

μ PandOS

Patrick Alfieri - 0001030013

May 2024

Contents

| | | |
|----------|---|----------|
| 1 | Introduzione | 2 |
| 2 | Fase 1 - Manager delle code | 2 |
| 2.1 | PCB | 2 |
| 2.2 | Msg | 2 |
| 3 | Fase 2 - Nucleo | 2 |
| 3.1 | Inizializzazione del nucleo | 2 |
| 3.2 | Scheduler | 3 |
| 3.3 | Eccezioni | 3 |
| 3.3.1 | Interrupt | 3 |
| 3.3.2 | TLB e Program Trap | 3 |
| 3.3.3 | Syscall | 3 |
| 3.3.4 | PassUpOrDie | 4 |
| 3.4 | SSI | 4 |
| 3.4.1 | Creazione Processi | 4 |
| 3.4.2 | Terminazione Processi | 4 |
| 3.4.3 | DOIO | 5 |
| 3.4.4 | GetCPUTime | 5 |
| 3.4.5 | WaitForClock | 5 |
| 3.4.6 | GetSupportData | 5 |
| 3.4.7 | GetProcessID | 5 |
| 3.4.8 | Clocks | 5 |
| 3.5 | Interrupts | 6 |
| 3.5.1 | PLT | 6 |
| 3.5.2 | Interval Timer | 6 |
| 3.5.3 | Dispositivi | 6 |
| 3.5.4 | Uscita dall'interrupt handler | 6 |
| 3.6 | Altro | 6 |
| 4 | Fase 3 - TODO | 7 |

1 Introduzione

Questo documento vuole essere una relazione che descrive le principali scelte implementative attuate nel progetto del corso di Sistemi Operativi, CdL Informatica.

Lo scopo era realizzare un sistema operativo didattico attraverso tre fasi di astrazione: Manager delle code, Nucleo (kernel) e Livello di supporto. La tipologia di nucleo da implementare è di tipo microkernel, quindi saranno presenti poche Syscall che permettono di gestire il sistema nella sua interezza, soprattutto quelle che riguardano la comunicazione tra processi.

Il linguaggio di programmazione utilizzato è C ed i programmi vengono compilati attraverso un makefile. Inoltre, l'emulatore di riferimento per questa implementazione è uMPS3.

2 Fase 1 - Manager delle code

Il primo passo è stato implementare alcune funzioni da utilizzare con e sulle strutture fondamentali fornite in principio, quali **PCB** (*Process Control Block*), che rappresentano le informazioni legate ai processi e **messaggi**, che rappresentano la base per le operazioni di comunicazione.

Briefing descrittivi delle varie funzioni possono essere trovati direttamente nel codice.

2.1 PCB

Nessuna scelta implementativa particolare è stata attuata in questa sezione: si è cercato di utilizzare il più possibile le macro e funzioni già presenti nei file `const.h`, `types.h` e `listx.h`.

Generalmente, possiamo individuare tre principali categorie di funzioni implementate: allocazione/deallocazione dei PCB, gestione degli alberi della struttura (figli, genitori dei processi) e gestione dei processi nelle code.

2.2 Msg

Anche in questo caso, non ci sono implementazioni degne di nota.

Possiamo distinguere tra: allocazione/deallocazione dei messaggi e gestione di questi nelle code (inserimento, rimozione, etc...).

3 Fase 2 - Nucleo

Il nucleo rappresenta il vero e proprio cuore del sistema operativo ed è stato costruito attraverso il fornito Livello 1 (ROM) e l'implementato Livello 2, dunque le strutture e funzioni definite nella fase precedente. Le funzionalità delegate a questa importante componente possono essere da sei categorie principali: inizializzazione del nucleo, scheduler, gestione delle eccezioni (Syscall comprese), gestione degli interrupt, passaggio degli altri eventi al livello supporto e gestione SSI. Queste funzionalità sono state implementate in cinque moduli, le cui scelte implementative particolari sono discusse di seguito.

3.1 Inizializzazione del nucleo

Il modulo `initial.c` fornisce l'entry point per μ PandOS attraverso la funzione `main()`. In esso vengono inizializzate strutture e variabili propedeutiche al funzionamento del programma, tra cui i contatori dei processi totali e bloccati, ma anche un puntatore che traccia il processo corrente.

