lab0 题

实验练习 1 补全 log_stdout.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include <stdarg.h>

char buf[1024];

int read_stdin(char* buf) {
    /*
    Description: Read stdin into buf
    Example:
        - read_stdin(); // Read the stdin into buf
    Parameters:
        - buf (char*): A buffer to store all characters
    Return:
        - 0 (int)
    */
    // Your code here
    // End
    return 0;
}

int log_stdout(uint i) {
    /*
    Description: Redirect stdout to a log file named i.log
```

```
    Example:
        - log_stdout(1); // Redirect the stdout to 1.log and return 0
    Parameters:
        - i (uint): A number
    Return:
        - 0 (int)
    */
    char log_name[15] = "0.log";
    // Your code here
    // End
    return 0;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        fprintf(2, "Usage: log_stdout number\n");
        exit(1);
    }
    if (log_stdout(atoi(argv[1])) != 0) {
        fprintf(2, "log_stdout: log_stdout failed\n");
        exit(1);
    }
    if (read_stdin(buf) != 0) {
        fprintf(2, "log_stdout: read_stdin failed\n");
        exit(1);
    }
    printf(buf);
    exit(0);
}
```

实验练习 2 补全 composites.c

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "kernel/fcntl.h"
#include "user/user.h"
#include <stdarg.h>

int log_stdout(uint i) {
    /*
    Description: Redirect stdout to a log file named i.log.
    Example:
        - log_stdout(1); // Redirect the stdout to 1.log and return 0.
    Parameters:
        - i (uint): A number
    Return:
        - 0 (int)
    */
    char log_name[15] = "0.log";
    // Your code here
    uint base = 1, i_temp;
    if (i != 0) {
        for (base = 0, i_temp = i; i_temp != 0; ++base, i_temp /= 10);
        for (uint base_temp = 0, i_temp = i; i_temp != 0; ++base_temp, i_temp /= 10) {
            log_name[base - base_temp - 1] = '0' + i_temp % 10;
        }
        strcpy(log_name + base, ".log");
    }
    close(1);
    if (open(log_name, O_CREATE | O_WRONLY) != 1) {
```

```
        fprintf(2, "log_stdout: open failed\n");
        return -1;
    }
    // End
    return 0;
}

void sub_process(int p_left[2], int i) {
    /*
    Description:
        - Pseudocode:
            prime = get a number from left neighbor
            print prime m
            loop:
                m = get a number from left neighbor
                if (p does not divide m)
                    send m to right neighbor
                else
                    print composite m
        - Be careful to close file descriptors that a process doesn't need, because otherwise your program will run xv6 out of resources before the first process reaches 35.
        - Hint: read returns zero when the write-side of a pipe is closed.
        - It's simplest to directly write 8-bit (1-byte) chars to the pipes, rather than using formatted ASCII I/O.
        - Use pipe and fork to recursively set up and run the next sub_process if necessary
        - Once the write-side of left neighbor is closed, it should wait until the entire pipeline terminates, including all children, grandchildren, &c.
    Example:
        - sub_process(4); // Run the 4th sub_process.
    Parameters:
```

```
    - i (int): A number
  Return:
    - (void)
  */
  if (log_stdout(i) < 0) {
    fprintf(2, "composites: log_stdout %d failed\n", i);
    exit(1);
  }
  char m, prime;
  int num_read, p_right[2], pid = 0;
  // prime = get a number from left neighbor
  // End
  printf("prime %d\n", prime);
  while (1) {
    // m = get a number from left neighbor
    // End
    // Use pipe and fork to recursively set up and run the next sub_process if necessary
    // End
    if (m % prime != 0) {
      // send m to right neighbor
      // End
    }
    else {
      printf("composite %d\n", m);
    }
  }
  // Once the write-side of left neighbor is closed, it should wait until the entire pipeline terminates, including all children, grandchildren, &c.
  // End
  exit(0);
```

