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

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);*[1]
  - 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

1) 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*[1] value is set and can be used by *perror*[2] 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);*[1]
  - 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

1) 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:
  - [1]STDIN_FILENO – a file descriptor for the standard input (keyboard)
  - [2]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]);*[1]
  - 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]);

1) 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);*[1]
  - 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

1) 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);*[1]
  - Invocation: *execl(program_path, variable_number_of_args, (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

1) 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);*[1]
  - 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

1) 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);*[1]
  - 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

1) 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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char* argv[]){
    char* comparingOperator = argv[1];
    float comparingOperand = atof(argv[2]);

    float val;

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

        }
    }

    return 0;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

int main(int argc, char* argv[]){
    char* operator = argv[1];
    float operand = atof(argv[2]);

    float val;

    while(scanf("%f", &val) != EOF){
        if(!strcmp(operator, "+")){
            printf("%f\n", (val + operand));
        }else if(!strcmp(operator, "-")){
            printf("%f\n", (val - operand));
        }else if(!strcmp(operator, "*")){
            printf("%f\n", (val * operand));
        }else if(!strcmp(operator, "/")){
            printf("%f\n", (val / operand));
        }else if(!strcmp(operator, "**")){
            printf("%f\n", pow(val, operand));
        }
    }

    return 0;
}
```

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

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

float getMax(int numElements, float* arr);
float getMin(int numElements, float* arr);
float getSum(int numElements, float* arr);
float getAvg(int numElements, float* arr);

int main(int argc, char* argv[]){
    char* operator = argv[1];

    int arraySize = 10, numElements = 0;

    float val;
    float* arr = (float*)malloc(arraySize * sizeof(float*));

    while(scanf("%f", &val) != EOF){
        if(numElements == arraySize){
            arraySize = arraySize * 2;

            arr = (float*)realloc(arr, arraySize * sizeof(float*));
        }

        arr[numElements++] = val;
    }

    if(!strcmp(operator, "max")){
        float maxVal = getMax(numElements, arr);
        printf("%f\n", maxVal);
    }else if(!strcmp(operator, "min")){
        printf("%f\n", getMin(numElements, arr));
    }else if(!strcmp(operator, "sum")){
        printf("%f\n", getSum(numElements, arr));
    }else if(!strcmp(operator, "avg")){
        printf("%f\n", getAvg(numElements, arr));
    }
}
```

Input file

Total Number of fmrNodes

filter operator, filter operand, map operator, map operand, reduce operand (for nodes 0, 1 and 2 respectively)

Needed only for Extra Credit fmrNetwork

fmrNode dependencies in the network (for nodes 0, 1 and 2 respectively) (e.g. fmrNode depends on fmrNodes 1 and 2)

list of elements to pass to the fmr pipeline (for nodes 0, 1 and 2 respectively)

input.txt - Notepad

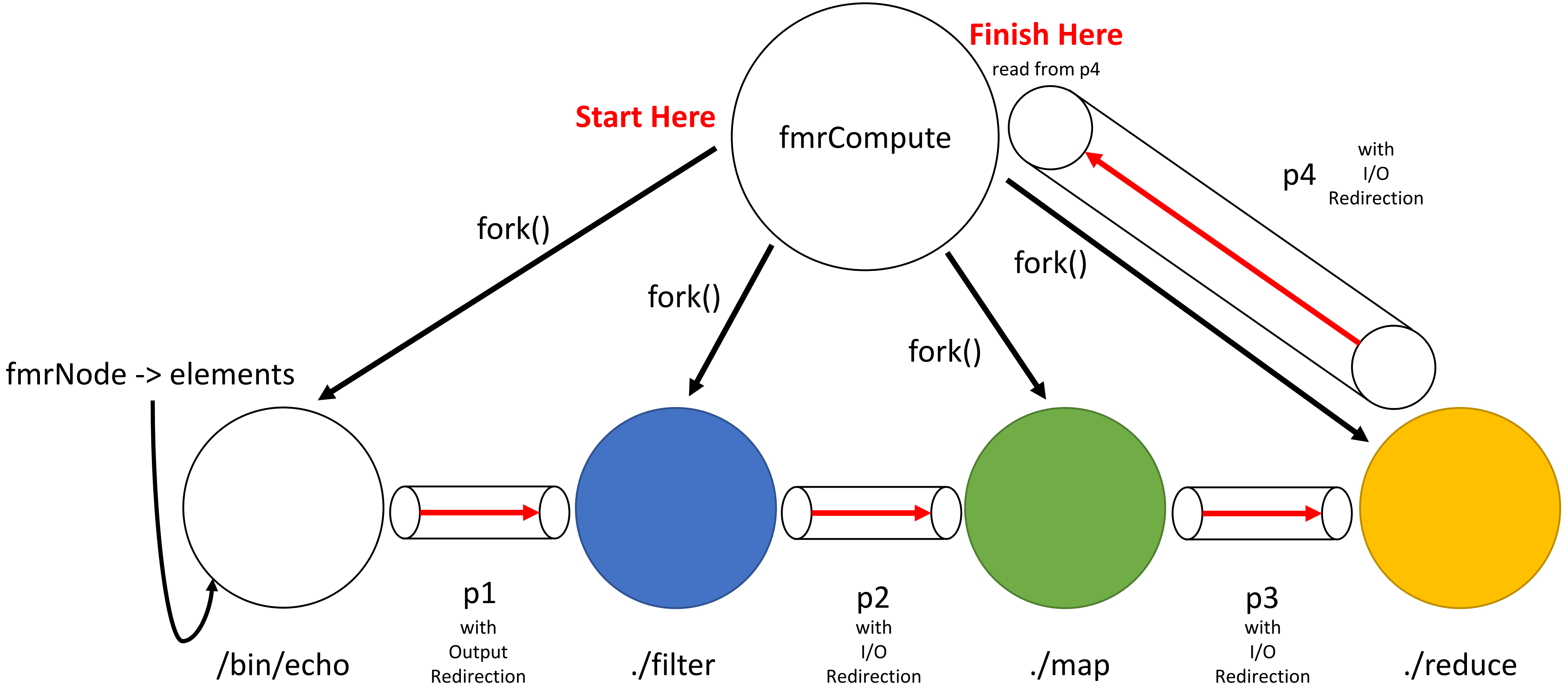
File Edit Format View Help

```
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
```

