*Operating Systems* **CSI3131** ***Lab 2a*** *Winter 2018*

# Interprocess Communications with Pipes

**Goal**: To explore IPC with UNIX/Linux pipes.

## Guideline

**We will reuse the files from the lab 1.**

**Step 1:** Start up ubuntu. Copy the provided file *lab2a.tar* your working directory. As in Lab 1, extract the files from this archive using *tar –xvf lab2a.tar*. You will find the same files as in Lab 1, but with an additional file *filter* (the code for this executable is in the *code* directory. **Note**: the code uses gets( ) method which is not supported by ubuntu 16 anymore. So we provide the code for your interest, but you cannot compile it with the up-to-date OS).This is an executable file that is launched without parameters, reads its standard input, assuming it is receiving the output of *procmon* from the last labs and outputs only the lines in which the state of the monitored program changed.

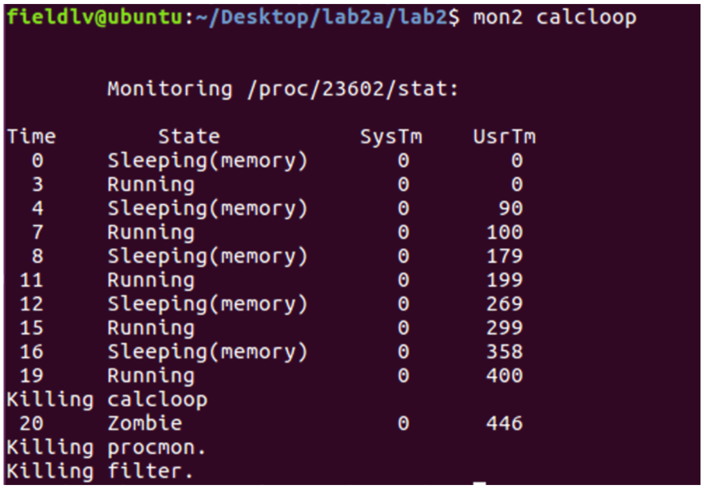
**Step 2:** Enhance the C program mon.c developed in Lab 1 using the following guidelines:

1. Like *mon* from Lab 1, *mon2* uses one command line argument: the name M of the program it has to launch.
2. It will launch the program M (the program M does not take any parameters) and determine the PID of the process running the program.
3. It will launch *procmon* with the argument PID.
4. It will launch another process to run the provided program *filter* with no arguments.
5. The output of *procmon* should be sent to the input of *filter*, i.e. the *filter* will read from its standard input what *procmon* is writing to its (procmon's) standard output.
6. It will sleep for 20s, then it will terminate program M, will sleep 2 more seconds and will (try to) terminate *procmon* and *filter*.
7. A template for *mon2.c* has been provided to help you develop the code. Note that much of the *mon.c* program can be copied into *mon2.c*.

**Step 3:** Compile your C progam by entering gcc mon2.c –o mon2 on the command line.

**Step 4:** Enter mon2 calcloop and observe. This launches the program mon2 you have just created, which in turn launches procmon to monitor calcloop's execution, and filter to output only the lines when the state of calcloop changes.

You should some similar outputs like:



```
fieldlv@ubuntu:~/Desktop/lab2a/lab2$ mon2 calcloop


        Monitoring /proc/23602/stat:

Time        State            SysTm    UsrTm
  0     Sleeping(memory)       0        0
  3     Running                0        0
  4     Sleeping(memory)       0       90
  7     Running                0      100
  8     Sleeping(memory)       0      179
 11     Running                0      199
 12     Sleeping(memory)       0      269
 15     Running                0      299
 16     Sleeping(memory)       0      358
 19     Running                0      400
Killing calcloop
 20     Zombie                 0      446
Killing procmon.
Killing filter.
```

**Step 5.** You can experiment with sending signals to the calcloop process: When you launch mon calcloop:

- Learn the pid of calcloop (for example by entering ps –a | grep calcloop)
- Note: To start a program and detach it from the terminal use &!. E.g. calcloop &!
- Send it a stop signal by entering kill –s SIGSTOP pid, where pid is the pid of calcloop that you learned in the previous step.
- Send calcloop a continue signal by entering kill –s SIGCONT pid

Note that you will have to be reasonably fast, so that you manage to do this within 20 seconds calcloop is running.

**Background you might need:**

- The execution of a C program starts in procedure main() which has two arguments: integer argc containing the number of command line arguments and an argc-element array of strings argv, containing the command line arguments. By convention, argv[0] is the name of the program, argv[1] is its first argument.
- the fork() command creates a new process. Both the parent and the child continue as if they returned from a fork() call, however the parent gets as return value the PID of the child, while the child gets 0. Type man fork to read more about fork().
- The execl(path, arg1, …) command replaces the current process with the specified program launched with the provided arguments. Type man execl to learn about the exact meaning of its arguments.
  The code 'execl(“calcloop”, “calcloop”, NULL)’ will replace the current process with the *calcloop* program. The code ‘execl(“/bin/ls”, “ls”, “-l”, NULL’ will launch ls –l.
- You will need to convert the integer PID into a string to pass to procmon. In C you can do this using the *sprintf* function as in *sprintf(buf,”%d%,pid)* where *pid* is an *int*

variable, and *buf* is a character array (*char buf[20];*) that will receive the character string to be passed to *procmon*.

- Function sleep() will cause the program to sleep for the specified number of seconds.
- Function kill(pid, sig) will send a signal sig to process pid.
  - have a look at signal.h to see the different signals
  - google is your friend, this search result on 'signal.h' might be helpful (there are many more)

  **http://www.opengroup.org/onlinepubs/009695399/basedefs/signal.h.html**
- A file descriptor is an integer number, that serves as a file handle, to identify an open file. The file descriptor is used with library functions and system calls such as *read()* and *write()* to have the OS operate on the corresponding file.
- The function *pipe(int *fd)* takes as an argument a pointer to an integer array (i.e. the name of an integer variable declared as array, i.e. int fd[2] , creates a pipe and sets the fd[0] to be the file descriptor used to access the read end of the pipe and sets fd[1] to be the file descriptor to be the write end of the pipe.
- *dup2(int newfd, int oldfd)* – duplicates the oldfd by the newfd and closes the oldfd. See http://mkssoftware.com/docs/man3/dup2.3.asp for more information. For example, the following program:

```
int main(int argc, char *argv[]) {
        int fd;
        printf("Hello, world!")
        fd = open("outFile.dat", "w");
        if (fd != -1) dup2(fd, 1);
                printf("Hello, world!");
        close(fd);
}
```

will redirect the standard output of the program into the file outFile.dat, i.e. the first "Hello, world!" will go into the console, the second into the file "outFile.dat".

- *read(int fd, char *buff, int bufSize)* – read from the file (or pipe) identified by the file descriptor fd bufSize characters into the buffer buff. Returns the number of bytes read, or -1 if error or 0 if the end of file has been reached (or the write end of the pipe has been closed and all data read).
- *write(int fd, char *buff, int buffSize)* – write into the file/pipe buffSize characters from the buffer buff
- *close(int fd)* – closes an open file descriptor
- perhaps this link might help you a bit with C: http://www.acm.uiuc.edu/webmonkeys/book/c_guide/