Relazione di Kaya 2010
a cura di
Barbara Iadarola e Vincenzo Ferrari

Il progetto di Laboratorio di Sistemi Operativi è stato sviluppato, scritto e testato su macchine **Intel x86** con sistema **Ubuntu Linux** del **Laboratorio Ercolani**.

Sono stati realizzati entrambi i livelli: phase1 e phase2.

PHASE 1

La Phase1 di Kaya2010 è stata realizzata gestendo i process control blocks (pcb) e i descrittori dei semafori (semd) tramite i seguenti moduli :

pcb.c – asl.c

PCB.C

Modulo per l'inizializzazione, l'allocazione, la deallocazione e la gestione delle code e degli alberi di pcb.

° Liste di PCB °

Con la funzione initPcbs(void) viene creata la lista di pcb non utilizzati, concatenando la testa "pcbfree_h" della medesima con le liste dei singoli pcb tramite la funzione freePcb(pcb_t *p).
I pcb disponibili sono presenti nel vettore statico "pcbtable".
L'implementazione prevede che la lista di pcb liberi sia composta da pcb concatenati tra loro tramite liste circolari doppiamente collegate.
Per prelevare e inizializzare un pcb dalla lista libera è necessario utilizzare la funzione allocPcb(void) la quale fa uso della macro "container_of()" per prelevare l'indirizzo della struttura pcb_t del pcb corrente.
Non è infatti possibile disporre direttamente dei vari campi di una struttura partendo da un suo campo interno.

° Code di PCB °

Utilizzando la funzione mkEmptyProcQ(struct list_head *emptylist) è possibile inizializzare una coda vuota di pcb, mentre per controllare la correttezza della stessa si può usare la funzione emptyProcQ(struct list_head *head).
Per inserire dei pcb all'interno di una coda si può far riferimento a insertProcQ(struct list_head *head, pcb_t *p); al contrario, la funzione removeProcQ(struct list_head *head) ne rimuove il primo da una coda.
Se invece si volesse eliminare un particolare pcb, è necessario usare l'alternativa outProcQ(struct list_head *head, pcb_t *p).
Infine, per avere a disposizione la testa di una coda, si può far uso della funzione headProcQ(struct list_head *head).

° Alberi di PCB °

Anche gli alberi sono gestiti tramite liste circolari doppiamente collegate.
Per controllare che un pcb non abbia alcun figlio è necessario utilizzare la funzione emptyChild(pcb_t *child), mentre per associare un figlio ad un altro pcb genitore occorre richiamare la insertChild(pcb_t *parent, pcb_t *child).
Nel momento in cui si desidera rendere orfano un pcb è possibile far ricorso a due funzioni distinte: removeChild(pcb_t *parent), la quale rimuove il primo dalla lista di figli di un pcb, oppure outChild(pcb_t *child) la quale rende orfano un particolare pcb figlio.

ASL.C

Modulo per la gestione dei descrittori di semafori. Contiene la creazione, l'inizializzazione e la gestione delle liste di descrittori di semafori attivi e inattivi.
Entrambe le liste sono circolari doppiamente collegate e hanno funzioni diverse.

La lista di descrittori attivi, la cui testa è identificata da "semd_h", è ordinata in base all'indirizzo dei semd presenti e ogni descrittore può avere nessuno, uno o più processi associati ad esso, organizzati in code.
E' possibile accedere a tali code tramite il campo s_procQ che permette la ricerca di specifici pcb all'interno delle stesse.
Per scorrere i descrittori è invece necessario utilizzare il campo s_next, che permette la ricerca di un semd dato uno specifico indirizzo.

La lista di descrittori inattivi, la cui testa è identificata da "semdfree_h", non è ordinata, ma contiene i semd non ancora inizializzati, creati tramite un'apposita funzione (InitSemd(void)) che li recupera da un vettore di descrittori disponibili memorizzati staticamente.