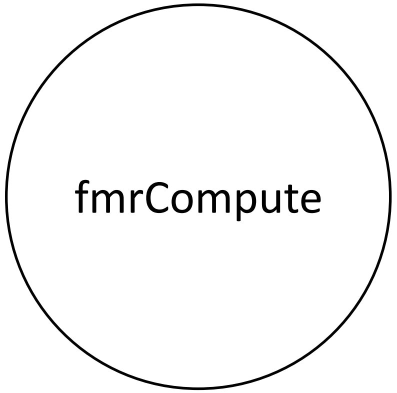
Fig. Input.txt

Lab Task

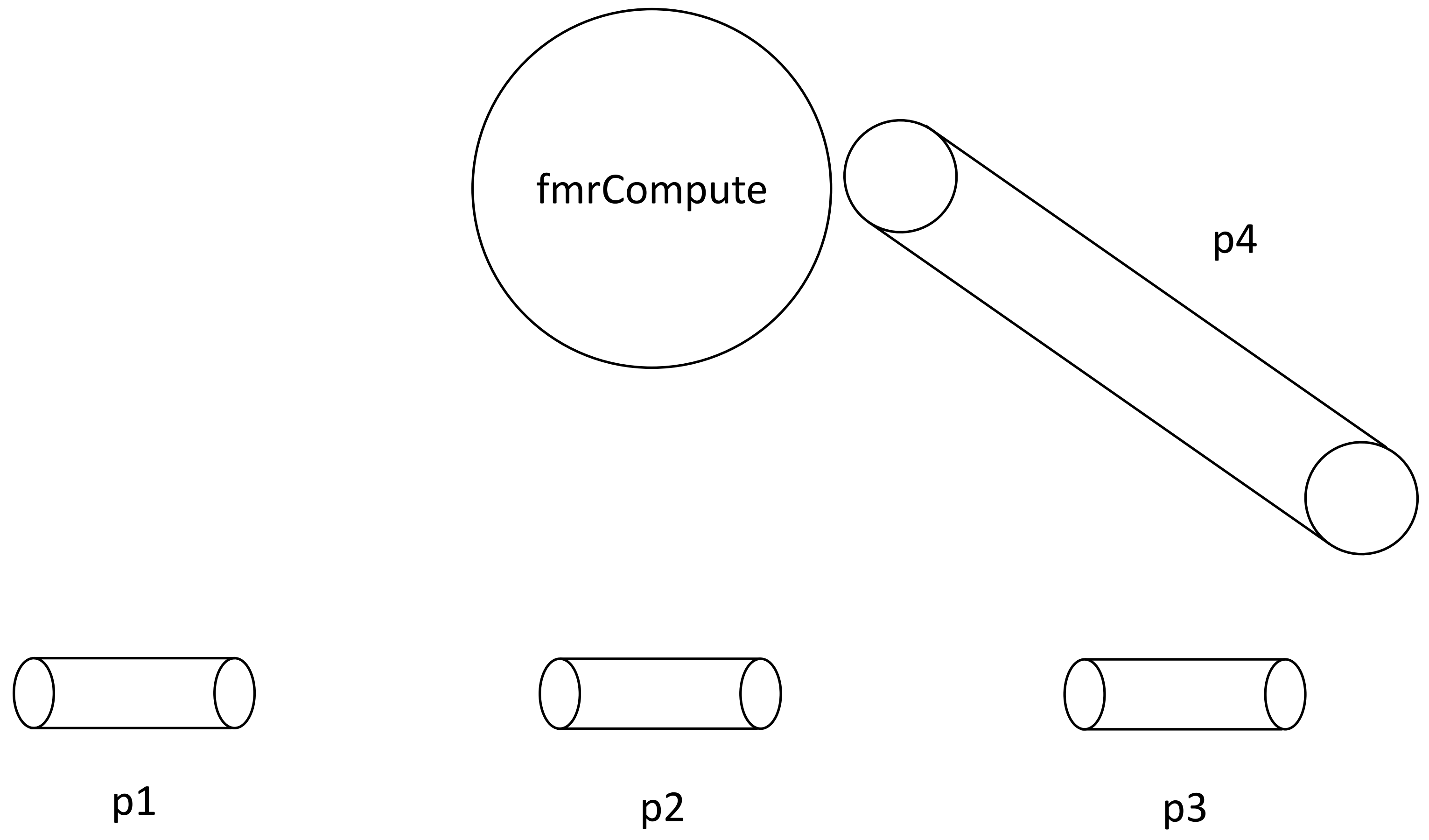
*(Echo – Filter – Map – Reduce) Pipeline in **fmrCompute** function*



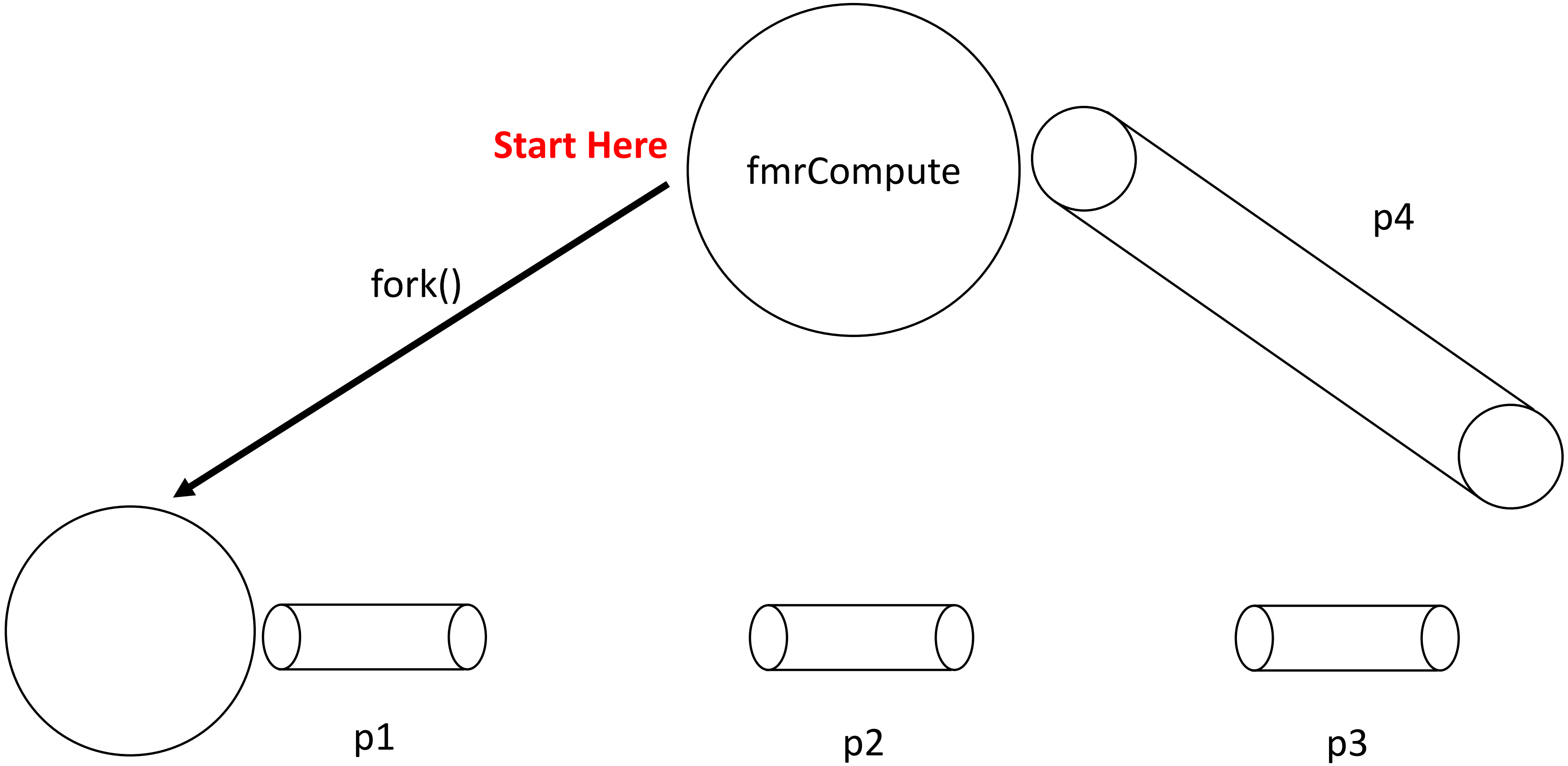
Lab Task Simulation (fmrCompute function)



