

# The Shell, Processes and Basic Inter Process Communication (IPC) with System Calls

EECS 678 Staff

# The Shell

- Shell is a program that lets users communicate with the Operating System
- Typically, in a shell, users are given access to a command line interface (CLI) through which users can input textual commands
- The shell accepts the command, interprets it and executes it
  - The command will be an executable program that either comes with the OS or is created by the user
  - The shell usually looks for the program from the entries in the system's PATH environment variable (you can check using: 'echo \$PATH' on your terminal)
  - If the program is not in any of the directories of the PATH variable, then we have to input the command using its full path
    - If the command is in the current directory, we can use './<command>'

# Processes

- Processes are nothing but programs that are being executed
- Processes are created in the following ways
  - When we execute a program, a process is created
  - A child process can be created from within another process using `fork()` system call (more on this later)

# IPC at a Glance

- Operating Systems provide mechanisms for processes to cooperate and communicate with one another through Inter Process Communication (IPC)
- Pipes are one such mechanism to provide unidirectional communication channel between two processes
- We will work with pipes in today's lab

# A Few System Calls

- System calls are special functions that are invoked through the kernel space
- We will use the following system calls for today's lab
  - fork      - used to create child processes
  - waitpid    - used when a parent process blocks itself until a specific child process exits
  - pipe      - used to create a pipe for IPC
  - dup2      - used to duplicate a file descriptor into another file descriptor
  - execl      - used to execute a specific program with arguments just like we would do on a shell
  - read      - used to read from a file
  - write      - used to write to a file

# fork

- fork is a system call that creates a child process
  - Declaration: *pid\_t fork(void)*;<sup>1</sup>
  - Invocation: *pid\_val = fork()*;
- The process that calls fork is the parent process
- The child process is created by duplicating the parent process
- Starting from the line after the fork call, all the remaining lines of code are executed for both the parent and child processes

<sup>1</sup> <https://www.man7.org/linux/man-pages/man2/fork.2.html>

# fork (continued)

- Since the remaining lines of code are executed for both parent and child processes, we can use the return value from `fork` to identify the code sections that we want to run for either the parent or child process
  - Return value ( `pid_val == 0` ): child process section
    - When the child process looks at the `pid_val`, it finds 0
  - Return value ( `pid_val > 0` ): parent process section
    - When the parent process looks at the `pid_val`, it finds the unique process id of the child process (useful to keep track of the child processes)
  - Return value ( `pid_val == -1` ): error
    - When an error occurs, the *errno*<sup>1</sup> value is set and can be used by *perror*<sup>2</sup> to print error message
- The parent and child processes run in separate memory spaces
- The child inherits “copies” of parent’s attributes (e.g. *file descriptors*)

1) <https://man7.org/linux/man-pages/man3/errno.3.html>

2) <https://linux.die.net/man/3/perror>

# waitpid

- waitpid is a system call that lets a parent process wait for the completion of a child process
  - Declaration: `pid_t waitpid(pid_t pid, int *status, int options);`<sup>1</sup>
  - Invocation (for this lab): `waitpid(pid_val, NULL, 0);`
- Waiting for the child process prevents “orphan” and “zombie” processes (both waste system resources)
  - Orphan process:
    - When parent process finishes execution before child, the child becomes orphan process
    - Orphan process is adopted by the init or system daemon process
  - Zombie process:
    - When child process finishes execution before parent, the child becomes zombie process
    - If the parent process “waits” to read the exit status of the child, the child process is reaped from the process entry table and prevents the child from remaining a zombie process
    - The waitpid system call lets a parent process read the exit status of finished child processes and reaps off zombie processes

<sup>1</sup> <https://linux.die.net/man/2/waitpid>



# File Descriptors

- A file descriptor is a unique, non-negative integer used to identify an open file
- File descriptors can be used with open, close, read and write system calls
- Some special file descriptors:
  - <sup>1</sup>STDIN\_FILENO – a file descriptor for the standard input (keyboard)
  - <sup>2</sup>STDOUT\_FILENO – a file descriptor for the standard output (computer display)
  - Pipe file descriptors – two file descriptors for a pipe's read and write end (more on these later)
- Inside the PCB of a process, there is a file descriptor table which:
  - Keeps a mapping of file descriptors (used by the process) to actual files on the system
  - Is inherited by the children of the process

1, 2) <https://stackoverflow.com/questions/12902627/the-difference-between-stdout-and-stdout-fileno/12902707#12902707>

# pipe

- Pipe is a “unidirectional” data channel that can be used by a process to communicate with other processes
  - Declaration: *int pipe(int pipefd[2]);*<sup>1</sup>
  - Invocation:  
*int fd[2];*  
*pipe(fd);*
- fd[0] is the read end of the pipe
- fd[1] is the write end of the pipe
- **Close pipe ends when they are no longer needed (very important):**  
*close(fd[0]);*  
*close(fd[1]);*

<sup>1</sup> <https://man7.org/linux/man-pages/man2/pipe.2.html>

# dup2

- dup2 is a system call that copies one file descriptor into another
  - Declaration: *int dup2(int oldfd, int newfd);*<sup>1</sup>
  - Invocation: *dup2(fd\_one, fd\_two);*
- Can be used for I/O redirection
  - Input redirection:
    - *dup2(pipe\_read\_end, STDIN\_FILENO);*
    - Input of the process is taken from the pipe
  - Output redirection:
    - *dup2(pipe\_write\_end, STDOUT\_FILENO);*
    - Output of the process is sent to the pipe

<sup>1</sup> <https://man7.org/linux/man-pages/man2/dup.2.html>

# exec

- exec represents a group of system calls that can be used to execute external programs just like we would from a terminal
- We will use *execl* for today's lab
  - Declaration: *int execl(const char\* pathname, const char\* arg, ..., (char\*)NULL);*<sup>1</sup>
  - Invocation: *execl(program\_path, variable\_number\_of\_args, (char\*)NULL);*
- First arg of the *variable\_number\_of\_args* will be the name/path of the program itself. i.e. *execl("/bin/ls", "ls", "-l", "-a", (char\*)NULL);*
- *exec* system calls replace the calling process's image by a new image
- So, lines of code after a successful *exec* call inside a "specific process" will not be executed

<sup>1</sup> <https://man7.org/linux/man-pages/man3/exec.3.html>

# read

- read is a system call that is used to read from a file descriptor
  - Declaration: *ssize\_t read(int fd, void \*buf, size\_t count);*<sup>1</sup>
  - Invocation:

```
char buf[n];  
int numOfBytesRead = read(fd, buf, sizeof(buf));
```
- Reading from pipes:
  - Reading from a pipe will block the caller and read as long as any process has open write descriptors for that pipe
  - If all processes close the write end of a specific pipe, reading from that pipe will no longer block and instead return 0

<sup>1</sup> <https://www.man7.org/linux/man-pages/man2/read.2.html>

# write

- Write is a system call that is used to write to a file descriptor
  - Declaration: *ssize\_t write(int fd, void \*buf, size\_t count);*<sup>1</sup>
  - Invocation:

```
char buf[n];  
int numOfBytesWritten = write(fd, buf, sizeof(buf));
```
- Writing to pipes:
  - If the read end of a pipe has been closed by all processes, writing to that pipe will result in SIGPIPE signal and the process trying to write that pipe will be terminated

<sup>1</sup> <https://www.man7.org/linux/man-pages/man2/write.2.html>

# Lab Task

- In today's lab, we will build a ( filter – map – reduce ) pipeline
- Filter, map and reduce are interesting programs that serve different purposes (more on these in the next slides)
- Shell pipeline that we will replicate in **fmr.c**
  - `echo 1 2 3 4 5 | ./filter operator operand | ./map operator operand | ./reduce operator`
- e.g. (please try this on your terminal)
  - `echo 1 2 3 4 5 | ./filter ">=" 3 | ./map "*" 2 | ./reduce sum`
  - Output should be 24
  - Double quote characters (" ") from the operators (e.g. >= or \*) are necessary only on the terminal; omit when using with `execl` system call
- Your task for this lab will be to complete the "**fmrCompute**" function in **fmr.c**, using *fork*, *waitpid*, *pipe*, *dup2*, *execl* (*/bin/echo*, *./filter*, *./map*, *./reduce*) and *read/write* system calls so that **fmr.c** replicates the above filter-map-reduce pipeline.

