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_head *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 per la gestione dei vari timer:

- `unsigned int getTOD()`: ritorna il valore del time of day clock, che viene nel nostro caso salvato nella variabile globale `start`: utilizzata per il calcolo del CPU time.
- `void updateCPUtime(pcb_t *p)`: chiama la funzione qui sopra descritta per aggiornare il valore del campo `p_time` del processo passato alla funzione.
- `void setIntervalTimer(unsigned int t)`: funzione che imposta il valore dell'interval timer.
- `void setPLT(unsigned int t)`: funzione che imposta il valore del processor local timer.
- `unsigned int getPLT()`: funzione che permette di ottenere il valore del processor local timer.

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 1 e 2.

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;

3.4 SSI

Essendo che µPandOS è un microkernel, le uniche syscall implementate sono la Send e la Receive; queste vengono usate dai processi per chiedere al processo SSI risorse; quanto detto è implementato nell'apposito modulo `ssi.c`, in particolare nella funzione `SSILoop()`, che implementa il polling del processo SSI: questa è eternamente in attesa di ricevere un messaggio da un qualsiasi processo che necessità una risorsa, prova a soddisfarlo attraverso l'apposita funzione `unsigned int SSIRequest(pcb_t* sender, ssi_payload_t *payload)` e se riesce viene inviato un riscontro al processo che ha effettuato la richiesta tramite la syscall send. Di seguito si forniranno maggiori dettagli riguardo quest'ultima funzione;

3.4.1 SSIRequest

All'interno di questa funzione vengono analizzati i parametri `pcb_t* sender, ssi_payload_t *payload` che contengono rispettivamente il processo che ha richiesto il servizio e il messaggio mandato col servizio richiesto; in particolare nel messaggio è determinante il `payload->service_code`, che serve a stabilire cosa serve al sender. Se vale:

- 1 (`CREATEPROCESS`): viene richiesta la creazione di un processo; questa richiesta viene soddisfatta solo se c'è spazio nella tabella dei processi liberi; in caso affermativo viene invocata la funzione `ssi_new_proces()` con parametro il sender che fungerà da parent e i dettagli del processo da creare;
- 2 (`TERMPROCESS`), che richiama l'apposita procedura `ssi_terminate_process()` passando parametro il processo da terminare, che è il sender se l'argomento (sempre passato fra i parametri) del messaggio è NULL, altrimenti quest'ultimo che è proprio il processo da terminare;
- 3 (`DOIO`), ...
- 4 (`GETTIME`), ...
- 5 (`CLOCKWAIT`), ...
- 6 (`GETSUPPORTPTR`), ...
- 7 (`GETPROCESSID`), che invoca la funzione `int ssi_getprocessid`; questa prende come parametro il sender e un argomento e ritorna il pid del sender se l'argomento è NULL, altrimenti il pid del processo padre del chiamante;
- Se il service code non contiene nessuno dei seguenti codici viene terminato il sender con la funzione `ssi_terminate_process()`.

4 Gestore delle eccezioni

La funzione che si occupa della gestione delle eccezioni è la funzione `void exceptionHandler()` dichiarata nel file phase2/include/exceptions.h e definita nel file phase2/exceptions.c. Questa funzione salva lo stato al tempo dell'eccezione dalla `BIOSDATAPAGE` ed in seguito trova il codice dell'eccezione eseguendo operazioni di manipolazione dei bit sul registro cause, ottenuto con la funzione `getCAUSE`. In particolare si esegue l'operazione `cause & GETEXECCODE` che permette di mantenere solo i bit che definiscono il codice dell'eccezione, i quali vengono shiftati a destra di 2 posizioni (costante `CAUSESHIFT`).

4.1 Interrupt Handler

Nel caso il codice dell'eccezione abbia valore 0 (costante `IOINTERRUPTS`) viene invocata la funzione per la gestione degli interrupt `void interruptHandler(int cause, state_t* exception_state)`. Qui viene utilizzata la macro `CAUSE_IP_GET(cause, line)`, grazie alla quale, passando il cause register e il valore di una linea di interrupt, è possibile sapere se c'è un interrupt su quella linea. Il controllo viene fatto per tutte le linee, seguendo l'ordine di priorità che va dall'interrupt causato dal processor local timer, all'interrupt causato da un dispositivo terminale. In base alla linea su cui avviene l'interrupt viene invocato un'opportuna funzione per la gestione di quello specifico interrupt.

4.1.1 Gestione Interrupt Processor Local Timer

L'interrupt causato dal processor local timer si verifica quando il tempo nella CPU per il processo corrente si esaurisce. Per un'opportuna gestione di questo interrupt usiamo la funzione `static void localTimerInterruptHandler(state_t *exception_state)`. In questa routine viene riconosciuto l'interrupt con la chiamata `setPLT(-1)`, in seguito si aggiorna il CPU time del processo corrente, si copia lo stato dell'eccezione nello stato del processo corrente, il quale infine viene inserito sulla ready queue. Dopo queste operazioni viene chiamato lo scheduler.

