μPandOS

Patrick Alfieri - 0001030013

September 2024

Contents

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 é C ed i programmi vengono compilati attraverso un makefile. Inoltre, l'emulatore di riferimento per questa implementazione é uMPS3.

2 Fase 1 - Manager delle code

Il primo passo é 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 é 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 é 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 funzionalitá 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 μPandOS 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 é 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` é 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

μPandOS 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 è 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`, cioé 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 giá 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 sará 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) 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` é stato castato alla struttura `ssi_do_io_PTR`, in modo che contenga `deviceCommand` per verificare su che tipo di dispositvo é 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 é identificato da una linea di interrupt è i dispositivi, valutati in ordine di prioritá, 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 é posto a 1, allora il dispositivo i sulla linea $j + 3$ ha un interrupt pendente.

É 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 sará pressoché uguale, ma cambierá la struttura.

Questo sistema di check sulle linee é 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-teminali → 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 é 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 é 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 é 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 é associato al codice 6, ossia la costante `GETSUPPORTPTR`.

3.4.7 GetProcessID

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

Questo servizio é 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 μPandOS: 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 é 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 é posto a 100ms, quindi ogni 100ms verrá 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 é 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 é aggiunto un campo `int` alla struttura del PCB, che indicasse su quale device é stato bloccato. Quindi, si fa uso della funzione `unlockPCBDevNo(devNo, queue)` per cercare il PCB bloccato sul device `devNo` nella coda `queue` e quindi sbloccarlo. Dunque, si salva il valore corrispondente nella variabile `status` 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 é 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 sará nullo, ma conterrá lo status, ossia il risultato dell'operazione di I/O che ne indicherá 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'é 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 `NETWORKINTERRUPT, CAUSEREGSIZE`...

4 Fase 3 - Supporto

L'obiettivo del Livello Supporto in μPandOS é realizzare un'ecosistema di funzioni per rendere possibile l'esecuzione dei processi utente (UPROCs) nel sistema operativo. Ció si basa in modo diretto sulle primitive definite nel Livello 1 e sui servizi richiedibili al SSI definiti nel Livello 2.

In uMPS, questi processi eseguiranno ognuno nello stesso spazio di indirizzamento e sará dunque fondamentale gestire un sistema di traduzione degli indirizzi logici associati in modo che il kernel possa operare con indirizzi di memoria fisici. A tale scopo, questa fase é stata organizzata in cinque moduli il cui contenuto verrá descritto nelle prossime sezioni.

4.1 Inizializzazione dei processi

Il modulo `initProc.c` contiene in primis la definizione di `test` che, nel contesto del Livello Supporto, sostituisce quella fornita esternamente (in `p2test.c`) e il cui entry point si trova comunque nel Nucleo, come specificato nel Livello 2.

Lo scopo di questa nuova funzione di testing, é inizializzare e creare i vari e necessari PCB per eseguire correttamente un certo numero `UPROCMAX` di UPROCs. Concretamente, questi processi utente non sono altro che routine definite nella directory `testers`, a cui il Nucleo fornisce servizi in modo da garantirgli una corretta esecuzione e terminazione.

Per potersi considerare concluso, `test` dovrá attendere la terminazione (con il servizio dell'SSI `termProcess`) di tutti i PCB che creerá e infine terminare se stesso.

4.1.1 initSwapStructs

All'interno dello spazio di indirizzamento degli UPROC, detto `kuseg`, essi si distinguono per un'identificatore speciale, detto `asid` contenuto nel registro del processore EntryHI. Per supportare la traduzione degli indirizzi degli UPROC, si utilizza una Swap Pool, ossia un set di frames riservati per mantenere le pagine virtuali. Per ogni entry della tabella, vengono inizializzati i campi per indicare che attualmente non risiede alcuna pagina, non associata a nessun UPROC.

La Swap Pool Table presenta un numero di entry pari a `POOLSIZE`, ossia il doppio di `UPROCMAX`.

Si fa notare come questa funzione, come suggerito dalle specifiche, sia definita nel modulo `vmSupport.c` ma descritta qui per un maggior senso di omogeneitá.

4.1.2 mutex

L'accesso alla Swap Pool Table é considerato sezione critica dato che gli UPROCs concorrono per effettuare traduzione e allocare pagine nei frame di memoria, pertanto deve essere mediato con un meccanismo di sincronizzazione.

Questo compito é dedicato a un PCB speciale `mutexSender`, che implementa un sistema di mutua esclusione basato sul message passing con le SYSCALL `send` e `recv` definite nel livello precedente.

L'idea é che `mutexSender` aspetta indefinitamente richieste di "guadagno" di mutua esclusione da parte degli UPROC, caso in cui quest'ultimo verrá marcato come detentore della mutex attraverso un PCB apposito, altrimenti dovrá attendere il rilascio da parte di un altro UPROC.

