Progetto di Sistemi Operativi

A.A. 2011/2012

PKaya

Autori:

Marco Di Nicola

Lorenzo Vinci

Nel seguito di questo documento faremo un sunto dell'implementazione del nostro progetto PKaya.
Procederemo nel passare in rassegna i moduli che costituiscono il progetto, descrivendo per ognuno di essi il funzionamento e le nostre scelte implementative.

Indice:

1. Boot
2. Scheduler
    - 2.1. Inizializzazione
    - 2.2. Assegnamento Processi
    - 2.3. Scheduling
        - 2.3.1. Priorità Processi
3. Interrupts
    - 3.1. Processor Local Timer
    - 3.2. Interval Timer
    - 3.3. Terminali
    - 3.4. Inter-Processors Interrupts
4. System Calls
    - 4.1. La funzione sysHandler
    - 4.2. Implementazione
        - 4.2.1. CREATEPROCESS
        - 4.2.2. CREATEBROTHER
        - 4.2.3. TERMINATEPROCESS
        - 4.2.4. VERHOGEN
        - 4.2.5. PASSEREN
        - 4.2.6. GETCPUTIME
        - 4.2.7. WAITCLOCK
        - 4.2.8. WAITIO
        - 4.2.9. SPECPRGVEC, SPECTLBVEC, SPECSYSVEC
        - 4.2.10. DEFAULT
    - 4.3. Breakpoints
5. Eccezioni
6. Testing
    - 6.1. Priorità
    - 6.2. Scalabilità
    - 6.3. How to use it

1. Boot

In fase di avvio del sistema, la prima operazione fondamentale è quella di creare in RAM una matrice di **state_t**.
Per ogni processore saranno dichiarati gli stati relativi alle OLD/NEW areas, in particolare (seguendo lo stesso ordine con cui sono contigue in ROM quelle per il processore 0) avremo: Interrupts, TLB Exceptions, Program Traps e System Calls.
Il numero di processori con cui si è configurato il sistema sarà indicato dalla costante **NUM_CPU**, il cui valore sarà letto dall'indirizzo di memoria apposito in ROM.
D'ora in poi assumeremo che ogni operazione che cicla su strutture assegnate ai processori abbia questa costante come limite di riferimento.
Non avremo bisogno di impostare gli state_t relativi alle OLD AREA (saranno i processori a farlo, scrivendovi i dati del processo precedentemente in esecuzione), ma per quelli relativi alle NEW AREA svolgeremo tre operazioni:

- Settare lo stato in modo che gli interrupts e memoria virtuale siano disabilitati, kernel mode e processor local timer attivi.
  N.B.: questa configurazione si applicherà, come vedremo in seguito, anche agli state_t per l'esecuzione dello scheduler. In generale a tutto ciò che è eseguito all'interno del nucleo.
- Impostare il Program Counter affinchè punti all'indirizzo di partenza degli handlers appositi (dichiarati come *extern*, in quanto implementati negli altri moduli).
- Settare lo Stack Pointer in modo che, per ogni processore, vada decrementando di FRAME_SIZE a volta, da RAMTOP. Cosi da evitare inconsistenze nella scrittura/lettura su stack nella memoria.

Procederemo poi a inizializzare le strutture dati sottostanti (pcb_t e semd_t), richiamando le funzioni **`initPcbs()`** e **`initASL()`** , implementate nella Phase1.
Setteremo quindi in un ciclo tutti i locks sui semafori su valore FREE.

Vale la pena di spendere due parole per il controllo nell'accesso a variabili condivise fra i processori. Tramite la funzione atomica **`CAS()`,** implementeremo un meccanismo di mutua esclusione nell'accesso a certe zone critiche.

Per massimizzare l'efficienza di questo meccanismo, abbiamo assegnato diverse variabili intere a questo scopo, dividendole per ambito.
Queste sono:

- **prid_Lock**: una per processore. Utilizzate per accedere alle code e ai contatori di ogni processore, dato che le operazioni vengono spesso effettuate da altre cpu (es: `V()` su un semaforo).
- **soft_Lock**: scrittura del contatore di processi bloccati in attesa di I/O.
- **sem_Lock**: uno per ogni semaforo. Utilizzate per proteggere accesso ai semafori nelle `P()` e `V()` , nonché nella funzione `outChildBlocked()`, per eliminare pcb bloccati sulle code dei semafori.
- **pcb_Lock**: allocazione e rilascio dei pcb_t.

Nel seguito di questo documento, non sempre ci riferiremo esplicitamente all'acquisizione e rilascio dei lock.
Inoltre certe operazioni di lettura non sono protette da locks, in quanto non lo abbiamo ritenuto necessario (spesso perché il lock su operazioni precedenti garantiva un ordine "sicuro" nell'eseguirle).

L'ultima operazione di boot consiste nel chiamare la funzione initScheduler(), esplicata nel capitolo successivo.

2. Scheduler

La componente fondamentale del nucleo risiede nello Scheduler.
Procediamo adesso nell'illustrare le scelte progettuali prese nella sua implementazione.
Come prima cosa abbiamo posto particolare enfasi nell'incapsulare il meccanismo, nella sua interezza, all'interno del modulo *scheduler.c*.
Le altre componenti non hanno accesso diretto alle strutture dati usate dallo scheduler, se non tramite le operazioni la cui interfaccia è specificata nell'header relativo.

