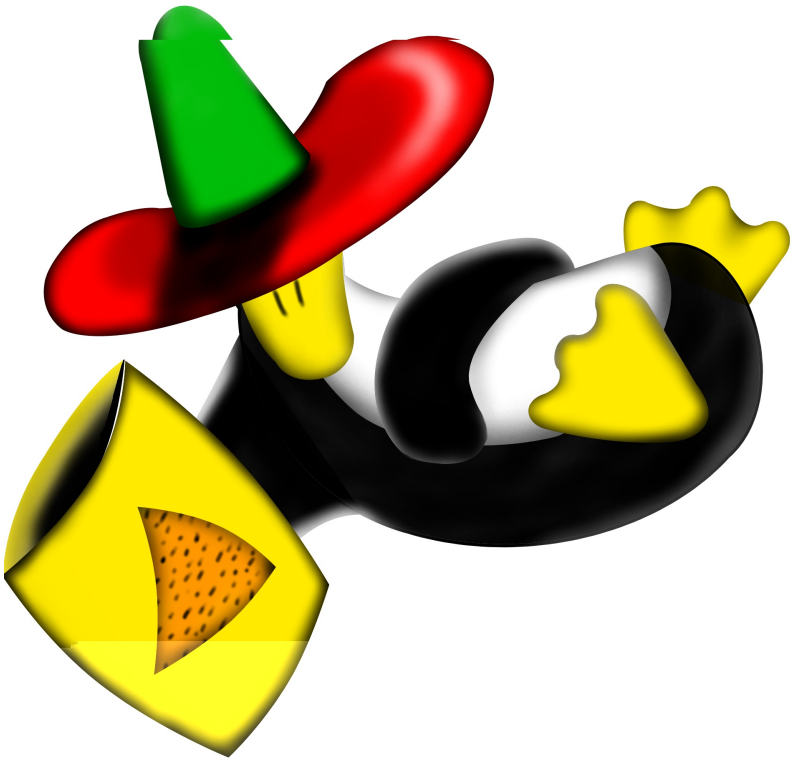
# Operating systems laboratories
# First year of master, SGP and SGM course units

Academic year 2015-2016



Isabelle Puaut
Laurent Perraudeau

# Contents

# Introduction

This document contains all the materials for the laboratories of the course units SGP (Operating Systems, Process Management) and SGM (operating systems, Memory Management), attended by first-year master students.

## Objectives of the labs

The labs focus on two complementary aspects of operating systems:

- System programming, through the use of the operating system API (systems calls and C library) of the Linux operating system.
- *Implementation* of parts of an operating system kernel. Due to the complexity of real operating systems kernels, the labs will consist in modifying an educational operating systems called NACHOS, having the good properties of being relatively simple, compact and runnable on simulated hardware, thus easing the debugging of the implemented code.

## Document organization

The document is organized as follows. The internal structure of NACHOS is presented in Chapter 1. The lab assignments are given in Chapters 2 and 3 for each of the two semesters. Finally, in the appendices are some notes on *gdb*, a debugger that can be useful in the debug phases (Appendix A) as well as a list of Frequently Asked Questions (FAQ) in case you wish to use NACHOS at home (Appendix B).

## Bugs, suggestions for improvements

If you find a bug in the source code of NACHOS or in the documentation, or have suggestions for improvements, please signal it to Isabelle Puaut (puaut@irisa.fr).

## History and acknowledgments

This software has been modified and extended since 2000 in Rennes (INSA Rennes, University of Rennes 1) to fit our local needs. The README file at the root of the source code gives a

(not necessarily exhaustive) list of the changes to the original NACHOS over the years[1].

Thanks to Erwan Abgrall for the cover picture, and to Bastien Pasdeloup for his unvaluable help when translating this document into English, as well as for suggesting the addition of question 9 in the NACHOS questionnaire.

[1]Contributors to our version of NACHOS are, by chronological order of last modifications: Matthieu Gabriac, Julien Gloaguen, Jérôme Le Dorze, Aurélien Letort, Antoine Mahé, Freddy Perraud, Anthony Remazeilles, Chloé Rispal, Ivan Leplumey, David Decotigny, Isabelle Puaut.

# Chapter 1

# The Nachos educational operating system

NACHOS (*Not Another Completely Heuristic Operating System* ) is an operating system for educational purposes originally designed at the University of Berkeley [1]. The goal of NACHOS is to allow students to understand the internals of operating systems, through the provision of mechanisms that exist in commercial operating systems.

This chapter aims to facilitate the understanding of NACHOS internals and intends to enter gradually in the details of its implementation. It is important to note that the present document describes the complete NACHOS, with all functionalities implemented. Some parts of the code were voluntarily cut; it will be your job to implement them during the labs.

## 1.1 Overview of NACHOS

In this section, we present the main components of NACHOS without going into the details of their implementation.

### 1.1.1 Nachos runs into a Unix process

The rationale behind the design of NACHOS is to allow students to learn how to implement the main operating system concepts, with achievable investment of the students. To do so, instead of executing NACHOS on bare metal, NACHOS runs on top of simulated hardware (processor, input/output devices). The combination of the simulated hardware, the NACHOS kernel and NACHOS user programs are altogether executed in a single Unix process, as depicted in Figure 1.1.

This approach has a two-fold benefit. On the one hand, the kernel development is simplified because the simulated hardware, although realistic enough, is deliberately very simple. On the other hand, debugging the NACHOS kernel code is facilitated, because an error in the code does not crash the machine, and off-the-shelf debugging tools (e.g. *gdb*, see Appendix A) can be used.

NACHOS is developed in `C++`, using only the basic features of the language, basically encapsulation. This allows to structure the code of NACHOS correctly, without requiring in-depth knowledge of the `C++` language.

[1] All information about to NACHOS can be found on NACHOS page at *http://www.cs.berkeley.edu/~tea/nachos*

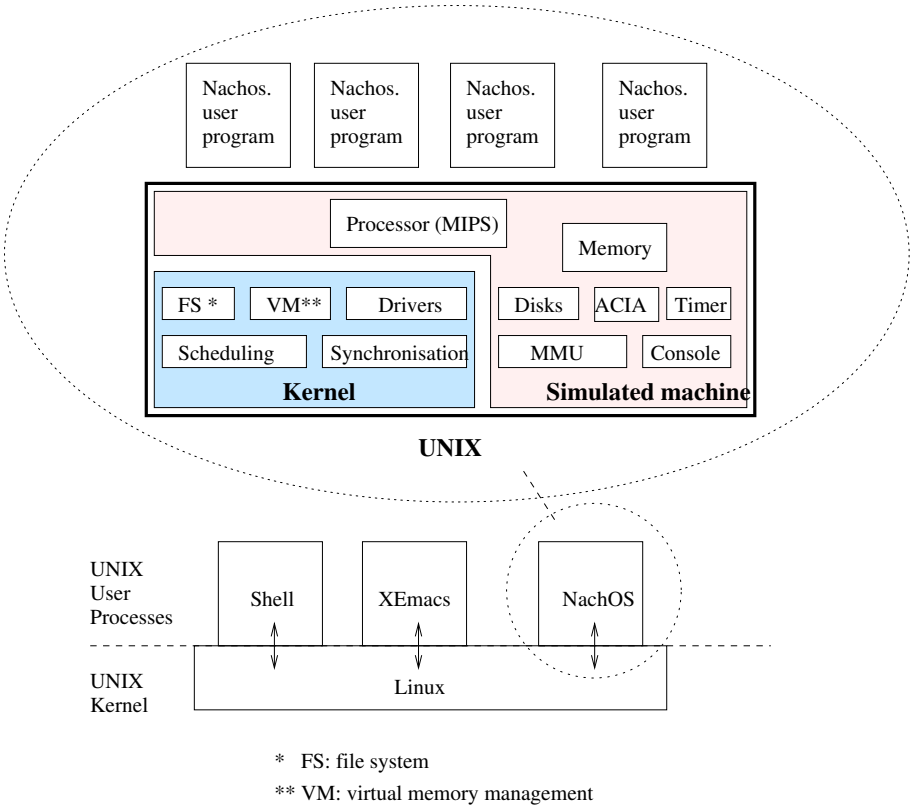



Figure 1.1: Internal Structure of NACHOS

### 1.1.2 NACHOS hardware

The main hardware components that are simulated in software are: a MIPS processor used to execute NACHOS user programs, a timer, disks, serial communication hardware, a console, and a Memory Management Unit (MMU). These elements will be detailed in Section 1.2.

### 1.1.3 Main kernel modules

NACHOS is structured in categories of functionalities:

- The *kernel* contain the vital features of NACHOS: lightweight processes (threads), synchronization between threads, scheduling, address spaces;
- The *device drivers* provide the interface to access the (simulated) hardware devices (disk, console),
- *Virtual memory* management (fully detailed later on);
- *File management*.

The files hierarchy of the source code of NACHOS partially reflects these different categories. Source code is spread in the following directories:

- *machine:* machine simulation (must not be modified);
- *kernel:* kernel code (management of threads and processes, synchronization, scheduling);
- *vm:* virtual memory management;
- *drivers:* code of device drivers;
- *filesys:* file management;
- *utility*: utility classes (lists, bitmaps, statistics, etc.).
- *userlib:* set of functions for NACHOS user programs (mini-libc);
- *test*: user programs;
- *doc*: NACHOS documentation

#### NACHOS kernel (directory *kernel*)

The kernel of NACHOS manages a set of lightweight processes (threads) that execute on the (simulated) MIPS processor. We call *process* the combination of an address space and the set of threads that share it.

The implemented scheduling policy is pure FIFO (First In, First Out) with no concept of thread priority and no time sharing. Provided synchronization tools are counting semaphores, condition variables and mutual exclusion locks.

When a thread is blocked on a synchronization primitive, a context switch occurs and the processor is allocated to the first thread in the kernel ready list (global variable *readyList*). When there is no ready thread, the kernel loops until an interrupt occurs.

The different NACHOS objects (threads, semaphores, locks, etc.) are identified by a type, coded with an integer (see file *kernel/system.h* type *ObjetTypeId*). The type is used in every system call to check at run time that it is applied to the correct object type.

**Device drivers (directory *drivers*)**

Device drivers provide to user programs an interface to the (simulated) hardware. They provide a synchronous (blocking) interface to the hardware components, that by construction provide an asynchronous interface. This is achieved by using a combination of interrupts and synchronization primitives (semaphores and locks).

**File system (directory *filesys*)**

NACHOS file system is built on top of the disk (simulated) hardware; each disk is simulated using a Unix file. The NACHOS file system provides primitives for:

- creating and deleting directories;
- creating and deleting files;
- reading and writing files.

It is possible to copy a Unix file, containing a NACHOS program or simply a data file, on the NACHOS disk when NACHOS is booted.

**Virtual memory management (directory *vm*)**

NACHOS has a complete virtual memory management system, that implements both address translation (using paging hardware), and demand paging.

Address translation consists, for every memory access, in transforming, using hardware support (MMU), the *virtual* addresses manipulated by user programs into *real, physical* addresses in the memory of the MIPS machine. The address translation mechanism uses page tables stored into the memory of the host machine. A *swap area*, stored into a dedicated NACHOS disk, is used in case the machine memory is full[2]. Conversely, pages are loaded into memory on demand; when there is no virtual to physical mapping for a page, the page fault mechanism loads the missing page into memory, or if it is not possible signals an addressing error.

### 1.1.4 Set of functions for user programs (directory *userlib*)

User programs are written in C. System calls, to communicate with the kernel, as well as utility routines, are grouped in a a few function written in C. Only the most important routines of the standard C library are provided (NACHOS is by construction a mini-system).

Since NACHOS programs do not directly run on the host hardware but on a (simulated) MIPS processor, the code of NACHOS programs is generated using an off-the-shelf MIPS cross-compiler (gcc).

### 1.1.5 System configuration (file nachos.cfg)

Some parameters of NACHOS can be changed in a configuration file, to avoid re-compiling NACHOS every time a parameter has to be changed. By default, unless option `-f` is used to provide a custom file, NACHOS searches for a configuration file named *nachos.cfg* in the

[2]The code, region which is read-only, will not be placed in the swap area.

current directory. The configurable elements are among others: the size of NACHOS memory, the name of the program to be executed when NACHOS is started, whether or not the disk has to be formatted at boot time, etc. The format of the configuration file is detailed in Section 1.9.6.

### 1.1.6 Statistics

A statistics module allows to monitor the performance of NACHOS and the impact of NACHOS configuration parameters on performance. Performance metrics are: the number of executed instructions, number of page faults, number of disk accesses, execution time. Statistics are given for each process individually. Performance counters for the threads of a process are cumulated.

## 1.2 The emulated machine (directory *machine*)

### 1.2.1 The MIPS processor (files *machine.cc, machine.h, mipssim.cc, mipssim.h*)

The processor emulated by NACHOS is a 32-bit MIPS architecture. The processor has general-purpose registers (integer and floating point) and specialized registers (stack pointer, program counter, condition code).

The *g_machine* object, of class *Machine* provides methods for starting the processor and managing its state (especially reading and writing registers):

- **void Run()**: starts the decoding and execution loop from the current contents of the program counter register;
- **int ReadIntRegister(int num)**: reads integer register number *num*;
- **void WriteIntRegister(int num, int value)**: writes *value* in integer register number *num*;
- **int ReadFPRegister(int num)**, reads floating point register number *num*;
- **void WriteFPRegister(int num, int value)**: writes *value* in floating point register number *num*;
- **char ReadCC(void)**: reads the condition code register
- **void WriteCC(char val)**: writes *val* into condition code register.

Constants are defined using the *#define* directive to refer to register numbers. Some integer registers have a special role during system calls for parameter passing (see section 1.9.1).

The *g_machine* object also contains the following member variables:

- *mainMemory:* main memory of the simulated machine, allocated as an array of bytes in the host machine;
- pointers to the objects that simulate the machine components: MMU (Memory Management Unit), ACIA (Asynchronous Communication Interface Adapter), interrupt controller, disk and console.

### 1.2.2 The interrupt controller (file *interrupt.cc, interrupt.h*)

The interrupt controller determines whether or not interrupts have to be taken into account. Global variable *interrupt*, of class *Interrupt* simulates the interrupt controller, and provides two methods:

- **IntStatus setStatus(IntStatus level)**: sets the current interrupt mask to *level* and returns the previous interrupt mask;
- **IntStatus getStatus()**: returns the current interrupt mask.

Type *IntStatus* is an enumerated type for the interrupt mask:

- *INTERRUPTS_OFF:* interrupts are disabled;
- *INTERRUPTS_ON:* interrupts are enabled.

### 1.2.3 The MMU (Memory Management Unit, file *mmu.cc mmu.h*)

All addresses referenced by user programs (to code and data) are *virtual*. The role of the memory management unit is to perform address translation; for every reference, the MMU transforms the virtual address into a physical address (address in the machine memory, variable *g_machine->mainMemory*, see section 1.2.1). Virtual to physical address mappings are stored into a translation table (variable **translationTable** of the MMU).

Class MMU provides the following methods:

- **bool ReadMem(int vaddr, int size, int *value)**: returns in *value* the *size* (1, 2 or 4) bytes located at virtual address *vaddr;*
- **bool WriteMem(int vaddr, int size, int * value)**: writes *value*, of *size* (1, 2 or 4) bytes at virtual address *vaddr*.

These two methods return *false* when there is no virtual to physical mapping for virtual address *vaddr*. They use private method **ExceptionType Translate(int virtAddr, int *physAddr, int size, bool writing)** that is responsible for address translation, by reading the address translation table. Two situations may occur during address translation:

- There is no virtual to physical mapping (validity bit in the page table is not set). The MMU then triggers a *page fault exception*. The kernel page fault handling routine will load the page in memory;
- There is a virtual to physical mapping, the physical address is then returned and the memory access proceeds immediately.