Si è scelto di implementare le liste dei PCB bloccati associati ai dispositivi, come code invece che array di dimensione `SEMDEVLEN - 1` per mantenere l'omogeneità rispetto alla coda dei processi pronti `ReadyQueue` e alle funzioni definite in Fase 1, che lavorano per la maggior parte su code.

Invece, `uTLB_RefillHandler` è una funzione placeholder fornita che agisce quando incorrono eventi TLB-refill (per traduzione indirizzi) e verrà sostituita durante l'implementazione di Fase 3 - Livello supporto.

Dunque, in questa sezione vengono definiti e allocati due puntatori PCB: `test`, ossia l'entry point per la funzione di test definita esternamente e `SSI`, ossia il processo "del sistema" costantemente in ascolto e in attesa che gli giungano richieste.

3.2 Scheduler

Lo scheduler si occupa di fare context-switch dei processi nella coda dei processi pronti. L'implementazione rappresenta uno scheduler round robin (quindi pre-emptive) con quanto di tempo assegnato ai processi di 5ms: se i processi non completano il proprio periodo di CPU Burst in questo lasso di tempo, verranno posti in fondo alla **ReadyQueue** e avverrà il dispatch di un altro processo (o lo stesso nel caso non ce ne siano altri).

Il meccanismo time slice é realizzato attraverso un clock interno che conta verso il basso: il PLT, che lancia un interrupt non caso sopracitato e permettendo al nucleo di gestire l'evento. Dunque, lo scheduling valuta il caso in cui la coda sia vuota (e quindi procede a fare deadlock detection attraverso i contatori) oppure ci sia almeno un processo e quindi lo carica.

3.3 Eccezioni

μ PandOS prevede diverse eccezioni che possono avvenire in seguito a eventi come **Syscall** e interrupts. Il modulo **exceptions.c** é delegato di riconoscere, prendere in carico e gestire tali casi, oltre che restituire il controllo. Si noti come tali vengono considerate operazioni privilegiate e quindi possono essere svolte solamente dal kernel, pertanto é necessario che il processore venga acceduto in kernel-mode. É possibile verificarlo attraverso il bit in posizione 1 (su 31) dello registro di stato.

Per accedere allo stato del processore si é scelto di utilizzare la macro **EXCEPTION_STATE**, che indica l'inizio del BIOS Data Page.

L'**exceptionHandler** viene chiamato con un nuovo stack ogni qualvolta viene lanciata un eccezione (TLB-refill esclusi) e attraverso lo stato del processore é possibile distinguere lo specifico tipo di eccezione avvenuta e quindi scegliere un gestore adeguato. I casi sono i seguenti.

3.3.1 Interrupt

Nel caso il risultato dell'operazione di riconoscimento, che é un AND bitwise tra il registro **cause** al momento dell'eccezione (32bit, interessano i bit 2-6) e la costante **GETEXCODE**, seguita da uno shift a destra di due posizioni, sia uno 0, allora si tratta di un interrupt e quindi deleghiamo al gestore apposito questo caso.

3.3.2 TLB e Program Trap

Le eccezioni TLB (codice 1...3) sono considerate dei page fault e dunque si delega alla funzione **passUpOrDie()** la gestione di questo caso. L'idea generale é che il controllo verrà passato al Livello Supporto (non ancora implementato), oppure il processo corrente e tutti i suoi figli verranno terminati. Questo caso é indicato dalla costante **PGFAULTEXCEPT**.

Lo stesso accade per le Program Traps (codice 4...7 e 9...12), il caso é indicato da **GENERALEXCEPT**.

3.3.3 Syscall

Come detto, le Syscall (codice 8) sono chiamate a funzione che permettono di interagire direttamente con il kernel e svolgere operazioni privilegiate. In quanto minimale, il kernel fornisce solo due servizi forniti attraverso Syscall attraverso viene gestito tutto il resto: **send()** e **recv()**. Prima di gestire i casi, é quindi necessario assicurarsi che il processore sia in kerne-mode (ossia che il bit 1 sul registro di stato sia posto a 0), in caso negativo viene richiamata la funzione **passUpOrDie()** in quanto Program Trap.

