MIKABOO

Il progetto è composto da 6 file principali, sommati ai file di test che dovranno essere svolti.
Ogni file C ha il corrispondente header dove sono descritte le funzioni.
Il file const.h contiene tutte le variabili globali che verranno utilizzate nel programma e tutti i #define utili alla comprensione del codice.

INTRODUZIONE

Una breve panoramica dei file: boot.c si occupa della inizializzazione delle strutture basi(create in phase1 nel file mikabooq.c), delle aree di memoria dedicate, del SSI e del programma test.
Successivamente è il momento dello scheduler.c che si occupa di settare a current_thread un thread della ready queue o, se possibile, dare precedenza all SSI. Si occupa inoltre di controllare i casi di terminazione, e di settare e controllare i time slice e la pseudo_clock queue.
SSI.c contiene tutte le funzioni necessarie a gestire i servizi.
Funzioni che non si occupano di rispondere direttamente al servizio richiesto, sono la check_death che controlla e sveglia se,una volta ucciso un thread, ci sono altri thread in attesa di un messaggio da quello ucciso. La exterminate thread e process si occupano di terminare rispettivamente un thread e/o un processo, con delle particolari scelte di progettazione che descriveremo precisamente dopo.
Non sempre è l'ssi a dover rispondere al thread che ha invocato il servizio, può succedere che sia un handler a dovere mandare il messaggio di risposta o un terminale che mandi l'acknowledge.
In exception.c vi è la gestione degli handler: system, program e tlb. Si trovano al suo interno le funzioni per inviare e ricevere messaggi, per mettere un processo nella ready queue(svegliarlo) o per metterlo nella wait queue(addormentarlo).
Interrupt.c invece gestisce gli interrupt attraverso inthandler; al suo interno vi sono le chiamate alle macro per gestire correttamente i device e una funzione per gestire i time slice.

MIKABOOQ

Piccolo inciso che riguarda modifiche effettuate a phase1(mikabooq.c)per poter implementare tutti i servizi richiesti: i processi ora contengono tre puntatori ai propri manager: system, tlb e prg. Anche i thread possiedono quattro nuovi campi, un campo è l'error number, un altro è un puntatore ad un pcb in caso che il thread sia un manager mentre gli ultimi due riguardano i tempi:cpu time esprime il tempo per il quale il thread ha occupato la CPU, mentre elapsed time esprime il tempo di attesa da quando c'è stata una chiamata alla wait for clock.
Inoltre abbiamo creato la reset state che setta tutti i campi di uno state a 0, la thread_in_queue che controlla se un thread è presente in una determinata coda oppure no.

Ora è il momento di guardare in maniera dettagliata i singoli file di questa phase e in alcuni casi, in maniera dettagliata pure le singole funzioni.

BOOT

Partiamo nuovamente da boot.c
Il main verrà eseguito una sola volta, il controllo non tornerà mai al boot ma rimbalzerà tra gli altri file.
Vengono inizializzate i device, le strutture create da mikabooq e successivamente le 4 aree di memoria con i relativi handler.
Poi vi è la creazione dei processi SSI e test e i relativi thread: in particolare gli interrupt sono abilitati e hanno lo status di kernel mode.
Successivamente, prima di eseguire lo scheduler per la prima volta, vengono posti nella ready queue.
Seguendo l'ordine, il prossimo file è lo scheduler.c

SCHEDULER

La funzione principale si chiama scheduler() e ha due compiti principali: gestire i casi di deadlock/wait/halt della macchina, e caricare il processo corrente.
Ad ogni chiamata dello scheduler viene settato il prossimo timer e la funzione che si occupa di ciò è set_next timer: ci sono tre casi possibili che vengono gestiti da questa funzione insieme al supporto della set_pseudo_clock.

Caso uno) è finito il time slice e setto il time successivo; caso due) il primo processo della pseudo clock queue deve attendere un tempo minore di un time slice: setto il timer a il tempo rimasto dello pseudo clock; caso tre) il primo processo della pseudo clock queue deve attendere un tempo maggiore di un time slice: setto il primo timer al time slice.
Questo avviene perchè essendo una coda fifo che tiene conto dei tempi dei thread necessarriamente, scorrendola i temppi (elapsed time) aumentino; quindi se il primo è maggiore del time slice di conseguenza anche tutti gli altri lo saranno.
La funzione is_time_slice invece è solo di supporto(al timer handler)e controlla se è finito o meno il timeslice del thread ed è stata inserita in questo file poichè usa delle variabili strettamente locali.

