Progetto Pandos

Fase 3

Indice

1 Inizializzazione dei Processi

L'inizializzazione dei processi comincia quando l'esecuzione giunge all'indirizzo di memoria contenuto nel campo `pc_epc` del primo processo, il quale viene preparato dal `main()` del file `init.c`.

1.1 Funzione `instantiatorProcess()`

La funzione a cui viene indirizzato il primo processo si trova nel file `initProc.c` e si chiama `instantiatorProcess()`. La prima operazione eseguita da tale funzione è l'impostazione della variabile globale `ramtop`, utilizzando la macro `RAMTOP()`, il quale assegna l'ultimo indirizzo di memoria della RAM nella variabile.

Successivamente vengono inizializzate alcune strutture dati del livello di supporto invocando le seguenti funzioni contenute nel file `vmSupport.c`:

- `initSwapPool()`: inizializza una serie di variabili globali del livello di supporto:
  - `frameNo`: variabile utilizzata dall'algoritmo di paging per individuare il frame della swap pool e la entry della `swapPoolTable` da restituire. Viene inizializzato con -1 a causa della struttura dell'algoritmo.
  - `swapPoolSem`: si tratta del semaforo che garantisce la mutua esclusione durante l'accesso alla `swapPoolTable`.
  - `swapPoolPtr`: è la variabile che mantiene l'indirizzo di memoria del primo frame disponibile nella RAM, cioè il primo frame successivo all'ultimo frame occupato dal OS. L'indirizzo di tale frame è calcolato chiamando la funzione `swapPoolBegin()`, il quale controlla il valore mantenuto in indirizzi specifici della zona `.text` del OS per ottenere la dimensione dei segmenti `.text` e `.data`.
  - `swapPoolTable[]`: è un array che indicizza i frame della swap pool fisica. Viene inizializzata con un ciclo `for` in modo che ogni frame risulti inutilizzato (campo `sw_asid` a -1)

- `initSupSem()`: questa funzione inizializza a 1 i semafori per i device di I/O del livello di supporto. Tali semafori sono definiti nella variabile `supDevSem[][]`, strutturato come un array 4x8, dove le gli indici `supDevSem[0][]` definiscono i semafori per i dispositivi Flash, gli indici `supDevSem[1][]` definiscono i semafori per i dispositivi Printer, gli indici `supDevSem[2][]` definiscono i semafori per i dispositivi Terminal in scrittura e gli indici `supDevSem[3][]` definiscono i semafori per i dispositivi Terminal in lettura. Ciascuno di essi mantiene 8 variabili, ognuno dei quali indica una specifica istanza del dispositivo associato al U-proc.

1.2 Inizializzazione dei Processi Utente

Una volta inizializzate le strutture del livello di supporto, `instantiatorProcess()` invoca la funzione `initUProc()`. Tale funzione esegue un ciclo `for` un numero di volte pari a `UPROCMAX`, dove ad ogni iterazione viene definito e inizializzata una variabile di tipo `state_t` chiamata `newState`:

- `newState.pc_epc`: il Program Counter dei U-proc viene inizializzato sempre all'indirizzo logico `0x800000B0`, ovvero dove incomincia il codice del processo.
- `newState.reg_sp`: come il Program Counter, lo Stack Pointer di tutti i processi viene inizializzato all'indirizzo logico `0xC0000000`.
- `newState.status`: gli U-proc vengono inizializzato con tutti gli Interrupt abilitati, Timer interrupt attivo e in modalità utente.
- `newState.entry_hi`: questo registro mantiene l'`ASID` del processo, che viene calcolato impiegando l'indice `i` del ciclo `for` e posizionato correttamente nella parola di memoria grazie alla costante `ASIDSHIFT`.

