FASE 2 ~ RELAZIONE

Scopo della seconda fase del progetto RIKAYA Operating System è quello di completare le funzionalità abbozzate nella fase 1.5, completando quest'ultime in modo da ultimare il livello del kernel. La struttura del kernel realizzata si basa quindi su moduli precedentemente programmati e che sono stati modifcati in modo da gestire nel modo corretto i servizi che il kernel deve fornire, secondo le specifiche della fase 2. In particolare il nucleo deve gestire:

- Inizializzazione del sistema
- scheduling dei processi
- gestione delle syscall
- gestione degli interrupt

Nella descrizione del codice scritto si procederà esponendo in primo luogo le scelte fatte in fase di inizializzazione e successivamente si discuterà delle modifiche fatte allo scheduler rispetto a quello della fase 1.5. In secondo luogo si passeranno in rassegna le varie syscall e gli interrupt per esporre le motivazioni che hanno portato alla stesura del codice.

N.B. nella relazione le parole in *italic* indicano che si sta parlando di una variabile, la parole in UPPERCASE sono costanti, mentre per le funzioni si usa il **bold**.

Entry point(file main.c)

L'esecuzione del nucleo parte dal file main.c, in cui oltre all' inizializzazione della fase 1.5, vengono aggiunte l'inizializzazione della Active Semaphore List (**initASL**), delle strutture dati dedicate ai semafori (**initSemDevices**) e del contatore dei processi bloccati su semafori (*ProcBlocked*). Il root process, data la sua importanza, viene subito allocato e settato, visto che il suo program counter contiene l'indirizzo di memoria della funzione di test e che deve essere il processo tutor di default (nel caso un processo divenisse orfano e non ci fossero tutor).

La funzione **initSemDevices** serve per allocare i semafori. Dato che ogni semaforo ha una chiave di tipo intero, è necessario allocare un numero di interi pari al numero di semafori utilizzati. Per fare ciò abbiamo deciso di utilizzare un array di interi con dimensione pari a MAX_DEVICES (l'ultimo elemento, di indice CLOCK_SEM, è quello dedicato alla system call WAITCLOCK). Tutti gli elementi dell'array di interi *semDev* sono inizializzati a 0. Successivamente si usa la struttura dati *semDevices*, facendo puntare la chiave di ogni semaforo (per ogni istanza di ogni tipo di device) ad un intero differente all'interno di

semDev. Completata l'inizializzazione dei semafori sarà possibile eseguire le syscall passeren e verhogen.

É importante notare che l'allocazione fatta in questo modo non viola il fatto che ci sia un numero limitato si semd_t. Infatti initSemDevices non sta allocando descrittori di semafori (che sono in numero limitato), ma interi a cui puntano le chiavi dei semafori.

Una volta inizializzati i semafori e aggiunto il pcb del root process alla ready queue si usa la funzione SET_IT che fa partire l'interval timer da 100 ms. SET_IT viene messa nel main perché l'IT deve generare un interrupt sulla seconda linea ogni 100 ms, perciò viene usata per la prima volta prima di iniziare l'esecuzione vera e propria che partirà richiamando lo scheduler.

Scheduler (scheduler.c e relativo header)

Rispetto alla fase precedente lo scheduler deve tenere traccia del tempo di esecuzione del processo in user mode, kernel mode oltre che del tempo totale dalla prima attivazione del processo. Inoltre lo scheduler deve gestire casi in cui la ready queue può trovarsi vuota a causa del fatto che potrebbero esserci processi in attesa di I/O.

Per quanto riguarda la gestione del tempo abbiamo deciso di utilizzare due variabili (una per il tempo 'old' e una per il tempo 'new') per segnare il tempo passato in user mode, due variabili per gestire quello in kernel mode e una per gestire il wallclock time. Queste cinque variabili vengono memorizzate in campi all'interno del pcb. Il meccanismo utilizzato per aggiornare le varibili è il seguente:

- si controlla se il tempo new è positivo: se ciò accade significa che si deve aggiornare il tempo totale passato in una modalità, ossia il tempo old che verrà quindi aggiornato sommandoci la differenza del valore del registro TODLOW e del valore di time new

Esempio: si sta eseguendo codice utente e si solleva un'eccezione di tipo SYSCALL. Si controlla *user_time_new* con il meccanismo appena descritto e se positivo si fa *user_time_old* += getTODLO() - *user_time_new*. Successivamente si mette il valore di *user_time_new* a 0 e si legge il valore del registro TODLOW assegnandolo a *kernel_time_new*. Una volta finito il tempo in kernel mode si ritorna in user mode gestendo il tempo come appena descritto.

