Phase 2: MiKABoO the kernel! DOCUMENTATION

gruppo 11

Informazioni Generali

Escludendo i files di phase1 e quelli necessari per iniziare phase2 il lavoro è composto da:

main.c file che contiene la funzione di inizializzazione e avvio del kernel e la funzione dello scheduler.

handlers.c file contenente le funzioni per la gestione delle eccezioni e la funzione per la verifica dello pseudoclock.

ssi.c che contiene la funzione della SSI e le procedure dei suoi servizi.

utility.h header file richiamato nei file .c sopra citati che contiene definizioni di funzioni e di variabili utili alle procedure implementate.

E' necessaria una nota riguardante lo stile di scrittura del codice.
Nell'ottica di un codice più comprensibile e lineare spesso
sono state dichiarate più variabili di quelle strettamente utili (1);
tra le diverse funzioni si è cercato di dare lo stesso nome a variabili che si riferiscono allo stesso oggetto (2) ,
ed sono state richiamate alcune parti dei file dati all'inizio (3).

Sono riportati alcuni esempi per spiegare quanto sopra detto.

(1) devaddr* devCommand;
devaddr* devStatus;

In questo caso ho usato due variabili (di tipo puntatore a **device address**) per differenziare il campo **command** da quello **status** nonostante ne sarebbe potuta bastare una da incrementare o decrementare all'occorrenza.

(2) struct pcb_t* pproc ;

Lo stesso nome di variabile è utilizzato in funzioni diverse per ragioni di velocità di scrittura del codice e uniformità dello stesso.

(3) struct {
struct tcb_t* s;
} req = { ... }

Nelle procedure per i servizi della SSI ho sempre richiamato la stessa struttura presente nel file nucleus.h al momento dell'inizializzazione della richiesta. In alcuni casi, come in quello proposto, la ripresa dell'intera struttura potrebbe sembrare una ripetizione, ma crea un parallelismo tra la funzione che chiede il servizio e quella che lo soddisfa.

Modifiche a phase1

Alcune aggiunte sono state fatte ai file di phase1 per avere strutture e funzioni più espressive e utili per la scrittura del codice di phase2.

Prima di spiegare le aggiunte fatte c'è da notificare una modifica al file **uARMconst.h** nei pressi di riga 205 : è stato aggiunto un

```
#ifndef NULL
#define NULL  // riga già presente
#endif
```

perché durante la compilazione venivano sollevati dei warning riguardanti la ridefinizione della costante NULL in questo file. Così la compilazione risulta molto più pulita e comprensibile.

Passiamo alle aggiunte a phase1.

(0) `#define T_STATUS_W4IO  5`

Questa costante verrà spiegata meglio in seguito durante la discussione della gestione dell' I/O. Evita errori nel caso in cui un thread venga ucciso mentre sta attendendo la fine di un'operazione di input/output.

(1) Nella `struct pcb_t` sono stati aggiunti tre nuovi campi:

```
struct tcb_t * pgmmgr;
struct tcb_t * tlbmgr;
struct tcb_t * sysmgr;
```

che sono i puntatori al **Program Manager**, **TLB Manager** e **System Manager**.
Questi puntatori sono inizializzati a NULL dalla funzione `struct pcb_t *`**`proc_init`**`(void);` per poi essere opportunamente settati dalla SSI quando richiesto.

(2) Nella `struct tcb_t` ho aggiunto altri campi:

```
unsigned int cputime;
int errno;
uintptr_t* t_command;
struct list_head t_generic;
```

che rispettivamente servono per :
> tenere l'accounting del tempo trascorso in CPU dal thread;
> memorizzare il codice di errore dell'ultima operazione(SYSCALL) fatta;
> puntare a una struttura contenente i valori necessari all'inizio di una operazione di input/output (questo campo verrà discusso più avanti quando parlerò della gestione dei device);
> inserire il thread in una coda "generica" che ha significato in base al contesto.
In particolare questo campo è utile nelle funzioni di gestione dello pseudoclock e nella gestione dell'input/output;

Come per gli altri , la funzione void **thread_init**(void); provvede ad inizializzare correttamente i campi aggiunti.