4.1.2 Gestione Interrupt Interval Timer

In questo caso l'ACK dell'interrupt è eseguito con la chiamata `setIntervalTimer(PSECOND)`. Dopodiché avvienelo sblocco di tutti i processi che erano in attesa dell'interrupt, rimuovendo ciascuno di essi dalla lista `Locked_pseudo_clock`, inserendoli sulla ready queue, dopo aver inviato loro un messaggio che consentirà ai processi interessati di sbloccarsi, quando rieseguiranno la SYS2 su cui si erano precedentemente bloccati. Ogni volta che viene rimossoun processo dalla lista dei processi in attesa dello pseudoclock tick, viene decrementata la variabile globale `soft_blocked_count`. Infine se il processo corrente è diverso da NULL, si esegue una LDST con lo stato dell'eccezione, altrimente viene chiamato lo scheduler.

4.1.3 Gestione Interrupt Device

La gestione degli interrupt legati a tutti gli altri device viene affidata alla funzione `static void deviceInterruptHandler(int line, int cause, state_t *exception_state)`, la quale ricava la bitmap degli interrupt per i dispositivi della linea d'interesse. Questo viene realizzato accedendo all'area di memoria riservata ai device, all'inidirizzo `BUS_REG_RAM_BASE`. In seguito si esegue l'and sui bit della bitmap con le costanti `DEVXON` con X $\in \{0, \ldots, 7\}$, con questa operazione si ottiene il numero del device sulla line che ha causato l'interrupt, per l'ordine con cui queste operazioni sono effettuate,il numero del device calcolato sarà sempre quello a priorità maggiore. Calcolato il numero, data la linea si sblocca il processo dalla lista associata alla linea cercandolo tramite il device number, grazie al campo aggiuntivo `dev_no` che abbiamo messo ai pcb. Questo campo viene settato dall'ssi quando viene bloccato il processo in attesa di interrupt durante il servizio DOIO. In caso di interrupt causato da un dispositivo terminale verifichiamo che il codice dell'operazione di trasmissione sia uguale a 5 (interrupt in attesa di essere riconosciuto), se così è allora significa che l'operazione è un'operazione di trasmissione di un carattere, altrimenti si tratta di un'operazione di ricezione.

Per accedere al registro del device usiamo la macro `DEV_REG_ADDR` nel modo seguente:
`dtpreg_t *device_register = (dtpreg_t *)DEV_REG_ADDR(line, device_number);`
In caso di dispositivi terminali, l'operazione è analoga:
`termreg_t *device_register = (termreg_t *)DEV_REG_ADDR(line, device_number);`
L'operazione di riconoscimento dell'interrupt avviene con l'istruzione
`device_register->command = ACK;`,
per i terminali a seconda di quale subdevice ha generato l'interrupt:
`device_register->transm_command = ACK;`
oppure
`device_register->recv_command = ACK;`.

L'accesso allo status avviene con l'operazione seguente: `device_register->status`.
Per i terminali:
`device_register->recv_status`.
`device_register->transm_status`.

Infine se il processo sbloccato è diverso da NULL, si mette lo status nel suo registro v0, gli viene inviato un messaggio avente la ssi come mittente e come payload lo status del device, si inserisce il processo sulla ready queue e si diminuisce di un'unità `soft_blocked_count`. In seguito se il processso corrente è diverso da NULL si chiama lo scheduler, altrimenti si esegue una LDST dello stato ottenuto dalla `BIOSDATAPAGE`.

4.2 Pass Up or Die

Tramite Pass Up or Die il kernel gestisce tutte le eccezione che non sono syscall o interrupt, abbiamo quindi implementato un'apposita funzione `static void passUpOrDie(int i, state_t *exception_state)` che controlla che la support struct del processo corrente sia diversa da NULL, se ciò è vero allora si salva lo stato dell'eccezione nello stato corretto della struttura di supporto:
`saveState(&(current_process->p_supportStruct->sup_exceptState[i]), exception_state);`
ed in seguito si esegue LDCTX, passando come parametri i valori del giusto constesto della struttura di supporto:

```
LDCXT(current_process->p_supportStruct->sup_exceptContext[i].stackPtr,
current_process->p_supportStruct->sup_exceptContext[i].status,
current_process->p_supportStruct->sup_exceptContext[i].pc
);
```

L'indice i, parametro della funzione, può assumere due valori a seconda del tipo di eccezione:

- `GENERALEXCEPT`: per trap generiche, con codici 4...7 e 9...12;
- `PGFAULTEXCEPT`: per eccezioni TLB, con codici 1...3;

  In caso di puntatore nullo, chiamiamo la funzione per la terminazione dei processi, che viene utilizzata dall'ssi per fornire il servizio `TERMINATEPROCESS`.

4.3 Eccezioni causate da SYSCALL

5 Fase 3 - ...

6 Crediti

6.1 Github

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

6.2 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