Relazione µPandOS: Phase 2

June 9, 2024

Contents

1 partecipanti

- Erik Dervishi 0001069599
- Gianluca Sperti 0001078941
- Francesco Fabio Di Mauro 0001078706
- Lorenzo Novi 0001090067

2 dep.h

È un header file che contiene dichiarazioni di variabili esterne, funzioni e macro utilizzate in altre parti del progetto.

```
extern int processCount, softBlockCount, currPid;

extern cpu_t prevTod;

extern struct list_head ready_queue;
extern struct list_head blockedForClock;

extern struct list_head blockedForDevice[NDEV];
```

3 Inizializzazione

L'inizializzazione del kernel avviene nella funzione `main`. I passaggi chiave includono la configurazione del vettore di passaggio, l'inizializzazione dei blocchi di controllo dei processi (PCB) e delle strutture di gestione dei messaggi, e la creazione dei processi iniziali. Codice:

```
int main(int argc, char const *argv[]) {
    passupvector_t *passupv;
    passupv = (passupvector_t *) PASSUPVECTOR;
    passupv->tlb_refill_handler = (memaddr) uTLB_RefillHandler;
    passupv->tlb_refill_stackPtr = (memaddr) KERNELSTACK;
    passupv->exception_handler = (memaddr) exceptionHandler;
    passupv->exception_stackPtr = (memaddr) KERNELSTACK;

    initPcbs();
    initMsgs();

    processCount = 0;
    softBlockCount = 0;
    currPid = 3;
```

```
    for(int i = 0; i < NDEV; i++){
        INIT_LIST_HEAD(&blockedForDevice[i]);
    }

    LDIT(PSECOND);

    ssi_pcb = allocPcb();
    ssi_pcb->p_pid = 1;
    ssi_pcb->p_supportStruct = NULL;
    ssi_pcb->p_s.status = ALLOFF | IEPON | IMON | TEBITON;
    RAMTOP(ssi_pcb->p_s.reg_sp);
    ssi_pcb->p_s.pc_epc = (memaddr) SSI_entry_point;
    ssi_pcb->p_s.gpr[24] = ssi_pcb->p_s.pc_epc;
    insertProcQ(&ready_queue, ssi_pcb);
    processCount++;

    pcb_t *toTest = allocPcb();
    toTest->p_pid = 2;
    toTest->p_supportStruct = NULL;
    toTest->p_s.status = ALLOFF | IEPON | IMON | TEBITON;
    RAMTOP(toTest->p_s.reg_sp);
    toTest->p_s.reg_sp -= (2 * PAGESIZE);
    toTest->p_s.pc_epc = (memaddr) test;
    toTest->p_s.gpr[24] = toTest->p_s.pc_epc;
    insertProcQ(&ready_queue, toTest);
    processCount++;

    scheduler();
}
```

In questa parte di codice, il kernel viene inizializzato, configurando il vettore di passaggio per la gestione delle eccezioni, inizializzando le strutture dei PCB e delle code dei processi, e creando due processi iniziali: uno per l'interfaccia del servizio di sistema (SSI) e uno per i test.

4 Interfaccia del Servizio di Sistema (SSI)

La SSI fornisce un insieme di chiamate di sistema per la gestione dei processi, la comunicazione tra processi e le operazioni di I/O dei dispositivi. La funzione di ingresso della SSI attende continuamente i messaggi in arrivo, elabora le richieste SSI e invia le risposte al mittente.

Codice:

```
void SSI_entry_point() {
    while (1) {
        unsigned int payload;
        unsigned int sender = SYSCALL(RECEIVEMESSAGE, ANYMESSAGE, (unsigned int)
        int ssiResponse = handle_request((pcb_t *)sender,
        (ssi_payload_t*)payload);
        if (ssiResponse != NOSSIRESPONSE)
            SYSCALL(SENDMESSAGE, (unsigned int)sender, ssiResponse, 0);
    }
}
```

In questa parte di codice, la funzione `SSI_entry_point` gestisce le richieste SSI in un ciclo infinito, ricevendo messaggi, elaborando le richieste e inviando risposte.

5 Gestione dei Processi

La gestione dei processi comprende la creazione, la terminazione e la gestione dello stato dei processi.

Codice per la creazione di un processo:

```
int createProcess(state_t *statep, support_t *supportp) {
    pcb_t *new_pcb = allocPcb();
    if (new_pcb == NULL) return -1;

    new_pcb->p_s = *statep;
    new_pcb->p_supportStruct = supportp;
    insertProcQ(&ready_queue, new_pcb);
    processCount++;
    return new_pcb->p_pid;
}
```

In questa parte di codice, la funzione `createProcess` alloca un nuovo PCB, lo inizializza con lo stato fornito e lo inserisce nella coda dei processi pronti.

Codice per la terminazione di un processo:

```
void ssi_terminate_process(pcb_t* process) {

    while(!emptyChild(process))
    {
```

```
        ssi_terminate_process(removeChild(process));
    }


    if (outProcQ(&ready_queue, process) != NULL ||
        outProcQ(&blockedForClock, process) != NULL ||
        processWaitingDevice(process)
        ) {

        softBlockCount--;

    }

    outChild(process);
    freePcb(process);
    processCount--;
}
```

In questa parte di codice, la funzione `terminateProcess` rimuove il processo dalla coda dei processi pronti, libera il PCB e decrementa il conteggio dei processi.

6 Scheduler

La funzione `scheduler` gestisce l'esecuzione dei processi in base alla loro prontezza e stato di blocco. Funzionamento:

- Se un processo è disponibile nella coda dei processi pronti, imposta un timer, salva il tempo corrente e carica lo stato del processo.
- Se solo il processo SSI è in esecuzione, il sistema si ferma.
- Se ci sono processi ma nessuno pronto, e ci sono processi soft-blocked, il sistema attende un evento.
- Se non ci sono processi pronti e nessun processo soft-blocked, il sistema va in panico.