# filter.c *(use as given)*

- Acts like a selection primitive
- Takes three types of inputs
  - List of input numbers to compare against
  - Operator for comparison
  - Operand for comparing against each of the input element
- Inputs a number at a time from “STDIN\_FILENO” (scanf)
- Outputs a number at a time to “STDOUT\_FILENO” (printf), if the comparison between the input number and the operand using the operator, is true

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main(int argc, char* argv[]){
6      char* comparingOperator = argv[1];
7      float comparingOperand = atof(argv[2]);
8
9      float val;
10
11     while(scanf("%f", &val) != EOF){
12         if(!strcmp(comparingOperator, "<") && (val <
13 comparingOperand)){
14             printf("%f\n", val);
15         }else if(!strcmp(comparingOperator, "<=") && (val <= comparingOperand)){
16             printf("%f\n", val);
17         }else if(!strcmp(comparingOperator, ">") && (val > comparingOperand)){
18             printf("%f\n", val);
19         }else if(!strcmp(comparingOperator, ">=") && (val >= comparingOperand)){
20             printf("%f\n", val);
21         }else if(!strcmp(comparingOperator, "==") && (val == comparingOperand)){
22             printf("%f\n", val);
23         }else if(!strcmp(comparingOperator, "!=") && (val != comparingOperand)){
24             printf("%f\n", val);
25         }else{
26
27         }
28     }
29
30     return 0;
31 }
```



# map.c *(use as given)*

- Acts like a transformation primitive
- Takes three types of inputs
  - List of input numbers to transform
  - Operator for transformation
  - Operand for transforming each input number
- Inputs a number at a time from “STDIN\_FILENO” (scanf)
- Outputs a number at a time to “STDOUT\_FILENO” (printf), after transforming the input number by the operand using the operator

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <math.h>
5
6  int main(int argc, char* argv[]){
7      char* operator = argv[1];
8      float operand = atof(argv[2]);
9
10     float val;
11
12     while(scanf("%f", &val) != EOF){
13         if(!strcmp(operator, "+")){
14             printf("%f\n", (val + operand));
15         }else if(!strcmp(operator, "-")){
16             printf("%f\n", (val - operand));
17         }else if(!strcmp(operator, "*")){
18             printf("%f\n", (val * operand));
19         }else if(!strcmp(operator, "/")){
20             printf("%f\n", (val / operand));
21         }else if(!strcmp(operator, "**")){
22             printf("%f\n", pow(val, operand));
23         }
24     }
25
26     return 0;
27 }
```

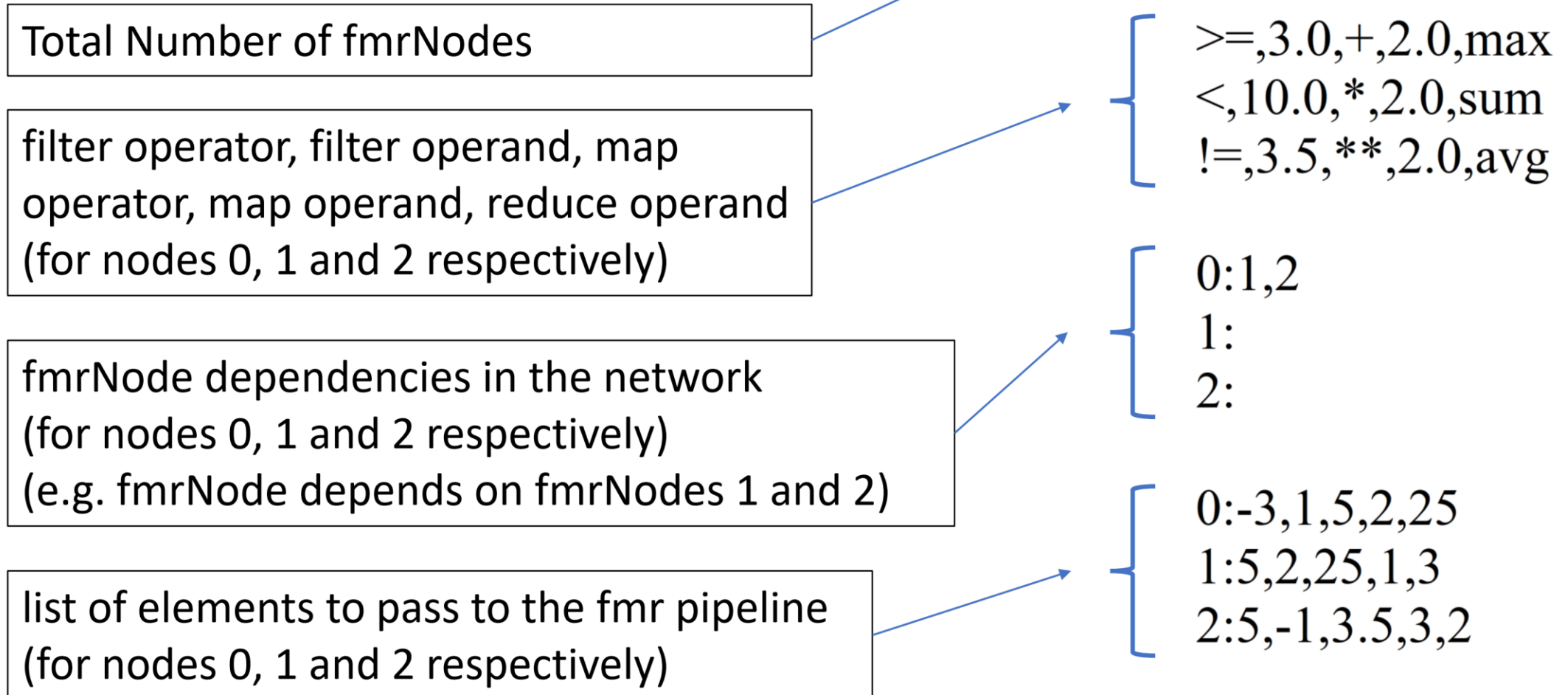
# reduce.c *(use as given)*

- Acts like an aggregation primitive
- Takes two types of inputs
  - List of input numbers to aggregate
  - Operator for aggregation
- Inputs a number at a time from “STDIN\_FILENO” (scanf)
- Outputs a number at a time to “STDOUT\_FILENO” (printf), after aggregating the input numbers using the aggregation operator

