Relazione di consegna per il progetto
AMIKaya11

Sviluppato da Alain Di Chiappari

Progetto per il corso di Sistemi Operativi, scritto, sviluppato e testato su Intel x86 con sistema operativo Ubuntu Linux v10.04 e umps2 v1.9.7. I livelli realizzati sono Phase1 e Phase2.

---------- **PHASE1**----------

Questo livello implementa due moduli, uno per la gestione dei thread, e uno per i messaggi.
Essenzialmente phase1 fornisce liste e gestione di queste utili ai livelli soprastanti.

TCB.C
Questo modulo fornisce supporto per la gestione dei thread, creando innanzitutto una lista libera di Thread Control Block (struttura atomica fondamentale del sistema) quindi la loro allocazione e distruzione (resa alla lista dei TCB liberi).
Vengono implementate inoltre funzioni per la creazione di code e liste di thread, gestibili grazie a quelle per l'aggiunta/rimozione/verifica di appartenza di TCB a tali code.
Fondamentali sono anche le funzionalità fornite per la gestione di alberi di TCB, utili a mantenere una gerarchia esecutiva nel sistema, con relazioni padre-figlio-fratello.

MSG.C
Molto simile al modulo per i TCB, per buona parte lo rispecchia riguardo alla gestione delle liste. Si differenzia in quanto fornisce funzionalità di aiuto al message passing di phase2, implementando ad esempio funzioni per le operazioni di POP e PUSH di messaggi nelle inbox di TCB.
Anche in questo caso abbiamo una lista di messaggi liberi nel sistema, dalla quale vengono rimossi o resi i messaggi, in base a quanto richiesto dal nucleo in esecuzione.

---------- ***PHASE2***----------

BOOT.C
Qui risiede l'entry point del nucleo, oltre alla dichiarazione di alcune importanti variabili globali utilizzate per tutto il tempo di vita dello stesso.
Innanzitutto vengono popolate le four New Areas, ovvero quelle zone di memoria che devono contenere lo stato che verrà caricato allo scattare di eccezioni.
A questo proposito il Program Counter viene impostato al punto di entrata dell'Handler e lo stato viene settato in modo che esso abbia interrupt disabilitati, che sia ovviamente in Kernel Mode e con memoria virtuale spenta; tutto ciò per garantise massima protezione, accesso, e non interrompibilità.
Al boot inoltre vengono inizializzate le costanti, create ed inizializzate liste e code del sistema, nel particolare la coda Ready, che conterrà i processi pronti per essere eseguiti, e la coda Wait, contenente quei processi in attesa di un qualche evento, e soprattutto messaggi.
Infine si crea uno stato per *l'SSI* e uno per il thread con entry point *test()* che vengono messi in Ready Queue in quest'ordine, viene fatto partire lo Pseudo Clock e il controllo viene fatto passare allo sheduler.

SCHEDULER.C

Questo modulo si occupa principalmente della gestione dei thread sulla CPU, e quindi del caricamento su di essa.
Lo scheduler individua innanzitutto due situazioni:

Nessun processo è attualmente attivo.
Si arriva in questa situazione subito dopo il boot o quando è scaduto il time slice di un thread, o ancora se è stato congelato/terminato un processo che era attualmente in esecuzione.
In questa situazione è necessario controllare innanzitutto se vi sono dei processi in Ready Queue da poter caricare, in caso favorevole, viene estratto il primo, vengono aggiornati i tempi per lo Pseudo Clock e il campo *cpu_slice* del TCB in questione viene posto a zero per partire, quindi si carica il tempo minimo fra il time slice e il rimanente Pseudo Tick sul Interval Timer e si carica lo stato del thread sulla CPU che quindi parte per l'esecuzione del proprio codice.
Nel caso in cui la Ready Queue fosse vuota vengono distinti ulteriori 3 casi.

- C'è un unico processo nel sistema. Questo processo sarà perciò in wait, e sarà presumibilmente l'SSI, quindi a questo punto è evidente che si è in uno stato in cui è necessario lo spegnimento. Si invoca quindi la routine della ROM, HALT().
- C'è più di un processo nel sistema, ma contemporaneamente nessuno sta aspettando un servizio o una risposta da I/O. In questo stato significa che si è creata una situazione circolare di wait fra thread, e perciò è individuato un *deadlock*. Ciò che resta da fare è invocare la routine PANIC().
- C'è più di un processo nel sistema, e almeno un thread sta aspettando la terminazione di un servizio o di un I/O. Ciò che resta da fare è aspettare che succeda qualcosa, quindi si setta lo stato del processore disponibile a ricevere un qualsiasi interrupt, e si mette lo scheduler in attesa (mediante un ciclo infinito, o con la routine WAIT()).