Understanding precisely how the MMU operates is not mandatory for the first assignments (thread management, I/O). More details on virtual memory management can be found in section 1.7, page 26.

### 1.2.4 The serial communication hardware (file *ACIA.h, ACIA.cc*)

The serial communication hardware included in NACHOS is simulated using UDP messages. The serial communication hardware is available through object *acia*, instance of class *ACIA* (ACIA stands for *Asynchronous Communication Interface Adapter*).

**The ACIA data registers**

Registers *outputRegister* and *inputRegister* are used respectively to send and receive one byte of data. They are accessed by calling the following methods:

- **void PutChar(char)** that puts a character in register *outputRegister*;
- **char GetChar()** that returns the character in register *inputRegister* if the register is full, or returns 0 otherwise.

The serial communication hardware emits an interrupt signal when register *inputRegister becomes* full, and when register *outputRegister becomes* empty (interrupts upon state transitions).

**The ACIA status registers**

Registers *ouputStateRegister* and *inputStateRegister* indicate the status of the serial communication hardware. They are read-only, and accessible through the following methods:

- **RegStatus GetOutputStateReg()**: returns the status of register *outputStateRegister* (`EMPTY` or FULL);
- **RegStatus GetInputStateReg()**: returns the status of *inputStateRegister* (`EMPTY` or FULL).

**The ACIA control registers**

Control register *mode* contains the current mode of operation of the serial communication hardware, that can be configured to emit interrupts of not. Method **void SetWorkingMode(int mod)** of class ACIA changes the hardware working mode, whereas method **int GetWorkingMode()** returns the current working mode. Register *mode* can take three values:

- *BUSY_WAITING:* no interrupt emitted; the device driver when willing to send or receive data has to repetitively read the ACIA's status registers;
- *SEND_INTERRUPT:* an interrupt is posted after sending every byte;
- *REC_INTERRUPT:* an interrupt is posted upon receipt of every byte.

Values *SEND_INTERRUPT* and *REC_INTERRUPT* are integer constants with only one bit set. A bitwise or (operator "`|`" in `C`) has to be used to enable both sending and receiving interrupts.

### 1.2.5 The Disks (files *disk.cc* , *disk.h*)

The machine uses two disk devices, one containing the file system and the other containing the swap area used by the demand paging system.

Each (emulated) disk contains *NUM_TRACKS* tracks, each being composed of *SECTORS-PER_TRACKS* sectors. Each sector has a fixed size that can be configured in the NACHOS configuration file (see section 1.9.6). The disk hardware is accessible through two methods:

- **void ReadRequest(int sectorNumber, char *data)**: reads sector *sectorNumber* into memory at address *data*;
- **void WriteRequest(int sectorNumber, char *data)**: writes data located at address *data* into sector *sectorNumber*.

Both methods are *asynchronous*: they return immediately without waiting for the end of the physical input/output operation. An end of I/O interrupt is emitted.

### 1.2.6 The console (file *console.cc, console.h*)

The emulated machine has a console that can display characters on the screen and receive characters from the keyboard. The console is managed by its dedicated device driver (see 1.3.2, page 17). Methods **void PutChar(char ch)** (resp. **char GetChar()**) allow to write (resp. read) a character. Similarly to the disk, end of I/Os are signaled by posting an interrupt.

## 1.3 The device drivers (directory *drivers*)

The device drivers provide the only means to access the peripheral devices (serial communication hardware, console, disk). The device drivers manage the synchronization with the devices and control their sharing.

### 1.3.1 The driver for the serial communication hardware (files *drvACIA.cc, drvACIA.h*)

This driver manages message transmission via the serial communication hardware. The driver appears as an object *g_acia_driver* of class *DriverACIA* created at boot time. Members of class *DriverACIA* are:

- *send_buffer* and *receive_buffer* that store the characters to be sent, and the characters received and not yet consumed;
- *send_sema* and *receive_sema*, used for synchronization;
- *ind_send* and *ind_rec*, that are the indices for the next character to be sent and received in the send/receive buffers.

The device driver exports two methods:

- **int TtySend(char *buff)**: sends a null-terminated string on the serial link. When configured in *interrupt* mode, this primitive is *non-blocking*; it fills-in the send buffer and initiates the transfer. When configured in *busy waiting* mode, the routine returns when all characters have been sent. In both cases, the method returns the number of characters sent.
- **int TtyReceive(char *buff, int lg)**: receives a null-terminated string and copies it to buffer *buff*. *lg* is the maximum allowed size of the string (memory allocated for *buff*). The method returns the number of characters actually copied into *buff*.

Two interrupt service routines (ISRs) are used by the device driver:

- **void interruptSend()**: interrupt service routine responsible for sending a character. It is called when send interrupts are allowed, when the output register becomes empty;
- **void interruptReceive()**: interrupt service routine responsible for receiving a character. It is invoked when received interrupts are allowed, upon arrival of a character.

The device driver can be configured in the NACHOS configuration file (see section 1.9.6) to operate in *busy waiting* or *interrupt* mode. The selected working mode is available in the driver code via *g_cfg->ACIA*, which can take values *ACIA_BUSY_WAITING* or *ACIA_INTERRUPT*.

### 1.3.2 The console driver (files *drvConsole.cc, drvConsole.h*)

This device driver provides an interface to the console. It is represented by the object *g_console_driver* from class *DriverConsole*. Two methods are exported:

- **void PutString(char *buffer, int size)**: displays string *buffer* on the console;
- **void GetString(char *buffer, int size)**: reads a string from the keyboard into array *buffer*.

The console is by nature asynchronous. The console driver export *synchronous* methods. The console driver is implemented using interrupts, end ensures mutual exclusion on the console.

The driver methods are invoked by the system calls **Write(string,length,ConsoleOutput)** and **Read (string,length,ConsoleInput)**. System calls *Read* and *Write* are called by the NACHOS mini libc (see Section 1.9.1), for instance in routine *n_printf*.

### 1.3.3 The disk driver (files *drvDisk.cc, drvDisk.h*)

The disk device driver controls the disk device. The device driver has three members: a pointer to the disk where the access must be performed; a semaphore used to wait for the end of I/O operations, and a lock to ensure mutual exclusion on the disk device. The disk driver exports the following two synchronous methods:

- **void ReadSector(sectorNumber int, char * data)**: reads sector *sectorNumber* into variable *data*;
- **void WriteSector(sectorNumber int, char * data)**: writes data *data* into sector *sectorNumber*;

The disk driver is implemented using interrupts (interrupt handling routine *DiskRequestDone* for the main disk, and *DiskSwapRequestDone* for the swap disk).

## 1.4 The kernel of Nachos (directory *kernel*)

### 1.4.1 Kernel internals

NACHOS is an operating system with heavyweight processes (with separate address spaces). Each process is itself parallel; several threads execute concurrently and share the process address space. Four main NACHOS classes are the basis for the kernel operations:

- **Process.** The process is the entity combining all the resources to execute a multi-threaded application. The process class contains members of the following classes:
  - **Thread**, that define lightweight processes;
  - **Addrspace**, that define the memory resources shared by the threads
- **Scheduler** that manages thread scheduling.

#### Class Process (files *process.cc, process.h*)

The **Process** class contains members for all the resources used by an application:

- **exec_file**: the executable file that contains the application code;
- **addrspace**: address space, shared by all threads in the process (see Section 1.4.1);
- **stat**: statistics of the process resource consumption (CPU time, number of memory accesses, ...).

Since every thread object contains a pointer to the belonging process (see below), a process implicitly contains a set of threads.

#### Class Thread (files *thread.cc, thread.h*)

A thread is a stream of instructions executed by the processor. The main fields of objects of class *Thread* represent the *context* of the thread, *i.e.* the state of the processor. Due to the use in NACHOS of an emulated processor (instead of a system directly running on the host processor), the context of a thread is not limited to the context of the emulated MIPS machine. Instead, it is split into two components:

- *thread_context*, consisting of the contents of MIPS registers: *thread_context.int_registers* and *thread_context. oat_registers*
- *simulator_context*, consisting of the state of the MIPS simulator, represented by variables *kernel_context.buf* and the stack pointer *simulator_context.stackPointer*.

The notion of context would not exist if NACHOS was directly running on a real (non emulated) MIPS processor. It only exists because the MIPS processor is simulated by software, in order to save the state of the processor simulator in addition to the state of the simulated processor.

The main methods of class *Thread* are:

- **int Start(Process *owner, VoidFunctionPtr func, int arg)**: starts a new thread within process *owner*. This method allocates memory and initializes the thread context, and puts the thread into the ready list. Parameter *func* is a pointer on the function to be executed by the thread (address of the first instruction in the thread's code);
- **Process *getProcessOwner()**: returns a pointer to the process associated with the thread;
- **int Join(int Idthread)**: blocks the calling thread until the termination of thread *Idthread*. Returns -1 if *Idthread* is invalid;
- **void Yield(void)**: puts the calling thread at the end of the ready list. This method is used to release voluntarily the processor in favor of the other ready threads;
- **void Sleep(void)**: puts the calling thread in sleep;
- **void Finish(void)**: marks the calling thread as finished, such that its resources are freed later on;
- **void SaveProcessorState(void)**: saves the state of the MIPS registers into the thread context;
- **void RestoreProcessorState(void)**: restores the state of the MIPS registers;
- **void InitSimulatorContext(int8_t *stackAddr, int stacksize)**, which initializes the thread's simulator context. This method is private and essentially corresponds to the initialization of the kernel stack;
- **void InitThreadContext(int32_t initialPCREG, int32_t initialSP, int32_t arg)**: initializes the user context of the thread: stack allocation, initialization of the MIPS registers (stack pointer, instruction pointer) from the parameters. This method is private.

### Class Addrspace (files *addrspace.cc, addrspace.h*)

The *Addrspace* class defines the resources (in particular memory) accessible to the threads of a given process. The class constructor is in charge of reading the executable file and setting up the memory (loading code, data, etc.). The executable file format is the standard format ELF (Executable and Linkable Format). Program loading when demand paging is used will be studied during the second semester (course unit SGM) and put into practice during the labs.

### Class Scheduler files (*scheduler.cc, scheduler.h*)

The *Scheduler* class is in charge of implementing thread scheduling. There is a single instance of this class, created at boot time. The scheduler manages a list of threads that are ready to execute (variable *readyList*), and implements context switching between threads. The active thread, by construction, is not part of the ready list, that only contains ready but not active threads. A pointer to the active thread is stored in variable *g_current_thread*.

The *Scheduler* class exports three methods:

- **void ReadyToRun(Thread *thread)**: inserts a thread at the end of the ready list (this function assumes that interrupts are masked);
- **Thread *FindNextToRun(void)**: returns a pointer to the first thread of the ready list and removes it from the list;
- **void SwitchTo(Thread *nextThread)**: allocates the processor to thread *nextThread* (usually the result of a call to **FindNextToRun**).

Context switching between two threads (from the same or different processes) is performed in method *SwitchTo* of class *Scheduler*. The purpose of this method is to save the context of the active thread and setup the one for the next thread to execute.

In an operating system directly running on the host machine, the applications *and* the operating system both execute on the same hardware. For such systems, the context is only made of the registers of the host machine. In the context of NACHOS, user programs execute on an emulated MIPS processor, whereas the NACHOS kernel directly executes on the host machine. Consequently, two contexts have to be managed during a context switch: the user context (MIPS) and the kernel context. The saving/restoration of the kernel context is directly provided to you and will not have to be developed during the labs (see functions *getcontext/setcontext* called in method *SwitchTo*).

### 1.4.2 Synchronization tools (files *synch.cc, synch.h*)

Three types of synchronization are defined in NACHOS: semaphores, locks and condition variables. All synchronization primitives need to be atomic. Since NACHOS targets uniprocessor systems only, atomicity will be implemented by disabling/enabling interrupts.

#### Semaphore

Semaphores in NACHOS are classical counting semaphores. A semaphore is represented by a counter and a wait queue. The implementation of semaphores will be identical to the one described during the lectures.

- **void P()**: decrements the semaphore's counter, and blocks the calling thread when the counter gets strictly negative;
- **void V()**: increments the semaphore's counter, and releases a waiting thread (if any).

#### Locks

Locks are mutual exclusion semaphores. The two operations provided by the *Lock* class allow respectively to enter a critical section and leave a critical section. Field *owner* points to the thread currently executing in critical section, or is NULL if the lock is free. Class *Lock* exports two methods:

- **void Acquire()**: the calling thread becomes the lock's owner if the lock is free, otherwise it is placed in the lock's wait queue and is blocked;
- **void Release()**: is called by the locks's owner only, to release the lock. In case a thread is blocked waiting for the lock, it is unblocked (moved into the ready list). Otherwise it sets the lock state to *free*.

**Condition variables**

Condition variables in NACHOS provide a very simple way to block/unblock threads. Primitive *Wait* is called when a thread has to wait for a specific event; it always blocks the calling thread. A call to *Signal* signals an event, and unblocks the waiting thread if any; if no thread is waiting, the call has no effect. A condition variable has its wait list. Three methods are exported:

- **void Wait()**: the calling thread is blocked and inserted at the end of the wait list;
- **void Signal()**: the first thread of the wait list is unblocked (removed from the wait list and added to the ready list); the method has no effect when the wait list is empty;
- **void Broadcast()**: same as *Signal* except that *all* blocked threads are unblocked.

## 1.5 System calls

User programs have access to a list of system calls. MIPS instruction *syscall*, when executed, generates an exception. Exceptions are handled by the NACHOS kernel (the exception handling routine is in file *exception.cc* , in directory *kernel*). The type of system call is identified according to the contents of the MIPS registers when instruction *syscall* is executed.

The *type* of system call is identified by the contents of register *r2*. By convention, the system call's parameters are stored, into registers *r4*, *r5*, *r6* and *r7* depending on the number of parameters. The return value of the call, if any, is returned via register *r2*.

The list of system calls supported by the NACHOS kernel is given in file *syscall.h* of directory *userlib*. The code of the library, linked with the user code and implementing system calls is in file *sys.s* in directory *userlib*.

We illustrate in this paragraph how a system call operates, using a very simple example: a program that creates a new process and waits for its termination. The C code of the example is given below:

```
#include <userlib/syscall.h>
int main() {
    ThreadId NewProc = Exec("/hello");
    Join(newProc);
}
```

When compiled to MIPS assembly code, the user program (restricted to the call to *Join*) is the following:

```
    lw $4,16($fp)
    jal Join
```

The first line stores local variable *NewProc* in register *r4* (compiler parameter passing convention). The second line calls function *Join*, implemented in file *sys.s*. The code of function *Join* is the following:

```
Join:
     addiu $2,$0,SC_Join
     syscall
```

It first sets register *r2* to constant *SC_Join* to identify the type of system call, and then executes instruction *syscall* to generate an exception.

The MIPS processor (in NACHOS emulated by software), when executing instruction *syscall*, calls the exception handling routine (located in file *kernel/exception.cc*). The contents of register *r2* (in our example *SC_Join*) allows to identify the type of system call and execute the corresponding code. The code in our example is the following:

```
// Variable that will contain the thread to wait for
Thread *idThread;

// Reads the system call parameter (register 4)
idThread = (Thread *)g_machine->ReadIntRegister(4);

// Calls the Nachos method implementing operation Join
currentThread->Join(idThread);

// Returns the result (here 0, a success) via register 2
g_machine->WriteIntRegister(2,0);
```

## 1.6 File system (directory *filesys*)

The file system provides functions to organize files into a Unix-like file hierarchy. Each file contains a header (*File Header*), stored into the disk, that describes the file organization on the disk.

