Progetto μMPS3, fase 2

Ahmed Ayad
ahmed.ayad@studio.unibo.it

Andrea Zanella
andrea.zanella7@studio.unibo.it

Gabriele Bernardi
gabriele.bernardi5@studio.unibo.it

Lorenzo Giarrusso
lorenzo.giarrusso@studio.unibo.it

April 27, 2024

Contents

1 Modifiche alle strutture dati della fase 1

L'unica modifica alle strutture dati create per la fase 1 riguarda l'aggiunta del campo `int dev_n` alla struttura `pcb_t`. Questo campo viene usato per rappresentare il numero del dispositivo su cui il PCB si è bloccato in attesa di un'operazione di I/O.

2 main.c

Il main segue semplicemente quanto richiesto dalle specifiche, dividendosi in 3 fasi:

1. Inizializzazione di strutture dati e variabili.
2. Istanziazione dei PCB per l'SSI e per il processo di test.
3. Chiamata dello scheduler

Le uniche scelte implementative visibili in questo file e degne di spiegazione sono le seguenti:

- I PCB bloccati sui dispositivi sono gestiti tramite più code, una per ogni tipo di dispositivo, con in particolare una gestione separata dei terminali per trasmissione e ricezione. Un'alternativa a questa scelta sarebbe stata l'idea di utilizzare una singola coda generale per tutti i processi bloccati. Sebbene inizialmente avessimo optato per la seconda alternativa, questa risultava richiedere controlli e operazioni onerose per lo sblocco dei PCB tale da garantire la fairness.
- I PCB bloccati in attesa di un messaggio non hanno una propria coda dedicata, ma sono distinti "per esclusione": se un PCB esiste ma non è nella ready queue, allora è necessariamente bloccato, dunque lo sblocco. Se era bloccato non in attesa di quel messaggio, semplicemente si bloccherà di nuovo.
- Poiché altri moduli avranno bisogno di poter accedere ad alcune variabili qui definite, ad esempio al puntatore `current_process` ed ai contatori di processi esistenti e di quelli bloccati (`process_count` e `softblock_count`), tutte queste variabili sono dichiarate come globali. Sebbene questo sia poco elegante, l'abbiamo ritenuto preferibile per evitare di avere funzioni con grandi quantità di parametri.

3 scheduler.c

L'implementazione dello scheduler segue quanto descritto nelle specifiche:

- Se c'è qualche processo nella ready queue, questo viene estratto dalla coda, si assegnano i valori appositi al PLT e alla variabile `startTime` e lo si mette in esecuzione caricando il suo stato.
- Altrimenti, se l'unico processo attualmente esistente è l'SSI, il sistema va in HALT.

- Altrimenti, se esistono anche altri processi ma sono tutti attualmente in attesa, il sistema va in WAIT per attendere le interrupt.
- Altrimenti, se esistono anche altri processi ma risulta che nessuno sia in attesa, si ha una situazione di deadlock, dunque il sistema va in PANIC.

4 exceptions.c

Oltre alle tre funzioni principali previste dalle specifiche (`passUpOrDie`, `syscallHandler` e `exceptionHandler`), sono qui definite alcune funzioni di servizio:

- `update_time`: funzione che aggiorna il campo `p_time` di un PCB in modo che esso contenga il tempo di CPU occupato dal PCB a partire dall'ultima volta in cui è stato messo in esecuzione.
- `copyProcessorState`: funzione che semplicemente copia un valore di tipo `state_t` da una locazione ad un'altra; poiché questa è un'operazione frequente, `copyProcessorState` serve principalmente ad evitare di riscrivere ogni volta il codice.
- `die`: funziona che si occupa del ramo "die" del `passUpOrDie`, inviando all'SSI la richiesta di terminare il processo attuale e chiamando lo scheduler. `send_msg`: funzione per creare un messaggio proveniente da un dato mittente ed inserirlo nell'inbox del destinatario. Gestisce il caso in cui i messaggi (disponibili in numero finito) siano già tutti usati ritornando il valore `MSGNOGOOD`; se invece l'invio ha successo, restituisce 0.

Come da specifiche, la funzione `passUpOrDie` viene usata dall'exception handler per passare al livello di supporto la gestione di tutti gli eventi non gestibili da interrupt handler e syscall handler. In particolare, se il campo `p_supportStruct` del processo attuale è NULL si ricade nel ramo "die", altrimenti avviene l'effettivo pass up delle informazioni di supporto trovate in `p_supportStruct`.

La funzione `syscallHandler` viene chiamata dall'exception handler quando riceve una richiesta SYS1 (syscall per inviare un messaggio) o SYS2 (syscall per ricevere un messaggio). Le due system call possono essere soddisfatte solo se il PCB che le ha richieste sta eseguendo in kernel mode, dunque si controlla il corrispondente bit nel registro di controllo Status (acceduto tramite `getSTATUS`, isolando il bit Kernel/User tramite un AND bitwise con la costante `USERPON`, e facendo lo shift di 3 posizioni). Se il bit Kernel/User vale 1 allora il processo stava eseguendo in user mode, dunque la richiesta non è lecita, e si lancia un'apposita eccezione di codice `GENERALEXCEPT` specificando la causa dell'eccezione tramite la costante `PRIVINSTR`. Se invece il bit Kernel/User è 0, allora si procede. La distinzione fra una richiesta SYS1 e una richiesta SYS2 viene effettuata basandosi sul valore del registro a0 nello stato del processore al momento del lancio dell'eccezione (`exc_state->reg_a0`).

