2 Phase 2: MiKABoO the kernel!

(versione 0.1.1)

Lo scopo della seconda fase del progetto è di realizzare il nucleo del sistema operativo Mikaboo. Il codice sviluppato dovrà implementare l'astrazione di processo e di thread, le primitive per la comunicazione, e la gestione delle periferiche di I/O.

Il livello del nucleo si deve occupare di gestire tutte le eccezioni e la schedulazione dei thread.

Mikaboo è un microkernel che fornisce due sole system call per spedire e ricevere i messaggi (rispettivamente *msgsend* e *msgrecv*). Per tutti i servizi del sistema esiste un thread del kernel chiamato System Service Interface (SSI), un processo server (daemon) che elabora le richieste dei processi. Operazioni quali la creazione di un processo o l'attivazione di una operazione di I/O vengono richieste dai processi spedendo un opportuno messaggio alla SSI ed attendendo la risposta.

2.1 Inizializzazione del livello del nucleo

Prima di iniziare il ciclo normale di funzionamento, il kernel deve inizializzare tutte le strutture dati necessarie.

In particolare il kernel deve:

- Inizializzare tutte le aree utilizzate per la gestione dei trap e degli interrupt (le aree NEW). Queste aree devono contenere stati del processore appropriati per attivare le funzioni specifiche del kernel per ogni tipo di trap e interrupt.
- Richiamare le funzioni di inizializzazione di phase1 per poter gestire le code dei processi, dei thread e dei messaggi.
- Creare lo stato per l'attivazione della SSI
- Creare lo stato del primo processo. Il program counter del primo processo deve essere posto all'indirizzo della funzione "test" che sarà il punto di inizio di esecuzione del p2test.
- Creare la coda ready inserendovi l'SSI e il thread del primo processo.

A questo punto si può attivare lo scheduler.

È bene notare che dopo aver chiamato lo scheduler, il controllo non tornerà più al programma principale. L'unico meccanismo per ritornare ad eseguire il codice del kernel è attraverso la gestione delle eccezioni. Fino

a quando ci saranno thread attivi da eseguire il processore eseguirà il loro codice e solo temporaneamente rientrerà nel nucleo per gestire le eccezioni.

2.2 Scheduler

Il nucleo deve garantire il *finite progess*: ogni processo ready deve avere la possibilità di proseguire la proria esecuzione. Per semplicità viene richiesta l'implementazione di uno scheduler di tipo round robin con time slice di 5 millisecondi. Si chiede che venga implementato un semplice meccanismo di verifica del deadlock. Quando tutti i thread sono in attesa di altri thread (e non attendono l'interrupt di un device) siamo in presenza di deadlock. (In questo calcolo non si tiene conto dell'SSI).

2.3 System Call

Il kernel di MiKABoO fornisce due sole system call: *msgsend* e *msgrecv*. I messaggi gestiti da queste system call sono numeri interi senza segno con ampiezza una parola di memoria (uintptr_t). Per scambiare dati più complessi i thread creeranno delle strutture (struct) e passeranno come messaggio il puntatore ad esse. Queste system call sono consentite solamente a thread in kernel mode. Nel caso vengano chiamate da processi in user mode si deve generare un errore di tipo "Istruzione Illegale". Le macro per richiamare le system call vengono fornite nel file *nucleus.h*.

2.3.1 msgsend

Msgsend è la system call numero 1.

```
int msgsend(struct tcb_t *dest, uintptr_t msg)
```

I parametri della msgsend potranno essere letti nell'area OLD relativa alle trap di tipo system call.

a0 contiene la costante 1 (msgsend)

a1 contiene l'indirizzo del thread destinatario

a2 contiene il messaggio

Il messaggio deve essere posto nella coda dei messaggi del thread di destinazione. Se il thread destinazione era in attesa di un messaggio (o da qualsiasi mittente o specificatamente dal processo che sta facendo la send), occorre risvegliare il processo, accodandolo alla ready queue. Il valore di ritorno (in a0) deve essere 0 se l'operazione ha avuto successo, un valore diverso da 0 (consigliato -1) se è avvenuto un errore.

2.3.2 msgrecv

Magrecv è la system call numero 2.

```
struct tcb_t *recv(struct tcb_t *src, uintptr_t *pmsg)
```

a0 contiene la costante 2 (msgrecv)

a1 contiene l'indirizzo del thread mittente. Il valore NULL indica che si vuole ricevere un messaggio senza selezionare il mittente.

a2 contiene il puntatore al buffer dove registrare il valore del messaggio. Se è NULL il messaggio non viene memorizzato (serve per messaggi di sincronizzazione/acknowledgement).

