PandOS plus: Fase 2

Relazione per il progetto del corso di Sistemi Operativi

J.G. Jaramillo Saa, E. Campanelli

Università di Bologna
April 13, 2022

1 Risorse

Durante lo sviluppo di questa fase, abbiamo consultato abitualmente e regolarmente le risorse che ci sono state messe a disposizione dal professore (link) oltre al gruppo telegram del progetto.

Inoltre ci sono state d'aiuto le risorse messe a disposizione dagli studenti degli anni scorsi, in particolare abbiamo consultato il seguente repository quando ci siamo trovati in difficoltà phase2.

2 Introduzione

Questa fase del progetto si occupa di implementare il livello 3 del sistema operativo: il Nucleo.

Il Nucleo si occupa di:

1. Inizializzare il sistema
2. Scheduling dei processi
3. Gestire le eccezioni

3 Moduli

Per questa fase è stato deciso di seguire la guida per pandosplus, suddividendo l'implementazione del Nucleo in 4 moduli:

- **initial.c** : Questo modulo implementa il main() e dichiara (e esporta) le variabili globali necessarie per la fase 2;
- **scheduler.c**: Questo modulo implementa lo scheduler e il deadlock detector;
- **exceptions.c**: Questo modulo implementa i gestori delle syscall, program e tlb traps.
- **interrupts.c**: Questo modulo si occupa di implementare i gestori degli interrupt dei device/timer.

3.1 Initial.c

Viene implementata la funzione main che si occupa di:

1. Inizializzare (e dichiarare) le variabili del livello 3:
   - *p_count* : Intero che indica il numero di processi che sono iniziati e ancora non terminano;
   - *soft_counter* : Intero che rappresenta il numero di processi che non sono terminati e che sono in stato "blocked" a seguito di un'operazione di I/O o una richiesta all'interval timer;
   - Ready queues; *ready_hq* , *ready_lq* : Si tratta di due variabili di tipo ***struct list_head*** che rappresentano rispettivamente la coda dei processi ready ad alta priorità e la coda dei processi ready a bassa priotità;
   - *current_p* : Puntatore al pcb del processo correntemente attivo;
   - *sem[N_DEVICE]* : Array di interi di dimensione *N_DEVICE* che rappresentano il totale dei semafori associati a tutti i (sub)device di uMPS3.
2. Inizializzare il Pass Up Vector (*passupvector*);
3. Inizializzare le strutture dati di livello 2 (*initASL()* e *initPcbs*);
4. Caricare l'interval timer con 100 millisecondi (*LDIT(100000)*);
5. Creare e inizializzare il processo init, a bassa priorità (**NB**: tutte le inizializzazioni sono fatte direttamente all'interno della funzione *allocPcb()*);
6. Chiamare *scheduler()*.

A questo punto il controllo non tornerà piú a *main*. L'unico modo perchè torni il controllo al Nucleo è solo attraverso le eccezioni, le quali saranno gestite opportunamente.

3.2 Scheduler.c

Viene implementato un deadlock-detector e uno scheduler di tipo preemptive round-robin con un time slice di 5 millisecondi per i processi a bassa priorità e non-preemptive **f.c.f.s.** per i processi ad alta priorità.

Se entrambe le code sono vuote e non si tratta di **deadlock**, lo scheduler esegue una delle seguenti azioni:

- Se vale *p_count == 0* allora il processore non ha piú nessun motivo per continuare a consumare energia, *HALT()* è chiamata;
- Se invece vale *p_count > 0 && soft_counter > 0* allora il processore deve entrare in stato di **WAIT** (attraverso la chiamata a *WAIT()*).

3.2.1 Deadlock-Detector

Si tratta di un semplice controllo: se entrambe le ready queues sono vuote e vale *soft_counter == 0 && p_count > 0* allora si chiama *PANIC()*, perchè c'è stato un deadlock.

3.2.2 Algoritmo di Scheduling

- Se la coda dei processi ad alta priorità non è vuota, allora si assegna la CPU a tale processo (**NB**: non è necessario re-inserire il *current_p* in *ready_?q* perchè, se necessario, tale re-inserimento verrà effettuato all'interno del gestore dell'eccezione);
- Altrimenti si assegna la CPU al processo a bassa priorità che sta aspettando da piú tempo di essere eseguito.

NB: Quando si assegna un processo alla CPU, è necessario rimuovere tale processo dalla ready queue corrispondente, e.g. *current_p = removeProcQ(&ready_hq)*. Inoltre, una volta aggiornato *current_p* al nuovo processo che monopolizzerà la CPU, l'assegnamento vero e proprio del processore al nuovo processo è eseguito dalla *LDST(&(current_p->p_s))*; tale istruzione carica lo stato di esecuzione del processo cosí che venga ripresa.

3.3 Exceptions.c

Al momento in cui occorre un'eccezione, il Pass Up Vector passerà il controllo al gestore delle eccezioni, *exception_handler()*.

