CSC10007 - OPERATING SYSTEM

PROJECT 1 – Xv6 and Unix utilities

No	Name	Student ID	Contribute
1.	Le Quang Khai	22120148	50%
2.	Nguyen Quang Thang	22120333	50%

No.	Exercise	Person in charge
1	sleep	Le Quang Khai
2	pingpong	Le Quang Khai
3	primes	Nguyen Quang Thang
4	find	Le Quang Khai
5	xargs	Nguyen Quang Thang

1. Setup

1.1. Environment Configuration

For this series of labs, we will utilize Ubuntu 24.04 LTS on WSL2 (Windows Subsystem for Linux 2) to build and execute our projects.

1.2. Installing Dependencies

Installing WSL2

To install WSL2, open PowerShell or Command Prompt and enter the following command:

```
wsl --install
```

After the installation is complete, restart your machine to apply the changes.

Installing Ubuntu 24.04 LTS

List all available distributions

```
wsl --list --online
wsl -l -o
```

Choose one distribution and install

```
wsl --install -d < name >
wsl --install -d Ubuntu-24.04
```

Installing Required Software

Required software for xv6 labs are RISC-V versions of QEMU 7.2+, GDB 8.3+, GCC, and Binutils. Run the following commands to install them

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install git build-essential \
gdb-multiarch qemu-system-misc gcc-riscv64-linux-gnu binutils-riscv64-linux-g
```

1.3. Testing The Installation

To verify that the installation was successful, check the version of QEMU by running:

```
$ qemu-system-riscv64 --version
QEMU emulator version 8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.2)
Copyright (c) 2003-2023 Fabrice Bellard and the QEMU Project developers
```

Next, confirm that at least one of the following commands works:

```
$ riscv64-linux-gnu-gcc --version
$ riscv64-unknown-elf-gcc --version
$ riscv64-unknown-linux-gnu-gcc --version
```

A successful output for one of these commands should look like this:

```
$ riscv64-linux-gnu-gcc --version
riscv64-linux-gnu-gcc (Ubuntu 13.2.0-23ubuntu4) 13.2.0
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

1.4. Boot xv6

To fetch the XV6 source code for the lab, clone the Git repository using the following command:

```
$ git clone git://g.csail.mit.edu/xv6-labs-2024
Cloning into 'xv6-labs-2024'...
...
$ cd xv6-labs-2024
```

To build and run XV6, execute:

```
make qemu
```

To exit QEMU, type `Ctrl-a x.`

1.5. Compiling code

Code of `sleep` command should be placed in `user/sleep.c` .Then add the command name in `UPROGS` of `Makefile` :

```
UPROGS=\
    $U/_cat\
  ...
    $U/_sleep\
```

Then compile code by running `make qemu` .

1.6. Marking

Run one of the following command to test `sleep` task

```
$ ./grade-lab-util sleep
$ make GRADEFLAGS=sleep grade
```

2. sleep (easy)

> Implement a user-level `sleep` program for xv6, along the lines of the UNIX sleep command. Your `sleep` should pause for a user-specified number of ticks. A tick is a notion of time defined by the xv6 kernel, namely the time between two interrupts from the timer chip. Your solution should be in the file `user/sleep.c`.

Implementation Details

The calling prototype for the program is specified as follows:

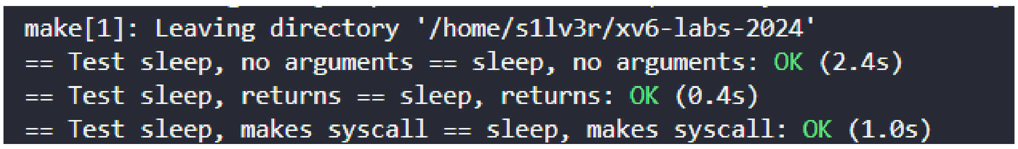
```
sleep NUMBER
```

- The program checks whether a numerical argument has been provided by the user. In the absence of an argument, a usage message is displayed, and the program exits with an exit code of `1`, indicating failure.
- The `NUMBER` argument will be passed as a string. It can be converted into an integer using the `atoi(NUMBER)` function in `user/ulib.c` file.
- Use `sleep()` system call declared in `user/user.h` to execute command.