Queste strutture dati consistono in:
Una serie di code di processi pronti per l'esecuzione, una per processore, usate per gestire l'avvicendamento dei processi (il cui tipo è ridefinito come **PCB_Queue** per evidenziarne la semantica).
Un vettore contenente un riferimento ai PCB in esecuzione (o comunque agli ultimi eseguiti) per ogni processore.
Un contatore dei processi bloccati su operazioni di I/O e più contatori di processi attivi nel sistema, assegnati a ogni processore, utilizzati per implementare una corretta politica di **load balancing** nell'assegnamento dei processi e una semplice **deadlock detection**.
Un vettore di interi che rappresenta lo stato di esecuzione di ogni processore (IDLE, in attesa di interrupts e processi da eseguire, o in esecuzione).
Una serie di variabili intere utilizzate per gestire mutua eclusione tra processori nell'accesso alle strutture dati condivise.
Degli state_t utilizzati per caricare nei registri dei processori i dati necessari ad eseguire il codice dello scheduler e per far eseguire alla CPU0 un "finto processo", qualora non si blocchi su stato IDLE dopo la chiamata alla funzione di ROM WAIT() .

2.1. Inizializzazione

Queste strutture dati saranno propriamente inizializzate mediante la chiamata (eseguita allo startup) alla funzione **`initScheduler()`.**
In particolar modo, lo state_t relativo allo scheduler eseguito da ogni processore avrà il program counter impostato sull'indirizzo di partenza della funzione **`schedule()`,** lo stack pointer per le chiamate a funzione e variabili locali posizionato sullo stesso indirizzo di memoria usato dagli handlers (interrupts, system calls, traps e tlb exceptions) di quel processore. Questo perché è escluso che, per un determinato processore, siano in esecuzione a tempi alterni prima di terminare, scrivendo sulla stessa zona di memoria.
Lo state_t usato per il processo dummy avrà interrupts abilitati, in quanto la sua esecuzione (ridotta ad un semplice loop infinito) è utile solo per permettere alla CPU0 di ricevere interrupt, qualora non si bloccasse su WAIT().

Su tutte le CPU con PRID > 0, ossia quelle non attive a inizializzazione di umps2, sarà chiamata la funzione **`INITCPU()`,** per metterle in stato IDLE e renderle quindi recettive a Interrupts Inter-Processore.
Passando come parametri lo state_t del loro scheduler (la cui esecuzione senza processi in attesa li condurrà in IDLE) e il puntatore alla prima di otto locazioni di memoria contigue contenenti le OLD/NEW area che useranno per gestire eccezioni, interrupts e system calls.

Verrà poi settato lo stato del primo processo (i.e. test), utilizzato come entry point.
La memoria su cui scriverà il suo stack sarà situata un frame sotto quella riservata al processore con PRID più alto (ossia l'ultimo).
L' **Interval Timer**, unico in tutto il sistema, verrà avviato scrivendovi un valore pari a quello definito dalla costante SCHED_PSEUDO_CLOCK (100'000 microsecondi) .
La funzione **`assignProcess()`** sarà chiamata, assegnando il processo alla CPU0, e partirà la primissima esecuzione di **`schedule()`** da parte della CPU0, che darà inizio a ogni successiva attività del nucleo.

2.2. Assegnamento Processi

Alla creazione di ogni processo (l'entry point e, successivamente, tutti quelli creati usando le system calls CREATEPROCESS e CREATEBROTHER) verrà chiamata la funzione **assignProcess()** sul pcb_t che lo descrive.
Come prima cosa la funzione determinerà a quale processore assegnare il processo.
La valutazione verrà effettuata semplicemente ciclando sui contatori di processi attivi relativi a ogni processore e determinando quale abbia il valore minore.
Una volta effettuata la scelta, il PRID del processore verrà scritto in un campo del pcb_t e quest'ultimo vi sarà legato per tutta la durata della sua esecuzione.
Questa è stata forse una delle scelte progettuali sulla quale ci siamo trovati più combattuti.
Perché non tiene conto del tipo di processo assegnato (I/O bound? CPU bound?), che del resto è impossibile determinare staticamente.
Abbiamo pensato quindi di implementare un continuo scambio di processi, utilizzando un meccanismo di ri-assegnazione a ogni esecuzione del nucleo, ma dai numerosi test svolti è trapelato il fatto che l'overhead derivante dagli ulteriori controlli a ogni scheduling (ricontrollare i contatori per determinare a quale processore ADESSO sarebbe meglio ri-assegnare il processo, etc.) non sarebbe trascurabile… peggiorando notevolmente le prestazioni del sistema.
Ci siamo quindi mantenuti su quest'approccio, il quale è comunque il migliore nella maggior parte dei casi (raramente numerosi processi CPU bound e IO bound sono legati nella loro esecuzione, per esempio sincronizzandosi mediante lo stesso semaforo) , promuovendo anche un uso più efficiente della cache di processore, che non si alternerà su troppi processi diversi in un breve istante di tempo.
Quindi il processo sarà accodato nella Ready Queue appartenente al processore selezionato.
Se il processore in questione si trova in stato IDLE, utilizzeremo un Inter-Processor Interrupt per risvegliarlo (macro-funzione **SEND_IPI()** , accende il bit corrispondente al processore con PRID selezionato all'interno del 2°/3° byte nel registro OUTBOX, adibito a questo scopo).
Da questo momento in poi, in seguito a eventuali operazioni di I/O o semplice round-robin, il processo verrà re-inserito nella Ready Queue della CPU assegnata staticamente, mediante il campo **cpu_id**, usando la funzione **enqueueProcess().**