Per prima cosa, il gestore va a recuperare lo stato di esecuzione del processo corrente al momento dell'occorrenza dell'eccezione. Tale stato è memorizzato all'inizio del BIOS Data Page (*state_t *exception_state = (state_t *) BIOSDATAPAGE)*. Una volta recuperato tale stato, per individuare di quale tipo di eccezione si tratta, si esamina il campo *ExcCode* (per estrarre tale codice, si effettua una operazione di AND bitwise e si shifta di 2 bit verso destra poichè il campo ExcCode occupa i bit 2-6 del *Cause Register*).

3.3.1 Syscall exception handling

Se vale 4 <= *ExcCode* <= 7 o 9 <= *ExcCode* <= 12, allora il processo in esecuzione ha chiamato una istruzione del tipo *SYSCALL*.

La funzione *syscall_handler()* si occupa di fare il trigger della giusta syscall da eseguire, a seconda del contenuto del registro a0 (che può essere letto tramite l'istruzione: *exception_state->reg_a0*):

-1. **Create_Process**: Un nuovo pcb è allocato e dato come figlio a *current_p*. Il campo pid del nuovo processo allocato è inizializzato come indirizzo del suo pcb: *new_proc->p_pid = (int) new_proc* (questa scelta è stata fatta per semplicità nella gestione della syscall con codice -2). Il pid del processo creato è ritornato nel registro *v0* del processo chiamante.

-2. **Terminate_Process**: Viene terminato il processo indicato o, se il pid è nullo, termina il processo corrente. Inoltre, anche tutta la discendenza del processo è terminata. Questo lavoro e' fatto dalla funzione ausiliaria *terminate_all()*. **NB**: quando un processo viene terminato è necessario liberare le risorse che ha allocate. *terminate_all()* si occupa anche di questo.

-3. **Passeren**: Viene eseguita una operazione di P su un semaforo binario passato come parametro.

-4. **Verhogen**: Viene eseguita una operazione di V su un semaforo binario passato come parametro.

    NB: ***passeren()*** e ***verhogen()*** sono state implementate allo stesso modo in cui sono state implementate le operazioni di P e V sui semafori binari viste a lezione (slides)

-5. **Do_IO_Device**: Effettua un'operazione di I/O scrivendo il comando cmdValue nel registro cmdAddr, e blocca il processo corrente. Poichè si tratta di richiesta di un'operazione di I/O, questa syscall deve effettuare una P sul semaforo appropriato. è stato deciso di calcolare tale indice tramite la seguente formula:

    ```
    int device_index = (line_no - 3) * 8 + device_no + 1
    ```

Si noti che secondo la riga appena scritta, l'indice del dispositivo non potrà mai essere 0. Non è un caso perchè *sem[0]* è riservato al semaforo dell'interval timer.

Per quanto riguarda la *line_no* e il *device_no*, questi sono calcolati nel seguente modo:

```
int line_size = (DEVPERINT * DEVREGSIZE);
int line_no = (((unsigned int) a1_cmdAddr - DEVREGSTRT_ADDR)
              / (line_size)) + 3;
int dev_reg_start_addr = ((line_no - 3) *
                          (line_size) + DEVREGSTRT_ADDR);
int device_no = ((unsigned int) a1_cmdAddr - dev_reg_start_addr)
                / DEVREGSIZE;
```

dove *DEVREGSTRT_ADDR* è una costante inizializzata all'indirizzo *0x10000054*.

-6. **Get_CPU_Time**: Viene restituito il tempo di esecuzione in microsecondi di *current_p*.

-7. **Wait_For_Clock**: Viene effettuata una P sul semaforo dell'Interval Timer (e quindi blocca il processo fino al prossimo tick del dispositivo).

-8. **Get_SUPPORT_Data**: Viene restituito un puntatore alla struttura di supporto di *current_p*.

-9. **Get_Process_ID**: Viene restituito il pid di *current_p* se il parametro *parent != 0*, altrimenti viene restituito il pid del suo genitore.

-10. **Yield**: Questa syscall permette al processo corrente di "cedere" la CPU agli altri processi.

NB: potrebbe non bastare fare una semplice *insertProcQ()* del processo corrente nella sua ready queue corrispondente perchè, nel caso il processo corrente fosse ad alta priorità e la *ready_hq* fosse vuota, non verrebbe ceduto il controllo della CPU ai processi a bassa priorità in attesa.

3.3.2 Pass Up Or Die

Se vale 1 <= *ExcCode* <= 3 o 4 <= *ExcCode* <= 7 o 9 <= *ExcCode* <= 12 o in caso di syscall non *syscode* non-negativo, l'eccezione è gestita dalla funzione Pass Up Or Die.

