Operating System

Checklist

May 27, 2010

Adrien Forest 870604-P152
fjxokt@gmail.com

Matthieu Maury 860928-P210
mayeu.tik@gmail.com

Christophe Carasco 870606-P812
carasco.christophe@gmail.com

Written with LaTeX

Contents

1 Requirement set A

1.1 Process handling

1.1.1 PH1: Process states

Processes should be possible to block.	√
A blocked process should not be able to run until it is unblocked.	
It should be possible to delay a process for a specified amount of time.	
When a process is delayed, it should not be able to execute any code.	

> To solve these requirements, we have a **state** attribute in each process control block (pcb). When a process enters in blocking (respectively delayed) state, its state becomes **blocked** (resp. **delayed**), it is put in the **pwaiting** list. The scheduler runs only processes that are in the **pready** and **prunning** list.

1.1.2 PH2: Process priorities

Each process should have a priority.	√
The number of priority levels should be at least 30.	√
Priorities should be specified at process start.	√
Priorities should be changeable at run-time.	√

> To solve these requirements, we have a **priority** attribute in each process control block (pcb) and we defined a range of priorities. At the process start, the priority is required. The shell, however creates processes with a base priority. From the shell, it is also possible to change the priority with the *chg_prio* program).

1.1.3 PH3: Process scheduling

A higher-priority process should be scheduled in preference to a lower-priority process.	
Between processes of the same priority, round-robin scheduling should be used.	
Higher-priority processes should be able to preempt lower-priority processes.	
There should be a recurring regular check for which process to run, unrelated to whether a particular task has finished its work.	
A lower priority process should not be allowed to run if there is a higher priority process ready to run. We expect strict highest-priority-first scheduling.	

1.1.4 PH4: Process identities

Each process should have a unique ID that can be used to address the process and perform operations on it.	√
A process may have a human-readable name.	√

> The process identifier **pid** is stored in the pcb of each process as well as the **name** (which matches with the program name).

1.1.5 PH5: Process Creation

Processes should be able to create new processes.	√
Processes are created by a spawn operation that starts a new process with a given code to run (see requirement PH7).	√
The spawn operation should return the ID of the created process.	√
If the process spawning fails for some reason, an error should be reported.	√

> The kernel creates a process init at the launch and this latter creates two processes (*Text scroller* and *Shell*). From the shell, the user will be able to lauch new programs (by creating new processes with the *fourchette* function). The *fourchette* function returns the **pid** of the newly created process or an **error code** in case of failure (the errors are listed in errno.h file).

1.1.6 PH6: Process Termination

Processes should be able to terminate themselves.	√
Processes should be able to terminate other processes.	√
The operating system should be able to terminate processes.	√

> To terminate themselves, processes can use the *exit(int n)* instruction. The value n is transmitted to the supervisor to make it know why it ended (normal termination or error ?). There is also a function *kill* that terminates a process identified by its **pid** (using a syscall). Obviously, the operating system can kill processes as well (as the *kill* function makes a syscall to an (operating system) function that kills the process.

1.1.7 PH7: Programs and Processes

Each process runs the code of a particular program.	√
The program to execute in a process is specified when the process starts.	√
Each program should have an ID that is used to identify it when starting a process.	√
Each program should have a human-readable name associated with it. This might be the same as the ID.	√
There should be a global table of available programs.	√
It should be possible to start every program in multiple simultaneous instances.	√

> The user has to specify a program name when he calls the *fourchette* function otherwise the process will not be created and an error will be returned. As a program name is unique, the name of the program acts as its identifier. The kernel owns a program list to check that the user launch an existing program.

1.1.8 PH8: Process Information

A process should be able to obtain information about another process.	√
The information should include: Process priority Process name, if such is used Name of the program running in the process Current scheduling state of the process	√
This information should be readable by the process requesting it.	√
The information should not be printed directly to the console; if it is to be presented to the user, a program (or function in the shell program) will have to be written for that purpose.	√

> The processes can request information about another process by using the *get_proc_info* function in the user programs. The information is returned as a structure which contains a copy of some information about the process (pid, name, priority, supervised processes, supervisor process). Moreover, there is *proc_info* user program that can be launched from the shell that prints out the received information.