2.3. Scheduling

La funzione principale nel modulo (nonché quella più eseguita) è probabilmente la **`schedule()`.** Questa internalizza l'essenza dello scheduler, prelevando processi dalla rispettiva Ready Queue e mettendoli in esecuzione.
Poiché il codice è unico ed eseguito da tutti i processori, l'id univoco di quello chiamante viene salvato in una variabile locale, tramite la funzione getPRID().
Ogni ulteriore operazione si baserà sul valore di questa variabile.
Come prima cosa il processore setta il proprio stato a Running (quindi idle a FALSE) nella posizione corrispondente del vettore; così da segnalare ad un eventuale assegnamento che non vi è bisogno di risvegliarla con un interrupt.
Particolarmente importante qui è l'acquisizione della propria variabile lock: impedirà che per un caso di race condition il processo venga inserito nella Ready Queue (senza un Inter-Processor Interrupt a seguire) dopo che il processore interessato l'avrà controllata, determinando che sia vuota e quindi il caso di mettersi in IDLE.
Questo lascerebbe il processo in attesa di uno scheduling che non avverrà fino ad un successivo interrupt!
Nel prelevare il primo processo dalla propria Ready Queue (per come è stato gestito il meccanismo di inserimento, al livello sottostante, il processo in testa sarà il primo inserito fra quelli con priorità massima), lo schedulerà scarterà quelli il cui flag **killed** sia settato su TRUE, a indicare il fatto che qualche altro processore abbia eliminato l'intera "famiglia" di quel processo.
Qualora ne trovasse, oltre che scartarli provvederà a rilasciarne le risorse (il pcb_t) e decrementare il proprio contatore.

2.3.1. Gestione Priorità

Il passo successivo sarà decrementarne la priorità corrente, riportandola a quella base nel caso raggiungesse 0.
Per evitare starvation di processi con priorità bassa, abbiamo implementato questa leggera variazione del meccanismo di aging (non è la priorità dei processi in attesa a incrementare, ma quella dei processi in esecuzione a calare).
Questo garantirà che, eventualmente, persino un processo con priorità alta arriverà al livello minimo, consentendo l'esecuzione dei "meno privilegiati".
Per un risultato di testing su questo meccanismo, si veda l'apposita sezione **6.1**.

Il motivo per cui non abbiamo usato code per processi con differenti priorità è che un miglioramento delle prestazioni degno di nota si rileva solo usando un processore.
Se il sistema ne usa di più, in certi casi abbiamo notato addirittura un degrado (poiché già usando code multiple, è difficile che più processi assegnati allo stesso processore siano distribuiti su più di una coda).
Ma poiché Pkaya è per definizione un sistema per ambiente multiprocessore, abbiamo pensato di adottare questo metodo.

Il Processor Local Timer sarà abilitato, durante l'esecuzione del processo scelto (cosi da permettere a ogni processore di gestire il proprio TIMESLICE).
Inoltre l'istante della messa in esecuzione sarà registrato nel campo **systemTOD** del pcb_t (per un corretto accounting del tempo di esecuzione. Vedi sezione sulle System Calls **4.**).
Quindi, mediante la funzione di ROM `LDST(),` lo state_t del processo scelto verrà caricato sul processore e la sua esecuzione avviata.

Tutto questo nel caso in cui la Ready Queue del processore non sia vuota.
Se lo fosse, avremmo due alternative:

- Il contatore dei processi assegnati è superiore a 0, ma nessuno di essi è bloccato per sincronizzazione o I/O. Questo può solo indicare che qualcosa sia andato storto durante l'esecuzione di una delle funzioni del nucleo.
  La conseguenza è il PANIC del processore.
  N.B: Questa condizione potrebbe non essere verificata se vi fossero dei processi bloccati su I/O, ma assegnati ad un altro processore (effettivamente però, sussisterebbe!)
  La nostra scelta è stata quella di usare comunque un unico contatore per i processi bloccati, ritenendolo necessario allo scopo.
- Il contatore dei processi assegnati è a 0, oppure ve ne sono alcuni bloccati in attesa di I/O.
  In tal caso la CPU segnala la transizione allo stato IDLE e ci si pone (dopo aver rilasciato il lock sulle proprie strutture) chiamando la funzione di ROM `WAIT().`

E' quindi il caso di spiegare l'esistenza del processo dummy, utilizzato nel caso la chiamata a `WAIT()` non inducesse la CPU in uno stato di IDLE.

Questo comportamento l'abbiamo riscontrato unicamente nella CPU0.. e non siamo stati in grado di determinare l'effettiva causa..sebbene di fatto avvenga raramente. Eseguendo numerosi test con breakpoint sulla funzione **`wait()`** (non quella di ROM, ma quella eseguita dal processo dummy), questi non si verificavano mai.

In ogni caso "better safe than sorry", se la CPU0 non si mettesse in IDLE, caricherebbe semplicemente un processo inutile, con interrupts abilitati, in attesa di un evento esterno.

Le altre funzioni, quali **`decProcessCounter(),incSoftCounter()`** e **`decSoftCounter(),`** sono solo dei wrappers per alterare in mutua esclusione il valore dei contatori, da altri moduli.
O semplicemente occorrono a prelevare un riferimento al processo corrente, mantenendo l'incapsulamento delle variabili, e riavviare lo scheduler.

3. Interrupts

Veniamo adesso alla gestione degli Interrupts.
Tra tutti, quelli che gestiremo con azioni adeguate saranno quelli scatenati dai Processor Local Timers (propri di ogni processore), Interval Timer, Terminali (ognuno di essi in lettura o scrittura) e quelli da un processore all'altro.
All'avvio del gestore, il program counter attuale e tutto lo stato del processo corrente verranno copiati dall' OLD AREA riservata agli interrupts (nella ROM per il processore 0 o nel nostro array in RAM per gli altri) nella struttura pcb del processo in esecuzione.
Dopo aver individuato, tramite le apposite funzioni di libreria, quale sia la causa dell'interrupt, lo si gestisce e si restituisce il processo allo scheduler.
Lo stato della CPU viene settato su Running (idle = FALSE), cosi che non si debba ricevere interrupts Inter-Processore inutili, uscendo dal gestore.

