Progetto Pandos

Fase 2

Indice

1 Inizializzazione

Dopo che il sistema operativo è stato caricato dal BIOS nelle giuste locazioni di memoria, la funzione `__start()` cede il controllo alla funzione `main()`, che esegue una serie di assegnamenti per garantire il corretto funzionamento una volta che l'esecuzione passerà allo scheduler, attraverso l'invocazione di `scheduler()`. Inoltre, il `main()` inizializza tutte le strutture dati che serviranno successivamente per la gestione dei *pcb*. Più in generale, il file `init.c` si occupa di fornire una serie di *semafori* e variabili globali a tutte le altre componenti per garantire un adeguato log dei tempi di utilizzo del processore. Infine, `main()` inizializza il primo *pcb* così da permettere allo scheduler di eseguire la funzione `test()`.

La prima operazione eseguita dalla funzione `main()` è l'invocazione di `initSystem()`, che esegue alcune delle principali istruzioni di inizializzazione.

1.1 Dichiarazione delle variabili globali

Il file `init.c` contiene la dichiarazione di tutte le variabili globali, impiegate dal resto delle funzioni del kernel, tra cui la gestione delle SYSCALL, degli interrupt e delle Trap. Queste variabili consistono in:

- `processCount`: intero che indica il numero di *pcb* attivi presente nel sistema, compresi quelli bloccati;
- `blockedCount`: intero che indica il numero di processi bloccati presso la queue associata ad un *semaforo*;
- `devSem` e `terSem`: si tratta delle matrici che rappresentano i *semafori* di ciascuna istanza di ogni dispositivo;
- `pseudoClock`: intero che rappresenta il *semaforo* associato all'Interrupt Timer;
- `processStartTime`: intero usato per calcolare il tempo di utilizzo del processore, è assegnato dallo scheduler quando un processo viene avviato;

- `bus_devReg_Area`: puntatore di tipo `devregarea_t` usato per accedere rapidamente ai registri dei dispositivi. Viene inizializzato all'indirizzo dove comincia il bus register area, cioè 0x1000.000.

1.2 Inizializzazione del Pass Up Vector

L'inizializzazione del Pass Up Vector consiste nell'usare un puntatore di tipo `passupvector_t` chiamato `passUpP0`. A tale variabile viene assegnato l'indirizzo 0x0FFFF900, che corrisponde alla posizione in memoria dove comincia il Pass Up Vector del Processore 0. Impiegando dunque la struttura dati `passupvector_t` si assegnano i vari campi eseguendo le istruzioni:

```
passUpP0->tlb_refill_handler = (memaddr) uTLB_RefillHandler;
passUpP0->tlb_refill_stackPtr = (memaddr) KERNELSTACK;
passUpP0->exception_handler = (memaddr) kernelExcHandler;
passUpP0->exception_stackPtr = (memaddr) KERNELSTACK;
```

che permettono di accedere ai 4 campi del Pass Up Vector, ossia l'indirizzo dei gestori delle eccezioni e del TLB-Refill Event, assieme ai rispettivi Stack Pointer.

1.3 Inizializzazione delle strutture dati

Sempre dentro la funzione `initSystem()` vengono inizializzate le strutture dati relative ai *pcb* e ai *semafori* attraverso la chiamata delle funzioni `initPcbs()` e `initASL()`.

1.4 Inizializzazione delle variabili gloabali

L'ultimo compito di `initSystem()` è l'assegnamento di adeguati valori alle variabili globali:

- `processCount` a 0 poiché non ci sono processi in esecuzione;
- `blockedCount` a 0 poiché non ci sono processi bloccati;
- `readyQ` inizializzato con la funzione `mkEmptyProcQ()` in modo che punti a NULL;
- `currentProcess` a NULL poiché non è presente un processo corrente;
- Chiamata della funzione `LDIT()` con parametro `PSECOND`, che ha come conseguenza il caricamento di 100 millisecondi nel timer globale.