1.1.9 PH9: Resource Recycling

When a process is terminated, all resources that it has allocated should be reclaimed.	√
Process IDs should be recycled.	√
The operating system should never leak memory, if there are dynamically allocated or assigned resources.	√

The processes are stored as a list. When a process is terminated its space is freed so that a future process can be set at this location. Process ids are recycled as well (**pid** are in range $0, \ldots, 2^{32-1}$ and when the pid reaches the maximum value, it goes back to the first free pid from 0 – if there is any).

1.1.10 PH10: Process Limitations

There can be a fixed upper limit on the number of processes allowed in the system.	√
Process IDs should be reused, as per requirement PH9.	√
Any fixed upper limits must be easy to modify, using a define or similar C construct.	√

Indeed, we use a maximum number of processes which is defined using the define instruction.

1.2 Input and Output

1.2.1 IO1: Text Consoles

The system should provide at least one text console for input and output.	√

1.2.2 IO2: Interrupt-Driven IO

Text input and output shall be implemented using interrupt-driven IO on the serial port of the system.	√

1.2.3 IO3: Text Output

It should be possible for processes to send text to a console.	√
It should be possible to send coherent messages, i.e. a process should be able to send a text message that is printed uninterrupted on a text console.	

1.2.4 IO4: Text Input

It should be possible for a process to request the user to input text.	
Text input should be returned as a string to the requesting process	

1.2.5 IO5: Malta LCD Output

It should be possible for a process to output text to the Malta LCD display	√

The API provides a function *fprint* to output text to the Malta display. It is also possible to launch the *malta* program that prints the string given (as a unique field – no space characters).

1.3 System Application Programming Interface (API)

1.3.1 A1: General

The API should be the only way to request services from the operating system kernel.	√

1.3.2 A2: Syscall

The MIPS Syscall exception mechanism should be used to implemented the API.	√

1.3.3 A3: Documentation

The API should be presented to the user programs as a set of C functions.	√
The API should be documented for the benefit of user programs.	√

1.4 Other

1.4.1 O1: No Shared Memory

Processes should never use shared variables or memory to communicate.	√
It is not necessary for the operating system to rigourously enforce this condition.	√

1.4.2 O2: Reentrancy

It should be possible to execute multiple instances of the same program concurrently.	√
A program cannot use "static" variables in C or global variables.	√
Any attempts at using names to identify processes has to allow for the dynamic creation of names.	??

> As a program is only an association of a *name* and the entry point of the program, processes can run the same code if they call the *fourchette* function with the same program name.

1.4.3 O3: Error Handling

Whenever the operating system cannot complete an operation, it should return an error code to the process trying to perform the operation.	√
Error codes should be returned to the calling process as the function result, so that it has the ability to deal with the error.	√
To help debugging programs, the OS may provide the facility to print diagnostic errors to the console automatically.	

> When a process cannot complete an operation it returns an *error code* to its supervisor to let it handle the error. The OS provides a function perror that allows the program to print a message with the error code.

1.5 User programs

1.5.1 UP1: Increment

A user level program that prints out an increasing number sequence 1, 2, 3, ..., N to the console.	√
Each number should be printed on a separate line.	√

1.5.2 UP2: Fibonacci

A user level program that prints out the Fibonacci number serie: 1, 1, 2, 3, 5, 8, 11, ... Max-Fib-Number.	√
Each number should be printed on a separate line.	√

1.5.3 UP3: Command Shell

The system should have a command shell that can do at least the following:	√
Start processes. For example, it shall be possible to start another interpreter.	√
Change priority of processes.	√
Obtain information about present processes.	√
Terminate processes.	√
Ouput to the Malta LCD display	√
It should be possible to have more than one shell running simultaneously.	√
The command shell should not be special-treated by the operating system in any way.	√
The command shell has to handle "backspace" functionality when entering commands.	√
The command shell should not use CPU time when waiting for input.	√

1.5.4 UP4: Text Scroller

There should be a scroller background process that scrolls text on the Malta board LCD display.	
The text can be varied or fixed, depending on the level of ambition.	
The process should provide for smooth scrolling even on a highly loaded system.	
The scroller should be a regular user process with high priority.	
The scroller should sleep between updates to the display.	
The scroller should start when the operating system starts.	