The NACHOS disk dedicated to the file system contains: (i) a map of free sectors (member of class *Bitmap*), (ii) the file system root directory. These two pieces of data are stored respectively into sectors 0 and 1. The disk representation of directories is the same as the representation of files; a directory contains the names of the files/directories it contains and the disk addresses of their *le headers*. When a directory is modified, the modification is committed to disk immediately.

The file system manages concurrent accesses to files, organizes the file system hierarchy and controls disk allocation for files (creation of large files, extension of file size). Figure 1.2 page 25 shows an overview of the data structures that the file system maintains in memory.

### 1.6.1 Class *FileHeader* (files *filehdr.cc, filehdr.h*)

The file system objects (directories and files) are represented in memory using a file header, or class *FileHeader*. A file header contains the following information: the size of the object, the location of its sectors on the disk. The file header is allocated into memory when the file/directory is opened. Class *FileHeader* exports the following methods:

- **bool Allocate(BitMap *bitmap, int fileSize)**: creates the header structure for a new file/directory, and allocates sectors on disk for its contents. Parameter *bitmap* is the map of free sectors, previously read from sector 0 of the disk (see class *Bitmap*, Section 1.8.3)
- **void Deallocate(BitMap *bitmap)**: deallocates the data sectors of the object on the disk
- **void FetchFrom(int sectorNumber)**: fetches the file header from disk;

- **void WriteBack(int sectorNumber)**: writes the file header back to the disk;
- **int ByteToSector(int offset)**: returns the sector number corresponding to a given offset in a file;
- **int FileLength()**: returns the object's size in bytes;
- **void Print()**: displays the contents of the file (for debugging purpose only);
- **bool isDir()**: returns true if the object is a directory, false otherwise;
- **void SetFile()**: sets the object's type as *le*;
- **void SetDir()**: sets the object's type as *directory*.

### 1.6.2 Class *FileSystem* (files *filesys.cc, filesys.h*)

This class provides methods to initialize the file system and to create, open or delete files and directories:

- **FileSystem(bool format)**: initializes the file system. If *format* is true, the NACHOS disk is formatted: it contains a single empty directory and the bitmap of free sectors is set accordingly.
- **int Create(char *name, int initialSize)**: creates a new file. This method allocates space on disk for the file header and data, adds the file in the appropriate directory, and writes back modifications to disk. Parameter *name* is the *absolute* name of the file (eg */foo/bar*), there is no concept of working directory in NACHOS.
- **OpenFile *Open(char *name)**: opens a file for reading and writing and loads its file header in memory.
- **int Remove(char *name)**: deletes a file.
- **void List()**: displays the contents of the file system (bitmap, root directory, file headers, actual file contents).
- **int Mkdir(char *name)**: creates a new directory. The method allocates sectors for the directory header and contents and writes back changes to disk.
- **int Rmdir(char *)**: deletes a directory, after having checked that it exists and is empty.
- **bool decompname(char *name, char *head, char *tail)**: the function is a private function used to manage the directories hierarchy. *decompname* takes as argument the absolute name of a file and decomposes it into two parts: the name of the directory where the file is stored (returned in *head*) and the reminder of the name (returned in *tail*). For instance, if *name = \/dir1/dir2/foo"* the function returns *dir1* and *dir2/ c*. Returns *true* if the structure of the name is valid, *false* otherwise.

- **int FindDir(char * name)**: takes as parameter the absolute name of a file and returns the number of the sector that contains the header of the directory containing the file. For example for *name = /dir1/dir2/foo*, the function returns the sector number of *dir2* and `name` becomes *foo*. This method calls method *decompname* to decompose the file name. **Caution:** the contents of the string pointed to by *name* is changed after the call!

- **OpenFile *GetFreeMapFile()**: returns an *OpenFile* that corresponds to the bitmap of free sectors from disk.

- **OpenFile *GetDirFile()**: returns a pointer to an *OpenFile* object for the root directory.

### 1.6.3 Class *Directory* (files *directory.cc, directory.h*)

Class *Directory* mainly maintains an array for each file/directory contained in the directory. For each element, the table associates the element's name with the sector number of its file header. Exported methods allow to manipulate directories in memory and read/write them to disk:

- **void FetchFrom(OpenFile *d)**: reads the contents of directory *d* from disk;

- **void WriteBack(OpenFile *d)**: writes the contents of directory *d* to disk;

- **int Find(char *name)**: returns the sector number of the file header of object *name*, or -1 if *name* is not found;

- **int Add(char * name, int newSector)**, adds a new pair (name, sector for file header) to the directory table;

- **int Remove(char *name)**: deletes a file/directory from the directory table;

- **void List()**: lists the directory contents;

- **void Print()**, list the directory contents, the number of the sectors containing the file headers and the file contents;

- **bool Empty()**: returns true iff the directory is empty.

### 1.6.4 Class *OpenFile* (files *openfile.cc, openfile.h*)

An instance of class *OpenFile* is created upon file/directory opening. A object of class *OpenFile* has as members: the file name, a pointer to the file header, and the current position in the file, to be used for read/write operations. The class exports the following methods:

- **OpenFile(int sector)**: opens a new file at the creation of the object; argument *sector* is the sector number of the file header on disk.

- **~OpenFile()**: closes a file at the destruction of the object;

- **void Seek(int position)**: changes the current position in the file.

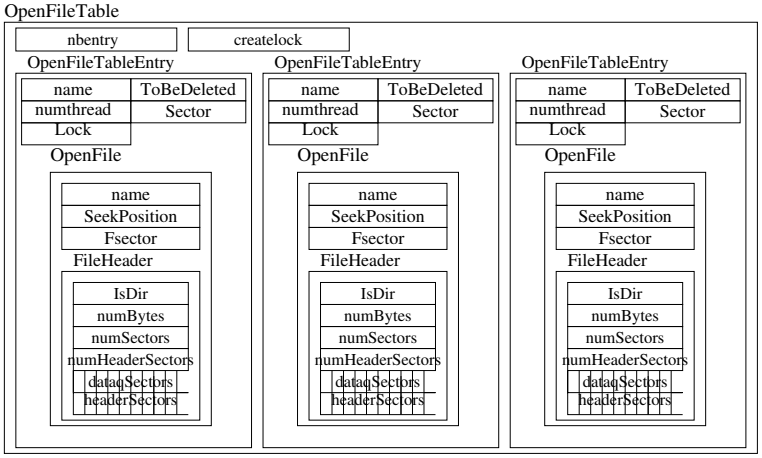



Figure 1.2: Open file table

- **int Read(char *into, int numbytes)**: reads *numbytes* bytes from the current position and places them at address *into*. There is no disk cache implemented (every call to *Read* results in a phsyical I/O). Returns the number of bytes actually read.

- **int Write(char *from, int numbytes)**: writes *numbytes* bytes, located at address *from* to disk at the current position. Writes are synchronous (the function returns when all bytes have been written). Returns the number of bytes actually written. Functions *Read* and *Write* update the current position in the file according to the number of bytes read/written.

- **int ReadAt(char *into, int numbytes, int position)**: the same as *Read* except that it reads at a specific position in the file, specified by parameter *position* instead of reading at the current position. The current position in the file is not modified by *ReadAt*.

- **int WriteAt(char * from, int numbytes, int position)**: the same as *Writes* except that it writes at a specific position in the file, specified by parameter *position* instead of writing at the current position. The current position in the file is not modified by *WriteAt*.

- **int Length()**: returns the file size in bytes.

- **Fileheader *GetFileHeader**() returns a pointer to the file header.

- **char *GetName()**: returns the file name.

- **void SetName(char *name)**: modifies the file name.

- **bool isDir()**: returns true if the opened object is a directory and false otherwise.

### 1.6.5 Classes *OpenFileTable* and *OpenFileTableEntry* (files *oftable.cc, oftable.h*)

Class *OpenFileTable* maintains a table of opened files and synchronizes the accesses to files (e.g. concurrent attempts to open the file). Each entry in the table (of class *OpenFileTableEntry*) contains the filename, a pointer to the *OpenFile* object, the number of threads that have opened this file, and a lock for synchronization, the sector number of the file header, and a boolean *ToBeDeleted* indicating that the file will have to be destroyed when all the threads will have closed the file. Class *OpenFileTable* exports the following methods:

- **OpenFile *Open(char *name)**: creates a new *OpenFile* and puts it into the table if the file is not already opened, then returns a pointer to the *OpenFile*. Parameter *name* is the absolute name of the file.
- **void Close(char *name)**: closes the file. It decreases the number of threads that have opened the file, and deletes the *OpenFile* from the table when the counter reaches 0.
- **void FileLock(char *name)**: asks for a (mutual exclusion) lock on the file.
- **void FileRelease(char *name)**: releases the lock on the file;
- **int Remove(char *name)**: deletes a file. The method sets *ToBeDeleted* to true; from that time, no new opening of the file is permitted, but the actual deletion is delayed until all threads have closed the file.

## 1.7 Virtual memory management (directories *vm, machine*)

As on most modern processors, the addresses of code and data on MIPS processors are *virtual*. Address translation hardware is used, for every memory access, should it concern code or data, to transform the virtual address into a *physical* address (address in the machine's RAM, *i.e.* variable *g_machine->mainMemory*). Each thread belongs to a unique address space, that defines the ranges of valid virtual addresses, depending on the size of code, data, stack, etc.

The *virtual memory management* code of NACHOS is in charge of maintaining the data structures (*page tables*) traversed by the address translation hardware. The code implements demand paging and page replacement, and has all benefits provided by virtual memory: processes are protected against each other; program loading and memory allocation is very simple, since the (virtual) addresses where a program is loaded are always the same regardless of already loaded programs. Programs, taken in isolation or altogether, may manipulate more memory than the amount of RAM available.

### 1.7.1 Address translation

The address translation mechanism, for every reference, transforms a virtual address into a physical address in the main memory of the MIPS processor. This is performed using hardware support, a *Memory Management Unit* (MMU), and *page tables* stored in RAM.

**The memory management unit: class *MMU* (directory *machine*, files *mmu.cc, mmu.h*)**

The MMU hardware, like all hardware elements in NACHOS, is emulated in software, through a class, named *MMU*, that provides two methods:

**ReadMem(int addr, int size, int *value)**

- *addr:* virtual address to be read;
- *size:* number of bytes to read (1, 2 or 4);
- *value:* value read from memory.

**WriteMem(int addr, int size, int *value)**

- *addr:* virtual address to be written;
- *size:* number of bytes to write (1, 2 or 4);
- *value:* value to be writtento memory.

These two methods call the private method *Translate* to transform a virtual addresses into a physical address:

**ExceptionType Translate(int virtAddr, int *physAddr, int size, bool writing)**

- *virtAdd*: virtual address to be transformed;
- *physAddr*: corresponding physical address (if any);
- *size*: number of bytes to access (1, 2, or 4);
- *writing*: type of access (read/write).

Method *Translate* first computes the *virtual page* corresponding to the virtual address as well as the *o set* in this page. Function *Translate* then scans the address space's page table stored in a register of the MMU (field *translationTable, see section 1.2.3*). There can be three different outcomes for address translation:

- There is a valid translation in the page table. The physical address is then obtained using the formula: *Physical address = Physical page number * Page size + O set* and the memory access proceeds immediately.
- The physical page is not currently in memory: it may have been swapped out to disk, may still be in the binary file (page from the text section, initial version of a data page) or may be an *anonymous* page (*i.e.* page from the uninitialized data section or stack page). In this case the MMU raises a *page fault* exception. The *page fault handler* is the software in charge of managing the page fault exception (loading the page into memory and modifying the page table accordingly). NACHOS page fault handler is in files *pageFaultManager.cc* and *pageFaultManager.h* (see Section 1.7.2 for a description).
- The referenced virtual address is illicit (outside the normal address range of the process). In this case, an *AddressErrorException* exception is raised.

The result of method *Translate* is of type *ExceptionType* and gives the outcome of address translation:

- *BUSERROR_EXCEPTION*: alignment problem;
- *ADDRESSERROR_EXCEPTION*: virtual address is outside the process's address space;
- *READONLY_EXCEPTION*: attempt to modify a read-only page;
- *PAGEFAULT_EXCEPTION*: no translation available (page fault);
- *NO_EXCEPTION:* address translation proceeded normally.

**Translation table: classes *TranslationTable*, *PageTableEntry* (directory *machine*, files *translationtable.cc, translationtable.h*)**

There is a page table per process, represented by class *TranslationTable*. The page table of a process is filled-in when the process is started (constructor of class *Addrspace*) according to the memory map of the process described in its executable file. The actual contents of a page table evolves over time due to *page faults* and *page replacements* (see 1.7.2).

| physicalPage | addrDisk | valid | U | M | swap | io | readAllowed | writeAllowed |
|---|---|---|---|---|---|---|---|---|

Figure 1.3: Structure of an entry in a page table

The structure of an entry in the page table (class *PageTableEntry*) is shown in Figure 1.3. Page tables in NACHOS are linear, the virtual page number is used to index the page table. The fields in a page table entry have the following classical meanings:

- bit *valid* determines the validity of the entry (*i.e.* valid is set to *1* when there is a virtual to physical mapping for the page);
- *physicalPage* is the physical page number; this field is disregarded by the MMU when field *valid* is 0;
- bit *readAllowed* (resp. *writeAllowed*) is set when it is allowed to read (resp. write) on the virtual page;
- bit *U* and *M* are the classical reference and modification (dirty) bits. These bits are set by the MMU hardware and cleared by software;
- *addrDisk* is the disk address of the page (in the binary file or swap area); in case it is an anonymous page, the value of *addrDisk* is *-1*;
- bit *swap* has a value of *1* when the disk image of the page is in the swap area;
- bit *io* is set to *1* when there is an ongoing I/O operation on the page; this bit is required to manage almost simultaneous page faults on the same page by concurrent threads.


Figure 1.4: Structure of an entry in the table of actual pages

There is one method per field in a page table entry to read and modify that field. The prefix of the method name (*get*, *set*, *clear*) identifies the type of operation to be performed on the field. For instance, the *swap* bit is read by calling method **bool getBitSwap(int virtualPage)**, whereas method **void setAddrDisk (int virtualPage, int addrDisk)** is used to modify the disk address of a virtual page.

**Physical page table: class *PhysicalMemManager* (directory *vm*, files *physMem.cc, physMem.h*)**

The NACHOS operating system maintains the status of all physical pages in a global data structure, named *physical page table*. An entry in the physical page table (see Figure 1.4) has the following structure:

- *free* indicates whether the page is free or not.
- *virtualPage* is the virtual page number corresponding to this page, if any;
- *owner* is a pointer to the address space of the owner of the physical page, if any;
- *locked* indicates that the page should not be requisitioned by the page replacement algorithm, because an I/O on that page is in progress (ongoing page fault on that page, ongoing copy back on disk);

Method **int AddPhysicalToVirtualMapping (AddrSpace *owner, long virtualPage)** of class *PhysicalMemManager* is in charge of finding a physical page to associate to a given virtual page, and modifying the physical page table accordingly. The method scans the physical page table in search for a free page (call to **int FindFreePage**) and requisitions one page in case all physical pages are used (call to **int EvictPage**). Method **void RemovePhysicalToVirtualMapping(int physicalPage)** modifies the physical page table to reflect that a virtual to physical mapping was removed.

