MIKONOS Operating System Project

documentazione phase 2

Alberti Michele
Cozza Giovanni
Pomili Luigi

Luglio 10, 2008

Introduzione:

Questo documento vuole descrivere, in breve, l'implementazione della phase2 del progetto “MIKONOS Operating System”. Viene dato maggiore spazio alle scelte e politiche implementative riguardanti le strutture dati utilizzate e in generale la struttura del codice. Per quanto riguarda la phase1, vedere il file di documentazione “DOCphase1AB”.

Layout del codice:

Il tree del progetto, in breve, puo' essere letto nel file README.
Piu' in dettaglio:

```
lso08az13                  /
|
|- AUTHORS                 gli sviluppatori e rispettive mail
|- LICENSE                 licenza di utilizzo
|- Makefile                file per una compilazione guidata, con check
                           dell'architettura host
|- README                  informazioni in breve sul progetto
|- doc/                    documentazione del progetto
   |
   |- DOCphase2            questo file
   |- DOCphase1AB          file di documentazione per phase1

|- h/                      tutti gli header files utili al progetto
   |
   |- base.h               definizioni di tipi, per comodita'
   |- const.h              definizioni di costanti e macro
   |- our_const.h          definizioni di nostre costanti
   |- types.h              definizioni di tipi riguardanti il progetto
   |- mikonos.h            definizioni delle strutture dati riguardanti
                           direttamente il progetto di MOKONOS
|- e/                      tutte le interfacce riguardanti phase1A,B / 2
   |
   |- tcb.e                gestione dei Thread Control Block e delle code
                           di TCB
   |- tid.e                gestione dei Thread ID e non solo
   |- interrupt.e          gestione interrupt HW e comunicazione Kernel-SSI
   |- syscall.e            implementazione delle 3 uniche syscall
   |- scheduler.e          gestione avvicendamento processi e non solo
   |- ssi.e                interfaccia per la SSIRequest
   |- prgtrap.e            interfaccia per le program trap
   |- tlbtrap.e            interfaccia per le tlb trap

|- phase1/                 tutti i sorgenti riguardanti phase1A,B
   |
   |- test/                tutti i sorgenti e interfaccie dei test
                           ufficiali
      |
      |- p1atest.c         file di test per la phase1A
      |- p1btest.c         secondo file di test per la phase1A
      |- p1b2test.c        file di test per la phase1B
```

```
      |- p1b2test.e        interfaccia per il file di test p1b2test.c
      |- Makefile          file per la compilazione guidata dei test

   |- src/                 tutti i sorgenti da noi sviluppati
      |
      |- tcb.c             gestione Thread Control Block e codi di TCB
      |- tid.c             gestione dei Thread ID e funzioni direttamente
                           utilizzabili per gestione dei TCB
      |- boot.c            gestione della sequenza di boot di MIKONOS
                           Operating System (codice per solo phase1)
      |- Makefile          file per la compilazione guidata dei sorgenti

|- phase2/                 tutti i sorgenti riguardanti phase2
   |
   |- test/                sorgente riguardante il test ufficiale
      |
      |- p2test.c          file di test per la phase2
      |- Makefile          file per la compilazione guidata del test

   |- src/                 tutti i sorgenti da noi sviluppati
      |
      |- boot.c            gestione della sequenza di boot di MIKONOS
                           Operating System, ampliato rispetto a quello
                           scritto per phase1B
      |- scheduler.c       gestione avvicendamento processi, secondo
                           politica round robin, e non solo
      |- interrupt.c       gestione interrupt HW e comunicazione tra Kernel
                           e thread SSI
      |- syscall.c         implementazione delle 3 uniche syscall
      |- ssi.c             implementazione servizi e SSIRequest
      |- prgtrap.c         gestione delle eccezioni causate operazioni non
                           permesse, istruzioni illegali etc.
      |- tlbtrap.c         gestione delle eccezioni causate da TLB miss
      |- Makefile          file per la compilazione guidata dei sorgenti
```

Il processo di compilazione e' gestito dal Makefile contenuto nella directory lso08az13. Per avere informazioni dettagliate su tale processo leggere il file README.