La system call numero 0 deve essere sempre considerata un errore. Tutte le system call di numero superiore a 2 devono essere trasformate in messaggi per il thread gestore delle system call (se definito con l'apposito servizio SETSYSMGR della SSI, vedi oltre), altrimenti trattate come messaggi al gestore delle trap di errore programma (se definito con il servizio SETPGMMGR), altrimenti causano la terminazione del thread chiamante.

2.3.3 System Service Interface (SSI)

La SSI è un thread di sistema che elabora le richieste dei processi. Si comporta come un demone di sistema, rimane sempre in attesa di messaggi contenenti richieste, li elabora e invia un messaggio di risposta. Il messaggio è un puntatore ad un intero contente il numero del servizio desiderato. Nelle parole di memoria successive sono memorizzati gli ulteriori parametri delle richieste. Nel file `nucleus.h` vengono fornite le funzioni che provvedono a preparare e spedire i messaggi opportuni per le richieste all'SSI e fornire il risultato al programma chiamante.

La SSI fornisce i propri servizi solamente a processi in Kernel mode, altrimenti deve generare un errore.

Qui di seguito viene riportata la tabella dei servizi da implementare.

GET_ERRNO (servizio 0). Nessun ha alcun ulteriore parametro, restituisce il codice di errore dell'ultima operazione fallita.

CREATE_PROCESS (servizio 1). Crea un nuovo processo ed un thread in esso. Ha come parametro lo stato iniziale del primo thread del processo da creare. Il nuovo processo deve risultare come "figlio" del processo che ha richiesto il servizio CREATE_PROCESS. Restituisce l'indirizzo del descrittore del thread appena creato (NULL il caso di errore).

CREATE_THREAD (servizio 2). Crea un nuovo thread del processo che richiede CREATE_THREAD. Ha come parametro lo stato iniziale del thread e restituisce l'indirizzo del descrittore del thread appena creato (NULL il caso di errore).

TERMINATE_PROCESS (servizio 3). Termina il processo corrente e tutti i processi nella progenie di quello corrente. Ovviamente tutti i thread dei processi terminati devono essere eliminati. Non ha parametri e non ha alcun valore di ritorno.

TERMINATE_THREAD (servizio 4). Termina il thread chiamante. Non ha parametri e non ha alcun valore di ritorno.

SETPGMMGR, SETTLBMGR, SETSYSMGR (servizi 5, 6, 7). Questi servizi servono per "nominare" i thread che dovranno operare come gestori rispettivamente degli errori di programma, degli errori di memoria e delle system call. Questi gestori vengono definiti a livello di processo ed a loro verranno riportate come messaggi tutte le eccezioni rilevate dal nucleo. Hanno come parametro l'indirizzo del decrittore del gestore e restituiscono lo stesso valore se l'operazione ha successo, NULL in caso contrario. I manager ricevono messaggi che hanno come sender il descrittore del thread che ha causato la trap ed un puntatore al suo stato come payload del messaggio.

GETCPUTIME (servizio 8). Questo servizio che non ha ulteriori parametri restituisce il numero di microsecondi di CPU utilizzati dal processo chiamante per la propria esecuzione.

WAIT_FOR_CLOCK (servizio 9). Questo servizio non ha parametri e non restituisce alcun valore ma sospende il processo richiedente fino al prossimo tick (100ms).

DO_IO (servizio 10). Questo servizio serve ai processi per fare operazioni di input-output. I parametri sono diversi a seconda del tipo di device utilizzato:

Disco: indirizzo del registro di device, valore da porre nel registro command e valore di DATA1 (cioè l'indirizzo di memoria usato per i trasferimenti in modalità DMA), mentre le altre informazioni di accesso, settore/cilindro o testina sono bit del campo command.

Nastro: indirizzo del registro di device, valore da porre nel registro command e valore di DATA1 (cioè l'indirizzo di memoria usato per i trasferimenti in modalità DMA)

Rete: indirizzo del registro di device, valore da porre nel registro command e valori di DATA1 e DATA2 (DATA1 è l'indirizzo del buffer, DATA2 è la lunghezza del pacchetto).

Stampante: indirizzo del registro di device, valore da porre nel registro command e valore di DATA1 (il carattere da stampare).

Terminale (canale di Input): indirizzo del registro di device e valore da porre nel registro command (dove viene resistrato il carattere letto).

Terminale (canale di output): indirizzo del registro di device e valore da porre nel registro command (dove è contenuto anche il carattere da visualizzare).

GET_PROCESSID (servizio 11). Questo servizio prende in input il puntatore al descrittore di un thread e restituisce il puntatore al processo al quale appartiene.