L'inizializzazione dei descrittori avviene nella fase di inserimento dei semd nella lista ASL (insertBlocked (S32 *semAdd, pcb_t *p)), la quale affronta il problema di collocare un particolare pcb nell'apposita coda del descrittore, assicurandosi che questo sia presente nella lista.
La ricerca avviene grazie a un ciclo, il quale scorre descrittore per descrittore controllando ogni indirizzo, ordinati in modo crescente.
La rimozione può avvenire in due diverse modalità: l'eliminazione del primo pcb dalla coda associata al descrittore del semaforo passato (*removeBlocked(S32 *semAdd)), in cui si utilizza lo stesso metodo di identificazione dei semd visto in precedenza, e l'eliminazione di un particolare pcb andando a recuperare il campo riguardante l'indirizzo del suo semaforo e comparandolo con quelli scorsi nel ciclo.

Infine, *headBlocked(S32 *semAdd) preleva dalla coda associata al semaforo il primo pcb, scorrendo nuovamente la ASL per trovare il descrittore corrispondente.
Grazie alla container_of() è possibile risalire all'indirizzo della struttura del pcb cercato.

PHASE2

La Phase2 di Kaya2010 è stata realizzata utilizzando le strutture dati della fase precedente e implementando la capacità di gestire le eccezioni PgmTrap, SYS/Bp, TLB, Ints che possono sollevarsi.
Oltre a questo, uno scheduler round robin sceglierà l'ordine di esecuzione dei processi, garantendo che ognuno di essi abbia la possibilità di eseguire.
Sono stati creati 4 appositi moduli per gestire il tutto:

initial.c - interrupts.c - scheduler.c – exceptions.c

INITIAL.C

Modulo per l'inizializzazione del nucleo.

E' il punto di entrata per Kaya, che include varie fasi di inizializzazione.
La popolazione delle 4 nuove aree nella ROM Reserved Frame (gestita dalla funzione populate (memaddr area, memaddr handler), una per ogni tipo di eccezione) consiste nel salvare in una nuova variabile (puntatore ad uno state_t) l'area passata per parametro, salvare lo stato corrente del processore nella nuova area (grazie a un'apposita funzione STST presente nella libreria di uMPS), impostare il PC con l'indirizzo della funzione che gestirà eccezioni di quel tipo (passato anch'esso per parametro) e il registro SP a RAMTOP.
Lo status register verrà inoltre impostato per mascherare tutti gli interrupt, essere in kernel-mode ed avere la memoria virtuale spenta, impostando o non impostando gli status fornitici (precedenti o correnti) nel file delle costanti di Kaya.
La populate accoglie dunque come parametri gli indirizzi delle nuove aree di ogni eccezione e gli indirizzi dei corrispondenti gestori (nel nostro caso: sysBpHandler(), pgmTrapHandler(), tlbHandler() presenti nel modulo "exceptions.c" e intHandler() presente nel modulo "interrupts.c").

Vengono poi chiamate le funzioni della prima fase: initPcbs() e initSemd() che inizializzano le strutture dati dei pcb e dei semafori.

Si inizializzano anche tutte le variabili mantenute nel nucleo: la readyQueue richiama la mkEmptyProcQ(struct list_head *emptylist) che inizializza la coda dei processi pronti a essere eseguiti, processCount e softBlockCount (rispettivamente contatori dei processi presenti nel sistema e dei processi bloccati in attesa che sia completato un I/O) posti a zero, currentProcess (il pcb in esecuzione) non punta ancora a nessun process control block e pidCount (utilizzato per dare ad ogni pcb un pid univoco tramite il semplice incremento di questa variabile per ogni nuovo pcb creato) è inizialmente zero.
Lo stesso vale per timerTick, altro contatore necessario per il calcolo del tempo trascorso nella gestione dell'Interval Timer.

I semafori dei device sono stati implementati come un'unica struttura chiamata "sem" composta da 6 vettori di interi, uno per ogni device.
Essi a loro volta hanno 8 indici, uno per ogni linea di interrupt.
I device presenti sono: disk, tape, network, printer, terminalR e terminalT (poiché il terminale è visto come due device separati: uno per ricevere e uno per trasmettere).
L'intera struttura è inizializzata a zero siccome ancora nessun processo è bloccato su uno di questi semafori.

