BiKayaOS

Progetto Sistemi operativi – Università di Bologna, 2020

1 INTRUDUZIONE

BiKaya è un sistema operativo basato su un'architettura a 6 livelli proposta da Edsger Dijkstra e ideata per l'istruzione alla programmazione di sistemi operativi.

Il progetto compiuto consiste nell'implementazione in due architetture (uARM e uMPS) del Kernel (livello 3) dell'architettura Kaya, completo di supporto per 8 syscall e gestori di interrupt hardware, trap ed eccezioni TLB.

2 STRUTTURA

I file sono divisi in modo funzionale, in base al contenuto:

- *asl* – funzioni per operazioni sui semafori/liste di semafori;
- *exceptions* – gestori e funzioni di aiuto per le eccezioni (syscall, trap e TLB);
- *init* – funzioni di startup e inizializzazione;
- *interrupts* – gestore e funzioni di aiuto per gli interrupt;
- *pcb* – funzioni per operazioni sui blocchi di controllo dei processi (pcb) /liste di pcb;
- *scheduler* – scheduler e funzioni di aiuto.
- *utils* – funzioni varie usate in più contesti.

3 SCELTE DI SVILUPPO

1.1 CRITICAL_WRAPPER

Il critical_wrapper è una funzione che funge da wrapper per le funzioni che rappresentano codice critico (syscall, handler di interrupt/trap e eccezioni TLB) e aggiorna il tempo kernel di tutti i processi.

Per essere compatibili con il wrapper tutte le syscall e le funzioni che astraggono gli handler degli interrupt dei device seguono la struttura:

```
int call(state_t* callerState)
```

Le funzioni syscall ritorneranno TRUE se è necessario richiamare lo scheduler (ad esempio se il processo corrente è bloccato o è stato terminato) o ricaricare lo stato del chiamante e riprendere l'esecuzione dall'istruzione successiva alla chiamata di syscall.

1.2 WAIT E IDLE_PROC

In inizializzazione è creato un processo fittizio idle, il quale program counter punta alla funzione `idle_proc`; Questo processo che fungerà da wait è posto in fondo alla coda dei processi, con priorità negativa e non è soggetto ad aging. In questo modo lo scheduler lo estrarrà dalla readyQueue solo quando ogni altro è bloccato.

1.3 GESTIONE TEMPO PROCESSI

Il time slice considera sia il tempo trascorso in modalità kernel che user. Il tempo trascorso in kernel mode non è sommato all'interval timer prima del mode switch a user mode.

1.4 SEMAFORI

I semafori device sono separati dai semafori “utente”: i semafori device sono opportunamente inseriti in ASL, ma in rimozione non vengono reimmessi nella lista dei semafori liberi.

4 DESCRIZIONE VARIABILI GLOBALI

4.1 ASL

- `semdev semDev:`
  Struttura contenente un array di strutture semafori (semd_t) per ogni device. Ogni array avrà dimensione il numero di linee di interrupt per ogni device.

1.5 SCHEDULER

- `struct list_head readyQueue:`
  Coda dei processi simbolicamente in stato ready. Contiene i processi in attesa di essere eseguiti.
- `pcb_t* currentProcess:`
  Puntatore al pcb del processo correntemente in esecuzione.
- `int processCount:`
  Contatore del numero di processi attivi; considera i processi pronti e in attesa/bloccati. Necessario per deadlock detection/halt.
- `int blockedCount:`
  Contatore dei processi bloccati su semafori device. Necessario per deadlock detection/halt.
- `pcb_t* idle_ptr:`
  Puntatore al pcb del processo idle.

5 DESCRIZIONE STRUTTURE DATI (MODIFICATE)

Le strutture dati usate si trovano nel file `types_bikaya.h`.