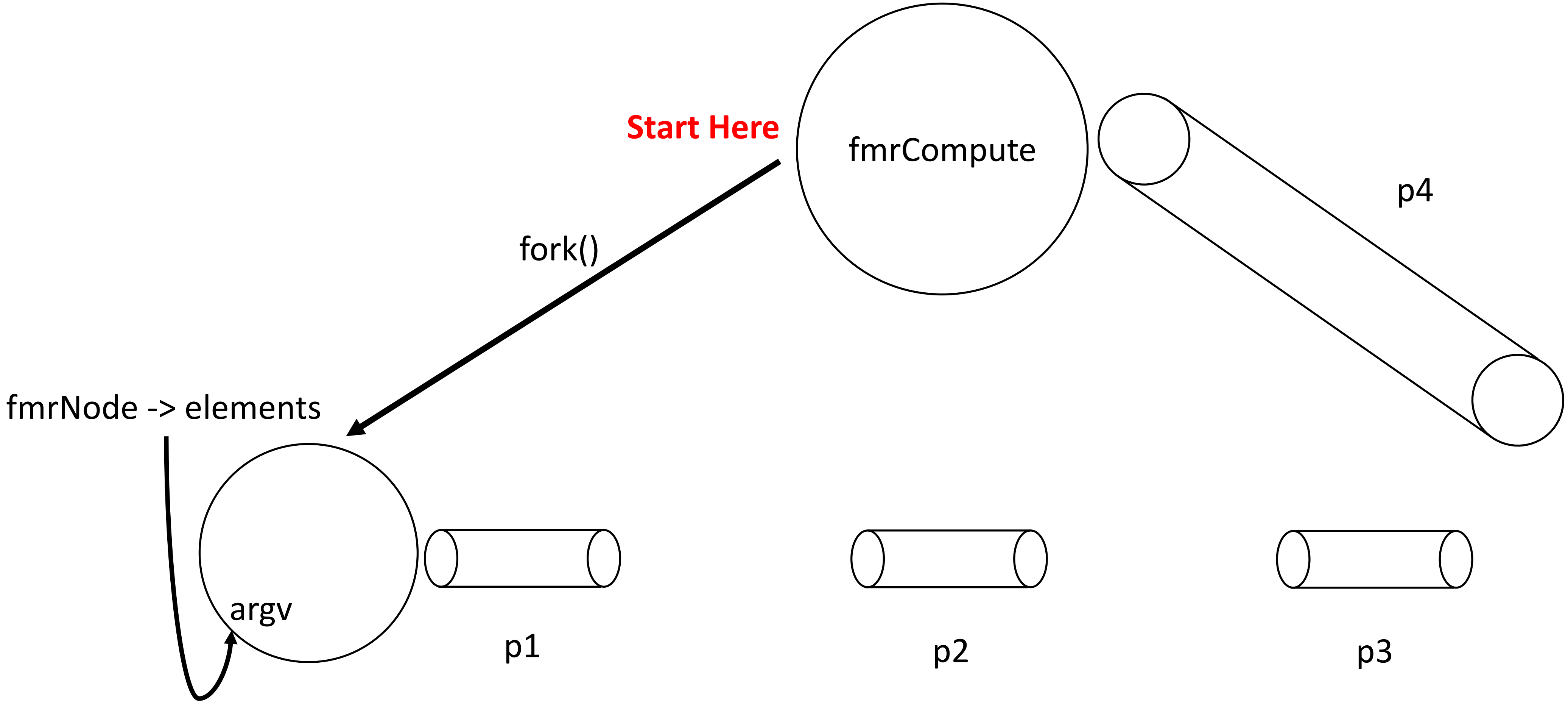
Lab Task Simulation (pipe creation)



Lab Task Simulation (first child process)

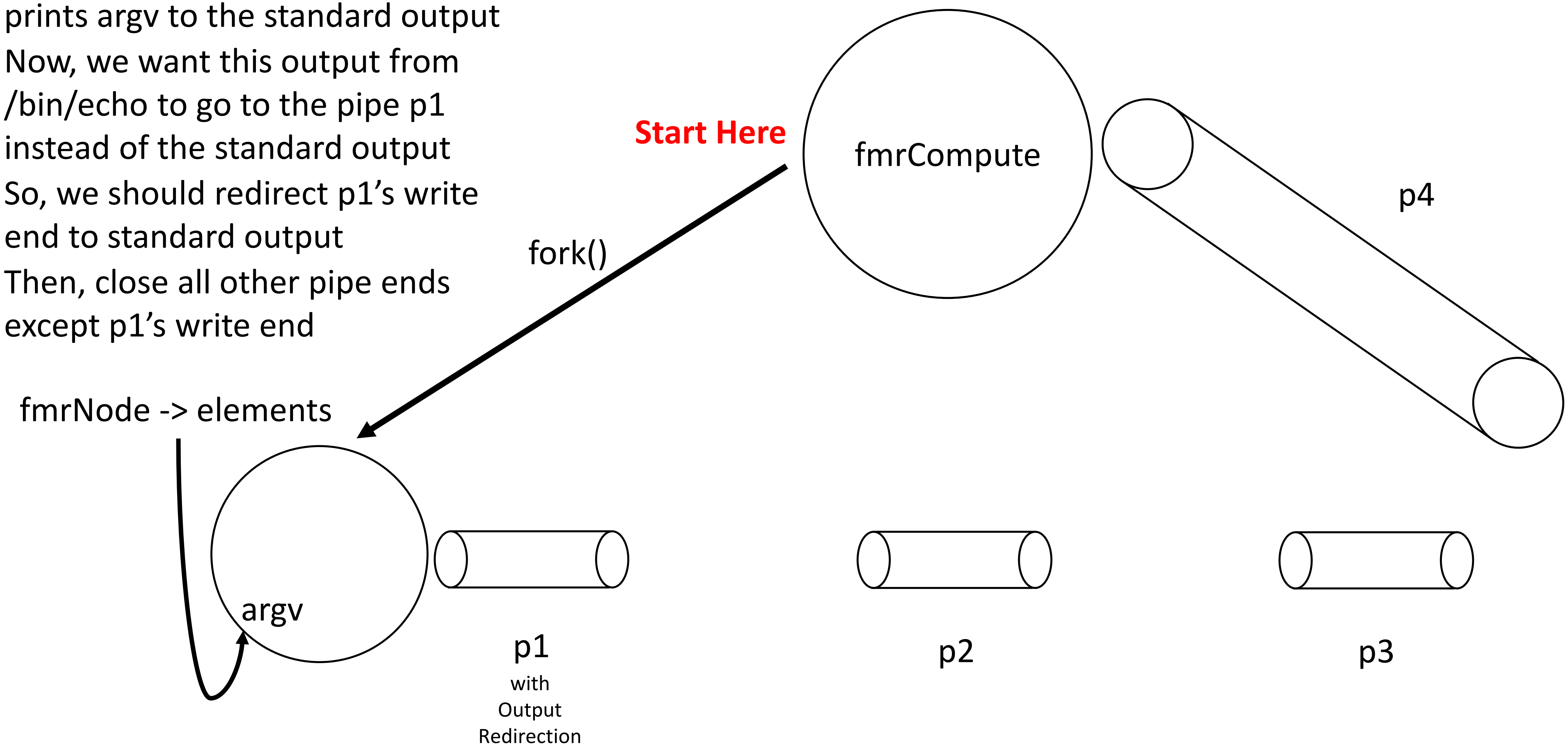


Lab Task Simulation (first child process)