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  float getMax(int numElements, float* arr);
6  float getMin(int numElements, float* arr);
7  float getSum(int numElements, float* arr);
8  float getAvg(int numElements, float* arr);
9
10 int main(int argc, char* argv[]){
11     char* operator = argv[1];
12
13     int arraySize = 10, numElements = 0;
14
15     float val;
16     float* arr = (float*)malloc(arraySize * sizeof(float*));
17
18     while(scanf("%f", &val) != EOF){
19         if(numElements == arraySize){
20             arraySize = arraySize * 2;
21
22             arr = (float*)realloc(arr, arraySize * sizeof(float*));
23         }
24
25         arr[numElements++] = val;
26     }
27
28     if(!strcmp(operator, "max")){
29         float maxVal = getMax(numElements, arr);
30         printf("%f\n", maxVal);
31     }else if(!strcmp(operator, "min")){
32         printf("%f\n", getMin(numElements, arr));
33     }else if(!strcmp(operator, "sum")){
34         printf("%f\n", getSum(numElements, arr));
35     }else if(!strcmp(operator, "avg")){
36         printf("%f\n", getAvg(numElements, arr));
37     }
38 }
```

# Input file

Needed only for Extra  
Credit fmrNetwork



input.txt - Notepad  
File Edit Format View Help

3

>=,3.0,+,2.0,max  
<,10.0,\*,2.0,sum  
!=,3.5,\*\*,2.0,avg

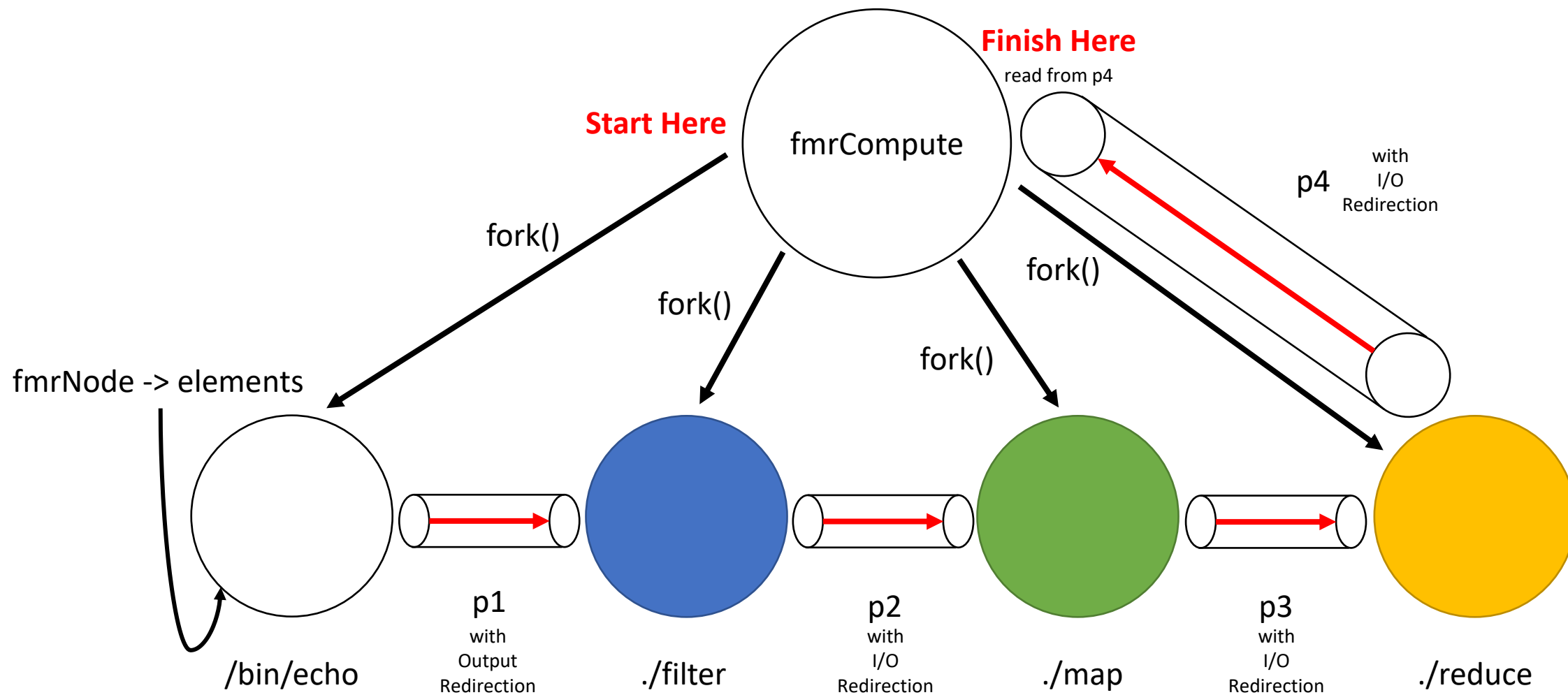
0:1,2  
1:  
2:

0:-3,1,5,2,25  
1:5,2,25,1,3  
2:5,-1,3.5,3,2

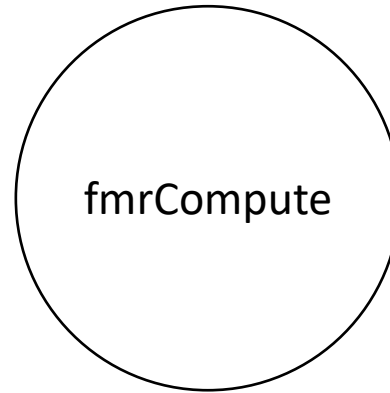
Fig. Input.txt

# Lab Task

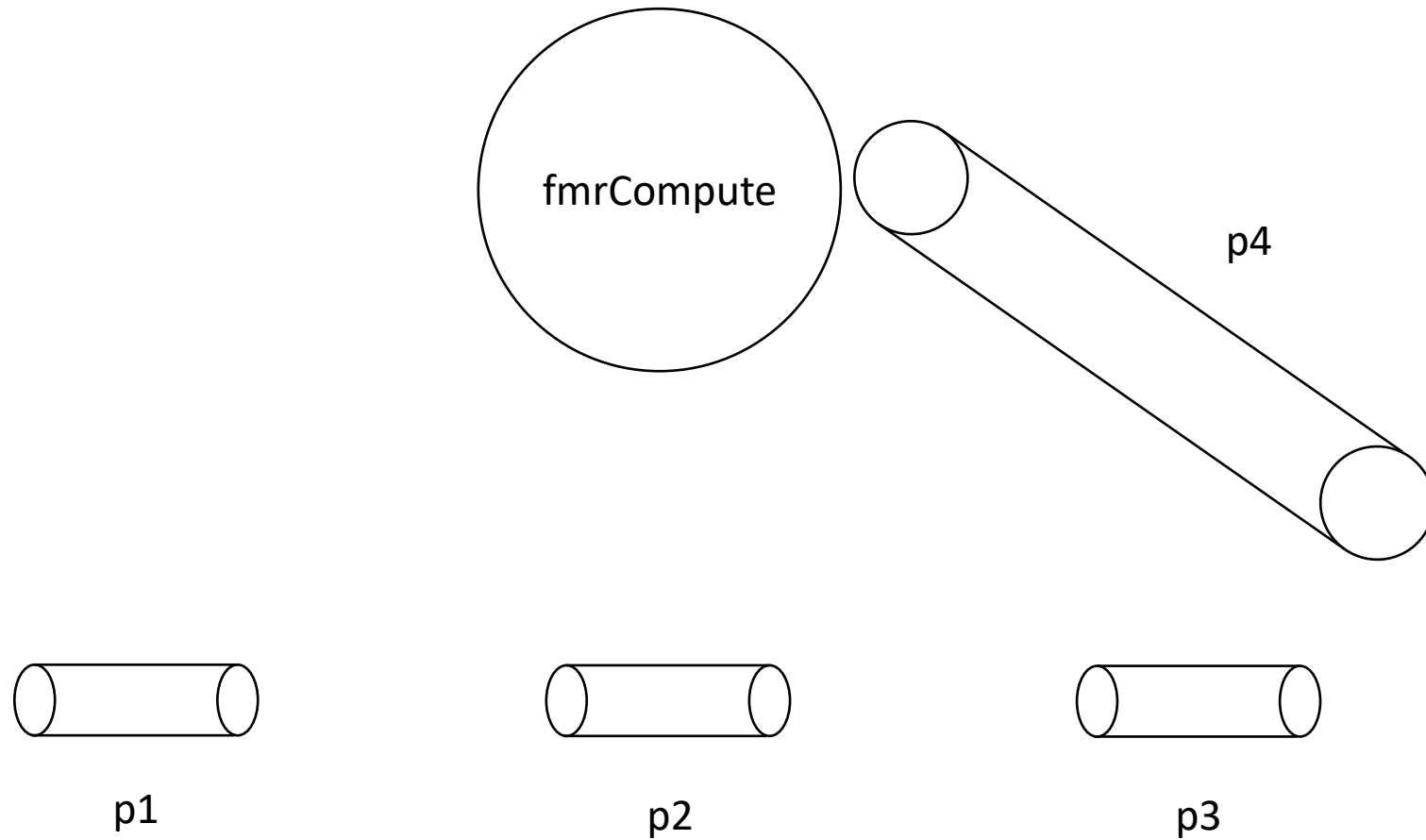
( *Echo – Filter – Map – Reduce* ) Pipeline in *fmrCompute* function



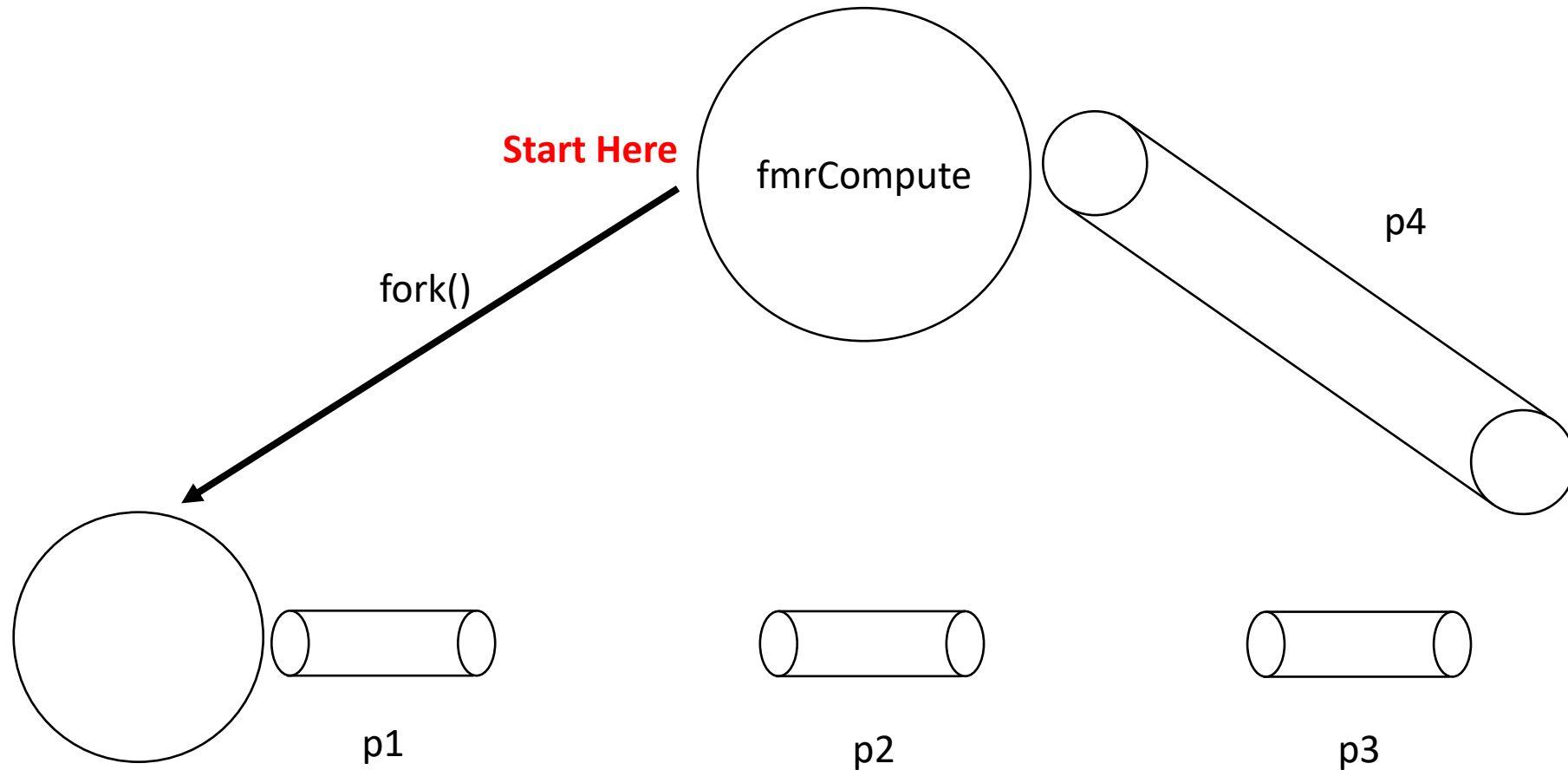
# Lab Task Simulation (fmrCompute function)



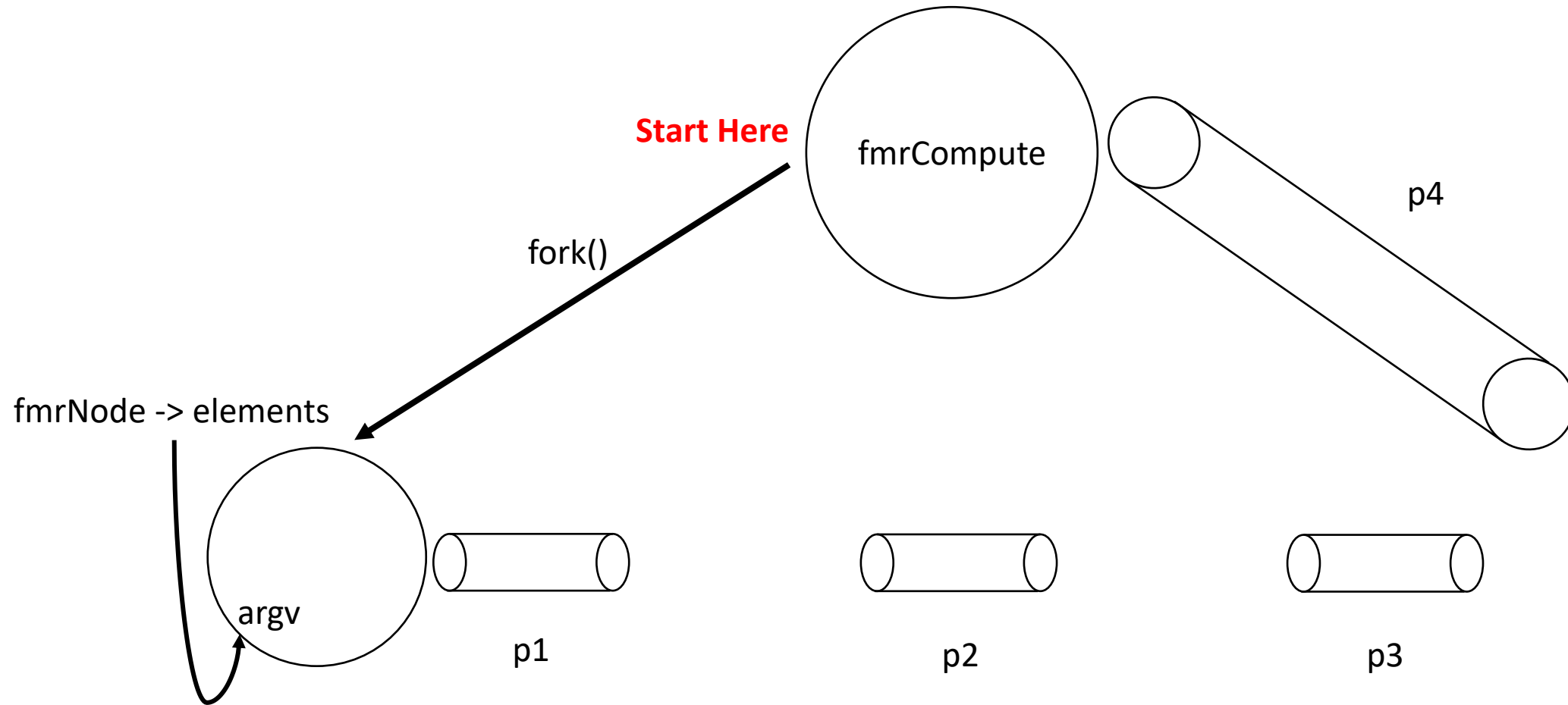
# Lab Task Simulation (pipe creation)



# Lab Task Simulation (first child process)



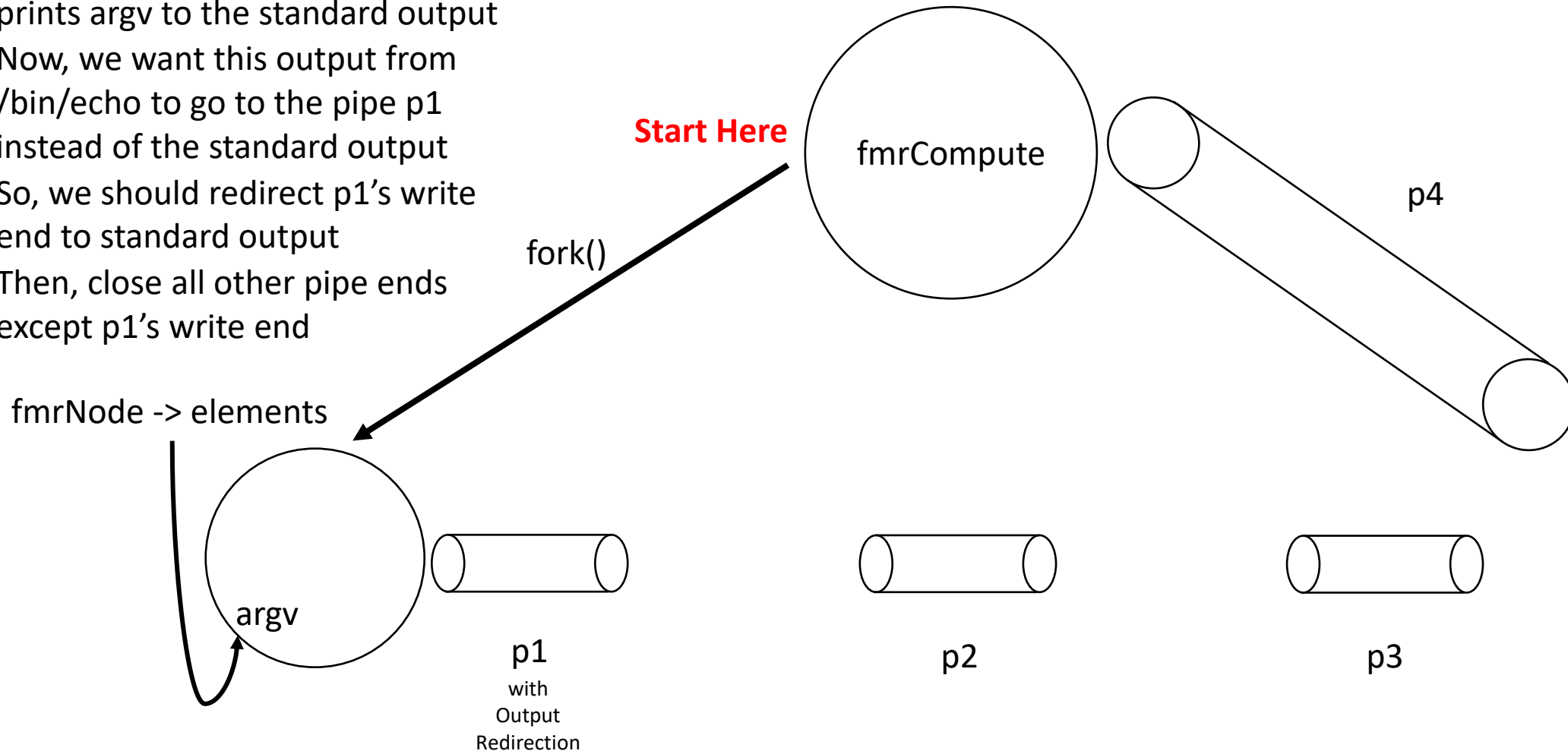
# Lab Task Simulation (first child process)





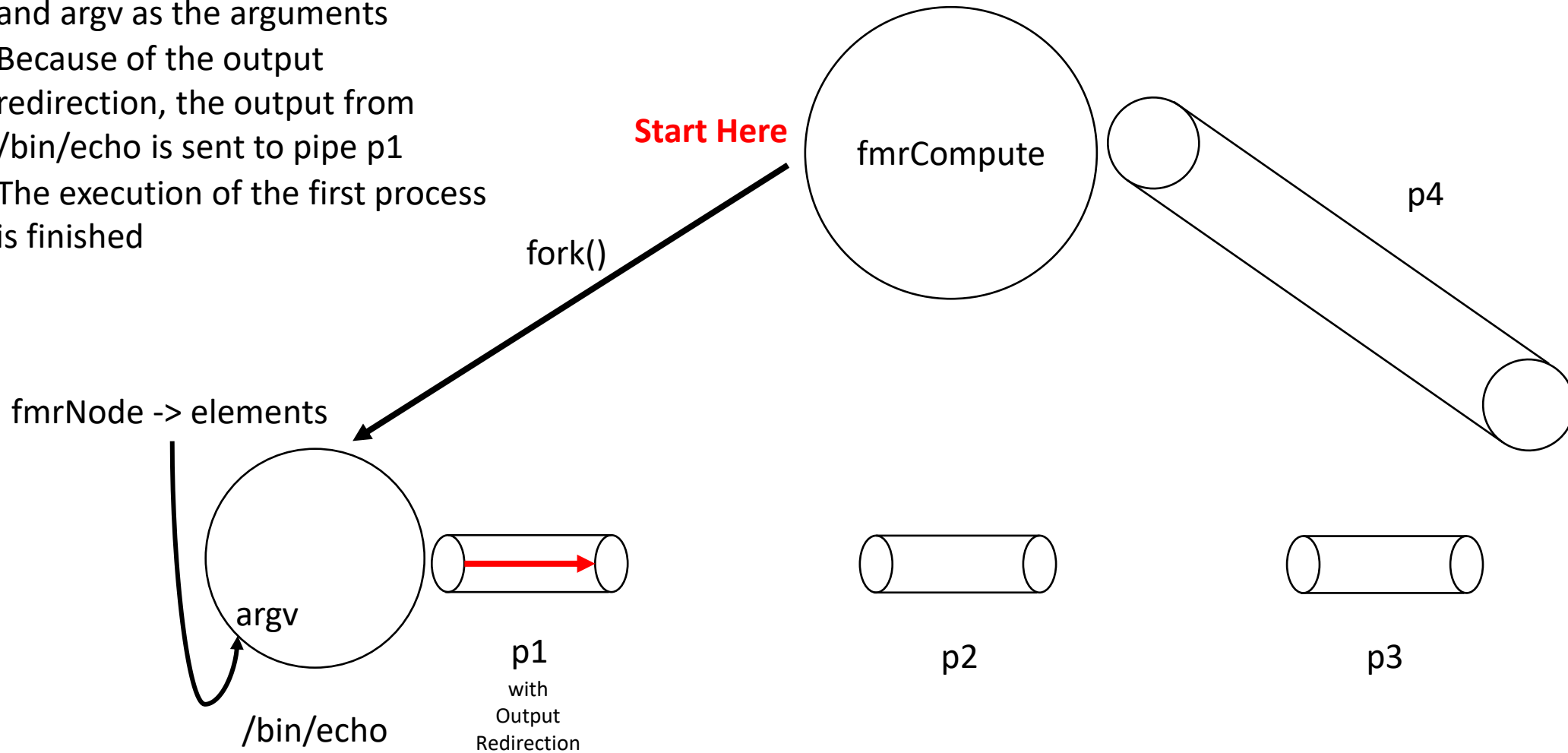
# Lab Task Simulation (first child process)

- The first child process will be used for /bin/echo which just prints argv to the standard output