Al momento c'è un processo attivo (il cui TCB è puntato da *current_thread*).
Si arriva in questa situazione tutte le volte che viene scatenato un evento (eccezione/interrupt) che non ha però avuto l'effetto di mettere in Wait Queue o terminare il processo corrente.
In questo caso viene aggiornato lo Pseudo Tick, e si carica sull'Interval Timer il minimo tempo rimanente fra lo scadere del time slice del processo corrente e il rimanente Pseudo Tick.
Infine viene ricaricato il processo in modo tale che riprenda la sua esecuzione per il time slice che gli resta.

In entrambi i casi prima di settare l'Interval Timer si controlla che lo Pseudo Tick non abbia superato il suo tempo massimo, evento che potrebbe accadere quando si passa il controllo allo scheduler dal gestore delle eccezioni/interrupt se è prossimo lo scadere del Tick; in questo caso si setta l'Interval Timer a 1 così che appena gli interrupt saranno riabilitati si potrà gestire con più precisione lo Pseudo Clock scattando un'eccezione per interrupt sulla linea 2.

--- HANDLERS ---

Interrupts.c ed Exceptions.c sono i due moduli che gestiscono gli avvenimenti “straordinari” che avvengono durante la vita del processore.
Principalmente implementano le quattro funzioni corrispondenti ai Program Counter delle four New Areas popolate al boot; le eccezioni possono essere per l'appunto di quattro tipi: Interrupt, Program Trap, TLB, SYS/BP.
Ciò che accomuna i quattro gestori sono le seguenti attività:

- Salvataggio dello stato del processo corrente (se al momento dell'eccezione c'è n'era uno attivo) dalla Old Area corrispondente in cui è stato posizionato dalla routine della ROM al TCB.
- Aggiornamento dei campi temporali del TCB e per la gestione dello slice.

- Eventuale aggiornamento Pseudo Tick.
- Chiamata allo scheduler alla fine della procedura.

Nello specifico invece:

INTERRUPTS.C

Questo modulo implementa il gestore di eccezioni interrupt.
AMIKaya11 non ha interrupt software (linee 0 e 1), quindi vengono gestite le rimanenti linee dalla 2 alla 7. Si ricorda che la linea 2 si può pensare come attiva per un unico “device”, che non possiede a differenza degli altri un Device Register, e che la linea 7 comprende due subdevice per ogni device (e quindi Device Register).
Inizialmente viene prelevato il registro *cause* salvato nella Old Area e si determina quale linea ha scatenato l'interrupt, data la possibilità di più linee che contemporaneamente possono aver generato un interrupt, si controlla dalla linea con indice minore (alta priorità) a quella con indice maggiore (bassa priorità), creando una sorta di loop per la gestione di tutti gli interrupt pendenti, dato che innanzitutto ne viene gestito uno alla volta, ed inoltre, appena *IEc* riabilita le interruzioni si rientra nell'handler.

Per quanto riguarda le linee dalla 3 alla 6 le attività sono praticamente identiche, ciò che viene fatto è prima di tutto caricare la bitmap della linea trattata per poi individuarne all'interno con lo stesso principio di priorità per le linee il device che ha generato l'interruzione; questo viene fatto mediante la funzione *which_device*, alla quale viene fornita una bitmap e restituisce l'indice del device a più alta priorità con interrupt pendente su quella linea.
A questo punto viene individuato il device register corrispondente, cosi da scrivere il comando di acknowledgment nel campo command ed inviare all'SSI un messaggio che ha come mittente proprio il Device Register trattato, e come payload il campo status dello stesso.
Il messaggio verrà riconosciuto dall'SSI mediante il campo *service* della struttura *SSI_request_msg* che conterrà il magic number INTERRUPT_MSG.

La linea 2 mette in moto il cuore della gestione delle tempistiche; si può dividere in due casi:
Interrupt per Pseudo Clock.
Viene riconosciuto questo stato nel caso in cui lo Pseudo Tick ha raggiunto o superato il suo massimo, quindi viene inviato un messaggio all'SSI (con priorità, quindi il messaggio è messo in testa), che si occuperà di svegliare tutti i thread in attesa di questo evento, con mittente BUS_INTERVALTIMER e *service* PSEUDOCLOCK_MSG.
Viene ovviamente riportato a zero lo Pseudo Tick e si fa ripartire il conteggio.
Interrupt per Time Slice.
Si entra in questo stato in modo non esclusivo rispetto al precedente, vista la possibilità del verificarsi di entrambe le condizioni quando si è in prossimità di entrambe le scadenze.
In questo caso si sposta il thread corrente nella Ready Queue e si setta a NULL *current_thread*.

