Laura Bugo (laura.bugo@studio.unibo.it) Giulia Cantini (giulia.cantini2@studio.unibo.it) Silvia Severini (silvia.severini3@studio.unibo.it)	**Progetto Sistemi Operativi Fase 2**	Anno accademico: 2015/2016

INTRODUZIONE

Il progetto del sistema operativo JaeOS16 e' stato sviluppato sull'emulatore uARM.
Si articola in tre fasi:

- Fase 0: creazione delle Macro necessarie alla gestione di una lista circolare (clist.h);
- Fase 1: implementazione delle strutture PCB e ASL con le relative funzioni per la loro gestione
- Fase 2: creare un ambiente in cui processi si avvicendino condividendo il processore. Il nucleo fornisce gestione delle eccezioni, program trap e eccezioni TLB e sincronizzazione tra processi.

La struttura del PCB

*struct pcb_t *p_parent:* puntatore al padre
*struct semd_t *p_cursem:* puntatore al semaforo su cui e' bloccato
state_t p_s: stato del processo
struct clist p_list: puntatore alla coda della lista dei processi
struct clist p_children: puntatore alla coda della lista dei figli
struct clist p_siblings: puntatore alla coda della lista dei fratelli
pid_t pid: identificativo del processo (vedi sez "Assegnazione dei pid**"**)
int res_wait: risorse richieste da un processo bloccato (sempre <= 0)

*cputime_t kernel_time: tempo di lavoro del kernel (*vedi sez...)
cputime_t global_time: kernel time + user time
state_t oldnew_areas[EXCP_COUNT]: exception state vector per memorizzare Old e New Areas

int tags[3]: I tag si riferiscono rispettivamente a syscall, tlb e program trap (tags[i] vale 1 se ha già chiamato il proprio handler, 0 altrimenti)

La struttura dei Semafori

*int *s_semAdd:* puntatore al semaforo
struct clist s_link: puntatore alla coda della lista dei semafori attivi
struct clist s_procq: lista dei processi bloccati sul semaforo

FASE 2

Il progetto si compone dei seguenti file:

- initial.c
- scheduler.c
- exceptions.c
- syscall.c
- interrupt.c

INITIAL.C

Contiene il main() e si occupa dell'inizializzazione delle strutture ed aree di memoria necessarie al corretto funzionamento del sistema.
Vengono inizializzate le new area con l'interrupt handler, il TLB handler, il program trap handler e il syscall handler.
Viene settato lo stato del processore in kernel mode e vengono mascherati gli interrupt.
Il main si occupa anche di creare il primo processo che andra' in esecuzione con i campi relativi al tempo settati a 0 e richiama lo scheduler con flag relativo alla modalita' di esecuzione settato a "START".
D'ora in avanti il controllo non tornera' piu' al main.

SCHEDULER.C

Inizialmente vengono mascherati gli interrupt per evitare che possano interferire nello scheduling dei processi e verranno riabilitati alla fine.
Lo scheduler e' strutturato in quattro diverse modalita' di lavoro a seconda del valore della flag:

1 - START: sara' tale quando lo scheduler viene chiamato per la prima volta dal main(). Viene settato l'interval timer a 5 ms, inizializzato a 0 lo pseudoclockflag (verra' posto a 1 solo quando saranno scaduti i 100 ms), settato lo pseudo_start al tempo corrente (verra' usato per calcolare i 100 ms) e viene chiamata la funzione loadnext().

2 - RESET: sara' tale quando lo scheduler viene richiamato dal timer handler ad indicare che sono scaduti i 100 ms e che quindi vanno rinizializzate le variabili relative alla gestione di questo caso (pseudoclockflag e pseudo_start) e viene richiamata la loadnext().

3 - LOADNEXT: sara' tale quando deve essere caricato il prossimo processo (non sono scaduti i 100ms e il processo corrente e' terminato oppure si e' bloccato su un semaforo cioe' il current process e' stato messo a null). Richiama quindi la funzione loadnext().

