Operating Systems
Lab Report Session 2

Lars Holdijk (s2534878)
Erik Voogd (s2393220)

March 9, 2018

1 Shell

1.1 Problem Description

For the first assigment we are asked to develop our own simple command shell. Besides being able to run programs found in the users path it should support the following features:

- I/O redirection.
- Run processes in the backgound
- Pipe process output

1.2 Problem Analysis

We use flex to convert a string of characters into a stream of tokens. When reading a '<', we redirect the following word to standard input. The word following a '>' symbol should be a file descriptor where standard output is redirected to. When a character '&' is added in the end, the parent process should not wait for the child to finish, but continue reading the next stream of tokens instead. The token '|' should execute what came before, and its output should be forwarded as input to the command that comes after.

1.3 Implementation

One stream of tokens should be executed as soon as a new line token is scanned, or a pipe symbol '|' is encountered. The command "exit" will exit the minishell.

1.4 Results

Input and output redirection work. Background processes will run, but it is not guaranteed yet that they will be killed when exiting. One pipe will work, but the shell will not continue.

2 Shared Memory

This section contains our not so brief report of the shared memory exercise. We tried to keep the report short but this was due to the complexity of the assignment not possible.

2.1 Problem Description

This assignment requires us to make a program with three processes, a parent process and two child processes. The two child processes are responsible for printing ping and pong on standard output, one child process will print ping while the other should print pong.

The children print ping and pong in turn. For this reason they are only allowed to print either ping or pong when a variable is set to the value corresponding to a value assigned to their process. For the child process printing ping this value should be 0, and for the other one it should be 1.

The child processes will be created by the parent process using the system call fork() and will thus only share their text segment but not their data segment. For this reason it will not be possible to modify each others data and by doing so signal the other process to print. However, we can simulate shared memory by the aid of virtual memory.

The goal of this assignment is therefore to simulate shared memory using virtual memory concepts. In doing so the parent process should function as the memory management process. As we are trying to simulate shared memory the child processes should not be aware of the memory being virtual, what this means is that the code for the child process should not need to be modified.

Besides not being able to alter the child process' code, the following restrictions are placed on our implementation of the shared virtual memory:

- The system call mprotect should be used to simulate a page fault during which the memory synchronization occurs.
- As the ping pong process code contains busy waiting access of the shared memory should not always result in communication with the memory management process.
- The shared virtual memory implementation should function in the same way a super computer handles its required memory not being on the current machine, therefore an implementation tailored specific for only two processes is not sufficient.
- Only the SIGUSR1 and SIGUSR2 signals can be re-purposed for communication between the child processes and the parent process.

2.2 Problem Analysis

As the restrictions for this assignment require us to provide a general solution for the shared memory problem we will try to refrain from talking about about a specific number of child process in our problem analysis.

We will start our problem analysis by investigating the three states each process could possibly be in and how transitions can be triggered. From this analysis we will construct a state diagram illustrating the possible states of each process. Additionally we will construct a table indicating the actions that will need to be performed for each state transition.

2.2.1 States

Each child process can be in one of three states, *synchronized*, *outdated*, or *leading*. In each state exactly four different transitions can occur, the process itself either writes or reads or another process writes or reads. Below you will find an analysis of each possible state. In figure 1 you will find a state diagram illustrating the states and transitions between them and in table 1 you will find the actions that we will need to support based on the the analysis.

Synchronized
All processes start initially start in the synchronized state. In this state the child knows that its memory is the same as the memory of all other processes, therefore the process is free to read the memory. When it changes the memory it should however notify all other processes that their memory is no longer synchronized. When this happens the state of the process changes to leading while all other processes become outdated.

Leading
Only one process can be in this state at the same time. As the leading process is the currently the only one with the correct memory it is free to read and write to it. The leading process can only transition to a different state when another process tries reads or writes to its memory.

Outdated
In the outdated state it is not safe for a process to read or write to its own memory. Therefore before either reading or writing to its memory it should retrieve the most up to date memory from the process in the leading state. After reading by the outdated process it is synchronized with the leading process. However, in the case of writing the previously leading child becomes outdated while the child that wrote to the memory becomes leading.

