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`, variabile globale utilizzata per calcolare il CPU time di ogni processo che viene eseguito.
- `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 `service_code`, che stabilisce l'oggetto della richiesta del sender. Attraverso l'utilizzo di uno switch su quest'ultimo parametro, la ssi fornisce il servizio richiesto. Si distinguono in particolare i casi:

- 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_process()` con parametri il sender, che fungerà da parent, e un puntatore a una variabile di tipo `ssi_create_process`, contenete i dati necessari alla creazione del nuovo processo; Nella funzione `ssi_new_process` viene chiamata `allocPcb()` per l'allocazione in memoria del nuovo processo e successivamente questo viene inserito nelle `ready queue` e nella lista child del pcb sender.

- 2 (`TERMPROCESS`), viene chiamata la procedura `ssi_terminate_process()` passando come parametro il processo da terminare. In questa funzione avviene ricorsivamente la terminazione di tutti i processi figli del sender e dei realtivi fratelli.

- 3 (`DOIO`), viene chiamata la procedura `ssi_doio` con parametri il puntatore al pcb del sender e un puntatore a una variabile di tipo `ssi_do_io_t` contenente i dati necessari a tale richiesta. Viene successivamente chiamata la procedura `addrToDevice()` che, dato l'indirizzo del device passato con la richiesta di DOIO, ne determina i rispettivi numero di device e linea, per poi chiamare la funzione `blockProcessOnDevice`. Quest'ultima, passati come argomenti il pcb del sender, il numero del device e un intero che indica se si sta facendo una lettura o una scrittura, rimuove il sender dalla `ready queue` e lo inserisce nella lista di processi bloccati del relativo device. La funzione `addrToDevice()` invidua la corretta linea e il corretto numero di device confrontando il command_address passato come argomento con tutti i campi di DEV_REG_ADDR[][]; per una maggiore efficienza questo viene prima confrontato con i campi relativi al terminale e successivamente, attraverso un ciclo for annidato, con i campi degli altri device.

- 4 (`GETTIME`), viene restituito come `unsigned int` un puntatore alla variabile `p_time` del processo sender.

- 5 (`CLOCKWAIT`), viene chiamata la funzione `ssi_clockwait` con argomento il puntatore alla struct `pcb_t` del processo sender. Quest'ultima inserisce il sender nella lista dei processi bloccati da pseudo clock. La procedura `pseudoClockInterruptHandler()` definita in `interrupts.c` si occupa della gestione di questa lista.

- 6 (`GETSUPPORTPTR`), viene restituito come `unsigned int` un puntatore alla variabile `p_supportstruct` del processo sender.

- 7 (`GETPROCESSID`), viene invocata la funzione `int ssi_getprocessid`. Quest'ultima prende come parametri il puntatore al pcb del sender e un puntatore generico arg, 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()`.

In alcune delle funzioni menzionate vengono inoltre modificati i valori di variabili globali quali:

- `soft_blocked_count` : viene decrementata in `ssi_terminate_process()` se il processo viene rimosso da una delle liste di processi bloccati dei device, o incrementata in `ssi_clockwait()` quando viene aggiunto.

- `process_count` : viene incrementata alla creazione di un nuovo processo in `ssi_create_process` e decrementata quando un processo viene terminato in `ssi_terminate_process()`.

3.5 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`).

3.5.1 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.

3.5.1.1 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.

3.5.1.2 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 rimosso un 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.

3.5.1.3 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`.

3.5.2 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`.

3.5.3 Eccezioni causate da SYSCALL

L'eccezioni da SYSCALL sono quelle il cui codice ha valore `SYSEXCEPTION`. Quando queste si verificano, viene chiamata la funzione `static void syscallExceptionHandler(state_t *exception_state)`. In primis la funzione controlla il valore del bit KUc, se questo è 1 (user-mode) viene generata una trap generica, in cui il valore del codice dell'eccezione viene settato a `EXC_RI` (istruzione riservata). Se il processo è in kernel-mode allora si continua e si controlla il registro a0 per capire quale servizio è stato richiesto:

