Adrien Forest
fjxokt@gmail.com
Christophe Carasco
carasco.christophe@gmail.com
Matthieu Maury
mayeu.tik@gmail.com

UPPSALA
UNIVERSITET
Department of Information Technology
Master Student

Operating System Project 10hp

System Description

March 26, 2010

University year 2009-2010

Written with LaTeX

Contents

1. Requirement Set A

1.1 Process Handling

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.

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.

PH3: Process Scheduling Scheduling should be highest-priority first, with preemption. In particular:

- 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.

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.

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 p.5).
- The spawn operation should return the ID of the created process.
- If the process spawning fails for some reason, an error should be reported.

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.

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.

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.

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.

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 p.**??**.
- Any fixed upper limits must be easy to modify, using a #define o or similar C construct.

1.2 Input and Output

IO1: Text Consoles

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

IO2: Interrupt-Driven IO

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

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.

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.

IO5: Malta LCD Output

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

1.3 System Application Programming Interface (API)

A1: General

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

A2: Syscall

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

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

O1: No Shared Memory

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

O2: Reentrancy It should be possible to execute multiple instances of the same program concurrently. In particular, this means that:

- 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.

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.

1.5 User Programs

The system would be no fun without a set of programs to run on it. For demonstration purposes, you should implement a set of simple user programs.

UP1: Increment number sequence 1, 2, 3, ..., N to the console. Each number should be printed on a separate line.

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.

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.

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

Processes in the operating system should communicate using asynchronous message passing.

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.

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.

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.

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.

UP5: Ring A user level program that demonstrates process communication.

- 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.

UP6: Dining Philosophers This demonstrates process synchronization (you will need to work out how to synchronize processes given the message passing paradigm). The dining philosophers program is a program solving a problem with resource sharing as follows. A group of philosophers are sitting around a dining table (formed as a circle). Between each pair of philosophers is one fork. Occasionally, one philosopher wants to eat and to do that he needs two forks. This means that two philosophers sitting next to each other can not eat at the same time. You should implement this with the following in mind:

- Each philosopher is a process of its own with a behavior that is similar to the following:

```
think(random())
get one fork
get other fork
eat(random())
drop forks
restart
```

- 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

So far, process lives on its own. Nobody else cares if a process terminates. In most real real-time operating systems, processes can be linked together in order to provide fault detection and tolerance.

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.

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.

3. Requirement Set C

We want to try to implement the Dynamic Memory Management as an optional part of the project.

EC3: Dynamic Memory Management (5 pts) In the basic specification, it is allowed to have a fixed limit on the number of processes in the system, as well as fixed-length message queues. It is also not at all necessary to provide dynamic memory allocation as a service to user processes. In this extra credit requirement, you should remove these limits. In particular:

- There should be a kernel module responsible for dynamically allocating memory for various purposes.
- There should no longer be a compile-time fixed upper bound on the number of processes in the system, process control blocks should be allocated dynamically.
- Message queues should be dynamic in size, allocating space only as needed.
- Out-of-memory situations have to be gracefully handled.
- A demonstration application should be written that demonstrates creation of processes and/or messages until memory runs out.
- A demonstration application should be written that demonstrates the use of dynamic memory allocation in a process.