YaeOS

Progetto di Sistemi Operativi – Phase 2

Anno Accademico 2017/18

Professore: Renzo Davoli

Gruppo 13:

Lorenzo Poluzzi

Lorenzo Turrini

Melania Ghelli

Daniel Veroli

Introduzione

Lo scopo della **phase 2** è di costruire il nucleo del Kernel. Questa fase si appoggia alle strutture dati costruite durante Phase 1 e le funzioni dalla ROM e della libreria messa a disposizione da uARM. Prevede lo sviluppo di 3 moduli specifici:

- Lo scheduler (per regolare l'alternarsi dei processi in esecuzione sul processore)
- Un'interfaccia di System Call che consenta ai processi di chiedere operazioni al nucleo quali creazione/terminazione di processi, sincronizzazione e operazioni I/O.
- Routine di gestione delle eccezioni per elaborare interrupt e trap, incluse operazioni per poter trasferire la gestione di alcune eccezioni ai livelli superiori dell'architettura del sistema operativo (passup)

Inizializzazione del Nucleo

File: init.c | init.h

Il nucleo inizia l'esecuzione dalla funzione *main* ().

Questa funzione è l'entry point di JaeOS, e include le seguenti operazioni:

- Inizializzazione le aree NEW nel frame riservate della ROM, con gli indirizzi che puntano alle routine di gestione (handlers).
- Inizializzazione delle strutture dati costruite in Phase 1
    - *initPcbs()*
    - *initASL()*)
- Inizializzazione delle variabili del Kernel
    - processCounter

    - SoftBlockCounter
    - curProc
- Inizializzazione del vettore dei semafori associati ai Device
- Creazione del primo processo (con parametri e registri correttamente inizializzati)
- Inserisco il primo processo nella readyQueue
- Viene chiamato lo Scheduler(), successivamente il controllo non ritornerà più alla funzione main ()

Lo Scheduler

File: scheduler.c | scheduler.h

YAEOS supporta uno scheduler preemptive a priorità con aging per la CPU.

Questo si occupa di regolare l'avvicendamento dei processi nella CPU: ogni processo ha un time-slice massimo di *3 ms* e questi vengono prelevati dalla readyQueue sulla base della loro priorità. A tal proposito per far sì che tutti i processi arrivino poi ad aggiudicarsi un posto nel processore, ogni *10 ms* viene lanciata una funzione di aging (***readyQueueAging()***) che si occupa di incrementare di 1 tutte le priorità dei processi nella readyQueue (purché questi non abbiano già priorità massima 10).

Lo scheduler comprende inoltre, un meccanismo di **deadlock detection** per prevenire e gestire nella maniera più opportuna possibili situazioni di stallo. Per fare ciò il sistema salva all'interno di due variabili specifiche (processCounter e SoftBlockedCounter) rispettivamente il numero di processi attivi nel sistema e il numero di processi soft-blocked ovvero in attesa del completamento di operazioni di I/O.

Nel caso non ci siano più processi nella readyQueue, possono dunque presentarsi **tre casi**:

- Non ci sono più processi attivi (processCounter == 0)
- Ci sono processi soft-blocked (processCounter > 0 e softBlockedCounter > 0)
- Non ci sono processi soft-blocked (processCount > 0 && softBlockedCount == 0)

Routine di gestione delle eccezioni

File: exception.c | exception.h

Classe che si occupa della gestione delle system call e delle eccezioni che possono essere sollevate di tipo Program Trap o TLB.

E’ composta da tre metodi principali (handler) per ogni tipo di eccezione sollevata:

- **pgmHandler**: metodo che si occupa di gestire le eccezioni di tipo Program Trap. Viene eseguito un controllo per vedere se è stato creato un gestore per quel tipo di eccezione, in caso affermativo viene inoltrata quest'ultima al gestore. In caso negativo viene terminato il processo.
- **tlbHandler**: metodo che si occupa di gestire le eccezioni di tipo TLB. Viene eseguito un controllo per vedere se è stato creato un gestore per quel tipo di eccezione, in caso affermativo viene inoltrata quest'ultima al gestore. In caso negativo viene terminato il processo.

- **sysHandler**: metodo che si occupa di gestire le system call e i breakpoint. Vengono eseguiti controlli specifici per evitare che vengano effettuate azioni che richiedono permessi specifici, da processi non privilegiati (syscall in user mode). In questo caso viene generata una Program Trap.

Se la richiesta è eseguita, invece, in kernel mode viene controllato che la system call chiamata sia tra 1 e 10 e in caso affermativo viene eseguita; mentre se la system call è maggiore di 10 si controlla che sia specificato un gestore di livello superiore per inoltrare l'eccezione, in caso negativo viene terminato il processo.

Sono state create delle funzioni ausiliare che svolgono il controllo per il gestore di livello superiore che ritornato 1 se esiste e 0 altrimenti.

- checkPGMHandler
- checkTLBHandler
- checkSysBpHandler

Time Management

File: pseudoTimer.c | pseudoTimer.h

In questo modulo viene gestita tutta la parte riguardante alla parte di TIMING del kernel.

Viene salvato l'istante di tempo in cui inizia il modo kernel/user attraverso metodi appropriati. Viene inoltre gestito l'aggiornamento e il reset della variabile clock necessaria a capire quanto tempo è trascorso. Questa viene resettata ogni qualvolta raggiunge un valore superiore o uguale a 10'000 ms.

Infine attraverso il metodo **setTimer** viene settato il valore del Interval Timer in modo da non perdere il tick dei 10000 ms, quindi viene controllato quale tra il time-slice e la differenza tra il TICK e il tempo trascorso è il minore tra i due.