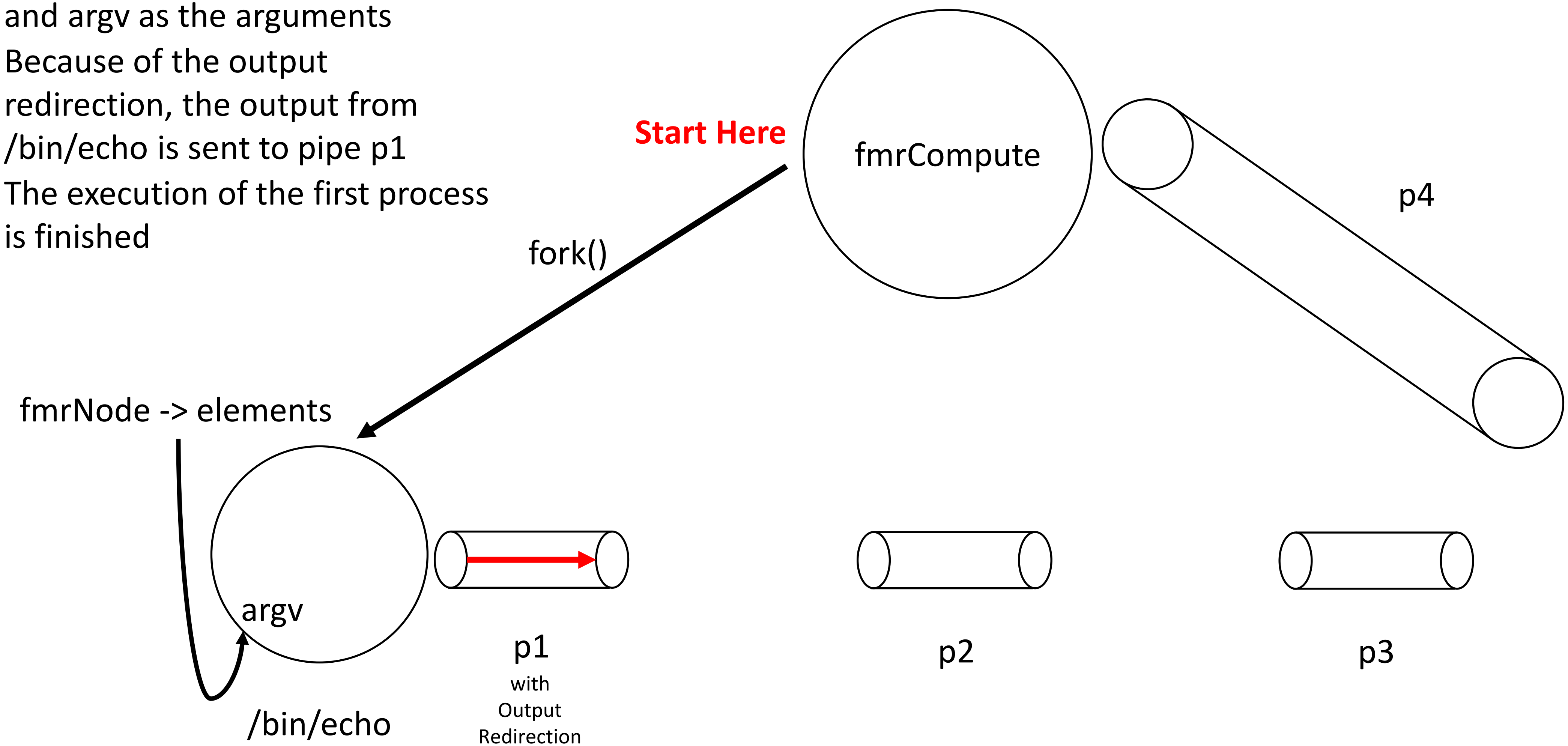
Lab Task Simulation (first child process)

- 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

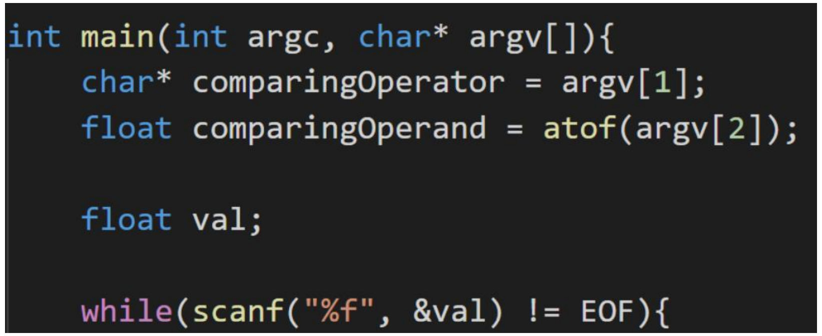


Lab Task Simulation (first child process)

- We finally call *execl* 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



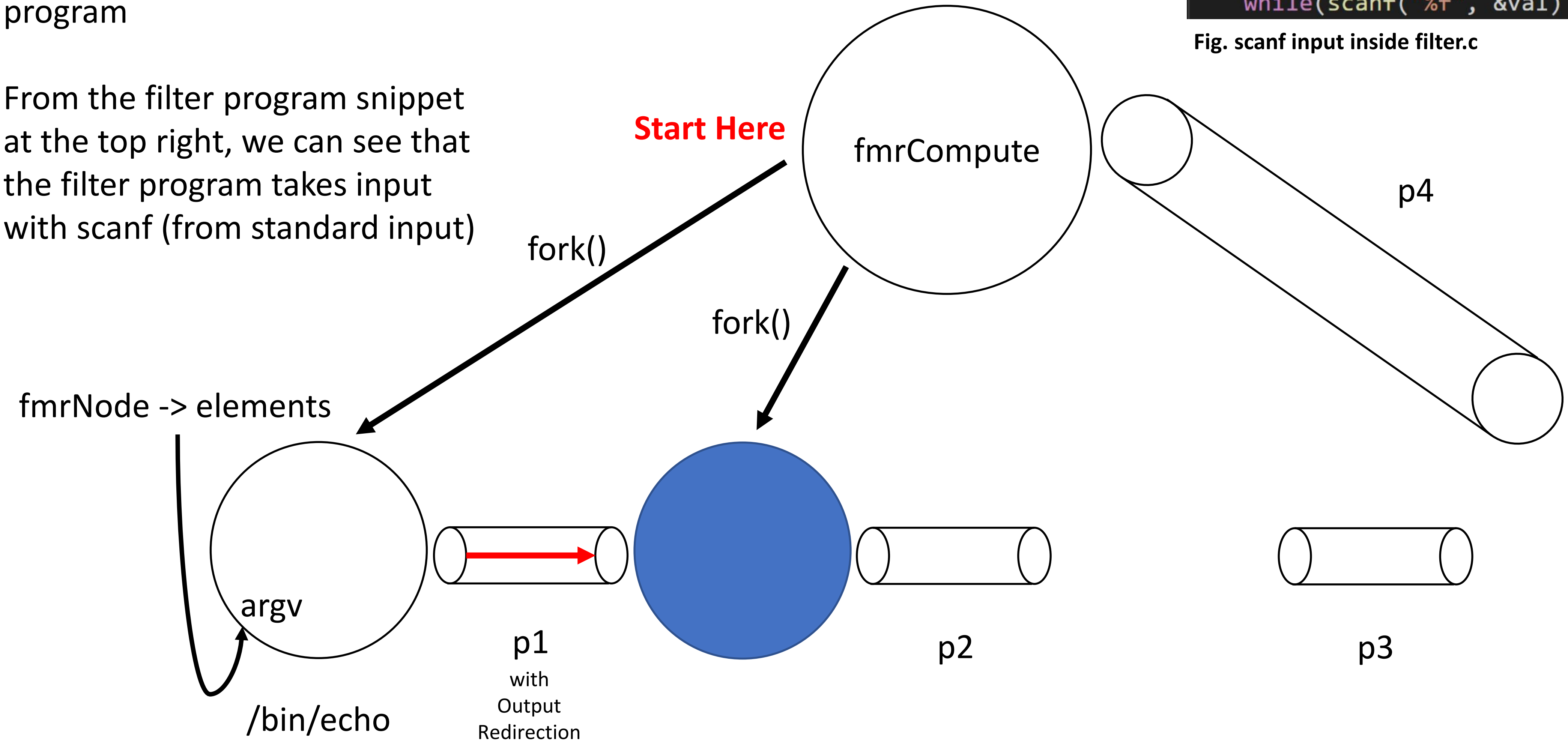
Lab Task Simulation (second child process)



```
int main(int argc, char* argv[]){
    char* comparingOperator = argv[1];
    float comparingOperand = atof(argv[2]);

    float val;

    while(scanf("%f", &val) != EOF){
```