### 1.7.2 Demand paging and page replacement

**Page fault routine: class PageFaultManager (directory *vm*, files *pageFaultManager.cc, pageFaultManager.h*)**

The page fault handling routine is called when a page fault exception is raised by the MMU. The only method of class *PageFaultManager* is **Exception PageFault(int virtualpage)** where *virtualpage* is the virtual page number. Method *PageFault* uses the *swap* bit and the *addrDisk* field to determine the location of the page on the disk:

- When *swap* = 1, the page is in the swap area (has been swapped out previously). *addrDisk* then contains the disk address of the page in the swap area (sector number). A value of of -1 for *addrDisk* means that the page is currently under copyback on the

swap area; the page fault handler must then wait for the termination of the I/O before reading the page.

- When *swap = 0* and *addrDisk* = -1, the page has no version on the disk at all, neither in the binary file, nor in the swap area. This case corresponds to first accesses to so-called *anonymous* pages: pages that have no initial version in the binary file (page in the *bss* section of the ELF file, see section 1.9.3, or stack page).

- When *swap = 0* and *addrDisk* $\neq$ -1, the page is on the disk, but not in the swap area. This case occurs when the page is in the binary file (code section or data section). *addrDisk* then contains the position of the page in the binary file (offset, in bytes).

In the first case, the swap manager (see below) is invoked to read the page from the swap area, and the physical memory manager is called to find a physical page. The last two cases differ from the first case in the way the physical page is filled:

- in case of an anonymous page, the page is simply filled-in with zeroes;

- in case of a page containing code or data, the page is filled from the contents of the executable file.

The swap manager, responsible for the storage in the swap area, is implemented in class *SwapManager* (files *swapManager.cc, swapManager.h*). The swap manager exports two methods: **void GetPageSwap (int numSector, char *SwapPage)** and **int PutPageSwap (int numSector, char *SwapPage)** to respectively read and write a page in the swap area. When *PutPageSwap* is called with *numSector=-1*, the method allocates a new sector in the swap area, whose number is returned. Operations on the executable file will use the file system interface (methods *ReadAt* and *WriteAt*).

In all cases, the page table is updated by the method *PageFault* which returns an exception describing the outcome of the page fault (value *NO_EXCEPTION* to indicate that everything went well).

### Page replacement algorithm: class PhysicalMemManager (directory *vm*, file *physMem.cc, physMem.h* )

The page replacement algorithm is implemented in the method **int EvictPage()** of class *PhysicalMemManager*. *EvictPage* is called when a page fault occurs and there is no free page in the physical memory. The page replacement algorithm selects one physical page to requisition and actually requisitions it (modifies the page table of its owner and copies that page to disk if needed).

The selection of the page to be evicted is performed using a circular scan of the physical page table, according to the so-called *clock algorithm* studied during the lectures. The page to be evicted is selected according to the following criteria: it must not have been used recently (*U = 0*), and should not be locked (*locked = 0*). During its walk of the physical page table, the page replacement algorithms clears the bit *U* of every examined page. When the victim page is modified (*M = 1*), the page is copied back to disk before its requisition.

Although the algorithm to select the page to be evicted is simple, some tricky issues caused by multi-tasking have to be solved:

- Risk of double evictions: when a dirty page is selected for eviction, it has to be copied back to disk. During the disk I/O, which is blocking under NACHOS, another process may trigger a page fault and thus may need to evict a page, and select the page currently under copy.
- It may happen that all physical pages are locked. In such a situation, the replacement algorithm has to be blocked until some pages get unlocked.
- When the page replacement algorithm gets blocked, for whatever cause of blocking, the state of the physical pages may have changed during the blocking period.

### 1.7.3 Executable file format

The source files are compiled into MIPS code by a gcc cross-compiler that generates executable code in the ELF format (*Executable and Linking Format*). This format is described in *elf32.h* located in directory *kernel*. ELF files are read in the constructor of class *Addrspace* to fill-in the page tables according to the memory layout described in the ELF file (see Section 1.9.3).

## 1.8 Utilities (directory *utility*)

NACHOS includes basic debugging facilities, and provides classes implementing lists and bitmaps. These utilities are described hereafter.

### 1.8.1 The debugging routines (files *utility.h, utility.cc*)

NACHOS offers functions to print debug messages. Debug messages have a type. When NACHOS is started, NACHOS command line arguments allow to select the list of message types to be displayed. Each type of debug message is identified by a flag. The list of flags can be extended if you need to. The following flags are predefined:

- '+' - groups all types of messages;
- 'a' - address spaces;
- 'd' - device drivers;
- 'e' - exceptions (including system calls);
- 'f' - file system;
- 'h' - machine peripheral devices (including disk);
- 'i' - interrupt handlers;
- 'm' - machine processor;
- 's' - synchronization tools;
- 't' - threads and processes;
- 'u' - utilities;

- 'v' - virtual memory management.

The two following routines are exported:

- **void DEBUG (char flag, char *format, ...)**: displays a message if the message type identified by *ag* is enabled. The format string passed as a parameter is the standard format string used by *printf*.

- **ASSERT(condition)**: displays an error message when *condition* does not hold. The displayed message identifies the line and name of the source file including the ASSERT statement.

### 1.8.2 Lists (file *list.h*)

NACHOS includes an interface to manipulate singly linked lists of elements. The type of elements in the list is generic. C++ templates are used for that purpose. List elements appear in the interface of class *List* as untyped pointers (void *, pointer to anything). Class *List* exports the following methods:

- **List()**: class constructor that creates an new empty list;

- **~List()**: destructor of class *List*. It empties the list, but does *not* deallocate the list elements;

- **void Prepend(void *item)**: inserts the element pointed to by *item* at the front of the list;

- **void Append(void *item)**: inserts the element pointed by *item* at the end of the list;

- **void *Remove()**: removes the first element from the list and returns a pointer to the removed element;

- **void Mapcar(VoidFunctionPtr func)**: applies function *func* to each element of the list;

- **bool IsEmpty()**: returns true if the list is empty, false otherwise;

- **void SortedInsert(void *item, Priority sortkey)**: inserts an item in the list in increasing order of priorities;

- **void * SortedRemove(Priority *keyPtr)**: removes the first element from a sorted list. The priority of the removed element is returned at address *keyPtr*;

- **bool Search(void *item)**: returns true if the element pointed to by *item* is contained in the list;

- **void RemoveItem(void *item)**: removes the element pointed to by *item* if contained in the list.

### 1.8.3 Bitmaps: class Bitmap (files *bitmap.cc , bitmap.h*)

NACHOS defines bitmap objects in class *Bitmap*. A bitmap is an array of bits that can be independently set and cleared. Bitmaps are used in NACHOS for the allocation of disk sectors. The *Bitmap* class exports the following methods:

- **Bitmap(int nitems)**: bitmap constructor for *nitems* bits. Memory for the bitmap is allocated by blocks of 32-bits. Initially, all bits are cleared;
- **void Mark(int which)**: sets bit number *which* to 1;
- **void Clear(int which)**: sets bit number *which* to 0;
- **bool Test(int which)**: returns true if bit number *which* is set, false otherwise;
- **int Find()**: returns the rank of the first bit of value 0 in the bitmap and sets it to 1. The method returns -1 in case all bits are set;
- **int NumClear()**: returns the number of cleared bits;
- **void Print()**: displays the contents of a bitmap;
- **void FetchFrom(OpenFile *file)**: fetches a bitmap from a NACHOS file;
- **void WriteBack(OpenFile *file)**: writes the contents of a bitmap to a NACHOS file.

## 1.9 My first Nachos program

This section first gives the complete list of NACHOS system calls and library functions. Instructions to develop, compile, execute and debug NACHOS programs are then given.

### 1.9.1 System calls (directory *userlib*, file *sys.s*)

The system calls interface is given in *syscall.h*, in directory *userlib*. The corresponding code, written in the MIPS assembly language, is given in file *sys.s*, in directory *userlib*. Unless it is otherwise stated, system calls return *NO_ERROR* on success, or -1 in case of error. An error code for the last system call invoked is kept by the NACHOS kernel (see files *kernel/msgerror.**), and a system call (*PError*) displays the corresponding error message.

#### Errors handling

- **void Perror(char *msg)**: displays the error message corresponding to the last system call, or message *\no error"* when the last system call terminated normally. Parameter *msg* is displayed before NACHOS error message to customize the display.

#### Threads management

- **void Halt()**: stops NACHOS, after having displayed statistics on the programs that were executed since NACHOS booted.

- **void SysTime(Nachos_Time *t)**: returns the current NACHOS time. Time is incremented for every operation (execution of an instruction, input/output operations, etc).

- **void Exit(int status)**: exits from the currently running multi-threaded process with exit code *status* (*NO_ERROR* in case of normal termination).

- **ThreadId Exec(char *filename)**: creates and executes a new process. Parameter *filename* is the name of the executable file. The newly created process initially executes in one single thread. The system call returns the identifier of the created thread.

- **ThreadId newThread (char *debug_name, VoidFunctionPtr func, int arg)**: creates and executes a new thread within the currently executing process. Parameter *debug_name*, used for debugging purposes, is the thread's external name. Parameter *func* is the address of the function to be executed by the thread, and *arg* is its unique parameter. The system call returns the identifier of the created thread. **Warning:** This system call should not be called directly by user programs, because it only supports threads that end with a call to *Exit*. NACHOS library function *threadCreate* should be called instead of *newThread*, because it manages thread termination even if the terminating thread does not explicitely call *Exit*.

- **int Join(ThreadId id)**: blocks the calling thread until thread of identifier *id* terminates.

- **void Yield()**: voluntarily relinquishes the processor; the calling thread stays ready but is put at the end of the ready list.

- **void PError(char *msg)**: displays a message corresponding to the last system call invoked, with string *msg* as a prefix.

**File system**

- **int Create(char *filename, int size)**: creates a new file of name *filename* and initial size *size*.

- **OpenFileId Open(char *name)**: opens the file with name *filename* and returns the file descriptor.

- **int Write(char *buffer, int size, OpenFileId id)**: writes *size* bytes from *buffer* to open file *id*, and returns the number of bytes actually written. Writing is performed at the current position in the file. If *id* is *ConsoleOutput*, then the contents of *buffer* is displayed onto the console.

- **int Read(char *buffer, int size, OpenFileId id)**: reads *size* bytes from the open file *id*, at the current position in the file, to *buffer* and returns the number of bytes actually read. If *id* is *ConsoleInput*, then the string is read from the keyboard.

- **int Close(OpenFileId id)**: closes the file identified by *id*.

- **int Remove (char *filename)**: deletes the file of name *filename*. Returns 0 on success.

- **int Mkdir(char *dirname)**: creates a new directory of name *dirname*.

- **int Rmdir(char *dirname)**: deletes the directory of name *dirname*.

- **int Mmap(OpenFileId f, int size)**: maps the first *size* bytes from file *f* onto the address space of the calling process. The result is the virtual address where the file can be accessed, or -1 on failure. Parameter *f* is the file descriptor (the file must be opened before calling *Mmap*). The requested size is rounded up to an integral number of pages.

**Synchronization**

- **Semid SemCreate(char * debug_name, int count)**: creates a semaphore named *debug_name* (used for debugging only). The initial counter value is set to *count*. The system call returns the identifier of the created semaphore.

- **int SemDestroy(SemId sema)**: deletes semaphore *sema*.

- **int V(SemId sema)**: calls primitive V on semaphore *sema*.

- **int P(SemId sema)**: calls primitive P on semaphore *sema*.

- **lockId LockCreate(char * debug_name)**: Creates a lock named *debug_name* and returns its identifier.

- **int LockDestroy(LockId lockid)**: destroys lock *lockid*.

- **int LockAcquire(LockId lockid)**: acquires the lock (blocks until the lock is released if currently used).

- **int LockRelease(LockId lockid)**: releases the lock and wakes-up one waiting thread (if any).

- **CondId CondCreate(char * debug_name)**: creates a condition variable named *debug_name*, and returns its identifier.

- **int CondDestroy(CondId condid)**: destroys condition variable *condid*.

- **int CondWait(CondId condid)**: waits for a condition variable to be signaled. The calling process is blocked until the condition is signaled.

- **int CondSignal(CondId condid)**: signals a condition, and wakes up exactly one thread in case there is at least one thread waiting for the condition. If no thread is waiting, the system call has no effect.

- **int CondBroadcast(CondId condid)**: same behavior as *CondSignal* except that *all* threads blocked on the condition are waken up.

**Serial communication interface**

- **int TtySend(char *msg)**: sends the null-terminated message *msg* using the serial communication hardware. Returns the number of bytes actually sent.

- **int TtyReceive(char *msg, int length)**: receives a message using the serial communication hardware and copies it at address *msg*. Parameter *length* is the maximum message length (size in bytes of allocated memory for *msg*). The returned value specifies the number of bytes actually received.

### 1.9.2 Nachos C library (directory *userlib*, files *libnachos.cc, libnachos.h*)

The NACHOS C library, in files *libnachos.c* and *libnachos.h* provides higher-level functions than the system calls when useful. In order to differentiate more easily functions of the NACHOS C library from functions of the standard C library, functions names of the NACHOS C library are prefixed by *"n_"* (e.g. *"n_printf"*).

**Thread management functions**

- **ThreadId threadCreate(char *debug_name, VoidNoArgFunctionPtr func)**: creates a new thread of name *debug_name* (used for debugging only). Parameter *func* is the address of the function to be executed by the new thread. Function *threadCreate* invokes system call *newThread* and handles properly thread termination (automatic call to *Exit* when *func* terminates). It returns the identifier of the created thread. This library function is the one to be called by user programs, that must not call system call *newThread* directly.

**Input/output functions**

- **void n_printf(char * format ,...)**: this overly simplified version of *printf* is a formatted display function. The format string *format* allows to print: characters (%c), strings (%s), integers in decimal representation (%d or %i), integers in hexadecimal representation (%x) or single-precision floating point values (%f). Like in the standard *printf* , special characters like \n (new line) or \t (tabulation character) can be used in the format string.

**String management functions**

- **int n_strcmp(const char *s1, const char *s2)**: lexicographically compares two null-terminated strings *s1* and *s2* and returns an integer greater than, equal to, or less than 0, depending if the string *s1* is greater than, equal to, or less than the string *s2* with respect of the lexical order from left to right;

- **char *n_strcpy(char *dst, const char *src)**: copies the null-terminated string *src* to *dst* (including the terminating '\0' character).

- **size_t n_strlen(const char *s)**: computes the length of the string *s* (terminating '\0' character excluded).

- **char *n_strcat(char *dst, const char *src)**: appends a copy of the null-terminated string *src* to the end of the null-terminated string *dst*, then adds a terminating '\0'. The string *dst* must have sufficient space to hold the result.

- **int n_toupper(int c)**: converts a lower-case letter to the corresponding upper-case letter.

- **int n_tolower(int c)**: converts an upper-case letter to the corresponding lower-case letter.

- **int n_atoi(const char *str)**: converts the string pointed to by *str* to integer representation.

- **int n_read_int ()**: reads an integer from the standard input.

**Memory manipulation operations**

- **void * n_memcpy(void *b1, const void *b2, size_t n)**: copies *n* bytes from memory area *b2* to memory area *b1*. Returns the original value of *b1*.

- **int n_memcmp (const void *b1, const void *b2, size_t n)**: compares memory area *b1* to memory area *b2*. Both areas are assumed to be *n* bytes long. The function returns zero if the two strings are identical, otherwise it returns the difference between the first two differing bytes

- **void * n_memset(void *b, int c, size_t n)**: sets *n* bytes of memory area *b* to value *c*.