Codice dello Scheduler:

```
void scheduler() {
    if ((current_process = removeProcQ(&ready_queue))) {
        setTIMER(TIMESLICE * (*((cpu_t *)TIMESCALEADDR)));
        STCK(prevTod);
        LDST(&(current_process->p_s));
        return;
    }
```

```
    if (processCount == 1) {
        HALT();
    }
    else if (processCount > 0 && softBlockCount > 0) {
        setSTATUS(ALLOFF | IECON | IMON);
        WAIT();
    }
    else if (processCount > 0 && softBlockCount == 0) {
        PANIC();
    }
}
```

In questa parte di codice, la funzione `scheduler` gestisce la pianificazione dei processi, caricando lo stato del prossimo processo pronto, aspettando un evento se ci sono processi soft-blocked o andando in panico se non ci sono processi pronti.

7 Gestione delle Eccezioni

Le eccezioni sono gestite dalla funzione `exceptionHandler`, che delega a gestori specifici in base al tipo di eccezione:

- `IOINTERRUPTS`: Gestito dalla `interruptHandler`.
- `SYSEXCEPTION`: Gestito dalla `syscallHandler`.
- `PGFAULTEXCEPT` e `GENERALEXCEPT`: Gestito dalla funzione `handleException`, che trasferisce il controllo alla struttura di supporto del processo o termina il processo se non è disponibile una struttura di supporto.

Esempio di codice:

```
void exceptionHandler() {
    state_t *currentExceptionState = (state_t *)BIOSDATAPAGE;
    int currentCause = getCAUSE();
    int exceptionCode = (currentCause & GETEXECCODE) >> CAUSESHIFT;

    switch (exceptionCode) {
        case IOINTERRUPTS:
            interruptHandler(currentCause, currentExceptionState);
            break;
        case 1 ... 3:
            handleException(PGFAULTEXCEPT, currentExceptionState);
            break;
        case SYSEXCEPTION:
            syscallHandler(currentExceptionState);
            break;
```

```
        case 4 ... 7: case 9 ... 12:
            handleException(GENERALEXCEPT, currentExceptionState);
            break;
        default:
            PANIC();
            break;
    }
}
```

In questo esempio, la funzione `exceptionHandler` distingue tra diversi tipi di eccezioni e chiama i gestori appropriati per ognuna di esse.

8 Gestione degli Interrupt

Gli interrupt sono gestiti dalla funzione `interruptHandler`, che identifica l'origine dell'interrupt e chiama il gestore appropriato.

- `handleLocalTimerInterrupt`: Gestisce l'interrupt del timer locale, aggiornando il tempo CPU del processo corrente e salvando il suo stato.
- `handlePseudoClockInterrupt`: Gestisce gli interrupt del pseudo-orologio, sbloccando i processi in attesa sull'orologio.
- `handleNonTimer`: Gestisce gli interrupt specifici del dispositivo per terminali, dischi, memoria flash, Ethernet e stampanti. Determina il dispositivo che ha generato l'interrupt e gestisce l'I/O del terminale o dell'I/O non terminale di conseguenza.

Esempio di codice:

```
void handleNonTimer(int line_num, int cause, state_t *exc_state) {
    devregarea_t *dev_reg_area = (devregarea_t *)BUS_REG_RAM_BASE;
    unsigned int intr_devices_bitmap = dev_reg_area->interrupt_dev[line_num - DE
    unsigned int dev_status;
    unsigned int dev_num;

    if (intr_devices_bitmap & DEV7ON) dev_num = 7;
    else if (intr_devices_bitmap & DEV6ON) dev_num = 6;
    else if (intr_devices_bitmap & DEV5ON) dev_num = 5;
    else if (intr_devices_bitmap & DEV4ON) dev_num = 4;
    else if (intr_devices_bitmap & DEV3ON) dev_num = 3;
    else if (intr_devices_bitmap & DEV2ON) dev_num = 2;
    else if (intr_devices_bitmap & DEV1ON) dev_num = 1;
    else if (intr_devices_bitmap & DEV0ON) dev_num = 0;
    else return;

    pcb_t* unblocked_proc = NULL;
```

```
    if (line_num == IL_TERMINAL) {
        termreg_t *dev_reg = (termreg_t *)DEV_REG_ADDR(IL_TERMINAL, dev_num);
        if (((dev_reg->transm_status) & 0x000000FF) == 5) {
            dev_status = dev_reg->transm_status;
            dev_reg->transm_command = ACK;
            unblocked_proc = releaseProcessByDeviceNumber(dev_num, &blockedForTra
        } else {
            dev_status = dev_reg->recv_status;
            dev_reg->recv_command = ACK;
            unblocked_proc = releaseProcessByDeviceNumber(dev_num, &blockedForRe
        }
    } else {
        dtpreg_t *dev_reg = (dtpreg_t *)DEV_REG_ADDR(line_num, dev_num);
        dev_status = dev_reg->status;
        dev_reg->command = ACK;

        struct list_head *proc_list;

        if ((proc_list = &blockedForDevice[EXT_IL_INDEX(line_num)])) {
            unblocked_proc = releaseProcessByDeviceNumber(dev_num, proc_list);
        }
    }

    if (unblocked_proc) {
        unblocked_proc->p_s.reg_v0 = dev_status;
        sendMessage(ssi_pcb, unblocked_proc, (memaddr)(dev_status));
        insertProcQ(&ready_queue, unblocked_proc);
        softBlockCount--;
    }

    if (current_process)
        LDST(exc_state);
    else
        scheduler();
}
```