Scelte e politiche implementative riguardanti le strutture dati:

1. Struttura dati tcb_t.

   Sono state apportate delle aggiunte alla struttura presentata nella documentazione di phase1:

   - ***outbox**, coda dei TCB che hanno fatto richiesta di ricevere dal thread proprietario, permette una gestione del tutto analoga rispetto alla coda *inbox;
   - **last_payload,** permette di accedere all'ultimo messaggio spedito dal TCB senza dover sempre accedere ai veri e propri registri;
   - **to_kill**, flag che permette la discriminazione dei TCB che debbono essere terminati. Ogni volta che il thread deve essere eliminato

(tramite richiesta alla SSI oppure perche' ha eseguito azioni non permesse) il flag to_kill viene posto a TRUE. Questo permette allo scheduler (nella funzione **schedule()**) di riconoscere e gestire facilmente i thread da eliminare;

- **cpu_snapshot**, immagine del processore proprio del thread in modo da poter sospendere e poi riprendere l'esecuzione del medesimo. Ogni qual volta che si entra in codice kernel (interrupt, syscall, etc) lo stato del processore viene salvato nella struttura state_t cpu_snapshot, questo permette di riprendere la computazione da dove si era interrotta al prossimo dispatch del thread;
- **cpu_time**, tempo di processore utilizzato dal thread dalla sua inizializzazione alla sua terminazione, viene aggiornato ogni volta che la computazione viene interrotta e si passa ad eseguire codice kernel;
- **cpu_remain**, tempo rimanente di processore da utilizzare per il thread al prossimo dispatch. Il tempo di processore (time slice) assegnato ad ogni thread e' di 5 ms, come da specifiche. Quando pero' il thread non ne usufruisce completamente, (ad esempio a causa di una interrupt) al successivo dispatch gli vengono assegnati i ms non utilizzati.
  Cio' permette di ottenere un sistema il piu' reale possibile ed una politica di scheduler corretta ed equa.
- **prgtrap_m**, **tlb_m, sys_m**, TID degli eventuali gestori delle program trap, tlb trap e syscall/breakpoint trap del thread. Vengono settati su richiesta alla SSI.

2. Struttura dati request_t (mikonos.h).

```
typedef struct {
     unsigned int SSIservice;
     unsigned int payload;
} request_t;
```

Questa struttura dati e' pensata per poter comunicare con la SSI, in particolare per richiedere un servizio tramite la SSIRequest(..).
Il primo campo denota il servizio che il thread richiede alla SSI.
Il secondo campo rappresenta le informazioni necessarie (eventualmente) alla SSI per adempiere al servizio richiesto.
Questa struttura dati viene passata per indirizzo alla SSI, ricordiamo infatti che l'oggetto dello scambio tra i thread deve essere un unsigned int. Grazie a questa struttura:

- si rispettano le richieste delle specifiche;
- la comunicazione con la SSI e' semplice ed efficiente;
- si evitano numerosi scambi di messaggi tra SSI e thread richiedente altrimenti necessari per comunicare il tipo di servizio e il payload eventualmente richiesto.

Questa struttura dati viene utilizzata esclusivamente all'interno del modulo della SSI e in particolar modo nell'implementazione della funzione SSIRequest(..).

3. Struttura dati ssi_dev (mikonos.h).

```
typedef struct {
     unsigned char   happened_req;
     unsigned char   happened_int;
     tid_t requestor[8];
} ssi_dev;
```

Questa struttura dati e' pensata per risolvere il problema esplicitato al punto “4.9.5 Data Structures for I/O Management” delle specifiche.
Il primo campo (pensato come bitmap) tiene traccia delle richieste pervenute alla SSI per l'i-esimo dispositivo.
Il secondo campo (pensato come bitmap) tiene traccia degli interrupt pervenuti alla SSI per l'i-esimo dispositivo.
Il terzo campo determina il TID del thread richiedente l'i-esimo dispositivo.

Per conoscere in dettaglio l'utilizzo e il funzionamento, vedere la descrizione della SSI piu' avanti.