(3) Sono state aggiunte alcune funzioni che spiegherò qui nel dettaglio:

(a) void **thread_generic_enqueue**(struct tcb_t *new, struct list_head *queue);

struct tcb_t ***thread_generic_qhead**(struct list_head *queue);

struct tcb_t ***thread_generic_dequeue**(struct list_head *queue);

static inline void **thread_generic_outqueue**(struct tcb_t *this)

Queste funzioni sfruttano il campo **t_generic** fornendo le usuali operazioni permesse su una struttura di tipo coda. Rispecchiano le funzioni per la gestione delle code dello scheduler utilizzando **t_generic** al posto di **t_sched**.

(b) void **ssi_push**(struct tcb_t *new, struct list_head *queue);

Questa funzione inserisce un thread in testa ad una coda dello scheduler.
La funzione richiede un po' di spiegazione dato che và contro la politica FIFO delle code.
La procedura è stata aggiunta per motivi di ottimizzazione e deve essere usata con l'unico scopo di inserire il thread SSI in cima alla **ready queue**, dopo che è stato nella **waiting queue**.
Nel codice la funzione è infatti utilizzata solo all'interno di
static inline void **wake_up**(struct tcb_t * tcb ,struct tcb_t* dest); con lo scopo sopra citato.
Essendo la SSI il thread più richiesto dell'intero sistema, ho deciso di fare in modo che abbia priorità su gli altri threads così che le loro richieste siano soddisfatte il prima possibile.
Quando la SSI è svegliata da una **msgsend** sarà il primo thread ad essere schedulato e potrà risolvere la richiesta il più velocemente possibile.
So che andare contro la politica FIFO non è canonico, ma ritengo che bisogni dare questa priorità alla SSI.

Analisi delle funzioni implementate

In seguito analizzerò le funzioni implementate suddividendole in base al file in cui si trovano. Mi soffermerò solo sulle parti degne di nota, tralasciando il codice che non ha bisogno di spiegazione.

main.c

int **main**(int argc, char* * argv[]);

Funzione che inizializza tutte le strutture necessarie all'avvio del sistema, prepara la SSI e il thread di test e chiama lo scheduler.
Da notare è l'assegnazione della memoria ai thread: i gestori delle eccezioni condividono tutti la stessa area di memoria (primo frame a partire da RAMTOP) dato che non è possibile avere più di un gestore attivo contemporaneamente.

La SSI lavora nel secondo frame a partire da RAMTOP e, come per i gestori, ha gli interrupt disabilitati così da operare in mutua esclusione ed evitare race condition.

void **schedule**(void);

La parte interessante dello scheduler riguarda il caso in cui la **ready queue** è vuota.
Se c'è un unico thread ed è la SSI allora spegniamo il sistema.
Altrimenti viene effettuata una verifica di deadlock :
se il **numero di thread** nel sistema è maggiore di 0 e il **numero di thread soft blocked** è uguale a 0 allora siamo in deadlock e il sistema va in KERNEL PANIC.
Con **thread soft blocked** si intendono tutti quei threads che stanno aspettando un servizio dalla SSI. Sono così chiamati perché la durata dell'attesa ha sicuramente un tempo finito, dato che prima o poi la SSI accetterà la loro richiesta.
Infine se non si ricade negli altri casi, significa che i threads stanno aspettando la fine di un'operazione di input/output oppure un tick dello pseudoclock .
Il sistema è posto in WAIT con tutti gli interrupt abilitati. Inoltre l' interval timer è settato alla costante TIMESLICE, così che ogni 5 millisecondi si possa controllare se è avvenuto il tick dello pseudoclock. In questo modo infatti, dopo TIMESLICE microsecondi di stato WAIT viene chiamata la routine di gestione degli interrupt che verifica lo pseudoclok.

handlers.c

static inline void **wake_up**(struct tcb_t * tcb ,struct tcb_t* dest);

static inline void **wait**(struct tcb_t * tcb , struct tcb_t* src, const int repeat);

Queste due funzioni "risvegliano" e "addormentano" un thread. Sono prese in considerazione nella documentazione dato che in queste procedure è maggiormente presente la logica di gestione dei **thread soft blocked**.
Inoltre la funzione **wait** permette di specificare se far ripetere o no al thread l'ultima istruzione eseguita.

static inline void **fulfil_timeslice**(struct tcb_t* tcb , const unsigned int timer);