- `reg_a0 = SENDMESSAGE`: il processo corrente vuole eseguire una `SYS1`, dal registro a1 viene estratto l'indirizzo del PCB al quale si vuole inviare un messaggio, si controlla se il destinatario è nella ready queue o se è il processo corrente, in entrambi questi casi si inserisce il messaggio nell'inbox del destinatario. Se il destinatario è sulla lista dei processi liberi, allora esso è inesistente e si aggiorna solamente il registro v0 al valore `DEST_NOT_EXISTS`. Se il processo destinatario non si trova sulla ready e queue e nemmeno nella lista dei processi liberi, allora dopo l'invio del messaggio, lo si inserisce nella ready queue (se il messaggio inviato non è quello per cui era in attesa, il processo si bloccherà di nuovo). La funzione che invia i messaggi è `int send(pcb_t *sender, pcb_t *dest, unsigned int payload)`. In questo caso come payload viene passato il valore del registro a2. Infine si incrementa il `PC` e si esegue `LDST(exception_state)`.

- `reg_a0 = RECEIVEMESSAGE`: il processo corrente vuole eseguire una `SYS2`, dal registro a1 si accede all'indirizzo del PCB dal quale il processo richiedente vuole ricevere un messaggio, questo valore viene passato, insieme alla inbox del processo corrente, alla funzione `popMessage`, la quale restituirà il messaggio o `NULL` se non lo trova. Nel primo caso la SYS2 non è bloccante, quindi si assegna al registro v0 l'indirizzo di chi ha inviato il messaggio, e si mette nell'area di memoria puntata dal registro a2 il payload del messaggio. Si incrementa il `PC` e si esegue `LDST(exception_state)`, poiché non è una syscall bloccante. Se invece la `popMessage` ha restituito `NULL`, allora la syscall sarà bloccante, lo stato dell'eccezione viene salvato nello stato del processo corrente, si aggiorna il CPU time del processo corrente ed infine si chiama lo scheduler.

- `reg_a0 >= 1`: viene generata una trap generica.

4 Fase 3 - Il livello di supporto

4.1 Inizializzazione

L'inizializzazione viene realizzata nel file initProc.c, in questo file il primo processo si occupa di creare tutti i processi necessari e di inizializzare le strutture dati necessarie al livello di supporto. Il codice di inizializzazione si trova nel file phase3/initProc.c, l'header corrispondente si trove nel file phase3/include/initProc.h .

4.1.1 Inizializzazione Swap Pool Table

La funzione che si occupa di questa operazione impostra l'asid di ogni entry della tabella al valore costante `NOPROC`.

4.1.2 Inizializzazione U-proc

In seguito viene realizzata l'inizializzazione delle strutture di supporto per gli U-proc. Per farlo allochiamo consecutivamente blocchi di memoria di dimensione fissa `PAGESIZE`. La memoria viene allocata a partire dall'indirizzo `RAMTOP - 3 * PAGESIZE`, ovvero il primo indirizzo disponibile, poiché i primi tre frame sono occupati dal processo ssi e dal processo test. In questa funzione, vengono assegnati alle support struct gli indirizzi dei gestori delle eccezioni del livello di support (`pager` e `generalExceptionHandler`). Nel frattempo anche gli stati dei vari u-proc vengono inizializzati, in particolare per questi processi si usa il medesimo indirizzo per lo stack pointer(`USERSTACKTOP`) e il medesimo indirizzo per il program counter (`USERPROCSTARTADDR`), in quanto essi fanno uso di indirizzo logici. Le strutture di supporto e gli stati vengono memorizzati in array, affinché gli sst possano poi creare gli u-proc, reperendo lo stato e la struttura di supporto corretti con facilità. Infine per ciascuna struttura di supporto avviene l'inizializzazione della rispettiva tabella delle pagine, impostando per ogni entry della tabella i valori per `pte_entryHI` (calcolo indirizzo logico) e `pte_entryLO` (accesione dirty bit).

4.1.3 Processo swap mutex

Per mantenere mutua esclusione sulla swap pool table si utilizza un processo che realizza tramite message passing un semaforo binario. Nella funzione `initSwapMutex()` viene allocata memoria per questo processo e al suo program counter viene assegnando l'indirizzo della funzione `swapMutex`, che implementa il semaforo eseguendo:

1. Receive senza mittente specificato (in questo modo il semaforo riceve le richieste da tutti i processi)
2. Send al mittente del messaggio ricevuto nel punto 1. (questo messaggio concede la mutua esclusione)
3. Receive dal mittente del messaggio ricevuto nel punto 1. (il semaforo attende il rilascio della mutua esclusione)