Dettagli sui moduli implementati:

- **Boot (rivisitato).**

  Una prima fase di scrittura del boot di MIKONOS e' stata fatta in phase1. Vedere DOCPhase1AB.
  Abbiamo aggiunto l'inizializzazione dello scheduler, **initScheduler()**, e il codice di creazione ed inizializzazione dei primi due thread del sistema, SSI e test.

- **Scheduler.**

  Il modulo che implementa lo scheduler e' il piu' importante del progetto dato che gestisce l'avvicendamento dei threads e mantiene la temporizzazione del sistema (pseudo clock). Questo modulo, monitora anche lo stato del sistema e le eventuali condizioni di deadlock che si potrebbero verificare tramite l'utilizzo di due unsigned int: thread_count (mantiene il conteggio dei thread nel sistema) e softb_count (mantiene il conteggio dei thread che stanno aspettando per la terminazione di operazioni di I/O o il tick dello pseudo clock).
  Lo scheduler gestisce i thread in base a due code di tcb_t: ready_queue e killable_queue.
  La prima contiene tutti i thread che sono in stato READY_THREAD e che possono eseguire, la seconda contiene tutti i thread che devono essere terminati chiamando la force_terminate(..).

  La funzione **schedule()** e' la funzione principale del modulo.
  Viene richiamata alla fine di ogni modulo implementato e inizialmente passa in rassegna tutta la ready_queue in modo da spostare eventuali tcb_t con flag to_kill (a TRUE) nella killable_queue. A questo punto scandisce tutta la killable_queue chiamando per ogni tcb_t presente

la funzione HIDDEN **force_terminate(..)** in modo da terminare effettivamente il thread e rilasciare tutte le sue risorse.
Se la ready_queue risulta vuota vengono eseguiti i controlli di verifica del deadlock o halt del sistema. Nel primo caso il sistema e' in “kernel PANIC” mentre nel secondo caso, SSI unico thread vivo, il sistema termina in modo regolare “System Halt”.
Vi e' un terzo caso, quando tutti i thread compresa la SSI sono sospesi in attesa. Questo stato si verifica quando tutti i thread hanno richiesto operazioni di I/O oppure stanno aspettando messaggi da altri thread. In questo caso l'interval timer viene settato al valore dello pseudo count (mantiene i ms rimanenti al prossimo tick dello pseudo clock) e vengono abilitate le interrupt ponendosi successivamente in attesa (loop infinito) che se ne verifichi una o che arrivi il tick dello pseudo clock.
Se la ready_queue non risulta vuota allora viene chiamata la funzione HIDDEN **upThread()** che gestisce il dispatch dei thread. La funzione non fa altro che prelevare il primo tcb_t in testa alla ready_queue e assegnargli un quanto di tempo (5ms o i ms restanti del precedente quanto di tempo) per poi caricare il suo cpu_snapshot nel processore. La upThread() gestisce anche la temporizzazione necessaria all'implementazione dello pseudo clock.

Qual'e' l'idea implementativa dello pseudo clock?

Il modulo “scheduler.c” mantiene una variabile unsigned int (pseudo_count) la quale viene inizializzata con il valore di SCHED_PSEUDO_CLOCK (100ms), come da specifica. Ogni volta che l'esecuzione di un thread viene sospesa per eseguire codice kernel (interrupt, syscall etc) viene calcolato il rispettivo tempo che passa prima del prossimo dispatch di un qualsiasi thread. Il tempo cosi' calcolato viene ogni volta, in **upThread()**, sottratto a pseudo_count. Quando questo valore risulta prossimo/minore (cercando di evitare underflow) di quello del time slice di default (5ms), l'interval timer viene settato al valore di pseudo_count.
Questa soluzione ci e' sembrata la piu' lineare da implementare e precisa. Ogni modulo del progetto ha in testa una sequenza di istruzioni per la gestione e il calcolo del tempo in codice kernel. Il tutto risulta facile da capire e uniforme da scrivere.

