**TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**



**BÁO CÁO ĐỒ ÁN 1**

**PROJECT 1 – XV6 AND UNIX UTILITIES**

**Môn học: Hệ điều hành**

**Lớp: CQ2022/4**

**GVLT:** Trần Trung Dũng

**GVTH:** Nguyễn Thanh Quân, Chung Thùy Linh

**Sinh viên thực hiện:**

Trương Thị Tú My – 22120214

Đinh Tấn Nhật – 22120251

Đào Ngọc Thảo Nguyên - 22120234

Mục lục

[**1.** **Pingpong** 2](#_Toc179483076)

[**2.** **Primes** 2](#_Toc179483077)

[**2.1 Đạt được** 2](#_Toc179483078)

[**2.2 Mô tả source** 2](#_Toc179483079)

[**2.3 Khó khăn** 2](#_Toc179483080)

[**3.** **Find** 3](#_Toc179483081)

[**4.** **Xargs** 6](#_Toc179483082)

# **Pingpong**

## **1.1 Đạt được**

* **Biết cách sử dụng lệnh fork()**: Tạo tiến trình con thông qua cơ chế clone từ tiến trình cha.
* **Biết cách sử dụng lệnh pipe(int p[])**: Tạo đường ống (pipeline), đặt file mô tả (file descriptors) lần lượt vào p[0] và p[1].
* **Biết cách sử dụng getpid, read, write, wait**.

## **2.2 Mô tả source**

* pipe và các biến(check, b) được khởi tạo.

int p[2];

pipe(p); // check là số giao tiếp giữa hai ống

int check = 1, b = 12; // b là số thêm vào tránh trường hợp tự ghi tự đọc

* Tạo tiến trình con và thực hiện theo thứ tự: Tiến trình cha: ghi số vào đầu ghi, chờ tiến trình con, kiểm tra số nhận được từ tiến trình con. Tiến trình con: đọc số từ đầu ghi, báo hiệu nhận thành công, ghi số vừa nhận được cộng thêm b vào đầu ghi của pipe.

if (fork()) {

write(p[1], &check, sizeof(int)); // ghi số vào ống

wait(0); // chờ tiến trình con nhận và gửi hoàn tất

int tmp;

read(p[0], &tmp, sizeof(int)); // đọc số từ ống

if (tmp == check + b) { // kiểm tra có lỗi khi nhận không

printf("%d: received pong\n", getpid());

} else {

printf("%d: received error\n", getpid());

}

} else {

int tmp = 0; // Khởi tạo để tránh lỗi khi cộng và ghi

read(p[0], &tmp, sizeof(int)); // đọc số từ ống

if (tmp == check) { // kiểm tra số nhận được

printf("%d: received ping\n", getpid());

} else {

printf("%d: received error\n", getpid());

}

tmp += b;

write(p[1], &tmp, sizeof(int)); // ghi số vào ống

}

## **1.3 Khó khăn**

* **Hai tiến trình hoạt động như thế nào, có thứ tự hoạt động trước sau không?**

**Tìm hiểu:** Hoạt động độc lập, có bộ nhớ, biến và file descriptor giống nhau (bản sao dữ liệu, độc lập). Thứ tự hoạt động phụ thuộc vào nhiều yếu tố khác nhau.

# **Primes**

## **2.1 Đạt được**

* **Biết cách sử dụng lệnh fork()**: Tạo tiến trình con thông qua cơ chế clone từ tiến trình cha.
* **Biết cách sử dụng lệnh pipe(int p[])**: Tạo đường ống (pipeline), đặt file mô tả (file descriptors) lần lượt vào p[0] và p[1].
* **Hiểu cơ chế quản lý tài nguyên**: Hệ điều hành quản lý tài nguyên thông qua việc đóng hoặc chặn các tiến trình khi không còn sử dụng.