1.5 Inizializzazione del primo processo

Il controllo viene restituito a `main()`, che inizializza i campi del primo *pcb* attraverso una variabile-puntatore e l'invocazione della funzione `allocPcb()`. Lo status del primo *pcb* viene impostato con kernel-mode attivo e interrupt abilitati, con tutte le linee di Interrupt Mask accese. In particolare, viene considerato il POP che verrà eseguito nello stack KU/IE, dunque vengono abilitati i bit di Interrupt precedenti anziché quelli correnti.

Viene poi impostato lo Stack Pointer del primo *pcb* attraverso la funzione `RAMTOP()` in modo che esso possa lavorare nell'indirizzo più "alto" della RAM disponibile. Successivamente, viene impostato il PC (assieme al registro `reg_t9`) così che l'esecuzione possa partire dalla funzione `test()` quando lo scheduler cederà il controllo al *pcb*.

Infine, attraverso la funzione `insertProcQ()` il primo *pcb* viene inserito nella coda dei processi pronti, ovvero nella `readyQ`. Dopodiché il controllo viene dato allo scheduler attraverso la chiamata di funzione `scheduler()`.

Osserviamo che viene incrementata la variabile `processCount` di 1 poiché ora è presente un processo nella `readyQ`. Al temine della funzione `main()`, il controllo non tornerà più in esso.

1.6 Implementazione dei *semafori*

Per l'implementazione dei *semafori* è stato deciso di mantenere due matrici: `devSem` e `terSem`. Il primo è definito come una matrice di interi 4x8 inizializzato a 0 in tutti gli elementi. Per accedere al *semaforo* corrispondente al i-esimo tipo di dispositivo (Disk, Flash, ecc.) nella sua j-esima istanza è necessario accedere alla cella `devSem[i-3][j]` dove $7 \leq i \leq 3$ e $7 \leq j \leq 0$. Poiché ogni dispositivo terminale necessita di due *semafori*, è stato deciso di allocare la matrice `terSem` di dimensione 2x8, dove la riga 0 rappresenta i *semafori* di trasmissione mentre la riga 1 rappresenta i *semafori* per la ricezione per ogni istanza di terminale. L'accesso avviene con un meccanismo analogo a quello per i dispositivi non terminali. È stato poi definito un intero dedicato al *semaforo* dell'Interval Timer chiamato `pseudoClock`, inizializzato a 0. Il valore del *semaforo* viene usato in valore assoluto come indicatore del numero di *pcb* bloccati presso di esso.

Le strutture `semd_t` vengono allocate e deallocate in base alla necessità di bloccare processi e vengono identificati assegnando al campo `semAdd` l'indirizzo di memoria del *semaforo* che mantengono.

Osserviamo che la presenza di processi zombie nella queue dei *pcb* bloccati è evitata dall'implementazione della SYSCALL 2

2 Scheduler

Lo scheduler si occupa di eseguire i processi presenti nella ready queue e, assieme alla gestione degli interrupt del Processor Local Timer, implementa un algoritmo Round-Robin per gestire l'allocazione del processore.

2.1 Implementazione dello Scheduler

Si tratta di uno scheduler molto semplice, la cui implementazione consiste nell'estrarre un processo dalla `readyQ` ed effettuare alcuni controlli per identificare quale sequenza di istruzioni svolgere.

L'estrazione dalla `readyQ` avviene usando la funzione `removeProcQ()`, che si occupa di rimuovere dalla queue il primo *pcb* accodato. Tale *pcb* viene poi puntato dalla variabile globale `currentProcess`, che mantiene il processo attualmente in esecuzione.

Se `currentProcess` punta effettivamente a una *pcb* allora si procede con la normale esecuzione dei compiti dello scheduler, cioè l'assegnamento di `processStartTime`, il caricamento nel PLT di 5 millisecondi e l'avvio del processo appena estratto tramite la funzione `LDST()`.