2 Requirement set B

2.1 Message Passing

2.1.1 MP1: Messages

Messages are unicast, going from one process to precisely one other process.	√
The messages should be able to contain some user data.	√
Messages should be tagged with sender and receiver process identities.	√
Messages should have a type or priority field for filtering by the receiver.	√
It should be possible to pass process identities using messages.	√

2.1.2 MP2: Message Queues

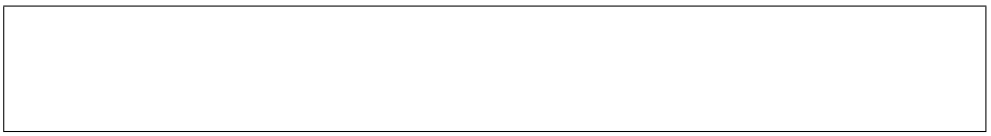
Each process should have a single message queue for incoming messages.	√
The message queue can have a fixed upper bound on the number of messages stored.	√

2.1.3 MP3: Sending Messages

Processes should be able to send messages to other processes.	√
The send should not depend on the state of the other process.	√
The send operation should report an error if the message queue of the receiving process is full.	√
The send operation should never block the sending process.	√

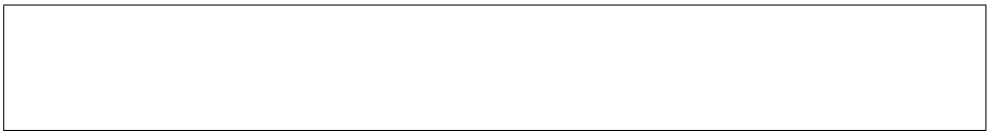
2.1.4 MP4: Receiving Messages

Processes should be able to receive messages from their message queue.	√
The receive operation should be able to filter messages based on the type or priority of the message.	√
The user-defined data specified in the message should be readable by the receiver.	√
The receive operation should block the receiving process if the message queue is empty.	
It should be possible to specify a time-out for a receive operation, after which a process will resume operation even if no message arrives, and report an error.	



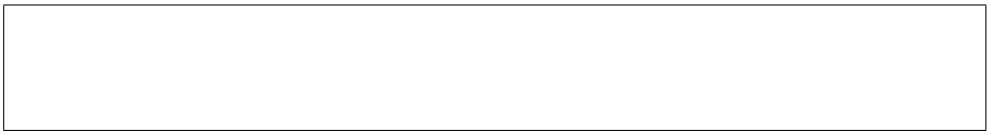
2.1.5 UP5: Ring

A program should start a set of other processes, P1 to Pn.	√
The processes should be set up in a communications ring, where P1 sends messages to P2, etc. on to Pn.	√
The demonstration should send some messages around the ring and show that they visit all processes along the way.	√
The user should only need to start a single process to start the ring demo: the main program should start the other processes involved.	√
The main ring program should return to the shell when done setting up the ring.	√
Timers for when to send messages should be set in such a way that it is possible to observe the processes involved in the ring from the shell.	√



2.1.6 UP6: Dining Philosophers

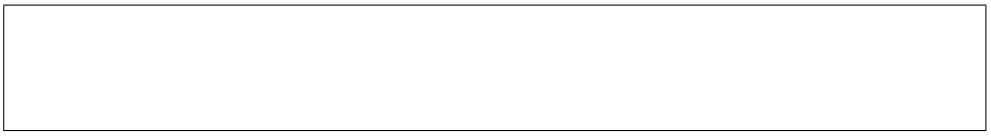
Each philosopher is a process of its own.	
Algorithm	
Like UP3, the entire setup should be started by a single process.	
Like UP3, the main program should return and make it possible to observe the processes running.	



2.2 Process Supervision

2.2.1 PS1: Provide Process Supervision

Processes can be appointed as supervisors of one or more other processes.	√
When a supervised process terminates, the supervisor is notified.	
It should be possible to differentiate between controlled and uncontrolled termination, i.e. it should be possible for the supervisor to see if a subordinate process has crashed or if it terminated in good order.	
To provoke crashes, the semantics of the send operation is changed so that a process terminates if the receiving process has a full message queue.	



2.2.2 UP7: Process Supervision

A demo application and application note should describe the mechanism and demonstrate that you have a working implementation of process supervision.	
The demo should include a supervisor that restarts its subordinates if they crash.