Progetto di Sistemi Operativi

Docente: Prof. Renzo Davoli
anno accademico: 2015/2016

componenti:

- Simone Preite simone.preite@studio.unibo.it
- Alessio Innocenzi alessio.innocenzi@studio.unibo.it
- Pierluigi Ballandi pierluigi.ballandi@studio.unibo.it
- Michelangelo Monoriti michelangel.monoriti@studio.unibo.it

FASE 0

Proposta per il primo anno, questa fase ha avuto lo scopo di rendere consapevoli gli studenti del funzionamento delle procedure da utilizzare poi in fase 1.
Si è trattato di implementare le macro, che ci sarebbero servite successivamente, nel file clist.h in sostituzione al listx.h del precedente anno che conteneva le macro e le funzioni in line implementate nel kernel linux.
Altra particolarità di quest'anno è stata la scelta della lista, per la quale si è optato per una lista cirolare con sentinella alla coda anziché una lista bilinkata.
Le macro implementate riguardano appunto la gestione delle funzionalità per questa tipologia di lista:
define clist_enqueue(elem, clistp, member)
define clist_empty(clistx)
define clist_foreach(scan, clistp, member, tmp)
define clist_tail(elem, clistx, member)
define clist_head(elem, clistx, member)
define clist_push(elem, clistp, member)
define clist_pop(clistp)
define clist_foreach_all(scan, clistp, member, tmp)
define clist_delete(elem, clistp, member)
define clist_foreach_delete(scan, clistp, member, tmp)
define clist_foreach_add(elem, scan, clistp, member, tmp)

la lista delle macro è stato fornita dal docente con una breve spiegazione del risultato da ottenere per ognuna di esse insieme alla struttura necessaria per il linking e alla container_of (macro necessaria per l'ottenimento dell'indirizzo dell'intera struttura in cui è contenuto il campo necessario per il linking)

```
struct clist {
        struct clist *next;
};
```

è la struttura necessaria per collegare i vari elementi della lista.

FASE I

Questa fase getta le basi necessarie per l'implementazione della seconda parte del progetto, ovvero si occupa della creazione delle funzioni necessarie a manipolare i processi e i semafori.
I processi come i semafori sono strutturati attraverso liste di elementi, la particolarità ed il limite è che non si ha a disposizione la funzione malloc quindi la dinamicità delle liste è troncata dal fatto che si ha necessità di allocare spazio in memoria per tutti i possibili processi già dall'inizio.
Per questo progetto ci siamo preposti il limite di 20 elementi ma ovviamente è facile sostituire questo numero con uno diverso in caso di necessità, resta comunque una limitazione perché bisogna decidere dall'inizio quanta memoria dedicare ai processi il che lascia intendere che potrebbe essere troppa, o peggio, troppo poca.

C'è da precisare che essendo un array una struttura statica e non è adeguata a gestire la lista dei processi liberi in quanto questi potrebbero occuparsi e liberarsi con tempistiche differenti e difficilmente queste tempistiche riescono a coincidere con l'ordine del vettore e quindi abbiamo il bisogno di linkare gli elementi in una lista per poterli gestire più facilmente.

Questa fase è stata strutturata in 2 parti principali costituite di parti più piccole all'interno di esse.

GESTIONE PROCESSI:

Questa parte si compone di tre sottoparti, ognuna delle quali gestisce un aspetto, le prime due si occupano di gestire le funzionalità dei processi e delle liste che li riguardano mentre la terza si occupa delle funzionalità riguardanti gli alberi e quindi i processi figli.

Di seguito le funzioni di ogni sottoparte viste nel dettaglio.

- ALLOCATION DEALLOCATION DEI PROCESSI:

  1. void initPcbs() :
     questa funzione viene chiamata solo una volta all'inizio e serve a linkare (e quindi inizilizzare) la lista dei processi liberi utilizzando il vettore dichiarato in precedenza.
  2. void freePcb(struct pcb_t *p) :
     attraverso la freePcb (libera processo), come lascia intendere, si libera appunto un processo che viene restituito alla lista dei processi liberi, incrementandola di un processo.

  3. struct pcb_t *allocPcb() :
     si occupa di controllare che esistano ancora processi liberi da poter allocare quindi in caso di risposta affermativa ne preleva uno dalla lista dei processi liberi e ne inizializza tutti i campi in modo tale da non avere residui dell'utilizzazione precedente.
     Ne restituisce il puntatore in modo che il processo possa essere utilizzato dal programma che chiama appunto la funzione in questione.

