School of Computer Science

COMP30640

Lab 7
Inter Process Communitation (IPC): Named Pipes

Teaching Assistant:	Thomas Laurent
Coordinator:	Anthony Ventresque
Date:	Friday 26th October, 2018
Total Number of Pages:	6

Like un-named/anonymous pipes, named pipes provide a form of IPC (Inter-Process Communication). With anonymous pipes, there's one reader and one writer, but that's not required with named pipes, any number of readers and writers may use the pipe.

Named pipes are visible in the filesystem and can be read and written just as other files are:

```
$> ls -l test_pipe
prw-rw-r-- 1 aventresque aventresque 0 Oct 26 15:38 test_pipe
```

You can create named pipes using the following command:

```
$> mkfifo test_pipe
```

which in this case would create a named pipe called `test_pipe`.

1 Playing with your First Named Pipes.

Create a named pipe called `pipe_test`.
Open **two terminals** and run the following commands:

```
$> read something < test_pipe; echo $something
```

in one of the terminals and:

```
$> echo "who are you?" > test_pipe
```

in the other terminal. What happens?

2 Named Pipes in Scripts.

Now create the two scripts below, where the writer writes in the pipe something that is given by the user (using `read` from the terminal) while the reader continuously reads from the pipe and displays it in the terminal.

`write_test.sh`:

```
#!/bin/bash
while true; do
        read input
        echo $input > test_pipe
done
```

`read_test.sh`

```
#!/bin/bash
while true; do
        read input < test_pipe
        echo received from the pipe: $input
done
```

Again, use two terminals to test your scripts. You can stop both scripts with control+c

3 Problem.

Now use **one** single script that reads from one pipe and writes on another pipe. The names of those two pipes should be given as arguments to the script. Try launching two processes of this script (with different parameters) and make them talk to each other (process 1 reads from process 2's output pipe and outputs to the pipe that process 2 reads from). Create 2 pipes and start 2 processes of the script, one in each terminal. In a 3rd terminal, send a message to either one of the two pipes. What happens?

Solution

You probably started out with something like this:

```
# read_write.sh
#!/bin/bash

if [ $# -ne 2 ]; then
  echo "Wrong number of arguments" >&2 # &2 is standard error output
  echo "Usage $0 in_pipe out_pipe" >&2 # &2 is standard error output
  exit 1 # the exit code that shows something wrong happened
fi

in_pipe=$1
out_pipe=$2

while true; do
    # Reading from the input pipe
    read input < $in_pipe
    echo I found this in the pipe: $input and I am going to send it on my out pipe
    # Writing to the output pipe
    echo $input > $out_pipe
done
```

But this will cause a problem when we try to run these programs at the same time. Say we write to pipe 1:

```
$> echo message > pipe1
```

- Process 1 takes input from pipe 1 (the input pipe).
- It then writes this input to the pipe 2 (output pipe)
- Process 2 takes input from pipe 2 (the input pipe).
- It then writes this input to the pipe 1 (output pipe)

Its easy to see that we have an infinite feedback loop here. The processes will never stop writing back and forth to each other.

Can you solve this issue by testing whether a message has been received already before (in short do not send back a message that you've received twice in a row)?

Solution

One way we can get around this is by checking the input to see if we have already received the message before. We create a received variable and initialise it (this can just be an arbitrary string).

```
# read_write_fixed.sh
# ! / bin / bash
in_pipe=$1
out_pipe=$2

received="yo"

while true; do
    read input < $in_pipe
    if [ $received != $input ]; then
         echo I found this in the pipe : $input and I am going to send it on my out pipe
         echo $input > $out_pipe
         received=$input
    fi
done
```

- If we havent received the message from pipe 1 before, process 1 tells the user what it has found in pipe 1 and we send it on to pipe 2. It then assigns this value to the *received* variable for the next loop iteration.
- Process 2 then reads in the input from pipe 2 and tells the user what has been found there. The input is then written out to pipe 1 and this value is assigned to the *received* variable.
- At this stage, process 1 gets its input from pipe 1 again, but it is the same as before (i.e. the received variable) so the *if* statement returns false. Since we have seen this value before, we wait for a new input.

4 IPC to Solve Concurrency Issues.

Can you use named pipes to solve the concurrency problem seen in the lab last week? **Hint:** Could we create something to pass between the pipes to tell the process it can run (write.sh)?

Solution Here we are going to pass a token through the pipe. When the program has the token ('n' for 'no', 'y' for 'yes') it can write to the file. We use the shift command to get past the pipe and token arguments. Once we are done writing, we can pass the 'y' token along again to the output pipe.

```
# write_ipc.sh
#!/bin/bash

if [ $# -lt 4 ] ; then
  echo "This script requires at least four parameters" >&2
  echo "$0 pipe_in pipe_out token files " >&2
  exit 1
fi
in_pipe=$1
out_pipe=$2
token=$3
# We shift along to the fourth argument so we can iterate through f1, f2, f3 later.
shift
shift
shift
if [ $token = "n" ]; then
    echo waiting for the token
    # We wait until something is written to the in_pipe
    read input < $in_pipe
    # Now something has been written to in_pipe, so we move on.
    echo got the token
else
    echo had the token at the start
fi
echo \$1 = $1
# Execute critical section
for elem in "$@" ; do
    if [ ! -e "$elem" ] ; then
        echo 1st $$ > $elem
    else
        echo next $$ >> $elem
    fi
done
# Pass our token to the output pipe.
# If the out_pipe here is the input pipe for another script,
# that will now be able to execute its critical section.
echo "y" > $out_pipe
```