Al momento della chiamata, nel registro **a0** si troverá il codice attraverso il quale distinguere i tipi di syscall, mentre troveremo gli altri parametri rispettivamente nei registri **a1**, **a2**. Abbiamo quindi due casi:

- **send(unsigned int sender, unsigned int dest, unsigned int payload) (SYS1).**
Alloca e manda un messaggio, composto dal mittente **sender** e dal **payload** a un certo processo destinatario **dest**, ponendolo nella sua struttura apposita, la **msg_inbox**, cioè una lista di messaggi. Si noti come questa operazione, essendo asincrona, non necessita dell'attesa di un processo "ricevente". L'implementazione valuta anche il caso il destinatario fosse già in attesa di questo messaggio, quindi viene sbloccato. É inoltre importante menzionare che nei registri sono sempre contenuti degli **unsigned int** rappresentanti gli indirizzi, quindi sarà necessario un casting a puntatori di PCB. Un'operazione di **send()** andata a buon fine carica il valore 0 nel registro **v0**.

- `recv(unsigned int sender, unsigned int payload) (SYS2)`.

Il processo richiedente controlla se ci sono messaggi da parte del processo `sender`, oppure `ANYMESSAGE` se il mittente é indifferente. Se non si é trovato alcun messaggio, si sospende il processo richiedente e si richiama lo scheduler, altrimenti si prende il contenuto del messaggio. Questo passaggio é particolarmente importante dato che `payload` specifica l'indirizzo dell'area di memoria nel quale porre il payload recuperato dal messaggio. Ciò significa che é prima necessario realizzare un puntatore da tale indirizzo, poi dereferenziare su tale puntatore per metterci il contenuto di `msg_payload`. Se tale `payload` (parametro) é posto a 0, si ignora questa operazione. Una volta fatto questo il messaggio non serve piú e lo deallochiamo.

In entrambi i casi, per evitare il loop di Syscalls é prima necessario sommare una costante `WORDLEN` al Program Counter e poi caricare lo stato.

3.3.4 PassUpOrDie

La funzione "passa" al Livello Supporto la gestione di determinate eccezioni e passa il controllo agli appositi handler. In quanto é possibile che non ci sia un `current_process` nel sistema, viene fatto un controllo aggiuntivo per impedire di passare un processo nullo. Questa operazione richiede necessariamente la struttura di supporto del processo, per cui se non presente, quest'ultimo e i suoi figli verranno terminati.

3.4 SSI

Il modulo `ssi.c` ha il compito di fornire funzioni quali creazione e terminazione dei processi, ma anche sincronizzazione a seguito delle operazioni di I/O.

La System Service Interface é implementata attraverso un processo (`ssi_pcb`) nel sistema e costantemente in attesa di richieste, sottoforma di messaggi, ai servizi offerti. Una volta elaborata la richiesta, il processo `ssi` manderá un messaggio al processo richiedente con il risultato di tale elaborazione. L'SSI deve sempre rimanere in attesa e mai terminare, indi per cui nel caso accadesse é necessario fermare completamente il sistema.

Quindi, si é scelto di implementare il tutto attraverso un ciclo infinito cosí definito:

1. Il processo `ssi` si mette in attesa di richieste.
2. Riceve una richiesta da un processo richiedente attraverso un messaggio.
3. Determina il servizio richiesto attraverso un codice e lo elabora. Per questo punto in particolare si é utilizzato il casting a una struttura specifica: `ssi_payload_PTR`.
4. Il risultato torna all'SSI, che provvede ad inoltrarlo al processo richiedente attraverso un msg.

Ogni richiesta che giunge all'SSI é composta da un processo `sender`, un `unsigned int service`, che identifica il tipo di servizio richiesto e un eventuale `void *arg` da supporto per servizi specifici.

Di seguito, i servizi offerti dall'SSI.

3.4.1 Creazione Processi

Viene allocato un processo che diventa figlio del processo richiedente e viene inserito nella coda dei processi pronti per essere eseguiti. In questo caso, `arg` rappresenta il supporto da associare al processo figlio e quindi é stato castato alla struttura `ssi_create_process_PTR`, composta da stato del processo e la sua struttura di supporto.

L'operazione fallisce nel caso la coda dei processi liberi sia piena.

Questo servizio é associato al codice 1, ossia la costante `CREATEPROCESS`.

3.4.2 Terminazione Processi

Nel caso `arg` sia `NULL`, il servizio termina il processo richiedente (e tutti i suoi figli) e restituisce un valore `NOPROC` all'`ssi`, altrimenti termina il processo indicato all'interno di `arg` stesso.