Ho deciso che qualora un thread non abbia finito il suo **timeslice** e le condizioni lo permettano, esso possa continuare ad utilizzare la CPU per il tempo rimanente.
In questo modo quando il thread esegue azioni non bloccanti, come per esempio una **msgsend** , una **msgrecv** in cui trova il messaggio oppure è fermato dalla gestione di un interrupt (che non sia del tipo INT_TIMER), questo possa completare il tempo che gli spetta nel processore.
Un'altra soluzione sarebbe stata quella di rimettere il thread in **ready queue**, ma mi è sembrata poco efficiente e poco logica soprattutto perché il sistema, essendo di tipo micro kernel, si basa fortemente sull'uso di **msgsend** e **msgrecv.**

static inline void **pseudoclock_check**(void);

Questa procedura gestisce la logica dello pseudoclock. Viene richiamata all'inizio di ogni funzione di gestione delle eccezioni e si occupa di verificare se il tick dello pseudoclock è avvenuto. Nel caso in cui siano passati 100 millisecondi risveglia tutti i threads che si trovano in una particolare coda.
La coda sopracitata è quella che ha come punto d'ingresso il campo **t_generic** della SSI; per questo thread, quel particolare campo è assegnato alla gestione dei thread che aspettano lo pseudoclok.
Il servizio WAIT_FOR_CLOCK implementato dalla funzione
static inline void **__wait_for_clock**(struct tcb_t* sender); non fa nient'altro che accodare il thread richiedente alla coda di cui stavamo parlando.
Come intervallo di tolleranza è stato scelto un range che va da 100 a 106 millisecondi. Ho scelto questo intervallo considerando che il sistema se va in stato WAIT vi ci può rimanere per 5 millisecondi. Scegliendo un range minore si rischia di entrare in stato WAIT attorno ai 100 millisecondi per poi uscirvi ai 105 e quindi avere un fallimento nella verifica dello pseudoclock.

void **SysBpExceptionHandler**(void);

Il primo gestore delle eccezioni di cui parliamo è quello che riguarda le SYSCALL.

Se la system call è di tipo SYS_SEND o SYS_RECV e il thread che la esegue è in STATUS_USER_MODE allora viene dichiarato nel campo **CP15_Cause** del **t_s** che è avvenuto un errore del tipo EXC_RESERVEDINSTR ed sono eseguite le procedure per l'attivazione del **ProgTrapExceptionHandler**. In questo modo sarà il gestore delle program trap ad occuparsi del thread .Il codice rispecchia quello che l'emulatore uarm esegue al momento del sollevamento di un'eccezione di quel tipo.

Se la syscall è fatta in STATUS_SYS_MODE allora viene presa in considerazione.
Sono trattati ora i casi di **msgsend** e **msgrecv**.
Nel caso della SYS_SEND, dopo aver preso tutti i parametri, la prima cosa che si verifica è se il thread a cui invio il messaggio è ancora vivo. Come condizione, quì e negli altri casi, viene verificato il campo **t_sched** del thread; infatti un thread è vivo se e solo se si trova in una delle tre code di scheduling. Ciò che ho appena spiegato è usato anche altrove come condizione per verificare se un thread è vivo o no. Un altro modo per eseguire questa verifica è quello di controllare il campo **t_status** del thread , se il campo ha valore T_STATUS_NONE il thread può essere considerato morto.
La send verso un thread morto porta a non inviare il messaggio .

Nel caso in cui il sender sia un manager del destinatario, sicuramente il messaggio che viene inviato serve per risvegliare il thread che stava aspettando una decisione del manager stesso. Il thread risvegliato non andrà però ad eseguire una **msgrecv** o una **msgq_get**, quindi, per non sprecare messaggi, basta solamente risvegliarlo senza eseguire una **msgq_add**. I controlli fatti per intercettare il messaggio giusto sono probabilmente più di quelli necessari

if (**is_mgr**(dest,tcb) && dest->t_wait4sender == tcb && payload == (uintptr_t)NULL)

li ho comunque messi tutti così da essere sicuri che verrà considerato solo il messaggio spedito da un manager a un thread in attesa, per risvegliarlo e consentire la sua continuazione.