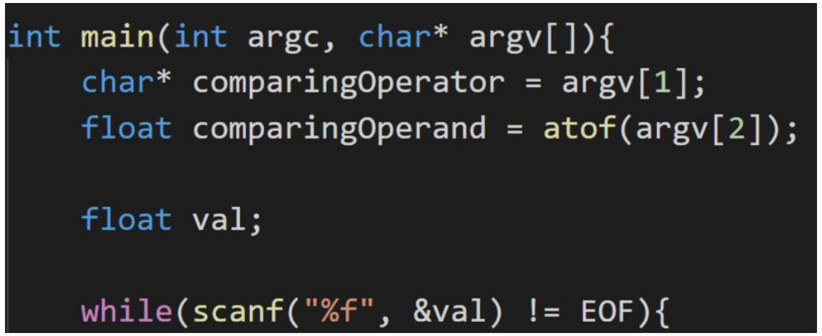
Fig. scanf input inside filter.c

- 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)



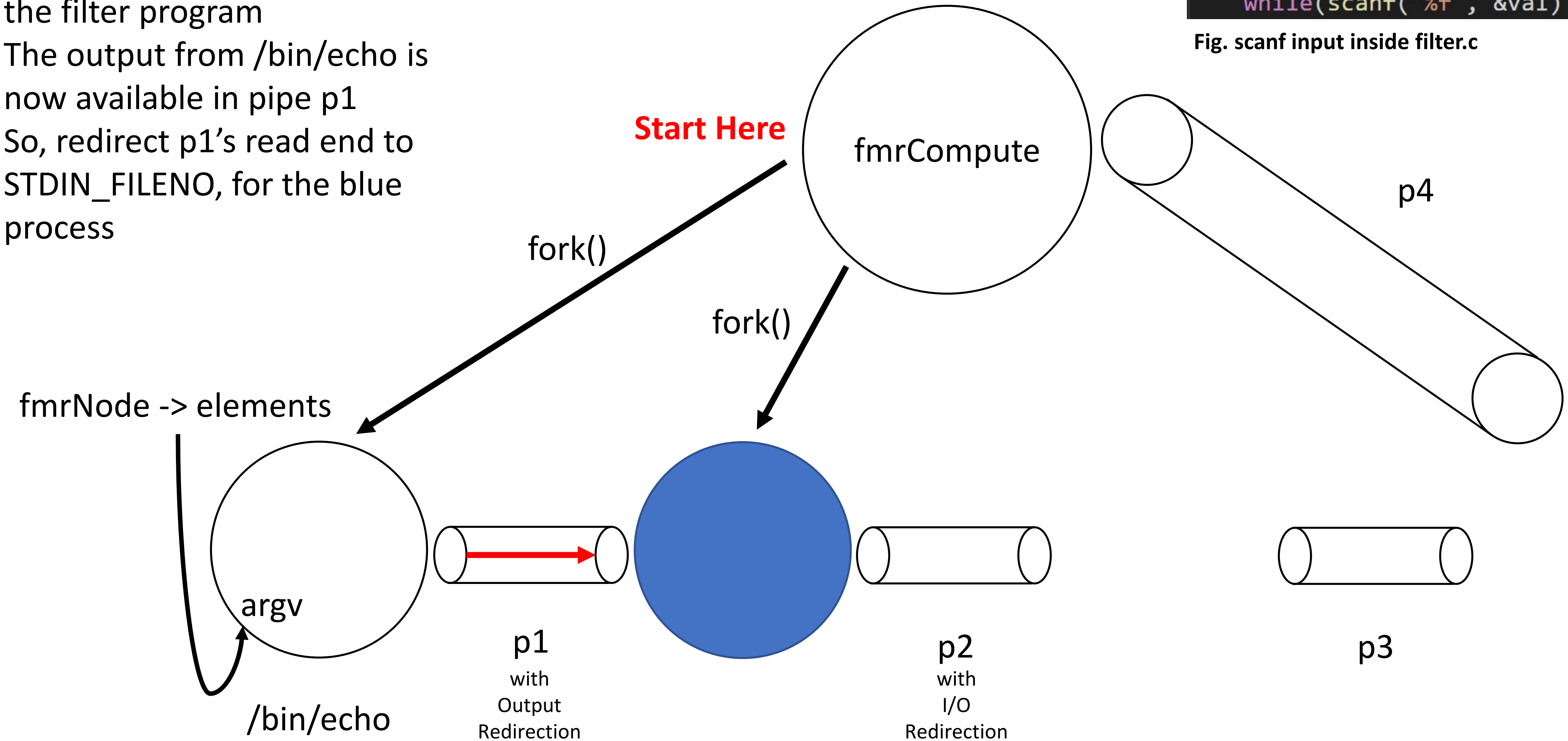
Lab Task Simulation (second child process)