## **2.2 Mô tả source**

* Sử dụng để tránh lỗi đệ quy vô hạn.

void primes(int read\_fd) \_\_attribute\_\_((noreturn));

* Pipeline được tạo và tiến trình con được tạo để ghi dữ liệu số từ 2 đến 280 xuống file descriptor tại đầu p[1] của pipeline.

int p[2];

pipe(p); // Tạo đường ống

// Tiến trình con để ghi dữ liệu từ 2 đến 280

if (fork() == 0) {

close(p[0]); // Đóng đầu đọc

for (int i = 2; i <= 280; i++) {

write(p[1], &i, sizeof(i)); // Ghi dữ liệu vào đầu ghi

}

close(p[1]); // Đóng đầu ghi

exit(0);

}

close(p[1]); // Đóng đầu ghi trong tiến trình cha

primes(p[0]); // Gọi hàm primes để đọc dữ liệu

* Hàm sàn số nguyên tố với pipe()

void primes(int read\_fd) {

int num;

if (read(read\_fd, &num, sizeof(num)) > 0) {

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

int p[2];

pipe(p); // Tạo đường ống

// Tiến trình con để lấy các số nguyên tố

if (fork() == 0) {

close(p[0]); // Đóng đầu đọc

while (read(read\_fd, &num, sizeof(num)) > 0) {

int cnt = 0;

for (int i = 2; i < num / 2; i++)

if (num % i == 0)

cnt++;

if (cnt == 0)

write(p[1], &num, sizeof(num)); // Ghi số nguyên tố vào ống

}

close(p[1]); // Đóng đầu ghi

exit(0);

} else {

close(read\_fd); // Đóng đầu đọc

close(p[1]); // Đóng đầu ghi

primes(p[0]); // Gọi lại hàm primes để tiếp tục đọc

}

}

* Để tránh lãng phí tài nguyên của hệ điều hành cần giải phóng các tiến trình con.

// Chờ tất cả các tiến trình con kết thúc

while (wait(0) > 0);

exit(0);

}

## **2.3 Khó khăn**

* **Tại sao phải đóng một đầu pipeline khi không sử dụng?**

**Tìm hiểu**: Để tránh việc file ghi ghi đề qua file đọc và file đọc không biết được khi nào EOF để dừng việc đọc dữ liệu.

* **Tại sao primes(int read\_fd) lại có kiểu dữ liệu là int?**

**Tìm hiểu:**

* + - **Mô tả tệp**: Là số nguyên không âm hệ điều hành dùng để nhận diện một tệp hoặc ống (pipe). Khi mở tệp hoặc tạo ống, hệ điều hành cấp phát số nguyên này.
    - **Cách hoạt động**: Lệnh pipe(p) tạo hai mô tả tệp, một cho ghi và một cho đọc. int read\_fd trong hàm primes là mô tả tệp dùng để đọc dữ liệu từ ống.
    - **Sử dụng int**: Dùng kiểu int cho mô tả tệp đơn giản, hiệu quả và dễ thao tác, phù hợp với thiết kế của C và hệ thống Unix.
    - **Kết luận**: read\_fd là cách đại diện cho mô tả tệp trong Unix, giúp hàm primes tương tác dễ dàng với tệp và ống để đọc dữ liệu.

# **2’. Prime\_1**

## **2.1 Đạt được**

* **Biết cách sử dụng lệnh fork()**: Tạo tiến trình con thông qua cơ chế clone từ tiến trình cha.
* **Biết cách sử dụng lệnh pipe(int p[])**: Tạo đường ống (pipeline), đặt file mô tả (file descriptors) lần lượt vào p[0] và p[1].
* **Hiểu cơ chế quản lý tài nguyên**: Hệ điều hành quản lý tài nguyên thông qua việc đóng hoặc chặn các tiến trình khi không còn sử dụng.