La linea 7 supporta anch'essa otto device, i quali però essendo terminali, sono composti a loro volta da due subdevice, uno per la trasmissione (alta priorità) e uno per la ricezione (bassa priorità).
Tutto quanto detto per le linee 3-6 vale anche per questa, eccezion fatta per il demultiplexing dei due subdevice, gestendo quindi due campi per lo stato, due campi per i comandi, due messaggi diversi a seconda del subdevice, e l'acknowledgment rivolto a quello che sto trattando.

In questo modulo le strutture per i messaggi per l'SSI sono mantenute a livello globale come un array (s*truct SSI_request_msg interrupt_msg_array[]*), e vengono usate in modo circolare con un indice interno al modulo, incrementato ogni qual volta se ne fa uso.

EXCEPTIONS.C

Oltre alle azioni sopra descritte, i moduli implementano procedure adeguate a seconda dell'eccezione avvenuta.

SYS/BP Exception Handler.

Si entra in questo gestore ogni qualvolta viene sollevata un'eccezione di tipo SYSCALL o BREAKPOINT; AMIKaya11 si basa per sostanzialmente su due sole system call, nello specifico sulle due primitive di message passing (send/receive), richiamabili in kernel mode con codice rispettivamente 1 e 2 perciò il cuore del sys/bp exception handler sta quindi nell'implementazione di questo message passing, eseguita nel momento in cui sono soddisfatte queste circostanze.

Si ricorda che vengono utilizzati i registri a0, a1, a2 per il passaggio di parametri, e ciò vale anche per le Syscall in questione.

- Send: innanzitutto viene fatto un controllo su mittente, destinatario e payload del messaggio, si vuole infatti distinguere il caso in cui un Trap Manager invia un messaggio di *TRAPTERMINATE* / *TRAPCONTINUE* ad un thread precedentemente bloccato in attesa di una decisione (vedi sotto). Viene quindi rispettivamente terminato, o risvegliato il thread obiettivo.

Nel caso "normale" invece viene invocata la funzione *send()* che prende come parametri un mittente, un destinatario ed un payload (che nel caso di *SSIRequest* sarà un puntatore alla struttura per le richieste appena creata).

Questa funzione, alloca un messaggio e lo inserisce nella inbox del destinatario nel caso in cui questo fosse in Ready Queue, il thread corrente, o in Wait Queue in attesa di un altro messaggio, in modo tale che potrà essere in futuro prelevato.

Nel caso in cui invece il thread destinatario aspetta un messaggio da questo thread (o da chiunque, ANYMESSAGE), esso viene spostato in Ready Queue e gli viene restituito il payload nella locazione specificata nella corrispondente *recv()*, salvata nel campo *reply* del suo TCB.

- Receive: per entrambi i casi trattati (ANYMESSAGE e con Thread specificato) viene prima di tutto effettuata *una popMessage()* per verificare la presenza del messaggio cercato.

In caso positivo viene liberato il messaggio, posizionato il payload in esso contenuto nella locazione specificata e restituito il destinatario; si tratta quindi di un'operazione non bloccante.

Diventa bloccante quando invece non è presente nella inbox il messaggio cercato, in quanto viene rimosso dal thread corrente il receiver e spostato in Wait Queue, specificando nei campi del suo TCB (*waiting_for* e *reply*), da chi sta aspettando un messaggio e dove vuole ricevere la risposta.

Associate a queste operazioni, SYS/BP Exception Handler riconosce le situazioni di scambio di messaggi da e verso l'SSI, fornendo innanzitutto supporto alla protezione del suo TCB trasformando il magic number MAGIC_SSI nell'indirizzo reale associato, ed inoltre incrementa/decrementa il *soft_block_count* ogni qual volta un thread richiede/ottiene un servizio.

In tutti gli altri casi, e quindi: SYS/BP non in Kernel Mode, SYSCALL diversa da SEND/RECV, BREAKPOINT, ***TLB Trap***, ***Program Trap***, l'eccezione viene gestita come segue.

Si verifica che il thread corrente abbia speficato un Trap Manager per l'eccezione trattata, mediante i servizi messi a disposizione dall'SSI.

In caso negativo viene direttamente terminato il thread e tutta la progenie (il sottoalbero dei processi).

