xv6_lab1实验报告

环境搭建

关于xv6的实验环境搭建,使用linux的话,环境搭建属实简单,直接跟着官方文档操作即可

---

exercise解题思路

`sleep` 解题思路

首先贴出程序:

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

int main(int argc, char** argv) {
  if (argc != 2) {
    fprintf(2, "usage: sleep times(ms)\n");
    exit(-1);
  }

  int times = atoi(*(++argv));

  if (times == 0) {
    fprintf(2, "times can not less than or equal 0\n");
    exit(-1);
  }

  sleep(times);
  return 0;
}
```

关于 `sleep` 的exercise没有太大难度,主要是接受参数,然后使用 `atoi` 进行转换并判断从而成功调用系统调用 `sleep`

`pingpong` 解题思路

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

#define MSG "ping"
#define REPLY "pong"
```

```
int main(int argc, char *argv[]) {
  int pipe1[2]; // parent -> child
  int pipe2[2]; // child -> parent
  char buf[10];

  if (pipe(pipe1) < 0 || pipe(pipe2) < 0) {
    fprintf(2, "pipe failed\n");
    exit(1);
  }
  int pid = fork();

  if (pid < 0) {
    fprintf(2, "fork failed\n");
    exit(1);
  } else if (pid == 0) {
    // child process
    close(pipe1[1]); // close write end of pipe1
    close(pipe2[0]); // close read end of pipe2

    // send message and receive message
    read(pipe1[0], buf, sizeof(buf));
    printf("[child] received: %s\n", buf);

    write(pipe2[1], REPLY, strlen(REPLY) + 1); // send pong
    printf("[child] sent: %s\n", REPLY);

    close(pipe1[0]);
    close(pipe2[1]);
    exit(0);
  } else {
    // parent process
    close(pipe1[0]); // close read end of pipe1
    close(pipe2[1]); // close write end of pipe2

    printf("[parent] sent: %s\n", MSG);
    write(pipe1[1], MSG, strlen(MSG) + 1); // send ping

    // Wait for the child to finish
    wait(0);

    read(pipe2[0], buf, sizeof(buf));
    printf("[parent] received: %s\n", buf);
```

```
    close(pipe1[1]);
    close(pipe2[0]);
  }
  return 0;
}
```

`pingpong` 主要是考查 `fork` 以及 `pipe` 的配合使用.(初步思考)

1. `fork` 会创建出*child process*,并且会返回两次,一次在父进程中返回子进程的PID,另一次在子进程中返回0.*child process*还会对父进程拥有的**system resource**进行copy
2. `pipe` 会创建出一个管道,并且返回两个文件描述符,一个用于读,一个用于写.管道是**Unidirectional**的,也就是说数据只能在一个方向上流动.因此,我们可以创建两个管道实现双向流通
3. `buf` 是一个**缓冲区buffer**,用于存储从管道中读取的数据

在xv6中运行得到以下结果:

```
[parent] sent: ping
[child] received:
[child] sent: pong
[parent] received: pong
```

> **Question**
>
> 这个结果我没有想明白为什么child process没有received `ping` 即child process的 `buf` 为空

而且如果代码中的:

```
    printf("[parent] sent: %s\n", MSG);
    write(pipe1[1], MSG, strlen(MSG) + 1); // send ping
```

这两行进行调转,还会出现输出乱码,如下:

```
[pare[cnhti]ld ] sreencte:i vepdi:n g

[child] sent: pong
[parent] received: pong
```

这个的问题应该就是:**child process**和**parent process**的一个关于I/O的并发问题,就是出现了 `race` ,我认为解决方案是在I/O输出前加入一个 `buffer` 提高性能同时解决问题.

`primes` 解题思路

1. **版本一**

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

void new_proc(int p[2]) __attribute__((noreturn));

// Implement the recursion of the process
void new_proc(int p[2]) {
  int prime;
  int n;

  close(p[1]);

  // If read returns 0, it means the previous place closed the write pipe
  // and there are no more numbers to process. So we can exit.
  if (read(p[0], &prime, sizeof(int)) == 0) {
    close(p[0]);
    exit(0);
  }

  // The first number is prime.
  printf("prime %d\n", prime);

  int fd[2];
  pipe(fd);

  int pid = fork();

  if (pid < 0) {
    fprintf(2, "fork failed\n");
    close(p[0]);
    close(fd[0]);
    close(fd[1]);
    exit(1);
  } else if (pid == 0) {
    // Child process of the child process
    close(p[0]);
    new_proc(fd);
  } else {
    // Child process
    close(fd[0]);

    // Read numbers from input pipe and filter them.
    while (read(p[0], &n, sizeof(int)) > 0) {
      if (n % prime != 0) { // Exclude the multiple of the prime
```

