Documentazione e scelte progettuali

Lorenzo Borelli e Salvatore Perri

Settembre 2019

Contents

1 Strutture dati

RIKAYA utilizza liste e alberi di processi, e semafori per gestire l'accesso a risorse condivise

1.1 PCB

```
typedef struct pcb_t {
    /* Campi coda dei processi */
struct list_head p_next;              puntatore al successivo PCB
in lista
struct pcb_t *p_parent;               puntatore al PCB genitore
struct list_head p_child, p_sib;      puntatore alla lista dei figli

state_t p_s;                          stato corrente del processore

/* Campi priorità del processo */
int priority;                         priorità attuale
int original_priority ;               priorità originale

    int *p_semkey;                        Indirizzo dell'intero del
semaforo su cui il processo è bloccato
```

```
    int tutor;                              Campo tutor (TRUE o FALSE)

/* Campi per i tempi del processo*/
cpu_t user_time, kernel_time;          tempo in user e kernel mode
cpu_t wallclock_time ;                 tempo di creazione del processo
cpu_t last_scheduled, start_kernel ;   ultima volta schedulato

/* Handlers */
state_t *sysbk_old, *sysbk_new;        handler Sys / Break
state_t *tlb_old, *tlb_new;            handler TLB
state_t *pgmtp_old, *pgmtp_new;        handler ProgTrap

}
```

Anzitutto le funzioni per inizializzare, allocare e liberare i Process Control Block liberi:

```
void initPcbs(void)
void freePcb(pcb_t *p)
pcb_t *allocPcb(void)
```

Una volta allocato e inizializzato un PCB con le funzioni *allocPcb* e *initPcbs*, si può pensare a come gestire la lista dei processi attivi:

```
void mkEmptyProcQ(struct list_head *head)
int emptyProcQ(struct list_head *head)
void insertProcQ(struct list_head *head, pcb_t *p)
pcb_t *headProcQ(struct list_head *head)
pcb_t *removeProcQ(struct list_head *head)
pcb_t *outProcQ(struct list_head *head, pcb_t *p)
```

Tramite *mkEmptyProcQ* si inizializza la lista, più nello specifico il suo elemento sentinella. *emptyProcQ* controlla se la lista puntata da head sia vuota o meno, restituendo true in caso positivo e false altrimenti. *headProcQ* restituisce il primo elemento della lista se presente, altrimenti ritorna NULL. Per inserire un nuovo processo nella coda viene invocata la *insertProcQ* con input un puntatore alla testa della lista e un puntatore al PCB da inserire, questa operazione viene fatta tenendo conto della priorità di ciascun processo (l'elemento in testa ha priorità più alta). Analogamente per rimuoverlo si usa *outProcQ*, nel caso se ne voglia eliminare uno nello specifico, altrimenti *removeProcQ* se si vuole rimuovere quello in testa. Una volta rimosso dalla lista si libera il PCB con *freePcb* per renderlo disponibile ad un eventuale nuovo processo.

In RIKAYA oltre alla visione da lista, si possono gestire i PCB anche con una visione ad albero, con i tipici attributi figlio, padre, fratello. Ogni genitore contiene un *list_head* (*p_child*) che punta alla lista dei figli. Ogni figlio ha un puntatore al padre (*p_parent*) ed un *list_head* che punta alla lista dei fratelli.

```
int emptyChild(pcb_t *this);
void insertChild(pcb_t *prnt, pcb_t *p);
pcb_t *removeChild(pcb_t *p);
pcb_t *outChild(pcb_t *p);
```

emptyChild fa una banale verifica sul PCB per controllare se abbia dei figli (false) o meno (true). *insertChild* inserisce il PCB puntato da p nella lista dei figli del PCB puntato da prnt. *removeChild* rimuove il primo figlio del PCB puntato da p. Se invece si vuole rimuovere uno specifico PCB dall'albero si usa *outChild* fornendo in input il puntatore al target da eliminare.

1.2 Active Semaphore List

In RIKAYA si fa uso dei semafori per gestire l'accesso alle risorse condivise, ad ognuno è associato un descrittore:

```
typedef struct semd_t {
    struct list_head s_next;    //Il prossimo semaforo nella lista
    int *s_key;                 //Il valore del semaforo
    struct list_head s_procQ;   //La lista di processi attualmente
bloccati sul semaforo
} semd_t
```

I semafori vengono organizzati in una lista di semafori attivi, gestita dalle seguenti funzioni:

```
semd_t* getSemd(int *key);
void initASL();

int insertBlocked(int *key,pcb_t* p);
pcb_t* removeBlocked(int *key);
pcb_t* outBlocked(pcb_t *p);
pcb_t* headBlocked(int *key);
void outChildBlocked(pcb_t *p);
```

getSemd restituisce il puntatore al semaforo il cui valore è puntato da key. Se non esiste restituisce NULL. Per inizializzare la lista dei semafori attivi viene invocata *initASL*. Per effettuare operazioni di modifica sulla suddetta lista si usano:

- *insertBlocked* per inserire un nuovo processo nella coda dei bloccati sul semaforo
- *removeBlocked* rimuove il primo PCB bloccato sul semaforo
- *outBlocked* rimuove il PCB puntato da p dalla coda del semaforo
- *headBlocked* ritorna il primo PCB bloccato sul semaforo
- *outChildBlocked* rimuove il PCB puntato da p dalla coda del semaforo e sblocca tutti i discendenti di p da eventuali semafori

2 Scheduler

Lo scheduler implementato è basato su priorità, è preemptive in quanto concede un determinato timeslice di 3ms ad ogni processo. Ricorre alla tecnica di aging per evitare starvation dei processi a più bassa priorità: infatti ad ogni scadere del timeslice viene incrementata la priorità di tutti i processi presenti della **Ready Queue**. Lo scadere del timer genera un interrupt che viene catturato dalla funzione *interrupt_handler*, in cui viene anche fatto l'aging.
La funzione principale dello scheduler è *schedule*.

```
void schedule(state_t *old);
```

A seconda della presenza di processi ready o di processi bloccati prende scelte differenti: se ci sono ancora processi in coda ready, sceglie il prossimo e salva il precedente e, prima di mandarlo in esecuzione, setta il timer e prepara valori per il calcolo dei tempi dei processi; altrimenti, se ci sono solo processi bloccati va in wait aspettando un interrupt(in questo caso il registro di status corrente viene settato nel modo appropriato).

```
void insertProcqReady(state_t *old, pcb_t *proc);
```

La funzione *insertProcqReady* viene chiamata da *schedule* per gestire l'inserimento in coda di un processo. Questa funzione tiene conto della possibilità che il processo da inserire sia quello corrente (running_process) o uno nuovo, per esempio appena creato dalla system call *CreateProcess*. La funzione che gestisce il timer è *set_timer*

```
void set_timer()
```

Il timer viene calcolato basandosi sul time of day, grazie al quale si decide se il prossimo timer da settare è il timeslice o lo pseudoclock, a seconda di quale scadenza è più vicina.

```
void update_usertime(pcb_t *userproc)
void update_kerneltime(pcb_t *kernelproc)
```

Queste due funzioni si occupano di calcolare il tempo utente e kernel di un processo. Lo user time viene calcolato utilizzando due variabili dei pcb: start_kernel e last_scheduled: la prima registra il tempo di inizio di esecuzione del codice kernel, e infatti viene inizializzata sia nell'handler delle system call sia in quello degli interrupt, mentre la seconda registra il momento in cui il processo viene mandato in esecuzione sulla cpu, e perciò viene inizializzato prima di ogni invocazione della funzione *LDST*. Lo user time è quindi la differenza tra start_kernel e last_scheduled. Il kernel time invece fa uso del tempo corrente (cioè quello durante il quale la funzione *update_kerneltime* viene invocata, e start_kernel, calcolandone la differenza.

3 Interrupt

Riprendendo quanto specificato nel manuale uMPS:

```
Int. Line  Device Class
    0          Inter-processor interrupts
    1          Processor Local Timer
    2          Bus (Interval Timer)
    3          Disk Devices
    4          Tape Devices
    5          Network (Ethernet) Devices
    6          Printer Devices
    7          Terminal Devices

  Word    Physical Address          Field Name
    0       0x1000.003C         Interrupt Line 3 Interrupting Devices
Bit Map
    1       0x1000.0040         Interrupt Line 4 Interrupting Devices
Bit Map
    2       0x1000.0044         Interrupt Line 5 Interrupting Devices
Bit Map
    3       0x1000.0048         Interrupt Line 6 Interrupting Devices
Bit Map
    4       0x1000.004C         Interrupt Line 7 Interrupting Devices
Bit Map
```

Gli interrupt sono gestiti da una routine *interrupt_handler* che innanzitutto distingue tra interrupt TIMESLICE e PSEUDOCLOCK.

```
void interrupt_handler();
```

Nel primo caso come si è già detto viene incrementata la priorità dei processi ready, nel secondo caso vengono risvegliati con una Verhogen eventuali processi che aspettavano il tick del clock di sistema (SYS6); in entrambi i casi, viene calcolato il kernel time e ridato il controllo allo scheduler.
Se invece l'interrupt è stato generato da un device, facendo riferimento alle specifiche dell'architettura di uMPS si risale al numero della linea di interrupt e al numero del device, dapprima identificando la linea tramite il registro cause, poi utilizzando la linea per ottenere la bitmap per il numero di device. Tenendo conto che le prime tre linee (0-1-2) non vengono considerate perchè vengono già gestite prima, la numerazione effettiva delle linee device parte da 0, corrispondente alla linea 3, per poi continuare fino a 4, cioè la linea 7. È importante distinguere tra device "normali" e terminali, in quanto quest'ultimi sono una coppia di dispositivi formata da un device di ricezione ed uno di trasmissione. Se l'interrupt è stato generato da un terminale verifichiamo prima se si tratta di una trasmissione o di una ricezione, dopodichè risvegliamo il processo bloccato sul terminale corrispondente ed infine inviamo un ACK allo stesso. Se invece

non si tratta di un terminale le operazioni sono pressochè uguali, si risveglia il processo sul device corrispondente e poi si invia un ACK.