Nel caso la chiamata di `removeProcQ()` non abbia restituito alcun *pcb* poiché la queue dei processi pronti è vuota, si procede con altri controlli. Si verifica innanzitutto se sono presenti processi attivi controllando la variabile `processCount`. Se tale variabile è a 0, significa che non sono presenti né processi pronti, né processi bloccati, di conseguenza si invoca la funzione `HALT()`.

Nel caso `processCount` è diverso da 0, si effettua un controllo sulla variabile `blockedCount`. Se tale variabile corrisponde a 0, significa che esistono processi attivi, ma non sono né bloccati, né pronti. Ciò è sintomo di Deadlock, di conseguenza viene invocata la funzione `PANIC()` per fermare l'esecuzione e stampare nel terminale il messaggio `kernel panic`.

Infine, nell'ipotesi in cui siano presenti processi attivi ma bloccati, si invoca la funzione `setSTATUS()` con i parametri `IECON|IMON` in modo da consentire gli interrupt da parte di tutti i dispositivi tranne il PLT, per poi entrare in un loop infinito chiamando la funzione `WAIT()` nell'attesa di un eccezione, che passerà il controllo al gestore elle eccezioni.

2.2 Gestione del tempo nello scheduler

La funzione `scheduler()`, oltre a caricare 5 millisecondi nel PLT, esegue un altro compito importante per la gestione del tempo: nel caso la `readyQ` restituisca un *pcb*, viene assegnata alla variabile globale `processStartTime` il valore del TOD, cioè viene mantenuto il momento in cui il *pcb* corrente incomincia l'esecuzione delle sue istruzioni. Tale valore sarà successivamente utile per calcolare il tempo di impiego del processore da parte del *pcb*.

3 Gestione delle eccezioni

Il gestore delle eccezioni a cui il BIOS cede il controllo è definito nella funzione `kernelExcHandler()` nel file `exceptions.c`. Tale file, oltre a contenere la funzione, mantiene anche una variabile globale `currentState`, visibile anche da altre funzioni, che permette di accedere al BIOSDATAPAGE, ovvero alla locazione di memoria 0x0FFFF000, dove viene salvato lo stato del processore nel momento in cui viene alzata un eccezione. E' infine presente la funzione `memcpy()`.

3.1 Il gestore delle eccezioni

Nel momento in cui il controllo giunge al gestore delle eccezioni, la prima operazione effettuata è l'estrazione, dal registro cause, dell'`excCode`, ovvero il codice che identifica l'origine dell'eccezione. Viene impiegato una struttura switch dove, a seconda di tale valore, l'esecuzione procede verso un differente handler, specializzato nella gestione di un particolare tipo di eccezione (SYSCALL, Interrupt, TLB-Refill, ecc).

3.2 La funzione `memcpy()`

Come accennato prima, il file `exceptions.c`, oltre alla variabile `currentState` e al gestore delle eccezioni, presenta anche la funzione `memcpy()`, necessaria per effettuare l'operazione di copiatura di strutture più grandi di una parola di memoria (32 bit), come ad esempio una struttura di tipo `state_t`. Tale esigenza si presenza quando si vuole copiare lo stato mantenuto nel BIOSDATAPAGE all'interno del campo `p_s` di un *pcb*, nel momento in cui esso sta per essere bloccato presso la queue di un *semaforo,* così da poter riprendere l'esecuzione dall'istruzione successiva a quella nel quale è stata bloccata (nel caso di SYSCALL) o dall'istruzione corrente (nel caso di PLT).

L'implementazione della funzione consiste in un semplice ciclo `for`, il quale impiega i parametri passati attraverso la chiamata per identificare l'indirizzo dove comincia la fonte, l'indirizzo dove comincia la destinazione e il numero delle parole da ricopiare a partire da questi indirizzi. Poiché il parametro `bytes` corrisponde ad una chiamata della funzione `sizeof()` da

parte del chiamante, viene effettuata una divisione per 4 per ottenere il numero effettivo di parole da ricopiare.