Successivamente viene inizializzato un indice della variabile globale `supportTable[]`, il quale mantiene le strutture di supporto per i diversi U-proc:

- `supportTable[i].sup_asid`: campo inizializzato con l'`ASID`, calcolato usando l'indice `i` del ciclo `for`.
- `supportTable[i].sup_exceptContext[].c_pc`: il Program Counter dei due gestori, uno per i Page Fault e uno per le eccezioni generali del livello di supporto. Viene inizializzato rispettivamente alle funzioni `Pager()` e alla funzione `generalSupHandler()`.
- `supportTable[i].sup_exceptContext[].c_stackPtr`: gli Stack Pointer per i gestori delle eccezioni del livello di supporto vengono posizionati al di sotto dello Stack Pointer del processo iniziale, a partire dai frame di RAM più alti, sfruttando `ramtop`.
- `supportTable[i].sup_exceptContext[].c_status`: lo status durante l'esecuzione dei gestori del livello di supporto viene impostato con Interrupt abilitati e modalità Kernel.

Viene dunque invocata la funzione `initUPageTable()`, il quale inizializza la Page Table dei processi. Ciò viene fatto attraverso un ciclo `for` che esegue un numero di volte pari a `USERPGTBLSIZE - 1`, dove vengono inizializzati i campi `pte_entryHI` e `pte_entryLO` in modo che contengano l'indirizzo logico dei frame del dispositivo Flash a cui è associato lo U-proc tramite l'`ASID` e l'indirizzo fisico mantenuto nel campo `pte_entryLO` sia invalido così da attivare un Page Fault nel momento dell'accesso.

Dopo il completamento del ciclo, si inizializza l'ultimo frame della Page Table in modo che mantenga l'indirizzo del frame rappresentante lo Stack Pointer per l'U-proc, all'indirizzo logico `0XBFFFF000`.

Una volta terminata la funzione `initPageTable()` viene finalmente creato il processo attraverso la SYSCALL 1. L'esito di tale chiamata viene mantenuta nella variabile `createStatus`, che nel caso di esito negativo comporta la cancellazione del processo iniziale e, di conseguenza, di tutti i processi generati, con conseguente `HALT` del sistema.

1.3 Terminazione del processo iniziale

Una volta terminata la funzione `initUproc()`, il processo iniziale ha terminato il suo compito. Viene dunque bloccato dietro ad un particolare semaforo, `masterSem`, il quale viene sbloccato ogni qual volta un processo termina in modo regolare o no. Ciò viene fatto così che il ciclo `for` che racchiude la SYSCALL 3 eseguita dal primo processo possa essere ripetuta ogni volta che un processo termina, e dopo la terminazione dei tutti i U-proc il processo iniziale possa chiamare la SYSCALL 2 in modo da terminare l'esecuzione e attivare un `HALT`.

2 TBL Refill Handler e Pager

La funzione `uTLB_RefillHandler()`, definito nel file `exceptions.c`, viene impostato nel Pass Up Vector in modo da essere eseguito nel caso di un TLB Refill Event, ovvero l'eccezione che viene alzata nel caso si provi ad accedere ad un indirizzo di memoria logico il quale non possiede una entry nel TLB.

In modo analogo, se durante l'accesso ad un indirizzo di memoria che richiede traduzione, la entry del TLB corrispondente è invalida o non modificabile, il controllo viene diretto verso la funzione `Pager()` impostato dal processo iniziale come gestore dei Page Fault per ogni processo creato.

2.1 Azioni nel caso di TLB Refill Event

La prima azione eseguita dal TLB Handler è l'estrazione della entry corrispondente all'indirizzo richiesto dalla Page Table del processo interrotto. Ciò viene fatto controllando il registro `entry_hi` salvato nel BIOSDATAPAGE, acceduto attraverso la variabile `currentState`.

La entry della Page Table viene puntata dalla variabile `pagTblEntry`, il quale è usato come parametro per invocare le funzioni `setENTRYHI()` e `setENTRYLO()`. Viene dunque invocata la funzione `TLBWR()` per scrivere il contenuto dei registri `entryHi` e `entryLo` del `CP0` in una entry casuale del TLB.