```
        write(fd[1], &n, sizeof(int)); // Pass it to the next place.
      }
    }

    close(p[0]);
    close(fd[1]);
    wait(0);
    exit(0);
  }
}

int main(int argc, char** argv) {
  int p[2];
  pipe(p);

  int pid = fork();

  if (pid < 0) {
    fprintf(2, "fork failed\n");
    exit(1);
  } else if (pid == 0) {
    // Child process
    new_proc(p);
  } else {
    // Parent process (main)
    close(p[0]);

    for (int i = 2; i <= 280; i++) {
      write(p[1], &i, sizeof(int));
    }

    close(p[1]);
    wait(0);
  }
  return 0;
}
```

Question

这个版本在编译器出现 `infinite recursion` 的时候,根据hints,添加了 `__attribute__((noreturn))` . 编译是成功编译了,但是在xv6中运行输出只有 `prime 2` 一个素数,其他素数都没有输出,我不知道问题出现在哪里?

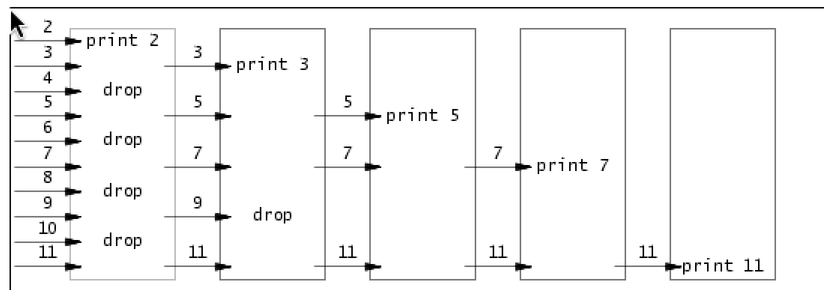
最后,根据ai给出的,找到另一个hacky solution,再引入一个函数进行封装即可

2. **版本二**

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

void new_proc(int p[2]);

// To avoid the CFLAGS in Makefile(warnning = error)
void child_branch(int p[2]) {
  new_proc(p);
}

// Implement the recursion of the process
void new_proc(int p[2]) {
  int prime;
  int n;

  close(p[1]);

  // If read returns 0, it means the previous place closed the write pipe
  // and there are no more numbers to process. So we can exit.
  if (read(p[0], &prime, sizeof(int)) == 0) {
    close(p[0]);
    exit(0);
  }

  // The first number is prime.
  printf("prime %d\n", prime);

  int fd[2];
  pipe(fd);

  int pid = fork();

  if (pid < 0) {
    fprintf(2, "fork failed\n");
    close(p[0]);
    close(fd[0]);
    close(fd[1]);
    exit(1);
  } else if (pid == 0) {
    // Child process of the child process
    close(p[0]);
    child_branch(fd);
```

```
  } else {
    // Child process
    close(fd[0]);

    // Read numbers from input pipe and filter them.
    while (read(p[0], &n, sizeof(int)) > 0) {
      if (n % prime != 0) { // Exclude the multiple of the prime
        write(fd[1], &n, sizeof(int)); // Pass it to the next place.
      }
    }

    close(p[0]);
    close(fd[1]);
    wait(0);
    exit(0);
  }
}

int main(int argc, char** argv) {
  int p[2];
  pipe(p);

  int pid = fork();

  if (pid < 0) {
    fprintf(2, "fork failed\n");
    exit(1);
  } else if (pid == 0) {
    // Child process
    new_proc(p);
  } else {
    // Parent process (main)
    close(p[0]);

    for (int i = 2; i <= 280; i++) {
      write(p[1], &i, sizeof(int));
    }

    close(p[1]);
    wait(0);
  }
  exit(0);
}
```

这个素数筛其实就是 埃拉托斯特尼筛法 的算法的思路,但是通过融合os的 `fork` 和 `pipe` 从分体现了os进程管理和进程间通信,以及os的魅力

思路其实就是这张图: 

实验主观心得

这次的实验算是比较有趣的, `pingpong` 以及 `primes` 的练习让我对 `fork` 和 `pipe` 有了更深的理解,尤其是 `primes` 的实现,让我体会到了os的进程管理和进程间通信的魅力.

管道这个一开始我对他的了解是在 `unix传奇` 这本书里讲到关于unix的历史发展过程. 这本书提及到unix设计时,有关于命令行工具的构想中,提及到可以利用水管这个具象化的东西来类比数据流动,从而设计出管道这个概念,让我对管道有了更深的理解.

通过这次练习的实践,也有了更深的认识.

关于 `fd(file descriptor)` 的理解,是在我将linux当为主力系统的时候,有了一定的认识的. 后面,有幸阅读到这篇blog后,对于 `fd` 有了更深的理解.

关于AI,这次的练习,上面出现的问题,AI没有给出我一个让我信服的解释,只给出一些思路去解决问题. 感觉AI也有一种心情指标,在某些场合,某些时间点上,我只需要简单一问,AI就可以给出让我信服的解释和解决方案(但是这个感觉是小概率事件😆) 不过总体而言,AI算是一个好帮手,在我编写 `primes` 练习的代码的时候,我写出个大纲之后,一些修修补补可以通过AI进行提示和解决. 就算写不出一个初步的代码,将你的大致思路交给AI之后也可以进行后期的处理🤔

这几个练习,在我一开始阅读 `xv6` 的时候,对于 `fork` 和 `pipe` 的理解只停留在概念层上,但是在我越往后阅读之后,加上这次的练习,让我对于 `fork` 和 `pipe` 算是有了更深的理解(算是起码能够写出代码了😄)

基本阅读完xv6之后再看这一章,真的是觉得 `xv6` 可以用一句话形容:**麻雀虽小五脏俱全**,自己也想写出这样的小型系统内核呢?🤔😮‍💨