SSI

SSI.c si occupa di gestire tutti i servizi che li richiedono e la sua funzione principale è SSI_main_task: controlla se è una richiesta accettabile e in caso affermativo, chiama la funzione corrispondente e inserisce in reply la risposta.
Ci sono due possibilità, che main_task restituisca True o False:
se è True: allora l' ssi deve rispondere al thread che ha richiesto il servizio;
se è False ci sono varie possibilità:
se era richiesto una do_io allora il messaggio di risposta proviene dal device_handler o terminal_handler;
se era richiesto un waitforclock allora il messaggio di risposta proviene dal timer_handler
se era richiesto il set di un manager il thread potrebbe essere stato ucciso e quindi non ha bisogna di ricevere una risposta;
se era stato richiesto un terminate allora il thread è stato ucciso e non deve ricevere una risposta.

In caso di uccisione di un thread viene inoltre controllato se il thread che deve essere ucciso è in realtà un manager invocando la funzione free_managing: se la risposta è affermativa setta a NULL il campo del pcb_t a cui è assegnato.

Nel caso di un' assegnazione di un manager viene salvato il tcb_t del manager nel campo dedicato del padre del thread che ha fatto la richiesta.
Una piccola scelta di progettazione è stata come avviene l'uccisione dei thread e dei processi. Quando un thread viene terminato ed è l'unico thread del processo, allora anche il processo viene terminato. Se un processo viene

terminato, allora anche tutti i suoi thread vengono uccisi. Se un processo è ucciso allora tutti i suoi processi figli vengono uccisi(questo perchè per specifiche di phase 1 non possono esistere processi con padre==NULL escluso il processo root).
Il numero di thread totali e di quelli softblockati viene controllato dalle funzioni create e exterminate aumentandoli o diminuendoli.
Nelle funzioni che eliminano processi/thread, vi è pure la chiamata ad una funzione che gestisce casi di errori:la check_death.
Essa si occupa di controllare se, una volta ucciso un thread, ci sono altri thread in attesa di una risposta da questo. Se la risposta è affermativa, setta il loro err_numb e li sveglia con un messaggio inviato dall'SSI.

Una funzione che neccessita una spiegazione dettagliata è la SSI_do_io:
dev_type e dev_numb sono utilizzati per calcolare il device su cui devo operare, tipo e numero e vengono ottenuti attraverso due macro descritte in const.h
Il dev_type è ottenuto in questo modo:
dev_type=(dev_reg_com-DEV_REG_START)/(DEV_PER_INT*DEV_REG_SIZE)
+DEV_IL_START;
poichè sottraggo DEV_REG_START per avere l' offset rispetto all' indirizzo iniziale nei device
DEV_PER_INT*DEV_REG_SIZE è la dimensione occupata da una tipologia di device, dividendo per DEV_PER_INT*DEV_REG_SIZE ottengo il tipo del device, ma mappato male quindi sommo DEV_IL_START per avere la stessa mappatura utilizzata da uARM
Per ottenere il dev_numb invece devo eseguire queste operazioni:
dev_numb=((dev_reg_com-DEV_REG_START-COMMAND_REG_OFFSET)%
(DEV_REG_SIZE*DEV_PER_INT))/DEV_FIELD_SIZE;
cioè: dev_reg_com-DEV_REG_START-COMMAND_REG_OFFSET è l' offset del inizio del device (non del campo command)
facendo il modulo della divisione per DEV_REG_SIZE*DEV_PER_INT ottengo l' offset nel tipo di device (esempio, Tape 4 o 5), misurato in word.
Ogni device occupa 2 word, anche se i terminali hanno una word dedicata all' input e una all' output, uso questa particolarità per riconoscere se è stata richiesta una lettura o una scrittura,
DEV_FIELD_SIZE vale 4, (4 byte in una word), ottengo (indice del device)*2, e (indice del device)*2+1 in caso di scrittura su device.
se (in caso di terminale) dev_numb%2==1 allora è scrittura e lo incoderò nella sua coda dedicata.

Utilizzando poi la macro ACTION_ON_DEVICE riempio i campi data0, data1 e infine command per compiere un operazione.
La macro presuppone che il parametro(addr) sia il messaggio ricevuto dalla DO_IO del nucleo.