Note:

`atoi` returns 0 for any negative input, therefore it is not necessary to explicitly check if the converted ticks value is negative.

Marking



```
make[1]: Leaving directory '/home/s1lv3r/xv6-labs-2024'
== Test sleep, no arguments == sleep, no arguments: OK (2.4s)
== Test sleep, returns == sleep, returns: OK (0.4s)
== Test sleep, makes syscall == sleep, makes syscall: OK (1.0s)
```

3. pingpong (easy)

> Write a user-level program that uses xv6 system calls to ''ping-pong'' a byte between two processes over a pair of pipes, one for each direction. The parent should send a byte to the child; the child should print `"<pid>: received ping"`, where `<pid>` is its process ID, write the byte on the pipe to the parent, and exit; the parent should read the byte from the child, print `“<pid>: received pong"`, and exit. Your solution should be in the file `user/pingpong.c`.

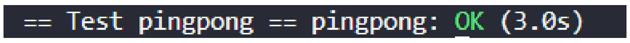
Implementation Details

The calling prototype for the program is specified as follows:

```
pingpong
```

- Create two pipes, one for transmitting data from the parent process to the child process, and another for sending data from the child process back to the parent process.
- Create a small `buffer` to read data from pipe(s).
- Reserve extra byte in `buffer` to prevent overflow. Always set last byte to `0` for NULL terminate.

Marking



```
== Test pingpong == pingpong: OK (3.0s)
```

4. primes (hard)

> Write a concurrent prime sieve program for xv6 using pipes and the design illustrated in the picture halfway down this page and the surrounding text.

> This idea is due to Doug McIlroy, inventor of Unix pipes. Your solution should be in the file `user/primes.c`.
>
> Your goal is to use pipe and fork to set up the pipeline. The first process feeds the numbers 2 through 280 into the pipeline. For each prime number, you will arrange to create one process that reads from its left neighbor over a pipe and writes to its right neighbor over another pipe. Since xv6 has limited number of file descriptors and processes, the first process can stop at 280.

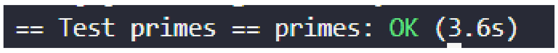
Implementation Details
The calling prototype for the program is specified as follows:

```
primes [NUMBER]
```

By default, `NUMBER` has value of 280.

- The `sieve(int in_fd, int first_prime)` function is designed to read data from the pipe of previous step, apply the sieve algorithm using the prime number `first_prime`, and then send the filtered data into a new pipe. It forks itself into two processes, one process returns `file descriptor` of new pipe for use in next step, while the other process performs the sieving of numbers from the previous pipe.
- The naive approach is pushing all 280 values into first pipe, and then call `sieve()` function to handle numbers (as in **Appendix**). However, this method will not work since 280 is large and pipe's size is only about `120 * sizeof(int)`, it is impossible to push all values into the pipe before processing. Therefore, another `fork` is implement here to create two processes: one for inserting values from 2 to 280 into the pipe, while the other retrieves and processes the data.

Marking



```
== Test primes == primes: OK (3.6s)
```

5. find (moderate)

> Write a simple version of the UNIX find program for xv6: find all the files in a directory tree with a specific name. Your solution should be in the file `user/find.c`.

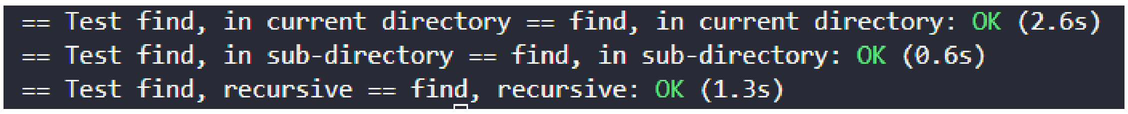
Implementation Details
The calling prototype for the program is specified as follows:

```
find PATH FILENAME
```

The program is implemented using the file system and commands such as `open`, `stat`, `read`, and `close` to traverse directories and compare file names.