Con la SYS_RECV cerco il messaggio nella nostra coda: se l'operazione fallisce, innanzi tutto controllo che il thread da cui aspetto il messaggio sia ancora vivo, poiché aspettare da un thread morto può portare a deadlock.
Se sto per riceve da un thread morto, ho scelto come soluzione di rendere la receive non bloccante e di ritornare NULL come valore del sender e del payload.
Questo caso è l'unico in cui è possibile avere come valore di ritorno della **msgrecv** la costante NULL nel sender. Ci sono casi in cui il payload è NULL e il sender è diverso da NULL; ho deciso, per ragioni di coerenza, di mettere anche il payload a NULL pur non essendo necessario.
Bisogna fare attenzione perché solo se la **msgq_get** fallisce controllo se il thread è morto, poichè se la funzione ha successo si ottiene un messaggio che è stato mandato da un thread che al momento della send era vivo. Nel momento in cui leggo il messaggio quel thread potrebbe anche essere morto; spetta al thread ricevente in questo caso controllare se il thread è vivo o morto a seconda delle sue necessità (vedi per esempio la void **SSI_function_entry_point**(void);).

Altrimenti metto il thread in attesa ponendolo nella **waiting queue** e facendo in modo che ripeta la SYS_RECV quando viene svegliato dal thread che manda il messaggio.

Se la SYSCALL ha valore 0 allora va trattata come un errore terminando il thread,
se ha valore maggiore di 2 faccio il pass-up ai managers del processo se sono stati dichiarati, altrimenti termino il thread. Quando faccio il pass-up eseguo la **wait** sul thread che ha fatto la SYSCALL facendo in modo che non ripeta l'ultima istruzione fatta, altrimenti il thread andrebbe a rieseguire la syscall errata se il manager ne concede il proseguimento.

void **ProgTrapExceptionHandler**(void);

void **TLBTrapExceptionHandler**(void);

Questi due handler sono trattati assieme in quanto molto simili.
L'unica considerazione da fare riguarda sempre la funzione
static inline int **wait**(struct tcb_t * tcb , struct tcb_t* src, const int repeat);
che mette il thread che ha sollevato l'eccezione in attesa, facendogli ripetere l'ultima istruzione eseguita ovvero quella che ha causato l'attivazione dell'handler. Dopo il pass-up ,se il manager decide la continuazione del thread e risolve l'eccezione, allora il thread dovrebbe essere in grado di rieseguire l'istruzione incriminata senza più problemi.
Altro punto da sottolineare è il fallimento della **msgq_add** , la quale ha un ruolo non secondario nella funzione. Quì, come già fatto in un caso nel **SysBpExceptionHandler**, viene adottata una soluzione drastica terminando il processo e la sua progenie.

void **InterruptExceptionHandler**(void);

Dato che questo handler è strettamente collegato alla funzione di gestione del servizio DO_IO della SSI

static inline void **__do_io**(uintptr_t* rinfo, struct tcb_t* sender);

in questa parte spiegherò anche questa funzione e la gestione generale dell'input e dell'output.

L' **InterruptExceptionHandler** è diviso in due fasi.
La prima è volta a riconoscere il dispositivo che ha sollevato l'interrupt e a capire quali sono i campi **device command** e **device status** da considerare.
La seconda parte , quella dopo l'acknowledgement dell'interrupt, è un po' più complicata e necessita di maggiore spiegazione (vedi dopo).

Prima di passare alla trattazione della gestione dell'I/O tratto il caso in cui l'interrupt sia di tipo INT_TIMER. L'handler semplicemente richiama lo scheduler e sarà lui ad occuparsi di resettare l'interval timer , notificando l'avvenuta gestione dell'interrupt e di schedulare il prossimo processo.

I/O management

Come accennato prima, ora parlerò dell'I/O management che coinvolge non solo l'**InterruptExceptionHandler** , di cui qui finirò la trattazione, ma riguarda anche le funzioni

```
static inline void __do_io(uintptr_t* rinfo, struct tcb_t* sender);

inline void __set_dev_fields(uintptr_t* rinfo);

static inline void get_dev(memaddr devStatus, int* line_no , int *device_no);
```

Inoltre per permettere una veloce gestione dell'input/output e consentire operazioni in parallelo su devices differenti è stata inizializzata nel main e dichiarata in **utility.h** una matrice

```
struct list_head io_matrix[DEV_USED_INTS][DEV_PER_INT];
```

che ha tante righe quanti sono le linee di interrupt (escluse le linee 0,1 e INT_TIMER) e tante colonne quanti sono i devices per linea (installati o non installati).
Ogni elemento io_matrix[i][j] rappresenta una lista di threads (collegati attraverso il campo **t_generic**) che attendono il completamento di un'operazione di input/output dal dispositivo 'j' della linea 'i'.