## **2.2 Mô tả source**

* Sử dụng để tránh lỗi đệ quy vô hạn.

void primes(int read\_fd) \_\_attribute\_\_((noreturn));

* Pipeline được tạo và tiến trình con được tạo để ghi dữ liệu số từ 2 đến 280 xuống file descriptor tại đầu p[1] của pipeline.

int p[2];

pipe(p);

int N = 280;

if (!fork()) { // tiến trình con ghi dữ liệu từ 2 đến N vào ống

for (int i = 2; i <= N; i++) {

write(p[1], &i, sizeof(i));

}

close(p[1]); // đóng ống khi đọc xong

exit(0); // thoát tiến trình

} else {

seive(p); // gọi hàm seive

}

* Hàm sàng số nguyên tố với pipe

void seive(int p[]) {

close(p[1]); // đóng ống ghi trước khi đọc

int prime;

if (!read(p[0], &prime, sizeof(prime))) { // đóng ống và thoát khi hết số

close(p[0]);

exit(0);

}

printf("prime %d\n", prime); // in ra số nguyên tố

int rp[2];

pipe(rp);

if (!fork()) { // tiến trình con tiến hành quá sàng số nguyên tố và ghi vào rp

int num;

while (read(p[0], &num, sizeof(num)) != 0) { // đọc các số từ ống

if (num % prime != 0) {

write(rp[1], &num, sizeof(num)); // ghi số không chia hết cho prime

}

}

} else {

close(p[0]);

seive(rp); // đệ quy với rp

}

exit(0);

}

## **2’.3 Khó khăn**

* **Khi ống được truyền vào hàm thì nó có tạo bản sao không? Tại sao nếu thêm dòng close(p[1]) trước lời gọi sieve trong hàm main và xóa close(p[1]) đầu hàm sieve thì khi chạy lại lỗi?**

**Tìm hiểu:** Trong C thì khi truyền mảng vào hàm thì không tạo bản sao, chỉ truyền con trỏ đến phần tử đầu mảng. Vấn đề sau chưa tìm được câu trả lời thích đáng.

# **Find**

## **3.1 Đạt được**

* Hiểu quy trình tìm kiếm đường dẫn đến một tệp. Một tệp có thể được liên kết đến thông qua nhiều đường dẫn khác nhau trong hệ điều hành XV6.
* **Sử dụng lệnh open(char \*file, int flags)**: Kiểm tra xem thư mục có tồn tại hay không.
* **Sử dụng lệnh int fstat(int fd, struct stat \*st)**: Xác định các thuộc tính của mô tả tệp, từ đó dễ dàng biết tệp đó thuộc loại thư mục hay tệp.

## **3.2 Mô tả source**

* Hàm nối 2 chuỗi

void concatenate\_path(char \*destination, const char \*source) {

while (\*destination) destination++; // Di chuyển đến cuối đường dẫn hiện tại

while ((\*destination++ = \*source++) != '\0'); // Sao chép phần mới của đường dẫn

}

* Tìm kiếm đường đi của thư mục dẫn đến tệp

void search\_directory(const char \*directory, const char \*filename) {

char full\_path[512]; // Biến để lưu đường dẫn đầy đủ

int file\_descriptor; // Mô tả tệp

struct dirent dir\_entry; // Mục thư mục

struct stat file\_status; // Thông tin tệp

// Mở thư mục

if ((file\_descriptor = open(directory, 0)) < 0) {

fprintf(2, "Error: Cannot open directory %s\n", directory);

return;

}

// Lấy thông tin trạng thái tệp

if (fstat(file\_descriptor, &file\_status) < 0) {

fprintf(2, "Error: Cannot get status for %s\n", directory);

close(file\_descriptor);

return;

}

// Nếu là thư mục, đọc nội dung của nó

if (file\_status.type == T\_DIR) {

while (read(file\_descriptor, &dir\_entry, sizeof(dir\_entry)) == sizeof(dir\_entry)) {

if (dir\_entry.inum == 0 || strcmp(dir\_entry.name, ".") == 0 || strcmp(dir\_entry.name, "..") == 0)

continue; // Bỏ qua các mục đặc biệt

// Xây dựng đường dẫn mới

full\_path[0] = '\0';

concatenate\_path(full\_path, directory);

concatenate\_path(full\_path, "/");

concatenate\_path(full\_path, dir\_entry.name);

// Kiểm tra xem đây có phải là tệp mà chúng ta đang tìm không

if (strcmp(dir\_entry.name, filename) == 0) {

printf("%s\n", full\_path);

}

// Tìm kiếm đệ quy trong các thư mục con

search\_directory(full\_path, filename);

