ĐẠI HỌC QUỐC GIA TPHCM
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

Báo cáo bài tập

Đề tài: xv6

Môn học: Operating System

Sinh viên thực hiện:

Phạm Quốc Nam Anh	23120111
Nguyễn Xuân Duy	23120121
Nguyễn Minh Hiếu	23120124
Huỳnh Thái Toàn	23120175

Giáo viên hướng dẫn:

Lê Viết Long

Ngày 29 tháng 10 năm 2025

Mục lục

1 Ping Pong

1.1 Mô tả bài toán

Chương trình `pingpong` được thiết kế để minh họa cơ chế giao tiếp liên tiến trình (IPC) thông qua pipes trong hệ điều hành xv6. Chương trình tạo ra hai tiến trình (cha và con) trao đổi một byte dữ liệu qua hai pipes riêng biệt theo cơ chế "ping-pong": tiến trình cha gửi byte đến con, con nhận và in thông báo "received ping", sau đó gửi byte ngược lại cho cha, và cha in thông báo "received pong".

1.2 Phương pháp thực hiện

Chương trình được tổ chức thành ba hàm chính: `child_process()` xử lý logic của tiến trình con, `parent_process()` xử lý logic của tiến trình cha, và `main()` đảm nhiệm việc khởi tạo pipes và phân chia tiến trình.

1.2.1 Hàm `int main(int argc, char *argv[])`

Chịu trách nhiệm thiết lập môi trường giao tiếp giữa hai tiến trình.

- Khởi tạo hai pipes: `p_to_c[]` (parent → child) và `c_to_p[]` (child → parent) bằng system call `pipe()`. Mỗi pipe có hai đầu: `[0]` là read end và `[1]` là write end.
- Sử dụng `fork()` để tạo tiến trình con. Sau khi fork, cả hai tiến trình đều có bản sao của các file descriptors.
- Dựa vào giá trị trả về của `fork()` (0 cho con, PID dương cho cha), chương trình gọi đến hàm xử lý tương ứng.

1.2.2 Hàm `void child_process(int *p_to_c, int *c_to_p)`

Thực hiện vai trò của tiến trình con trong quá trình trao đổi dữ liệu.

- **Đóng các pipe ends không sử dụng**: Con chỉ cần đọc từ `p_to_c[]` và ghi vào `c_to_p[]`, nên đóng `p_to_c[1]` và `c_to_p[0]`. Việc này quan trọng để tránh deadlock và đảm bảo pipe có thể báo hiệu EOF đúng cách.

- **Đọc byte từ cha**: Sử dụng `read()` để đọc 1 byte từ `p_to_c[0]`. Đây là blocking call – tiến trình sẽ đợi cho đến khi có dữ liệu.
- **In thông báo**: Sau khi nhận được byte, in ra "`<pid>: received ping`" với PID lấy từ `getpid()`.
- **Gửi byte ngược lại**: Ghi byte vừa nhận được vào `c_to_p[1]` để trả về cho cha.
- Đóng tất cả file descriptors còn lại và kết thúc bằng `exit(0)`.

1.2.3 Hàm `void parent_process(int *p_to_c, int *c_to_p)`

Thực hiện vai trò của tiến trình cha trong quá trình trao đổi dữ liệu.

- **Đóng các pipe ends không sử dụng**: Cha chỉ cần ghi vào `p_to_c[]` và đọc từ `c_to_p[]`, nên đóng `p_to_c[0]` và `c_to_p[1]`.
- **Gửi byte đến con**: Khởi tạo buffer với ký tự 'X' và ghi vào `p_to_c[1]` bằng `write()`.
- **Đọc byte từ con**: Sử dụng `read()` để đọc 1 byte từ `c_to_p[0]`. Tiến trình cha sẽ block tại đây cho đến khi con gửi dữ liệu ngược lại.
- **In thông báo**: Sau khi nhận được byte, in ra "`<pid>: received pong`".
- Đóng file descriptors, sau đó gọi `wait(0)` để đợi tiến trình con kết thúc (tránh zombie process).