Quando un thread richiede il servizio DO_IO viene messo dalla SSI nella apposita coda della io_matrix , ottenuta grazie alla funzione **get_dev**, e trova nel suo campo **t_command** il puntatore alla struttura che contiene tutti i dati necessari per l'operazione di I/O. Se il thread è l'unico in quella coda, viene eseguita l'operazione grazie alla funzione **__set_dev_fields**, altrimenti il thread rimane in coda poiché qualcun altro sta aspettando la conclusione di un operazione su quel device.

Se un altro o più threads sono in coda, uarm non permette di accodare le richieste al device driver, così l' io_matrix bufferizza i threads che necessitano di I/O e l'insieme di dati necessari a questa richiesta.

Una volta completata un'operazione sarà la seconda parte dell'**InterruptExceptionHandler** ad occuparsi della io_matrix.
Il primo thread in coda viene tolto e gli viene mandato un messaggio con mittente la SSI e come campo il valore del registro **device status**. La **msgq_add** in questo particolare caso è estremamente importante, quindi ho deciso che un suo fallimento, dovuto all'assenza di messaggi nella lista libera, debba portare a un KERNEL PANIC con relativo messaggio di errore.
In seguito l'handler verificherà se c'è almeno un altro thread nella stessa coda e darà inizio all'operazione di I/O grazie al campo **t_command** presente nel thread.

La costante T_STATUS_W4IO entra qui in gioco per evitare uno sfasamento nella gestione dell'I/O causato dalla morte del thread che sta attendendo il completamento dell'operazione. Prima di notificare la conclusione dell'operazione al thread verifichiamo che sia effettivamente lui ad averla richiesta controllando il suo campo t_status. Per ogni coda dell'io_matrix può, infatti, esserci al più un thread (quello di testa) con quel campo settato alla costante sopracitata. Se risulta essere lui il thread che sta attendendo l'I/O procediamo come sopra descritto altrimenti ci limitiamo a proseguire con il prossimo senza inviare messaggi a destinatari non corretti.

Nonostante il campo **t_command** contenga sempre un valore opportuno ,è possibile considerarlo valido se e solo se il thread si trova nella **io_matrix**, altrimenti il suo valore non deve essere preso in considerazione(viene comunque posto a NULL quando il thread non è nella matrice per coerenza e per evitare che si acceda a una struttura non più valida).
Nel caso in cui ci sia un altro thread nella coda, non sarebbe necessario prima fare l'acknowledge dell'interrupt e poi inserire il nuovo comando, dato che per uarm un nuovo comando notifica quello precedente. Ho però scelto di non sovrapporre le cose, così da avere maggiore chiarezza tenendo separati i due momenti.
Questa scelta è inoltre stata fatta considerando il funzionamento della rete.
Questo device infatti è l'unico che si comporta in modo diverso dagli altri nel caso in cui operi nella modalità a interrupts abilitati. In questa modalità bisogna tener conto di un interrupt in più, quello che viene sollevato dall'arrivo di un pacchetto. Ho deciso che se arriva questo tipo di interrupt, semplicemente lo notifico e basta; successivamente un processo eseguirà una READNET per leggere il pacchetto e si comporterà come una qualsiasi altra operazione di input.

Quindi all'arrivo di un interrupt controllo

if (line_no == INT_UNUSED && *devStatus >= READPENDING)

ovvero se l'interrupt è dalla rete e se lo **status** è maggiore di 128, costante che indica l'abilitazione degli interrupt e la presenza di un pacchetto. In questo caso non si fa nulla, poiché l'interrupt che ci è arrivato notifica solo l'arrivo di un pacchetto, e non l'avvenuta lettura da parte di un thread. Il manuale di uarm è poco specifico in questa sezione e in particolare non dice nulla sull'esecuzione di un comando READNET prima dell'arrivo dell'interrupt. Ho supposto che in questo caso il doppio interrupt non si verifichi ma ne avvenga solo uno, quello di notifica dell'operazione eseguita. Un'altra supposizione fatta riguarda il contenuto del campo **device status** in caso di un operazione di READNET il quale deve essere minore di READPENDING dato che il pacchetto è stato letto.