Infine, si restituisce l'esecuzione al processo corrente invocando `LDST()`.

2.2 Azioni nel caso di Page Fault

La funzione `Pager()` viene impostato dal processo iniziale come Program Counter delle eccezioni Page Fault nella struttura di supporto per ogni processo figlio.

Tale funzione viene eseguita nel caso venga lanciata una eccezione con codice 1, 2 o 3. Infatti, in tal caso il `kernelExcHandler()` invocherà la funzione `passUp_OrDie()` con paramento `PGFAULTEXCEPT`. Se il processo colpevole sarà in possesso di una struttura di supporto, verrà ceduto il controllo alla funzione `Pager()` con la chiamata di `LDCXT()`.

2.3 Gestione del Pager

Quando l'esecuzione giunge alla funzione `Pager()`, la prima operazione è ottenere la struttura di supporto tramite l'apposita SYSCALL salvandola nella variabile `sPtr`, in modo da determinare la causa dell'eccezione. Tale causa viene assegnata alla variabile `exCause` e, nel caso corrisponda ad una *TLB Mod* (con codice di eccezione 1), viene artificialmente lanciata una Program Trap invocando una SYSCALL con codice maggiore di 13. La gestione della Program Trap è spiegata nel capitolo 3.

Se l'eccezione non fosse dovuta a una *TLB Mod*, si opera una P-operation sul semaforo della Swap Pool al fine di avere una mutua esclusione, così da non riscontrare problemi di coerenza.

Ottenuto l'accesso alla Swap Pool Table, possiamo avviare il cambio di pagina. Per poter proseguire tuttavia, dobbiamo ottenere 3 informazioni:

- `excPageNo`: in tale variabile viene salvato il numero del blocco della Page Table rappresentate l'indirizzo che ha causato il Page fault. Viene estratto dall'`entry_hi` dello stato salvato in `exState`.
- `frameAddr`: mantiene l'indirizzo fisico del frame della Swap Pool dove scrivere il blocco richiesto. Viene passato come puntatore alla funzione `pagingAlgo()` in modo da essere aggiornato.
- `poolTablePtr`: si tratta di un puntatore a `swap_t` a cui viene assegnata la entry della Swap Pool Table corrispondente al frame estratto. Viene aggiornato con il parametro di ritorno di `pagingAlgo()`.

2.4 Algoritmo di Paging

L'algoritmo per la selezione del frame dalla Swap Pool è contenuto nella funzione `pagingAlgo()`, definito nel file `vmSupport.c`. Tale algoritmo sfrutta la variabile `frameNo`, inizializzato nella funzione `initSwapPool()` all'intero -1 per selezionare un frame della Swap Pool in base ad un particolare criterio.

Concettualmente, la Swap Pool è stata suddivisa in 2 parti di egual dimensione: frame dedicati al codice e frame dedicati allo stack. Ciò è stato fatto in seguito ad alcune considerazioni:

1. Abbiamo osservato che un processo di test, per eseguire, necessita di 2 frame di memoria, uno per il blocco del codice e uno per lo Stack. Essi devono essere contemporaneamente presenti nella Swap Pool.
2. Con un algoritmo Round-Robin che assegna sequenzialmente i frame della Swap Pool, ci possono essere situazioni in cui il frame scelto per contenere lo stack di un processo X corrisponda al frame contenete il codice del processo X stesso e viceversa.
3. Se il numero di frame della Swap Pool è minore o uguale al numero dei processi in esecuzione, questo può comportare l'impossibilità di eseguire un processo a causa di continue sostituzioni tra il blocco di codice e il blocco di stack dello stesso processo.

Dedicando frames della Swap Pool per specifici tipi di blocco, si può evitare questa situazione, permettendo di eseguire tutti gli 8 processi di test con una Swap Pool costituita da almeno 2 frame.