Si fa notare come questa funzione sia definita nel modulo `utils.c` in quanto maggiormente coerente con le altre funzioni ausiliarie che si trovano al suo interno.

4.1.3 initUproc

Vengono inizializzate `UPROCMAX` strutture di tipo `state_t` e poste in un array. Ció é necessario, ma non sufficiente, per richiedere l'effettiva creazione del singolo processo utente al SSI mediante il servizio `createProcess`.

Gli UPROC saranno intuitivamente posti in user-mode, con interrupt attivi e PLT abilitato. Vengono assegnati anche `asid` e Stack. Il Program Counter viene posto all'inizio dell'area .text (`UPROCSTARTADDR`), in cui si trovano i compilati degli eseguibili in modo contiguo.

4.1.4 initSupportStruct

Il secondo elemento necessario per la creazione é fornire una struttura di supporto, di tipo `support_t`. Esse conterranno tutti i campi necessari a supportare le tipologie di eccezioni nel livello supporto: infatti vengono associati al Program Counter gli handler per traduzione indirizzi (`pager`) e generali.

Per lo Stack Pointer si considerano due frames (di dimensione `PAGESIZE`) della RAM installata.

Si fa notare che lo stack in uMPS cresce verso il basso a partire dall'indirizzo di tetto `RAMTOP`, il quale fungerá di indirizzo di riferimento per l'allocazione dei frames. Non é peró possibile utilizzare i primi due frame, in quanto occupati da `ssi_pcb` e dall'entry point `test` nel Nucleo, perció si alloca partendo dal terzo frame.

Quando avverrá un page fault che chiamerá in causa una ricerca nel Translation Lookaside Buffer (TLB) e conseguente traduzione degli indirizzi logici, si fará riferimento alle TLB entries. Ogni UPROC dovrá essere fornito quindi mediante la sua struttura di supporto di un array, chiamato Page Table (dimensione `MAXPAGES`), le cui entries saranno del tutto uguali alle TLB entries e utilizzate per riempire quest'ultime. In tal senso, vengono inizializzati i 64bit delle entries in due porzioni: EntryHI, che contiene il Virtual Page Number e l'`asid` e EntryLO, che contiene il Physical Frame Number e i bit di controllo (valid, dirty, ...).

4.1.5 initSST

Il System Service Thread é un tipo di interfaccia che offre servizi in maniera similare all'SSI, in modo tale che ogni UPROC, che verrá appunto creato dal processo SST associato, possa richiederli. Questo significa che esistono `UPROCMAX` PCB di tipo SST che creano i loro PCB figli UPROC, ognuno dei quali domanderá un servizio al genitore mediante message passing. Le strutture di supporto definite in precedenza verranno utilizzate per creare i vari SST e poi ereditate dagli UPROC figli. Lo stato del processore di ogni SST é impostato in kernel-mode, con interrupt e PLT attivi.
In questa funzione vengono anche lanciati gli `UPROCMAX` SST, quindi é possibile considerarlo come il punto di lancio per l'esecuzione del test.

4.1.6 initPeripheralProc

Le routine degli UPROC si basano sullo scrivere caratteri su due tipi di device, terminali e stampanti. Siccome ogni UPROC dovrá operare in modo esclusivo sul proprio device associato, vengono definiti `UPROCMAX` PCB aggiuntivi per entrambe le tipologie per proteggere il sistema da situazioni spiacevoli. Questi eseguono, in maniera lontanamente similare all'SSI, un procedimento finito che provvederá a stampare la stringa inviata dalla routine degli UPROC all'SST nel device interessato attraverso operazioni di `DOIO`. Questa procedura é definita per-device e attraverso dei puntatori a funzione, assegnati al Program Counter di questi processi.

Essa verrá descritta nelle sezioni successive.

4.2 Eccezioni

Come nel Livello Nucleo, anche qui si presentano eccezioni ed handler dedicati.

Esso ne presenta due tipologie: eccezioni TLB, il cui handler é definito nel modulo `vmSupport.c` e non, i cui handler sono definiti nel modulo `sysSupport.c`. In questa sezione verrá trattato quest'ultimo caso.

A differenza del gestore di eccezioni del Nucleo, `supExceptionHandler` non troverá lo stato del processore con cui riconoscere il singolo caso attraverso la macro `EXCEPTION_STATE`, bensí nella struttura di supporto del processo corrente passata al livello superiore mediante la `passUpOrDie`, all'indice `GENERALEXCEPT`. Quindi, dopo aver reperito tale struttura attraverso la funzione `getSupStruct`, il codice con cui determinare l'handler si ottiene in modo uguale a come visto nel Livello precedente.

4.2.1 Syscall

Nel Livello Supporto, sono definite le Syscall (codice 8) "user-mode" che fungono fondamentalmente da wrapper per `send` e `recv`, perció non si tratta di implementazioni particolari.