Ricordo in oltre che l'input da terminale si comporta come una qualsiasi altra operazione di I/O perchè come afferma il manuale di uarm :

"A character is received, and placed in RECV STATUS.RECVD-CHAR only after a RECEIVECHAR command has been issued to the receiver."

La **__set_dev_fields** impartisce il comando al device stando attenta alle differenze tra essi (per esempio solo la rete usa il campo DATA1);
la **get_dev,** dato l'indirizzo di un **device status,** ne ricava la linea di interrupt ed il numero del device. Serve per capire a quale elemento della io_matrix mi sto riferendo.

ssi.c

Nonostante in questo file siano presenti molte funzioni, tratterò solo le più significative ovvero quelle in cui ci sono scelte implementative da spiegare. La maggior parte delle funzioni in questo file non necessita di spiegazione in quanto il funzionamento dovrebbe essere facilmente comprensibile dalla lettura del codice.

void **SSI_function_entry_point**(void);

Funzione che gestisce tutte le richieste a servizi del nucleo; dopo aver ricevuto un messaggio si controlla se il **sender** è ancora vivo per poi andare a soddisfare la sua richiesta. Il controllo sul sender è necessario poiché esso potrebbe essere morto a causa di un suo parente (fratello , padre o antenato). In questo modo si evita che la SSI vada a compiere operazioni su un thread che si trova nella **free list**, e che comunque non è valido nel sistema non essendo in alcuna coda di schedulazione.

static inline void **__do_io**(uintptr_t* rinfo, struct tcb_t* sender);

inline void **__set_dev_fields**(uintptr_t* rinfo);

static inline void **get_dev**(memaddr devStatus, int* line_no , int *device_no);

Queste funzioni sono già state spiegate nella sezione dell'I/O management.

inline void **__terminate_thread**(struct tcb_t* sender);

Questa funzione è di grande importanza, in quanto si occupa di terminare un thread e fare in modo che vengano risvegliati i threads bloccati ad aspettare messaggi dal thread morto.
Inizialmente il thread viene tolto dalle code in cui si trova, la sua coda di messaggi viene svuotata ed è rimesso nella lista libera. Sono aggiornate le variabili di sistema **thread_count** e **soft_block_count.**

Successivamente si controlla la **waiting queue** alla ricerca di thread che aspettano un

messaggio dal thread morto. Questi threads sono risvegliati e, come nel caso in cui viene fatta una **msgrecv** ad un thread morto, troveranno il valore di ritorno a NULL e l'**errno** settato. Molto importante è l'istruzione

tcb->t_s.pc = tcb->t_s.pc + 0x4;

la quale evita che il thread ripeta la receive. Ricordo infatti che, quando un thread è messo in waiting per un messaggio, si fa in modo che ripeta la receive quando viene risvegliato. Anche in questo caso avrei potuto semplicemente risvegliare il thread senza settare l'errno e porre **ts.a1** a NULL, facendogli rifare la **msgrecv**; sarebbero però comparsi problemi di reincarnazione.

Compilazione

Per la compilazione è necessario avere tutti i files nella stessa directory.
Il file **.tar.gz** consegnato contiene tutti i file necessari per il funzionamento della phase2.

Vi notifico un piccolo cambiamento che ho fatto nella compilazione, in particolare al momento in cui tutti i files sorgenti e le librerie di uarm sono linkati.
Per phase1 è stato utilizzato il comando

arm-none-eabi-ld

che però in questo caso porta ad un errore durante la compilazione del tipo

undefined reference to `memcpy'

Errore dovuto all'assegnamento di strutture: il compilatore aggiunge la memcpy nelle parti del codice in cui le strutture vengono assegnate/copiate.

Mi sono documentato su internet e la soluzione che propongo è quella di usare il comando

arm-none-eabi-gcc –specs=nosys.specs

In questo modo il problema sembra risolto.

Gruppo 11
Frioli Leonardo