2 Primes

2.1 Mô tả bài toán

Chương trình `Primes` là một chương trình minh họa sàng nguyên tố đồng thời (concurrent prime sieve) bằng cách sử dụng `pipe` và `fork` trong hệ điều hành xv6, dựa trên ý tưởng của Doug McIlroy. Chương trình tạo một chuỗi tiến trình liên kết bằng `pipe`. Mỗi tiến trình đọc dữ liệu, in ra số nguyên tố đầu tiên rồi lọc bỏ các bội số của nó, truyền phần còn lại cho tiến trình kế tiếp. Chương trình minh họa cơ chế giao tiếp và tạo pipeline tiến trình trong hệ điều hành xv6.

2.2 Phương pháp thực hiện

Chương trình mô phỏng sàng Eratosthenes bằng pipeline tiến trình: tiến trình gốc phát dãy 2→280 vào một pipe. Mỗi "nút" (một tiến trình) đọc số đầu tiên làm prime, in ra, rồi lọc bỏ các bội số và truyền phần còn lại sang pipe kế tiếp, đồng thời fork sớm để nút sau chạy song song. Mấu chốt là đóng chính xác các mô tả tệp (FD) để EOF lan truyền và tránh rò rỉ tài nguyên; tiến trình gốc `wait()` đến khi toàn bộ chuỗi con cháu kết thúc. Dữ liệu được truyền dưới dạng `int 32-bit` để tránh chi phí định dạng và đảm bảo hiệu năng I/O ổn định.

2.2.1 Hàm `__attribute__((noreturn)) void exec_pipe(int fd)`

Hàm này thực hiện vai trò của một "nút" trong chuỗi pipeline. Mỗi nút tương ứng với một số nguyên tố và chịu trách nhiệm đọc dữ liệu từ pipe đầu vào, xác định prime, rồi tạo pipe kế tiếp để chuyển tiếp các giá trị không chia hết cho prime đó.

- Đầu tiên, hàm đọc một số nguyên từ đầu vào `fd`. Nếu kết quả trả về khác `sizeof(int)` tức là không còn dữ liệu (EOF), tiến trình đóng `fd` và kết thúc bằng `exit(0)`.
- Nếu đọc thành công, giá trị đầu tiên được xem là *prime* và được in ra màn hình bằng `printf("prime %d`
  `n", prime)`.
- Tiếp theo, tiến trình tạo một pipe mới `p[2]` và gọi `fork()` để tạo tiến trình con.
  - Trong tiến trình con: đóng đầu ghi `p[1]`, đóng `fd` cũ, và đệ quy gọi lại `exec_pipe(p[0])`. Nhờ đó, pipeline được mở rộng thêm một "nút" mới.

  - Trong tiến trình cha: đóng đầu đọc `p[0]` và bắt đầu đọc các giá trị từ `fd`. Với mỗi giá trị đọc được, nếu không chia hết cho `prime`, giá trị đó được ghi vào đầu ghi `p[1]`. Sau khi hoàn tất, cha đóng cả hai đầu pipe, `wait(0)` để đợi tiến trình con kết thúc rồi thoát bằng `exit(0)`.

- Cơ chế này đảm bảo mỗi prime chỉ được xử lý một lần, và các tiến trình hoạt động đồng thời theo cấu trúc pipeline.

2.2.2 Hàm `int main(int argc, char *argv[])`

Hàm `main()` chịu trách nhiệm khởi tạo pipeline đầu tiên và cung cấp dữ liệu đầu vào cho hệ thống sàng.

- Ban đầu, chương trình tạo một pipe bằng `pipe(p)`. Nếu thất bại, chương trình kết thúc bằng `exit(1)`.

- Sau đó gọi `fork()` để tạo tiến trình con đầu tiên.

  - Trong tiến trình cha: đóng đầu đọc `p[0]` và ghi các số từ 2 đến 280 vào đầu ghi `p[1]`. Khi ghi xong, cha đóng `p[1]` để gửi tín hiệu EOF cho tiến trình con, rồi `wait(0)` để đợi toàn bộ pipeline kết thúc.