- Se il processo non ha specificato un modo per gestire l'eccezione, viene terminato;
- Altrimenti, viene "passata" la gestione dell'eccezione al passupvector della support struct.

3.4 Interrupts.c

Se vale *ExcCode == 0* allora l'*exception_handler* si occupa di passare l'esecuzione alla funzione *interrupt_handler()*, che si occupa di fare il trigger del corretto handler da chiamare in base alla priorità della linea che ha un interrupt in corso secondo il campo **Cause.IP** (letto tramite *ip = exception_state->cause&IMON* e shiftando il valore ottenuto di 8 bit verso destra poichè il campo IP occupa i bit 8-15 del *Cause Register*):

3.4.1 Non Timer Interrupts

Se l'interrupt è stato generato da un device non-timer, allora prima di tutto è necessario calcolare l'indirizzo del device register appropriato cosí da poter assegnare al registro *v0* del processo che stava aspettando l'operazione di I/O lo *status code* del device register. Inoltre è necessario effettuare l'*acknowledge* dell'interrupt in corso.

Tali operazioni sono fatte dal seguente frammento di codice, all'interno della funzione *acknowledge()*:

```
switch (type) {
    case GENERAL_INT:
        to_unblock_proc->p_s.reg_v0 = dev_register->dtp.status;
        dev_register->dtp.command = ACK;
        break;
    case TERMTRSM_INT:
        to_unblock_proc->p_s.reg_v0 = dev_register->term.transm_status;
        dev_register->term.transm_command = ACK;
        break;
    case TERMRECV_INT:
        to_unblock_proc->p_s.reg_v0 = dev_register->term.recv_status;
        dev_register->term.recv_command = ACK;
        break;
}
```

dove *type* è la variabile che ci aiuta a distinguere, se si tratta di un interrupt generato da un device di tipo terminale, a quale tra i due tipi di sub-device ci si riferisce (*transm* o *recv*).

Il valore di *type* è calcolato in base alla linea (deve valere *line == 7*) e in base al valore dello status code del sub-device register di tipo *transm*. A seguire lo pseudocodice che descrive come viene calcolato *type*:

```
if (line != TERMINT)
    type = GENERAL_INT;
else if (dev_reg->term.transm_status != READY
         && dev_reg->term.transm_status != BUSY)
    type = TERMTRSM_INT;
else
    type = TERMRECV_INT;
```

NB: Si noti che è stato tenuto conto che gli interrupt da parte di sub-device di tipo *transm* hanno priorità piú alta rispetto ai sub-device di tipo *recv*.

L'indirizzo del device register che ha generato l'interrupt è calcolato in base al valore, alla linea e al numero del device che ha generato l'interrupt (*device_interrupting*, è ricavato dalla interrupting device bitmap [*Subsubsection 5.2.2, pops*]). Di seguito è riportato il frammento di codice che si occupa di fare ciò:

```
memaddr *bitmap_word_addr = (memaddr *) ((BITMAPSTRT_ADDR)
                            + (line - 3) * 0x04);
int device_interrupting = get_dev_interrupting(bitmap_word_addr);
memaddr dev_reg_addr = (memaddr) (DEVREGSTRT_ADDR
                       + ((line - 3) * 0x80)
                       + (device_interrupting * 0x10));
devreg_t *dev_reg = (devreg_t *) dev_reg_addr;
```

dove *get_dev_interrupting* è la funzione che, dato l'indirizzo di inizio della word della *line* che ha generato l'interrupt, restituisce il numero del device che ha generato l'interrupt.

3.4.2 PLT Interrupt

Se l'interrupt è stato generato dal PLT, il *current_p* ha terminato il suo quanto di tempo, quindi deve tornare nella ready queue e deve essere chiamato lo *scheduler()*. La funzione *plt_handler()* si occupa di eseguire tutte le operazioni appena descritte.

3.4.3 Interval Timer interrupt

Se l'interrupt è stato generato dall'interval timer, devono essere liberati tutti i processi che stavano aspettando sul semaforo di indice 0 (*sem[0]*) e, se possibile, deve essere continuata l'esecuzione del processo corrente. Il gestore di tale interrupt è implementato da *interval_handler()*.

4 Debugging

Per debuggare l'intero codice, ci siamo avvalsi di alcuni strumenti messi a disposizione dall'emulatore tra cui la possibilità di aggiungere breakpoints, le suspect memory regions e le traced memory regions, insieme alla libreria che ci è stata fornita per la stampa di messaggi in memoria cosí da poterli visualizzare in memoria.

Utilizzando le suspect e le traced memory regions abbiamo potuto tracciare il contenuto di alcune variabili cosí da riuscire a capire quando e dove il nostro codice era sbagliato (e.g. abbiamo avuto problemi con il valore di un semaforo il cui valore era corrotto e, grazie all'emulatore, ci siamo resi conto che veniva incrementato dopo aver eseguito una *terminate_process*, abbiamo provveduto subito).