4 - LOADCURRENT: sara' tale quando deve essere ricaricato il processo corrente (non e' scaduto il timer e non e' stata sollevata un'eccezione che ha annullato il current process) . Cio' sara' fatto abilitando gli interrupt del processo e ricaricandolo.

loadnext() : se il current process non e' nullo (ha finito il timeslice) sistema il global time, inserisce il processo nella ready queue e lo annulla.
Poi controlla se stanno per scadere i 100 ms (approssimando a 96ms perche' non ci sarebbe piu' tempo per l'esecuzione di un'intera time slice) e in caso affermativo setta il timer al tempo rimanente allo scadere, pone a 1 lo pseudoclockflag che servira' da indicazione al timer handler e setta il nuovo pseudo_start. In caso negativo setta normalmente il timer a 5ms.
Infine si occupa di mandare in esecuzione il prossimo processo pronto nella readyqueue settando lo startTOD (tempo di inizio di esecuzione), abilitando gli interrupt del processo.
Da notare che se l'estrazione del prossimo processo dalla ready queue non va a buon fine significa che quest'ultima era vuota percio' si procede ad una deadlock detection nelle modalita' specificate nella documentazione.

EXCEPTION.C

In ogni handler presente si svolgono le seguenti azioni:

- vengono mascherati gli interrupt cosi' che non possando interferire nella gestione dell'eccezione sollevata e verranno riabilitati alla fine;
- viene memorizzato il tempo attuale (kernel_start) che servira' al termine dell'handler per calcolare il tempo trascorso al suo interno ed aggiornare il kernel time del processo.

copy_state(state_t *dest, state_t *src): funzione ausiliaria per copiare lo stato da sorgente a destinazione

pgmtrap_handler(): gestisce un'eccezione di tipo programtrap che avviene quando si cerca di eseguire azioni non consentite.

tlb_handler(): gestisce un'eccezione di tipo tlb che avviene quando si verifica un errore di traduzione di indirizzi da virtuali a fisici.

syscall_handler(): gestisce una richiesta di system call o di break.

Setta il flag per lo scheduler a LOADCURRENT che eventualmente verra' modificato all'interno delle system call.

La gestione delle system call si divide in base al numero della system call richiesta ed alla modalita' di esecuzione (kernel o user mode):

- Se si richiede una system call tra 1 e 11 in user mode si solleva un'eccezione di tipo program trap;
- Se si richiede una system call tra 1 e 11 in kernel mode si richiama la funzione per eseguirla in syscall.c e si richiama lo scheduler;
- Se si richiede una system call con valore maggiore di 11 e non si e' specificato in precedenza un handler di alto livello, allora si termina il processo e si richiama lo scheduler, altrimenti si sistemano le varie aree di memoria e si passa il controllo all'handler specificato.

SYSCALL.C

pid_generator(): assegna un identificativo al processo che viene creato.

int create_process(state_t *statep): crea un nuovo processo e lo inserisce nella ready queue

void terminate_process(pid_t pid):

- se il pid e' uguale a zero o uguale a quello del current process si terminano i figli di quest'ultimo con la funzione ausiliaria terminate_children e poi si termina il processo stesso settando il flag dello scheduler a LOADNEXT.

  void terminate_children(struct pcb_t *p, struct pcb_t *p_const);

- altrimenti ricerca il pid indicato tramite la lookfor_progeny la quale si occupa anche di terminare il processo trovato e la sua progenie

  void lookfor_progeny(struct pcb_t *parent, pid_t pid, int *found)

void semaphore_operation(int *semaddr, int weight): si esegue una verhogen o una passeren in base al segno di weight sul semaforo specificato. Se vengono richieste delle risorse attualmente non disponibili allora il processo viene bloccato sul semaforo, si aggiorna il campo res_wait e viene messo a null con conseguente modifica del flag per lo scheduler a LOADNEXT. Altrimenti se vengono rilasciate delle risorse si aggiorna il valore del semaforo e si sbloccano i processi bloccati su di esso se possibile. Per verificare se le risorse rilasciate sono sufficienti a sbloccare il primo processo sulla coda dei bloccati e' dichiarata una variabile che contiene il numero di risorse non ancora assegnate.