}

}

close(file\_descriptor); // Đóng mô tả tệp

}

* Hàm main có thực hiện kiểm tra lệnh nhập vào đúng yêu cầu hay không.

int main(int argc, char \*argv[]) {

if (argc != 3) {

fprintf(2, "Usage: find <directory> <filename>\n");

exit(1);

}

search\_directory(argv[1], argv[2]); // Gọi hàm tìm kiếm

exit(0);

}

## **3.3 Khó khăn**

**Tại sao cần bước bỏ qua . và .. ?**

* **Tìm hiểu:**

Vòng lặp sẽ tiếp tục tìm kiếm trong thư mục hiện tại. Điều này dẫn đến việc gọi lại hàm search\_directory với cùng một thư mục (/home/user/), và chương trình sẽ quay trở lại đọc các mục trong chính thư mục đó mà không bao giờ dừng lại. Điều này có thể dẫn đến một vòng lặp vô hạn, khiến chương trình chạy mãi mãi mà không có kết quả.

* **Ví dụ:**

/home/user/

├── documents/

│ ├── file1.txt

│ └── file2.txt

└── images/

├── photo1.jpg

└── photo2.png

**Bước 1: Đọc mục trong thư mục**

* Khi mở thư mục /home/user/ và đọc các mục, bạn sẽ nhận được các mục sau:
  + .: Đại diện cho thư mục hiện tại (/home/user/).
  + ..: Đại diện cho thư mục cha (/home).
  + documents: Thư mục con của /home/user/.
  + images: Thư mục con khác của /home/user/.

**Bước 2: Xử lý các mục**

* Khi vòng lặp read đọc từng mục, nó sẽ gặp các mục này một cách tuần tự. Đối với mục .:

if (strcmp(dir\_entry.name, ".") == 0) continue; // Bỏ qua mục thư mục hiện tại

* **Khi gặp mục .**: Điều kiện strcmp(dir\_entry.name, ".") == 0 sẽ đúng, và câu lệnh continue sẽ được thực thi. Điều này có nghĩa là chương trình sẽ bỏ qua mục này và không thực hiện bất kỳ hành động nào đối với thư mục hiện tại.

**3’. Find**

1. **Đạt được**
   * Kiến thức cơ bản về hàm find trong XV6. Cụ thể, find cho phép tìm kiếm đệ quy, kiểm tra các thư mục và các mục của chúng để đối chiếu với một tên tệp tin cụ thể, đồng thời minh hoạ cách duyệt thư mục và mục trong XV6.
2. **Mô tả source**
   * Dựa vào file ls.c, nhóm sử dụng lại hàm “fmtname” và tạo một hàm mới “find”. Cụ thể,

|  |
| --- |
| void find(char \*path, char \*filename) {  char buf[256], \*p;  int fd;  struct dirent de;  struct stat st;  if ((fd = open(path, O\_RDONLY)) < 0) {  fprintf(2, "find: cannot open %s\n", path);  return;  }  if (fstat(fd, &st) < 0) {  fprintf(2, "find: cannot stat %s\n", path);  close(fd);  return;  }  if (st.type == T\_DIR) {  if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {  printf("find: path too long\n");  close(fd);  return;  }  strcpy(buf, path);  p = buf + strlen(buf);  \*p++ = '/';  while (read(fd, &de, sizeof(de)) == sizeof(de)) {  if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) continue;  memmove(p, de.name, DIRSIZ);  p[DIRSIZ] = 0;  if(stat(buf, &st) < 0){  printf("ls: cannot stat %s\n", buf);  continue;  }  if (strcmp(de.name, filename) == 0) {  printf("%s\n", buf);  }    find(buf, filename);  }  }  else if (st.type == T\_FILE) {  if (strcmp(fmtname(path), filename) == 0) {  printf("%s\n", path);  }  }  close(fd);  } |

* + Hàm “fmtname” dùng cho việc xử lý chuỗi (ở đây là đường dẫn) theo một định dạng khác.