4.1.4 Inizializzazione processi device

Questi processi sono processi che ricevono dal rispettivo sst la stringa da stampare in output e questi realizzano un loop in cui ad ogni iterazione viene inviata una richiesta alla ssi per il servizio `DOIO` per la stampa del carattere della stringa in questione. Di questi processi ne sono stati realizzati 16 (8 terminali e altrettante stampanti). Eseguono tutti lo stesso codice, consistente in una funzione con segnatura `void print(int device_number, unsigned int *base_address)` che calcola i registri del dispositivo partendo da `base_address` (indirizzo della prima stampante o del primo terminale) calcola l'indirizzo del dispositivo in questione utilizzando la seguente formula:
`base_address + device_number * 4`
Poi dall'indirizzo calcolato si calcolano il registo di comando, e (nel caso delle stampanti) anche il

registro data0 nel quale andare ad inserire il carattere della stringa da stampare. Una volta realizzate queste operazioni preliminari può avvenire la ricezione della stringa e la conseguente procedura di I/O. In seguito per ciascun dispositivo si realizza una funzione wrapper che chiama la funzione parametrizzata qui sopra discussa, in modo da conferire dei nomi chiari e che consentano il raccoglimento di queste funzioni con un array di puntatori ad esse. Questo procedimento è stato implementato con il seguente codice:

```
   void print_term0() { print(0, (unsigned int *)TERM0ADDR); }
void print_term1() { print(1, (unsigned int *)TERM0ADDR); }
void print_term2() { print(2, (unsigned int *)TERM0ADDR); }
void print_term3() { print(3, (unsigned int *)TERM0ADDR); }
void print_term4() { print(4, (unsigned int *)TERM0ADDR); }
void print_term5() { print(5, (unsigned int *)TERM0ADDR); }
void print_term6() { print(6, (unsigned int *)TERM0ADDR); }
void print_term7() { print(7, (unsigned int *)TERM0ADDR); }

   void printer0() { print(0, (unsigned int *)PRINTER0ADDR); }
void printer1() { print(1, (unsigned int *)PRINTER0ADDR); }
void printer2() { print(2, (unsigned int *)PRINTER0ADDR); }
void printer3() { print(3, (unsigned int *)PRINTER0ADDR); }
void printer4() { print(4, (unsigned int *)PRINTER0ADDR); }
void printer5() { print(5, (unsigned int *)PRINTER0ADDR); }
void printer6() { print(6, (unsigned int *)PRINTER0ADDR); }
void printer7() { print(7, (unsigned int *)PRINTER0ADDR); }

void (*terminals[8])() = {print_term0, print_term1, print_term2, print_term3, print_term4,
print_term5, print_term6, print_term7};
void (*printers[8])() = {printer0, printer1, printer2, printer3, printer4, printer5, printer6,
printer7};
```

In questo modo nell'inizializzazione che seguirà sarà possibile assegnare il program counter accedendo agli elementi di questi array.

4.1.5 Inizializzazione SST

Dopo i processi device l'inizializzazione continua, sempre allocando la memoria dalla RAM, decrementando il valore di `curr` ad ogni iterazione. Anche qui si inizializza lo stato per ogni SST, in particolare il program counter ha valore `SST_loop`, che è la funzione che implementa gli SST.

4.1.6 Terminazione

Dopo questa serie di inizializzazioni, il processo test rimane in attesa (tramite receive) che ciascun SST comunichi ad esso la fine della propria esecuzione. Una volta che tutti gli SST hanno finito l'esecuzione, il processo test richiede la propria terminazione all'SSI. In seguito l'SSI sarà l'unico processo nel sistema e si avrà lo stato di HALT.

4.2 Gestore delle eccezione a livello di supporto

La funzioen che si occupa della gestione delle eccezioni a livello di supporto è la funzione `void generalExceptionHandler();` che prima di poter realizzare una corretta gestione della trap, deve ottenere dalla SSI (servizio `GETSUPPORTPTR`) la support struct dell'u-proc interessato. Da questa si ottiene lo state nel modo seguente:
`state_t* eception_state = &(sup_struct_ptr->sup_exceptState[GENERALEXCEPT]);`. Dopodiché si osserva il codice dell'eccezione dal cause register in questo modo:
`int val = (exception_state->cause & GETEXECCODE) >> CAUSESHIFT;`
In base al codice ricavato viene eseguito l'handler associato. Prima di ciò viene incrementato il program counter dello state ottenuto in precedenza.