Notiamo tuttavia che per processi di grande dimensione, che richiedono diversi frame per il codice, tutti contemporaneamente presenti in memoria, sarebbe adeguato ridimensionare le due parti in modo che i blocchi dedicati al il codice siano proporzionalmente maggiori rispetto a quelli dedicati allo stack.

Inoltre, osserviamo che la scelta del frame sarebbe potuta avvenire impiegando una funzione di hash, ma questa soluzione avrebbe solamente mitigato il problema anziché eliminarlo.

Abbiamo dunque deciso di sacrificare in termini di velocità e prestazione, in modo da consentire l'esecuzione di diversi processi anche con una RAM ridotta.

2.5 Implementazione dell'Algoritmo di Paging

Come detto precedentemente, l'algoritmo di paginazione utilizza la variabile `frameNo` inizializzata a -1. Tale variabile deve sempre mantenere un valore compreso tra -1 e `(POOLSIZE/2)-1` poiché esso viene impiegato per il calcolo dell'indirizzo del frame della Swap Pool e l'indice associato nella Swap Pool Table.

Successivamente, viene controllato il secondo parametro passato alla funzione `pagineAlgo()`, la variabile `blockNo`, il quale indica l'indice all'interno della Page Table che rappresenta il blocco di memoria richiesto. A seconda del valore mantenuto da `blockNo`, distinguiamo tra frame di codice e frame dello stack. Nel primo caso, l'offset per individuare il frame della Swap Pool viene calcolato con la formula `(0x1000 * (frameNo % (POOLSIZE / 2)))`, in modo che il frame scelto ricada sempre nella metà inferiore dei frame della Swap Pool. Nel secondo caso l'offset viene calcolato con `(0x1000 * ((frameNo + (POOLSIZE / 2)) % POOLSIZE))`, e di conseguenza il frame individuato sarà contenuto necessariamente nella metà superiore della Swap Pool.

In entrambi i casi, l'indirizzo fisico individuato viene salvato all'indirizzo puntato dal puntatore `frame`, in modo che esso sia accessibile dalla funzione chiamante.

Infine viene restituito il puntatore alla entry del `swapPoolTable[]` che identifica il frame di memoria della Swap Pool.

2.6 Sostituzione del Frame della Swap Pool

Individuata la pagina, si verifica se questa è attualmente in utilizzo e, in tal caso, si procede con la sostituzione. Per fare ciò, si controlla il valore mantenuto dal campo `sw_asid` dell'entry della `swapPoolTable[]` associata al frame scelto. Se tale valore è diverso da -1, significa che il frame contiene dati di un qualche processo, ed è necessario ricopiarli nel suo backup device (nel nostro caso i dispositivi Flash stessi) in modo da mantenere le eventuali modifiche.

L'effettivo meccanismo tramite cui avviene la ricopiatura è gestito dalla funzione `RWFlash()`, il quale è esplorato nel paragrafo successivo. Tuttavia, prima di poter chiamare tale funzione, è necessario effettuare alcune operazioni preliminari.

Innanzitutto, è necessario disabilitare le interruzioni cambiando lo stato del processore attraverso la chiamata della funzione `setSTATUS()`. Si prosegue invalidando la entry nella Page Table del processo e verificando se tale entry è presente nel TLB. In caso positivo è necessario aggiornare il TLB invalidando anche la sua entry in modo che venga attivato un Page Fault nel caso vi si acceda.

È dunque possibile riabilitare le interruzioni ed estrarre il numero del blocco della Page Table da aggiornare e l'indirizzo fisico del frame da ricopiare, salvandoli rispettivamente nelle variabile `blockToUpload` e `PFNtoUpload`. Tali valori sono necessari per effettuare la chiamata della funzione che performa effettivamente la ricopiatura dalla memoria RAM nel dispositivo Flash, `RWflash()`.