- Now, we want this output from /bin/echo to go to the pipe p1 instead of the standard output
- So, we should redirect p1's write end to standard output
- Then, close all other pipe ends except p1's write end



# Lab Task Simulation (first child process)

- We finally call *exec()* inside the first child process, with */bin/echo* and *argv* as the arguments
- Because of the output redirection, the output from */bin/echo* is sent to pipe p1
- The execution of the first process is finished

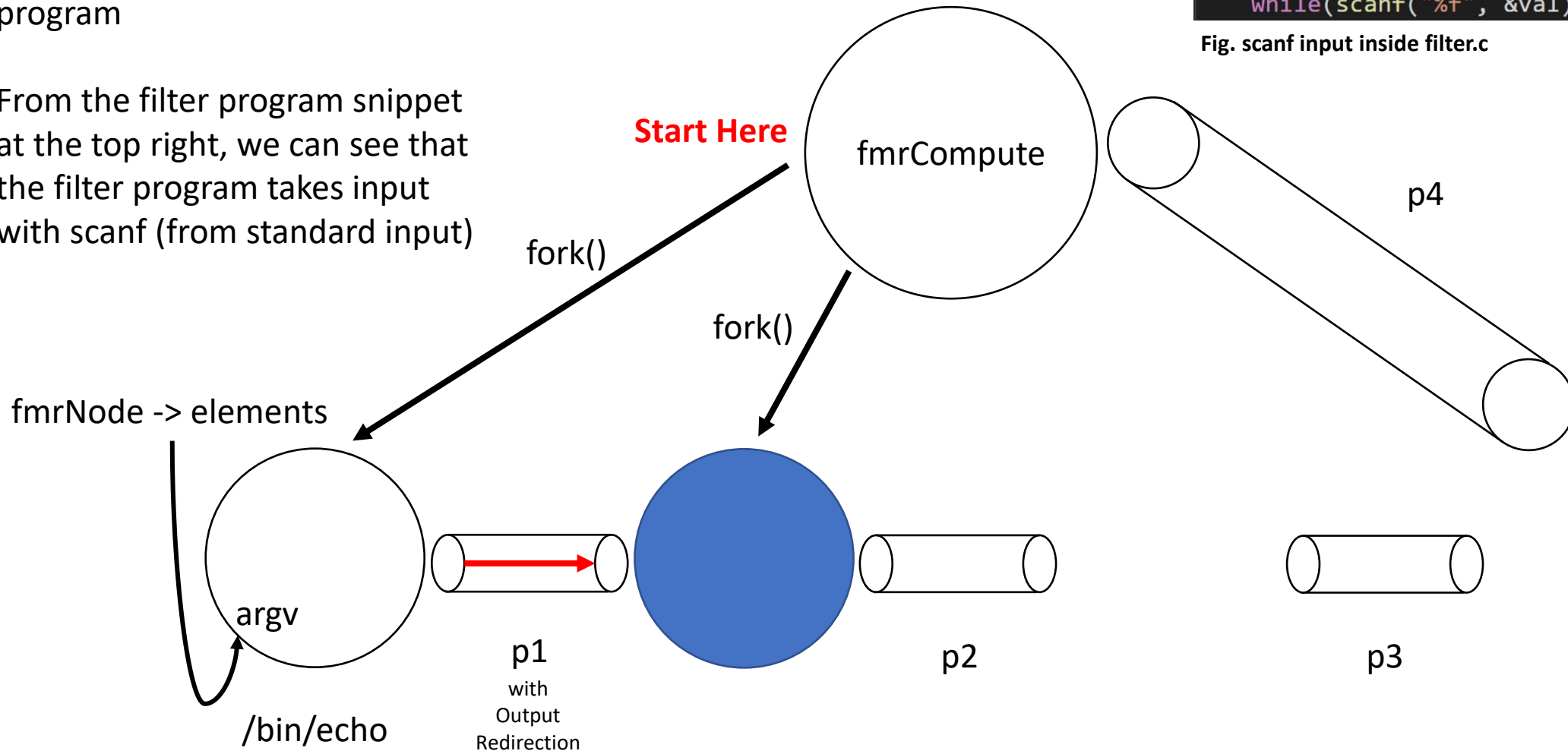


# Lab Task Simulation (second child process)

- The blue child process will be used to execute the `./filter` program
- From the filter program snippet at the top right, we can see that the filter program takes input with `scanf` (from standard input)

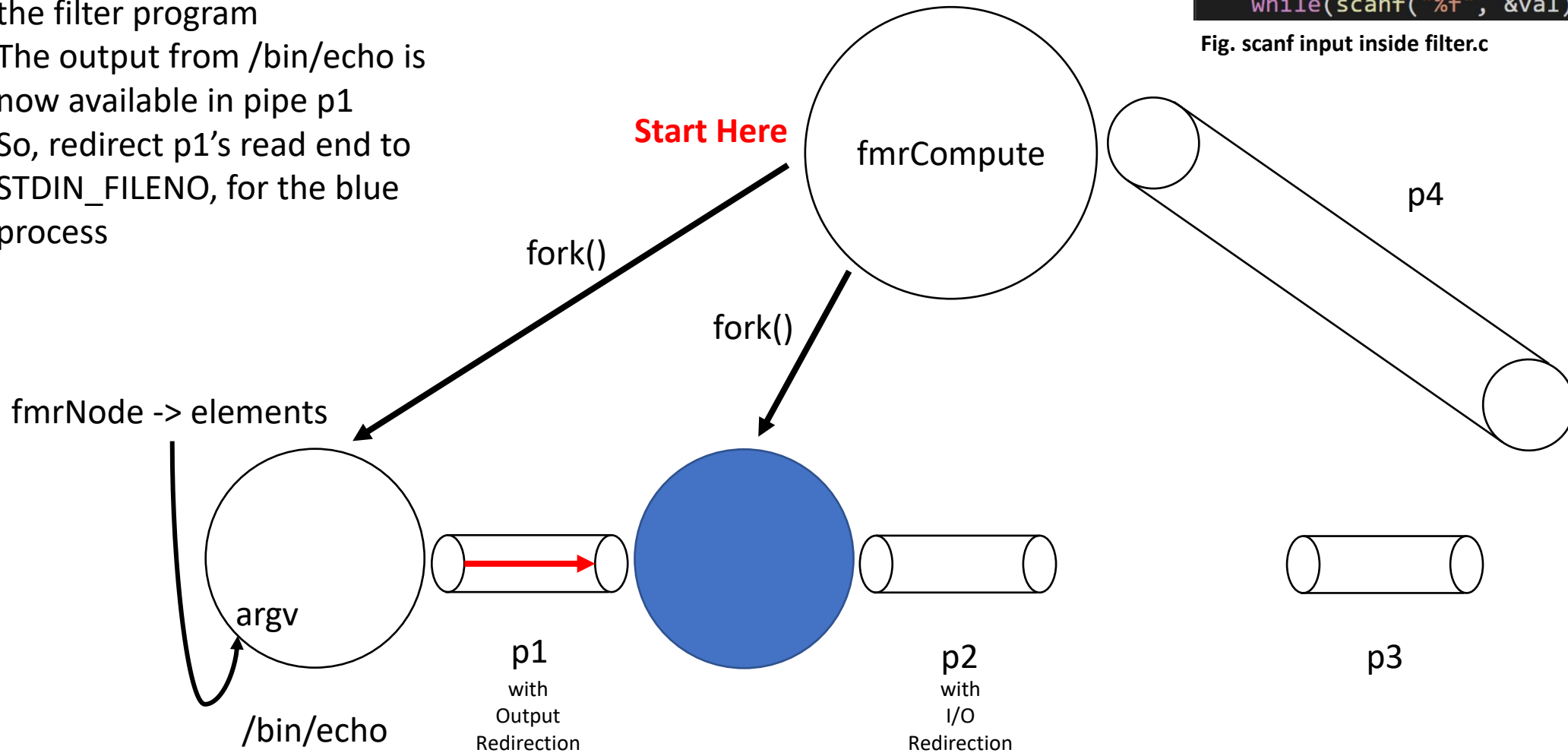
```
int main(int argc, char* argv){  
    char* comparingOperator = argv[1];  
    float comparingOperand = atof(argv[2]);  
  