Gestione degli Interrupt

File: interrupt.c | interrupt.h

Questo modulo si occupa della gestione degli Interrupt che posso essere sollevate dai vari Device e dal Interval Timer. E' composta dal metodo principale (intHandler) per la gestione degli Interrupt che vengono sollevati, per determinare quale device ha generato andiamo a controllare il campo CAUSE e dopo di che passiamo il controllo a Handler specifico per il device:

- **deviceHandler ()**

  In questo handler gli viene passato il tipo di Device e attraverso questo si va a cercare il deviceRegister che ha generato l'interrupt cosi da determinare indice dell’array dei semafori dei device. Infine viene richiama l’acknowledge per liberare il device che ha generato l'Interrupt.

- **terminalHandler ()**

  In questo handler gli viene passato il tipo di Device e attraverso questo si va a cercare il deviceRegister che ha generato l'interrupt cosi da determinare indice dell’array dei semafori dei device. Questo è un caso particolare visto che il terminale è un device doppio (Ricezione, Trasmissione) quindi bisogna determinare in modo corretto l’indice dell’array dei semafori dei device. Infine si richiama l’acknowledge per liberare il device che ha generato l'Interrupt.

- **terminalHandler ()**

  In questo handler viene gestito lo pseudoClock aggiornando o resettando il suo valore. Viene controllato se è scaduto il tick del *10000 ms* in cui viene richiamato il metodo dello scheduler per aumentare la priorità e incrementato il contatore dei tick cosi da capire quando si arriva a *100000 ms* per sbloccare i processi bloccati dalla sys8.

Per concludere abbiamo creato una funzione ausiliaria:

- **acknowledge ()**

  Questo metodo serve per sbloccare il processo bloccato su device che ha generato l'interrupt e setta come valore di ritorno lo stato del risultato del commando che è stato chiamato dal processo su quel device. Infine setta nel campo command del device il valore DEV_C_ACK che indica che è stato gestito l'interrupt.

System call

File: syscall.c | syscall.h

All'interno di questi file sono contenute le implementazioni delle 10 System Call del nucleo. Riporto le funzionalità di queste ultime per completezza:

- **SYS 1**: Create Process

  Crea un nuovo processo come figlio del chiamante. Se la system call ha successo il valore di ritorno è zero altrimenti è -1. Se la chiamata ha successo cpid contiene l'identificatore del processo figlio (indirizzo del PCB).

- **SYS 2**: Terminate Process

  Termina il processo indicato (o il processo chiamante se pid==NULL) e tutta la progenie del processo indicato. Ritorna 0 se l'operazione ha avuto successo, -1 in caso contrario (ovviamente se pid==NULL oppure è il pid proprio o di un proprio avo, la chiamata se ha successo NON ritorna).

- **SYS 3**: P (Passeren)

  Realizza l'operazione di P su un semaforo. Il valore del semaforo è memorizzato nella variabile di tipo intero passata per indirizzo. L'indirizzo della variabile agisce da identificatore del semaforo.

- **SYS 4**: V (Verhogen)

  Realizza l'operazione di V su un semaforo. Il valore del semaforo è memorizzato nella variabile di tipo intero passata per indirizzo. L'indirizzo della variabile agisce da identificatore del semaforo.

- **SYS 5**: Specify Trap Handler

  Registra quale handler di livello superiore debba essere attivato in caso di trap di Syscall/breakpoint (type=0), TLB (type=1) o Program trap (type=2). Il significato dei parametri old e new è lo stesso delle aree old e new gestitre dal codice della ROM: quando avviene una trap da passare al gestore lo stato del processo che ha causato la trap viene posto nell'area old e viene caricato lo stato presente nell'area new. La system call deve essere richiamata una sola volta per tipo. Se la system call ha successo restituisce 0 altrimenti -1.

- **SYS 6**: Get Times

  Restituisce il valore di tre "tempi" del processo:

  - Il tempo usato dal processo in modalità user
  - Il tempo usato dal processo in modalità kernel (gestione system call e interrupt relativi al processo)
  - Il tempo trascorso dalla prima attivazione del processo.

- **SYS 7**: Wait for Clock

  Sospende il processo fino al prossimo tick di 100ms. Lo pseudoclock produce un tick ogni 100ms (esatti) e risveglia tutti i processi che hanno chiesto la wait for clock. Occorre fare in modo che non si accumulino gli errori di sincronizzazione dello pseudo clock (la scadenza del prossimo tick deve sempre essere posta a 100ms esatti dal precedente).

- **SYS 8**: I/O Operation

  Attiva l'operazione di I/O copiando il comando (command) nel campo comando del device register (*comm_device_register). Il chiamante verrà sospeso fino a completamento della operazione di input output; il valore di ritorno è il valore del registro di stato status (che indica quindi il successo o meno dell'operazione). I terminali sono device "doppi": c'è un campo command per ricevere e uno per trasmettere (un solo processo alla volta accede ad uno specifico device, i processi si sincronizzano tramite sezioni critiche).

- **SYS 9**: GetPIDS

  Restituisce il pid del processo stesso e del processo genitore. Se il campo pid o ppid è NULL il valore corrispondente non viene restituito. Per il processo radice *ppid è NULL.

- **SYS 10**: Wait Child

  Attende la terminazione di un processo figlio.

Sono state inoltre create altre **funzioni ausiliarie**:

- **checkIf_RCVMODE (unsigned int** *comm_device_register**, int** interrupt_line**, int** device_number**)**

  Controlla che la differenza fra il command_device_register e l'indirizzo del sub-device rientri nella prima di metà di indirizzi del device_register associata ai registri di RICEZIONE, o nella seconda metà associata invece alla TRASMISSIONE. System Call 8.

- **saveCurState(state_t *** state**, state_t *** newState**)**

  Copia I registri contenuti in “state” dentro i registri contenuti in “newState”