3.3 Il gestore degli Interrupt

Nel caso `excCode` corrisponda al valore 0, il controllo viene passato al gestore degli interrupt, definito nel file `interrupt.c` e identificato con la funzione `interruptHandler()`. Quando ciò accade, viene effettuata un'operazione simile a quella nel gestore delle eccezioni, ovvero l'estrazione dal registro cause dell'interrupt Mask, che viene mantenuto nella variabile `interruptCode` e corrisponde ai bit 8-15 del registro stesso. `interruptCode` permette di individuare la prima linea da cui proviene l'interrupt. L'implementazione di questo handler consente la gestione di al più un interrupt alla volta (ad eccezione del gestore degli interrupt provenienti dal terminale), vertendo sul fatto che, nel momento in cui gli interrupt saranno riabilitati, l'esecuzione verrà subito restituita al kernel nel caso di altri interrupt pendenti.

3.3.1 PLT handler

Il Processor Local Timer invia nella linea 1 un segnale di interrupt nel momento in cui il PLT, caricato dallo scheduler a 5 millisecondi, effettua il passaggio 0x000.000 → 0xFFFF.FFFF. Attraverso la condizione `(interruptCode & LOCALTIMERINT)` è possibile controllare la sua linea. Se è attiva, viene invocato il gestore `PLTHandler()`.

La prima operazione effettuata dal gestore è aggiornare il campo `p_time` del *pcb* che sta venendo interrotto, attraverso la chiamata della funzione ausiliaria `updateCPUtime()`. Tale funzione effettua una sottrazione tra il TOD attuale e il TOD nel momento in cui è stato caricato il *pcb* nel processore. Con questa scelta si è deciso di associare al *pcb* non solo il tempo di utilizzo della CPU, ma anche il tempo impiegato per la gestione delle SYSCALL che richiede, nonostante il fatto che durante la gestione di esse, le istruzioni eseguite siano del kernel.

Successivamente viene ricopiato lo stato corrente all'interno del *pcb* attraverso la funzione `memcpy()`, in modo che l'esecuzione possa successivamente riprendere dall'istruzione in cui il processo è stato interrotto. Infine, si accoda il processo corrente in fondo alla `readyQ`, si effettua un ACK della linea PLT caricando in esso un valore simbolico e si invoca lo scheduler così che l'esecuzione possa proseguire dal prossimo *pcb* nella `readyQ`.

3.3.2 Interval Timer handler

Seguendo la priorità definita dal gestore degli interrupt, nel caso non ci siano interrupt pendenti dalla linea del PLT, si verifica se è attiva la linea del interval Timer.

Se tale linea è attiva, si procede alla sua gestione invocando la funzione `timerHandler()` che, seguendo la semantica richiesta per l'Interval timer, estrae ogni processo bloccato nel suo *semaforo* e lo inserisce in fondo alla `readyQ`. Ciò avviene attraverso un ciclo `while`, che effettua una verifica sul valore di `pseudoClock` in modo da effettuare un numero di V operation pari al numero di *pcb* bloccati.

Osserviamo che ad ogni V operation, il valore di `pseudoClock` decrementa di 1. Tale meccanismo viene spiegato nel paragrafo 3.4.5.

Viene poi effettuato un ACK dell'interrupt caricando 100 millisecondi nell'interval timer attraverso la funzione `LDIT()`. Successivamente, con la chiamata di `checkCurrent()`, si verifica se al momento dell'interrupt ci fosse effettivamente un processo in esecuzione: in tal caso viene restituito il controllo invocando `LDST()`, altrimenti viene chiamato lo scheduler, che si occuperà di rimuovere un *pcb* dalla `readyQ` ed eseguirlo.

3.3.3 Disk, Flash, Network e Printer handler

Dalla lettura delle specifiche per gli interrupt non-timer, è stato deciso di aggregare i diversi gestori dei dispositivi Disk, Flash, Network e Printer in un'unica funzione, chiamato `IOdeviceHandler()`, a cui viene passato un parametro per identificare il tipo di dispositivo. Assieme alle precedenti linee dei timer, esiste una priorità con questi quattro dispositivi, garantito dalla struttura del gestore degli interrupt.