Sulla terminazione dei thread:

Di particolare importanza e' la funzione **force_terminate(tid_t bill)** utile alla terminazione di un qualsiasi thread. Abbiamo cercato di adottare una politica di terminazione/kill dei thread uniforme. Qualsiasi thread che dovesse essere terminato (o per richiesta alla SSI o per avvenuto errore) viene “marcato” settando il campo to_kill, della propria struttura tcb_t, a TRUE. Il rispettivo tcb_t viene poi re-inserito nella ready_queue in modo che in un secondo momento la funzione schedule(), vedi sopra, lo termini chiamando in ultimo luogo proprio la force_terminate(..).
Da notare come la force_terminate(..) preveda il re-inserimento in ready_queue di tutti quegli eventuali TCB che fossero accodati in inbox/outbox del TCB da terminare. Questo scelta implementativa

permette di riabilitare tutti i TCB che sono per un qualche motivo nelle code di un thread che deve essere terminato. In particolare, questa politica permette di eliminare (ad una seconda schedule()) effettivamente tutti quei TCB “marcati” (to_kill) accodati nelle code del thread che subisce la force_terminate(..). Si evitano cioe' dei thread zombie, cosa non richiesta nelle specifiche ma che ci sembrava giusto da implementare.

- **Syscalls.**

  MIKONOS e' un microkernel, pertanto i meccanismi di gestione della comunicazione e sincronizzazione avvengono tramite “message passing”. Le uniche syscalls fornite da MIKONOS sono tre, rispettivamente il servizio per spedire messaggi (sempre bloccante), quello per riceverli (in alcuni casi bloccante) e quello per rispondere ad un messaggio (mai bloccante). Per una visione dettagliata del funzionamento di tali servizi, vedere le specifiche.
  MsgSend, MsgRecv e MsgReply non sono altro che macro contenenti l'istruzione SYSCALL correlata dei corretti parametri. Ogni volta che uMPS esegue una SYSCALL carica i registri $a0, $a1 e $a2 con i valori necessari alla chiamata - seguendo la convenzione di chiamata MIPS – e ritorna il valore di questa nel registro $v0.

  Come ogni altro modulo da noi implementato, il codice del modulo “syscall.c” viene eseguito grazie al pass-up fornito dalla ROM. In particolare, la funzione principale **sys_bp_handler()** viene eseguita ogni volta che un thread esegue una SYSCALL oppure un BREAK.
  Come in tutti gli altri moduli, le prime istruzioni mirano a contribuire ad una giusta gestione della temporizzazione, dell'accounting del tempo di cpu del thread appena sospeso e del salvataggio dello stato del processore all'interno dello state_t cpu_snapshot del thread stesso.

  A tal proposito descriviamo la funzione **initCPUSnap(..)** utilizzata da tutti i moduli implementati, che intervengono quando l'esecuzione di un thread viene sospesa. Questa funzione (vedi tcb.c) serve al salvataggio - registro dopo registro - dello stato attuale del processore all'interno dello state_t cpu_snapshot passatogli. Si riesce cosi' a riprendere l'esecuzione del codice del thread da dove era stata sospesa, con il medesimo stato del processore.

  Le implementazioni dei comportamenti che seguono la chiamata di una syscall sono in SYSMsgSend, SYSMsgRecv e SYSMsgReply.

  - **SYSMsgSend:** controlla se nella coda di outbox e' presente il TCB a cui si vuole spedire il messaggio. Se lo trova lo “risveglia” ponendolo in testa alla ready_queue, comunque sia accoda il TCB del mittente nella inbox del destinatario. Va fatto notare come la presenza della coda tcb_t *outbox generi una gestione omogenea delle implementazioni delle “send” e delle “recv”. Il codice ne guadagna in leggibilita' e semplicita'.

  - **SYSMsgRecv:** controlla se vi sono segnalazioni di avvenuti interrupt

(questo solo per la SSI) altrimenti si comporta in modo speculare rispetto alla SYSMsgSend.

- **SYSMsgReply:** l'implementazione della “reply” ad una precedente “send”, impedisce che si verifichino delle situazioni di errore, come ad esempio risposte a thread che non sono accodati nella inbox. Per raggiungere questo scopo, particolare importanza e' l'uso degli stati in cui i thread possono trovarsi. Da notare inoltre come il protocollo di Send-Recv-Reply sia rispettato anche nei confronti delle segnalazioni (interrupt, pseudo clock) provenienti dal kernel.