La terminazione é implementata ricorsivamente sui figli e fratelli fino a che il processo padre non ne ha piú, dunque si termina il padre. Nel servizio é presente anche un controllo che verifica se il processo richiedente/specificato dal richiedente nel payload fosse bloccato in una delle liste per i processi bloccati associate ai dispositivi, nel quale caso affermativo si decrementa il contatore dei processi bloccati (dato che si fa uscire "forzatamente" dalla lista).

La terminazione avviene con le funzioni `outChild(sender)` e `freePcb{sender}`.

Questo servizio é associato al codice 2, ossia la costante `TERMPROCESS`.

3.4.3 DOIO

Su questa funzione e la sua controparte lato interrupts si sono concentrati i maggiori sforzi implementativi. Un'operazione di DOIO indica intuitivamente una richiesta di I/O associata ad un determinato dispositivo, cioè è necessario che il processo venga bloccato sulla coda associata.

In questo caso, `arg` è stato castato alla struttura `ssi_do_io_PTR`, in modo che contenga `deviceCommand` per verificare su che tipo di dispositivo è stata effettuata l'operazione di I/O e `commandValue` per far lanciare l'interrupt.

Le operazioni di I/O vengono gestite attraverso le linee di interrupt: ogni dispositivo è identificato da una linea di interrupt e i dispositivi, valutati in ordine di priorità, sono indicati attraverso un device number che va da 0 a 7. Questi dispositivi hanno gli interrupt pendenti sulle linee 3-7, questo significa che quando il bit i nella word j è posto a 1, allora il dispositivo i sulla linea $j + 3$ ha un interrupt pendente.

È importante distinguere tra dispositivi terminali (Transmission e Receive) e quelli non-terminali (Disco, Flash, Ethernet, Printer), in quanto il calcolo dell'indirizzo di base per ottenere il registro ad essi associato sarà pressoché uguale, ma cambierà la struttura.

Questo sistema di check sulle linee è realizzato attraverso due for: quello superiore itera sulle linee (3-7) e quello annidato sui dispositivi (0-7). A seconda della linea su cui si trova l'interrupt pendente, si riconosce se si tratta di un dispositivo terminale (e quindi distinguere ulteriormente tra i due terminali) oppure non-terminali. Quindi:

- Terminali → si utilizza un puntatore alla struttura di tipo `termreg_t` per identificare l'indirizzo di base del registro del dispositivo, quindi si effettua un controllo sui campi `transm_command` e `recv_command` di tale registro e si pone il PCB nella coda adeguata, bloccandolo.
- Non-terminali → similmente a sopra, ma si usa una struttura di tipo `dtpreg_t`, poi si blocca il PCB sul device corrispondente a `devNo`.

Non appena è stato bloccato un PCB su una qualsiasi coda, si esce dal ciclo. Questo perché possono esserci interrupt multipli su multiple linee, ma devono essere gestiti uno per volta. Infine, ponendo `commandValue` nell'area di memoria specificata da `commandAddr` si fa partire un interrupt.

Questo servizio è associato al codice 3, ossia la costante `DOIO`.

3.4.4 GetCPUTime

Viene semplicemente ritornato il valore del campo `p_time` del processo richiedente.

In questo caso, `arg` non ha utilizzo.

Questo servizio è associato al codice 4, ossia la costante `GETTIME`.

3.4.5 WaitForClock

Si inserisce il processo richiedente nella coda dei processi in attesa dei tick dello pseudo-clock.

In questo caso, `arg` non ha utilizzo.

Questo servizio è associato al codice 5, ossia la costante `CLOCKWAIT`.

3.4.6 GetSupportData

Si ritorna il valore (indirizzo) del campo `p_supportStruct` del processo richiedente.

In questo caso, `arg` non ha utilizzo.

Questo servizio è associato al codice 6, ossia la costante `GETSUPPORTPTR`.

3.4.7 GetProcessID

Si ritorna il PID del processo richiedente se `arg` è `NULL`, altrimenti quello del padre del processo richiedente.

Questo servizio è associato al codice 7, ossia la costante `GETPROCESSID`.