In caso positivo invece viene bloccato il thread e spostato in Wait Queue, settando il campo *waiting_for* del suo TCB con il Trap Manager adeguato, al quale verrà inoltre inviato un messaggio contenente il registro *cause* salvato nella Old Area ed in base a questo il Manager deciderà cosa fare, rispedendo un messaggio TRAPTERMINATE/TRAPCONTINUE al thread bloccato, gestito come sopra descritto.

Terminate()
Questa funzione, utilizzata abbondantemente in questo modulo, ma anche dall'SSI, ha la funzione di terminare un sottoalbero di thread, prendendo come parametro la radice dello stesso.
La funzione agisce ricorsivamente, terminando foglia per foglia tutto l'albero, fino ad arrivare alla radice, richiamando la funzione stessa su ogni figlio, fin quando ce ne sono.
Nel caso base, la *terminate()* si occupa di "pulire" tutto quanto è possibile della presenza del thread nel sistema, eliminandolo perciò dall'eventuale lista in cui si trova, annullando la variabile *current_thread* se è il thread attualmente in esecuzione, restituendo alla lista dei liberi tutti i messaggi che aveva in inbox, cancellandolo dai Manager nel caso ne facesse parte e decrementando il *soft_block_count* se era in attesa di servizi o I/O.

--- --- ---

SSI.C
Ciò che fornisce un'interfaccia al sistema è questo modulo, attraverso i servizi implementati e accedibili comunque mediante le due SYSCALL send e recv.
Ciò che viene innanzitutto implementato è il wrapper *SSIRequest()*, una funzione che in modo molto semplice agisce in questo modo:

- Crea una struttura *SSI_request_msg*.
- Mappa i campi passati (service, payload, reply) nei tre campi appositamente creati della struttura request.
- Invia un messaggio (con una *MsgSend*) all'SSI che ha come destinatario un fake address MAGIC_SSI e come payload, il puntatore alla struttura creata.
- Aspetta un messaggio (con una *MsgRecv*) dall'SSI, che avrà nel corso della sua esecuzione "spacchettato" la struttura passata per ottenerne le informazioni utili all'attivazione della procedura corretta.

Il thread dell'SSI vive la sua esecuzione in un continuo loop, all'inizio del quale si mette in attesa di un messaggio, che viene poi demultiplexato alla procedura corretta in base al campo *service* estratto dalla struttura *SSI_request_msg*.
La priorità viene data innanzitutto ai messaggi che arrivano dall'Interrupt Handler, e quindi per lo Pseudo Clock e per la gestione degli interrupt associati ai WAITFORIO.
Per i dettagli sull'interfaccia dei servizi si consigliano le specifiche delle stesse; per quanto riguarda invece le scelte di implementazione sui meccanismi propri della progettazione, possono essere riassunte come segue.

- Viene dichiarato un *devStatus_array* globale, nel quale, al momento di gestione del messaggio per gli interrupt, verrà memorizzato il campo *status* del Device Register da cui arriva il messaggio (inviato dall'Interrupt Handler) qualora non sia ancora arrivato un WAITFORIO relativo a questo device.
- C'è dualmente un *devTcb_array* globale, nel quale si memorizzerà il puntatore a TCB per il thread che ha appena fatto WAITFORIO per un device dal quale Device Register non è ancora arrivato un messaggio di interrupt.
- Allo stesso modo, per il WAITFORCLOCK, viene creato un *clockTcb_array*, dove si memorizzeranno i puntatori a TCB di quei thread in attesa dello Pseudo Tick, che saranno poi in ordine svegliati dalla routine apposita dell'SSI.

I primi due array vengono gestiti con una politica di indicizzazione, aiutata dalla funzione *rest_index()*, che associa ad ogni Device Register (indirizzo) un indice che per l'appunto rappresenta una posizione nell'array di aiuto all'SSI. Vista la presenza dei due subdevice per la linea 7, ci sono due indici per ogni device, associati all'indirizzo del campo status degli stessi.

MANAGER.C

Per finire, questo modulo è di semplice supporto per la “contabilità” del nucleo per quanto riguarda i Trap Manager specificati dai thread con le apposite chiamate a SSI.
Manager.c implementa infatti un *trap_managers* array, contenente i puntatori a TCB dei manager attualmente presenti nel sistema, e un piccolo insieme di funzioni basilari utili alla gestione dello stesso.

---------- *NOTE*----------

- Testato e funzionante su umps2 v. 1.9.7.
- Il tar.gz contiene già pronta una configurazione del kernel (l'ultima).

[Dare il comando "make clean && make all" per crearne una nuova.]

- Presenti warning del tipo: "warning: implicit declaration of function `PANIC'" su routine fornite come PANIC, HALT, STST etc.