2.7 `RWflash()`: Scrittura e Lettura dal Flash Device

Come accennato precedentemente, la scrittura e lettura da Flash Device è eseguita dalla funzione `RWflash()`. Tale funzione viene impiegata in due situazioni:

1. Ricopiatura del frame di memoria della Swap Pool verso il dispositivo Flash. Viene eseguito nel caso il frame scelto dall'algoritmo di Paging è già occupato.
2. Ricopiatura del blocco di memoria del dispositivo Flash contenente l'indirizzo che ha causato il Page Fault verso la memoria RAM. Avviene ogni qual volta è segnalata un Page Fault.

La prima situazione è un'eventualità. Se il frame della Swap Pool non è occupato, si procede direttamente alla seconda situazione, che è una chiamata certa ogni volta che il controllo giunge al Pager.

La funzione `RWflash()` richiede 4 parametri per operare:

- `BlockNo`: Il numero del blocco nella page all'interno della Page Table del processo.
- `RamAddr`: L'indirizzo fisico del frame della Swap Pool da cui si vuole leggere/scrivere
- `FlashDevNo`: un intero che indica il numero di Flash device.
- `Flag`: un intero che indica l'operazione da eseguire, se di lettura o scrittura.

La prima operazione eseguita è l'ottenimento della mutua esclusione della risorsa attraverso una P-operation sul rispettivo semaforo del livello di supporto.

Successivamente, grazie alla funzione `getDevReg()`, si salva l'indirizzo dei registri del device nella variabile `devreg`. Si inserisce poi nel registro `data0` l'indirizzo di frame dove si vuole operare. Si procede disabilitando gli interrupt in modo da poter bloccare il processo nel semaforo del dispositivo prima che giunga l'interruzione dallo stesso. Si scrive dunque nel registro `command` il comando che si vuole eseguire, utilizzando il parametro `flag` per distinguere tra lettura o scrittura, per poi chiamare la SYSCALL `IOWAIT` in modo da attendere la fine dell'elaborazione del comando e la restituzione del controllo al processo bloccato.

Una volta che il processo attuale è nuovamente caricato nel processore, l'esecuzione riprenderà dall'istruzione successiva alla SYSCALL 5. Vengono quindi riabilitati gli interrupt e si prosegue verificando l'esito della richiesta di lettura/scrittura. Nel caso non ci sia stato esito positivo, si chiama una particolare System Call avente come parametro un valore maggiore di 13. Ciò attiva

una Program Trap ma, come si vedrà nel capitolo successivo, il Trap Handler effettua uno speciale controllo sul primo parametro della SYSCALL con l'obbiettivo di determinare se il processo colpevole del Program Trap detiene in quel momento il semaforo `swpaPoolSem`. Per indicare ciò, il processo deve effettuare un SYSCALL con primo parametro il valore 20. Come conseguenza, prima di eliminare il processo il Trap Handler libererà il suddetto semaforo.

Se invece l'operazione di lettura/scrittura è andata a buon fine, si effettura una V-operation sul semaforo del livello di supporto associato al Flash device per poi riprendere l'esecuzione nel Pager.

2.8 Operazioni del Pager dopo l'aggiornamento della Swap Pool

Dopo la chiamata (le chiamate) di `RWflash()`, il Pager effettua delle ultime operazioni prima di restituire il controllo alle istruzioni del processo.

Viene innanzitutto aggiornata la entry della Swap Pool Table associata al frame di memoria appena scritto. Vengono dunque disabilitati gli interrupt in modo da aggiornare il Page Table del processo e la entry del TLB. Il secondo viene fatto impiegando una specifica funzione, `updateTLB()`.

Questa funzione richiede come parametro la entry della Page Table aggiornata poco prima e, grazie ad essa, effettua una chiamata della funzione `TLBP()` in modo da ricercare l'eventuale entry TLB già presente. A seconda del risultato di tale chiamata (il quale è scritto nell'ultimo bit del registro `INDEX` del processore), viene effettuata una chiamata di `TLBWI()` o di `TLBWR()`. Il primo aggiorna la entry del TLB associata al frame che è giù presente, la seconda aggiorna una entry scelta casualmente.