I campi aggiunti nella struttura `pcb_t` sono:

1.6 PCB

1.6.1 Campi tempo

- `cpu_time user_timer:`
  Tempo trascorso in user mode.
- `cpu_time kernel_timer:`
  Tempo trascorso in kernel mode.
- `cpu_time first_activation:`
  TOD del primo avvio del processo.
- `cpu_time last_restart:`
  TOD dell’ultimo restart del processo.
- `cpu_time last_stop:`
  TOD dell’ultimo mode switch a kernel.

1.6.2 Puntatori a stati

- `state_t* sysNew; state_t* sysOld:`
  Stato dell’handler di syscall custom.
- `state_t* TlbNew; state_t* TlbOld:`
  Stato dell’handler di eccezioni TLB custom.
- `state_t* pgtNew; state_t* pgtOld:`
  Stato dell’handler di trap custom.

6 DESCRIZIONE FUNZONI

6.1 ASL

- `semd_t* getSemd(int *key):`
  Ritorna il semaforo in ASL con chiava pari a key, null se non trovato.
- `void initASL():`
  Inizializza la tabella dei semafori dei processi e li inserisce della lista dei semafori liberi, ed initializza la chiave dei semafori dei dispositivi.
- `void insertSem(semd_t* semaphore):`
  Inserisce semaphore nella ASL, con priorita in base alla chiave.
- `int insertBlocked(int *key,pcb_t* p):`
  Inserisce p nel semaforo in ASL con chiave key, oppure se il semaforo non è in ASL ne alloca uno e lo inserisce in ASL.
- `pcb_t* removeBlocked(int *key):`
  Rimuove e ritorna il primo pcb bloccato dal semaforo con chive key. Se il semaforo diventa vuoto lo rimuove da ASL e lo si reinserisce in nella lista dei semafori liberi.
- `pcb_t* removeBlockedonDevice(int* key):`
  Rimuove il processo bloccato nel semaforo del device con chiave pari a key. Se il semaforo diventa vuoto lo rimuove da ASL.
- `pcb_t* outBlocked(pcb_t *p):`
  Ritorna e rimuove il pcb puntato da p dal suo semaforo, se il semaforo diventa vuoto viene tolto da ASL e inserito nella lista dei semafori liberi.
- `pcb_t* DevicesOutBlocked(pcb_t* p):`
  Ritorna e rimuove il pcb puntato da p dal suo semaforo per I device, se il semaforo diventa vuoto viene tolto da ASL.
- `unsigned int isBlocked(pcb_t* p):`
  Ritorna true se p è bloccato in un semaforo.
- `pcb_t* headBlocked(int *key):`
  Ritorna il primo pcb bloccato nel semaforo con chiave key, senza rimuoverlo.
- `void outChildBlocked(pcb_t *p):`
  Rimuove p e tutti I pcb radicati in p dai rispettivi semafori.

6.2 EXCEPTIONS

6.2.1 Handlers:

- `void trapHandler():`
  Gestore trap interrupts, evocato quando una trap è sollevata.
- `void syscallHandler():`
  Gestore syscall, evocato quando è chiamata alla funzione SYSCALL. Legge la cause di chiamata e chiama la syscall richiesta.
- `void TLBManager():`
  Gestore TLB, evocato quando c'è una violazione delle aree di memoria.

6.2.2 Syscalls:

- `int get_cpu_time(state_t* callerState):`
  SYS1. Ritorna tempo user, kernel e totale del processo chiamante.
- `int create_process(state_t* callerState):`
  SYS2. Crea un nuovo processo figlio del chiamante.
- `int kill(state_t* callerState):`
  SYS3. Termina il processo richiesto e l'intera progenie.
- `int verhogen(state_t* callerState):`
  SYS4. Effettua una V sul semaforo passato.

- `int passeren(state_t* callerState):`
  SYS5. Effettua una P sul semaforo passato.
- `int do_IO(state_t* callerState):`
  SYS6. Richiede un'operazione I/O sul registro indicato.
- `int spec_passup(state_t* callerState):`
  SYS7. Specifica e salva un nuovo handler per syscall/trap/TLB.
- `int get_pid_ppid(state_t* callerState):`
  SYS8. Ritorna il pid e/o il pid del parent del processo attuale.