|  |
| --- |
| char\* fmtname(char \*path) {  static char buf[DIRSIZ+1];  char \*p;  // Find first character after last slash.  for(p=path+strlen(path); p >= path && \*p != '/'; p--);  p++;  // Return blank-padded name.  if(strlen(p) >= DIRSIZ) return p;  memmove(buf, p, strlen(p));  memset(buf+strlen(p), ' ', DIRSIZ-strlen(p));  return buf;  } |

* + Trong hàm “find”, ta truyền hai tham số kiểu dữ liệu char\* bao gồm đường dẫn thư mục và tên file cần tìm.

|  |
| --- |
| void find(char \*path, char \*filename) |

* + Ngoài ra, một số biến được sử dụng với các mục đích sau:

|  |
| --- |
| char buf[256], \*p; // lưu đường dẫn và con trỏ chỉ vị trí  int fd; // khai báo file descriptor khi mở thư mục  struct dirent de; // lưu thông tin đường dẫn mục  struct stat st; // khai báo trạng thái thư mục đã mở |

* + Đầu tiên, mở đường dẫn bằng lệnh open(). Nếu có lỗi (giá trị trả về âm) thì báo lỗi.

|  |
| --- |
| if ((fd = open(path, O\_RDONLY)) < 0) {  fprintf(2, "find: cannot open %s\n", path);  return;  } |

* + Tiếp theo, cập nhật trạng thái (status) của thư mục. Nếu cập nhật lỗi (giá trị trả về âm) thì báo lỗi.

|  |
| --- |
| if (fstat(fd, &st) < 0) {  fprintf(2, "find: cannot stat %s\n", path);  close(fd);  return;  } |

* + Sau đó, thực hiện kiểm tra trạng thái (status) là thư mục “T\_DIR” hay mục “T\_FILE” từ đó mà xử lý riêng.

|  |
| --- |
| if (st.type = T\_DIR) {  // directory handling  }  else if (st.type = T\_FILE) {  // file handling  } |

* + Đối với thư mục “T\_DIR”, trước hết kiểm tra độ dài ký tự của các thành phần sau có vượt qua độ dài của buf không. Nếu có thì báo lỗi.

|  |
| --- |
| if (strlen(path) + 1 + DIRSIZ + 1 > sizeof buf) {  printf("find: path too long\n");  close(fd);  return;  } |

* + Tiếp, thực hiện sao chép đường dẫn vào buf bằng strcpy() và dịch chuyển con trỏ đến vị trí cuối của chuỗi (đường dẫn) kèm theo “/”.

|  |
| --- |
| strcpy(buf, path);  p = buf + strlen(buf);  \*p++ = '/'; |

* + Thực hiện duyệt các đường dẫn mục (de – directory entry) bằng vòng lặp. Kiểm tra đường dẫn rỗng, chứa “.” (thư mục hiện tại), chứa “..” (thư mục cha) thì bỏ qua.

|  |
| --- |
| while (read(fd, &de, sizeof(de)) == sizeof(de)) {  if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) continue;  …  } |

* + Trong vòng lặp, sao chép tên đường dẫn mục (de – directory entry) vào p kèm theo “\0”. Tiếp đó, kiểm tra trạng thái đường dẫn, nếu có lỗi thì báo lỗi, nếu không có lỗi thì tiếp tục với mục tiếp theo và kiểm tra tên đường dẫn mục (de) khớp với tên tệp tin cần tìm (filename) hay không, nếu có thì in ra. Cuối cùng, thực hiện đệ quy tìm kiếm với đường dẫn là buf và tên tệp tin cần tìm.

|  |
| --- |
| while (read(fd, &de, sizeof(de)) == sizeof(de)) {  if (de.inum == 0 || strcmp(de.name, ".") == 0 || strcmp(de.name, "..") == 0) continue;  memmove(p, de.name, DIRSIZ);  p[DIRSIZ] = 0;  if(stat(buf, &st) < 0){  printf("ls: cannot stat %s\n", buf);  continue;  }  if (strcmp(de.name, filename) == 0) {  printf("%s\n", buf);  }  find(buf, filename);  } |

* + Cuối cùng, nếu trạng thái thư mục đã mở là “T\_FILE” thì kiểm tra nếu chuỗi (tên đường dẫn đã được xử lý) và tên tệp tin cần tìm là giống nhau thì in ra đường dẫn.

|  |
| --- |
| if (strcmp(fmtname(path), filename) == 0) {  printf("%s\n", path);  } |

**3. Khó khăn**

- Sử dụng đệ quy khi duyệt. Cụ thể, cần phải kiểm soát được chiều sâu khi duyệt file và xử lý các đường dẫn mục sao cho hợp lý để tránh bị đệ quy không ngừng.

