Pandos Phase 2 Documentation

Angelo Galavotti, Denis Pondini, Antonin Avdei, Adriano Pace

April 2021

1 Introduction

GitHub repository : https://github.com/Adrianorieti/Progetto_Sistemi

This Pdf represents the documentation for the second phase of the Pandos project. In this document the reader will find every single implementation decision made by the authors. For a better understanding of what Pandos is please follow the link to the official documentation www.cs.unibo.it/ renzo/so/pandos/docs/pandos.pdf

This level implements **kernel-mode** processes management and synchronization primitives in addition to **multiprogramming**, a **process scheduler**, **device interrupt handlers**, and **deadlock detection**. As a superior layer of **Phase 1** level, this level is a more complex challenge due to the subtle interactions between all the moving parts,rather than one single component. In more details, the phases's goal include a better understanding of the internal kernel logics, and guide the students to the creation of every (simplified) part of it, like for example the **scheduler**, the **exception handler**,the **system call handler**, and the **interrupt handler**.

2 Implementation

Here follow the most important decisions about the "Nucleus", or more simply the Kernel, making of.

2.1 Main.c

The main function consists of the very first function called and it's responsible for the "**Bios** initialization". The Bios is the software part located between the hardware and the kernel. The **init** process is created and correctly set. Also the **pass-up vector** is initialized with the addresses of the kernel handler for TLB-refill events and the value of the Stack pointer and the exception handler. In the end the Scheduler is called, and the control should never come back to Main.

2.2 Scheduler.c

The scheduler is in charge of the decision of processes rotation, in the Pandos project it is a simple preemptive round-robin algorithm. Whenever a new process has to be set in running state, the scheduler checks out whether the **ready-queue** is empty or not, in the first case the **HALT** function is raised, in the second case the process information contained in it's status are loaded and a time slice is tolerated. In case of deadlock the system ends up in **PANIC**.

2.3 ExceptionHandler.c

As said before, the **exception handler** address is stored in the pass-up vector during the initializing process by the kernel. It retrieves the saved exception state located at the start of the **BIOS Data Page** in order to extract the exception code from the cause register to check its value. For exec code 0 the interrupt handler is called , for 1 to 3 a TLB trap (i.e. page fault) is happening, exec code 8 means a syscall is happening.

2.3.1 DevInterruptHandler.c

Interrupt handling consist of various levels, the first level is in charge of reading the first bit set to 1 of the IP section of the **exception state's Cause register**, an 8-bit field indicating on which interrupt lines interrupts are currently pending. If an interrupt is pending on interrupt line ***i***, then **Cause.IP[i]** is set to 1. After that, the interrupt is then managed in the correct way.

- **Line interrupt 1** is for PLT Timer Interrupts, meaning that the Current Process has used up its time quantum/slice but has not completed its *CPU Burst*. Here the **PLT** is loaded with a new value (setTimer() is a builtin function that helps us in this way), the processor state at the time of the exception (located at the start of the BIOS Data Page) is saved into the Current Process's *pcb*, which is then placed int the ready queue, thus transitioning from *running* to *ready* state. In the end, the scheduler is called.

- **Line interrupt 2** is dedicated to the Interval Timer; with this interrupt we free the queue of blocked processes in the Interval Timer semaphore, hence we set its counter to 0 and once finished if there's a process which was blocked it will resume its state, otherwise if there's no current process the scheduler is called.

- **Lines interrupt 3 to 7** are for **non timer interrupts**, in this case many actions are executed. First, we scan the interrupt line's device bit map to see which device has had an interrupt. For each device whose bit map was on, a handler function is called (*Non Timer Handler*).

```
1     if (line > 2) /* Non-timer device interrupt line*/
2     {
```

```
3           memaddr* device = (memaddr*) (IDEVBITMAP + ((line - 3)
        * 0x4));
4           int mask = 1;
5           for (int i = 0; i < DEVPERINT; i++)
6           {
7               if (*device & mask)
8                   NonTimerHandler(line, i);
9               mask *= 2;
10          }
11      }
```

