Documentazione progetto SO fase 2

Scelte implementative

Sono qui sotto elencate cosa abbiamo fatto nei casi in cui la scelta è stata lasciata libera

Semafori

Abbiamo implementato semafori binari, quindi gli unici valori accettati sono 0 e 1. Una P sul valore 0 è bloccante, una V sul valore 1 è bloccante.

Le operazioni di P e V sono implementate in un file a parte. Abbiamo fatto questa scelta perché in questo modo è possibile avere più flessibilità di utilizzo e meno codice ripetuto. Infatti vengono richiamate nelle systemcall `passeren` e `verhogen`, ma anche in altri contesti. Le due funzioni restituiscono un valore booleano: *true* nel caso in cui la chiamata sia bloccante, *false* altrimenti.

Gestione dei semafori del nucleo

I semafori dei device gestiti dal nucleo li abbiamo implementati tramite un array di 48 elementi, questo perchè ci sono 5 interrupt line utilizzate dai device e per ognuna 8 device, più il terminale che ha 16 device (8 trasmissione, 8 ricezione), in totale `4*8+16 = 48`. Per accedere comodamente a questo array esiste una macro `dev_sem_addr(type, n)` che dato il tipo e il numero di device restituisce l'indirizzo del semaforo corrispondente.

Esiste anche una variabile globale `int pseudoclock_semaphore` che gestisce il semaforo dell'interval timer.

Gestione delle eccezioni

La funzione che gestisce le eccezioni è `exception_handler()` e si occupa di richiamare la procedura corretta come da specifica. Può richiamare `syscall_handler()` che si occupa solo delle syscall. Oppure può richiamare `interrupt_handler()` che si occupa solo degli interrupt.

Terminazione delle syscall

Subito prima della terminazione di ogni syscall viene richiamata una funzione `syscall_end(bool terminated, bool blocked)` che si occupa di generalizzare la corretta procedura per la terminazione. I casi da gestire sono tre:

1. il processo corrente termina
2. il processo corrente non termina e passa dallo stato *running* a *waiting*
3. il processo corrente non termina e continua la sua esecuzione

I due parametri booleani servono per identificare in quale stato si trova, **terminated** è *true* se siamo nel primo caso, se è *false* si ricorre a **blocked**, infatti se è

true siamo nel secondo caso, altrimenti si ricade nel terzo caso.

Gestione I/O

Per quanto riguarda la systemcall 5 `do_io(int *cmdAddr, int *cmdValues)`, per prima cosa estrapoliamo dall'indirizzo il tipo e il numero del device. Viene fatto tramite le macro `getTypeDevice(addr)` e `getNumDevice(addr)`. Visto che è necessario scrivere l'esito dell'operazione al posto del comando, salviamo l'indirizzo in una variabile globale `value_bak`.

Terminazione dei processi

La terminazione di un processo avviene tramite una funzione ricorsiva sui figli di tale processo.

Process Id

Abbiamo deciso di utilizzare l'indirizzo del pcb come PID del processo. Questo perché è più comodo rispetto a gestire un indice incrementale, inoltre semplifica anche la funzione che trova il pcb dato il PID.

Funzioni per gestire i device

I device e i loro indirizzi sono stati gestiti attraverso piccole funzioni e macro, ciò semplifica la lettura ed evita di ripetere blocchi di codice. All'avvento di un interrupt il device viene dedotto scorrendo iterativamente bit a bit la bit map degli interrupts.

PLT timer sempre caricato con il tempo corretto anche per fare l'ACK

Il comando `setTIMER()` viene sempre eseguito usando `TIMESLICE_TICKS`, questo perché veniva lasciata libera scelta dal manuale su come fare l'ACK sul PLT. Per coerenza abbiamo preferito usare sempre lo stesso tempo.

Tempo del Kernel assegnato ai processi che lo chiamano

Abbiamo deciso di assegnare il tempo durante il quale viene eseguito il Kernel (`exception_handler`, `interrupt_handler`, systemcall, ecc) al processo per il quale era stato richiamato. Questo viene fatto salvando il tempo dei processi subito prima di richiamare lo scheduler.

File `pandos_utils.h`

Abbiamo dichiarato le variabili globali del Kernel, come `current_proc`, i semafori, ecc, nel file `pandos_utils.h`. Abbiamo utilizzato la keyword `extern` perché in questo modo le variabili del file `initial.c` sono accessibili e visibili in tutto il Kernel.

Inoltre in quest'ultimo file abbiamo inserito una serie di utils come macro e definizioni che aiutano la leggibilità del codice. Queste ultime sono molto generiche e servono in molteplici parti del Kernel.

Problemi riscontrati

Problema con lo stato WAIT

> Reference: Section 7.2.2 Kernel-Mode MIPS Assembly Instructions from uMPS3 Principles of Operation

Come da manuale l'istruzione `WAIT()` potrebbe comunque lasciare eseguire codice successivo ad essa in caso di un masked interrupt. A questo punto abbiamo prima inserito una chiamata allo scheduler subito dopo la `WAIT()`, poi abbiamo inserito la chiamata in un ciclo while, così da effettuare un controllo nel caso sia tornato libero un processo. Infine abbiamo scelto di implementarlo disattivando nella maschera degli interrupt i bit relativi a PLT e all'Interval Timer.