6.2.3 Funzioni di supporto:

- `int trapHelper(state_t* callerState):`
  Supporto per l'handler delle trap.
- `void recursive_kill(pcb_t* process):`
  Supporto per la (kill) SYS3. Termina I processi figli di `process`.
- `int TLBHelper(state_t* callerState):`
  Supporto per l'handler di eccezioni TLB.

6.3 INIT

- `void newArea(unsigned int address, unsigned int handler):`
  Imposta il registro di stato puntato da `address` per gestire syscall/interrupts con l'handler puntato da `handler`.
- `void newProcess(memaddr functionAddr, int priority):`
  Imposta l'area di memoria per un nuovo processo da far partire dopo lo startup.
- `void initROM():`
  Imposta gli handler per gestire syscall, trap, violazioni TLB e interrupts.
- `void initData():`
  Istanzia le strutture dati per processi e semafori.
- `void initialProcess():`
  Crea tutti i processi da far partire.
- `int main():`
  Chiama tutte le funzioni precedenti e infine lo scheduler.

6.4 INTERRUPTS

6.4.1 Handler:

- `void interruptHandler():`
  Gestore interrupt principale, evocato quando un interrupt è sollevato. Controlla le richieste in entrata e le inoltra al corretto subhandler mantenendo la priorità del device.

Sub-handlers:

- `int timerHandler(state_t* oldstatus):`
  Gestore per interrupt del interval timer.
- Handler di device (`diskHandler, tapeHandler, netHandler, printerHandler, termHandler`):
  Inoltrano la richiesta a `devHandler` passando la corretta linea di interrupt.

6.4.2 Funzioni di supporto:

- `void devHandler(int line, state_t* oldstatus):`
  Gestore per interrupt di linea>3, gestisce sia gli interrupt generici che quelli di terminale.
- `void exitInterrupt(state_t* oldstatus):`
  Operazioni di chiusura del gestore dell'interrupt. Copia il vecchio stato nello stato del processo attuale e reinserisce il processo corrente nella coda dei processi.

6.5 PCB

- `void initPcbs(void)`:
  Inizializza la tabella dei pcb e li inserisce nella lista dei pcb liberi.
- `void freePcb(pcb_t *p)`:
  Aggiunge il pcb puntato da p nella lista libera.
- `pcb_t *allocPcb(void)`:
  Estrae dalla lista dei pcb liberi un pcb e lo iniziallizza, ritorna il pcb inizializzato.
- `void mkEmptyProcQ(struct list_head *head)`:
  Inizializza una listhead, con entrambi I campi di head che puntano a se stessa.
- `int emptyProcQ(struct list_head *head)`:
  Ritorna true, se la lista puntata da head è vuota.
- `void insertProcQ(struct list_head *head, pcb_t *p)`:
  Inserisce il pcb puntato da p nella lista puntata da head, a seconda della priorita di p.
- `pcb_t *headProcQ(struct list_head *head)`:
  Ritorna il primo pcb nella lista puntata da head.
- `pcb_t *removeProcQ(struct list_head *head)`:
  Rimuove e ritorna il primo pcb dalla lista puntata da head.
- `pcb_t *outProcQ(struct list_head *head, pcb_t *p)`:
  Rimuove e ritorna il pcb puntato da p nella lista puntata da head, ritorna null se p non appartiene alla lista.
- `int emptyChild(pcb_t *this)`:
  Ritorna true se il pcb puntato da this non ha figli.
- `void insertChild(pcb_t *prnt, pcb_t *p)`:
  Inserisce p come figlio di prnt.
- `pcb_t *removeChild(pcb_t *p)`:
  Rimuove e ritorna il primo figlio di p.
- `pcb_t *outChild(pcb_t *p)`:
  Ritorna e rimuove p dalla lista dei figli di p->parent.

