Pandos Phase 3 Documentation

Angelo Galavotti, Adriano Pace, Denis Pondini

September 2021

1 Introduction

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

This Pdf represents the documentation for the third 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 **support-layer**, which adds support for user-mode processes as well as virtual-memory and paging.
Besides the components of the two previous phases, the support-layer includes three new modules: **vmSupport.c**, **initProc.c** and **sysSupport.c** .
The challenges of this layer were mostly due to the interaction between processes, and the sharing of resources between them. Consequently, atomic access to variables, addresses and devices was often required in order for the OS to work properly. The goals of this phase also included the understanding of the workings of virtual memory, as well as the implementation of the **pager** (which is the subroutine that handles page-faults).

2 Modules and implementation

Here follow the most important decisions about the making of the support-layer, as well as the description of each of its components.

2.1 initProc.c

This module consists of some procedures which execute right after the boot up of the nucleus. The function `test()` is associated with the first process of the system, and initializes the swap structures as well as the semaphores. Lastly, eight user-process instances are created.

2.1.1 Support Layer's device semaphores

The *support layer* uses a **different set of device semaphores**. Like in Phase 2, these semaphores are used for mutual exclusion of shareable devices, therefore

they are initialized with a value of `1`. Since we are only operating with flash devices, terminal devices (which need two different semaphores each) and printer devices in the support level, the number of semaphores that are needed is much lower.

2.1.2 Swap pool and its dedicated structures

The **swap pool** is a set of RAM frames dedicated to virtual memory (and consequently, they will be occupied by each process' pages when they're needed). The total number of frames in the swap is two times the maximum number of concurrent processes (`UPROCMAX*2`).
Associated to the swap pool is also a **swap table** which defines the current status of the frames in the *swap pool* (i.e. whether they're occupied or not, and if so by which page etc.).
A **semaphore** is also needed for the swap pool, in order for each process to gain mutual access.

2.1.3 Process creation and initialization

As mentioned before, the `initProc` module also handles the creation of user-processes. To create a user process, it is necessary to initialize a **support structure** as well as a **processor state structure**.
In the *support structure*, the *ASID* (which is ID of each process) is set, in addition to information and memory space dedicated to the handling of *page faults* and the *General Exception Handler*. Moreover, the **page table** of each process is initialized.

2.1.4 The masterSemaphore

The *masterSemaphore* is a semaphore with the initial value set to `0`. After initializing and creating all user-processes, the `test()` function performs a `P()` operation on the semaphore eight times. Clearly, right after the first `P()`, control shifts to the next process in the *readyQueue*. Only after a process terminates a `V()` on the *masterSemaphore* is performed. Thus, when all processes conclude, control is completely returned to the *init process* (which is the process that invoked the `test()` function). Consequently, the test will end with a 'System halted' message.

2.2 sysSupport.c

This module includes the instructions for the *General Exception Handler*, which takes care of *Program Trap Exceptions*, as well as syscalls of code 9 to 13.

2.2.1 Program Trap Exception Handler

PandOS handles program trap exceptions received by the *"Pass Up Or Die"* procedure in the Nucleus by terminating the process.

2.2.2 SYS9 - Terminate

This function is used to terminate a process.
Before calling a SYS2, the **swap table entries** associated with the process are emptied, and a `V()` operation is performed in every semaphore in which the process is blocked.

2.2.3 SYS10 - GetTOD

Returns the 'Time of Day' in a simple fashion.

2.2.4 SYS11 - Write_To_Printer

This function is used to write to a printer device. The calling process is suspended until the printer device associated with it receives a string of characters. The string sent to the printer is located in the `a1` register, and each character of the string is written in the `data0` field of the device register. Before each *'command'* operation, interrupt are disabled in order for the subsequent `WaitForIO` syscall to catch the correct interrupt. After the `WaitForIO` syscall is completed, the interrupts are enabled again. Before executing these instruction, the character pointer is checked to evaluate if the address is within **valid memory space**.