Il modulo prevede anche un minimo di controlli atti a prevedere situazioni che porterebbero il sistema in uno stato di errore.
Notare come infatti sia presente il controllo che evita di spedire/ricevere/rispondere a thread inesistenti. Sono stati previsti anche i controlli che gestiscano le situazioni nelle quali possono essere coinvolti i gestori esterni delle system call, proprieta' dei thread che ne avessero fatto richiesta con successo.
Particolare attenzione va posta nel codice di SYSMsgRecv eseguito dalla SSI. Quest'ultima e' infatti l'unico thread che puo' accedere alle variabili che segnalano avvenuti interrupt HW.
Il modulo fa un utilizzo massiccio di alcune macro, qui descritte:

```
#define INTERRUPTED_I(int_i, n) \
     while(!((int_i) & (n))) ((n) = (n) << 1) :
```

permette di individuare il primo bit di valore 1 all'interno della bitmap int_i;

```
#define BASE_OFFSET(num)  ((num) * DEV_REG_SIZE) + DEV_REGS_START :
```

calcola l'indirizzo base, in memoria, dato l'indice del dispositivo passato;

```
#define BIT_NUM(bit, n)  while (((bit) = ((bit) >> 1))) ((n)++) :
```

trasforma la bitmap passata nel numero corrispondente.

- **Interrupts.**

L'unica via tramite la quale la CPU puo' comunicare al software il comportamento dell'HW del sistema, e' attraverso il meccanismo degli interrupt. Si entra ad eseguire il codice di questo modulo ogni qual volta avviene una interrupt proveniente o dai dispositivi o dallo “interval timer”.

Dopo le istruzioni previste per la gestione della temporizzazione, l'accounting del tempo di cpu per il thread appeno sospeso e il salvataggio dello stato del processore tramite la **initCPUSnap(..)**, viene individuata la linea di interrupt (tra quelle pendenti) piu' importante con successiva gestione.
Particolare attenzione va posta per quanto riguarda la gestione delle linee di interrupt dalla 3 alla 7 (anche la linea 2 nel caso di tick

dello pseudo clock). La gestione di tali dispositivi chiama la funzione HIDDEN **KtoSSI(..)**, scritta per la comunicazione tra kernel e “demone” SSI.

Il problema: in MIKONOS il message passing tra thread avviene tramite l'accodamento dei tcb_t, cioe' qualsiasi thread che voglia spedire un messaggio ad un secondo thread si accoda nella coda inbox di quest'ultimo. Analogamente avviene per una ricezione di messaggio. Ora il problema e' chiaro, il kernel vero e proprio non e' un thread, non ha un proprio tcb_t. Come fare?

Variabili utilizzate (tutte dichiarate HIDDEN):

- **just_psuedo**: utilizzata per segnalare avvenuto tick dello pseudo clock;
- **interrupted_device**: U32 pensata come bitmap, ogni bit determina l'avvenuto interrupt di un particolare dispositivo;
- **interrupted_terminal**: U32 pensata come bitmap, vengono utilizzati solo i primi 16 bits. Ricordiamo che i terminali sono 8, ma e' come se fossero il doppio dato che ricezione e trasmissione possono avvenire in concorrenza. Ecco spiegato il perche' di una bitmap separata.

Notare come le bitmap consentono di gestire piu' interrupt contemporaneamente, cosa potenzialmente possibile.

La **KtoSSI(..)** implementa la “comunicazione” tra kernel e SSI. L'idea alla base e' abbastanza semplice; ad ogni interrupt la KtoSSI(..) setta in modo appropriato le variabile sopra elencate - a seconda del dispositivo e della linea di interrupt - in modo da poter segnalare l'accaduto alla SSI. Dall'altra parte, quest'ultima eseguendo una usuale MsgRecv(ANYTID, ..), controlla le stesse variabili riuscendo a capire (ed in seguito gestire) l'avvenuto interrupt.

Perche' non utilizzare la MsgSend(..)?