Se il registro a0 contiene il valore della costante `SENDMESSAGE`, allora si deve tentare di effettuare l'invio di un messaggio. In particolare, il destinatario è specificato in `reg_a1`, il payload in `reg_a2`, mentre dobbiamo salvare in `reg_v0` un valore rappresentante l'esito dell'invio (`DEST_NOT_EXIST` se il destinatario è un PCB attualmente non allocato, 0 se l'invio è stato completato, `MSGNOGOOD` altrimenti). La logica è la seguente:

1. Controlla se la destinazione esiste (ovvero se il PCB destinatario *non* è nella coda dei PCB liberi `pcbFree_h`) e se è nella ready queue;

2. Se il PCB destinatario non esiste, copia il valore della costante `DEST_NOT_EXIST` in `reg_v0`, senza fare altro.

3. Altrimenti, se il PCB destinatario è nella ready queue (e dunque non ha bisogno di essere sbloccato) o se il messaggio è diretto al mittente stesso (in qual caso sicuramente non è bloccato), il messaggio viene semplicemente inserito nell'inbox del PCB destinatario tramite la funzione `send_msg` ed il suo valore di ritorno viene assegnato a `reg_v0`.

4. Altrimenti, il PCB destinatario è sicuramente bloccato; dobbiamo dunque inserire il messaggio nel suo inbox tramite `send_msg`, assegnare il valore di ritorno a `reg_v0`, e sbloccare il PCB (re-inserendolo nella ready queue).

Se il registro a0 contiene il valore della costante `RECEIVEMESSAGE`, il chiamante sta richiedendo una ricezione di messaggio. In particolare, il mittente desiderato è specificato in `reg_a1` (se il valore è `ANYMESSAGE`, va bene qualsiasi mittente), la locazione in cui eventualmente salvare il payload è indicata in `reg_a2`, mentre il `reg_v0` del chiamante deve alla fine contenere l'identificatore del mittente del messaggio ricevuto, come da specifica. La logica è la seguente:

1. Cerchiamo nell'inbox del chiamante un messaggio proveniente dal mittente desiderato.

2. Se non esiste un messaggio soddisfacente, il chiamante deve venire bloccato; dunque si salva il suo stato, ne si aggiorna il campo `p_time` e si chiama lo scheduler.

3. Altrimenti, il messaggio soddisfacente trovato viene ricevuto senza bloccare il chiamante; ovvero, l'identificatore del mittente viene memorizzato nel `reg_v0` del chiamante, il payload viene eventualmente copiato nella locazione specificata in `reg_a2`, il messaggio viene re-inserito nella lista di messaggi liberi e l'esecuzione procede normalmente (dopo aver incrementato il Program Counter).

La funzione `exceptionHandler` si occupa di gestire tutte le exceptions ricevute, semplicemente controllandone l'exception code per stabilire quale funzione chiamare per gestirle. Innanzitutto, la funzione salva le informazioni sull'exception state nel puntatore `state_ t* exc_state`, che verrà poi utilizzato in ogni caso per la gestione. Come spiegato nelle specifiche, `BIOSDATAPAGE` contiene le informazioni sull'exception state, mentre la causa dell'eccezione è codificata nel campo `.exc_code` del registro Cause nell'exception state. Dunque, l'exception code viene estratto con le seguenti righe di codice:

```
int cause = getCAUSE(); //Valore del registro Cause
unsigned int exc_code = cause & GETEXECCODE; //Estrai exception code
exc_code = exc_code >> CAUSESHIFT; //Shift per avere valori comodi
```

In base al valore dell'`exc_code` possiamo semplicemente stabilire a chi inoltrare l'eccezione:

1. Se `exc_code == IOINTERRUPTS` (costante pari a 0), l'eccezione è un'interrupt, dunque si chiama l'`interruptHandler`.
2. Se l'`exc_code` ha valore fra 1 e 3 si tratta di una Page Fault Exception, dunque si chiama `passUpOrDie` specificando che il tipo di eccezione è `PGFAULTEXCEPT`.
3. Se l'`exc_code` ha valore fra 4 e 7 oppure fra 9 e 12 si tratta di una Program Trap, dunque si chiama `passUpOrDie` specificando che il tipo di eccezione è `GENERALEXCEPT`.
4. Se `exc_code == SYSEXCEPTION` (costante pari a 8), si chiama `syscallHandler`.
5. Se nessuna delle condizioni precedenti è soddisfatta, il valore dell'`exc_code` non può essere gestito, dunque il sistema va in PANIC.