6.6 SCHEDULER

- `void schedInsertProc(pcb_t* process)`:
  Inserisce il pcb `process` nella readyQueue.
- `void aging()`:
  Esegue il meccanismo di aging su tutti i processi nella readyQueue (escluso idle_proc).
- `void scheduler()`:
  Nucleo dello scheduler. Realizza uno scheduler preemptive, round-robin a priorità, con aging. Estrae un processo P dalla readyQueue, inizializza il time slice in cui P esisterà, aggiorna i tempi di attivazione e restart e infine carica lo stato di P. È anche presente deadlock detection, che evoca kernel panic nel caso di deadlock.

6.7 UTILS

- `void mymemcpy(void *dest, void *src, int n)`:
  Funziona come una memcpy, copia l'area da memoria da dest a src.
- `int critical_wrapper(int (*call)(), state_t* callerState, cpu_time start_time, pcb_t* currentProcess)`:
  Tiene traccia del tempo user e kernel in un modo ordinato. Chiama la funzione passata tramite il puntatore `call` e ritorna TRUE o FALSE a seconda se deve essere chiamato lo scheduler o ripresa l'esecuzione del processo corrente alla fine della zona critica.
- `cpu_time update_user_time(pcb_t* currentProcess)`:
  Funzione di aiuto del critical_wrapper. Aggiorna il tempo utente e ritorna il TOD di chiamata.
- `void set_return(state_t* caller, int status)`:
  Imposta il registro di ritorno di una syscall del `caller` a `status`.
- `void set_command(devreg_t* reg, unsigned int command, int subdevice)`:
  Imposta il comando `command` nel registro di device in `reg`. Il `subdevice` rappresenta se il terminale è in trasmissione o in ricezione.

- `void get_line_dev(devreg_t* reg, int* line, int* dev):`
  Ritorna linea e numero di device dato il registro di un device. Il calcolo è fatto trovando l'offset tra l'indirizzo del registro e l'inizio dell'area di memoria dei registri di device, applicando il dividendo per la dimensione di ogni blocco di registri dei device (<dimensione registro> * <numero di linee ogni interrupt>) trovare la linea e applicando il modulo per trovare il numero di device.
- `unsigned int get_status(devreg_t* reg, int subdevice);`
  `unsigned int tx_status(termreg_t *tp);`
  `unsigned int rx_status(termreg_t *tp):`
  Ritorna lo stato di un device generico o terminale (tx_status per il transm status, rx_status per il recv status).
- `void debug(int, int):`
  Funzione di debug per leggere parametri con coppia chiave/valore.
- `semd_t * getSemDev(int line, int devnum, int read):`
  Ritorna la struttura dati del semaforo device richiesto da `line`, `devnum` e `read`.
- `void verhogenKill(pcb_t* process):`
  Effettua le operazioni necessarie sui semafori a seguito di una SYS3.
- `void verhogenDevice(int line, unsigned int status, int deviceNumber, int read):`
  Effettua una V sul semaforo del device, a seguito della gestione di un interrupt.
- `void ACKDevice(unsigned int* commandRegister):`
  Invia un segnale di ACK al device specificato (scrive il valore di ACK nel registro specificato).
- `int pid_in_readyQ(pcb_t* pid):`
  Ritorna true se il pcb `pid` è nella coda dei processi in attesa.
- `int set_register(unsigned int reg, unsigned int val):`
  Imposta nel registro `reg` il valore `val`.
- `int log2(unsigned int n):`
  Implementazione ricorsiva del logaritmo binario. Supporto della funzione `getDeviceNumber`.
- `int lowest_set(int number):`
  Trova il LSB di `number`, necessario come supporto di `getDeviceNumber`.
- `int getDeviceNumber(int line):`
  Ritorna il numero di device del device alla linea che sta sollevando l'interrupt `line`. Per ottenere la linea di interrupt partendo dalla causa si prende la posizione ordinale da destra (trovata con `log2`) del bit di minimo valore (trovata con `lowest_set`) della causa di interrupt.