- 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

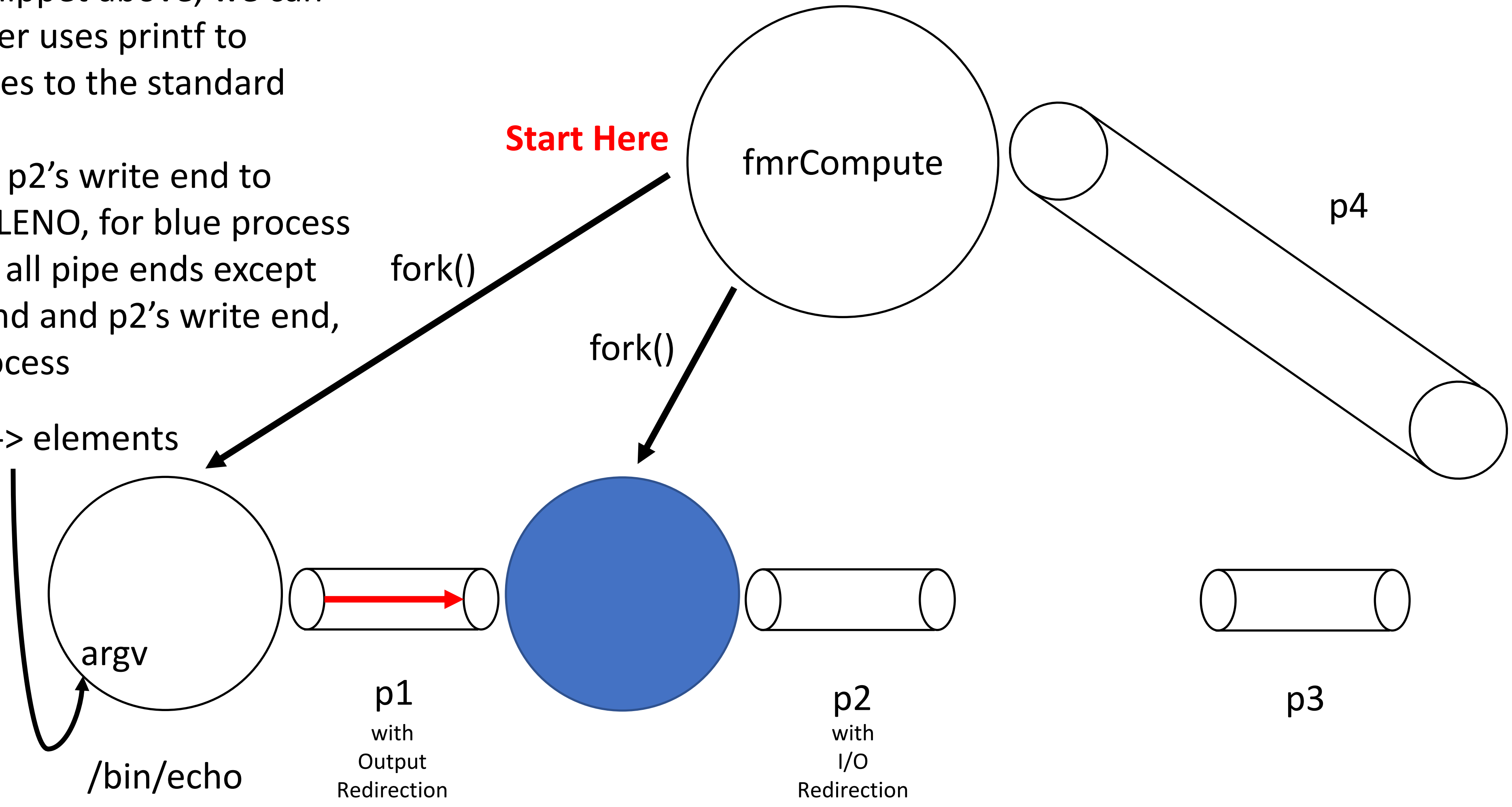


Lab Task Simulation (second child process)

```
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

- 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

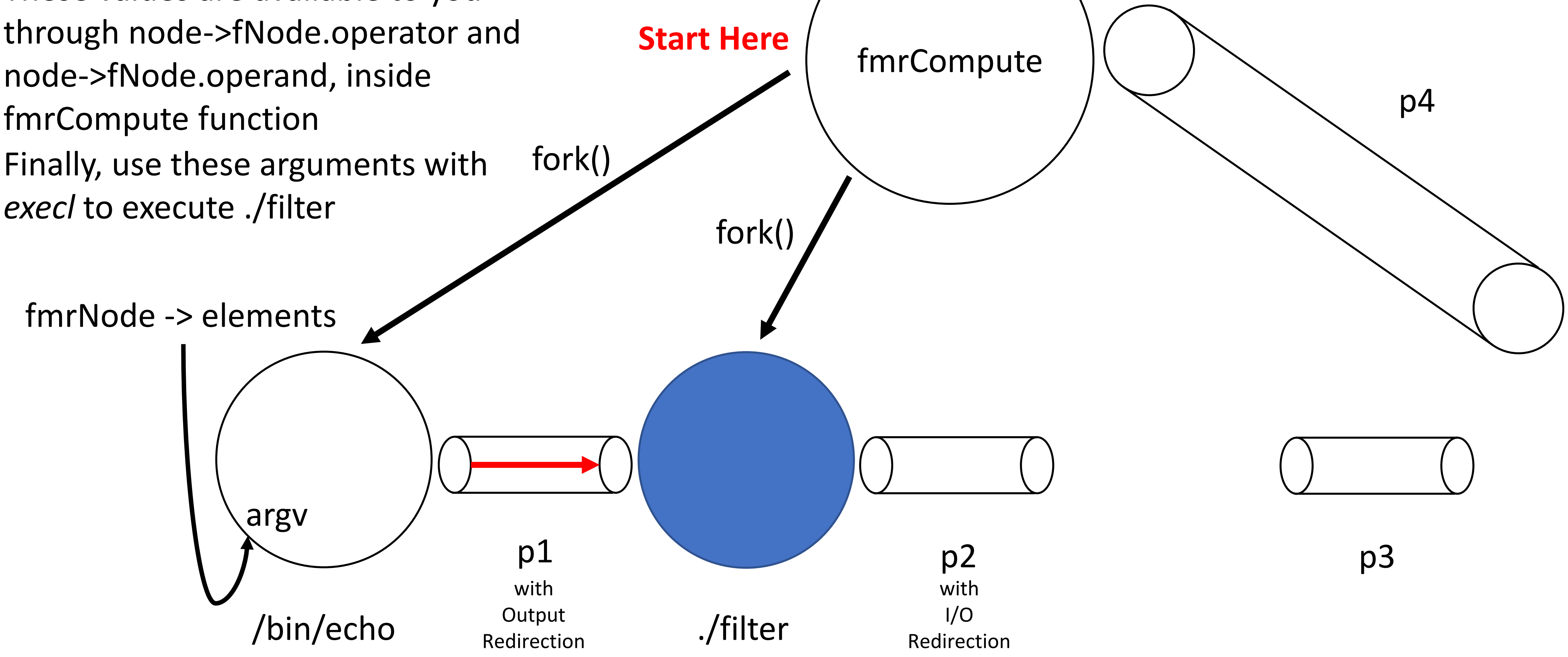


Lab Task Simulation (second child process)

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

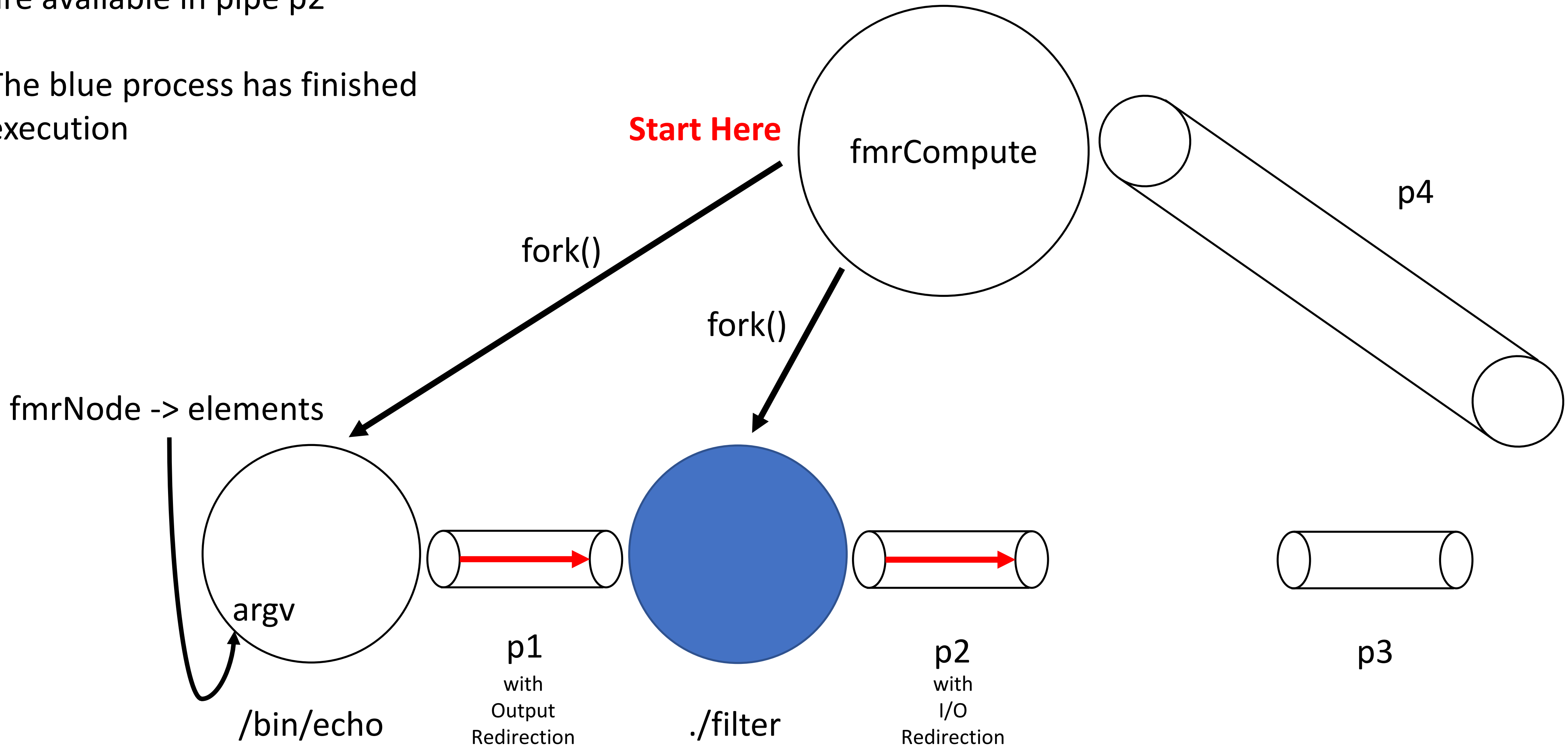
```c
int main(int argc, char* argv[]){
    char* comparingOperator = argv[1];
    float comparingOperand = atof(argv[2]);