Infine, terminata l'esecuzione di `updateTLB()`, vengono riabilitati gli interrupt e si libera l'accesso al semaforo della Swap Pool Table. Come ultima operazione si prosegue l'esecuzione delle istruzioni del processo, che ora avrà a disposizione gli indirizzi di memoria che avevano causato il Page Fault.

3 Trap Handler e Syscall del Livello di Supporto

La funzione `generalSupHandler()`, definito nel file `sysSupport.c`, si occupa della gestione dei Program Trap e delle SYSCALL con parametro maggiore di 8. Il controllo giunge ad esso a seguito della chiamata della funzione `passUp_orDie()` con parametro `GENERALEXCEPT`.

3.1 La funzione `generalSupHandler()`

La funzione `generalSupHandler()` opera in modo simile al `kernelExcHandler()` del kernel, cioè si occupa di smistare la gestione dell'esecuzione verso la corretta funzione. Tuttavia, a differenza del secondo, il primo si occupa di meno tipi di eccezioni.

Giunti alla funzione, è necessario determinare se l'esecuzione deve procedere verso il gestore delle SYSCALL di supporto oppure è necessario invocare il Trap Handler. Per fare ciò è sufficiente richiedere la struttura di supporto del processo corrente grazie alla SYSCALL `GETSUPPORTPTR` ed esaminare la causa dell'eccezione, salvandolo in `genExCause`.

Nel caso tale variabile contenga il valore 8, verrà chiamata la funzione `supportSyscallHandler()`, in caso contrario la funzione `trapHandler()`.

Osserviamo che il puntatore che mantiene la struttura di supporto, cioè la variabile `sPtr`, viene passata come parametro a quasi tutte le funzioni nel file `sysSupport.c`. Questo poiché dichiarare tale variabile come globale comporta grossi problemi nel caso di *Context Switch*.

3.2 Gestore delle Syscall del Livello di Supporto

La funzione `supportSyscallHandler()` si occupa di determinare quale SYSCALL del livello di supporto è stato richiesto del processo.

Viene prima di tutto incrementato il Program Counter in modo da proseguire l'esecuzione dall'istruzione successiva alla chiamata della SYSCALL. Si usa dunque il primo parametro passato

con la chiamata all'interno di uno switch statement per continuare la gestione verso la corretta funzione che esegue l'operazione richiesta.

3.2.1 SYSCALL 9: Terminate

La SYSCALL 9 è gestita dalla funzione `terminate()`. Si occupa di chiamare la SYSCALL 2, ma prima di fare ciò, invalida tutti le entry del Swap Pool Table che occupa il processo, in modo che un successivo U-proc che debba ricopiare il suo contenuto di tale frame nel Flash device.

Viene inoltre eseguito una V-operation sul `masterSem` in modo che il processo iniziale venga sbloccato e reinserito nella `readyQ`.

3.2.2 SYSCALL 10: Get TOD

La SYSCALL 10 è gestita dalla funzione `getTOD()`. Esso si preoccupa di scrivere nel registro di ritorno il Time Of Day, per poi restituire il controllo al processo.

3.2.3 SYSCALL 11: Write To Printer

La SYSCALL 11 è gestita dalla funzione `Write_To_Printer()`. Esso richiede 3 parametri:

- `string`: contiene il puntatore all'indirizzo di memoria virtuale dove comincia la stringa da scrivere sul dispositivo Printer.
- `len`: si tratta della lunghezza della stringa da stampare.
- `sPtr`: è il puntatore alla struttura di supporto del processo che ha richiesto la SYSCALL.

La prima operazione effettuata è la verifica di due vincoli:

- Gli indirizzi da cui la stringa è letta devono rientrare entro i limiti degli indirizzi logici del processo.
- La lunghezza della stringa non può essere minore di 1 carattere o eccedere i 128 caratteri.

Infrangere uno dei due vincoli comporta immediatamente la terminazione del processo.