E' presente un ulteriore semaforo, lo pseudo-clock timer, sempre inizializzato a zero, associato agli pseudo-clock tick trascorsi.

Una volta inizializzate variabili globali, aree e strutture, viene allocato il primo pcb, ovvero init. Questo viene messo nella Ready Queue, e accedendo al suo status register si attivano gli interrupt, si spegne la memoria virtuale e si attiva il kernel-mode, con le stesse modalità viste in precedenza.
Il registro SP viene inizializzato a RAMTOP - FRAMESIZE e in PC è memorizzato l'indirizzo della funzione test() che sarà utilizzata per debuggare il nucleo.
Si incrementa quindi pidCount, assegnando l'identificatore 1 a init, e processCount; si avvia lo startTimerTick, col tempo corrente del sistema avuto tramite la GET_TODLOW, che sarà utilizzato nella gestione dello pseudo-clock timer e si fa partire lo scheduler().
Chiamato questo, il controllo non tornerà più al main() di initial.c.

SCHEDULER.C

Modulo per la gestione dell'avvicendamento dei processi sul processore.
Identifica stati di deadlock.

Lo scheduler() distingue due casi: la presenza di un processo sulla CPU o meno.
Nel primo caso, si imposta il campo p_cpu_time del processo corrente come nuovo tempo di partenza del processo sulla CPU e si aggiorna il tempo trascorso dall'ultimo pseudo-clock tick.
Con la macro SET_IT si imposta poi l'Interval Timer con il tempo minore tra lo pseudo-clock tick rimanente e il time slice.
Solo a questo punto viene caricato lo stato del processo corrente.
Nel secondo caso, si controlla se la coda dei processi ready è vuota.
Se così è, e non ci sono processi nel sistema, non si deve far altro che arrestare con HALT().
Se invece ci sono processi nel sistema ma nessuno di questi è bloccato allora qualcosa è andato storto e si cade in uno stato di deadlock (PANIC()).
Altrimenti, se ci sono processi nel sistema e qualcuno di essi è bloccato, si entra in uno stato di attesa in cui gli interrupt sono attivati e non mascherati, così da poter avere la possibilità di sbloccare qualche processo in coda su di un semaforo che potrebbe essere “Vato” da lì a poco.

Nel momento in cui sono presenti dei processi nella coda ready, si prende il primo, si calcola quanto tempo è trascorso dall'ultimo pseudo-clock tick, si imposta il campo p_cpu_time a zero come tempo di partenza del processo sulla CPU e si imposta poi l'Interval Timer con il tempo minore tra lo pseudo-clock tick rimanente e il time slice.
Si carica quindi lo stato del processo corrente.

EXCEPTIONS.C

Modulo per la gestione delle eccezioni SYS/BP, PgmTrap e TLB.

SYS/BP Handler
Il primo gestore riguarda eccezioni SYSCALL e Breakpoint.
La funzione saveCurrentState (state_t *current, state_t *new) provvede a salvare lo stato corrente in nuovo stato passato per parametro.
Nel momento in cui viene sollevata un'eccezione SYS/BP è necessario caricare lo stato della vecchia area SYS/BP nello stato del processo corrente.
Viene incrementato di 4 il registro PC poichè esso punta all'indirizzo dell'istruzione che ha causato l'eccezione e restituire il controllo al processo che ha richiesto la SYSCALL produrrebbe un loop infinito di SYSCALL.
A questo punto si deve verificare il tipo di eccezione avvenuta (tramite la macro CAUSE_EXCCODE_GET che la recupera dal registro "cause") e la modalità in cui è avvenuta (recuperata dallo status della vecchia area della SysBp).