Seguiamo adesso la gestione dei singoli casi:

3.1. Processor Local Timer

Gli interrupts dai **Processor Local Timer** vengono ricevuti e gestiti da tutti i processori. Questa indipendenza li rende un ottimo strumento per il time-slicing individuale.
Infatti, se l'interrupt è di questo tipo, ci limitiamo ad aggiornare il contatore del tempo consumato da quel processo, sommandoci la differenza fra l'istante attuale (funzione **GET_TODLOW()**) e quello registrato al caricamento del processo, nel campo **systemTOD** del pcb.

3.2. Interval Timer

Gli interrupts da **Interval Timer** vengono gestiti unicamente dalla CPU0.
Alla ricezione, viene chiamata la funzione **`Vclock()`**, implementata in *utils.c.*

La funzione acquisisce il lock sul semaforo assegnato allo PSEUDO CLOCK e sblocca tutti i processi bloccati su esso.
Vi sono due ragioni per cui abbiamo implementato questa operazione in una funzione a parte, e non nella `V()` eseguita per tutti gli altri semafori:
Come prima cosa, vogliamo che il lock sul semaforo sia posseduto fino a che non rimangono processi bloccati su esso, cosi da evitare un eventuale loop (usando più processori) nel quale man mano che la CPU0 rilascia dei processi, ne arrivano di altri.
Come seconda cosa, dato che la system call **WAITCLOCK** ha lo scopo principale di sincronizzare uno o più processi, preferiamo evitare che il contatore del semaforo sia incrementato, quando non vi sono processi bloccati su esso.

3.3. Terminali

Gli interrupts dai **Terminali** sono quelli che richiedono una gestione più articolata.
Tanto per cominciare, non sono un device unico: ve ne sono 8.
Per determinare quale terminale abbia generato l'interrupt, leggiamo il valore della **TERMINAL_PENDING_BITMAP** (il cui indirizzo ci è reso noto dalla documentazione) e lo confrontiamo con un altro, multiplo di 2, shiftando a sinistra il valore di quest'ultimo a ogni confronto negativo, cosi da determinare quale bit sia acceso.
Una volta individuato il numero di terminale, leggiamo il primo byte del registro di stato (sia in ricezione che in trasmissione) per stabilire se il carattere è stato letto/trasmesso con successo.
In tal caso, sblocchiamo (se ce n'è uno) il processo che ha effettuato l'operazione e salviamo nel suo registro v0 il valore di stato letto.
Questo nell'eventualità che il processo che ha scritto/letto abbia chiamato la system call **WAITIO** prima che l'interrupt sia stato gestito, bloccandosi cosi sul semaforo del terminale.
Altrimenti (interrupt gestito prima della chiamata a WAITIO), salviamo il valore nella posizione relativa al nostro numero di terminale, nell'array **deviceResponseR** o **deviceResponseT**, a seconda che sia ricezione o trasmissione.
Cosi che sarà il gestore della system call, qualora la **`P()`** chiamata non fosse bloccante, a prelevare il valore di risposta dall'array.
Qualora un processo fosse stato sbloccato, occorrerà decrementare il contatore di processi in attesa e re-inserirlo nella Ready Queue del processore corrispondente.
La successiva nonché ultima operazione sarà quella di scrivere il codice ACK nel registro di comando del terminale, così da segnalare ad esso l'avvenuta ricezione.

3.4. Inter-Processor Interrupts

Gli interrupts **Inter-Processore** non richiedono una gestione particolare.
Li usiamo solo per ragioni di scheduling, quindi per risvegliare processori IDLE qualora vi siano nuovi processi assegnati ad essi, o vecchi che hanno completato l'I/O. Ci limitiamo a scrivere nel registro **INBOX** , segnalando l'avvenuta ricezione con un acknowledgement.

Nel caso arrivasse un interrupt da qualche altro dispositivo, risulterebbe in un PANIC(), dato che la loro gestione accurata non è prevista dalle specifiche di questo progetto.

4. System Calls

Il file `syscalls.c`, unito al file `util.c`, contiene l'implementazione delle system calls richieste da specifiche. Generalmente, si è deciso di implementare il gestore delle system calls come un'unica grande funzione detta ***`sysHandler()`***, che esegue al suo interno uno `switch` per determinare quale system call si sia verificata, procedendo poi di conseguenza con la sua esecuzione.

All'inizio del file sono state inserite le dichiarazione dell'array contenente le new/old area delle CPU nonché l'intestazione della funzione ***`trapHandler().`*** I primi servono a recuperare il numero della system call verificata nonché i suoi eventuali parametri (salvati automaticamente dall'hardware nella old area della CPU su cui si è verificata) mentre la seconda è da richiamare in un caso particolare di esecuzione di una system call.

Alla fine di tutto, sia che sia trattato di una system call o di un breakpoint, dopo averli gestiti, non resta altro che ricedere il controllo allo scheduler, che provvederà a ricaricare su quella CPU un altro processo per l'esecuzione. Se la system call è stata bloccante, il processo chiamante non apparirà momentaneamente nella ready queue e quindi non sarà mandato in esecuzione fino ad un'operazione `V()` sul semaforo su cui è bloccato. In caso contrario, l'esecuzione del processo chiamante della system call, o dei nuovi eventuali processi creati, riprenderà appena sarà il loro turno.

4.1. La funzione sysHandler

Al verificarsi di una chiamata di sistema, viene attivato tale gestore che procede subito col recuperare il registro cause, l'identificatore della CPU si cui si è verificata la system call, nonché il processo che l'ha richiesta, salvato poi nella variabile `current`.