- MAINTENANCE DELLA CODA DEI PROCESSI:

  1. void insertProcQ(struct clist *q, struct pcb_t *p) :
     inserisce il processo puntato da p nella lista puntata in coda da q.

  2. struct pcb_t *headProcQ(struct clist *q) :
     serve a individuare la testa della lista puntata da q e a ritornarne l'indirizzo senza eliminare alcun processo, nel caso in cui la lista risulti vuota ritorna NULL.

  3. struct pcb_t *removeProcQ(struct clist *q) :
     decrementa la lista puntata da q del processo che si trova alla sua testa e lo ritorna al chiamante, torna NULL se la lista è vuota.

  4. struct pcb_t *outProcQ(struct clist *q, struct pcb_t *p) :
     rimuove il particolare processo puntato da p dalla lista puntata da q.

- MAINTENANCE DELL'ALBERO DEI PROCESSI:

  1. int emptyChild(struct pcb_t *p) :
     controlla se il processo in questione ha dei figli o meno attraverso il campo che punta ai processi figli della struttura.

  2. void insertChild(struct pcb_t *parent, struct pcb_t *p) :
     realizza una enqueue del processo p nella lista dei processi figli di parent.
  3. struct pcb_t *removeChild(struct pcb_t *p) :
     decrementa la lista dei processi figli di p attraverso una dequeue.

  4. struct pcb_t *outChild(struct pcb_t *p) :
     toglie il processo puntato da p dalla lista dei processi del suo parent e ritorna il puntatore del processo eliminato,
     se il processo p non ha un parent torna NULL

GESTIONE SEMAFORI:

La gestione della lista dei semafori attivi è controllata da 5 funzioni che ne gestiscono funzionalità e caratteristiche.

- ACTIVE SEMAPHORE LIST:
  1. void initASL() :
     come per la initPcb anche questa funzione viene chiamata una sola volta per inizializzare la lista dei semafori liberi.

  2. int insertBlocked(int *semAdd, struct pcb_t *p) :
     serve ad inserire un processo bloccato nella lista dei processi bloccati puntata dal semaforo semAdd.
     È stata una delle funzioni più complesse perché si occupa di gestire diverse situazioni, che possiamo riassumere in tre casi.
     - cerca il semaforo nella lista aslh e nel momento in cui lo trova inserisce il processo puntato da p nella lista dei processi bloccati in quel semaforo e setta il puntatore al semaforo a semAdd.
     - se il semaforo non c'è lo alloca prendendolo dalla lista semdFree e inserisce il processo in quel semaforo.
     - se il semaforo non c'è e la lista semdFree è vuota ritorna TRUE.

3. struct pcb_t *removeBlocked(int *semAdd) :
   rimuove il primo processo dalla coda dei processi bloccati nel semaforo puntato da semAdd.

4. struct pcb_t *outBlocked(struct pcb_t *p) :
   rimuove dalla coda dei processi bloccati nel semaforo puntato da p->p_cursem il processo p.

5. struct pcb_t *headBlocked(int *semAdd) :
   ritorna il puntatore alla testa dei processi bloccati in semAdd.

Per questa fase del progetto abbiamo deciso che la struttura più consona, in modo da non pregiudicare la chiarezza dello stesso e garantirne la portabilità per la seconda parte, era comporre il programma in questo modo:

clisth.h : file sviluppato in fase0, contiene le macro necessarie per le funzioni descritte in precedenza.
pcb.c : per l'implementazione delle funzioni riguardanti i processi.
Pcb.h : è il file header dove è dichiarata la struttura dei processi e l'intestazione di tutte le funzioni necessarie.

asl.c : contenente l'implementazione delle funzioni riguardanti i semafori.
asl.h : funge allo stesso modo del pcb.h per pcb.c

una particolarità è che tutta la parte dei semafori ha bisogno di conoscere struttura e funzioni riguardanti i processi.

Const.h : è un file d'appoggio nel quale definire alcuni elementi ausiliari allo sviluppo.

P1test.c : file formito dal docente per testare il funzionamento delle funzioni implementate.

È stato implementato anche un makefile per rendere agevole la compilazione del programma e un file README.md che spiega brevemente come eseguirla insieme a suggerimenti utili per l'utilizzazione del programma su piattaforma uARM.