Abbiamo implementato la comunicazione tra kernel e SSI in questa maniera perche' ci sembrava il modo piu' corretto dato che:

- il modulo di gestione delle interrupt fa parte del vero e proprio kernel, quest'ultimo non ha motivo di invocare una syscall per eseguire tali operazioni;
- invocare MsgSend(..) - che una syscall - all'interno di codice kernel non e' del tutto corretto, almeno concettualmente.

Puntualizzazioni:

- la funzione KtoSSI(..) e' dichiarata HIDDEN, questo per evidenziare il fatto che nessun thread puo' invocarla;
- notare che all'interno della SYSMsgRecv soltanto alla SSI e' permesso l'accesso alle variabili sopra elencate, oltretutto tramite delle funzioni get/set in stile OOP.

Il modulo “interrupt.c” non fa uso di particolari strutture dati ma utilizza in modo massiccio alcune macro da noi definite (vedi anche h/our_const.h). Riportiamo qui le piu' importanti:

```
#define DEV_BASE_ADDR_I(int_no, dev_no) \
DEV_REGS_START + (((int_no) - 3) * 0x80) + ((dev_no) * 0x10) :
```

viene utilizzata per calcolare l'indirizzo base del dispositivo, data la linea di interrupt e il numero di dispositivo. All'indirizzo ottenuto si trovano i quattro registri utilizzati per la gestione del dispositivo stesso.

```
#define F_DEV_PEND(int_no, n, i) \
while ((i++ < 8) && (!(*((memaddr*)(PENDING_BITMAP_START \
+ (WORD_SIZE * ((int_line) - 3)))) & (n)))) \
     ((n) = ((n) << 1)) :
```

stabilisce il numero del dispositivo piu' importante allacciato alla linea di interrupt int_no. Una volta stabilito il numero del dispositivo che ha causato l'interrupt, tramite la DEV_BASE_ADDR_I, e' possibile calcolare l'indirizzo base del dispositivo per poi leggere/scrivere i valori dei suoi registri.

- **System Service Interface (SSI).**

La SSI e' un componente fondamentale del kernel di MIKONOS. E' un demone, fornisce un insieme di servizi che vengono utilizzati da tutti gli altri thread presenti nel sistema. Per conoscere i suoi dettagli rimandiamo al punto “4.3. System Service Interface and Trap Management Threads” delle specifiche.

Il ciclo di esecuzione della SSI e' il seguente:

```
while (1) {
      ricezione di una richiesta
      gestione della stessa
      risposta al thread richiedente
}
```

il quale permette al kernel di minimizzare i “tempi morti” di attesa e di ridurre i ritardi nella gestione dei servizi.

Abbiamo implementato la SSI come un thread in interrupt abilitate in modo da poter supportare una corretta gestione della temporizzazione e soprattutto dello pseudo clock. Piu' in particolare abbiamo implementato la parte di “ricezione di una richiesta” in interrupt disabilitare, in modo da evitare problemi di sincronizzazione; le restanti due parti pero', sono in interrupt abilitate.
Riportiamo qui i due servizi piu' particolari che la SSI fornisce:

- **WAITFORCLOCK.**

Tutti i thread richiedenti tale servizio si sospendono fino al prossimo tick dello pseudo clock.
Implementare il lato della richiesta e' risultato abbastanza facile: i thread richiedenti vengono accodati nella coda tcb_t*wfc.
Il lato della risposta e' stato invece piu' difficile da concepire ed implementare; la SSI gestisce la risposta a WAITFOKCLOCK quando il “sender” della richiesta risulta il kernel (ANYTID, vedere a tal proposito pagina 18 delle specifiche) e il payload ha valore BUS_INTERVALTIMER, l'indirizzo base del clock di sistema.
Conseguentemente conclude la gestione del servizio riattivando tutti i thread che fossero bloccati in *wfc, cioe' piazzandoli tutti in cima alla ready_queue mantenendo quell'ordine stabilito dall'arrivo delle richieste.