4.2.1 Gestore eccezioni causate da SYSCALL

In base al valore contenuto nel registro a0, uno dei due servizi sarà eseguito:

- `reg_a0 = SENDMSG`: il processo corrente vuole eseguire una `SYS1`, in questo caso serve controllare il registo a1 per verificare a chi inviare il messaggio:
  - `reg_a1 = PARENT`, in questo caso l'u-proc sta cercando di inviare un messaggio al suo SST, per farlo viene quindi eseguita l'istruzione seguente:
    ```
    SYSCALL(SENDMESSAGE, (unsigned int)current_process->p_parent,
    exception_state->reg_a2, 0);
    ```
  - `reg_a1 != PARENT`, in questo caso l'u-proc sta semplicemente inviando un messaggio a un altro processo, per farlo viene eseguita l'istruzione seguente:
    ```
    SYSCALL(SENDMESSAGE, exception_state->reg_a1,
    exception_state->reg_a2, 0);
    ```
- `reg_a0 = RECEIVEMSG`: il processo corrente vuole eseguire una `SYS2`, viene quindi semplicemente eseguita questa istruzione:
  `SYSCALL(RECEIVEMESSAGE, exception_state->reg_a1, exception_state->reg_a2, 0);`

4.2.2 Gestore trap generiche a livello utente

Qualsiasi altro codice ricavato dal cause register viene interpretato come una trap inaspettata, di conseguenza l'u-proc interessato verrà terminato, ma non prima di aver rilasciato la mutua esclusione sulla swap pool, nel caso l'avesso precedentemente ottenuta.

4.3 Gestione memoria virtuale

In questa parte, i principali attori che rendono possibile il funzionamento della memoria virtuale sono:

- Il TLB, una cache utilizzata dalle unità di gestione della memoria della CPU per velocizzare la traduzione degli indirizzi virtuali in indirizzi fisici;
- Il TLB Refill Handler: quando un indirizzo virtuale non è presente nella TLB si verifica quello che in gergo è definito TLB Miss, ed entra in gioco proprio questo componente che sposta nel TLB la pagina che ha causato la trap per velocizzarne accessi futuri;
- La swap pool table;
- Il processo che permette di avere mutua esclusione (swap_mutex, discusso prima).

Il codice che implementa il meccanismo di gestione della memoria virtuale è nel file `vmSupport.c`

4.3.1 Pager

Il pager è quel componente responsabile dell'avvicendamento della pagine in memoria, in particolare andando a esaminare nel dettaglio l'implementazione:

- Come prima cosa, chiede la Support struct mandando un messaggio al processo SSI e attendendo una sua risposta;
- Una volta ottenuta, è necessario andare a vedere la causa dell'eccezione, in particolare se è una TLB-Modification exception procedo uccidendo il processo tramite la funzione apposita kill_proc(), altrimenti proseguo;
- è fondamentale per proseguire avere mutua esclusione, poichè queste sono operazioni sono molto delicate e non può essere che più di un processo entri nel pager; la mutua esclusione si ottiene mandando un messaggio al processo che se ne occupa e aspettando che questo risponda concedendola;
- una volta ottenuta, prendo dalla support struct la pagina virtuale con la macro `GET_VPN()`, e la pagina da rimpiazzare guardando prima fra quelle vuote e poi in caso facendo entrare in gioco l'algoritmo FIFO per selezionare una pagina vittima;
- dato il numero della pagina vittima è necessario calcolare il suo indirizzo, tenendo conto della dimensione della singola pagina e dell'offset della swap area:

```
memaddr victim_addr = (memaddr)SWAP_POOL_AREA + (i * PAGESIZE);
```

- nel caso la pagina vittima fosse scelta dall'algoritmo di rimpiazzamento, è necessario aggiornarla poichè precedentemente occupata da un altro frame appartenente ad un altro processo; dunque serve "pulirla" poichè ormai obsoleta (ovviamente tutto ciò in modo atomico per evitare inconsistenza);
- dopo aver letto il backing store del current_process, modifico il contenuto della entryHI nella swap table per far si che sia aggiornata con nuovi contenuti;
- disabilito gli interrupt momentaneamente (per non essere disturbati durante questo procedimento) e aggiorno la tabella delle pagine del processo corrente per la pagina p, indicando che la pagina è presente (V bit) e occupa il frame i;
- viene rilasciata la mutua esclusione restituendo il controllo al processo;