L'assunzione principale su cui si basa la scelta di scrivere un unico gestore sta nel fatto che qualunque status (Device not installed, Error ecc) sarà completamente gestito dal processo che ha effettuato il comando. Di conseguenza il gestore si limita a effettuare una V operation sul *semaforo*, verificare che ci siano effettivamente processi bloccati e salvare lo status del device nel registro `reg_v0` del *pcb* appena sbloccato.

Viene impiegata una serie di funzioni ausiliarie per individuare ed estrarre correttamente il registro del dispositivo che ha mandato il segnale di interrupt:

- `getBitmap()`: prende come parametro il numero del tipo di device (3 per disco, 4 per Flash ecc.) e restituisce il bitmap degli interrupt delle sue 8 istanze;
- `getLine()`: prende come parametro un bitmap e restituisce l'indice del primo bit a 1;
- `getDevReg()`: prende come parametro il numero del tipo di device ed un istanza del dispositivo, calcola e restituisce l'indirizzo in cui inizia il registro di tale istanza.

Con queste funzioni, si individua il registro del dispositivo in interrupt che viene poi puntato dalla variabile `deviceRegister`, usato per leggere e scrivere i campi del registro ed effettuare un ACK scrivendo il codice 1 nel campo COMMAND.

Infine, per restituire il controllo si verifica innanzitutto se è presente un `currentProcess` (invocando `checkCurrent()`) da cui riprendere l'esecuzione attraverso un `LDST()`, oppure proseguire verso lo scheduler.

3.3.4 Terminal handler

L'esigenza di scrivere un gestore separato per i terminali nasce dal fatto che essi possono operare come device di input e di output. Di conseguenza ogni linea di interrupt può identificare un interrupt di trasmissione, di ricezione o entrambi, dunque l'handler del terminale è l'unico in grado di gestire due interrupt allo stesso tempo, a patto che provengano dalla medesima istanza di terminale.

L'identificazione del registro del terminale in interrupt avviene in modo analogo agli altri dispositivi. Tuttavia, questo handler differisce nel fatto che il blocco di codice che effettua la V operation e scrive lo status nel registro `reg_v0` è racchiuso dentro un ciclo `for` che ripete 2 volte,

con i due cicli che operano in modo leggermente diverso. Assumendo che l'indice `i` codifichi con 0 il ciclo che gestisce la ricezione e con 1 il ciclo che gestisce la trasmissione:

- *Primo ciclo*: prima di entrare nel `for`, alla variabile locale `status` viene assegnato lo status della ricezione. Nel ciclo si impiega un blocco `if-then-else` per trasmettere un ACK nel campo COMMAND che identifica la ricezione. Si procede poi a effettuare la V operation e scrivere `status` nel registro `reg_v0`. Infine, si cambia la variabile `status` assegnandole ora lo status della trasmissione;
- *Secondo ciclo*: come nel primo ciclo, si segnala un ACK per la trasmissione, si effettua la V operation e si scrive `status` in `reg_v0`, ma stavolta in un *pcb* diverso poiché estratto dalla queue di un *semaforo* differente.

Con questi due cicli, è possibile gestire gli interrupt pendenti sia dalla ricezione che dalla trasmissione, se questi provengono dallo stesso dispositivo terminale.

Come per gli handler precedenti, si invoca `checkCurrent()` per verificare la presenza di un `currentProcess` per poi procedere con `LDST()` oppure con lo scheduler.

3.4 Il gestore delle SYSCALL

Nel caso l'`excCode` estratto dal registro `cause` abbia valore 8, il gestore delle eccezioni cede il controllo all'handler delle system call: la funzione `syscallHandler()`. Tale funzione è definita nel file `syscall.c` e, come i precedenti due "smistatori", la prima operazione che effettua è identificare quale tipo di SYSCALL è stato richiesto. Per fare ciò, impiega il contenuto del registro `reg_a0` come criterio per cedere il controllo alla funzione corretta.