### 1.9.3 Compiling a user program

User programs are written in C and are compiled using a standard MIPS cross-compiler (gcc) that generates executables in the ELF format. To compile a user program *prog.c*, simply add the new target *prog* in file *Make le* located in directory *test*:

```
PROGRAMS = hello halt matmult shell out prog
```

Then, type *gmake* in directory *test* and all user programs identified in the *Make le* will be generated. When your program will be loaded by NACHOS, the regions of the program address space will be displayed as they appear in the ELF file:

```
**** Loading file /prog:
        - Section. .sys: file offset 0x1000, size 0x240, 0x2000 addr VM, R/X
        - Section. .text: file offset 0x2000, 0x1e30 size, addr 0x4000 VM, R/X
        - Section. .rodata: file offset 0x4000, size 0x54, addr 0x8000 VM, R
        - Program start address: 0x4000
```

The ELF executable file format divides programs into *sections* corresponding to the different types of memory regions found in any program: *.text* for the code, *.bss* for uninitialized data, *.data* and *.rodata* for initialized data, *.sys* (specific to NACHOS) for the operating system). An ELF file may contain all or only a subset of these section types.

A section header, as shown above, contains the locations of the regions in the executable file ( *le o set*, in bytes), the sections sizes (*size*, in bytes), the virtual addresses of the sections (*VM addr*), and the sections protection attributes (*R* for reading, *W* for writing, *X* for execution[3]).

### 1.9.4 Compiling NACHOS

To compile NACHOS, just type *gmake* at the root of NACHOS directory. Thanks to the *Make le* file, the modified source files will be compiled and linked together to generate the NACHOS executable file (named *nachos* and located at the root of the NACHOS source tree). To speed-up compilation, only source files that were modified since last compilation are re-compiled. Type *gmake clean* to clean-up NACHOS source tree (removes object files, dependency files, etc).

Should you need to add new source files (which normally is not required), you will have to change the main *Make le* or one of the *Make le* files located in sub-directories (*kernel*, *machine*, ...).

### 1.9.5 Executing Nachos (directory *kernel*, files *main.cc, system.cc*)

To execute user programs, the NACHOS executable has to be launched. NACHOS takes parameters as command-line arguments or through a configuration file. NACHOS command-line arguments are the following:

**-d <flags>:** will display debug messages of the indicated type(s). For instance, nachos-d fm displays messages about the file system and processor simulation. If no message type is given, or if message type '+' is given, *all* message types are displayed (which is far too verbose). Message types are defined in Section 1.8.1, page 31.

**-s:** starts NACHOS in step-by-step mode; the processor registers and pending interrupts are displayed after executing each instruction; typing any character causes the execution of the next instruction.

**-x < file>:** starts NACHOS and executes user program with name *le* after booting NACHOS. The user executable file has to be loaded in the NACHOS file system.

**-z** displays the NACHOS copyright message.

**-f <filename>:** uses the configuration file <filename> instead of the default configuration file *nachos.cfg*.

### 1.9.6 Nachos configuration file (directory *utility*, file *config.cc, config.h*)

The NACHOS configuration file allows to the parameters of NACHOS without re-compiling it. By default, NACHOS uses as a configuration file *nachos.cfg* located in the current directory. The name of the configuration file can be changed using command-line option *-f*. NACHOS configuration file is made of a sequence of lines following this syntax:

```
<parameter name> = <parameter value>
```

Parameters fall into the categories listed below.

[3]The execution privilege (X) is currently ignored by NACHOS.

**General parameters of Nachos**

`PrintStat = [0 | 1]`

If 1, displays statistics on all executed processes at the termination of Nachos.

`NumPhysPages = <integer>`

Number of physical pages in the memory of the emulated machine.

`MaxVirtPages = <integer>`

Maximum number of virtual pages in the address space of a process.

`UserStackSize = <integer>`

Size (in bytes) of user stacks.

`ProcessorFrequency = <integer>`

Specifies the frequency of the MIPS emulation (in MHz). Used to study the (approximative) impact of the CPU frequency on performance of applications, with identical device latency.

`SectorSize = <integer>`

Size in bytes of a disk sector. Must be a power of two.

`PageSize = <integer>`

Size in bytes of a virtual page (also size of a physical size). Must be a power of two, and must be identical to the sector size.

`UseACIA = [None | BusyWaiting | Interrupt]`

Indicates whether the serial communication hardware is not present (value *None*), is present and is managed using busy waiting (value *BusyWaiting*) or is present and configured in interrupt mode (Value *Interrupt*). By default, if you are not using the serial communication hardware set to *None* since the simulation of the serial communication hardware is very time-consuming.

`TargetMachineName = <name>`

The name of the target machine, used for simulating the serial communication hardware.

`NumPortLoc = <integer>`

The local port number ($> 1024$) used for serial communication emulation (TCP local port number). This parameter is ignored when *UseACIA* is set to *None*.

`NumPortDist = <integer>`

The remote port number ($> 1024$) used for serial communication emulation (TCP remote port number). This parameter is ignored when *UseACIA* is set to *None*. The two port numbers (local and remote) must be inverted in the configuration files of two machines that communicate.

**File system**

These parameters allow to configure the contents of the file system before NACHOS is started. The resulting disk accesses are not accounted for in NACHOS statistics.

`ProgramToRun = <name>`

Gives the absolute name of the file to be executed after NACHOS is started. Equivalent to the `-x` command-line argument.

`FormatDisk = [0 | 1]`

If *FormatDisk* is set to 1, NACHOS disk is emptied at boot time. Else, disk contents is kept from the previous runs of NACHOS.

`ListDir = [0 | 1]`

Displays the contents of the file system hierarchy (file names together with their location in the directory hierarchy).

`PrintFileSyst = [0 | 1]`

Displays the contents of all files in the file system, byte by byte.

`FileToPrint = <filename>`

Displays the contents of a NACHOS file, byte by byte.

`FileToCopy = <Unix filename> <Nachos filename>`

Copies a Unix file (of absolute or relative filename *Unix lename*) to the NACHOS file with absolute name <*Nachos lename*>.

`FileToRemove = <Nachos filename>`

Removes file of absolute name *Nachos lename* from the NACHOS file system.

`NumDirEntries = <integer>`

Specifies the maximum number of entries per directory.

`DirToMake = <Nachos dirname>`

Creates a directory with absolute name <*Nachos lename*> into the NACHOS file hierarchy.

`DirToRemove = <Nachos dirname>`

Deletes the directory with absolute name <*Nachos lename*> from the NACHOS file hierarchy.

Several directives of type *FileToCopy, FileToRemove, DirToMake* and *DirToRemove* can be provided to initialize the NACHOS file system at boot time. They are executed in the order of their declaration in the configuration file.

Every parameter has a default value (see file *con_g.cc*, constructor of class *Con_g*). When NACHOS is started, all parameters are stored in a global object *g_cfg*, of class *Con_g*. Class has public members only, one for every configuration parameter. For instance,

```
g_cfg->PageSize;
```

returns the page size.

# Chapter 2

# SGP assignments (18h)

The first three assignments consist in using the Linux API for managing processes (heavyweight processes and lightweight processes, pthreads) and synchronization tools. All system calls are not fully documented in this document. For full documentation, use the *man* pages of the system calls (type *man function_name*, *man -s name 2 function_name* or *man -s name 3 function_name*).

Please respect the naming conventions of the files you will have to send to the professors.

## 2.1 Assignment 1 (Linux) - Use of system calls fork, exec, wait, pipe

### 2.1.1 System calls fork, exec and wait

The *fork* system call causes the creation of a new (heavyweight) process. The new process (child process) is an exact copy of the calling process (parent process), code, data, stack, except for the following:

- the child process has a different process identifier;
- the child process has its own copy of the parent's file descriptors;

After a call to *fork*, both the parent and the child process continue their execution after the call to *fork*. The difference between the two processes is that *fork* returns a value of 0 to the child process and returns the process identifier of the child process to the parent process.

The *exec* family of functions replaces the current process image with a new process image. In this assignment, the *execl* variant will be used.

Finally, system call *wait* allows a process to wait for the termination of child processes..

**Assignment.** Write a program, named *part1.c* that creates a child process using *fork*. The child process calls *execl* to execute a program generated from a source file *part2.c*, which displays a message (for example *"I am the child"*) and exits with a specific exit status. The parent process waits for the termination of the child and displays the child's exit status.

### 2.1.2 Pipes and named pipes

#### Pipe

A pipe is a uni-directional FIFO (*First-In First-Out*) communication channel between two processes. A pipe has a limited capacity, causing the blocking of the writing process in case the pipe is full. Pipes are created using the *pipe* system call, which creates a new pipe and returns a pair of file descriptors referring to the read and write ends of the pipe. Reading and writing are then performed using the standard file operations (*read* and *write*). The typical way to share a pipe between two processes is to create a pipe and then fork the process such that the child process inherits the read and write file descriptors.

Attempts to write in a pipe with read and write descriptors closed causes an abortion of the calling process. Calls to read on an empty pipe with write descriptors closed returns 0.

**Assignment.** Use a pipe to implement a producer-consumer scheme between two processes. The producer reads a sequence of lines from the standard input, using function *fgets*, and writes the lines in a pipe. The producer process stops producing when an empty line is read. The consumer process reads the produced information from the pipe and simply displays the received information. The source code will be in a single file *prodcons.c* and the producer process will create the consumer process using *fork*.

#### Named pipe

A named pipe, also known as a FIFO, is an extension to the traditional pipe concept, also known as *anonymous* pipe. A traditional pipe persists only for the duration of the process that uses it. In contrast, a named pipe is persistent and has to be explicitly deleted. Named pipes appear in the file system; a specific entry type character (p) is printed when using command *ls -l* to indicate that the entry is a pipe and not a regular file.

System call *mknod*, when called with parameter *mode* equal to *S_IFIFO | 0666* creates a new named pipe with read and write permissions for all processes (0666). Once the named pipe is created, standard file operations (*open, read, write, close*) can be used.

If a process tries to open a named pipe for reading, the process is suspended if no process has opened the pipe for writing; the opening proceeds when a process opens the pipe for writing. Similarly, a process that opens a pipe in write mode will be suspended until the pipe in opened in read mode.

**Assignment.** Re-develop the previous assignment using named pipes. The producer and consumer processes will be independent processes, with no parent-child relationships. They will be located in separated files *prod.c* and *cons.c*. The producer will create the named pipe and the consumer will delete it.

### 2.1.3 System call interface

- #include <sys/types.h >
  #include <unistd.h>
  pid_t fork(void)

This function creates a new process (child process) which is an exact copy of the calling process (parent process). The child process inherits some characteristics from its father process (environment variables, file descriptors, file creation mask, ... ). Upon successful completion, *fork* returns a value of 0 to the child process and returns the process identifier of the child process to the parent process. A return value of -1 is returned when the execution of *fork* fails.

- #include <unistd.h>
  int execl (const char *path, const char *arg0, ..., const char *argn, char * / * NULL * /)

  The function *execl* replaces the current process image with a new process image. Arguments *arg0, arg1, ..., argn* together describe a list of pointers to null-terminated strings that represent the arguments list available to the executed program (through argc/argv parameters). The first argument, by convention, should point to the file name associated with the file being executed. The list of arguments must be terminated by a NULL pointer. The function returns *-1* in case of error.

- #include <stdlib.h>
  void exit (int status)

  Terminates the calling process. The *status* parameter is the return status of the process (by convention, 0 indicates a normal termination). The return value is made available to a parent process which has called *wait*.

- #include <sys/types.h>
  #include < sys/wait.h>
  pid_t wait(int *stat_loc)

  The *wait* function suspends execution of the calling process until a child process terminates, or a signal is received. The function returns the identifier of the terminated process, or -1 in case of error. When parameter *stat_loc* is not NULL, it reports termination information about the process that exited. Macros *WIFEXITED* and *WEXITSTATUS* may be used respectively to detect if the child process has terminated normally, and to return its termination status.

- #include <unistd.h>
  int pipe (int fd [2]);

  The *pipe* function creates a pipe and allocates a pair of file descriptors. *fd[0]* is the file descriptor for reading from the pipe, whereas *fd[1]* is the file descriptor for writing into it. On successful creation of the pipe, 0 is returned. Otherwise, a value of -1 is returned.

- #include <fcntl.h>
  #include <unistd.h>
  #include < sys/stat.h>
  int mknod (const char *path, mode_t mode, dev_t dev)

  The *mknod* function allows (among others) to create named pipes that will appear in the file system under name *path*. The *mode* parameter is used to specify at the same time the type of object to be created and the protection on the object (here, S_IFIFO | 0666). The parameter *dev* is ignored when a FIFO is created. *mknod* returns 0 upon successful completion and -1 otherwise.

- #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>
  int open(const char *filename, int oflag)

  Opens the file named *filename* with the open mode specified by the parameter *oflag* (O_RDONLY for reading, O_WRONLY for writing). If successful, *open* returns a non-negative integer (file descriptor). It returns -1 on failure.

- #include <unistd.h>
  int close(int descr)

  Closes the file identified by file descriptor *descr*.

- #include <unistd.h>
  int read(int descr, void *buf, size_t n)

  Reads *n* bytes from the file identified by descriptor *descr* into the buffer pointed to by *buf*. Returns the number of bytes read, 0 on end of file, or -1 on failure.

- #include <unistd.h>
  int write (int descr, const void *buf, size_t n)

  Writes *n* bytes from the buffer pointed to by *buf* in the file identified by *descr*. Returns the number of bytes written or -1 on failure.

- #include <stdio.h>
  char *fgets (char *buf, int size, FILE *stream)

  Reads at most *size* bytes from the given stream and stores them into the buffer *buf*. Reading stops when a newline character is found, at end-of-file or when an error occurs. The newline, if any, is retained. If any characters are read and there is no error, a '\0' character is appended to end the string. Upon successful completion, the function returns a pointer to the string. If end-of-file occurs before any characters are read, it returns NULL. To read from the standard input parameter *stream* should be set to *stdin*.

- #include <stdio.h>
  void perror(const char *s)

  The function finds the error message corresponding to the current value of the global variable errno and writes it, followed by a newline, onto the standard error stream. If the argument *s* is non-NULL and does not point to the null character, this string is prepended to the message

- #include <unistd.h>
  int unlink(const char *path)

  The *unlink* function removes the link named by *path* from its directory and decrements the links count of the file which was referenced by the link. If that operation reduces the links count of the file to zero, then all resources associated with the file are reclaimed.

## 2.2 Assignment 2 (Linux) - Use of signals, setjmp, longjmp and ptrace

### 2.2.1 Signals

A signal is an asynchronous notification sent to a process in order to notify it of an event that occurred. When a signal is sent to a process, the operating system interrupts the process's normal execution flow to deliver the signal. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed. Signals correspond to abnormal situations (invalid memory references, division by zero, etc) or can simply be used for inter-process communication.

The table below lists the main signals available on our platforms: signal name, signal number, action performed by the signal handler, and description of the signal. The complete list of signals can be found in the included file <*signal.h*>. A more detailed description can be found in the manual page dedicated to signals (*man signal*).

| Name | Number | Action | Description |
|---|---|---|---|
| SIGHUP | 1 | Term | Hangup detected on controlling terminal or death of controlling process |
| SIGINT | 2 | Term | Interrupt from keyboard |
| SIGQUIT | 3 | Core | Quit from keyboard |
| SIGILL | 4 | Core | Illegal Instruction |
| SIGABRT | 6 | Core | Abort signal from abort |
| SIGFPE | 8 | Core | Floating point exception |
| SIGKILL | 9 | Term | Kill signal |
| SIGSEGV | 11 | Core | Invalid memory reference |
| SIGPIPE | 13 | Term | Broken pipe: write to pipe with no readers |
| SIGALRM | 14 | Term | Timer signal from alarm |
| SIGTERM | 15 | Term | Termination signal |
| SIGUSR1 | 30,10,16 | Term | User-defined signal 1 |
| SIGUSR2 | 31,12,17 | Term | User-defined signal 2 |
| SIGCHLD | 20,17,18 | Ign | Child stopped or terminated |
| SIGCONT | 19,18,25 | Cont | Continue if stopped |

A default signal handling routine exists for all signals. By default, most signals result in the termination of the process receiving them; some signals instead are simply discarded if the process has not requested otherwise.