specify_sysbp, specify_tlb, specify_pgmtrap: servono a specificare un proprio handler del processo che sia diverso da quello di default.

void exit_trap(unsigned int excptype, unsigned int retval): carica lo stato del processore per ritornare da un handler di alto livello.

void get_cputime(cputime_t *global, cputime_t *user): ritorna i valori del global e user time del processo.

void wait_clock(): esegue una V sul semaforo dello pseudoclock.

unsigned int io_devop(unsigned int command, int intlNo, unsigned int dnum): esegue l'operazione indicata dal parametro command e blocca il processo sul semaforo del device corretto.

INTERRUPT.C

void interrupt_handler(): Gestisce tutti gli interrupt che vengono sollevati.

Sono mascherati gli interrupt cosi' che non possano interferire nella gestione dell'eccezione sollevata e verranno riabilitati alla fine. Si memorizza il tempo attuale (kernel_start) che servira' al termine dell'handler per calcolare il tempo trascorso al suo interno ed aggiornare il kernel time del processo. Viene settata la flag per lo scheduler a LOADNEXT. E' importante notare che il program counter dell'eventuale processo corrente deve essere decrementato di WORD_SIZE cosi' da poter ripartire dal momento precedente al sollevamento dell'interrupt. A seconda della causa dell'interrupt, si esegue il device handler, il terminal handler o il timer handler.

void dev_handler(int int_type): funzione ausiliaria che gestisce gli interrupt da device.

void timer_handler(): verifica attraverso la pseudoclockflag se sono terminati i 100 ms e in

caso affermativo setta il flag dello scheduler a RESET e sblocca i processi bloccati sul semaforo dello pseudoclock.

void terminal_handler(): gestisce gli interrupt da terminale, distinguendo se il terminale e' in scrittura o in lettura.

unsigned int find_dnum(unsigned int *dev_bitmap): funzione ausiliaria utilizzata all'interno della dev_handler e terminal_handler per per cercare il corretto numero di device a partire dalla linea di interrupt.

ASSEGNAZIONE DEL PID

Nella allocPcb, si considera l'indirizzo del procBlock che e' in testa alla pcbFree e lo si assegna temporaneamente a pid. In seguito, nella create_process, viene assegnato al pid il valore di ritorno della funzione pid_generator, che concatena il valore salvato in pid con la variabile counter incrementata ogni volta.

GESTIONE DEL TEMPO

uARM possiede un solo timer che si utilizza per mandare interrupt ogni 5 ms.

Per poter gestire anche lo scadere dei 100ms, vengono introdotte le variabili pseudo_start e pseudoclockflag. La prima e' utilizzata per memorizzare il valore del tempo attuale all'inizio e ogni volta che scadono i 100ms. La seconda e' una variabile di controllo che viene settata a TRUE quando stanno per scadere i 100ms.

Queste variabili vengono aggiornate quando ci si trova nello scheduler con flag che ha valore START o RESET.

Nella funzione loadnext() si controlla se stanno per scadere i 100 ms facendo la differenza tra il tempo attuale e la variabile pseudo_start e confrontandola con la differenza tra SCHED_PSEUDO_CLOCK e SCHED_TIME_SLICE (95 ms). E' stato scelto di fare questo confronto approssimando i 100ms perche’ e' piu' vicina la scadenza di quest'ultimi rispetto al quella dei 5ms. Viene quindi settato il time slice al tempo rimanente allo scadere dei 100ms, viene posto a 1 lo pseudoclockflag che indichera' al timer handler come procedere e setta il nuovo pseudo_start al tempo corrente.
Se il confronto non viene verificato, setta normalmente il timer a 5ms.
Nel timer_handler si controlla attraverso la pseudoclockflag se sono terminati i 100 ms e in caso affermativo si setta il flag dello scheduler a RESET e si sbloccano i processi bloccati sul semaforo dello pseudoclock. Alla fine viene settato il timer a SCHED_BOGUS_SLICE per mandare un ack all'interval timer ed evitare che il timer scatti prima di ritornare nello scheduler.