RELAZIONE DI AMIKAYA2011

Gianluca Iselli
Giulio Biagini
Simone Rondelli

La Phase 1 di AMIKaya2011 é stata realizzata gestendo i moduli per l'inizializzazione, l'allocazione, la de allocazione e la gestione delle code e degli alberi dei TCB e dei MSG.

La Phase2 di AMIKaya2011 è stata realizzata utilizzando le strutture dati della fase precedente e implementando la gestione di eccezioni che possono sollevarsi, un boot, il quale si occuperá di inizializzare il sistema, un ssi che si occuperá di soddisfare le richieste dei thread ed uno scheduler round robin, il quale si occuperá di scegliere l'ordine di esecuzione dei processi. É stato inoltre aggiunto un file con le nostre funzioni.

Il progetto é stato cosí organizzato:

- msg.c — file per la gestione dei messaggi
- tcb.c — file per la gestione dei thread
- boot.c — file per l'inizializzazione del sistema
- scheduler.c — file per la gestione dell'avvicendamento nell'uso della cpu da parte dei processi
- ssi.c — file per la gestione dei servizi richiesti dai thread
- exception.c — file per la gestione delle system call
- interrupt.c — file per la gestione degli interrupt
- prgTrap.c — file per la gestione delle prgTrap
- tlbTrap.c — file per la gestione delle tlbTrap
- utils.c — file con le nostre funzioni

msg.c

Qui sono presenti le funzioni utilizzate per gestire le code di messaggi.
Per maggiori informazioni sull'implementazione relativa alla gestione dei messaggi guardare il file DOC_MSG_TCB.txt.

tcb.c

Qui sono presenti le funzioni utilizzate per gestire i Tread Control Block.
Per maggiori informazioni sull'implementazione relativa alla gestione dei thread guardare il file DOC_MSG_TCB.txt.

boot.c

É il punto di entrata per AMIKaya2011. Questo modulo si occupa dell'inizializzazione del sistema. Qui vengono inizializzate le 4 aree nel ROM Reserved Frame. L'inizializzazione avviene impostando il PC e il registro t9 con l'address della funzione nel nucleo che deve gestire le eccezioni di questo tipo, impostando il registro SP al RAMTOP e impostando opportunamente il registro di stato in modo che gli interrupt vengano mascherati, venga disattivata la memoria virtuale e si passi in kernel mode.
Vengono poi inizializzate le strutture dati di Phase 1, in particolare vengono chiamate le funzioni initTcbs() ed initMsg(), e le variabili del nucleo, come ad esempio il currentThread.

Infine vengono allocati ed inizializzati 3 thread: il thread init per tutti i processi orfani, il thread ssi e un thread test. I thread ssi e test vengono inizializzati con la memoria virtuale off, gli interrupt abilitati, e in modalitá kernel.
Come ultima operazione viene chiamato lo scheduler.

scheduler.c

Lo Scheduler implementato mediante l'algoritmo round robin che esegue i processi nell'ordine d'arrivo ed applica prelazione al processo in esecuzione, ponendolo alla fine della coda dei processi in attesa, qualora l'esecuzione duri più del quanto di tempo stabilito, e facendo proseguire l'esecuzione al successivo processo in attesa.
Prima di tutto vengono inizializzate quattro code: la readyQueue, cioé la coda dei processi in attesa di essere eseguiti, la waitQueue, la coda dei thread in attesa di essere sbloccati e la waitForPseudoClockQueue, cioé la coda dei thread in attesa dello pseudo clock.
Si é inoltre scelto di implementare 5 ulteriori linee di wait da 8 code ognuna, una per ogni linea di device. In queste code stanno i processi che attendono I/O dal relativo device e sono manipolate dall'ssi.
In questo modulo stanno anche variabili che tengono conto del numero di thread nel sistema (threadCount) e il numero di thread bloccati in attesa di I/O o di completare una richiesta dall'ssi (softBlockCount).
Il nostro scheduler compie varie azioni:

- verifica se é scaduto momento per lo pseudo clock e ci sono processi nella waitForPseudoClockQueue, imposto la variabile isPseudoClock e impostandoo il valore dell'interval timer a 0

- se non ci sono thread pronti da eseguire e c'é solo un thread, questo é l'ssi e si ha un normale shutdown chiamando la HALT ROM routine

- se non ci sono thread pronti da eseguire ma nel sistema ci sono thread, nessuno in attesa di I/O o in attesa di completare una richiesta dall'ssi, allora si ha deadlock e si chiama la PANIC ROM routine

- sempre nel caso non ci siano thread pronti da eseguire ma nel sistema ci sono thread in attesa di I/O o in attesa di completare una richiesta dall'ssi, allora vuol dire che i thread sono in attesa di un interrupt. Se ci sono thread in attesa dello pseudo tick carico il valore dello pseudo clock nell'interval timer. Altrimenti impostiamo lo stato del processore con gli interrupt abilitati e entriamo in un loop

- se non c'è nessun thread in esecuzione ma c'e n'è almeno uno nella readyQueue allora carico un thread, imposto le variabili per la gestione del tempo e l'interval timer con il valore del time slice. É stato scelto di inserire tre nuove variabili per la gestione del time slice all'interno della struttura tcb_t: startTime, ovvero il tempo di inizio di esecuzione di quel thread, elapsedTime, cioé il tempo passato dall'inizio dell'esecuzione di quel thread ed executionTime, ovvero il tempo totale di esecuzione del thread

- se invece é passato il time slice inserisco il thread che era in esecuzione in coda alla readyQueue, carico un nuovo thread e aggiorno nuovamente l'interval timer con il time slice, a meno che non fosse scattato lo pseudo clock, in tal caso non modifico l'interval timer dato che scatterá subito l'interrupt dello pseudo clock e come ultime operazioni imposto la variabile elapsedTime a zero e startTime con l'attuale tempo. L'ultima operazione da fare é quella di caricare lo stato del currentThread nella cpu.

ssi.c

Il System Service Interface é un componente fondamentale in quanto fornisce servizi necessari per la costruzione dei livelli piú alti di AMIKaya, come la sincronizzazione dei thread con le operazioni di I/O, la gestione dello pseudo clock, la gestione della proliferazione e morte dei thread e la gestione dei thread specificati come gestori di trap.
Il ciclo di vita di un ssi é in modo molto semplificato simile a questo:

```
while(TRUE) {
        receive a request;
        satisfy the received request;
        send back the results;
}
```

Le funzioni da noi implementate in questo modulo sono:

- void SSIRequest(U32 service, U32 payload, U32 *reply)
  Tramite questa funzione un thread può richiedere un servizio.
  I parametri sono:
    service: codice che identifica il servizio
    payload: contiene un'argomento, se richiesto, per il servizio
    reply: punterà all'area dove verra messa la risposta nel caso sia richiesta
  La prima operazione é quella di fare una SEND, cioé richiedere il servizio all'ssi per poi mettersi in attesa con una RECV ed una volta sbloccato mettere la risposta nel campo puntato da reply.

- void SSI_function_entry_point()
  É il punto d'entrata dell'ssi. É costituito da un loop che come giá detto in si occupa di ricevere una richiesta, soddisfarla ed inviare la risposta. Viene dunque fatta una RECV che intercetta la richiesta fatta dal thread e viene invocata la funzione SSIdoRequest(ssimsg_t * msg_ssi, U32 *reply) che una volta terminata restituirá un la risposta al servizio nel campo puntato da reply e verrá fatta una SEND per sbloccare il thread che era in attesa.

- void SSIdoRequest(ssimsg_t * msg_ssi, U32 *reply)
  É colui che identifica la richiesta fatta dal thread e se la richiesta fatta dal thread non esiste allora esso e tutta la sua progenie verranno uccisi chiamando la funzione terminate(tcb_t *killed), altrimenti, questa verrá gestita chiamando la funzione relativa.