Inizialmente, ci si deve domandare subito una cosa: se per caso tale chiamata si sia verificata in *user mode.* Se questo è successo, e lo si capisce confrontando lo *status* processo chiamante (che ricordiamo essere stato salvato nella old area della CPU) con la costante `STATUS_KUp.` Qualora tale confronto abbia esito positivo la sistem call che si sta tentando di eseguire viene trattata come un' eccezione , dunque viene copiata la old area delle system call in quella delle eccezioni ed in quest'ultima è settato il registro `cause` come `EXC_RESERVEDINSTR` che significa "istruzione riservata". Ciò fatto, viene richiamato il gestore delle eccezioni che provvederà a gestire la cosa.

In caso contrario, tutto viene gestito in modalità standard in kernel mode.

Per prima cosa ci preoccupiamo di ottenere il processo che ha richiesto la system call , prendendolo dall'array `HIDDEN pcb_t *running[MAX_CPU]` , attraverso una funzione apposita detta `getRunningProcess()`, che contiene, per ogni CPU, il processo in essa attualmente in esecuzione. Lo salviamo nella variabile `current`. Poiché le new/old area della CPU numero zero sono mappate direttamente in memoria a certi indirizzi fissi mentre le altre sono state allocate in un array, a seconda della CPU su cui si è verificata la chiamata andiamo a recuperare dalla sua old area lo state_t del processo che l'ha richiesta e lo copiamo nello state_t della variabile `current` servendoci della macro `copyState()` presente nel file `utils.h` e creata appositamente per lo scopo. Cosa molto importante, è ricordarsi di fare scorrere avanti di un'istruzione il registro *program counter (pc_epc)* del processo chiamante così da evitare che la stessa system call venga eseguita di nuovo in un loop. A questo punto verifichiamo che nel registro *cause* sia contenuto `EXC_SYSCALL` che significa che si è verificata una system call, il cui numero viene messo nel registro `reg_a0` del processo che l'ha richiesta e su tale registro viene fatto lo `switch`. A seconda dei casi, il gestore si comporta di conseguenza.

4.2. Implementazione

4.2.1. `case CREATEPROCESS`

La system call numero 1 permette di creare un nuovo processo che verrà inserito come figlio di quello chiamante. A tale scopo occorre ottenere prima un nuovo

descrittore di processo *(pcb)* di cui verranno settati tutti i campi e poi mandato in esecuzione. Poiché tutti i descrittori di processo disponibili stanno in un'unica lista detta `pcbFree,` occorre potere accede a tale lista in mutua esclusione tra le varie CPU. Questo viene fatto attraverso un *wrapper* C creato appositamente e chiamato `getPcb().` Tale funzione esegue una CAS per ottenere il lock (chiamato `pcb_Lock)` sulla lista dei processi liberi o eventualmente attendendo che qualche altra CPU ne rilasci la mutua esclusione. Ciò fatto, basta chiamare la funzione `allocPcb()` (implementata in fase 1 nel modulo `pcb.c`) la quale ci restituisce il primo descrittore di processi libero e ne inizializza i campi. Se non è possibile ottenere un nuovo descrittore di processo, in quanto sono esauriti, viene restituito il valore NULL e la system call termina con errore (valore -1) nel registro di ritorno del processo, il registro `reg_v0`. La CREATEPROCESS quando invocata ha due parametri: la priorità del nuovo processo nonché il suo state_t da cui inizierà l'esecuzione, contenuti rispettivamente nei registri reg_a2 e `reg_a3.`

Se anche lo state_t, passato per indirizzo, dovesse risultare come NULL, allora la system call termina con errore. Altrimenti lo state_t e la priorità passati alla chiamata sono assegnati al nuovo processo il quale poi è inserito come figlio del processo chiamante attraverso la funzione `insertChild()` e poi messo in una ready queue delle CPU attraverso la funzione `assignProcess()`. Effettivamente questo è l'unico punto in cui ad un processo viene assegnata una CPU, all'atto della sua creazione. E vi resterà in esecuzione fino al termine. Infine il processo chiamante è re-inserito nella ready queue del processore che gli è stato assegnato precedentemente. Quando la system call ha avuto esito positivo, al processo chiamante viene restituito il valore 0 sempre nel registro `reg_v0`.

4.2.2. **case** CREATEBROTHER

Questa system call, la numero 2, si comporta esattamente come la precedente con l'unica differenza che il nuovo processo creato viene inserito come figlio del padre del processo che ha invocato la chiamata.

In parole povere, si esegue una `insertChild()` sul campo `p_parent` del processo chiamante.

4.2.3. `case` TERMINATEPROCESS

Tale system call, numero 3, si limita a chiamare la funzione `terminateProcess()` la quale è stata implementata come funzione esterna nel modulo `utils.c` in quanto è richiamata in vari punti del sistema a seconda di situazioni diverse che possono accadere. Riguardo alla terminazione del processo, si è deciso di inserire un ulteriore campo nella struttura dati `pcb_t`, chiamato `killed`. Sostanzialmente si tratta di un flag che viene settato a TRUE qualora il processo sia stato eliminato, lasciando allo scheduler il compito di reinserirlo nella lista dei processi liberi qualora una CPU ne trovi nella sua ready queue. La `terminateProcess()` richiama al suo interno la funzione di phase1 `outChildBlocked()` la quale è stata modificata appositamente affinché faccia quanto descritto sopra e procede per passi nel seguente modo.

Inizialmente controlla se il processo che si intende eliminare è bloccato su qualche semaforo (ossia se il campo p_semkey è diverso da -1). In caso di esito positivo, si estrae il descrittore del processo dalla coda del semaforo tramite la funzione `outBlocked()` aggiornando subito dopo il contatore del semaforo incrementandolo di 1. Bisogna inoltre ricordarsi di decrementare la variabile di sistema che tiene conto dei processi bloccati nonché di aggiornare il contatore di processi attivi di quella CPU a cui era stato assegnato il processo bloccato appena eliminato poiché adesso quest'ultima ha un processo in meno da eseguire. Il descrittore di processo è infine reinserito nella coda dei processi liberi.