- The `fmtname` function takes a path as input and returns the final file name from that path. It finds the file name after the last `'/'` and returns the formatted file name in a buffer. The `find` function is responsible for searching for a file that matches a specific name within the directory tree starting from the specified path. This function operates by opening the file or directory using the `open()` command, then checking the type of the object (file or directory) using the `fstat()` command.
    - If the object is a file, the function uses the helper function `fmtname()` to retrieve the file name from the path and compares it with the target file name (passed as the argument `filename`). If the two names match, the program prints the full path of that file.
    - If the object is a directory, the function reads each file and subdirectory inside it using the `read()` command and recursively calls itself to continue searching in the subdirectories. This process continues until all files and subdirectories have been traversed. The function also checks for and skips special directories like `.` and `..` to avoid repeating the current directory and the parent directory.

Marking



```
== Test find, in current directory == find, in current directory: OK (2.6s)
== Test find, in sub-directory == find, in sub-directory: OK (0.6s)
== Test find, recursive == find, recursive: OK (1.3s)
```

6. xargs (moderate)

> Write a simple version of the UNIX `xargs` program for xv6: its arguments describe a command to run, it reads lines from the standard input, and it runs the command for each line, appending the line to the command's arguments. Your solution should be in the file `user/xargs.c`.

Implementation Details

The calling prototype for the program is specified as follows:

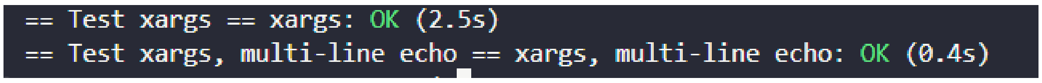
```
COMMAND_1 ARG_1_1 ARG_1_2 ... | xargs COMMAND_2 ARG_2_1 ARG_2_2 ...
```

- Result of `COMMAND_1` is in standard input stream. It will be read later in the program.
- `xargs` program will take `argv` is a two-dimension pointer `["xargs", "COMMAND_2", "ARG_2_1", "ARG_2_2", ... ]`.
- A new argument pointer, `args` is created:
    - `args[i]` points to `argv[i]` , for all `i < argc` .
    - After that, `args[i]` will point to extra arguments passed through pipe.

- A `read_buffer` is allocated so it can read faster. Instead of reading byte by byte from pipe, the program reads `ARRAY_MAX_LENGTH` bytes at once.
- When the iterator meets space detected by `is_space()` function, it marks and create new arguments in `args` , and skip sequence of spaces (if any).

> **Note**
>
> Programmer might meet wrong approach that convert all arguments into string (one-dimension array of `char`) and declare `args` with size of `3` , as showed in **Appendix**. It will not pass the test case.

Marking



```
== Test xargs == xargs: OK (2.5s)
== Test xargs, multi-line echo == xargs, multi-line echo: OK (0.4s)
```

References

1. https://learn.microsoft.com/en-us/windows/wsl/install
2. https://pdos.csail.mit.edu/6.1810/2024/tools.html
3. https://pdos.csail.mit.edu/6.1810/2024/labs/util.html