Al momento della chiamata, nuovamente, nel registro `a0` si troverá il codice per distinguere tra le tipologie, mentre nei successivi rispettivamente indirizzo del destinatario e payload. Abbiamo perció due casi:

- **SENDMSG** (USYS1). Se `a1` contiene `PARENT`, allora il processo utente invia un messaggio, contenuto in `a2`, al proprio genitore. Se cosí non fosse, in `a1` troviamo semplicemente il destinatario.
- **RECVMGS** (USYS2). Wrapper elementare per una `recv`.

4.2.2 Program Trap

Un eventuale Program Trap catturata nel Livello Precedente, viene gestita in questo handler. L'idea é quella di catturare situazioni di eccezioni inaspettate terminando, attraverso l'SSI, il processo chiamante e tutta la sua progenie.

Prima di fare ció, vengono liberati tutti i frame che il PCB occupava nella Swap Pool Table in modo da prevenire operazioni non volute sui device. Nel caso a terminare sia il detentore della mutex viene anche inviato un messaggio al processo `mutexSender` in modo da rilasciarla, cosicché non si entri in deadlock.

4.3 TLB e Pager

Le eccezioni di tipo TLB vengono invece catturate dal `pager`, ossia l'handler che esegue lo swap-in/swap-out delle pagine in memoria, attraverso la traduzione degli indirizzi logici.

Il TLB non é altro che una cache efficiente per la traduzione degli indirizzi, che: nel caso di TLB Hit, significa che la pagina virtuale é presente ed é quindi possibile subito effettuare una traduzione da indirizzo logico a fisico, mentre un eccezione TLB Miss, implica che é necessario un evento di TLB Refill per caricarla. Lo scheletro del `uTLB_RefillHandler` era giá presente, ma é stato ridefinito a questo livello (sebbene sia rimasto nella stessa posizione, in `/phase2/initial.c`).

La nuova implementazione prevede di ricavare lo stato del processore (come fatto nel Nucleo, con la macro `EXCEPTION_STATE`) e quindi il numero della pagina che concerne il Refill attraverso `ENTRYHI_GET_VPN`. Si nota come é stato necessario porre un bound check (a 31), in quanto la macro produceva valori fuori dal range possibile. Dunque, si settano le due porzioni della TLB entry dalla page table presente nella struttura di supporto e si pongono nel TLB, infine si carica lo stato dell'istruzione precedente.

Il `pager` gestisce invece i page fault, che possono essere determinati similmente a come fatto per le Syscall, ma nell'indice `PGFAULTEXCEPT`. Nel caso si tratti di una TLB Modification, il processo viene terminato esattamente come si farebbe per una Program Trap.

A questo punto, all'UPROC sará necessario accedere alla Swap Pool Table per allocare una pagina virtuale in uno dei frame in memoria, pertanto sará necessario in primis richiedere la mutex (e quindi eventualmente attendere) e poi determinare il numero della pagina che ha generato un page fault. Quindi, si controlla se ci sono frame in liberi in memoria in cui é possibile piazzare una pagina e abbiamo due casi:

- Se il frame in memoria é occupato, si utilizza un algoritmo di rimpiazziamento FIFO `pick_frame` per vittimizzare una pagina, invalidandola attraverso i bit e aggiornando il TLB per "impedire" agli altri processi di effettuarne una traduzione. É importante notare come queste azioni richiedano atomicitá ed é dunque necessario disattivare (e poi riattivare) gli interrups del processore settando il bit `IECON`.
  Quindi, si procede aggiornando il backing store associato allo specifico UPROC. In μPandOS, un backing store di un UPROC é una forma di memoria secondaria che contiene l'immagine logica completa del processo stesso, ed a questo livello viene trattato come un device flash, su cui quindi sono attuabili operazioni di `DOIO`. Infine, si riprende dal passo descritto di seguito.

- Si effettua un operazione di lettura sul backing store in modo da poter aggiornare coerentemente i contenuti di Swap Pool Table, Page Table (ponendo la pagina come valida e impostando il PFN) e dunque TLB. Queste ultime due operazioni vengono nuovamente eseguite atomicamente.

Infine, il `pager` segnala che l'UPROC detentore della mutex ha terminato con la Swap Pool Table e pertanto puó rilasciare la risorsa nei confronti di un altro UPROC e ricarica lo stato del processore.

4.4 SST

Il modulo `sst.c`, come accennato in precedenza, definisce i servizi elargibili dai processi SST ai figli. La routine eseguita da un pcb SST é fondamentalmente la stessa del SSI: attende una richiesta, la riceve attraverso message passing, determina ed elabora il servizio e fornisce un responso al richiedente.

Prima di ció, ogni SST crea il proprio figlio attraverso il servizio `createProcess` del SSI, facendogli ereditare la propria struttura di supporto. Di seguito, i servizi offerti dall'SST.