Nel caso in cui il processo da eliminare non si trovi bloccato su un semaforo, abbiamo due casi:

- Se il processo da eliminare era assegnato alla CPU che sta attualmente eseguendo l'eliminazione (era uno dei suoi), allora semplicemente il processo viene reinserito nella lista di quelli liberi ed è aggiornato il contatore di processi attivi del processore.
- In caso contrario, se il processo da eliminare appartiene invece ad un'altra CPU diversa da quella che sta eseguendo la `terminateProcess()`, allora ne viene settato il flag `killed` come TRUE lasciando poi allo scheduler di quella CPU il compito di reinserire tale processo nella lista di quelli liberi, quando si tenta di eseguirlo.

Tutte queste operazioni vengono rieseguite richiamando ricorsivamente la funzione `outChildBlocked()` a tutta la progenie del processo appena eliminato, accedendo quindi alla lista dei suoi figli tramite il campo `p_child` e facendo uso della funzione data `list_for_each_entry()` per scansionarla tutta.

La decisione di procedere in questo modo è stata presa in quanto è risultata quella di più semplice e diretta implementazione. Un'implementazione alternativa considerata poteva essere quella di doversi scorrere tutte le ready queue alla ricerca dei figli del processo da eliminare ed estrarli per rimetterli nella lista dei disponibili e la stessa cosa era da fare per i semafori. Ma l'ostacolo maggiore sarebbe stato quello in cui uno dei processi figli da eliminare si trovava attualmente in esecuzione su un'altra CPU. In questo caso occorreva mandare ad essa un segnale `INITCPU` in modo che interrompesse il processo e ricominciasse con lo scheduler. Il tutto contornato da grossi problemi di mutua esclusione. Pertanto si è deciso per questa soluzione, a discapito che uno dei processi figli potrebbe continuare ancora per uno `SCHED_TIME_SLICE` la propria esecuzione prima di essere terminato.

4.2.4. **case** VERHOGEN

La system call numero 4 esegue un'operazione V sul semaforo passato come chiave nel registro reg_a1.

Prima di passare all'operazione V vera e propria si controlla se il semaforo a cui si vuole accedere non sia un semaforo dei device riservati al sistema o un semaforo inesistente (di chiave negativa), ma effettivamente un semaforo per uso utente (poiché l'operazione di `V()` su semafori dei device spetta alla gestione degli interrupt). Per uso utente, sono i semafori compresi tra 0 e USER_SEM (che vale 19 in quanto il primo semaforo di sistema è quello del device PSEUDO CLOCK che è di chiave 20). Se questo è vero, si può procedere con la funzione `V()` implementata come funzione esterna nel modulo `utils.c` poiché molto usata anche dal modulo `interrupts.c`.

Per avere accesso in mutua esclusione ai semafori ed alle loro operazioni e quindi rendere le operazioni P e V come "atomiche", è stato creato l'array

```
unsigned int sem_Lock[NUMSEM]
```

dove ogni posizione all'interno corrisponde alla chiave del semaforo. Gli unici valori possibili che ogni elemento dell'array può assumere sono `BUSY (0)` e `FREE (1)` che indicano rispettivamente che il semaforo è occupato, in quanto un processo vi sta lavorando sopra, oppure libero. Perciò prima di iniziare a lavorare su un semaforo, occorre prima ottenerne l'accesso in mutua esclusione o eventualmente attendere e questo viene eseguito dalla funzione CAS, che rimane in attesa se il `sem_Lock[key]` è BUSY e per prendersi poi l'accesso appena ritorna FREE. Ogni operazione/accesso a semaforo è quindi protetta da una CAS di questo tipo.

Dunque, si può procedere con l'estrarre il primo processo dalla coda del semaforo con la funzione di fase 1 `removeBlocked()` nonché con l' aggiornarne il valore accedendovi tramite indirizzo con la funzione `getSemd().` Infine si rilascia la mutua esclusione sul semaforo e si restituisce il puntatore al processo sbloccato. Notiamo che il compito di rimettere il processo nella ready queue viene fatto fuori dalla funzione `V()` vera e propria in quanto si è voluto fare in modo che dentro questa operazione (nonché la `P()`) si operasse solo esclusivamente sul semaforo data la sua chiave, senza fare altro. Il resto viene lasciato all'interno della funzione `sysHandler()` come il reinserimento del processo nella ready queue del processore che gli è stato assegnato ed il decremento della variabile indicante il numero di processi bloccati tramite la funzione `decSoftCounter().`

4.2.5. `case PASSEREN:`

Tale system call, numero 5, esegue gli stessi controlli di mutua esclusione e tipo di semaforo cui si vuole accedere svolti dalla VERHOGEN, prima di chiamare la `P()` passandole, oltre alla chiave del semaforo, anche il puntatore al processo chiamante che può essere potenzialmente bloccato. La `P()` ottiene la mutua esclusione al semaforo nello stesso modo della `V()` e decrementa subito il valore del semaforo. Qualora quest'ultimo risulti essere minore di 0, il processo chiamante viene bloccato. Questo significa che viene tolto dalla ready queue ed inserito nella coda del semaforo tramite la funzione di fase 1 detta `insertBlocked().` La funzione `P()` restituirà quindi un valore di FALSE che indicherà alla `sysHandler()` che non deve riaccodare il processo sulla ready queue poiché è bloccato. Inoltre, nel momento in cui il processo si ferma su semaforo, occorre dichiarare che c'è un processo bloccato in più incrementando la variabile di sistema con `incSoftCounter()`ed aggiornare il suo tempo di esecuzione fino al momento in

cui si è bloccato. Altrimenti, se il valore del semaforo era maggiore di 0, la `P()` non bloccherà e quindi verrà restituito TRUE e dunque il processo potrà essere riaccodato per l'esecuzione.