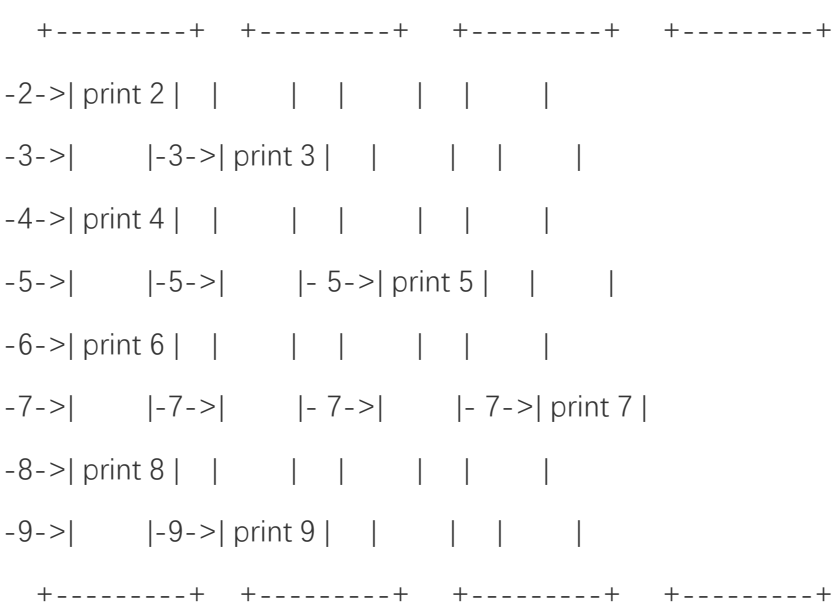
}

void composites() {

/*

Description:

- A generating process can feed the numbers 2, 3, 4, ..., 35 into the left end of the pipeline: the first process in the line eliminates the multiples of 2, the second eliminates the multiples of 3, the third eliminates the multiples of 5, and so on:



- Be careful to close file descriptors that a process doesn't need, because otherwise your program will run xv6 out of resources before the first process reaches 35.

- Once the first process reaches 35, it should wait until the entire pipeline terminates, including all children, grandchildren, &c. Thus the main composites process should only exit after all the output has been printed, and after all the other composites processes have exited.

- You should create the processes in the pipeline only as they are needed.

Example:

- sub_process(4); // Run the 4th sub_process.

Parameters:

Return:

- (void)

*/

```
    int p_right[2], pid, i = 0;
    // Use pipe and fork to recursively set up and run the first sub_process
    // End
    // The first process feeds the numbers 2 through 35 into the pipeline.
    // End
    // Once the first process reaches 35, it should wait until the entire pipeline terminates, including all children, grandchildren, &c. Thus the main primes process should only exit after all the output has been printed, and after all the other primes processes have exited.
    // End
    exit(0);
}

int main(int argc, char* argv[]) {
    if (argc != 1) {
        fprintf(2, "Usage: composites\n");
        exit(1);
    }
    composites();
    exit(0);
}
```

实验练习 3 xargs.c

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.

The following example illustrates xarg's behavior: `$ echo hello too | xargs echo bye` `bye hello too` `$` Note that the command here is "echo bye" and the additional arguments are "hello too", making the command "echo bye hello too", which outputs "bye hello too".Please note that xargs on UNIX makes an optimization where it will feed more than argument to the command at a time. We don't expect you to make this optimization. To make xargs on UNIX behave the way we want it to for this lab, please run it with the -n option set to 1. For instance

```
$ (echo 1 ; echo 2) | xargs -n 1 echo
1
2
$
```

Some hints:

- Use fork and exec to invoke the command on each line of input. Use wait in the parent to wait for the child to complete the command.
- To read individual lines of input, read a character at a time until a newline ('\n') appears.
- kernel/param.h declares MAXARG, which may be useful if you need to declare an argv array.
- Add the program to UPROGS in Makefile.
- Changes to the file system persist across runs of qemu; to get a clean file system run make clean and then make qemu.

xargs, find, and grep combine well:

$ **find . b | xargs grep hello** will run "grep hello" on each file named b in the directories below ".".To test your solution for xargs, run the shell script xargstest.sh. Your solution is correct if it produces the following output:

$ **make qemu** ... init: starting sh $ **sh < xargstest.sh** $ $ $ $ $ $ hello hello hello $ $