Poiché ogni processo che ha effettuato la system call, venga esso bloccato o no, dovrà riprendere l'esecuzione dall'istruzione successiva a quella della chiamata di sistema. Viene quindi inglobata nel `syscallHandler()` un'istruzione che incrementa di 1 parola (cioè 4 bytes) il Program Counter dello stato corrente, salvato nel BIOSDATAPAGE. Nel caso di SYSCALL bloccante, tale stato verrà salvato nel *pcb* bloccato in modo che successivamente, quando la sua esecuzione potrà riprendere, continuerà dall'istruzione successiva.

Viene infine impiegato uno switch statement per cedere il controllo alla funzione che esegue la specifica system call richiesta, a cui vengono passati gli adeguati parametri.

3.4.1 Gestione del tempo nelle SYSCALL

Il campo `p_time` dei *pcb* viene aggiornato unicamente quando un processo viene bloccato presso un *semaforo* (o reinserito nella ready queue, fatto dal gestore dei PLT Interrupt), dunque le SYSCALL 3, 5 e 7. Ciò avviene attraverso la chiamata della funzione `updateCPUtime()`, il quale effettua una semplice sottrazione tra il TOD attuale e il TOD nel momento in cui lo scheduler ha messo il processo in esecuzione. Il motivo di questa gestione è spiegato nel paragrafo 3.3.1.

Osserviamo che l'effettiva chiamata della funzione `updateCPUtime()` avviene solo nella SYSCALL 3, mentre la 5 e 7 si appoggiano alla 3 per la gestione del tempo.

3.4.2 SYSCALL 1: Create Process

La prima operazione effettuata da `create_Process()` consiste nell'estrarre un nuovo *pcb* dalla coda dei *pcb* liberi, implementata nella fase 1, chiamando la funzione `allocPcb()`. Se tale estrazione non andasse a buon fine, significherebbe che non ci sono più *pcb* liberi. Si comunica ciò al processo corrente caricando il valore -1 nel suo registro `reg_v0`.

Se invece `allocPcb()` avesse successo, è possibile proseguire con la preparazione del nuovo *pcb* con le seguenti operazioni:

- Si ricopia lo stato fornito nel campo `p_s` del nuovo *pcb* invocando la funzione `memcpy()` con gli adeguati parametri;
- Si assegna al nuovo *pcb* la sua struttura di supporto. Nel caso essa non fosse presente, ci si aspetta che il processo corrente abbia fornito `NULL` come parametro;
- Si inizializzano adeguatamente i campi del *pcb*;
- Si invocano le funzioni `insertChild()` e `InsertProcQ()` in modo da rendere il nuovo processo figlio del processo corrente e inserire il *pcb* nella `readyQ`;
- Si comunica l'esito positivo della creazione inserendo il valore 0 nel registro `reg_v0` del processo corrente e si incrementa il contatore dei processi attivi `processCount`;
- Si restituisce il controllo all'esecuzione corrente chiamando `LDST()`.

3.4.3 SYSCALL 2: Terminate Process

E' stata presa la decisione di implementare la system call 2 attraverso un algoritmo ricorsivo, dove il processo corrente, prima di essere eliminato, invoca la funzione ricorsiva `terminate_ProcessRec()`, passando come parametro il puntatore al suo primo figlio.

All'interno di tale funzione, viene innanzitutto verificata la condizione di terminazione della ricorsione, cioè che il parametro passato sia `NULL`. Ciò può avvenire in 2 situazioni: quando non ci sono figli oppure si è giunti alla fine della lista dei fratelli. Queste due situazioni sono verificate con le 2 chiamate ricorsive

```
terminate_ProcessRec(pcbPointer->p_child);
terminate_ProcessRec(pcbPointer->p_next_sib);
```

che garantiscono l'eliminazione dei figli e dei fratelli prima del *pcb* passato come parametro.