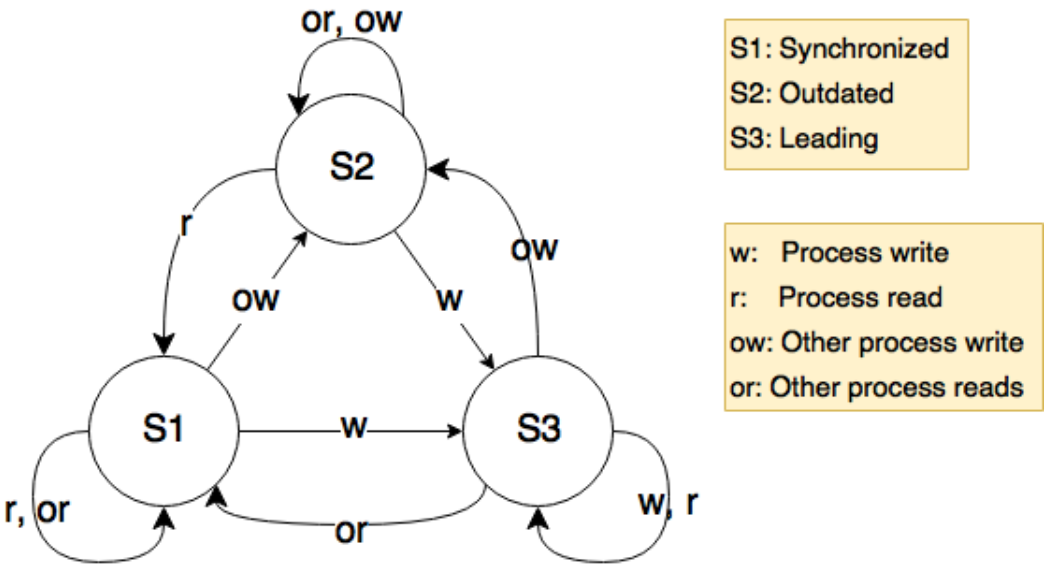

Figure 1: State diagram

Table 1: State Transitions

	Synchronized	Leading	Outdated
Self read	-	-	Retrieve most up to date memory - Read memory - **Synchronized**
Self write	Write to memory - Notify other process of memory update. - **Uptodate**	-	Retrieve most up to date memory - Write to memory - Notify other process of memory update - **Uptodate**
Other read	-	Provide memory - **Leading**	-
Other write	Be notified of memory update - **Outdated**	Provide memory - Be notified about memory update - **Oudated**	-

2.3 Implementation

In this section we will discuss how we implemented the solution to the problem described above. First we will discuss how we represent the state of each process. In the second section we will discuss how we use the memory management process(MMP) to perform all actions related to state transition. The third section describes how we use pipes for synchronizing the virtually shared memory. In the fourth section we will discuss how our implementation differs from the theory. Finally we will look at two more pitfalls is section five.

2.3.1 Process state

We considered to possible implementations for our process states. We first considered keeping track of the states of each process in the MMP. Doing this would require to communicate with the MMP for every read and write by a process.As this was not allowed by the restrictions set we decided to go with another approach.

In our final version the state of each process is stored within the process itself using an state enumeration. This also allows us to directly relate the state to the protection level of the process memory. We used the following mprotect protection levels for each process state:
- **Synchronized** - PROT_READ
- **Leading** - PROT_READ | PROT_WRITE
- **Outdated** - PROT_NONE

2.3.2 State transitions

Due to encoding the process state by protecting the memory all state transitions are automatically triggered by either a read or write on the memory. Therefore we can initiate all state transitions and memory management in our SIGSEGV signal handler.

Within the SIGSEGV signal handler of a process we will need to perform the actions described in table 1. For this we will use the SIGUSR1 and SIGUSR2 signals. Using SIGUSR signals for this has the additional benefit that these signals are immediately handled by the receiving process, solving most of our timing issues.

2.3.3 Synchronization

When a process in the outdated state tries to read from or write to its memory we need to update it first. As required by the problem description we will be updating the memory of the processes page wide.

Initially we want to do this by always keeping an updated version of the page at the MMP. However, due to the SIGSEGV signal handling occuring before the memory was actually updated we cant sent the new page together with the signal that indicates and update of the memory.

For the reason given above we decided to only retrieve the most up to date

list from the leading process upon request from an outdated process. This requires us to use two pipes, one for retrieving the most uptodate page from the leading process and one for sending it to the outdated process.

In contrast to the state transitions described in the problem analysis the leading process changes its state to synchronized after providing the MMP with the current page. The reason for this will be further described in the section about process termination. From this point on the MMP will function as the leading process.

2.3.4 Process termination

Contrary to our theoretical exploration of the states in the problem analysis the processes in out system won't run indefinitely. For this reason we have implemented some safe guards for terminating processes.

First of all we keep track of a list of running processes in the MMP. Only these processes will receive a signal when the shared memory has been updated. Once a child terminates it is removed from this list. This is done using a SIGCHLD handler.

Before explaining the second part of the SIGCHLD handler we need to look at what happens when a leading process tries to stop. If we would allow this it means that we would lose the most up to date version of our memory. Therefore we decided to implement a closing method that is executed at exit of a process. Within this method we implement a fail safe that keeps all leading processes running until it loses its leading state.

As this would result in the possibility of always having one leading process running we made the decision to make the MMP the leading process when there is only one process running this is the function of the second part of the SIGCHLD handler.

2.3.5 Other pitfalls and issues

Sleep
During the creation of our ping pong processes we use the sleep function. Without this sleep function we would risk the first process already altering its memory before the second processes is created.

Despite our reservations for using a sleep function we decided that this would be the best option for our program. Preferably we would have used a signal to signal the processes that they can start their ping pong procedure. This would however result is more complexity that could distract from the real purpose of this exercise.

SIGSEGV == 10
For some odd reason memory protected by mprotect does not trigger a segmentation fault when being read from or written to. Instead it throws signal SIGBUS. For this reason we redirected both signals to our handler.