Se è stata chiamata una SYSCALL si controlla dunque di essere in kernel mode, altrimenti deve essere generata una PgmTrap, impostando a istruzione riservata il registro "cause".
Se non è stata eseguita una specSYSvect per l'eccezione allora il processo corrente deve essere terminato, altrimenti viene salvata la vecchia area della SysBp all'interno del processo corrente e caricata la nuova.
Gestita in kernel mode, vengono salvati i parametri delle SYSCALL (contenuti nei registri da a1 a a3), mentre da a0 si estrapola la SYSCALL stessa.
Anche in questo caso, se in a0 non c'è una delle 13 SYSCALL privilegiate si controlla se non sia già stata eseguita una specSYSvect e in tal caso si termina il processo corrente.
Ogni SYSCALL una volta terminata tornerà allo scheduler().
Se è stata chiamata una Breakpoint si controlla se non sia già stata eseguita una specSYSvect e anche in questo caso si termina il processo corrente.
Nel caso in cui il gestore di eccezioni SYS/BP non riconosca la causa avviene un PANIC().

Le 13 SYSCALL privilegiate sono spiegate di seguito.
La prima (createProcess(state_t *statep)) crea i processi da inserire nella Ready Queue utilizzando una funzione della fase precedente (allocPcb()) e caricando lo stato del processore in quello del processo, aggiornando il contatore dei processi presenti nel sistema e assegnandoli un pid univoco. Esso diventa il nuovo figlio del processo chiamante.

La seconda (terminateProcess(int pid)) termina un processo il cui pid è passato per parametro, insieme a tutta la sua progenie.
Il processo da terminare è ricercato nell'intero sistema: nella readyQueue, sul semaforo dello pseudo clock, sui semafori dei device, se è il processo corrente e infine su un semaforo esterno.
La funzione è ricorsiva e uccide la progenie passando -1 come pid alla nuova chiamata di terminateProcess.

La terza (verhogen(int *semaddr)) semplicemente effettua una V su un semaforo passato per parametro, rimuovendo il processo bloccato su questo e inserendolo nella Ready Queue.
E' il contrario per la quarta SYSCALL, passeren(int *semaddr), che effettua una P sul semaforo passato per parametro: nel caso il valore diventi negativo inserisce il processo corrente in coda al semaforo specificato.
Il processo corrente non punta più a nessun pcb e sarà lo scheduler ad assegnargliene uno nuovo.

getPid() e getPPid() sono utilizzate per accedere al campo PID del processo corrente e del suo genitore.

getCPUTime() indica quanto tempo un processo è stato sulla CPU, accedendo al campo p_cpu_time e aggiornandolo con la differenza tra il tempo attuale e quello di partenza sulla CPU (mantenuto dalla variabile processTOD).
Questa si fa ripartire quindi con il tempo attuale per calcolare daccapo il tempo speso sul processore da parte del processo.

waitClock() effettua la passerenIO sul semaforo dello pseudo-clock.
Questa P è diversa dalla SYS4 siccome viene incrementato il softBlockCount.

waitIO(int intlNo, int dnum, int waitForTermRead) effettua una passerenIO sul semaforo del device e la cui linea di interrupt è passata per parametro.
Questa funzione ritorna lo status word del device specificato.
E' stata utilizzata una matrice di 6x8 (numero dei device per numero di linee di interrupt) per mantenere una tabella di tutti gli status word.
Il parametro waitForTermRead è utile per capire se la linea di terminale da considerare è quella di

ricezione o di trasmissione.
specTLBvect(state_t *oldp, state_t *newp), specPGMvect(state_t *oldp, state_t *newp) e specSYSvect(state_t *oldp, state_t *newp) servono a salvare il vecchio stato del processore al momento dell'eccezione e il nuovo che dovrà essere caricato nello stato del processo corrente.
Ogni pcb ha difatti, per ogni eccezione, un campo per il vecchio stato del processore e uno per il nuovo.
Siccome è possibile richiamare ognuna di queste funzioni una sola volta nella vita di un processo, è stato aggiunto un campo addizionale al pcb che contiene un vettore i cui indici corrispondono al tipo di eccezione sollevata per processo (nell'ordine: TLB, PgmTrap e SYS/BP).
Nel caso in cui uno di questi sia già arrivato a 1, allora si richiama la terminateProcess (int pid).