2.2.5 SYS12 - Write_To_Terminal

This function is used to **write to a terminal device**. The calling process is suspended until the terminal device associated with it receives a string of characters.
The same instructions as a `SYS11` are executed in this syscall. However, the data to write in the command field of the device register and the retrieval of the status code are slightly different.

2.2.6 SYS13 - Read_From_Terminal

This function is used to **read the input given by a user** to a terminal device. The calling process is suspended until a line of input ending with the `EOL` (`'\n'`) character is transmitted to the terminal.
The process of reading an input line is done by using a continuous while loop in order to read the string character by character, until `EOL` is given as input. Moreover, the status returned by the device register specifies if a character has actually been transmitted, thus only in this situation the character is able to be decoded.
A buffer is also used to save each character of the string. The address of the buffer in which the string must be stored is given in the `a1` register.

2.3 vmSupport.c

The third and final module, vmSupport, contains the code for the **Pager** and the **TLB Refill Handler**.

2.3.1 The Pager

The **Pager** is the component of the *support layer* which handles *page faults*, and could also be regarded as its most important part.
To handle a ***page fault***, the requested page must first be identified: this is done by reading the exception state in the support structure of the current process (we can retrieve the structure by using a `SYS8`). Both the `ASID` and the `VPN` (*Virtual Page Number*, which is the position\index of the page in the process' page table) of the requested page can be retrieved by reading the `EntryHI` field in the support structure.
Subsequently, mutual exclusion must be gained on the swap pool, since a handful of operations have to be done onto it. If a free frame is found in the swap pool, that frame will be the frame selected in which the requested page will be put. Otherwise, a FIFO replacement algorithm will chose a frame, and the occupying page will be put back into the process' backing store, which is essentially a virtual flash device used to store the process' pages.
In order to do that, the valid (V) bit must be set to off (since the page is not located in RAM anymore) and the TLB must be updated to overwrite the changes applied to the case (in this case, setting the V bit off). After this, the page must be overwritten in the process backing store using the `backStoreManager()` function implemented in this module.

2.3.2 The Pager - writing a page into memory

Following the free frame retrieval (whether because it was free from the star or was freed), the page which caused the page fault needs to be written into RAM. To write the particular page into the newly found frame, the function `backStoreManager()` is used again but with a different purpose.
In addition, the swap table has to be updated with the information of the status of the frame (the `ASID` and the `VPN` of the new occupying page, as well as a pointer to its page table entry). The `EntryLO` of the page also has to be updated, in order to include the PFN of the located frame and to set its V and D bit on (DISCLAIMER: the `PFN`, which stands for *Physical Frame Number*, is in fact NOT the Physical Address of the frame associated with the page, but rather a part of it). Once again, the TLB has to be updated with the new information of the page.
Since the *swap pool* and *swap table* are shared resources, these operation have to be done **atomically**, with interrupts disabled.

2.3.3 backStoreManager()

This function manages the reading/writing requests made to a flash device associated with a particular `UPROC`. Depending on the operations, the `block_address` and the `data_address` specify the *block* to read and to write into respectively, or to write and read the data from respectively. Since the size of a *block* is the same as the size of a page, the each *flash device memory area* can be visualized as divided into pages. Therefore, the *block number* usually corresponds to the `VPN` of the requested page.

2.3.4 TLB Refill Handler

This functions contains the instructions dedicated to filling the *Translation Lookaside Buffer*. Once a **TLB miss** occurs, the `VPN` of the page which caused it is retrieved directly from the `currentProcess` data structure. Afterwards, the `EntryLO` and `EntryHI` of the page are loaded into the respective CPU registers and, using the the function `TBWR()`, are then loaded into the *TLB*.
Access to the `currentProcess` data structure is possible since this is considered a Nucleus subroutine.

2.3.5 updateTLB

The `updateTLB()` function updates a single entry of the *TLB*. This is done by loading the `EntryHI` field of the page into the `EntryHI` register of the CPU. Subsequently, `TLBP()` is called, which probes the *TLB* for a TLB entry that matches with the current values set into the CPU register. If a matching entry is found, the value `0` is returned in the `P` value of the `Index` register. Therefore, we can write the page information into the *TLB entry* using `TBWI()`.

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 umps-debugging