4.2.6. **case** GETCPUTIME

La system call numero sei restituisce il tempo di esecuzione del processo fino al momento della sua chiamata. Esso viene accumulato di volta in volta, alla fine di ogni time slice o al verificarsi di una system call bloccante, all'interno del campo `cpu_time` che è stato aggiunto alla definizione di *pcb* appositamente per questo scopo.

Per aiutarci nel calcolo di questo tempo, abbiamo introdotto nella definizione di *pcb* anche il campo `systemTOD`. Quest'ultimo viene inizializzato dallo scheduler al tempo attuale di sistema subito prima di mandare in esecuzione il processo. Alla fine del time slice o ad una system call bloccante, per capire quanto tempo è passato si fa la differenza col tempo attuale (`GET_TODLOW`) con quello da cui si era partiti e si somma il risultato accumulandolo al `cpu_time` di volta in volta. Dunque la `GETCPUTIME` non fa altro che restituire questo valore aggiungendovi anche il tempo trascorso fino alla sua chiamata.

4.2.7. **case** WAITCLOCK

La system call numero 7, non fa altro che richiamare la funzione `P()` sul semaforo associato `allo PSEUDO CLOCK TIMER` che nel nostro caso è quello di chiave 20. L'unica differenza è che tale `P()` è sempre bloccante poiché il valore del semaforo è mantenuto al massimo sempre a 0. I processi che si bloccano sul semaforo dello PSEUDO CLOCK verranno poi svegliati dall'interrupt del TIMER tramite l'operazione di `VClock().`

4.2.8. **case** WAITIO

Questa system call, numero 8, esegue un'operazione `P()` sul semaforo associato al dispositivo che si ricava secondo i parametri passati alla system call: la linea di interrupt su cui il dispositivo si trova (nel nostro caso solo quella dei terminali, la

numero 7) il numero del dispositivo di quella linea e specificando se l'operazione è in lettura o scrittura. Discriminando sul valore del registro reg_a3 (che assume il valore di 1 per un'operazione di lettura o 0 per una di scrittura) si ricava la chiave del semaforo su cui fare l'operazione P() secondo la macro

TERMINAL_SEM_R o TERMINAL_SEM_T le quali partendo dal valore base di 20 (che ricordiamo essere il primo semaforo associato ai device, in particolare lo PSEUDO CLOCK TIMER) ricava la chiave degli altri. Ciò fatto viene effettuata una normale chiamata alla funzione P() come descritta sopra. Nel caso quest'ultima non dovesse essere bloccante e quindi il device era subito pronto, si va a recuperare il registro di stato di quest'ultimo direttamente dall'array deviceResponseR o deviceResponseW.

Poiché in questo sistema ci occupiamo solo di terminali come device I/O, una qualsiasi richiesta di questo tipo di operazione su altre linee viene trattata come PANIC().

4.2.9. case SPECPRGVEC, case SPECTLBVEC, case SPECSYSVEC

Le seguenti system call, numeri 9, 10 e 11, essenzialmente fanno le stessa cosa ossia quella di ridefinire un gestore personalizzato per processo delle tre eccezioni possibili: program trap, system call e tlb.

Per potere salvare tali gestori, che sono tipici per ogni processo, è stato inserito un array di 6 puntatori a *state_t* di nome def_areas[ ] nella definizione di *pcb* dove abbiamo che:

def_areas[0] : old area per eccezioni TLB;

def_areas[1] : new area per eccezioni TLB ;

def_areas[2] : old area per eccezioni PROGRAM TRAP

def_areas[3] : new area per eccezioni PROGRAM TRAP

def_areas[4] : old area per eccezioni SYSTEM CALL

`def_areas[5]` : new area per eccezioni SYSTEM CALL

All'evocazione di una di queste tre system call, viene salvato nella new area di `def_area[ ]` il puntatore allo *state_t* del gestore personalizzato, passato come parametro alla system call, per quell'eccezione di quel processo. Questa operazione è permessa una sola volta e se un processo tenta di rieseguirla allora viene immediatamente terminato.

4.2.10. `default`

Nel caso non sia stata chiamata nessuna delle 11 system call dichiarate sopra, si ricade in questo ramo. Si tratta quindi di una system call che differisce da quelle standard offerte dal sistema, per cui esistono due possibilità. Se il processo ha definito il proprio gestore delle system call con la chiamata SPECSYSVEC discussa prima (e quindi il suo campo `def_areas[5]` è diverso da NULL) allora si copia nella old area specifica (`def_areas[4]`) lo stato attuale del processo sostituendolo con quello del gestore personalizzato per poi essere rimesso nella ready queue dove, una volta ottenuta la CPU, verrà eseguito. In caso contrario, se questo gestore apposito non è stato dichiarato, significa che si è tentato di fare una chiamata ad una system call inesistente che non può essere gestita in alcun modo, e dunque viene terminato il processo.

4.3. Gestione dei Breakpoint

Se si è verificato un BREAKPOINT, e lo si riconosce se dentro al registro `cause` vi troviamo il valore `EXC_BREAKPOINT`, allora si verifica se esiste un gestore personalizzato per i breakpoint per quel processo (la cui area di definizione coincide con la stessa new area di quella delle system call). Se esiste, si copia lo *state_t* di quel gestore nel processo attuale e si rimette il tutto nella ready queue. Altrimenti viene terminato il processo.