GET_MYTHREADID (servizio 12). Questo servizio non ha parametri e restituisce il descrittore del thread richiedente.

GET_PARENTPROCID (servizio 13). Questo servizio, dato come parametro il descrittore di un processo, restituisce il descrittore del processo genitore. (NULL se è il processo radice).

2.4 Alcuni dettagli

Molti sono gli aspetti progettuali che i gruppi devono affrontare. In questa sezione vengono presentati quelli che hanno creato maggiore discussione durante le lezioni di coordinamento sulle specifiche.

2.4.1 Pseudoclock

Esiste un solo interval timer. Quando un processo viene posto in stato running occorre caricare opportunamente l'interval timer per gestire sia il timeslice del processo sia la funzione dello pseudoclock.

Grande attenzione deve essere posta per non accumulare gli errori di temporizzazione. Non sarà possibile riattivare i processi che hanno chiesto il servizio WAITFORCLOCK esattamente allo scadere dei 100 millisecondi, è però importante che l'errore compiuto non postponga l'istante calcolato per il prossimo tick perchè altrimenti dopo molti cicli della WAITFORCLOCK l'errore potrebbe diventare rilevante.

Occorre anche tenere conto che per il ritardo indotto dalla gestione dell'interrupt e del salvataggio dei parametri un unico interrupt potrebbe causare la fine del timeslice e il tick dello pseudoclock.

2.4.2 SSI e scheduler

La SSI deve accedere a strutture dati che vengono utilizzate anche da altre parti del nucleo (e.g., code dei thread). Occorre porre attenzione per evitare race condition.

Allo stesso tempo il gruppo dovrà discutere come gestire la SSI nella politica di scheduling per ottimizzare le prestazioni.

2.4.3 Trap e riattivazione thread

Se una trap viene gestita, occorre riattivare il thread che l'ha causata.

Occorre fare attenzione che per trap di tipo TLB (memoria) o di programma, "gestire" la trap significa aver reso possibile l'istruzione cha ha causato l'errore, per esempio avendo caricato la pagina di memoria mancante a seguito di un errore di page fault. Queste trap devono ripetere quindi l'istruzione che causato la trap.

Al contrario le per le system call occorre continuare l'esecuzione dall'istruzione successiva (altrimenti il sistema operativo entrerebbe in loop).

2.4.4 Interrupt

Gli interrupt vengono trasformati in messaggi. L'interrupt deve riattivare il thread che ha richiesto l'operazione di I/O tramite il servizio DO_IO.

Il thread deve ricevere un messaggio particolare che ha l'SSI come mittente. Il gruppo deve discutere come implementare questa funzionalità, mantenendo il codice pulito, comprensibile, efficiente e senza inutili duplicazioni.

2.4.5 Deadlock e terminazione

Per verificare se vi è deadlock, viene richiesta una verifica minima: quando tutti i thread sono in attesa di un messaggio (che non sia per una operazione di I/O) si può concludere che il sistema sia in deadlock. In un sistema operativo microkernel questo potrebbe portare a fraintendere il caso di terminazione con il caso di deadlock. Quando tutti i thread hanno terminato la propria esecuzione rimane solamente la SSI in attesa di messsaggi. Occorre discutere come riconoscere la terminazione. I gruppi possono anche implementare politiche più elaborate di riconoscimento del deadlock e della terminazione.

2.5 Test di Phase2

Viene fornito un programma di test (p2test.c) e un file di testata (nucleus.h) contenente definizioni utili all'esecuzione del nucleo e del test.

Il punto di ingresso del test di phase2 è la funzione *test*.

Se il test non funziona, molto probabilemnte sono presenti errori nell' implementazione del nucleo. Se al contrario il test completa l'esecuzione nulla garantisce che il nucleo sia corretto.

Tutto ciò che non viene precisato nelle specifiche è oggetto di scelte progettuali che dovranno essere discusse, descritte nella documentazione allegata al progetto consegnato ed implementate nel codice realizzato da ogni gruppo.

I gruppi sono liberi di fornire *anche* altri programmi di test più completi anche per mostrare il funzionamento di tutte le funzionalità specifiche.

Il risultato del lavoro deve essere cosegnato sotto forma di **un unico file** nella directory:

```
/home/students/LABSO/2017/submit_phase2.${session}/lso17az${groupno}
```

Dove `session` è la sessione di scadenza `june, july o final` e `groupno` è il numero del gruppo.

Parametri fondamentali della valutazione saranno il grado di professionalità del codice, degli strumenti utilizzati, della documentazione e della struttura del progetto.