The *kill* system call allow a process to send a signal, identified by its number, to another process.

Except for the SIGKILL and SIGSTOP signals, the *signal* function allows a signal to be caught or to be ignored. The more general function *sigaction*, with a slightly different interface can also be used. If the signal handling routine associated to the received signal does not terminate the target process, the instruction where the program was interrupted is re-executed.

Each process has a signal mask, which defines the set of masked signals (those signals that are blocked from delivery). The *sigprocmask* function, when called, examines and/or changes

the current signal mask.

**Assignment.** Write a program which uses the *signal* function to catch "divide by zero" exceptions (a signal of type *SIGFPE* is sent when dividing an integer by value 0). The source file containing your code will be named *div0.c*.

**Assignment.** Write a program that displays its progress (e.g. value of a loop index) periodically, every second. The program, contained in file *tictock.c* will call the *alarm* function, which delivers a (unique) *SIGALRM* signal to a process after a given delay.

### 2.2.2 Functions setjmp and longjmp

The *setjmp* function saves the current execution environment of the calling process (basically, processor registers) into a buffer given as parameter. The *longjmp* function restores the execution environment previously saved by the *setjmp* function.

**Assignment.** Modify the source code of *tictock.c* file such that every time a `ctrl-C` is typed, the process re-starts its execution from the beginning. A simple numerical application such as matrix-multiplication will be used as an example. Note that in the situation when a signal handling routine calls the *longjmp* function, the signal handled by the routine is not unmasked and thus subsequent occurrences of the signal will not be caught. Function *sigprocmask* will be called to solve this issue. The source file containing your work will be called *tictockforever.c*.

### 2.2.3 The ptrace function

The *ptrace* function provides tracing and debugging facilities. It allows one process (the tracing process) to control another (the traced process). Most of the time, the traced process runs normally, but when it receives a signal, it stops. The tracing process is expected to notice this via a call to the *wait* function, examine the state of the stopped process, and cause it to terminate or continue as appropriate. *ptrace* is the mechanism controlling all those actions.

**Assignment.** Write a program mainly composed of a loop with a loop index. The execution of that program will be controlled by another process, that will periodically display the progress of the controlled process (value of the loop index). To support address space randomization (ASLR) easily, *fork()* will be used to have the two processes (else ASLR raises issues to obtain the address of the monitored variable). Your source files will be named *monitor.c*.

### 2.2.4 Interface of system calls

- #include <sys/types.h>
  #include <signal.h>
  int kill( pid_t pid, int sig);

  The *kill* function sends the signal specified by *sig* to process with identifier *pid*. Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned.

- #include <signal.h>
  void (*signal( int sig, void(*func)(int))) (int);

  The *signal* function connects a signal handling routine *func* to signal number *sig*. Setting *func* to *SIG_DFL* connects the default signal handling routine, whereas setting *func* to *SIG_IGN* causes the signal to be ignored. The previous action is returned on a successful call. Otherwise, -1 is returned.

- #include <unistd.h>
  unsigned int alarm(unsigned int sec);

  Sends signal *SIGALRM* to the calling processes after *sec* seconds.

- #include <setjmp.h>
  int setjmp(jmp_buf env);

  Saves the execution environment of the calling process (basically, the processor registers) into buffer *env*. The function returns 0 if returning directly, and any other value when returning from a call to *longjmp*. In this cas, the returned value is the parameter passed to *longjmp*.

- #include <setjmp.h>
  void longjmp(jmp_buf env, int val);

  *longjmp* restores the environment saved by the last call to *setjmp* with the corresponding *env* argument. After *longjmp* is completed, program execution continues as if the corresponding call of *setjmp* had just returned the value *val*.

- #include <sys/ptrace.h>
  long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);

  The value of request determines the action to be performed (only some of the requests are described below, see the man page for further details):

  - PTRACE_TRACEME: indicates that this process is to be traced by its parent. Any signal (except SIGKILL) delivered to this process will cause it to stop and its parent to be notified via wait. Parameters *pid*, *addr* and *data* are ignored.
  - PTRACE_PEEKTEXT, PTRACE_PEEKDATA: reads a word at location *addr* in the child's memory, returning the word as the result of the *ptrace* call. (The argument *data* is ignored.)
  - PTRACE_POKETEXT, PTRACE_POKEDATA: copies the word data to location *addr* in the child's memory.
  - PTRACE_CONT: restarts the stopped child process. If *data* is non-zero and not *SIGSTOP*, it is interpreted as a signal to be delivered to the child; otherwise, no signal is delivered. Thus, for example, the parent can control whether a signal sent to the child is delivered or not. (*addr* is ignored.)

## 2.3 Assignment 3 (Linux) - POSIX threads (pthreads)

### 2.3.1 Introduction to the pthreads library

POSIX specifies a set of interfaces (functions, header files) for threaded programming, commonly known as POSIX threads, or pthreads. A single process can contain multiple threads, all of which are executing the same program. These threads share the same global memory (data and heap segments), but each thread has its own stack (automatic variables).

The *pthreads* library provides, among others, primitives for:

- thread creation and deletion;
- control of thread scheduling;
- synchronization between threads, through:
  - mutual exclusion semaphores;
  - condition variables;
  - reader/writer semaphores.

The main primitives provided by the *pthreads* library are listed below. The table does not contain all functions (for a detailed list of functions, type *man pthreads*). All primitives return 0 on success and an error code on error. File *pthread.h* has to be included in your source code. Option *-lpthread* has to be put in your link command to link your code with the *pthreads* library.

| **Name** | **Role** |
|---|---|
| pthread_create | Creates a new thread within a process |
| pthread_detach | Marks a thread as detached (will be deleted when finished) |
| pthread_exit | Terminates the currently running thread |
| pthread_join | Suspends the execution of the current thread and waits for the target thread to terminate |
| pthread_self | Returns the thread identifier of the calling thread |
| pthread_attr_setschedparam | Sets the scheduling parameter attribute of the calling thread |
| pthread_attr_setschedpolicy | Sets the scheduling policy attribute of the calling thread |
| pthread_mutex_init | Initializes a mutex |
| pthread_mutex_lock | Locks a mutex; blocks the lock becomes available |
| pthread_mutex_unlock | Unlocks a mutex |
| pthread_cond_broadcast | Unblocks all threads blocked on the specified condition variable |
| pthread_cond_init | Initializes a condition variable |
| pthread_cond_signal | Unblocks at least one of the threads blocked on a condition variable |
| pthread_cond_wait | Blocks on a condition variable |
| pthread_rwlock_init | Initializes a read-write lock |
| pthread_rwlock_rdlock | Locks a read-write lock for reading |
| pthread_rwlock_unlock | Unlocks a read-write lock |
| pthread_rwlock_wrlock | Locks a read-write lock for writing |

### 2.3.2 Assignment

Write a parallel matrices multiplication program, in file *matmult.c*. The source matrices M1 and M2 and the result matrix M will be 64x64 matrices of floats. We recall the formula for computing the product of two square matrices:

$$\forall i \in [0;63], \forall j \in [0;63], M[i,j] = \sum_{0}^{63} M1[i,k] * M2[k,j]$$

The parallel version of your matrices product will use 8 *slave* threads created by the main *master* thread. Row $i$ in the result matrix will be assigned to slave thread *i modulo 8*.

## 2.4 Assignment 4 (Nachos) - getting started with Nachos (homework)

### 2.4.1 Instructions for the assignments

- For us to easily find your code in the middle of the existing code, you are asked to surround your code by *ifdef/endif* directives as follows:

```
#ifndef ETUDIANTS_TP
  <the old code that do not want to be executed>
#endif
#ifdef ETUDIANTS_TP
  <the beautiful code you have written>
#endif
```

  These instructions are **MANDATORY** and are also very useful (for the professors to identify the code you have changed when we will grade your work, and for you to detect your changes during the debugging phase).

- You **must not** change the code in the *machine* directory (who would change the machine on real hardware?). Similarly, the code portions in charge of saving/restoration of the simulation context must not be changed.

- Make intensive use of NACHOS tracing facilities that allow you to trace what's going on at various levels of detail.

- Make intensive use of a debugger. *gdb* is installed on all machines at ISTIC, and a small tutorial is given in Appendix A.

### 2.4.2 Homework: exploration of Nachos

You are asked to answer the following set of questions about NACHOS internals. The overall objective of these questions is to make you read (again and again) this document, the source code of NACHOS and the set of *html* pages automatically generated from the source code. You might use command *grep* to search for a string (variable, function name) in a file or set of files (type *man grep*).

#### System call mechanism

Review and document the execution of a user program that invokes a system call: list and explain all the files, functions/methods involved in the execution of a system call, both on user and system mode. For instance, carefully follow the execution of *test/hello.c*, which simply displays a *hello world* message.

You are asked to list the main functions and methods called all along the execution of the system call, in user *and* kernel mode, and for each function/method describes its role.

**Management of threads and processes**

1. Describe what method *SaveProcessorState* will contain when it is coded
2. Which variable corresponds to the list of threads that are ready to execute? Does the active thread belong to that list? Which variable identifies the active thread?
3. What is the role of variable *g_alive*? What is the difference with the *readyList* member of class *Scheduler?*
4. How do the list management routines manage the memory when inserting new elements? Do they dynamically allocate memory for the newly inserted elements or do they simply link the element in the list structure? Explain the rationale behind this design choice.
5. What happens to an object of class *Thread* when the thread is blocked on a semaphore?
6. How is the atomicity of system calls ensured?
7. What is the role of the *SwitchTo* method of class *Scheduler*? What are the respective roles of variables *thread_context* and *simulator_context* of class *Thread*? What should the *SaveSimulatorState* and *RestoreSimulatorState* methods do?
8. Explain the purpose of the *typeId* field present in all objects handled by the kernel (semaphores, tasks, threads, etc.)..
9. Think of one sadistic but intelligent question about NACHOS that you would like to see appearing here next year

**Development environment**

1. List the tools provided by NACHOS to help you debugging your code. For instance, what is provided to display the sequence of instructions executed by the MIPS simulator?
2. Is *gdb* able to debug NACHOS kernel code? NACHOS user code? Explain your answer.

### 2.4.3 Which document is expected by the professors

- For the homework: a document (whatever the format) answering the NACHOS exploration questionnaire given above.
- For the next assignments: the source code (modified files only, including the user programs developed to test your work) and a short report explaining what you have done, how it was tested and if everything executes as expected.

## 2.5 Assignment 5 (Nachos): scheduling and synchronization (4 lab slots)

To complete this assignment, it is only required to fully understand the contents of directories *kernel* and *userlib*, the documentation, and to a lesser extent the contents of directories *machine* and *utility*.

The initial version of NACHOS that was delivered to you compiles properly. Unfortunately, when the NACHOS executable is launched, user programs cannot be started (not even loaded) because synchronization tools are not implemented yet.

### 2.5.1 Synchronization tools

The objective here is to implement synchronization tools as described in § 1.4.2. In order to fully implement the synchronization tools, you have to:

1. Fill-in the *synch.cc* file, initially almost empty, to implement the code of semaphores, locks and condition variables. Condition variables don't need to tested at this early stage since they are not mandatory to boot the kernel. Keep in mind that atomicity of synchronization primitives has to be ensured.

2. Complete file *exception.cc* to implement the handling of exceptions relative to synchronization (check of system call arguments, call to the appropriate method in the kernel code, error handling). You have to pay attention to the following points: *(i)* parameters to the system calls have to be valid (using global variable *objectIDs*) and *(ii)* objects passed as parameters must have the expected type (checked using field *typeId*).

### 2.5.2 Thread management

In the initial version of NACHOS, only one process can execute at a time, it is single-threaded. The following changes will have to be made to support multi-threaded programs:

- Implement methods *SaveProcessorState* and *RestoreProcessorState* of class *Thread* that manage thread context.

- Implement method *Start* of class *Thread*. This method initializes all members of class *Thread*: allocates and initializes the user and kernel stacks, increments the number of threads per process, puts the new thread in the list of created threads and in the ready list. The following methods/functions will be used: method *StackAllocate* of class *AddrSpace* to allocate a new user stack; function *AllocBoundedArray* (see *machine/sysdep.h*) to allocate a new kernel stack; *InitSimulatorContext* and *InitThreadContext* of class *Thread*. The user stack size is specified in the configuration file. The size of the kernel stack is constant (constant *SIMULATORSTACKSIZE* in file *kernel/thread.h*).

- Implement method *Finish* of class *Thread* to manage thread deletion. Since the thread is still running when method *Finish* is invoked, the actual deletion is delayed until the next context switch. Method *Finish* only puts the thread to sleep (call to method *Sleep*) and marks the thread such that it will be deleted at the next context switch (update of field *g_thread_to_be_destroyed*). The context switch code (method *SwitchTo*) will have to be updated to actually delete the thread.

**Test programs.** Test your code extensively through the writing of programs testing:

- semaphores (e.g. a *rendez-vous* between two threads, aproducer/consumer scheme);
- locks and condition variables.

**Bonus (for the brave).** Develop one of the following add-ons:

- priority-based thread scheduling;
- time-sharing thread scheduling;
- barriers as an additional synchronization mechanism;
- addition of parameters at thread creation (*threadCreate*);
- addition of parameters to user programs.

## 2.6 Assignment 6 (Nachos): implementation of a character device driver (2 lab slots)

This assignment will require a study of the contents of directory *drivers* (files *drvACIA.h* and *drvACIA.cc*).

The objective of this assignment is to develop the driver for the serial communication hardware. Asynchronous transmission methods will be implemented. Two modes of operation for the driver will be developed: using busy waiting and using interrupts. Synchronization tools developed in the previous assignment need to be fully operational.

The device driver implements the interface described in § 1.3.1. The code is very similar to the code that was written during the tutorials.

To implement the device driver, you will have to:

1. Implement in file *drivers/drvACIA.cc:*
   - the class constructor *DriverACIA*
   - methods **ttySend** and **ttyReceive** in busy waiting mode. Mutual exclusion on the serial communication hardware will have to be ensured.
2. Write a test program with a sending process running on a machine and a receiving process running on another machine. Test and debug your code.
3. Repeat steps 1 to 3 to now operate in interrupt mode. Implement the interrupt service routines **interruptSend** and **interruptReceive**.
4. Compare the two methods operation using the statistics obtained.

**The configuration file.** Make sure to update the NACHOS configuration file *nachos.cfg* correctly: parameters *useACIA*, *TargetMachineName* and numbers of communication ports have to be changed in the default configuration file. Two configuration files are required: one for the sender, and one for the receiver. Make sure that port numbers are inverted in the two files.

# Chapter 3

# Travaux pratiques du module SGM (18h)

## 3.1 TP1 (Nachos) - Gestion de mémoire virtuelle (10h encadrées)

L'objectif de ce TP est de réaliser les deux éléments centraux d'un système de gestion de mémoire virtuelle :

- la routine de traitement des défauts de page qui charge en mémoire à la demande les pages depuis le disque ;
- un algorithme de remplacement de page, appelé également *voleur de page*.

On étudiera pour ce TP le contenu du répertoire *vm*. Le système de gestion de mémoire virtuelle doit correspondre à la description donnée au paragraphe **??**. Il est recommandé de tester votre système progressivement, lorsque vous aurez réalisé chacune des trois étapes suivantes.