Successivamente, viene estratto il registro del dispositivo Printer associato al processo chiamando la funzione `getDevReg()` utilizzando parametri dalla variabile `sPtr`. Si procede poi richiedendo l'accesso al dispositivo effettuando una P-operation sul rispettivo semaforo del livello di supporto.

Finalmente, l'esecuzione entra in un ciclo `while` che viene ripetuto fino a quando non si giunge al termine della stringa. Questo ciclo si occupa scrivere un carattere della stringa nel campo `data0` del registro. Vengono quindi disabilitati gli interrupt per gli stessi motivi spiegati nel paragrafo 2.7, in modo da poter dare il comando `PRINTCHR` e bloccare il processo nell'adeguato semaforo del dispositivo Printer, salvando l'esito dell'operazione nella variabile `opStatusP`. Ad ogni ripetizione del ciclo, viene controllato questa variabile che, in caso di fallimento dell'operazione, comporta l'uscita dal while statement. Se invece l'operazione ha avuto successo, si procede al carattere successivo incrementando la variabile `string`.

All'uscita dal ciclo, si verifica lo status dell'ultima operazione. Se esso indica un successo, significa che tutte le operazioni hanno avuto successo e il ciclo while è terminato a causa del suo vincolo. Viene dunque scritta la lunghezza della stringa stampata nel registro di ritorno, come

richiesto dalle specifiche. Se invece `opStatusP` è diverso da 1, significa che l'ultima operazione non ha avuto successo e l'uscita dal ciclo è avvenuto in modo anomalo. Dunque si scrive l'inverso dello status nel registro di ritorno.

Infine si libera l'accesso al semaforo del dispositivo Printer e si restituisce l'esecuzione al processo attraverso la chiamata di `LDST()`.

3.2.4 SYSCALL 12: Write To Terminal

La SYSCALL 12 è gestita dalla funzione `Write_To_Terminal()`. In molti aspetti, questa funzione è simile alla gestione della SYSCALL 11. Richiede gli stessi parametri ed effettua gli stessi controlli su di essi, ed estrae in modo analogo il registro del terminale.

Tuttavia, una maggiore complessità deriva dal fatto che il registro del terminale è suddiviso in 2 sezioni, ed è impiegato sia per la gestione della scrittura e della lettura. Di conseguenza, il carattere da trasmettere e il comando vengono assegnati nello stesso campo del registro.

Per il resto, questa funzione opera in modo uguale a quella descritta nel paragrafo precedente.

3.2.5 SYSCALL 13: Read From Terminal

La SYSCALL 12 è gestita dalla funzione `Read_To_Terminal()`. Diversamente dalle precedenti, la gestione di questa SYSCALL richiede due parametri:

- `stringAddr`: è un puntatore all'indirizzo da cui cominciare a scrivere i caratteri che vengono letti dal terminale.
- `sPtr`: si tratta del puntatore alla struttura di supporto del processo che ha richiesto la SYSCALL.

Viene effettuato la verifica del vincolo precedente, ovvero che gli indirizzi dove verranno scritti i caratteri rientrino nell'address space del processo

In modo analogo alle due SYSCALL precedenti, si estrae il registro del terminale associato al dispositivo. Ma diversamente da prima, bisogna dichiarare una ulteriore variabile `count` in modo da calcolare il numero di caratteri ricevuti e, se tutto procede positivamente, scriverlo nel registro di ritorno al termine del ciclo.

Dopo aver richiesto l'accesso esclusivo al terminale, il codice contenuto nel ciclo si occupa di impartire il comando di lettura nel campo `recv_command`, per poi bloccare il processo sul semaforo associato al dispositivo.

Con il medesimo meccanismo precedente, si procede ad uscire dal ciclo nel momento in cui un operazioni fallisce. Ad ogni ripetizione del ciclo, la variabile `count` viene incrementata.