    float val;  
  
    while(scanf("%f", &val) != EOF){
```

Fig. scanf input inside filter.c



# Lab Task Simulation (second child process)

- We want to use the output from /bin/echo as the scanf input for the filter program
- The output from /bin/echo is now available in pipe p1
- So, redirect p1's read end to STDIN\_FILENO, for the blue process

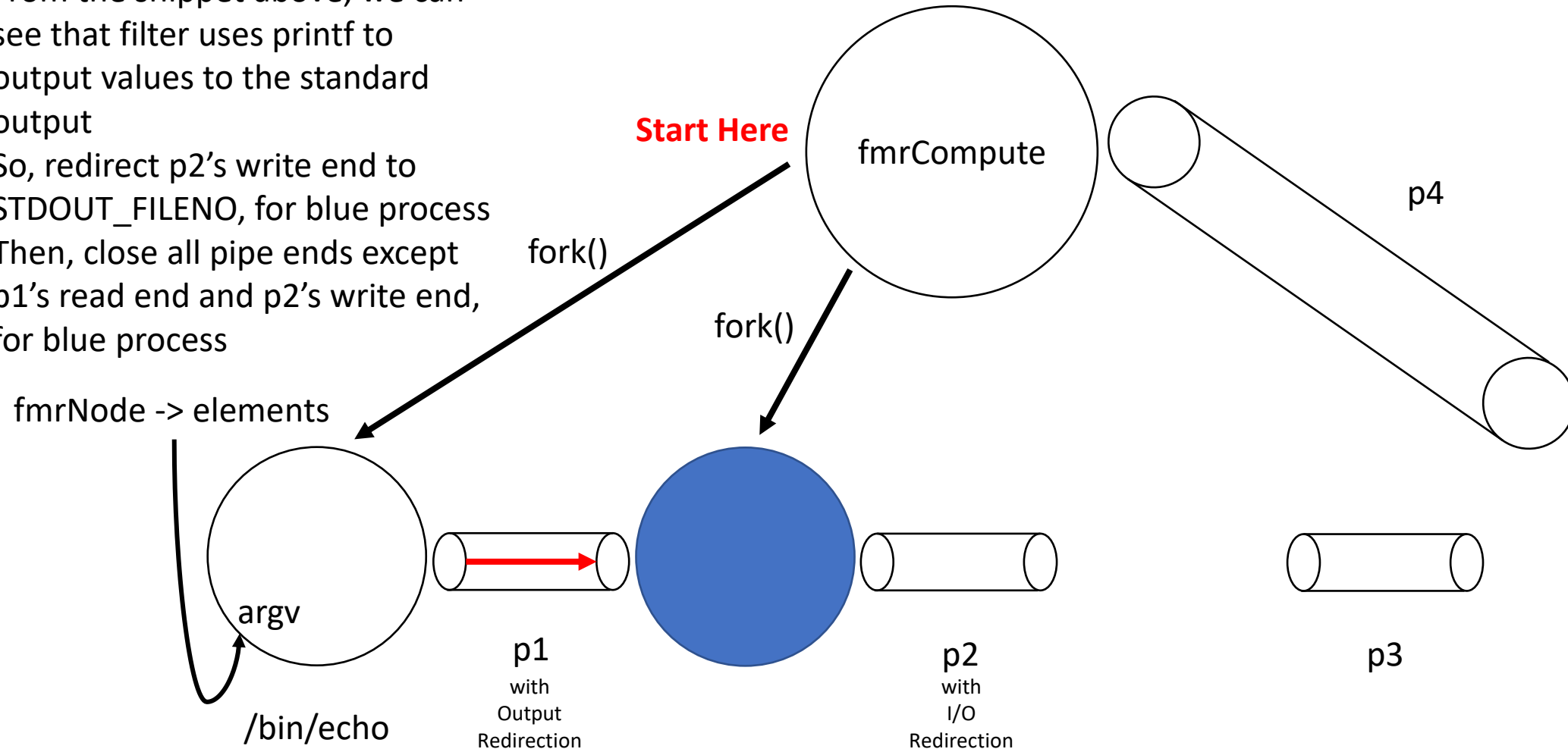