- **WAITFORIO.**

  Questo servizio e' di gran lunga piu' complicato da implementare rispetto al precedente, soprattutto perche' richieste e eventi provenienti dai dispositivi possono avvenire in qualsiasi ordine.
  Compito della SSI e' quello di gestire l'accoppiamento tra richieste e eventi per/di un particolare dispositivo. Abbiamo a tal proposito ideato una struttura dati, ssi_dev, (vedi paragrafo “Scelte e politiche implementative riguardanti le strutture dati”) la quale permette di eseguire il “matching” in tempo costante e senza uno spreco eccessivo di memoria. Viene utilizzato un array di sei elementi (wfio[6]), tanti quanti le linee di interrupt presenti (da 3 a 7) considerando anche il fatto che i terminali trasmettitore e ricevitore sono gestiti separatamente.
  Per ogni linea di interrupt, quindi, una ssi_dev permette di monitorare lo stato degli otto dispositivi allacciati, capire cioe':
  - tramite la bitmap unsigned char “happened_int” quale dispositivo ha segnalato una interrupt;
  - tramite la bitmap unsigned char “happened_req” a quale dispositivo e' stata fatta richiesta di I/O;
  - non solo, tramite l'array tid_t requestor[8] sapere il TID del thread richiedente.

  Il lato della richiesta, attiva a seconda del dispositivo voluto, il giusto bit della bitmap “happened_req” e controlla in seguito se la segnalazione del kernel per tale dispositivo e' gia' avvenuta osservando il corrispondente bit nella bitmap “happened_int”. In caso positivo “risveglia” il thread inserendo il suo TCB nella ready_queue, altrimenti lo sospende inserendo il TID nella giusta posizione all'interno dell'array requestor.
  Il lato della risposta (gestione della segnalazione da parte del kernel) si comporta in modo analogo ma contrario rispetto all'utilizzo delle variabili.
  Sia nell'implementazione del lato della richiesta che in quello della risposta, utilizziamo due operazioni che vogliamo qui spiegare piu' dettagliatamente.
  Sono le seguenti:

- dev_base % 8 : indice del dispositivo, tra 0 e 7, utile per il settaggio del bit all'interno delle bitmap “happened_req” ed “happened_int”;
- dev_base / 8 : indice della linea di interrupt, utile per individuare quale struttura ssi_dev modificare nell'array wfio.

Il modulo “ssi.c” ospita anche l'implementazione della funzione **void** SSIRequest**(unsigned int** service**, unsigned int** payload**, unsigned int*** reply**)**, interfaccia attraverso la quale tutti i thread richiedono servizi alla SSI.

Come avviene la “comunicazione”?

Il thread invocante la **SSIRequest(..)** si ritrova a fare una “send” alla SSI, bloccandosi in attesa della risposta da parte di quest'ultima. Oggetto del messaggio e' sempre un unsigned int, questa volta pero' pensato come puntatore ad una struttura dati request_t (vedi paragrafo “**Scelte e politiche implementative riguardanti le strutture dati**”). Il thread viene “svegliato” tramite “reply” da parte della SSI, ritornando il risultato della richiesta per modifica del registro $v0 del thread richiedente.

Anche il modulo che implementa il demone SSI utilizza alcune macro, qui riportate:

#define OFFSET_BASE(ind) ((ind) - DEV_REGS_START) / DEV_REG_SIZE :

calcola l'indice del device dato il suo indirizzo base in memoria;

#define NUM_BIT(bit, n) while (((n)--) > 0) ((bit) = ((bit) << 1)) :

trasforma il numero passato nella bitmap corrispondente.

- **Program/TLB Trap Handler.**

  Rimandiamo alla visione dei moduli “prgtrap.c” e “tlbtrap.c” dato la loro semplicita'.

Ultima nota sull'esecuzione del test di phase2 (p2test):

Il test termina in “kernel PANIC()”; non e' un errore come si potrebbe pensare. Tale risultato e' dovuto al fatto che non tutti i thread vengono terminati, rimangono correttamente attivi i seguenti quattro thread:

- SSI (non verrebbe comunque terminato);
- printthread;
- p5trapm;
- p9trapm.

Gli ultimi tre thread non vengono terminati perche' non vi sono specifiche richieste, alla SSI, di terminazione e soprattutto perche' MIKONOS non implementa nessuna sorta di parentela tra thread.