    float val;

    while(scanf("%f", &val) != EOF){
```

Fig. argv arguments inside filter.c



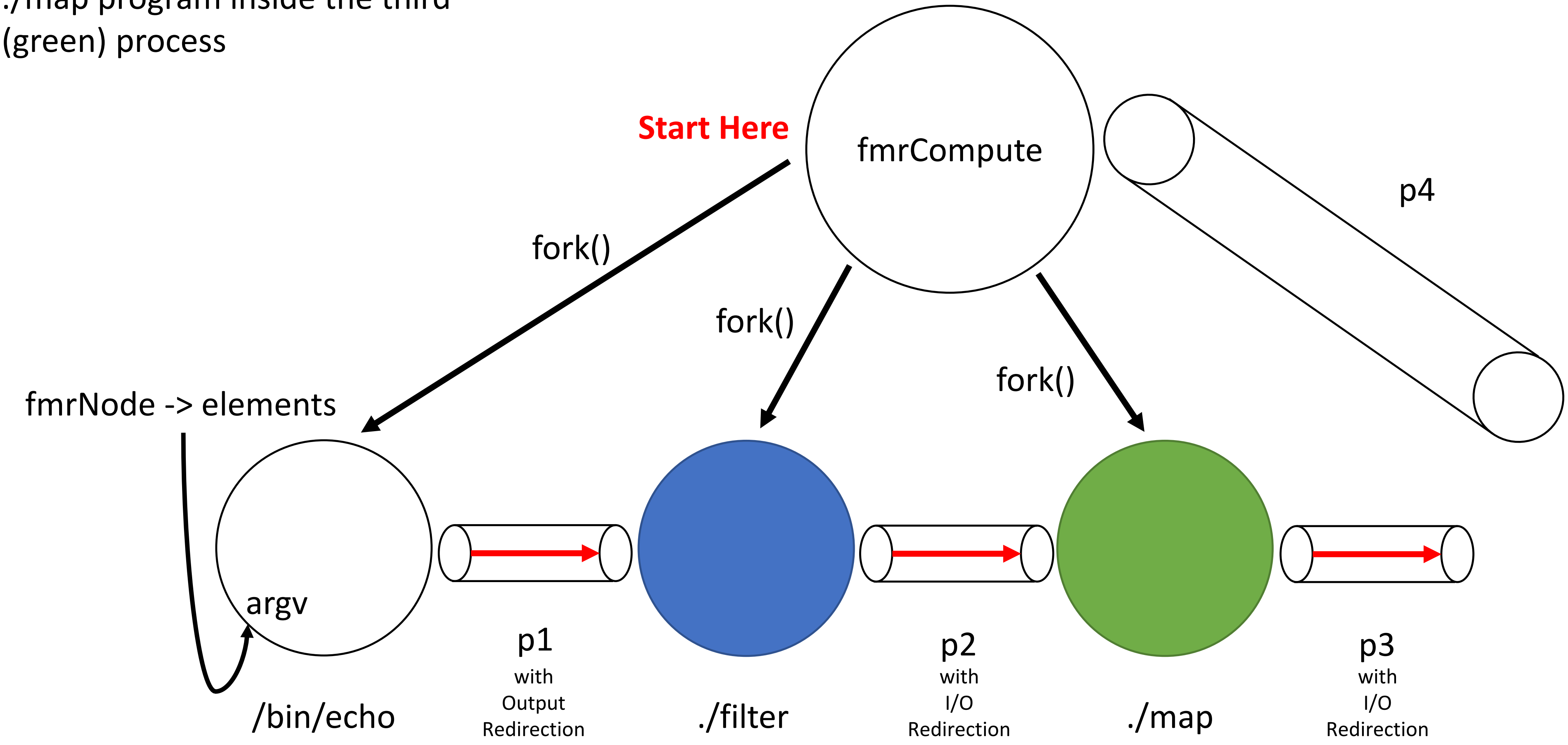
Lab Task Simulation (second child process)

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



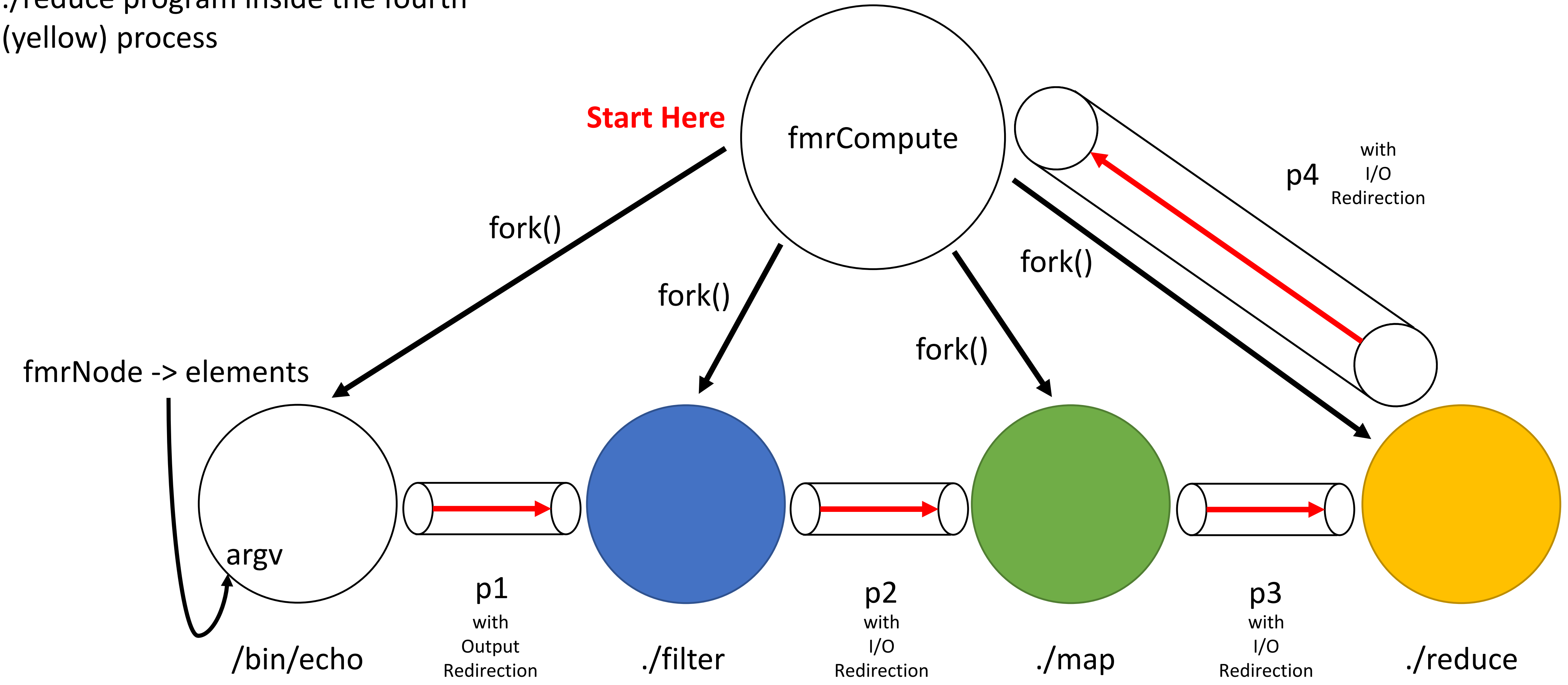
Lab Task Simulation (third child process)