- tcb_t *createBrother(state_t *state)
  Crea un fratello al thread sender, il fratello deve avere come stato inziale lo stato passato come parametro. Se la creazione va a buon fine ritorna l'address del figlio altrimenti restituisce CREATENOGOOD.

- tcb_t *createSon(state_t *state)
  Crea un figlio al thread sender, il fratello deve avere come stato inziale lo stato passato come parametro. Se la creazione va a buon fine ritorna l'address del figlio altrimenti restituisce CREATENOGOOD.

- void killProgeny(tcb_t *parent)
  Funzione che uccide tutta la pogenie del thread passato come parametro.

- void terminate(tcb_t *killed)

Uccide il thread passato come parametro e tutta la sua progenie richiamando la killProgeny(tcb_t *parent).

- cpu_t *getCpuTime(tcb_t *tcb)
  Funzione che dato un thread ritorna il tempo totale di CPU che egli ha usato dall'inizio della sua creazione fino all'invio della richiesta stessa.

- void waitForClock(tcb_t *tcb)
  La funzione toglie il TCB passato come parametro dalla readyQueue e lo mette nella waitForPseudoClockQueue.

- U32 waitForIO(ssimsg_t * msg_ssi)
  Inserisce il thread nella coda dei thread da bloccare in I/O avvalendosi della funzione select_io_queue_from_status_addr(memaddr status_addr) presente nello scheduler.c e modifica una variabile in modo che la SEND dell'ssi non venga inviata. Sará poi la SEND all'interno della wakeUpFromIO(ssimsg_t * msg_ssi, U32 *reply) che vedendo il thread all'interno della coda lo sbloccherá. Se invece l'interrupt é giá arrivato viene comunque settata la variabile che evita che venga effettuata la SEND dell'ssi e viene inviata una SEND al thread che non viene inserito nella coda di attesa.

- U32 wakeUpFromIO(ssimsg_t * msg_ssi, U32 *reply)
  Toglie il thread dalla coda in attesa di I/O del relativo device e lo rimette nella readyQueue, gli invia una SEND e modifica un'apposita variabile che impedisce di inviare la SEND nell'ssi. Se nella coda di I/O non c'é alcun thread perché il device é stato piú veloce inserisco il payload in una struttura dati e setto la variabile in modo che l'ssi non invii la SEND e sará poi la SEND all'interno della waitForIO(ssimsg_t * msg_ssi) che vedendo che é arrivato prima l'interrupt sbloccherá il thread.

Dunque il servizio waitForIO viene richiesto da un thread, mentre il servizio wakeUpFromIO viene richiesto dal gestore degli interrupt. Il thread chiede la waitForIO e se arriva prma l'interrupt il gestore chiama la wakeUpFromIO che non vedendo il thread in coda alloca solo il messaggio ed una volta terminata la gestione degli interrupt l'ssi vede che é arrivato prima l'interrupt stesso e sblocca il thread. Altrimenti se la gestione dell'interrupt non interrompe l'ssi, questo mette il thread in coda e sará poi il gestore dell'interrupt che chiamando la wakeUpFromIO trovando il thread in coda lo sbloccherá.

- struct list_head* select_io_queue_from_status_addr(memaddr status_addr)
  É una funzione ausiliaria utilizzata dall'ssi che ci serve per scoprire in base all'address mappato in memoria del registro status del device, in quale coda di wait (delle 5 presenti nello scheduler) mettere il thread che sta aspettando I/O dal device stesso.

interrupt.c

La funzione principale presente in questo modulo é la intsHandler(). É questo il punto di entrata per la gestione degli interrupt. Le prime azioni da compiere sono l'aggiornamento della variabile elapsedTime del TCB in esecuzione e salvare il suo stato d'esecuzione. Poi si procede con l'identificazione di quale device ha generato l'interrupt. Se l'interrupt é stato generato da disk, tape, printer o terminal si chiama la funzione acknowledge, se invece é stato generato dal timer, allora chiameremo a funzione ftimer(), e come ultima azione chiamiamo lo scheduler.
La funzione ftimer(), ovvero la procedura chiamata per gestire interrupt generati dal timer controlla se l'interrupt é stato causato dallo pseudo clock controllando la variabile booleana isPseudoClock. Se risulta vero resettiamo la variabile stessa ponendola uguale a false e se sono presenti thread nella