### 3.1.1 Espaces d'adressage séparés

Dans la version de NACHOS mise en place lors du premier TP, un seul processus (multi-threadé si vous avez terminé la première partie de ce TP) s'exécute. Ce qui est demandé ici est de permettre l'exécution de plusieurs processus ayant des espaces d'adressage séparés (*i.e.* ayant chacun leur table des pages privée). Pour ce faire, étudier et modifier si nécessaire les méthodes *SaveProcessorState* et *RestoreProcessorState* de la classe *Thread* (gestion du champs *translationTable* de la MMU.

Dès que vous aurez terminé cette partie du TP, vous pourrez utiliser le *shell* fourni avec NACHOS.

### 3.1.2 Chargement des programmes à la demande

Jusqu'à présent, le code et les données des programmes étaient chargés en mémoire dès leur lancement. Il s'agit ici de changer le chargement de l'exécutable et l'allocation de la pile utilisateur de manière à ne pas allouer les pages en mémoire dès le chargement, mais plutôt de déclencher un défaut de page lors de leur premier accès, en mettant le bit *valid* de la table

de traduction d'adresses à *false*. Le code concerné est le constructeur de la classe *AddrSpace* et la méthode *StackAllocate* de la classe *AddrSpace*.

La routine de traitement des défauts de page s'occupera alors d'allouer une page physique, et de la remplir avec le contenu de la page virtuelle demandée, avant de redonner la main au thread ayant provoqué le défaut de page.

**Routine traitement des défauts de pages**

Écrire la routine de traitement des défauts de page (méthode *PageFault* de la classe *PageFaultManager*). Dans un premier temps, on supposera qu'il existe toujours une page de libre en mémoire réelle, et qu'il n'y a qu'un seul thread qui s'exécute dans le programme que l'on exécute. Les actions à réaliser par la routine sont les suivantes :

1. charger la page manquante depuis le disque (le numéro de secteur sera trouvé dans la table de traduction d'adresses, et aura été initialisée au chargement du programme en mémoire) dans une page temporaire. Dans le cas de pages anonymes (voir la section **??**), aucun chargement depuis le disque n'est à faire : il suffit de remplir la page temporaire avec des 0.

2. recherche d'une page libre en mémoire réelle (on utilisera pour cela le gestionnaire de mémoire physique - classe *PhysicalMemManager* du fichier *physMem.cc*). Copie du contenu de la page temporaire vers cet emplacement.

3. modifier la table de traduction d'adresses et la table des pages réelles en conséquence, en particuler dans la méthode *AddPhysicalToVirtualMapping*.

Les structures de données et constantes utilisées par la routine de traitement des défauts de page sont :

- la table des pages réelles ;
- le nombre de pages réelles, et la taille d'une page, qui sont accessibles via l'objet de configuration global *g_cfg* (champs *g_cfg->NumPhysPages* et *g_cfg->PageSize*) ;
- la table des pages du processus courant ;
- les objets modélisant le gestionnaire de swap (objet *g_swap_manager*) ;
- la mémoire de la machine, accessible via l'objet global g_machine (champ *g_machine->mainMemory*).

### 3.1.3 Algorithme de remplacement de page

Lorsque la mémoire physique est pleine et qu'un processus veut charger en mémoire une page qui est absente de la mémoire physique, la routine de traitement des défauts de page appelle un algorithme de remplacement de page, qui réquisitionne une page à un processus, en la recopiant sur disque si nécessaire. On vous demande ici d'implanter l'algorithme de remplacement de page qui a été présenté dans le paragraphe **??**. Ceci sera réalisé en complétant la méthode *EvictPage* de la classe *PhysicalMemManager*.

### 3.1.4 Trucs et astuces

Nous énumérons ci-dessous quelques trucs vous permettant d'éviter des problèmes lors de la mise au point de votre système de gestion de la mémoire virtuelle :

- Un processus P1 qui recopie une page réelle X sur disque perd la main sur l'entrée/sortie disque. Pour éviter qu'un autre processus P2 veuille aussi recopier cette même page X sur disque suite à son propre appel au voleur de page, on a introduit le bit *locked* associé à chaque page en mémoire réelle.

- Soient deux threads *T1* et *T2*. Si *T2* déclenche un défaut de page pendant que *T1* est en train de résoudre ce même défaut de page, il faut que *T2* se bloque pendant la résolution du défaut de page. Pour ce faire, on utilisera le bit *IO* présent dans la table de traduction d'adresses.

- Quand un processus P1 perd la main sur une E/S disque, le processus P2 qui prend la main peut accéder au voleur de page : il faut donc faire attention à ce que P1 retrouve son contexte quand il reprendra la main. Pour cela, utiliser une *variable locale local_i_clock* pour le parcours de la table des pages réelles dans l'algorithme de remplacement de page par chaque processus, tout en maintenant une variable globale *i_clock*.

### 3.1.5 Bonus

On pourra implanter une des fonctionnalités suivantes :

- Ajout d'un TLB en plus de la MMU. Examiner l'impact sur les performances en modifiant les statistiques.

- Table des pages hiérarchiques ou inversées à la place des tables des pages linéaires telles qu'elles sont prévues dans NACHOS

- Segments de mémoire partagés par plusieurs processus

- Un mécanisme de copy-on-write

## 3.2 TP 2 (Nachos) : Introduction de fichiers mappés (4h encadrées)

Dans ce TP, nous vous demandons de mettre en œuvre des fichiers mappés (voir le cours pour une description du principe de fonctionnement). On procédera de la manière suivante :

- Ajout d'un nouvel appel système, nommé *Mmap*. La partie assembleur est déjà écrite, il vous suffira de récupérer l'exception dans le fichier *kernel/exception.cc*.
- Remplissage d'une nouvelle méthode nommée *Mmap* dans la classe *AddrSpace* pour réaliser ce nouvel appel système. La méthode réservera un ensemble de pages consécutives dans l'espace d'adressage du processus (méthode *Alloc* de la classe *AddrSpace*) et y associera les adresses disques dans le fichier que l'on désire mapper (déplacement dans le fichier mappé).

  L'adresse disque utilisée sera un déplacement dans le fichier mappé. La routine de défaut de page lira le contenu du fichier en utilisant la méthode *ReadAt* . La routine de défaut de page devra être modifiée, car elle est prévue au départ pour lire des secteurs dans les fichiers exécutables uniquement. Pour ce faire, on maintiendra une liste de fichiers mappés par processus, contenant pour chaque fichier mappé: la première page mappée, le nombre de pages mappées, le descripteur de fichier correspondant (*OpenFile** ).
- Ecrire un programme de test des fichiers mappés, par exemple un programme de tri des éléments d'un tableau d'entiers, stockés au préalable dans un fichier.
- Gérer le cas de l'éviction de la mémoire d'une page correspondant à un fichier mappé, ainsi que la fin d'un processus ayant mappé un fichier. Dans ces deux cas, il est nécessaire de recopier la page dans le fichier mappé si elle a été modifiée, afin de ne pas perdre les modifications.

Par souci de simplification, on ne recyclera jamais les pages virtuelles qui ont été un jour mappées dans un fichier (on ne réalisera pas de méthode *Munmap*).

## 3.3 TP3 (Linux) - Utilisation des segments de mémoire partagée, sémaphores, files de messages

### 3.3.1 Utilisation des sémaphores et des segments de mémoire partagée

#### Sémaphores

L'utilisation de sémaphores est effectuée à travers 3 fonctions :

- *semget*: cette primitive crée un ensemble de n sémaphores ;
- *semctl*: cette primitive modifie les paramètres des sémaphores (valeur initiale, destruction) ;
- *semop*: cette primitive réalise les fonctions P et V.

- #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/sem.h>
  int semget(key_t key, /* Voir ci-dessous */
  int nsems, /* nsems représente le nombre de sémaphores à allouer */
  int semflg); /* semflg sera positionné à 0660 (droits d'accès) */

  La fonction rend un entier qui est l'identificateur du groupe de sémaphores alloués (*nsems* est le nombre d'éléments de ce groupe). Les sémaphores alloués sont numérotés à partir de 0. L'identificateur rendu par *semget* est nécessaire pour l'appel des deux autres fonctions. On mettra dans le paramètre *key* la valeur *IPC_PRIVATE* si le groupe de sémaphores est créé avant un appel à fork (sinon, regarder la page de manuel de *semget()*). Le paramètre *sem g* représente les droits d'accès sur le groupe de sémaphores, et sera positionné à *0660*. En cas d'erreur la fonction rend -1.

- int semctl(int semid, int semnum, int cmd, union semun arg);

  Cette fonction permet d'appliquer une action, définie par les paramètre *semid* et *semnum* (groupes de sémaphores et numéro du sémaphore dans le groupe). L'action à effectuer est définie par le paramètre *cmd* (SETVAL : initialisation, IPC_RMID : destruction). *arg* est un argument optionnel à l'action. Lorsque cette fonction est appelée dans l'objectif de détruire des sémaphores, le numéro du sémaphore dans le groupe de sémaphore n'est pas exploité et tous les sémaphores du groupe sont détruits. Lorsque *cmd* est initialisé à *SETVAL*, on doit initialiser le paramètre *arg.val* à la valeur du compteur du sémaphore. A noter que le type union *semun* et la variable de ce type doivent être déclarés par l'utilisateur. En cas d'erreur la fonction rend la valeur -1.

- int semop(int semid, struct sembuf *sops, int nsops);

  Cette fonction permet d'effectuer un P ou un V sur un groupe de sémaphores identifié par *semid*. Le paramètre *sops* est un tableau d'actions, la taille de ce tableau étant définie par le paramètre *nsops*. En cas d'erreur la fonction rend la valeur -1.

  La structure *sembuf* est composée de trois champs :

- short sem_num; /* Numéro du sémaphore concerné dans le groupe*/
- short sem_op; /* Type de l'opération : P : -1 ou V : 1 */
- short sem_flg; /* sem_flg sera initialisé à 0 */

### Segments de mémoire partagée

L'utilisation de segments de mémoire partagée est effectuée à travers 4 fonctions :

- *shmget* : cette primitive crée un segment de mémoire partagée
- *shmctl* : cette primitive modifie les paramètres des segments de mémoire partagée (destruction)
- *shmat* : cette primitive attache un segment de mémoire partagée au processus
- *shmdt* : cette primitive détache un segment de mémoire partagée du processus

- #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/shm.h>
  int shmget(key_t key, int size, int shmflg);

  La fonction crée un segment de mémoire partagée. *key* définit la visibilité du segment de mémoire partagée, et sera fixé à *IPC_PRIVATE* si le segment de mémoire partagée est créé avant un appel à *fork()* (sinon, consulter la page de manuel). *size* définit la taille du segment de mémoire partagée, et *shm_ags* définit les droits d'accès sur ce segment (on utilisera la valeur *0660*). La fonction rend un entier qui est l'identificateur du segment de mémoire partagée. Cet identificateur est nécessaire pour l'appel des trois autres fonctions. En cas d'erreur la fonction rend la valeur -1.

- int shmctl(int shmid, int cmd, struct shmid_ds *buf);

  Effectue une action sur le segment d'identificateur *shmid*, cette action étant définie par *cmd*. Pour détruire un segment, *cmd* doit être initialisé à 0 et *buf* à 0. En cas d'erreur la fonction rend la valeur -1.

- void *shmat(int shmid, const void*shmaddr, int shmflg);

  Cette fonction attache un segment de mémoire partagées dans l'espace d'adressage du processus appelant. Si le paramètre *shmaddr* est 0, le système choisit l'adresse de projection du segment et la retourne en résultat. Le paramètre *shm_g* définit le mode d'attachement du segment et est un ou bit à bit des valeurs suivantes : SHM_RND pour un choix d'adresse du segment par le système, SHM_RDONLY pour un segment en lecture seule. La fonction retourne l'adresse du segment de mémoire partagée ou -1 en cas d'erreur.

- int shmdt(void*shmaddr);

  Détache un segment de mémoire partagée. En cas d'erreur la fonction rend la valeur -1.

**Exercice.** Réaliser un mécanisme de producteur-consommateur par le biais d'un segment de mémoire partagée. La synchronisation entre les deux processus (père et fils) utilisera les sémaphores. Pour mieux vous faire comprendre ce que l'on peut faire et ne pas faire avec des segments de mémoire partagée, le segment de mémoire partagée contiendra une liste chainée de 5 entiers. On pourra utiliser la fonction *nice* pour diminuer la priorité du processus consommateur et ainsi observer les différents entrelacements possibles entre productions et consommations.

**Attention.** Les outils sémaphore (et par la suite segments de mémoire partagée et files de messages) ne sont pas détruits automatiquement lors de la terminaison du processus qui les a créés. Par conséquent, en cas de terminaison anormale d'un processus à cause d'une erreur de programmation, les sémaphores que vous auriez du détruire à la fin de l'exécution du processus sont conservés. La commande *ipcs* permet de visualiser les sémaphores, segments de mémoire partagée et files de messages qui restent actifs. La commande *ipcrm* permet de les détruire. Le nombre d'outils de communications par machine étant limité, vérifiez avant de vous déconnecter (et de temps en temps en cours de session) que vous les avez bien tous détruits.

### 3.3.2 Utilisation de files de messages

L'utilisation de files de messages est effectuée à travers quatre fonctions :

- *msgget* : cette primitive crée une file de messages ;
- *msgctl* : cette primitive modifie les paramètres associés à une file de messages (destruction) ;
- *msgsnd* : cette primitive met un message dans une file de messages ;
- *msgrcv* : cette primitive retire un message d'une file de messages.

- #include <sys/types.h>
  #include <sys/ipc.h>
  #include <sys/msg.h>
  int msgget(key_t key, int msgflg);

  La fonction rend un entier qui est l'identificateur de la file de messages créée. Les paramètres sont similaires à la création de segments de mémoire partagée. En cas d'erreur la fonction rend la valeur -1.

- int msgctl(int msqid, int cmd, struct msqid_ds *buf);

  Pour détruire une file de message, mettre *cmd* à IPC_RMID et *buf* à 0. En cas d'erreur la fonction rend la valeur -1.

- int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

  Envoi d'un message situé à l'adresse msgp, de taille *msgsiz* en octets, sur la file *msqid*. Le paramètre *ms fg* sera positionné à 0. En cas d'erreur la fonction rend la valeur -1.

- int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);

  Réception d'un message sur la file *msqid*. Les paramètres *msgp* et *msgsz* définissent l'adresse et la taille en octets de la zone dans laquelle le message sera recopié. Le paramètres *msgtyp* sera initialisé à 0 (tous les types de messages seront acceptés). Le paramètre *msg g* sera initialisé à 0. En cas d'erreur la fonction rend la valeur -1.

Dans les routines *msgsnd* et *msgrcv*, le tampon utilisateur *msgp* doit débuter par un entier long identifiant le type du message (valeur non nulle) et se poursuivre par le corps du message. L'allocation du tampon est à la charge de l'utilisateur de ces deux routines.

On utilisera la fonction *strcpy* pour recopier les chaînes de caractères.

**Exercice.** Utiliser les files de message pour faire communiquer deux processus cycliques: un processus producteur, qui envoie une suite de messages, et un processus consommateur, qui reçoit et affiche le contenu des messages. Les fichiers source seront nommés *emetteur.c* et *recepteur.c*.

## 3.4 TP4 (Linux) - Fonctions Unix mprotect, mmap, munmap

L'objectif de ce TP est d'utiliser trois fonctions de gestion de la mémoire, les fonctions *mmap*, *munmap* et *mprotect* (par curiosité, on pourra regarder également les routines *mlock*, *munlock*, *memcntl*).