To run this, we pass the token (to one of the processes first) and the pipes to the program when we run them. We can do this using the following script:

```
#!/bin/bash
mkfifo pipe1 pipe2
./write_ipc.sh pipe1 pipe2 n f1 f2 f3 & ./write_ipc.sh pipe2 pipe1 y f1 f2 f3
rm pipe1 pipe2
```

If this seems a little bit confusing, take a look at the flow chart in Figure 1 and follow it along with the code. You should then be able to see the token being passed from one process to another.

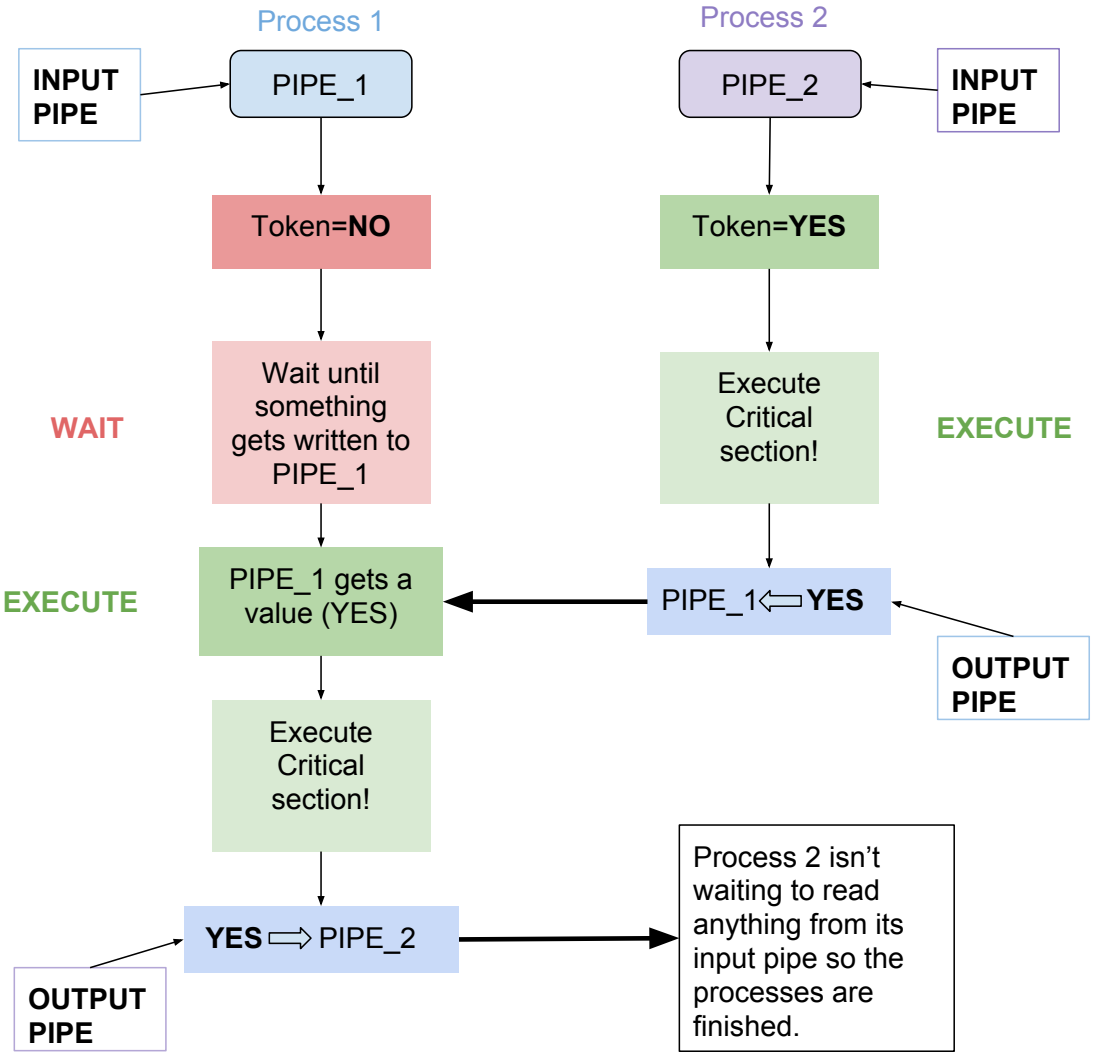

Figure 1: Token passing between two processes.

Using pipes here was not as efficient as the semaphores we used last week. We are now locking the whole for loop instead of each file separately, which is not ideal. This was just an example get you to use named pipes.

You can see that in the script we use to run the two write_ipc.sh processes, we also create and delete our pipes. It is customary to do this, as it prevents anything from getting "stuck" in the pipes and interfering with our next execution.