µPandOS

Contents

1 Cos'è?

µPandOS è un sistema operativo microkernel sviluppato per fini didattici; in particolare questa implementazione è fatta al fine di svolgere un progetto propedeutico all'esame per il corso `08574 - Sistemi Operativi` (anno accademico 2023/24) per l'università di Bologna.

2 Fase 1 - Definizione operazioni su liste di pcb e messaggi

2.1 Obiettivi

In questa fase andremo a scrivere le basi per quanto riguarda questo progetto, ovvero definiremo i metodi di due strutture fondamentali per quanto riguarda PandOS, ossia i messaggi e i PCB

2.2 Prototipi delle funzioni

2.2.1 Allocazione e deallocazione dei PCB:

- `void initPcbs()`: tramite la funzione `freePcb`, vengono aggiunti in coda gli elementi della `pcbTable` (da 1 a MAXPROC) nella lista dei processi liberi;
- `void freePcb(pcb_t *p)`: mette l'elemento puntato da `p` nella lista dei processi liberi;
- `pcb_t *allocPcb()`: rimuove il primo elemento dei processi liberi, inizializza tutti i campi e ritorna un puntatore ad esso.

2.2.2 PCB Queue:

- `void mkEmptyProcQ(struct list_head *head)`: inizializza una variabile come puntatore alla testa della coda dei processi;
- `int emptyProcQ(struct list_head *head)`: se la coda la cui testa è puntata da `head` è vuota ritorna TRUE, altrimenti FALSE;
- `void insertProcQ(struct list_head *head, pcb_t *p)`: inserisce il PCB puntato da `p` in fondo alla coda dei processi puntata da `*head`;
- `pcb_t *headProcQ(struct list_head *head)`: ritorna NULL se la coda dei processi è vuota, altrimenti il PCB in testa;
- `pcb_t *removeProcQ(struct list_head *head)`: rimuove la testa della coda dei processi puntata da `*head` e ritorna un puntatore dell'elemento in questione; se la lista è vuota ritorna NULL;
- `pcb_t *outProcQ(struct list_head *head, pcb_t *p)`: cerca mediante un `for_each` il PCB `p` nella lista puntata da `head` e lo rimuove; se lo trova ritorna `p` stesso, altrimenti NULL.

2.2.3 PCB Trees:

- `int emptyChild(pcb_t *p)`: ritorna l'esito della chiamata alla funzione `list_empty`, alla quale viene passato come parametro l'indirizzo del `list_head p_child` di `p`;
- `void insertChild(pcb_t *prnt, pcb_t *p)`: si assegna `prnt` al puntatore `p_parent` di `p`. Dopo si aggiunge `p` alla lista dei fratelli, tramite `list_add` (se non ci sono altri figli) e `list_add_tail` (per rispettare la FIFOness), alle quali viene passato come parametro gli indirizzi del `list_head p_sib` di `p` e del `list_head p_child` di `prnt` (`p` diventa fratello dei figli di `prnt` e quindi figlio di `prnt`).
- `pcb_t *removeChild(pcb_t *p)`: il controllo sulla presenza o meno di figli avviene con la funzione `emptyChild`. Se ci sono figli, si sceglie il primo figlio tramite la macro `container_of`, chiamata sull'elemento successivo al `list_head p_child`. In seguito il figlio viene eliminato tramite la funzione `list_del` e viene troncato il legame con il padre, assegnando il valore NULL al puntatore `p_parent` del figlio.
- `pcb_t *outChild(pcb_t *p)`: se `p` ha un padre, rimuovo `p` dalla lista dei suoi fratelli chiamando `list_del` a cui passo come parametro l'indirizzo di `p_sib` di `p`, in seguito rimuovo il legame con il padre assegnando NULL al puntatore `p_parent` di `p`.

2.2.4 Allocazione e deallocazione dei messaggi:

- `void freeMsg(msg_t *m)`: Inserisce l'elemento puntato da `m` in testa alla lista dei messaggi.
- `msg_t *allocMsg()`: Ritorna `NULL` se la lista dei messaggi è vuota. Altrimenti rimuove un elemento dalla testa, imposta a 0 la variabile `m_payload` di ogni messaggio presente nell'array `msgTable` e ritorna un puntatore all'elemento rimosso.
- `void initMsgs()`: Inserisce gli elementi presenti nell'array `msgTable` in coda alla lista dei messaggi.