3.4.8 Clocks

Nel modulo `ssi.c` sono implementate anche le funzioni necessarie per gestire i tre clock presenti in μ PandOS: TOD (Time-of-Day), PLT (Processor Local Timer) e Interval Timer. Il primo serve per tenere traccia e aggiornare il campo `p_time` dei processi, il secondo per far scattare interrupt nei confronti delle politiche Round Robin dello scheduler (quando scade il quanto di tempo) e il terzo per sbloccare i processi nella coda che aspettano i ticks dello pseudo-clock (attraverso interrupt).

3.5 Interrupts

Gli interrupt sono una tipologia di eccezioni che presenta un proprio handler dedicato, costruito nel modulo `interrupts.c`. Ci sono a loro volta diversi tipi di interrupt: PLT, Interval Timer e Dispositivi I/O. Questi vengono distinti attraverso una maschera calcolata dal registro `cause`.

Ai fini di questa specifica versione di PandOS, sono stati ignorati gli interrupts inter-processor, dato che essa è pensata per un ambiente uniprocessore.

3.5.1 PLT

Un interrupt PLT avviene nel momento in cui `current_process` ha esaurito il tempo assegnato fornitogli dal quanto di tempo, ma non ha ancora completato il suo periodo di CPU burst.

Dunque, tale processo viene rimesso nella coda dei processi pronti e si richiama lo scheduler in modo che possa avvenire il context-switch. Il clock viene disabilitato durante la gestione degli interrupt e riattivato solamente quando si esce dal gestore (settando il bit corrispondente del registro `status` a 0/1).

3.5.2 Interval Timer

Un interrupt Interval Timer "sveglia" tutti i processi nella coda che stanno aspettando i tick dello pseud-clock, ossia quelli che hanno richiesto il servizio `WaitForClock`.

Questo clock è posto a 100ms, quindi ogni 100ms verrà lanciato un interrupt. I processi vengono sbloccati iterativamente facendo sì che l'SSI gli mandi un messaggio e che escano quindi dalla condizione di `recv` bloccante; ma mandarlo all'SSI attraverso `SYS1` e farlo inoltrare da lì risultava complicato e di difficile gestione (chi sarebbe stato il mittente? L'SSI stesso?), quindi si è optato per creare un messaggio da parte dell'SSI "fittizio" e porlo nel campo `msg_inbox` direttamente, senza fare uso della `send`.

I messaggi inviati non presentano un payload.

3.5.3 Dispositivi

Controparte della DOIO. Attraverso la linea dell'interrupt in presa in questione e una bitmask calcolata con le costanti `DEVXON`, si ottiene il numero di device corrispondente su cui sbloccare il processo. Di nuovo, due casi:

- Terminali → sempre utilizzando un puntatore alla struttura `termreg_t`, si controlla i contenuti dei campi `transm_status` e `recv_status` per determinare l'operazione avvenuta. Per sbloccare il processo, si è aggiunto un campo `int` alla struttura del PCB, che indicasse su quale device è stato bloccato. Quindi, si fa uso della funzione `unblockPcbDevNo(devNo, queue)` per cercare il PCB bloccato sul device `devNo` nella coda `queue` e quindi sbloccarlo. Dunque, si salva il valore corrispondente nella variabile `devStatus` e si fa l'`acknowledge` dell'operazione ponendo un valore adatto nel campo `trasm/recv_command`.
- Non-terminali → similmente a sopra, con la struttura `dtpreg_t`. Il device è individuato attraverso `devNo` e la linea dell'interrupt in questione. Anche qui si salvano status e si fa `ack`.
A questo punto, riprendendo l'idea realizzata per l'Interval Timer, viene mandato un messaggio ai processi da parte dell'SSI in modo da poterli sbloccare. Il payload in questo caso non sarà nullo, ma conterrà lo status, ossia il risultato dell'operazione di I/O che ne indicherà il successo/fallimento.

3.5.4 Uscita dall'interrupt handler

L'operazione di uscita dall'handler avviene per gli Interval Timer e i device (in quanto PLT richiama scheduler). Innanzitutto, si riattiva il clock PLT. Poi, se non c'è nessun `current_process` si carica lo stato del processore, altrimenti si richiama lo scheduler.

3.6 Altro

Oltre alle modifiche alla struttura PCB nel file `types.h` per permettere di individuare su quale dispositivo un PCB venga bloccato, nel file `const.h` sono state aggiunte alcune costanti con scopi differenti, quali `NETWORK_INTERRUPT`, `CAUSEREGSIZE`...

4 Fase 3 - TODO