  - Trong tiến trình con: đóng đầu ghi `p[1]` và gọi `exec_pipe(p[0])`. Vì hàm này có thuộc tính `noreturn`, tiến trình con sẽ không quay lại mà tự xử lý toàn bộ pipeline cho đến khi kết thúc.

3 Find

3.1 Mô tả bài toán

Chương trình `find` được thiết kế để tìm kiếm tất cả các tệp trong cây thư mục có tên trùng với tên cho trước. Chương trình nhận vào hai tham số: *đường dẫn bắt đầu tìm kiếm* và *tên file cần tìm*. Khi thực thi, chương trình sẽ duyệt qua toàn bộ thư mục con và in ra đường dẫn đầy đủ của các tệp phù hợp.

3.2 Phương pháp thực hiện

Dựa trên cấu trúc của chương trình `ls.c` trong hệ điều hành xv6, xây dựng hàm `find()` có khả năng duyệt thư mục theo cơ chế đệ quy. Chương trình được chia thành hai phần chính: hàm `find()` đảm nhiệm việc kiểm tra và xử lý từng đường dẫn, và hàm `main()` chịu trách nhiệm nhận tham số đầu vào cũng như gọi hàm tìm kiếm.

3.2.1 Hàm `void find(char *path, char *search)`

Thực hiện việc mở đường dẫn, xác định loại đối tượng (file hoặc thư mục), và nếu là thư mục thì tiếp tục đọc các entry bên trong để gọi đệ quy tìm kiếm sâu hơn.

- Đầu tiên, hàm mở đường dẫn được truyền vào bằng `open()` và lưu lại trong một file descriptor.
- Tiếp theo, sử dụng system call `fstat()` để lấy thông tin về đối tượng, nhằm xác định đó là file hay thư mục.
- Nếu là **file**, hàm so sánh phần tên cuối cùng trong đường dẫn với từ khóa cần tìm bằng `strcmp()`. Nếu trùng khớp, in ra đường dẫn của file.
- Nếu là **thư mục (T_DIR)**, hàm đọc từng entry trong thư mục bằng `read()`, bỏ qua hai entry đặc biệt "." và "..". Sau đó nối đường dẫn con và gọi lại chính hàm `find()` để tiếp tục tìm kiếm trong các thư mục con.

3.2.2 Hàm `int main(int argc, char *argv[])`

Chịu trách nhiệm kiểm tra tham số đầu vào và gọi hàm `find()` để thực hiện việc tìm kiếm.

- Chương trình yêu cầu người dùng nhập đúng hai tham số: *đường dẫn bắt đầu* và *tên file cần tìm*. Nếu số lượng đối số nhỏ hơn 3 (tức là thiếu tham số), chương trình sẽ in thông báo hướng dẫn sử dụng qua `fprintf()` và kết thúc bằng `exit(1)`.

- Khi nhận đủ đối số, hàm `find()` được gọi với `argv[1]` là đường dẫn và `argv[2]` là tên file cần tìm. Hàm `find()` trả về số lượng file được tìm thấy, và nếu giá trị này bằng 0, chương trình sẽ in thông báo "*No files found matching 'filename'*" để cho biết không có kết quả nào trùng khớp. Cuối cùng, chương trình kết thúc bằng `exit(0)`.

4 xargs

4.1 Mô tả bài toán

Chương trình `xargs` được thiết kế để xây dựng và thực thi các lệnh từ đầu vào tiêu chuẩn. Chương trình nhận vào một lệnh và các đối số ban đầu từ command line, sau đó đọc thêm các đối số từ stdin (thường được pipe từ các lệnh khác) và thực thi lệnh với tất cả các đối số đã thu thập được. Điều này cho phép xử lý hàng loạt các đối số một cách hiệu quả, đặc biệt hữu ích khi kết hợp với các lệnh như `find`.