```
int main(int argc, char* argv){  
    char* comparingOperator = argv[1];  
    float comparingOperand = atof(argv[2]);  
  
    float val;  
  
    while(scanf("%f", &val) != EOF){
```

Fig. scanf input inside filter.c

# Lab Task Simulation (second child process)

- The output of the filter program should be sent to pipe p2
- From the snippet above, we can see that filter uses printf to output values to the standard output
- So, redirect p2's write end to STDOUT\_FILENO, for blue process
- Then, close all pipe ends except p1's read end and p2's write end, for blue process



```
while(scanf("%f", &val) != EOF){  
    if(!strcmp(comparingOperator, "<") && (val < comparingOperand)){  
        printf("%f\n", val);  
    }else if(!strcmp(comparingOperator, "<=") && (val <= comparingOperand)){  
        printf("%f\n", val);  
    }  
}
```

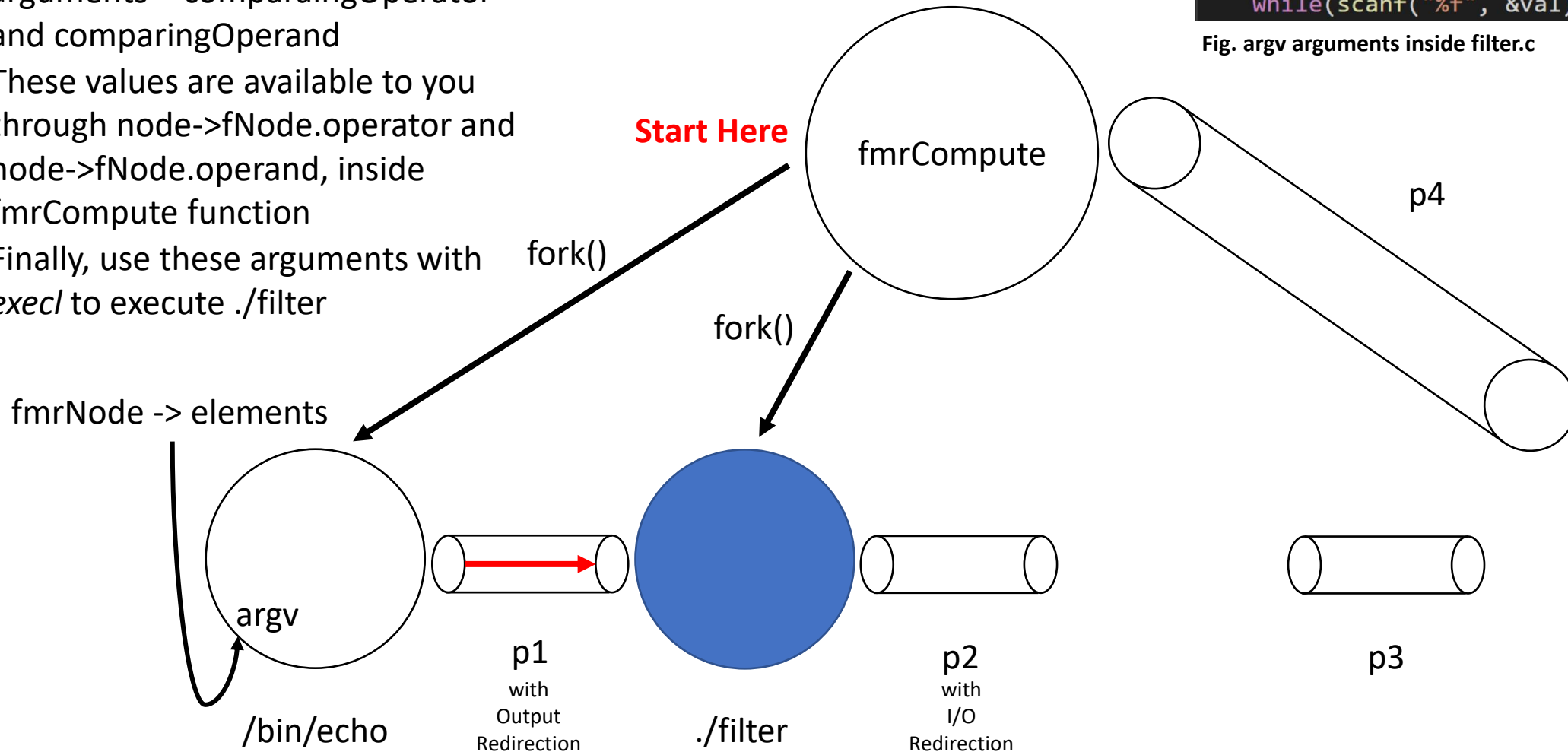
Fig. printf output inside filter.c

# Lab Task Simulation (second child process)

- From the snippet to the right, we can see that filter program takes two arguments – comparingOperator and comparingOperand
- These values are available to you through node->fNode.operator and node->fNode.operand, inside fmrCompute function
- Finally, use these arguments with `fork()` to execute `./filter`

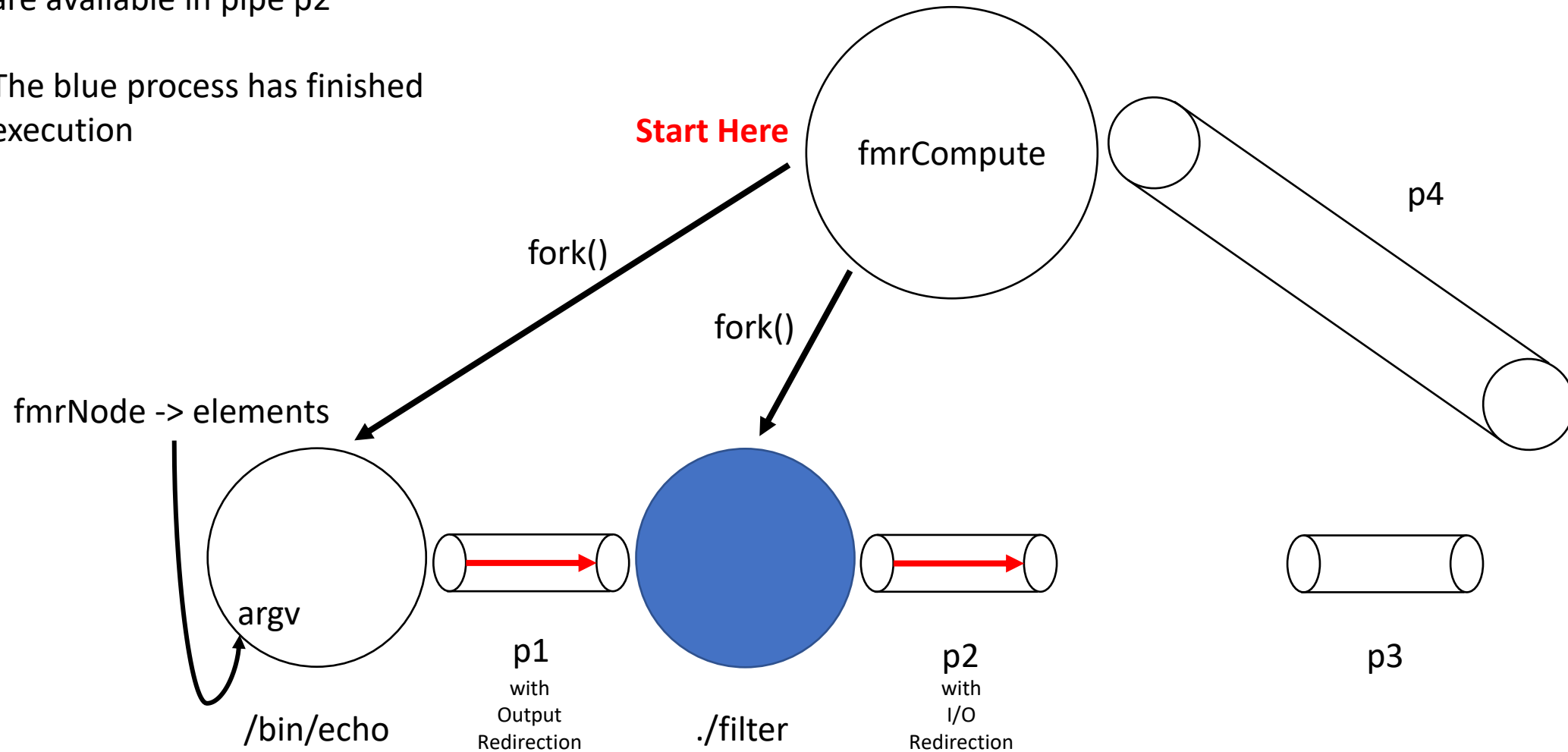
```
int main(int argc, char* argv[]){  
    char* comparingOperator = argv[1];  
    float comparingOperand = atof(argv[2]);  
  
