JaeOS

Progetto di Sistemi Operativi

Prof: *Renzo Davoli*

Anno Accademico: 2015/2016

Gruppo 28:

Matteo Del Vecchio

Simone Preite

Intro

Lo sviluppo del progetto si è composto di tre fasi che avevano come scopo ultimo quello di scrivere un sistema operativo minimale *ARM-based*. Si è dimostrato molto utile per lo sviluppo l'utilizzo del software *git* che ha reso semplice e ordinata l'organizzazione e l'evoluzione del progetto.

Organisation

Per la struttura organizzativa si è deciso di dividere i file in base alle fasi di cui è composto ed il *makefile* è in grado di compilare tutto il progetto o singolarmente le varie fasi (vedi README.md). L'albero dei sorgenti parte dalla directory **"source"** che contiene le directory "**phase0**", "**phase1**" e "**phase2**", che contengono rispettivamente una directory per gli header e una per i sorgenti.

Per la prima fase, l'impostazione era dettata per lo più dalle specifiche mentre la seconda fase si compone di cinque file:

- ***initial.c***

  Main del programma, è l'entry point del progetto. Questo file si occupa di gestire le operazioni iniziali, popolare le aree di memoria e fornire i punti di accesso agli handler per le eccezioni e gli interrupt. Inoltre, vengono disabilitati gli interrupt per queste funzioni in modo tale che quando vi si accede, le operazioni non vengano interrotte appunto da interrupt.

  Viene allocato il primo processo e inizializzato, quindi messo in *readyQueue* per poi richiamare lo scheduler. Il controllo non tornerà mai più nel main.

- ***scheduler.c***

  Contiene, come si evince dal nome, lo *scheduler*, che regola l'avvicendarsi dei prcessi ed è strutturato in due casi.

  Quando esiste un processo corrente, significa che è avvenuto un interrupt o una system call che non ha bloccato il processo su di un semaforo e quindi deve essere ricaricato. In alternativa, ovvero quando il processo corrente è *NULL*, siamo nel caso in cui il processo che era in esecuzione si è bloccato su un semaforo oppure è terminato.

  Ora bisogna valutare due alternative:

  - C'è un processo in *readyQueue*, quindi viene semplicemente prelevato e caricato.
  - Non c'è un processo in *readyQueue*, quindi va applicato un semplice algoritmo di *deadlock detection*:
    1. *processCounter* è zero, significa che non ci sono più processi e la macchina può essere arrestata
    2. *processCounter* è maggiore di zero e *softBlockCounter* è uguale a zero potrebbe significare che il sistema è in stato di deadlock che viene gestito mandando il sistema in **PANIC**
    3. L'ultimo caso è che sia *processCounter* che *softBlockCounter* siano maggiori di zero, significa che ci sono processi bloccati su semafori di device quindi il sistema vieme messo in stato di **WAIT** e aspetta un interrupt, in modo che un processo venga liberato e possa tornare ad essere eseguito.

- ***interrupts.c***

  Questo file contiene i gestori degli interrupt per il terminale, il timer e gli altri device. Un handler centrale viene invocato quando avviene un interrupt, dopodichè la ***getDeviceNumberFromLineBitmap*** si occupa di capire quale device lo ha provocato, permettendo di richiamare l'handler per il tipo di device corretto.

  La funzione ***acknowledge*** viene richiamata da tutti i gestori di device e si occupa di gestire le operazioni comuni (opportunamente parametrizzate) ad essi.

- ***syscall.c***

  In questo file specifico sono state implementate tutte le system call privilegiate richieste dalle specifiche.

- ***exceptions.c***

  Gli handler per le eccezioni sono sviluppati in questo

modulo: qui ci sono i punti di accesso in caso di invocazione di una system call, program trap o accessi a zone di memoria non autorizzati. Anche in questo caso, come nella gestione degli interrupt e delle system call, una funzione, la ***handlerSYSTLBPGM***, si occupa di gestire le operazioni comuni ai tre handler.

Features

Durante la scrittura del codice ci siamo imbattuti in diversi problemi gestionali e sono state quindi prese delle decisioni per cercare di svilupparli nel modo più efficiente. Di seguito sono riportarte le informazioni che ci è sembrato più opportuno fornire nella relazione, il resto sarà commentato all'interno del codice:

- ***Generazione dei PID:***

  Per evitare la ripetizione di pid durante l'esecuzione viene utilizzato il timer, quindi la funzione ***genPid*** si comporta da generatore pseudo random: il concetto base è quello di prendere i due byte inferiori (del valore del timer) ed utilizzarli per modificare i primi due byte dell'indirizzo del processo ed assegnare il risultato al pid. Si noti infatti che essendo i processi inizializzati in un array, i loro indirizzi fisici sono contingui quindi la parte più significativa dell'indirizzo è quella meno variabile.