Inside this function, the status code from the device's device register is stored in a new fresh variable, and an ACK is sent to the device, to inform it that the interrupt is being managed. This is done by writing the acknowledge command code in the interrupting device's device register. A V operation is then executed to unblock the process that were waiting for the I/O operation (through the waitforIO syscall). The previously stored status code is inserted in the newly unblocked pcb's v0 register, this one is than inserted in the ready queue and get transitioned from *blocked* to *ready* state.

```
1       softBlockCount--;
2
3       device_semaphores[index] += 1;
4       pcb_t* unlockedProcess = removeBlocked(&device_semaphores[
        index]);
5
6       if (unlockedProcess != NULL)
7       {
8           unlockedProcess->p_s.reg_v0 = status_word;
9           unlockedProcess->p_semAdd = NULL;
10          unlockedProcess->p_time += CURRENT_TOD -
        interruptStartTime;
11          insertProcQ(&readyQueue, unlockedProcess);
12      }
13
```

The function ends after LDST action is performed, by which control is returned to current process. In case there was no running current process, the scheduler is called.

2.3.2 TLB refill events and program Trap exceptions

In case any of these type of exception occurs, a PassUpOrDie procedure is executed. This means that we're trying to make the Support Level handle the exception. If the support info is provided, then the we perform a LDCXT on the support fields of the current process, in order to pass the handling of the exception. If this information is not provided, then we simply kill the process.

2.3.3 Syscall.c

The complete description of every single syscall present in Pandos is given for the curious reader in the Pandos-manual, but we will briefly discuss how each

of them was implemented. A system call handler has the responsibility for the syscall to trigger depending on the number stored in the a0 field of the general purpose register.

1. **Create Process :** new pcb is allocated and loaded with the right infos for the process.

2. **Terminate Process :** a process is terminated, and recursively all progeny of this process are terminated as well. Auxiliary functions are used in order to make this happen. These auxiliary functions actuate the killing procedure of the progeny. Each process is removed from the ready queue and, if it's blocked, the process is removed from its semaphore queue. Then, the process descriptor is freed. After the process and the progeny have been killed, the scheduler is called.

3. **Passeren :** performs a P operation on a semaphore (requested by the calling process). If the value is negative, the exception state is store in the current process and the latter is inserted in the process queue of the semaphore. In this way, the process will not be dispatched by the scheduler until it is reinserted again. Then, the scheduler is called in order to choose the next process to run.

4. **Verhogen :** performs a V operation on a semaphore, removes the process from the queue of the semaphore and insert it on the ready queue, so that it can be dispatched again when the scheduler is called.

5. **Wait for IO device :** this system call is issued from the process requesting an I/O operation, and consists in executing a P operation on the requested device semaphore.

```
1     device_semaphores[index] -= 1;  /* performing a V on the
      device semaphore */
2     currentProcess->p_s = *cached_exceptionState;
3     insertBlocked(&(device_semaphores[index]), currentProcess)
      ;
4     Scheduler();
```

6. **Get Cpu time :** this syscall should return the value in the **Current Process'** "time field" (the amount of processor time used by each process) plus the amount of CPU time used during the current quantum/time slice.

7. **Wait for Clock :** used to perform a P operation on the Pseudo-clock semaphore, this particular semaphore is a synchronization one so the current process is blocked on the interval timer semaphore (the pseudo-clock semaphore is the last index in the device-semaphores array).

8. **Get Support Data :** issuing this operation leads to the saving of the current process's **support struct** into the **exception state v0 register**.

```
1 void Get_Support_Data_SYS8()
2 {
3     cached_exceptionState->reg_v0 = currentProcess->
      p_supportStruct;
4 }
```

As easy as pie.

2.4 How to compile

Compiling is easy thanks to the makefile located in the same code files directory. So, to compile the code, you can use the ‘make‘ command. In order to delete all the files created by the compilation process, you can make use of the ‘make clean_all‘ command

2.5 Debugging notes

In order to debug the whole program, we used a file called debugger.c in which we added a number of breakpoint functions. These are very useful and can be easily be loaded in thanks to the ”Add Breakpoint” function in umps3. This way we were able to observe the slices of code which our program actually executed, and the ones that weren't properly executed. More informations about **umps** debugging can be found at this location ump-debugging