4.3.2 TLB Refill Handler

Quando avviene un cache-miss, il TLB ha come compito quello di inserire la voce mancante in tabella e far ripartire dall'ultima istruzione

4.3.3 Algoritmo di rimpiazzamento

Nel caso in cui non ci siano pagine libere, (ossia scorrendo la swap table non troviamo nessuna pagina con asid -1 [`NOPROC`]), se serve caricarne una nuova in memoria è necessario trovare un modo per scegliere la pagina vittima da rimpiazzare; questa logica è implementata dalla funzione `getPage()`, che dopo aver verificato che tutte le pagine siano occupate, sceglie la pagina da rimpiazzare tramite l'algoritmo FIFO. Di seguito si riportano le righe di codice che implementano tale algoritmo:

```
static int i = -1; return i = (i + 1) % POOLSIZE;
```

4.3.4 RWBackingStore

Funzione ausiliaria che si occupa della DoIO sui vari flash device; fa ciò inserendo nel campo DATA0 del dispositivo flash l'indirizzo fisico iniziale del blocco da leggere o scrivere; successivamente utilizza il servizio DoIO del SSI per scrivere nel campo COMMAND del dispositivo flash; per capire se leggere o scrivere sulla pagina desiderata viene passato come parametro una variabile discreta (w, che prende come valori solo 0 e 1), in particolare è scrittura se [w = 1] o lettura se [w = 0].

4.3.5 kill_proc

Funzione ausiliaria usata per gestire gli errori, che banalmente richiede la terminazione di un processo al SSI; prima di terminarlo libera tutte le pagine occupate da quest'ultimo.

4.3.6 cleanDirtyPage

Funzione ausiliaria usata dal pager che prende in input una pagina per invalidarla e aggiornare il TLB (in modo atomico disabilitando gli interrupt).

4.4 System Service Thread (SST)

Ogni System Service Thread (SST) ha lo scopo di creare il proprio processo figlio u-proc, e successivamente fornire a questo i servizi richiesti. Dopo la creazione del proprio figlio, sst rimane costantemente in attesa di richieste da parte di questo. Quanto detto è implementato nell'apposito modulo sst.c, in particolare nella funzione `SST_Loop()`, che implementa il polling dei processi SST e chiama la funzione SSTRequest per soddisfare la richiesta del processo figlio.

4.4.1 `SST_loop()`

In particolare `SST_loop()` contiene:

- Inizializzazione delle strutture di supporto necessarie alla creazione del processo figlio;
- Chiamata alla funzione `create_process()` definita in `initproc.c`, la quale crea un nuovo processo facendo una richiesta alla `SSI`;
- While loop contente system call bloccante per l'attesa di nuove richieste, chiamata alla funzione `SSTRequest`, e system call per la restituzione al processo figlio di un eventuale valore di ritorno (utile per `getTOD`).

4.4.2 `SSTRequest()`

Attraverso l'utilizzo di uno switch sulla variabile `service_code`, che determina il tipo di servizio richiesto da u-proc e viene passata all'interno della struct payload, questa funzione soddisfa la richiesta chiamando a sua volta apposite funzioni. Si distinguono in particolare i casi:

- `GET_TOD` Viene invocata la funzione getTOD(), la quale consente di recuperare il numero di microsecondi trascorsi dall'ultima operazione del sistema avviato/ripristinato. In questo caso viene impostato come valore di ritorno il numero di microsecondi.
- `TERMINATE` Viene invocata la funzione `sst_terminate()` la quale effettua una richiesta di terminazione di u-proc alla ssi e si occupa di mandare un messaggio al test per comunicare la terminazione del processo.
- `WRITEPRINTER e WRITETERMINAL` Viene invocata la funzione `sst_write()` la quale attraverso una syscall manda la stringa da stampare alla stampante o al terminale con lo stesso ASID del mittente. A `sst_write()` viene passato come argomento un parametro chiamato `device_type`, il quale determina la tipologia del dispositivo su cui verrà stampata la stringa passata nel payload (stampante o terminale). Viene quindi utilizzato uno switch su quest'ultimo parametro per differenziare i due casi.

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