Appendix

```
// user/xargs.c, wrong approach

#include "kernel/types.h"
#include "kernel/param.h"
#include "user/user.h"

void
xargs(int argc, char ** argv);

int
main(int argc, char * argv[]) {
  if (argc < 2) {
    fprintf(2, "Usage: xargs COMMAND\n");
    exit(1);
  }

  // Remove "xargs" from argc and argv, and execute command
  xargs(argc - 1, argv + 1);
  exit(0);
}
```

```
int
convert_args_to_string(char * des, int DES_MAX_LENGTH,
                int argc, char ** argv);

void
xargs(int argc, char ** argv) {
  const int MAX_CMD_LENGTH = MAXARG * MAXARG;
  char cmd[MAX_CMD_LENGTH + 1];
  // convert only content of command into char*
  int CMD_POS = convert_args_to_string(cmd, MAX_CMD_LENGTH,
                            argc - 1, argv + 1);

  // Prepare the command to be executed.
  // `args` will be fed into `exec` function
  char * args[3];
  args[0] = argv[0];
  args[1] = cmd; // Set the command arguments
  args[2] = 0; // null termination

  // Buffer to read from pipe.
  // Iterate in array is faster than read byte by byte from pipe.
  const int READ_BUFFER_SIZE = MAXARG * MAXARG;
  char read_buffer[READ_BUFFER_SIZE + 1];

  // Read args from pipe and append to (cmd + cmd_pos) position
  int current_cmd_pos = CMD_POS;
  int bytes_read;
  while ((bytes_read = read(0, read_buffer, READ_BUFFER_SIZE)) > 0) {
    for (int i = 0; i < bytes_read; ++i) {
      // '\n' marks the end of one command
      if (read_buffer[i] != '\n') {
        cmd[current_cmd_pos++] = read_buffer[i];
        if (current_cmd_pos >= MAX_CMD_LENGTH) {
          fprintf(2, "error: command too long\n");
          exit(1);
        }
        continue;
      }

      cmd[current_cmd_pos] = 0; // Null-terminate the command string

      // Create a new process to execute command
      if (fork() == 0) {
        exec(argv[0], args);
        exit(0);
      } else {
        wait(0);
      }
      // return to postion to write new command
      current_cmd_pos = CMD_POS;
    }
  }

  exit(0);
}

int
convert_args_to_string(char * des, int DES_MAX_LENGTH,
            int argc, char ** argv) {
  int total_length = 0;

  // Check if the total length of arguments exceeds the maximum allowed
  for (int i = 0; i < argc; ++i) {
    // +1 for the space after each argument
    total_length += strlen(argv[i]) + 1;
  }
  if (total_length >= DES_MAX_LENGTH) {
    fprintf(2, "error: too long arguments\n");
    exit(1);
  }

  // Copy the arguments into the destination buffer
  total_length = 0; // Reset total_length to use it as a position tracker
  for (int i = 0; i < argc; ++i) {
    strcpy(des + total_length, argv[i]);
    total_length += strlen(argv[i]);
    des[total_length++] = ' ';
  }
  des[total_length] = 0; // Null-terminate
  return total_length;
}
```

```
// user/primes, wrong approach

#include "kernel/types.h"
#include "user/user.h"

void primes(int fd) __attribute__((noreturn));

int main(int argc, char* argv[]) {
    int THRESHOLD = atoi(argv[1]);
    printf("THRESHOLD = %d\n", THRESHOLD);
    int p[2];
    pipe(p); // Create the initial pipe

    // Write numbers from 2 to THRESHOLD to the pipe
    for (int i = 2; i <= THRESHOLD; ++i) {
        write(p[1], &i, sizeof(int));
    }
    close(p[1]); // Close the write end after writing all numbers
    primes(p[0]); // Start processing primes
    close(p[0]); // Close read end in the main process
    exit(0);
}

void primes(int fd) {
    int p[2];
    pipe(p); // Create a new pipe for the next layer of primes

    // Read the first prime
    int current_prime;
    if (read(fd, &current_prime, sizeof(int)) == 0) {
        // If there's nothing to read, exit
        close(p[0]);
        close(p[1]);
        exit(0);
    }

    printf("prime %d\n", current_prime);

    int num;
    // Read numbers from the previous pipe and filter
    while (read(fd, &num, sizeof(int)) > 0) {
        if (num % current_prime != 0) {
            // Pass primes to the next layer
            write(p[1], &num, sizeof(int));
        }
    }

    // Close write end after all numbers are filtered
    close(p[1]);

    // Create the next process
    if (fork() == 0) {
        close(fd);
        primes(p[0]); // Process the next layer of primes
        close(p[0]); // Close read end after processing
    } else {
        close(p[0]); // Close read end in the parent
        close(fd); // Close read end of the previous pipe
        wait(0); // Wait for the child process to finish
    }

    exit(0);
}
```