5. Eccezioni

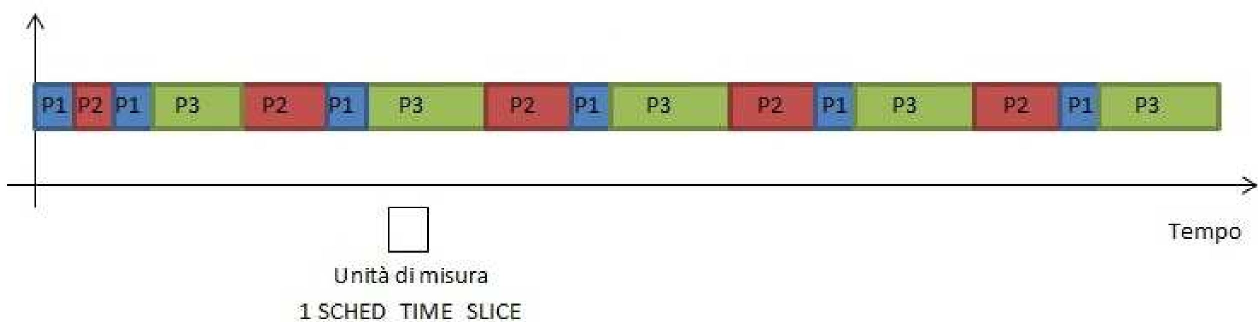
Program Traps e TLB Exceptions.
Quando si verifica un'eccezioni di uno di questi due tipi, dopo essersi ricavato il descrittore del processo che ha scatenato l'eccezione chiamando la funzione getRunningProcess(), in entrambi i casi sostanzialmente si verificano due cose. Se il processo ha definito il proprio gestore di eccezioni (ossia `def_areas[3]` diverso da NULL per le PROGRAM TRAP e `def_areas[1]` diverso da NULL per le TLB TRAP) allora viene salvato nelle old area dell'array `def_areas[ ]` l'old area del processore e caricato invece nello *state_t* del processo lo *state_t* del gestore personalizzato. Subito dopo il processo è rimesso nella ready queue e quando verrà scelto dallo scheduler ne verrà eseguito il gestore. Altrimenti se non è stato definito nessun gestore, il processo viene terminato con la chiamata alla `terminateProcess().` In entrambi i casi, qualunque cosa si sia verificata viene dato di nuovo il controllo allo scheduler.

6. Testing

Dedichiamo questo capitolo a riportare i risultati di alcuni test da noi effettuati.
Il primo è un semplice testing di come la gestione di aging della priorità dei processi da noi implementata ne condizioni l'avvicendamento.
Il secondo consiste semplicemente nei risultati del test di scalabilità fornitoci, con diverso numero di processori.
Su quest'ultimo abbiamo effettuato alcune modifiche, spiegate dettagliatamente nel paragrafo apposito.

6.1. Priorità

Mostriamo qui il diagramma di Gantt relativo all'esecuzione di 3 processi, chiamati rispettivamente P1,P2 e P3.
Ai processi è stata assegnata una priorità rispettivamente di 1,2 e 4.
L'esecuzione, per evidenziarne l'alternanza, è sulla macchina in configurazione monoprocessore.



6.2. Scalabilità

Cominciamo evidenziando le correzioni da noi apportate:

In precedenza, lo Stack Pointer dei processi creati durante il test veniva settato a RAMTOP – QTABLE, e cosi via scendendo di 1024 indirizzi a volta.
Di conseguenza alcuni dei primi processi avevano lo Stack posizionato in potenziale sovrapposizione con quello dei nostri Handlers/Schedulers (quelli assegnati al processore 1 partivano da RAMTOP – FRAME_SIZE).
Abbiamo risolto la cosa sostituendo l'assegnamento ai vari Stack Pointers con quello utilizzato nel p2test: il 1° processo ha uno stack che inizia 1024 indirizzi sotto quello del processo test, il 2° 1024 indirizzi sotto quello del 1° e cosi via...fino all'ultimo.

Il secondo problema nel test era più incisivo:
Per il Test 4, vengono creati, tra gli altri, 5 processi di tipo "proc_cpu".
Questi prima di iniziare il loro CPU burst, chiamano la PASSEREN sul semaforo con chiave STARTCPU, senza però che siano state effettuate tutte le VERHOGEN necessarie su quel semaforo.
Di conseguenza l'esecuzione si interrompe senza possibilità di proseguire, dato che il processo principale attenderà una VERHOGEN da quelli stessi sotto-processi.
Abbiamo risolto con un semplice:

```
for (i=0; i < 4; i++)
SYSCALL(VERHOGEN, STARTCPU, 0, 0);
```

Prima della creazione dei processi.

Riportiamo adesso i risultati del test (che per inciso va a buon fine con qualunque configurazione di processori usati), con tempo di esecuzione in microsecondi:

N° CPU	TEST 1	TEST 2	TEST 3	TEST 4
1	0152898033	0227730307	0000656738	0038004286
2	0077414079	0148548852	0000411299	0029197351
4	0038485290	0076588087	0000221123	0028951310
8	0019228148	0038277356	0000169525	0009649720
16	0009672848	0019210993	0000169556	0009628160

E’ osservabile che il tempo di esecuzione è inversamente proporzionale al numero di processori usati, per i test con processi maggiormente CPU bound (TEST 1 e TEST 2).

6.3. How to use it

Il nostro MAKEFILE è scritto in modo tale da consentire due differenti configurazioni: quella con il p2test e quella con il p2test_scalability.
Lanciare *make p2s* per compilare il p2test_scalability e linkarne il file object all’eseguibile risultante.
Il make con *p2* come parametro, o senza parametri, risulta nella compilazione e linking del p2test.