4.4.1 GetTOD

Viene ritornato il valore del clock TOD.

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

4.4.2 Terminate

Chiamando questo servizio, l'UPROC richiede la terminazione del SST associato e dei propri figli (quindi anche se stesso).

Ció viene naturalmente fatto attraverso il servizio `terminateProcess` offerto dal SSI. Similmente alla gestione di una trap, anche qui vengono liberati i frame in memoria occupati, reinizializzando le entry nella Swap Pool Table. Non é invece necessario mandare un messaggio a `mutexSender`: quando si richiede questo servizio si suppone che tutte le operazioni che coinvolgono la Swap Pool Table siano giá state portate a compimento.

Ogni PCB SST quando terminerá invierá un messaggio al `test`, che, una volta ricevuti `UPROCMAX` messaggi, provvederá a richiedere la propria terminazione. Oltre che l'SST, si fa in modo di terminare dichiaratamente anche i PCB device associati all'asid corrente che si considera al momento della chiamata, per evitare che possano tornare nella Ready Queue a seguito di qualche `recv` successiva. Ció é stato necessario, poiché le routine particolarmente pesanti nella CPU causavano talvolta comportamenti inaspettati.

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

4.4.3 WriteTerminal

Il servizio permette la stampa di una stringa sul `asid`-esimo device terminale specificato.

Le routine degli UPROC passano tramite messaggio un payload di tipo `sst_payload_PTR`, contenente la stringa da stampare e la sua lunghezza. Dunque, questa funzione non fa altro che inoltrare la richiesta al processo device (`terminalPcbs[asid]`) che viene sbloccato dalla `recv` nella funzione di stampa `printDevice` reperendo la stringa.

Quindi nel dettaglio, la procedura eseguita segue questo protocollo:

1. Il pcb `terminalPcbs[asid]` chiama la funzione di stampa definita in `utils.c` e attende che giunga una richiesta dall'UPROC identificato da `asid` per scrivere una stringa di char sul device terminale numero `asid` (0-7).

2. La routine dell'UPROC fa giungere una richiesta.

3. L'UPROC, una volta completate le operazioni di traduzione, fa una richiesta al SST mediante il servizio `writeTeminal`, che passa il payload alla funzione di stampa.

4. La funzione di stampa dunque determina l'indirizzo del device register su cui scrivere partendo dal primo e andando avanti di 4 bytes, ossia la dimensione di ogni registro. Da questo si ricava anche il campo `command`.

5. Si itera su ogni carattere della stringa finché non si incontra il null-terminatore `'\0'` e si costruisce il payload per la singola `doio` da richiedere con il campo command e `value`, determinato da un operazione di `|` tra lo shift a sinistra di 8bit sull'ASCII del carattere e la costante di stampa `PRINTCHR`.

6. Esauriti i caratteri, si manda un messaggio per sbloccare l'SST, che puó quindi fornire il responso all'UPROC.

Questo servizio é associato al codice 4, ossia la costante `WRITETERMINAL`.

4.4.4 writePrinter

Fondamentalmente identico al servizio `writeTerminal`, ma con alcune leggere modifiche dovute al diverso tipo di dispositivo, a partire dal fatto che ci si riferisce naturalmente all'array di PCB `printerPcbs`. In particolare:

- L'indirizzo del primo registro é calcolato diversamente (si usa la costante `PRINTOADDR`), ma viene utilizzato allo stesso modo per calcolare eventuali indirizzi dei device register successivi. Inoltre, qui non c'é bisogno di differenziare tra transm e recv, dunque `command` é calcolato normalmente e il command `value` é semplicemente la costante di stampa.

- uMPS3 prevede che per i dispositivi flash (ossia i backing store) venga usato il campo `DATA1`, mentre le stampanti `DATA0`, che contiene nei primi 8 bit il carattere da trasmettere al device stesso.

Si noti inoltre come si scriverá sempre e comunque sul device stampante numero 7, ma é stato scelto di conservare interamente l'array di PCB per coerenza con la funzione precedente e una maggiore generalizzazione.

Questo servizio é associato al codice 3, ossia la costante `WRITEPRINTER`.

4.5 Utilities e altro

Il modulo `utils.c` contiene funzioni wrapper per richiedere servizi (terminazione, struttura di supporto, ...) all'SSI, compresa la funzione di stampa sui device terminali/stampanti, oltre funzioni che concernono il passaggio di mutua esclusione.

Inoltre, sono state aggiunte definizioni di costanti in `const.h`, come `TLBINVLDM, TERMOADDR,...`

Infine, sono state apportate alcune lievi modifiche nel Livello Nucleo per permettere una gestione piú efficiente della CPU, cosicché le routine ricorsive degli UPROC potessero eseguire correttamente.