TLB Handler
Il secondo gestore riguarda eccezioni TLB e si occupa di salvare lo stato della vecchia area del TLB nello stato del processo corrente.
Se non è stata eseguita una specTLBvect per quell'eccezione allora il processo corrente deve essere terminato, altrimenti viene salvata la vecchia area del TLB all'interno del processo corrente e caricata la nuova.

Program Trap Handler
Il terzo gestore, riguardante le eccezioni PgmTrap, esegue le medesime istruzioni su nuova e vecchia area del PgmTrap.

INTERRUPTS.C

Modulo per il riconoscimento e la gestione degli interrupt.

Varie cause possono scatenare un interrupt.
Per riconoscerle è necessario accedere al registro "cause".
Con un'apposita macro, la CAUSE_IP_GET ,è possibile riconoscere all'interno di questo registro quale linea di interrupt tra le 8 possibili (considerando che il terminale vale per due) è coinvolta.

Nel caso in cui la responsabile sia la seconda linea (la linea 0 e la linea 1 vengono ignorate poichè Kaya non genera interrupt software) allora è necessario distinguere tra due eventi: il termine di uno pseudo-clock tick oppure la fine del time slice di un processo.
Se l'interrupt è stato causato per il primo motivo, bisogna assicurarsi che la waitClock() non abbia decrementato più volte il valore del semaforo dello pseudo-clock.
In tal caso, è necessario sbloccare tutti i processi bloccati su quel semaforo; se compiendo una V sul semaforo non viene sbloccato nessun processo allora si decrementa lo pseudo-clock.
Altrimenti si compie una V sullo pseudo-clock, che deve aumentare automaticamente di 1 ogni 100 millisecondi.
Si riavvia poi il tempo per il calcolo dello pseudo-clock tick.
Se l'interrupt invece è stato causato per il secondo motivo, allora si reinserisce il processo corrente tra i processi in coda ready e dopo aver aggiornato i contatori del tempo trascorso si richiama lo scheduler().

Nel caso in cui le altre linee siano responsabili, si compie la medesima azione per tutte, con qualche accorgimento per l'ultima (suddivisa in due casi).
Avendo a disposizione nel file delle costanti l'indirizzo iniziale delle bitmap contenenti gli interrupt pendenti per ogni linea di interrupt, si accede a queste calcolando la dimensione di una parola moltiplicata per il numero di linea che ha causato l'interrupt e sommandogli l'indirizzo iniziale.
Per trovare poi quali bit indicano i device con interrupt pendenti per quella linea si utilizza la recognizeDev(int bitMapDevice) che riconosce il settaggio dei bit compiendo degli OR su questi e

restituendo il numero che rappresenta l'indice del device.
Si cerca quindi l'indirizzo di questo per recuperare i campi status e command, che serviranno per impostare il registro v0 del processo che verrà sbloccato e mandare l'ACK di avvenuto riconoscimento dell'interrupt pendente.
Nel caso del terminale, si dovrà riconoscere in base allo status del device se il carattere sarà trasmesso o ricevuto.
Si compie poi una V sul semaforo associato al device che ha causato l'interrupt.
Terminate queste operazioni si ritorna sempre allo scheduler().

NOTE

Per evitare dei “warning” per l'utilizzo della macro container_of() presente nel file "listx.h", è stato scelto di non inserire la flag "-pedantic" nel comando di compilazione.

Il progetto è suddiviso nelle seguenti cartelle :

- ./. : contiene i file di creazione configure, Makefile.am, ChangeLog, COPYING, Doxyfile, README, INSTALL, NEWS
- ./include : contiene i file header "listx.h", "const.h", "types10.h", "uMPStypes.h", "base.h"
- ./phase1/e : contiene i file di definizione "asl.e", "pcb.e"
- ./phase1/src : contiene i file sorgenti "asl.c", "pcb.c", "p1test.0.1.2.c"
- ./phase2/e : contiene i file di definizione “initial.e”, “scheduler.e”, “exceptions.e”, “interrupts.e”
- ./phase2/src : contiene i file sorgenti “initial.c”, “scheduler.c”, “exceptions.c”, “interrupts.c”, “p2test.0.1.c”
- ./doc : contiene i file di documentazione "phase1doc"