4.2 Phương pháp thực hiện

Chương trình được xây dựng gồm hai phần chính: hàm `readline()` đảm nhiệm việc đọc dữ liệu từ stdin theo từng dòng, và hàm `main()` chịu trách nhiệm nhận tham số đầu vào, phân tách đối số và thực thi lệnh.

4.2.1 Hàm `int readline(char *buf, int maxlen)`

Thực hiện việc đọc một dòng dữ liệu từ stdin và lưu vào buffer.

- Hàm sử dụng system call `read()` để đọc từng ký tự một từ `stdin`.
- Quá trình đọc tiếp tục cho đến khi gặp ký tự xuống dòng (`\n`) hoặc đạt đến giới hạn buffer (`maxlen - 1`).
- Nếu gặp EOF (end of file) khi đọc:
  - Nếu đã đọc được ít nhất một ký tự, hàm kết thúc chuỗi bằng null terminator và trả về số ký tự đã đọc.
  - Nếu chưa đọc được ký tự nào, hàm trả về 0 để báo hiệu hết dữ liệu.
- Khi gặp ký tự xuống dòng, hàm thay thế bằng null terminator và trả về số ký tự đã đọc (không bao gồm `\n`).
- Cuối cùng, hàm đảm bảo chuỗi luôn được kết thúc bằng null terminator trước khi trả về số lượng ký tự.

4.2.2 Hàm `int main(int argc, char *argv[])`

Chịu trách nhiệm kiểm tra tham số đầu vào, xử lý các đối số từ stdin và thực thi lệnh với các đối số đã thu thập.

- **Kiểm tra tham số**: Chương trình yêu cầu ít nhất một tham số (tên lệnh cần thực thi). Nếu số lượng đối số nhỏ hơn 2 (tức là thiếu tên lệnh), chương trình sẽ in thông báo hướng dẫn sử dụng qua `fprintf()` và kết thúc bằng `exit(1)`.
- **Sao chép đối số từ command line**: Chương trình sao chép tất cả các đối số từ `argv[1]` đến `argv[argc-1]` vào mảng `args[]`. Các đối số này sẽ là phần cố định của lệnh cần thực thi.
- **Đọc và xử lý từng dòng stdin**: Sử dụng vòng lặp `while` để gọi hàm `readline()`, đọc từng dòng dữ liệu từ stdin cho đến khi hết dữ liệu (EOF).
- **Phân tách đối số từ dòng đầu vào**:
  - Sử dụng con trỏ `p` để duyệt qua chuỗi `line`, bỏ qua các ký tự khoảng trắng và tab ở đầu.
  - Với mỗi từ được tìm thấy, lưu địa chỉ của từ đó vào mảng `args[]` tại vị trí `j` (bắt đầu từ `argc - 1`).
  - Di chuyển con trỏ `p` qua các ký tự của từ cho đến khi gặp khoảng trắng, tab hoặc null terminator.
  - Thay thế ký tự phân cách (khoảng trắng hoặc tab) bằng null terminator để tạo các chuỗi độc lập, sau đó bỏ qua các ký tự phân cách tiếp theo.
  - Quá trình lặp lại cho đến khi hết dòng hoặc đạt giới hạn số đối số (`MAXARG - 1`).
- **Thực thi lệnh**:
  - Đánh dấu kết thúc mảng đối số bằng cách gán `args[j] = 0`.
  - Gọi `fork()` để tạo process con. Nếu `fork()` thất bại, in thông báo lỗi và thoát với mã lỗi 1.
  - Trong process con (`pid == 0`): gọi `exec()` với `args[0]` là tên lệnh và `args` là danh sách đối số. Nếu `exec()` thất bại, in thông báo lỗi và kết thúc process con.

  - Trong process cha (`pid > 0`): gọi `wait(0)` để đợi process con hoàn thành trước khi tiếp tục đọc dòng tiếp theo.

- Sau khi xử lý hết tất cả các dòng từ stdin, chương trình kết thúc bằng `exit(0)`.