I momenti in cui fare time management, sono questi:

- prima di fare LDST di un processo si fa partire il tempo utente, dato che una volta che un processo viene bloccato perché finisce il suo quanto, quando riprende l'esecuzione sarà di sicuro in user mode, dato che durante la gestione delle eccezioni gli interrupt sono disabilitati.

- Quando un processo sta eseguendo il codice utente e si richiede l'utilizzo dei servizi offerti dal sistema operativo con una system call, si passa da user mode a kernel mode
- Quando una syscall termina, alla fine del case dello switch delle eccezioni che gestisce le syscalls, si passa da kernel mode a user mode(da notare che quando si fa una P, si ferma il tempo passato in kernel mode dato che il processo attende su un semaforo e viene richiamato lo scheduler, dunque non si fa ripartire il tempo utente perché il processo non sta eseguendo né codice utente né codice di sistema)
- Quando un interrupt si conclude e viene mandato l'acknowledgement, il controllo ritorna all'interno dell'handler delle eccezioni dove si fa ripartire il tempo utente
- Quando si solleva un'eccezione generica il tempo utente finisce e va quindi aggiornato

A differenza della fase 1.5, in cui lo scheduler controlla solo se la ready queue è vuota(in tal caso si fa HALT), lo scheduler della fase2 controlla come prima cosa se ci sono processi all'interno della ready queue. Se ci sono processi si richiama la funzione context, che fa context switch schedulando un nuovo processo presente nella coda dei processi ready, altrimenti controlla che il processo corrente non sia vuoto: se il *curr_proc* è NULL si richiama context, altrimenti si controlla se il contatore di processi bloccati è a zero. Se *ProcBlocked* è 0 allora non ci sono più processi de eseguire e si richiama HALT, altrimenti si usa la funzione WAIT, facendo partire il timer del processore (perché magari allo scadere dei 3ms la ready queue può essere non vuota) e abilitando gli interrupt (altrimenti non succede nulla, dato che, non essendoci processi da eseguire, l'unica eccezione che può verificarsi è l'interrupt di un processo bloccato su un semaforo di un device o della waitclock).

Handler exceptions (handler.c e relativo header)

Ogni volta che si solleva un interrupt, una system call o una trap, il nucleo passa l'esecuzione alla funzione Handler, il cui indirizzo in memoria è caricato all'interno delle 4 areas del ROM reserved frame. Dunque, come descritto nel manuale, possono verificarsi quindici diversi tipi di eccezione. Per gestire il tutto in modo più efficiente si è deciso di sfruttare uno switch utilizzando come espressione da valutare l'exception code, un campo di 5 bit all'interno del registro Cause (ottenuto con la funzione getCAUSE). L'utilizzo di uno switch per gestire tutte le eccezioni consente si semplificare il time management, oltre che fare un controllo sull'accesso alla funzione Handler in user mode (se ciò accade si genera una program trap). Di conseguenza dato che non si sta più

eseguendo codice utente, quest'ultimo viene aggiornato e fermato. Per far partire il kernel time, invece si aspetta, in quanto gli interrupt, per semplicità, non vengono considerati come tempo kernel (sicuramente gli interrupt sollevati dal processor local timer e dall'interval timer non sono inputabili a processi). Il kernel time viene fatto partire solo all'inizio del case delle syscall.

L'utilizzo di una funzione come Handler per gestire tutte le eccezioni consente anche di controllare che l'esecuzione quando il processore è in stato waiting sia corretta: infatti se il processo corrente è NULL e si solleva un' eccezione richiamando Handler, tale exception altro non può essere che un interrupt e di conseguenza il *cause_code* (intero su cui fare lo switch), va impostato su EXC_INTERRUPT (inoltre una volta che la funzione Interrupt_Handler è terminata, se il *curr_proc* è NULL allora bisogna richiamare lo scheduler: ciò avviene per esempio quando c'è un processo bloccato a fare I/O su un terminale e quest'ultimo conclude, dunque se il processore era in stato di wait il processo risvegliato con l'interrupt dovrà essere schedulato). Una volta ottenuto il *cause_code*, il controllo passa ad uno dei case e quindi si richiamano le funzioni che gestiscono le varie eccezioni: interrupt (Interrupt_Handler), syscall (ulteriore switch all'interno del case per capire quale funzione richiamare) o trap (PGMtrap, TLBtrap, SYSBPtrap).

Da notare è lo switch all'iterno del case delle system calls (*cause_code* = 8), in cui dopo aver fatto time management si incrementa il program counter del processo corrente e si identifica la syscall richiamata usando come espressione per lo switch il valore del registro *curr_proc→p_s.rega0*. Si utilizza la variabile *result* per memorizzare il valore di ritorno di alcune syscalls dato che tra queste sono presenti sia procedure che funzioni. Tra tutte le system calls, è interessante notare il case della Do_IO: l'assegnamento *wakeup_proc = curr_proc* fatto prima della chiamata è fondamentale, altrimenti si farebbe una dangling reference alla prima chiamata di Do_IO, dato che al suo interno il valore di *wakeup_proc* serve per capire se il processo che richiede Do_IO sia il processo corrente o una risveglaito da un interrupt.