EXCEPTION

In exception.c ci sono varie funzioni "di supporto":put_thread_sleep and wake_me_up sono una l'opposto dell'altra: mentre la prima posiziona un thread nella wait_queue e aumenta il numero di softblock la seconda sposta nella ready_queue e diminuisci il numero di thread softblockati.
Un'altra con un compito nato da una esigenza particolare è check_thread_alive. Mentre la check_death si occupa di svegliare thread che erano in attesa di un thread ucciso, la check_thread_alive controlla che non si voglia inviare ad un thread che non fa parte di quelli vivi: se non esiste, allora il messaggio non viene spedito e nel campo apposito err_numb viene settato l'errore.
Sys_send_msg si occupa di inviare un messaggio da un thread ad un altro, ma ha una particolarità in più:se il ricevente aspetta già un messaggio dal mittente, non viene incodato in messaggio ma viene direttamente svegliato il thread in attesa.
Ora è il momento di descrivere i tre handler:
tlb_handler (gestisce i tlb miss) e pgm_handler (gestisce le trap) vengono chiamati quando si verifica un tlb miss o una trap,
controllano se il causante ha definito un manager e in caso affermativo, mettono nella wait queue il thread corrente, svegliando il manager.
Il thread è risvegliabile solo dal manager, ma non ha fatto una receive perciò sarà il manager a occuparsi di risolvere la trap o caricare la pagina richiesta.

Il terzo handler, sys_bp_handler, è la funzione cuore del file: essa viene invocata quando un thread richiede un servizio o una system call(send recv).
E' necessario quindi controllare i permessi del thread: se è in user mode e richiede una system call, deve essere invocato il suo system manager; se è provocata una trap, il program manager; se è in kernel mode invece, può mandare direttamente send e recv.
La send prima di tutto controlla che non si stia inviando ad un morto, successivamente controlla se il messaggio è la risposta di un manager Uper risvegliare il thread) e altrimenti si manda la send con la funzione apposita e si avanza di una word size per evitare di rientrare nel codice della syscall.

Anche la recv controlla che si stia ricevendo da un processo esistente e dopo aver fatto la msgq_get controlla se è disponibile un messaggio dal mittente richiesto. Se c'è un messaggio disponibile va messo nella wait queue, settati i campi e non va avanzato di una wordsize; se invece la msgq_get va a buon fine, carico i vari campi in maniera adeguata e faccio il load del processo.
Tutti i casi di errore di questa sezione vengono risolti con l'uccisione del thread.

INTERRUPT

L'ultimo file che è rimasto da descrivere è interrupts.c e come enuncia il nome, si occupa di gestire tutti gli interrupt che arrivano da parte dei device virtuali di uArm. La funzione principale è int_handler la quale controlla la causa dell'interrupt e aggiorna il tempo passato dal thread nella cpu; a seconda del tipo di interrupt ricevuto chiama la funzione adeguata per poi richiamare lo scheduler e far continuare l'esecuzione del programma.
Tutti gli interrupt vengono gestiti dal device_handler, escludendo la causa TIMER che viene gestita dal timer_handler, essa viene causata quando finisce un time slice o uno pseudo clock, se è finito il time slice cambio il thread in esecuzione.
Per la gestione dei device, è necessario leggere lo status dal registro del device che ha causato l' interrupt, rispondergli con un ack, e rispondo al thread richiedente con lo status dell' operazione.
Il messaggio viene mandato da parte dell' ssi.
Per individuare il device che ha causato l' interrupt uso CAUSE_IP_GET per il tipo di device, e CDEV_BITMAP_ADDR per l' offset del device.
Device con offset minore hanno priorità maggiore, ottengo l' offset del device scorrendo una maschera di bit.
Per individuare il thread a cui rispondere abbiamo preparato una coda per ogni device, prendo il primo thread dalla coda giusta.
Il thread è poi da riportare nella giusta coda di appartenenza (wait o ready), poteva essere in ready queue se il suo timeslice è finito subito dopo la send e il device ha risposto prima che il thread facesse la receive; altrimenti era nella wait queue.
I terminali hanno una gestione leggermente differente e per questa ragione hanno la loro funzione specifica: oltre ai controlli base dei device vi è pure il controllo se la richiesta è di scrittura o di lettura.

P2P

Inoltre abbiamo portato un ulteriore test:p2p.c
Esso ha le stesse funzioni del test base solamente che ogni funzione è abbinata ad un terminale e in questo modo possono tutti lavorare in maniera parallela e testare anche tutti i terminali.