4 System call

```
void SYS_handler ();
```

Nel sistema RIKAYA vengono implementate 10 System Call, gestite in kernel mode dal sistema. L'handler predisposto controlla il contenuto del registro cause per fare una distinzione tra System Call (EXC_SYSCALL) e Breakpoint (EXC_BREAKPOINT). Nel primo caso ci si accerta che attualmente sia acceso il bit della kernel mode, nel qual caso si determina quale sia la System Call richiesta, il cui numero è contenuto del registro A0 della CPU, mentre i parametri si trovano nei registri A1, A2, A3. Nel qual caso venga richiesta una System Call da eseguire in user mode verrà sollevata un'eccezione da istruzione riservata (EXC_RESERVEDINSTR) che verrà gestita dal ProgramTrap handler.

```
int specifiedHandler (int type, state_t* s);
```

Funzione ausiliaria per trovare un eventuale handler di livello superiore, se presente lo stato corrente viene salvato nell'area old e verrà caricato lo stato new corrispondente (sysbk_new, pgmtp_new, tlb_new).

```
void programtrap_handler();
```

Routine che passa la gestione all'handler specificato per il processo, se presente. Nel caso non venga rilevato il processo viene terminato.

```
void tlb_handler();
```

Come per il ProgTrap verifica la presenza di un handler e nel qual caso non lo trovi termina semplicemente il processo.

Poi ci sono le system call

```
void SYSCALL(GETCPUTIME, unsigned int *user, unsigned int *kernel,
unsigned int *wallclock);
int SYSCALL(CREATEPROCESS, state_t *statep, int priority, void ** cpid);
int SYSCALL(TERMINATEPROCESS, void ** pid, 0, 0);
void SYSCALL(VERHOGEN, int *semaddr, 0, 0);
void SYSCALL(PASSEREN, int *semaddr, 0, 0);
void SYSCALL(WAITCLOCK, 0, 0, 0);
int SYSCALL(IOCOMMAND, unsigned int command, unsigned int *register,
0);
void SYSCALL(SETTUTOR, 0, 0, 0);
int SYSCALL(SPECPASSUP, int type, state_t *old, state_t *new);
void SYSCALL(GETPID, void ** pid, void ** ppid, 0)
```

- Restituisce il tempo di esecuzione del processo chiamante, che sia user, kernel o wallclock (tempo trascorso dalla prima attivazione)

- Crea un nuovo processo come figlio del processo chiamante(se esiste), settando lo stato iniziale in base al parametro statep. Ritorna 0 in caso di successo, -1 altrimenti. Se cpid è passato come parametro conterrà il pid del nuovo processo. Inizializza la priorità, kernel e user time, e lo inserisce nella coda ready.

- Termina il processo con il pid passato come parametro, se figlio del processo corrente. Se non viene specificato alcun pid, viene terminato il processo corrente. Se il processo terminato ha figli, questi diventano figli di un tutore, specificato dal chiamante, o dalla radice se non esiste. Se il processo è bloccato sul semaforo, viene sbloccato, altrimenti viene tolto dalla coda ready (se non era il corrente). Viene infine riportato nei processi liberi. In caso di successo ritorna 0, -1 altrimenti.

- Operazione di V (incremento, rilascio) sul semaforo il cui valore è puntato da semaddr, e conseguente sblocco di un processo, se il valore del semaforo è minore o uguale a 0.

- Operazione di P (decremento, richiesta) sul semaforo il cui valore è puntato da semaddr. Se il processo corrente è stato quello ad essere bloccato, viene sospeso, resettandone la priorità e salvandone lo stato.

- Sospende il processo corrente fino al prossimo tick di sistema. Realizzata mediante un'operazione P sul semaforo dello PSEUDOCLOCK.

- Operazione bloccante per il chiamante che viene sospeso fino all'interrupt che segnala il completamento del comando. Il parametro command viene copiato nel campo comando del registro puntato da register. Il valore di ritorno è il contenuto del registro status del dispositivo.

- Setta il campo tutor del processo chiamante a TRUE, così che agisca da tutore per i processi discendenti che diverranno orfani.

- Con questa SysCall si specifica quale handler di livello superiore vada attivato in caso di: SYSCALLBREAKPOINT (type$\bar{0}$), TLB (type$\bar{1}$), PROGRAM_TRAP (type$\bar{2}$)

- assegna il l'identificativo del processo corrente a *pid e l'identificativo del processo genitore a *ppid.

Le System Call con numero superiore a 10 vengono intoltrate ad un handler di livello superiore se specificato, altrimenti causano la terminazione del processo.