Se si riceve un valore di syscall superiore a 10, si cerca un gestore di tipo syscall/breakpoint, ma se questo non è stato specificato attraverso una specpassup, il processo viene terminato.

Tutte le altre eccezioni che si possono verificare oltre a interrupt e syscall sono trap. In base all'exception code di cause si richiama il gestore per le program trap, le tlb trap e le sysbp trap.

Se il valore dell'exception code non è tra 0 e 14, il processo viene eliminato.

Una volta determinata l'eccezione viene dunque eseguito il gestore della relativa eccezione, si inizi ad esaminare il gestore degli interrupt.

Interrupts (interrupts.c e relativo header)

Per gestire gli interrupt abbiamo considerato i seguenti punti:

1. Capire quale linea ha generato l'interrupt
2. La funzione che gestisce gli interrupt deve essere strutturata io modo da consentire che interrupt a priorità più elevata ricevano l'ack prima di interrupt a priorità più bassa
3. Se ci sono più istanze per tipo di device come nelle linee di interrupt dalla 3 alla 7, è necessario capire quale o quali device hanno un interrupt pendente
4. Se ci sono più interrupt attivi insieme, allora si restituisce l'ack a tutti gli interrupt pendenti

Le scelte fatte per fronteggiare queste richieste sono le seguenti:

1. Per capire quale sia la linea che ha generato un interrupt abbiamo utilizzato la macro definita nel file const_rikaya.h CAUSE_IP_GET(*cause*, *int_no*), che ritorna uno se la linea di interrupt line_no ha generato un interrupt, facendo delle operazioni bit a bit con il registro cause della INT_OLDAREA. Ciò suggerisce di fare degli if, uno per ogni linea di interrupt.
2. Per dare una gerarchia agli interrupt abbiamo fatto il seguente ragionamento, che si eplicita meglio con un esempio: l'interrupt del PLT ha priorità maggiore dell'interrupt del terminale, perciò basta mettere il controllo con CAUSE_IP_GET(*cause*, INT_T_SLICE) prima di CAUSE_IP_GET(*cause*, INT_TERMINAL) nel corpo di Interrupt_Handler
3. Per risolvere questo problema abbiamo sfruttato un'altra macro presente nel file const_rikaya, INTR_CURRENT_BITMAP(*line_no*), che ritorna l'indirizzo in memoria della linea di interrupt line_no nella Interrupting Devices Bit Map. Dereferenziando tale puntatore si ottiene un numero che considerato in binario consente di capire se un device i di tipo j ha un interrupt pendente. Dato che tutti gli interrupt vanno soddisfatti tutti una volta sollevata un'eccezione di tipo interrupt, si controlla se ogni istanza di ongni tipo di device ha un interrupt pendente. Dunque una volta scoperto che la linea *line_no* ha un interrupt pendente viene naturale analizzare gli 8 bit della sequenza di bit restituita da INTR_CURRENT_BITMAP(*line_no*) con un for, dato che più di DEV_PER_INT devices non ci possono essere per ogni tipo di device.
4. Per dare l'ack a più interrupt pendenti contemporaneamente non si possono usare degli if-then-else statement od egli switch, altrimenti si da l'ack ad un solo interrupt. Si fa degli if-then statement a cascata per ogni linea di interrupt.

In generale per gestire un interrupt di un dispositivo di I/O si controlla se il device ha uno stato diverso da DEV_S_READY, in tal caso, dato che si è sollevato l'interrupt, il device ha finito l'operazione di I/O e sta attendendo di iniziare una nuova operazione o di ricevere l'ack. Dunque si risveglia un processo dalla coda dei processi bloccati sul semaforo del device: l'azione richiesta è quella di una Verhogen, ma quest'ultima non ritorna il pcb, necessario per identificare il processo appena risvegliato. Si è deciso di utilizzare le istruzioni che compongono la nostra Verhogen in modo da adattarle al caso specifico degli interrupt, quindi è necessario salvare all'interno di *wakeup_proc→p_s.regv0* il valore dello stato del device al momento del risveglio, operazione non possibile con la normale Verhogen.

Infine si noti che quando si solleva un interrupt, non ci possono essere altri interrupt o eccezioni che ne bloccano l'esecuzione, dato che si entra nella procedura Interrupt_Handler con gli interrupt disabilitati.