waitForPseudoClockQueue inviamo una richiesta all'ssi di WAKE_UP_PSEUDO_CLOCK per svegliarli, altrimenti significa che l'interrupt del timer é stato generato dal time slice e non facciamo niente siccome lo scheduler viene chiamato in ogni caso come ultima azione dalla funzione intsHandler().
La funzione acknowledge(U32 intLineNo) é invece la funzione chiamata quando é stato identificato come generatore dell'interrupt un qualsiasi device tranne il timer. Le azioni che compie questa funzione sono prima di tutto guardare qual'é il device number del device che ha sollevato l'interrupt e scrivere nel suo command register. É cosí che un device riceve un ack. Siccome il numero del device che ha generato l'interrupt si trova nella propria interrupting devices bitmap, come prima cosa mi calcolo l'indirizzo della bitmap, poi con un ciclo la scorro tutta fino a trovare un uno. La posizione in cui l'uno é presente indica il numero del device. Se il device che ha generato l'interrupt non é stato il terminale allora calcolo l'indirizzo del suo status register e ci scrivo DEV_C_ACK.
Nel caso in cui a generare l'interrupt sia invece stato il terminale, dobbiamo tenere conto che ha due registri command, uno per la trasmissione e uno per la ricezione e quindi per sapere su quale mandare l'ack dobbiamo conoscere se l'azione che ha generato l'interrupt era stata una send o una recv. Trovo dunque come nei casi precedenti l'indirizzo dello status register e a seconda che a generare l'interrupt sia stata una trasmissione e/o una ricezione, scrivo DEV_C_ACK nel relativo deviceCommandAck.
In ogni caso, come ultima azione chiameremo la funzione void fakeSend(U32 dest, U32 payload) che alloca un nuovo messaggio e lo mette nella inbox dell'ssi come servizio WAKE_UP_FROM_IO. Questa funzione sostituisce la SEND per evitare che si entri nella gestione delle eccezioni, che potrebbe generare problemi.

exception.c

La parte più importante per la gestione dei messaggi. In questo modulo sono presenti le 2 SYSCALL principali del microkernel:

- MsgSend(): alloca un messaggio con i dati passati come payload e lo inserisce nella coda inbox del tcb_t destinatario.
  Abbiamo diversificato 2 casi possibili: il primo caso l'abbiamo quando il messaggio deve essere inviato al SSI l'altro quando deve essere inviato a un tcb normale.
  In caso di successo restituisce MSGGOOD altrimenti MSGNOGOOD.

- MsgRecv(): è una SYSCALL bloccante. Quando viene chiamata resta in attesa di un messaggio solo nel caso in cui la lista inbox sia vuota. Nel momento in cui arriva il messaggio viene sbloccato il thread che ha invocato la recive passandogli come payload quello del messaggio che ha ricevuto. Restituisce il puntatore al tcb che ha inviato il messaggio.

prgTrap.c e tlbTrap.c

In questi moduli sono presenti le funzioni prgHandler() e tlbHandler(), cioé le funzioni che gestiscono le Trap Exception, eccezioni che si sollevano quando il thread in esecuzione tenta di compiere azioni illegali o indefinite.
Servono a salvare il vecchio stato del processore al momento dell'eccezione e il nuovo che dovrà essere caricato nello stato del processo corrente.

Nel caso arrivi un'eccezione e non è presente un handler il thread che l'ha generato verrà eliminato assieme a tutta la sua progene.

utils.c

oltre alla funzione memcopy() che serve a fare la copia di 2 aree di memoria per la fase di debugging ci siamo riscritti le funzioni principali(modificate per i nostri scopi) della libC.
La piu importante è la ker_printf(). Questa si usa come la printf() solo che invece di stampare su terminale “stampa” su un'area di memoria.