Vengono poi effettuate una serie di operazioni per garantire la corretta rimozione del *pcb* dal sistema:

- Il *pcb* passato come parametro viene rimosso dalla lista dei figli a cui appartiene chiamando `outChild()`.
- Nel caso il *pcb* fosse nella ready queue, viene rimosso chiamando `outProcQ()`.
- Viene chiamato `outBlocked()` per rimuovere il *pcb* dall'eventuale *semaforo* presso cui è bloccato. Se tale estrazione ha successo, è necessario aggiornare il valore del *semaforo* e decrementare la variabile globale che tiene traccia dei processi bloccati.
- Si restituisce il *pcb* alla lista dei pcb liberi, e si decrementa il numero di processi attivi, `processCount`.

Al termine delle chiamate ricorsive il processo corrente viene rimosso dall'eventuale lista di figli a cui appartiene, viene restituito ai *pcb* liberi, si decrementa un ultima volta il numero di processi attivi e si cede il controllo allo scheduler poiché il processo corrente non esiste più.

3.4.4 SYSCALL 3: PASSEREN

La gestione della P operation esegue l'usuale decremento del *semaforo*, ed eventuale blocco del processo presso la queue associata a quel *semaforo*.

La funzione `passeren()` viene implementata prendendo come input l'indirizzo di memoria del *semaforo* da decrementare. Eseguita questa operazione, si controlla il valore mantenuto dal *semaforo*. Se è negativo, significa che la risorsa richiesta non è disponibile ed è necessario accodare il processo alla queue associata a quel *semaforo*, nell'attesa che arrivi il suo turno. Ciò viene fatto con una serie di operazioni:

- Viene mantenuto lo stato corrente dell'esecuzione ricopiandolo nello stato del *pcb* chiamando la funzione `memcpy()` con i parametri adeguati.
- Viene aggiornato il tempo mantenuto dal campo `p_time` chiamando la funzione `updateCPUtime()`.
- Impiegando la funzione della fase 1 `insertBlocked()` è possibile individuare/allocare una struttura `semd_t` per gestire il *semaforo* e bloccare presso la sua queue il *pcb*. Se tale funzione restituise `NULL`, significa che non ci sono più `semd_t` disponibili, e si blocca il sistema alzando il kernel panic.
- Se il blocco del *pcb* è andato a buon fine, si incrementa il valore del contatore dei processi bloccati e si invoca lo scheduler per proseguire l'esecuzione.

Se invece, il valore mantenuto dal *semaforo* non è negativo, si può procedere con l'esecuzione della sua sezione critica, restituendo il controllo all'esecuzione corrente.

3.4.5 SYSCALL 4: VERHOGEN

La gestione della V operation è stata suddivisa su due funzioni, a seconda dell'origine della chiamata. Se la richiesta arriva dall'esterno del kernel, la gestione è catturata da `verhogenExternal()`, che effettua una chiamata di `verhogen()` per poi restituire il controllo all'esecuzione corrente.

`verhogen()` incrementa il valore del *semaforo*. Se è negativo o corrispondente a 0, significa che è presente un processo bloccato presso il *semaforo*, dunque si estrae il primo *pcb* dalla queue dei bloccati, lo si inserisce nella `readyQ` e si aggiorna il contatore dei processi bloccati, per poi restituire il processo bloccato.

Questa suddivisione della SYSCALL 4 è stata decisa poiché la funzione `verhogen()` viene impiegata internamente dal kernel nella gestione dei *semafori* dei diversi interrupt, i quali necessitano di accedere al campo `p_s.reg_v0`. Di conseguenza `verhogen()` è stato implementato in modo da restituire il *pcb* appena sbloccato, se presente.

3.4.6 SYSCALL 5: Wait for IO

Quando viene richiesto il servizio fornito dalla system call 5, il controllo viene passato alla funzione `wait_for_IO_device()`. Tale funzione si limita a identificare presso quale *semaforo* dei dispositivi bisogna bloccare il *pcb* corrente, per poi effettuare una chiamata della funzione `passeren()` con gli adeguati parametri.

Viene effettuato un controllo particolare per identificare quale dei due *semafori* associati ai terminali è necessario bloccare, usando il parametro passato `termReadFlag`, il quale contiene 1 se si vuole bloccare dal *semaforo* che gestisce la ricezione, 0 se gestisce la trasmissione.