2.2.5 Message

- `void mkEmptyMessageQ(struct list_head *head)`: Inizializza una una lista di messaggi vuota.
- `int emptyMessageQ(struct list`$_h ead * head)$: $Ritorna 1 se la lista puntata da head è vuota, altrimenti 0$.
- `void insertMessage(struct list_head *head, msg_t *m)`: Inserisce il messaggio puntato da `m` in coda alla lista puntata da `head`.
- `void pushMessage(struct list_head *head, msg_t *m)`: Inserisce il messaggio puntato da `m` in testa alla lista puntata da `head`.
- `msg_t *popMessage(struct list_head *head, pcb_t *p_ptr)`: Rimuove il primo messaggio trovato nella lista puntata da `head` che è stato inviato dal thread p_ptr.
  Se `p_ptr` è NULL, ritorna il primo messaggio in coda.
  Se `head` è vuota o se non viene trovato alcun elemento mandato dal thread `p_ptr`, ritorna null.
- `msg_t *headMessage(struct list_head *head)`: Se la lista puntata da `head` è vuota ritorna NULL, altrimenti ritorna il messaggio in testa ad essa.

3 Fase 2 - Definizione del Nucleo, Scheduler, SSI, Interrupt ed eccezioni

Di seguito sono riportate le scelte progettuali per quanto riguarda i moduli sviluppati:

3.1 Utility

3.1.1 `timer.c`

In questo modulo abbiamo delle funzioni/procedure ausiliarie richiamate degli altri moduli:

- `unsigned int getTOD()`
- `void updateCPUtime(pcb_t *p)`
- `void setIntervalTimer(unsigned int t)`
- `void setPLT(unsigned int t)`
- `unsigned int getPLT()`

3.2 Inizializzazione nucleo

3.2.1 Dichiarazione e inizializzazione variabili globali

Nel modulo `initial.c` viene implementato il **main()**, la dichiarazione delle variabili globali:

- `int process_count` ossia il contatore dei processi attivi;
- `int soft_blocked_count` ossia il contatore dei processi bloccati;
- `int start` ...
- `int pid_counter`, usato per assegnare in maniera sequenziale i PID ai processi man mano che vengono creati;
- `pcb_t *current_process` ossia il puntatore al PCB del processo corrente;
- `pcb_t *ssi_pcb`, che è il puntatore al PCB del SSI;

3.2.2 Dichiarazione e inizializzazione strutture dati

Vengono inoltre implementate le strutture dati principali:

- attraverso le funzioni `initPcbs()` e `initMsgs()` vengono inizializzate le strutture della fase 1;
- `Ready_Queue`, ossia la lista dei proessi pronti ad essere eseguiti;
- 8 liste per i processi bloccati in attesa dei device o per il terminale (una per input e una per output);
- `void initPassupVector()` è una procedura che viene richiamata per definire il `pass up vector`, ossia è la struttura dati a livello hardware che indica a quale funzione passare il controllo quando si verifica un interrupt.

3.2.3 Interval timer

Viene caricato l'interval timer a 100 ms attraverso la chiamata alla procedura ausiliaria `setIntervalTimer(PSECOND)` definita in `timers.c`

3.2.4 Processi SSI e Test

Infine, prima di richiamare lo `Scheduler`, attraverso la procedura `void initFirstProcesses()` vengono inseriti nella `Ready Queue` i processi del SSI e del test. Questi avranno lo status settato in modo da avere la maschera dell'interrupt abilitata, l'interval timer abilitato e che siano in modalità kernel. Avranno rispettivamente pid 0 e 1.

3.3 Scheduler

Lo Scheduler è il componente che gestisce la coda dei processi pronti ad essere eseguiti (**Ready Queue**); la procedura principale che svolge tutto ciò è `void scheduler()`; questa parte con un controllo iniziale sulla `Ready Queue` vedendo se è vuota (con `emptyProcQ(&Ready_Queue)`):

- se non è vuota prendo il processo che deve essere preso in carico dalla CPU (`current_process`) con la funzione `removeProcQ(&Ready_Queue)`, setto il Timer attraverso la funzione `setPLT()` a 5 ms (con la costante **TIMESLICE**) per implementare il Round Robin, e infine viene caricato lo stato del processo corrente nel processore (con `LDST()`);
- altrimenti (se vuota), si effettua la Deadlock detection; in particolare può decidere se effettuare un `HALT()` quando non ci sono più processi da eseguire; se ci sono altri PCB entrerà in `WAIT()`; se la ready queue è vuota e ci sono processi bloccati si entra in deadlock invocando `PANIC()` fermando così l'esecuzione;

4 Fase 3 - ...

5 Crediti

5.1 Github

Il sorgente del progetto è reperibile nella seguente repository su Github.

5.2 Autori

- Fiorellino Andrea, matricola: 0001089150, andrea.fiorellino@studio.unibo.it
- Po Leonardo, matricola: 0001069156, leonardo.po@studio.unibo.it
- Silvestri Luca, matricola: 0001080369, luca.silvestri9@studio.unibo.it