5 ssi.c

In questo modulo si definiscono due funzioni di servizio per il bloccaggio dei PCB:

- `blockPCB` si occupa di bloccare il PCB sulla linea specificata inserendolo nella apposita coda di attesa. Nel caso dei terminali, si usa il parametro `term` per sapere se si tratta di una trasmissione o una ricezione.
- `blockPCBfromAddr`, dato un indirizzo nel parametro `addr`, calcola il dispositivo corrispondente e ci blocca il PCB specificato chiamando `blockPCB`.

La funzione `ssi` implementa il funzionamento dell'SSI come descritto nelle specifiche: all'interno di un ciclo infinito, l'SSI attende di ricevere una richiesta, tenta di soddisfarla ed eventualmente comunica il risultato al richiedente. In particolare, le richieste vengono processate tramite la funzione `ssi_request`, che controlla is `service_code` nel payload del messaggio di richiesta per chiamare la funzione apposita per quel tipo di richiesta. I servizi offerti sono quelli definiti nella sezione 7 delle specifiche:

1. `ssi_create_process`: crea un processo come figlio del richiedente.
2. `ssi_terminate_process`: termina un processo e, ricorsivamente, tutto il sotto-albero di cui è radice. Inoltre, rimuove il PCB in questione dalla coda di attesa in cui eventualmente si trovava.
3. `ssi_DoIO`: semplicemente blocca il richiedente sulla coda del dispositivo corrispondente. μMPS3 supporta infatti solo I/O sincrono, come da specifiche.
4. Il servizio `GetCPUTime` è implementato semplicemente restituendo il campo `p_time` del chiamante.
5. `ssi_waitForIT`: mette il chiamante in attesa del prossimo tick dell'Interval Timer inserendolo nell'apposita coda.
6. Il servizio `GetSupportData` è implementato semplicemente restituendo il campo `p_supportStruct` del chiamante.

7. `ssi_getPID`: per ottenere il PID del chiamante oppure del suo PCB padre, come da specifiche.

6 interrupts.c

In questo modulo si definiscono due funzioni di servizio:

- `unblockPcbByDevNum`: semplicemente sblocca il PCB bloccato sul dispositivo indicato dal numero di dispositivo dato tramite il parametro `devnum`.
- `getHighestPriorityDevNo`: data come parametro la bitmap dei dispositivi su una certa linea, restituisce il numero del dispositivo con massima priorità fra quelli che attualmente hanno un'interrupt attiva.

La funzione `interruptHandler` si occupa di processare gli interrupt ricevuti smistandoli verso le apposite funzioni in base alla provenienza dell'interrupt. Per sapere quali linee hanno attualmente un'interrupt attiva, basta controllare il contenuto del campo IP del registro Cause e farne un AND bitwise con le apposite costanti per ogni Interrupt Line (`IL_CPUTIMER`, `IL_TIMER`, ecc). Si noti che il controllo delle Interrupt Line viene fatto partendo dalla 0 fino alla 7 per garantire la priorità come definita nelle specifiche (se più linee hanno un'interrupt attiva, va gestita prima l'interrupt della linea con numero minore). In base all'interrupt, `interruptHandler` chiama un'apposita funzione per la gestione:

- `PLT_interruptHandler`: chiamata per una PLT Interrupt, ovvero quando il processo attualmente in esecuzione esaurisce il tempo a sua disposizione; semplicemente resetta il timer, aggiorna il campo `p_time` del processo, lo re-inserisce in fondo alla coda della ready queue e chiama lo scheduler.
- `IT_interruptHandler`: chiamata per un'interrupt generata dall'Interval Timer, ovvero per un tick dello Pseudo-clock; re-inizializza il valore dell'Interval Timer assegnandogli il valore di `PSECOND`, sblocca tutti i processi in attesa di un tick dello Pseudo-clock, e continua con l'esecuzione del processo attuale se esistente, altrimenti chiama lo scheduler.
- `deviceInterruptHandler`: chiamata per un'interrupt proveniente da un dispositivo; innanzitutto identifica il dispositivo di cui "ascoltare" l'interrupt considerando quello di priorità massima con un'interrupt attiva sulla linea di priorità massima. Se si tratta di un terminale si ha una gestione a se stante per separare il caso della trasmissione e quello della ricezione. Individuato il dispositivo da gestire, ne si considera lo status (contenuto nel campo `transm_status` nel caso di trasmissione verso terminali, in `recv_status` nel caso di ricezione da terminali, in `status` per tutti gli altri dispositivi), lo si salva e si "invia" un acknowledgement al dispositivo scrivendo la costante `ACK` sul registro del dispositivo. In seguito, si sblocca un PCB fra quelli in attesa sulla coda di quel dispositivo: se nessun PCB era in attesa (ad esempio nel caso in cui un PCB con una richiesta in corso è stato terminato prima di ricevere risposta) semplicemente si procede, altrimenti il PCB va sbloccato inviandogli un messaggio ed inserendolo nella ready queue dopo aver scritto il valore dello status del registro del

dispositivo sul suo registro `reg_v0`. Infine, se prima dell'interrupt c'era un processo in esecuzione questo viene ri-caricato e continuato, altrimenti si chiama lo scheduler.