### 3.4.1 Fonction mprotect

La fonction *mprotect* sert à changer les attributs de protection d'une zone de l'espace d'adressage virtuel d'un processus, cette zone devant être cadrée sur un multiple de pages. La taille d'une page pourra être obtenue par la routine *getpagesize*.

```
#include <sys/mman.h>
int mprotect(void *addr, size_t len, int prot);
```

La fonction *mprotect* change les attributs de protection de la zone *[addr, addr + len)*, en alignant si nécessaire la fin de la zone sur une frontière de page. prot est la nouvelle protection sur les pages de la zone, et est construit comme un ou bit-à-bit entre les valeurs suivantes :

- PROT_READ : droits en lecture ;
- PROT_WRITE : droits en écriture ;
- PROT_EXEC : droits en exécution ;
- PROT_NONE : aucun droit d'accès.

La fonction retourne 0 en cas de succès et -1 en cas d'erreur, le code d'erreur étant indiqué dans la variable *errno*.

### 3.4.2 Fonctions mmap et munmap

La fonction *mmap* permet d'établir une correspondance entre un fichier et une région de l'espace d'adressage d'un processus (fichier mappé). Une fois mappé, le fichier n'est plus accédé par des fonctions d'entrée/sortie (*read*, *write*) mais directement par des accès mémoire.

```
#include <sys/mman.h>
void *mmap(void *addr, size_t len, int prot, int flags, int fildes, off_t off);
```

- *addr* : adresse à laquelle on veut établir la correspondance (0 si on laisse le système choisir)
- *len* : taille (en octets) de la région d'espace d'adressage concernée
- *prot* : détermine les accès permis sur la région (lecture, écriture, exécution). Ce paramètre doit être soit PROT_NONE ou un ou bit à bit des valeurs PROT_READ, PROT_WRITE ou PROT_EXEC. prot doit être en accord avec les droits autorisés sur le segment mémoire
- *ags* : options pour établir la correspondance. Ce paramètre sera positionné à MAP_SHARED pour signifier que les écritures devront être reportées dans le fichier.
- *ledes* : descripteur de fichier concerné (obtenu par open)
- *o* : offset (en octets) concerné par la routine mmap (la mise en correspondance concerne len octets à partir de l'offset *o* dans le fichier). Doit être aligné sur une frontière de page.
- valeur de retour : adresse à laquelle la correspondance est établie en cas de succès, -1 en cas d'erreur

#include <sys/mman.h>
int munmap(void addr, size_t len);

- supprime la correspondance qui existait à l'adresse virtuelle *addr* sur len octets. *addr* doit correspondre à une adresse ayant été retournée au préalable par *mmap*.
- la fonction retourne 0 en cas de succès, -1 en cas d'erreur

### 3.4.3 Exercice

On considère un programme P qui utilise une structure de données (par exemple une grosse matrice M) qui sera stockée dans un fichier. Par exemple, le programme pourra afficher dix valeurs aléatoires dans le fichier.

Utiliser la routine *mprotect* pour compter et afficher à l'écran le pourcentage des pages de M accédées par le programme (par rapport au nombre total de pages de M). On nommera le fichier source *matrice.c*.

Le principe retenu sera le suivant. On travaillera directement sur M en mémoire, en utilisant la fonction *mmap*. Pour simplifier, la taille de M sera directement un multiple de la taille d'une page (voir la fonction *getpagesize*).

On protégera les pages de M contre tout type d'accès. Le signal *SIGSEGV* sera détourné pour détecter les violations de protection et comptabiliser le pourcentage de pages accédées. Pour détourner le signal, on utilisera la fonction *sigaction* plutôt que la fonction *signal*, car *sigaction* permet de récupérer non seulement le numéro de signal mais aussi des informations sur chaque type de signal (par exemple dans le cas d'un *SIGSEGV* l'adresse en cause).

#include <signal.h>
int sigaction(int sig, const struct sigaction *act,struct sigaction *oact);

- sig : numéro de signal
- act : adresse d'une structure définissant l'action à connecter au signal
- ocat : adresse d'une structure permettant de récupérer l'ancienne action

Les champs suivants des deux structures devront être initialisés :

- void (*sa_sigaction)(int, siginfo_t *, void *); : adresse de l'action associée au signal
- int sa_flags; type de connexion : ici uniquement *SA_SIGINFO*, indiquant que la fonction de traitement du signal recevra non seulement le numéro de signal sig, mais aussi deux paramètres supplémentaires, dont *sip*, de type *siginfo_t*. sip->si_addr dans le cas d'un signal *SIGSEGV* contient l'adresse en cause.

# Appendix A

# Short notes on the usage of gdb

The GNU Debugger (*gdb*) allows you to debug programs (finding where and why a program has crashed, run your program line-by-line, instruction-by-instruction, stopping the program at a given line in the source code, printing the value of variables, etc.). Except for very simple programs, *gdb* (or any other debugger) should be preferred to calls to *printf*. In particular, when your program has crashed, *gdb* will give you the exact location where the crash occurred!

For the ones addicted to graphical interfaces but who find that *Eclipse* is a bit heavyweight, *ddd* will provide you a graphical interface to *gdb*.

## A.1 Compiling with debug options

`C` or `C++` files have to be compiled with option `-g` (or `-ggdb`) such that *gdb* has all required information to debug your program (symbol table, code line information, etc):

```
shell> cat foo.c
#include <stdlib.h>

void set_to_zero(int * tab, int len)
{
  int i;
  for (i = 0 ; i < len ; i++)
        tab[i] = 0;
}

int main()
{
        int size = 5678;
        int * my_array = (int*) malloc(size);
        set_to_zero(my_array, size);
        return 0;
}
shell> gcc -g -c foo.c
shell> gcc -o foo foo.o
shell> ./foo
bash: segmentation fault (core dumped)  ./foo
```

By default, NACHOS is compiled with option `-g`, no need for you to modify the makefiles.

The last message is not an insult! It's a priceless help the system provides you: when an application crashes, it create a file, named *core* that contains an image of the memory of the application just before the crash. Thanks to this image, *gdb* will allow you to inspect the contents of variables (see section A.2.3).

## A.2 Using gdb

### A.2.1 Starting gdn

To start *gdb*, simply type `gdb`, the *gdb* prompt will appear, showing that gdb is waiting for tour commands:

```
shell> gdb
GNU gdb (GDB) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
(gdb)
```

The following sections present the main commands of *gdb*. An on-line help is available at any time, by typing command *help* or *help command* with *command* one of the commands.

### A.2.2 Loading a program and displaying its source code

To load a program, here `foo`, type `file foo`. To display its source code, type `list`. The debugger will then display the first lines of code. To see the next lines, re-type `list`, or press enter, which is equivalent in gdb to re-executing the last command.

```
(gdb) file foo
Reading symbols from foo...done.
(gdb) list
3       void set_to_zero(int * tab, int len)
4       {
5         int i;
6         for (i = 0 ; i < len ; i++)
7               tab[i] = 0;
8       }
9
10      int main()
11      {
12              int size = 5678;
```

### A.2.3 Identifying the location of a program crash

There are two methods for identifying the location of a program crash:

- By executing the program from gdb;
- By executing gdb giving it the *core* generated when the program crashed.

#### Direct execution from gdb

Command `run` starts the execution of the program that was loaded before. When the program crashes, gdb will display the source code where the fault occurred:

```
(gdb) run
Starting program: /udd/puaut/foo

Program received signal SIGSEGV, Segmentation fault.
0x105a0 in set_to_zero (tab=0x20800, len=5678) at foo.c:7
7               tab[i] = 0;
```

#### Starting gdb from a *core* file

A post-mortem analysis of a crashed program can be done by feeding gdb with the code file generated at crash time[1]. To start gdb from the core file, first load the program as explained in section A.2.2, then use the *core* command tp specify the location of the `core` file:

```
(gdb) core /udd/puaut/core
Core was generated by `./foo'.
Program terminated with signal 11, Segmentation Fault.
Reading symbols from /usr/lib/libc.so.1...done.
Reading symbols from /usr/lib/libdl.so.1...done.
Reading symbols from /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1...done.
#0  0x105a0 in set_to_zero (tab=0x20800, len=5678) at foo.c:7
7               tab[i] = 0;
```

### A.2.4 Breakpoints and step-by-step execution

Programs can be stopped at a given location by putting a *breakpoint* at that location. For example here, we want to debug function `main` and thus put a breakpoint at the start of function, by typing `break main`. The location of breakpoints can also be specified as a specific line in the source code: `break 12`, `break foo.c:main` or `break foo.c:12` in case of multiple files or function names:

```
(gdb) break foo.c:main
Breakpoint 1 at 0x105c4: file foo.c, line 12.
```

Once our first *breakpoint* is put, the program can be started, using command `run`. The program is stopped whenever function `main` is executed:

```
(gdb) run
Starting program: /udd/puaut/foo
Breakpoint 1, main () at foo.c:12
12              int size = 5678;
```

It is then possible to execute instructions one by one, using command `next`. When execution reaches a function call, command `next` considers the function call as a single instruction. If the objective is to enter the called function, command `step` must be used instead of `next`:

[1] If the crash does not generate a core file, type `man limit` or `man ulimit`, depending on the *shell* you are using, to modify system parameter `coredumpsize`.

```
(gdb) next
13              int * my_array = (int*) malloc(size);
(gdb) next
14              set_to_zero(my_array, size);
(gdb) step
set_to_zero (tab=0x20800, len=5678) at foo.c:6
6         for (i = 0 ; i < len ; i++)
```

To continue the execution until the next breakpoint (or program normal or abnormal termination), use command `cont`:

```
(gdb) cont
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x105a0 in set_to_zero (tab=0x20800, len=5678) at foo.c:7
7               tab[i] = 0;
```

In this example, no other *breakpoint* was encountered. The execution has then continued and crashed as before.

### A.2.5 Displaying the execution stack

Command `backtrace` displays the execution stack. For example typing `backtrace` after the crash of our program displays:

```
(gdb) backtrace
#0  0x105a0 in set_to_zero (tab=0x20800, len=5678) at foo.c:7
#1  0x105f0 in main () at foo.c:14
```

From bottom to top, the display indicates that fonction `main()` called function `set_to_zero()`, and that the crash occurred in function `set_to_zero()`. et c'est l'endroit où s'est produit l'erreur. It is then possible to display the values of the different variables (for example, the value of variable `i`, see next section), in addition to the function parameters, which are displayed by default (here `tab` and `len`).

### A.2.6 Displaying the value of a variable

The value of a variable can be printed by using command `print`:

```
(gdb) print i
$1 = 3586
```

This indicates in our example that the crashed occurred when the 3587th element of the array was written.

The values to be printed can be more complex expressions than simple variable names like `i` or `size`: accepted arguments must follow the syntax of the C language:

```
(gdb) print tab[12]
$2 = 0
```

```
(gdb) print tab
$3 = (int *) 0x20800
(gdb) print *tab
$4 = 0
(gdb) print &tab
$5 = (int **) 0xffbee464
(gdb) print &tab[12]
$6 = (int *) 0x20830
```

### A.2.7 Changing the current stack frame

Command `print` allows to print global variables, as well as local variables (variables of the current stack frame), here for example local variables of function `set_to_zero()`). Local variables of other functions of the execution stack can be printed by changing the current stack frame, using command `frame`. The argument of this command is the number of the stack frame, as displayed by command `backtrace`. For example, to print the value of variable `size`, local to function `main()` when the current stack frame is `set_to_zero()`, the following commands are typed:

```
(gdb) backtrace
#0  0x105a0 in set_to_zero (tab=0x20800, len=5678) at foo.c:7
#1  0x105f0 in main () at foo.c:14
(gdb) frame 1
#1  0x105f0 in main () at foo.c:14
14              set_to_zero(my_array, size);
(gdb) print size
$7 = 5678
```

### A.2.8 Interpreting the messages and finding the error

gdb will give you all the hints to find the origin of your programming errors. However, it will be *your* job to find the relevant information to display, to put the breakpoints at interesting locations and deduce why your program does not behave correctly. Unfortunately, gdb is not smart enough for that, you are the smart person.

In our simple example, we notice that array `my_array` was allocated only a 5678 bytes (variable `size` of function `main()`, see section A.2.7). However, for i=3586 (in `set_to_zero()`, see section A.2.6), the index is 14344 (*ie* 3586 * `sizeof(int)`) from the array start, 8666 bytes after the allocated area for the array... Finding the source of the error may be long, correcting it is in general simpler (we let you correct the program as an exercise).

### A.2.9 Quitting gdb

Command `quit` is used to exit from gdb:

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

## A.3 Non exhaustive list of the main gdb commands

**help**: Print list of commands

### A.3.1 Contrôle de l'exécution

**break** [**line—function**]: Set breakpoint at specified line or function.

**delete** [**num**]: Deletes the specified breakpoint.

**run**: Start or re-start debugged program.

**cont**: Continue program being debugged, after signal or breakpoint.

**next**: Step program, proceeding through subroutine calls.

**step**: Step program until it reaches a different source line.

### A.3.2 Visualisation

**print** [**expr**]: Print value of expression expr. Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

**display** [**expr**]: Print value of expression expr each time the program stops.

**backtrace**: Print backtrace of all stack frames

### A.3.3 Misc

**frame** [**index_backtrace**]: Select and print a stack frame.

**pwd**: Print working directory.

**cd** [**dir**]: Set working directory to dir for debugger and program being debugged.

**quit**: Exit gdb.

**file** [**f**]: Use f as program to be debugged.

**list**: List the source code at the current location.

**list** [**f:num**]: Print the code of file `f` at line number `num`.

# Appendix B

# Frequently asked questions (FAQ)

- **Is it possible to install NACHOS on my machine, running Linux ?**
- Yes. If you type *make*, the kernel should compile correctly. However, unless you have installed a *mips* cross-compiler on your machine, you will not be able to compile user programs into *mips* code (but *mips* programs generated on ISTIC machines are executable by NACHOS on your machine). There are two possibilities to compile user programs: copy the cross-compiling chain on your machine (create symbolic links such that the binaries of the cross-compiler are at the same location than on ISTIC machines); (ii) fully install a cross-compiler (see documentation on gcc site, or other tutorials on the Internet).
- **My Linux is a 64-bits machine, will it work?**
- Yes, since the 2013-2014 academic year. It was not working befare, because of parameter passing of pointers between the *mips* architecture (32-bits) and the kernel (64-bits). Parameter passing was modified to remove any problem.
- **Problem installing NACHOS on your machine?**
- The professors will only guarantee you a correct installation on the ISTIC machine. Very limited support (if any) will be provided to help you installing NACHOS on your machine. However, if you can provide us feedback on your problems/solutions, we will be very happy to share to the other students (on this FAQ, the source code/makefile, etc.)
- **Does NACHOS execute on other platform than Linux?**
- There is currently no support for Cygwin. NACHOS was tested on macOS 10.8 and 10.9. We noted some problems on older versions of macOS not implementing *makecontext/swapcontext* (these functions are used to implement context switches between NACHOS threads). To execute NACHOS on macOS, add -D_XOPEN_SOURCE to HOST_CPPFLAGS in file Makefile.config.
- **What is mandatory to know from C++ to complete the assignments?**
- If you are a C programmer, not much. The only features of C++ that are used are classes (no inheritance, templates, stl, etc). The interface of classes is given to you in

include files (.h), the code to be written will be only in .cc files. If you do not know at all the C language, you urgently have to learn.

- **I've successfully installed NACHOS on my personal computer. Am I authorized to skip the labs?**
- No, attendance is mandatory. If you skip 2 lab slots or more, your mark will be zero.

# Index