Altra nota importante è il meccanismo di ack dei termianli: anche qui, con un ragionamento simile fatto al punto 4, per dare l'ack sia in ricezione che trasmissione si fanno due if-then statement uno di seguito all'altra, controllando che il terminale stia effettivamente aspettando l'ack all'interrupt (lo stato del device è diverso da DEV_S_READY) e che ci siano processi bloccati (utilizzando il valore della chiave del semaforo).

Syscalls (syscalls.c e relativo header)

Nel file syscalls.c sono definite tutte le funzioni e procedure che impletano le dieci possibili diverse system calls. Il registro *curr_proc→p_s.rega0* identifica quale sycall è stata chiamata, mentre il registro *curr_proc→p_s.rega1* contiene il primo parametro da passare alla syscall, in *curr_proc→p_s.rega2* il secondo e *curr_proc→p_s.rega3* il terzo. DI seguito vengono descritte alcune scelte fatte durante l'implementazione delle syscall:

- Get_CPU_Time: il meccanismo di time management è già stato descritto. Nota per questa syscall sono le verifiche che i puntatori user, kernel e wallclock (i parametri formali) non siano NULL, altrimenti si rischia di fare dangling reference.
- Terminate_Process: sia che il processo de eliminare sia il *curr_proc* o un suo discendente ci sono delle istruzioni comuni da eseguire in entrambi i casi. In entrambi si cerca un tutor per la progenie del processo da eliminare, si elimina il processo dalla lista dei figli del padre e si ritorna il pcb alla lista dei pcb liberi. Invece se il processo da elimnare è il *curr_proc* allora in più bisogna anche richiamare lo scheduler e porre *curr_proc* uguale a NULL(altrimenti, visto com'è costruito il nostro scheduler, il curr_proc potrebbe essere reinserito nella ready queue). Quando il figlio da eliminare invece è un figlio del *curr_proc* allora è

necessario anche rimuovere il processo da eventuali code di semafori su cui è bloccato o dalla ready queue.

- Verhogen: importante è porre il campo p_semkey del pcb risvegliato a NULL, altrimenti non si capisce se tale pcb è bloccato o no su un semaforo.
- Passeren: in modo speculare alla Verhogen è importante porre il campo p_semkey uguale alla chiave del semaforo su cui il processo si blocca. Time management già discusso.
- Do_IO: per capire quale sia il device su cui il processo deve fare I/O, abbiamo deciso di fare delle operazioni tra indirizzi come confronti e sottrazioni, considerando l'indirizzo a cui parte la memorizzazione del registro del disco 0, l'indirizzo a cui parte la memorizzazione del registro del terminale 0 e dell'ultimo terminale. In più si sfrutta la macro definita in const_rikaya.h DEV_ADDRESS(*LINENO*, *DEVNO*) per calcolare gli indirizzi da cui parte la memorizzazione dei vari tipi di device. Ovviamente il *DEVNO* deve essere pari a 0. Si considerano anche le grandezze dei vari registri nel calcolo di *offset*, per capire su quale semaforo bloccarsi. Nota importante è il fatto che la Do_IO blocca a prescindere un processo per ciò per far ritornare lo status in modo corretto, si salva il valore di status quando il processo verrà risvegliato nell'interrupt (come descritto prima, il campo regv0 contiene lo status del device).
- Spec_Passup: per implementare questa syscall e il meccanismo di gestione delle trap, l'idea più semplice che ci è venuta è stata quella di creare una coppia di puntatori, un puntatore per il vecchio gestore di trap e un puntatore per il nuovo gestore di trap, per ognuna delle tre possibili trap. Queste coppie di puntatori sono poi salvate nel pcb del processo che richiama la Spec_Passup.

Non sono state inserite tutte le syscalls perché la maggior parte di esse sono già state commentate a dovere nel codice.

Traps (traps.c e relativo header)

Quando si solleva un'eccezione e non si tratta di un interrupt o syscall, ci si trova nel caso trap. In base all'exception code su cui si fa lo switch in Handler, si può dover gestire una program trap, una tlb trap o una sysbp trap. Si esponga il metodo di gestione della trap utilizzando come esempio il tentativo di accedere in memoria ad una locazione non esistente: il *cause_code* di Handler assume il valore EXC_BUSINVFETCH. Il case dedicato a EXC_BUSINVFETCH contiene la funzione PGMtrap (definita nel file utils.c) che copia lo stato della PGMTRAP_OLDAREA all'interno del *curr_proc*, si incrementa

il program counter e si richiama il PgmTrapHandler presente nel file traps.c. PgmTrapHandler verifica che sia specificato con la Spec_Passup un gestore per la program trap e se non è presente termina il processo corrente.

Fonti da cui si è preso spunto in alcuni passi del progetto:

-https://github.com/anaptfox/kaya

-https://github.com/thepankydoodler/kaya

Componenti del gruppo:

Gaspari Michele, Mollica Francesco, Faieta Stefano, Cotugno Giosuè.