3.4.7 SYSCALL 6: Get CPU Time

L'implementazione della system call 6 si limita a scrivere il valore del campo `p_time` nel registro `reg_v0` per comunicare la processo corrente il tempo per cui esso ha occupato la CPU.

3.4.8 SYSCALL 7: Wait For Clock

Quando viene richiesta la system call 7, il processo chiamante vuole attendere che l'Interval Timer segnali un interrupt, di conseguenza l'implementazione si limita a chiamare la funzione `passeren()` passando come parametro il *semaforo* `pseudoClock`.

3.4.9 SYSCALL 8: Get SUPPORT Data

In modo simile alla system call 6, nella system call 8 viene richiesto unicamente un'informazione, che in questo caso si tratta dell'indirizzo presso cui si trova il campo `p_supportStruct`. L'implementazione si occupa di scrivere tale indirizzo nel registro `reg_v0` per poi restituire il controllo all'esecuzione corrente.

3.5 Pass Up or Die

Il passaggio del controllo al livello di supporto avviene nella funzione `passUp_orDie()`, il quale viene invocato sia dal gestore degli interrupt che da quello delle SYSCALL in varie situazioni. La funzione è definita nel file `syscall.c`.

Il primo controllo effettuato dalla funzione è la verifica dell'effettiva presenza, della struttura di supporto. Nel caso essa manchi, il processo viene subito terminato. Nel caso sia presente, si procede a ricopiare lo stato dell'esecuzione corrente presso il campo

```
currentProcess->p_supportStruct->sup_exceptState[contextNumber]
```

dove `contextNumber` è il parametro passato dal gestore chiamante per indicare la posizione in cui salvare lo stato. Ciò viene fatto con la chiamata della funzione `memcpy()`.

Vengono poi estratti `stackPtr`, `status` e `procCounter` dalla struttura di supporto, e si chiama `LDCXT()` a cui vengono passati i valori appena ottenuti in modo da poter proseguire l'esecuzione verso il livello di supporto.

Considerazioni finali

Durante lo svolgimento della fase 2 abbiamo dovuto effettuare diverse modifiche al codice delle funzioni della fase 1. In particolare, è stato necessario modificare il meccanismo con cui venivano estratti e inseriti i *pcb* dalle queue poiché talvolta venivano cancellati valori significativi.

La prima implementazione del calcolo del tempo di utilizzo della CPU era stata pensata in modo da consentire esattamente 5 millisecondi di esecuzione **del codice del processo**, escludendo dunque il tempo di gestione delle system call, e mantenendo anche il tempo consumato prima di essere bloccato, usato come base dallo scheduler per impostare il PLT. Ma tale meccanismo si è dimostrato più complesso del previsto e causa di errori in diverse altre componenti, dunque è stato deciso un approccio più largo con allocazione dei tempi più generoso, dove ogni processo messo in esecuzione dallo scheduler ha sempre 5 millisecondi, indipendentemente dal tempo consumato prima che venisse bloccato.

La divisione dei file è stata fatta in 4 componenti:

- `init.c`: contiene il `main()` ed esegue l'inizializzazione del sistema.
- `scheduler.c`: contiene lo `scheduler()`.
- `exceptions.c`: contiene il gestore globale delle eccezioni e la funzione `memcpy()`. Il primo si occupa di reindirizzare l'esecuzione verso il corretto gestore.
- `interrupt.c`: contiene il gestore degli interrupt che reindirizza l'esecuzione verso il corretto handler di interrupt.
- `syscall.c`: contiene il gestore delle system call e la funzione `passUp_orDie()`. Il prima si occupa di reindirizzare l'esecuzione verso la corretta funzione che si occupa di fornire quel servizio, mentre il secondo viene invocato per richiedere servizi dal livello di supporto.

Il Makefile è stato acquisito dal sito VirtualSquare ed è stato moderatamente modificato per assecondare le nostre esigenze.