    float val;  
  
    while(scanf("%f", &val) != EOF){
```

Fig. argv arguments inside filter.c



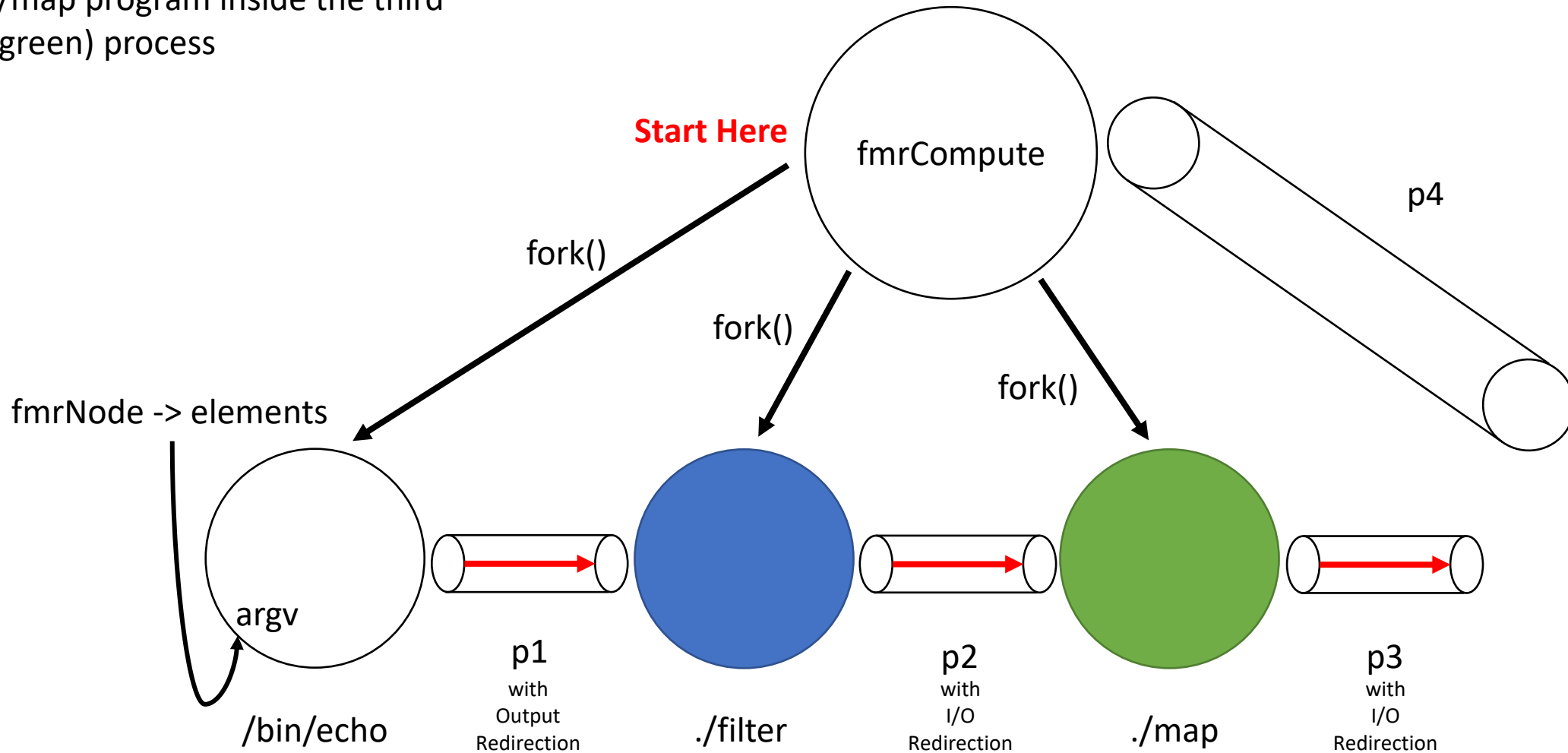
# Lab Task Simulation (second child process)

- Now, the “filtered” values from filter are available in pipe p2
- The blue process has finished execution



# Lab Task Simulation (third child process)

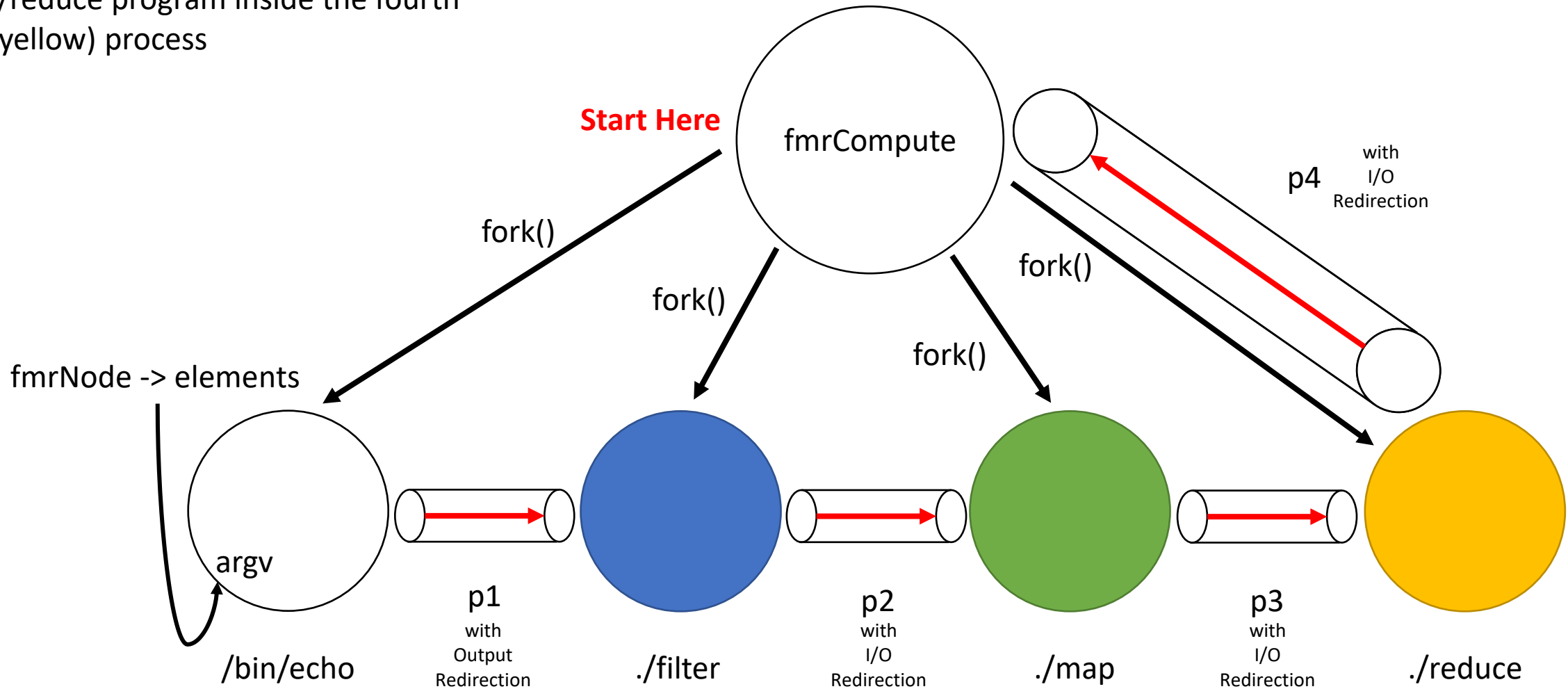
- Follow a similar pattern for executing `./map` program inside the third (green) process





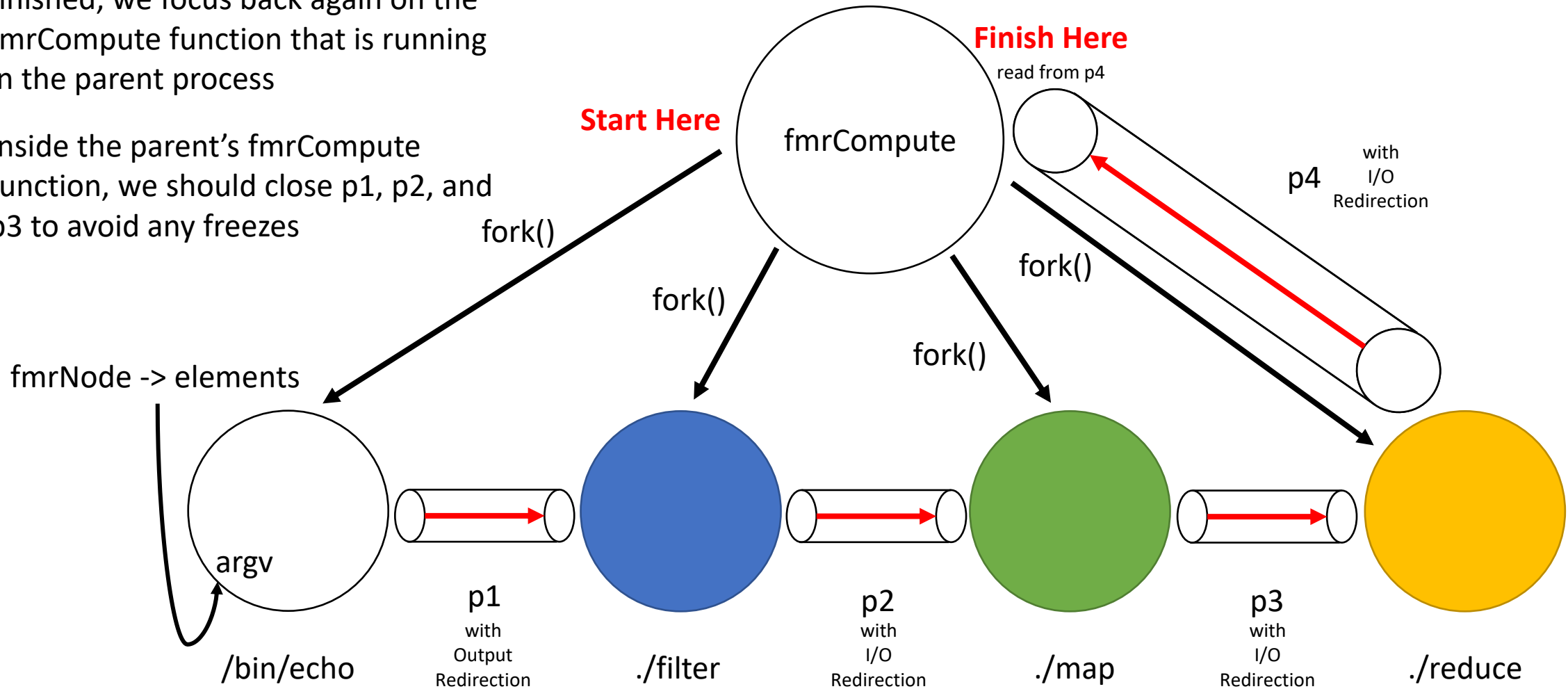
# Lab Task Simulation (fourth child process)

- Follow a similar pattern for executing `./reduce` program inside the fourth (yellow) process



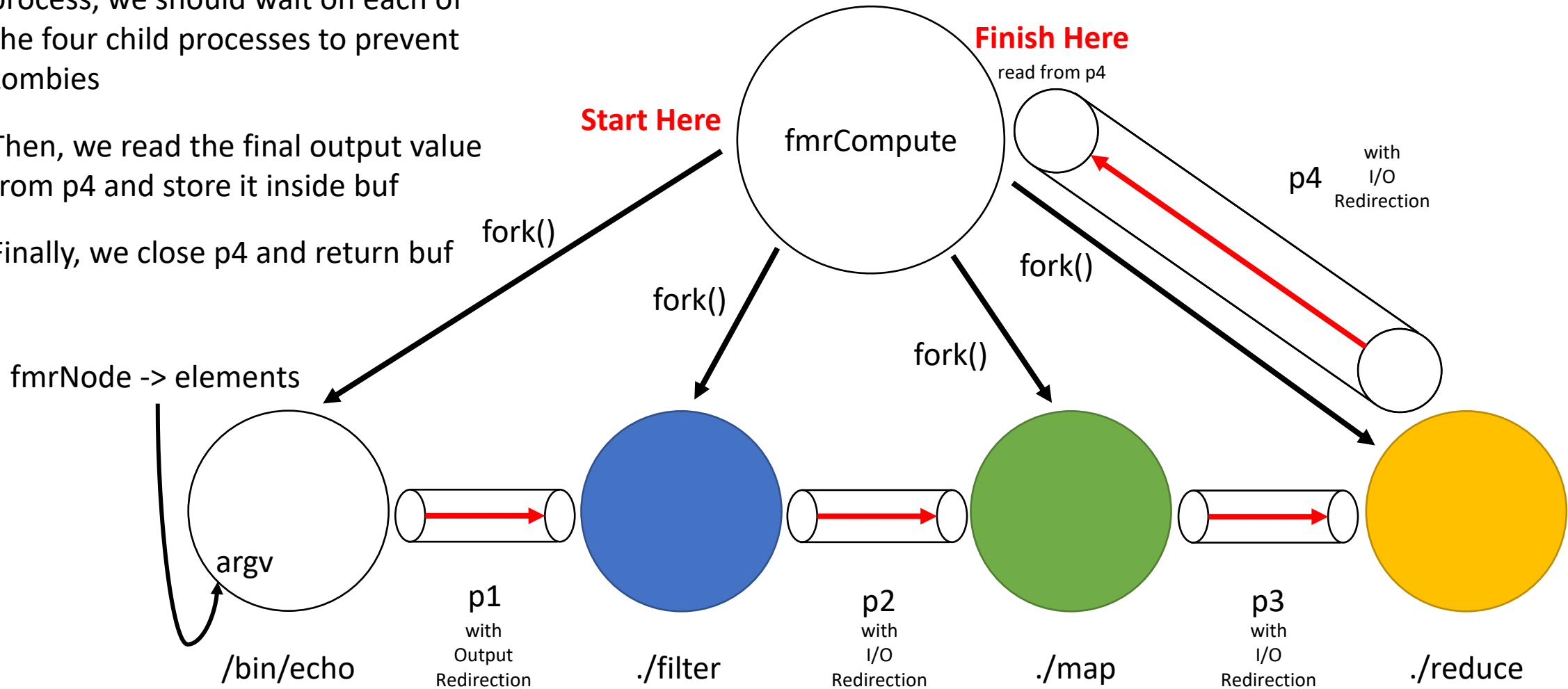
# Lab Task Simulation (back to the parent)

- Once all the four child processes are finished, we focus back again on the fmrCompute function that is running in the parent process
- Inside the parent's fmrCompute function, we should close p1, p2, and p3 to avoid any freezes



# Lab Task Simulation (back to the parent)

- Then, inside the same parent process, we should wait on each of the four child processes to prevent zombies
- Then, we read the final output value from p4 and store it inside buf
- Finally, we close p4 and return buf



# Extra Credit

- Only attempt extra credit after you have finished the fmrCompute function for lab task
- In the extra credit section, you will have to finish the fmrNetwork function
- The fmrNetwork function will create a tree of ( echo-filter-map-reduce ) pipelines
- In the tree, every node will depend on fmrCompute values from its children
- Every node will add all of the children fmrCompute values to its own elements array
- After inserting children fmrCompute values, the node will perform its own fmrCompute