */
            if ( p == 0 ) {
                  restore(ps);
                  return(NULLPTR);
            }
            pp = memp(p);                 /* puntatore al paragrafo p */
            if ( memlen(pp) < alloc ) {   /* blocco di dimensione ... */
                  qp = pp;                /* minore del necessiario */
                  continue;
            }
            nalloc = memlen(pp) - alloc;  /* blocco di dimensione maggiore */
            if ( nalloc == 0 ) {          /* blocco esatto */
                  left = memnext(pp);\    /* punta al prossimo blocco */
                  break;
            }
            /* si deve suddividere il blocco */
            left = p + alloc;             /* n° paragrafo frammentato */
            leftp = memp(left);           /* punta al frammentato */
            memnext(leftp) = memnext(pp); /*    collega    frammento    e    blocco
successivo */
            memlen(leftp) = nalloc;       /*   aggiorna   dimensione   blocco   in
paragrafi */
            break;
```

```
	}
	memnext(qp) = left;
	memlist.mlen -= alloc;		/* dimensione del blocco suddiviso */
	restore(ps);
	return(pp);
}


/* freemem.c - freemem */

#include <conf.h>
#include <kernel.h>
#include <mem.h>
#include <dos.h>

/*------------------------------------------------------------------------------
 *  freemem  --  libera un blocco di memoria, ritornandolo a memlist
 *------------------------------------------------------------------------------
 */
SYSCALL freemem(bp, size)
char *bp;
int size;
{
	int   ps;
	char  *pp, *qp;
	para  b, p, q, top, retsize;

	retsize = roundp(size) >> 4;
	if ( retsize == 0 || bp == NULL || off(bp) != 0 )
		return(SYSERR);
	b = seg(bp);
	if ( b < endaddr || b > maxaddr || (maxaddr-endaddr) < retsize )
		return(SYSERR);
	disable(ps);
	q = 0;
	qp = (char *) &memlist;
	p = memnext(qp);
	while ( p != 0 && p < b ) {
		q = p;
		qp = memp(q);
		p = memnext(qp);
	}
	if ( q != 0 && (top=q+memlen(qp)) > b ) {
		restore(ps);
		return(SYSERR);
	}
	if ( p != 0 && b + retsize > p ) {
		restore(ps);
		return(SYSERR);
		}
	if ( q != 0 && top == b )
		memlen(qp) += retsize;	/* consolidate q & b blocks   */
	else {
		memlen(bp) = retsize;
		memnext(bp) = p;
		memnext(qp) = b;
		q = b;
		qp = bp;
	}
	if ( p != 0 && q+memlen(qp) == p ) {
		pp = memp(p);		/* consolidate q & p blocks   */
		memlen(qp) += memlen(pp);
		memnext(qp) = memnext(pp);
	}
	memlist.mlen += retsize;
	restore(ps);
	return(OK);
```

```
}
```