# **Xargs**

1. **Đạt được**
   * Kiến thức cơ bản về hàm xargs trong XV6. Hàm xây dựng và thực thi các dòng lệnh từ chuẩn đầu vào (standard input).
2. **Mô tả source**
   * Ta có hàm xargs như sau:

|  |
| --- |
| #include "kernel/types.h"  #include "kernel/stat.h"  #include "user/user.h"  #include "kernel/fs.h"  #include "kernel/param.h"  int main(int argc, char \*argv[]){  char\* argv\_1[MAXARG];  int idx = 1;  int temp = 1;  if (strcmp(argv[1], "-n") == 0) idx = 3;    argv\_1[0] = malloc(strlen(argv[idx])+1);  strcpy(argv\_1[0], argv[idx]);  for (int i = idx + 1 ; i < argc ; i++) {  argv\_1[temp] = malloc(strlen(argv[i])+1);  strcpy(argv\_1[temp], argv[i]);  temp++;  }  argv\_1[temp] = malloc(256);  char ch;  int i = 0;  while(read(0, &ch, 1)){  if (ch == '\n') {  argv\_1[temp][i++]='\0';  if (fork() ==0 ) {  exec(argv[idx], argv\_1);  } else {  i = 0;  wait(0);  }  } else {  argv\_1[temp][i++] = ch;  }  }  exit(0);  } |

* + Lưu ý với trường hợp lệnh có “-n 1” thì sẽ xử lý khác (vị trí) so với trường hợp còn lại trong việc sao chép câu lệnh. T
  + Đầu tiên, sao chép tên câu lệnh. Đối với câu lệnh “-n 1”, sao chép từ vị trí 3.

|  |
| --- |
| if (strcmp(argv[1], "-n") == 0) idx = 3;  argv\_1[0] = malloc(strlen(argv[idx])+1);  strcpy(argv\_1[0], argv[idx]); |

* + Sau đó, sao chép các đối số phụ (additional arguments) vào từ argv\_1[i] với i >= 1

|  |
| --- |
| for (int i = idx + 1 ; i < argc ; i++) {  argv\_1[temp] = malloc(strlen(argv[i])+1);  strcpy(argv\_1[temp], argv[i]);  temp++;  } |

* + Cấp phát một vùng nhớ để lưu đầu vào đọc từ chuẩn đầu vào.

|  |
| --- |
| argv\_1[temp] = malloc(256); |

* + Tiếp, đọc vào đầu vào theo ký tự và thực thi câu lệnh.

|  |
| --- |
| while(read(0, &ch, 1)) |

* + Xét 2 trường hợp ký tự là “\n” và các ký tự khác.
  + Nếu ký tự là “\n”, thực hiện thêm vào “\0” và tạo một tiến trình mới dùng fork(), nếu fork() trả ra 0 thì thực thi câu lệnh với các đối số trong argv\_1 (tiến trình còn), nếu không thì chờ cho đến khi tiến trình con xong.

|  |
| --- |
| if (ch == '\n') {  argv\_1[temp][i++]='\0';  if (fork() ==0 ) {  exec(argv[idx], argv\_1);  } else {  i = 0;  wait(0);  } |

* + Nếu là các ký tự khác thì ta cứ thêm vào ký tự đó.

|  |
| --- |
| else {  argv\_1[temp][i++] = ch;  } |

1. **Khó khăn**
   * Xử lý trường hợp ký tự đọc vào là “\n”. Cụ thể, lệnh được cung cấp nhiều hơn một đối số tại một thời điểm. Từ đây, cần tạo một tiến trình mới (dùng fork()) và xử lý riêng lẻ.