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



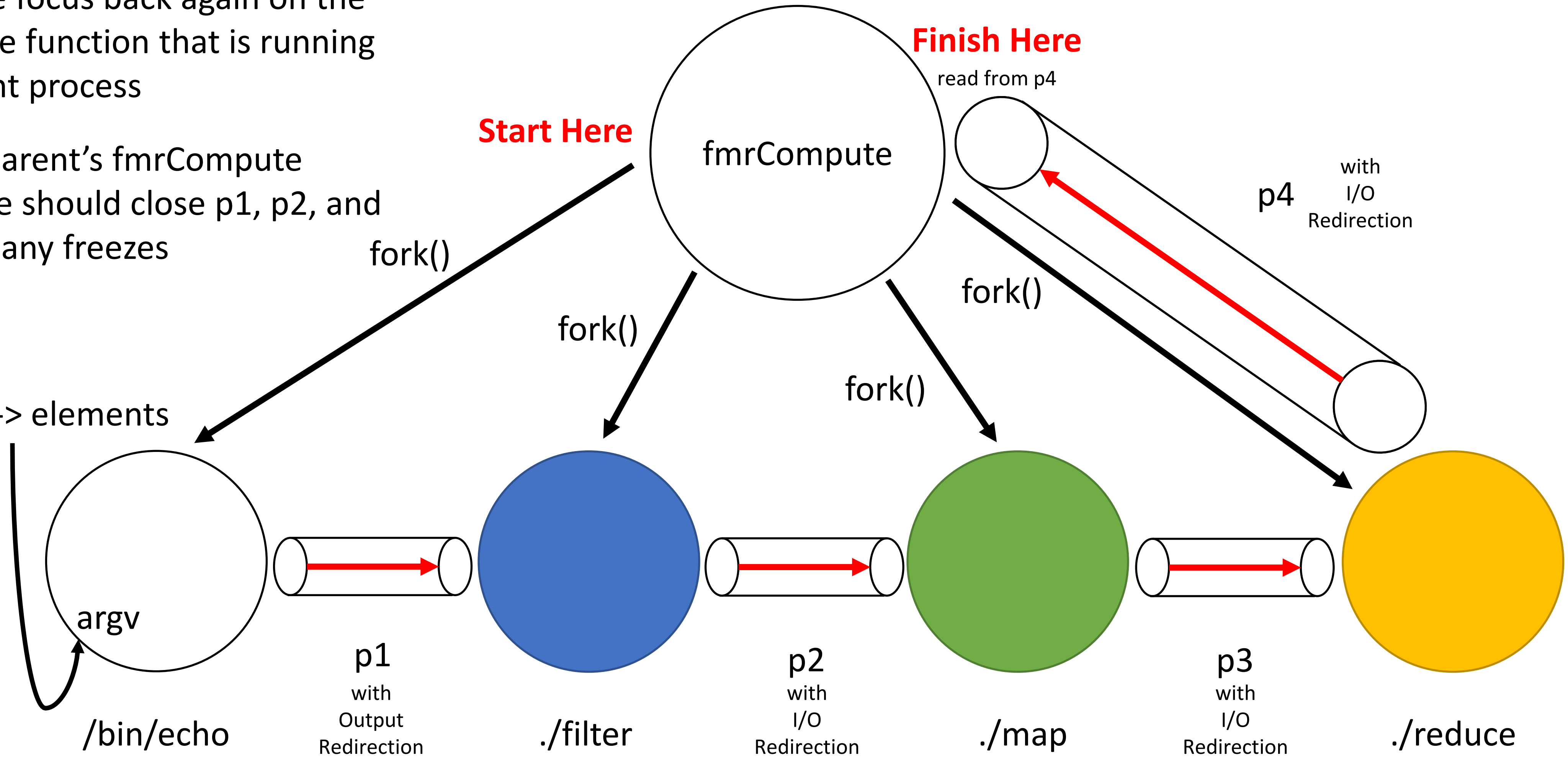
Lab Task Simulation (fourth child process)

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



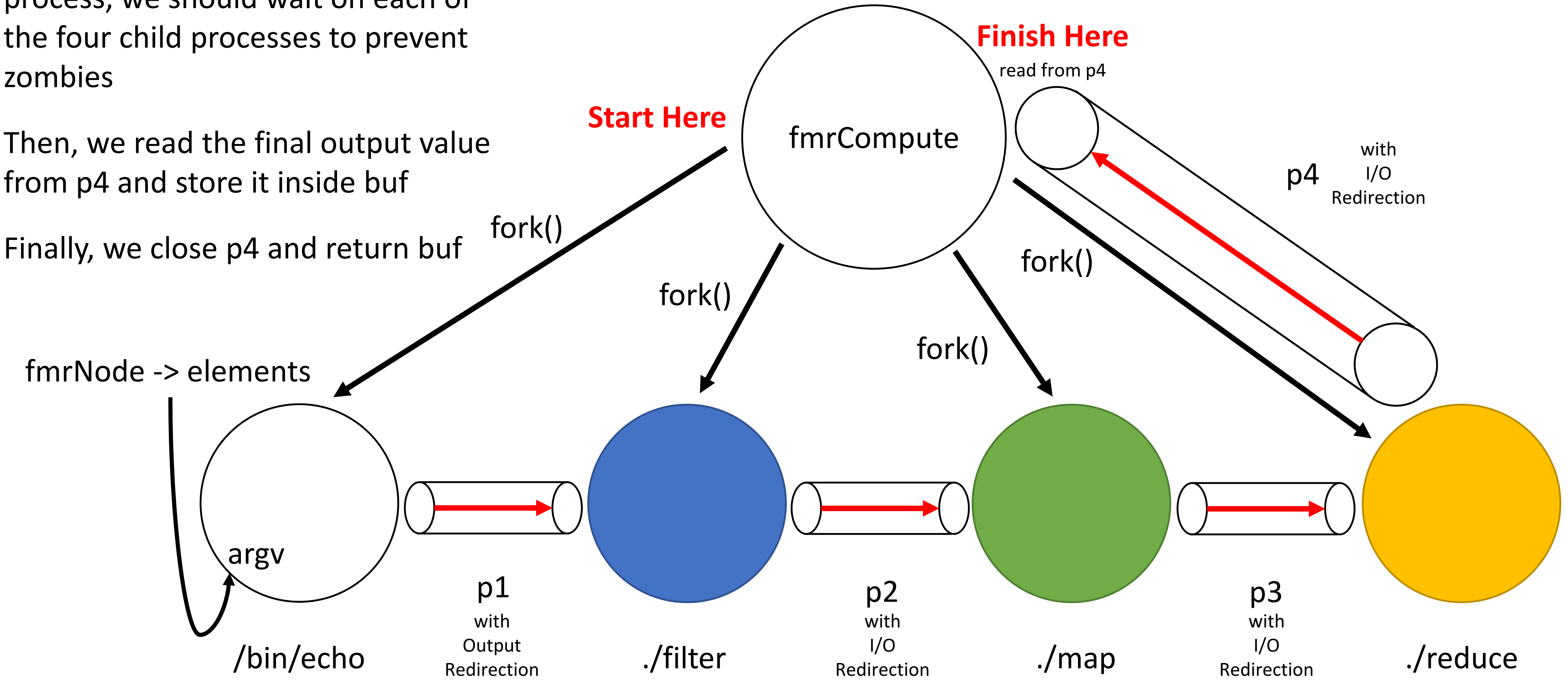
Lab Task Simulation (back to the parent)

- 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