  Inoltre, questo metodo evita di arrivare a situazioni di **overflow** a causa di variabili incrementali.

- ***Verifica inizializzazione handler (system call 4, 5, 6):***

  All'interno della struttura del processo è stato aggiunto il campo *tags* dichiarato come *unsigned int*: l'ispirazione è venuta dal modo in cui Linux gestisce i permessi dei file. Viene inizializzato a 0 e man mano che le system call vengono chiamate per specificare l'handler, viene fatto uno **XOR** con la maschera per accendere il bit corrispondente (**sys:** 0b001 (1), **tlb:** 0b010 (2), **pgmtrap:** 0b100 (4)); se la variabile ha già il bit corrispondente accesso significa che la system call era stata chiamata in precedenza e il processo viene terminato.

  Lo stesso meccanismo è utilizzato per verificare se l'handler è stato specificato quando lo si controlla nelle eccezioni.

- ***Utilizzo di funzioni per evitare ripetizione del codice:***

  Durante la stesura non si è potuto non notare che molte parti del codice erano comuni a più elementi quindi sono stati raggruppati in funzioni ausiliare. Questo ci ha permesso di evitare le ripetizioni, rendendo tutto più compatto e facilmente manutenibile.

- ***Kernel time per interrupt:***

  Il tempo trascorso in **kernel time** a causa di un *interrupt* ***non*** viene addebitato al processo corrente perché l'interrupt non è stato causato di sicuro per una richiesta fatta da quest'ultimo ma da uno bloccato su un semaforo in attesa dell'interrupt.

- ***Set del timer dopo il risveglio da un semaforo:***

  Può succedere che un processo resti bloccato su un semaforo ed al suo ritorno in esecuzione è giusto che il timer non venga reinizializzato a *5 ms* ma che utilizzi il tempo restante che gli era stato concesso.

  La tecnica utilizzata è quella di tenere traccia del time slice rimanente in una variabile all'interno della struttura del processo, *remaining*, che viene decrementata del tempo che ha già eseguito ogni volta che si ferma su un semaforo.

  Questa variabile viene inizializzata a 5000 (micro-secondi) nel momento in cui il processo viene allocato e ogni volta che scade il suo time slice.

- ***Accounting della vita di un processo:***

  Quello che si cerca di fare è di fornire in maniera dettagliata e coerente il tempo che un processo ha trascorso in esecuzione, ovvero la sua vita, estrapolando da questo il tempo trascorso in kernel mode. In questo modo si possono tenere in considerazione tre variabili: *kernel-time, user-time e global-time*.

  Il tempo viene registrato ad ogni context switch, quindi nei momenti in cui avviene un interrupt del timer o il processo viene bloccato su un semaforo; inoltre per fornire un dettaglio temporale maggiore, i tempi vengono aggiornati anche nella chiamata alla system call **GETCPUTIME**.
  Per tenere traccia dell'esecuzione si fa ausilio di una variabile, *processStart*, che viene inizializzata ogni volta che il processo viene ricaricato e di questo si occupa lo scheduler.

  Mentre, per quanto riguarda il tempo trascorso in Kernel mode, viene utilizzata un'altra variabile, *kernelStart*, che viene inizializzata non appena si entra in un handler (a parte quello degli interrupt come detto precedentemente). Il ricalcolo del kernel time del processo viene effettuato quando l'esecuzione della system call torna il controllo all'handler, prima di richiamare lo scheduler.

- ***Pseudo clock 100 ms:***

  Concettualmente, la gestione dello pseudo clock avviene in modo incrementale, tenendo traccia del tempo trascorso tra un qualsiasi evento e quello precedente (ad

esempio la fine di un time slice oppure il caricamento di un processo). Vengono infatti utilizzate due variabili: *clockTick*, per tenere traccia della durata dell'ultimo intervallo di tempo tra gli eventi, e *clock* che rappresenta il vero valore dello pseudo clock.

Man mano che gli eventi si susseguono, il valore di clockTick farà incrementare quello di clock, per poi essere nuovamente inizializzato. In particolare, quando tale valore arriva o supera i 100 ms, il timer handler si occupa di calcolare il ritardo e reinizializzare opportunamente lo pseudo clock per tentare di recuperare tale ritardo.