Successivo al while statement, si effettuano verifiche analoghe alle SYSCALL precedenti, per poi restituire il controllo al processo.

3.3 Gestione dei Program Trap

I Program Trap sono gestiti dalla funzione `trapHandler()`, il quale viene invocato in due diverse situazioni all'interno di `sysSupport.c`:

- All'interno di `generalSupHandler()`, nel caso la causa dell'interrupt non sia una SYSCALL.
- All'interno del gestore delle SYSCALL di supporto nel caso il numero della System Call sia maggiore di 13.

In entrambi i questi casi, la funzione `trapHandler()` libera tutti i semafori del livello di supporto occupati dal processo corrente.

Successivamente, si verifica se il numero di SYSCALL richiesto corrisponde al valore speciale `BLOCKEDTRAP`, il quale indica al Trap Handler che la funzione colpevole detiene in quel momento il semaforo `swapPoolSem`. In tal caso, si libera la mutua esclusione e si procede invocando il gestore della SYSCALL 9.

Correzioni e cambiamenti delle fasi precedenti

Ci sono stati segnalati 2 errori contenuti nello svolgimento della fase2 del progetto PandOS:

- Il valore restituito da SYSCALL 6 non era sufficientemente preciso, poiché non veniva sommato il tempo intercorso per la gestione della SYSCALL stessa.
- Non è stato effettuato il calcolo del time scale durante l'impostazione del Processor Local Timer.

Tali errori sono stati affrontati nei seguenti modi:

- È stata dichiarata una variabile globale `excTOD`, il quale viene impostato al TOD appena il controllo giunge alla gestore delle eccezioni nel file `exceptions.c`. Viene poi dichiarata un'ulteriore variabile locale, `currTOD`, nella funzione di gestione della SYSCALL 6 `get_CPU_time()`. Quest'ultima variabile viene inizializzata al TOD e, sottraendo ad essa `excTOD`, è possibile ottenere il tempo intercorso per la gestione di questa SYSCALL.
- Il calcolo del Time Scale viene ora effettuato dallo Scheduler durante la chiamata di `setTIMER()`.

Oltre a queste correzioni, è stato necessario modificare la funzione `terminalHandler()` nel file `interrupt.c`, che si occupa della gestione degli interrupt provenienti dai dispositivi terminali. Infatti, a causa del meccanismo della gestione, veniva effettuato una V-operation su entrambi i semafori del terminale nonostante l'interrupt provenisse da una sola line. Di conseguenza, nel caso di stampe e letture successive da uno stesso terminale, diverse operazioni venivano saltate poiché i processi non venivano bloccati presso il semaforo utilizzando una SYSCALL 5.

Una volta vincolato la V-operation alla linea da cui proviene l'interrupt, le interazioni con il terminale hanno avuto successo.

Conclusioni

La realizzazione della Fase 3 del progetto PandOS ha richiesto meno gestione di basso livello e ha dato spesso l'occasione di impiegare le funzioni scritte nelle fasi precedenti come base per il funzionamento.

Le diverse funzioni sono state così suddivise nei diversi file:

- `initProc.c`: contiene la funzione `instantiatorProcess()`, che viene eseguito al primo processo preparato dal kernel, dove vengono dichiarate e inizializzate le diverse strutture per tutti i processi e inizializzati i semafori e le strutture dati del livello di supporto.
- `vmSupport.c`: sono qui dichiarati i semafori e le strutture dati del livello di supporto. Inoltre sono definite le funzioni di inizializzazione chiamate dal `instantiatorProcess()`, il pager e le funzioni che implementano l'algoritmo di paging e l'aggiornamento del TLB.

- `sysSupport.c`: Qui è definito il gestore delle eccezioni del livello di supporto, il gestore delle System Call di supporto, il Trap Handler, oltre a tutte le funzioni che eseguono le effettive operazioni richieste dalle System Call.

Il Makefile è stato realizzato a partire da quello della fase 2 in modo da compilare anche i diversi file della fase 3.