Lab 4. xv6 utils

1. Mục đích

Thông qua bài thực hành này, sinh viên sẽ làm quen với hệ điều hành xv6, và sử dụng các system call của nó để xây dựng các ứng dụng chạy ở user space (**user application**)

2. Tóm tắt lý thuyết

Mỗi process đang chạy có vùng nhớ riêng chứa instruction, data và stack (**process virtual address space**). Các instruction của process được thực thi trong môi trường **user space**. Khi một process gọi đến một system call, môi trường thực thi chuyển sang **kernel space** để thực thi system call đó bên trong kernel. Sau khi system call được xử lý xong, kết quả được trả về cho process trong user space và process sẽ tiếp tục thực thi như bình thường

Hình bên dưới là các system call của xv6. Các system call này lần đầu được giới thiệu trong hệ điều hành Unix của Ken Thompson và Dennis Ritchie. Dù đơn giản, nhưng chúng kết hợp với nhau rất tốt và tạo ra sự linh động đáng ngạc nhiên

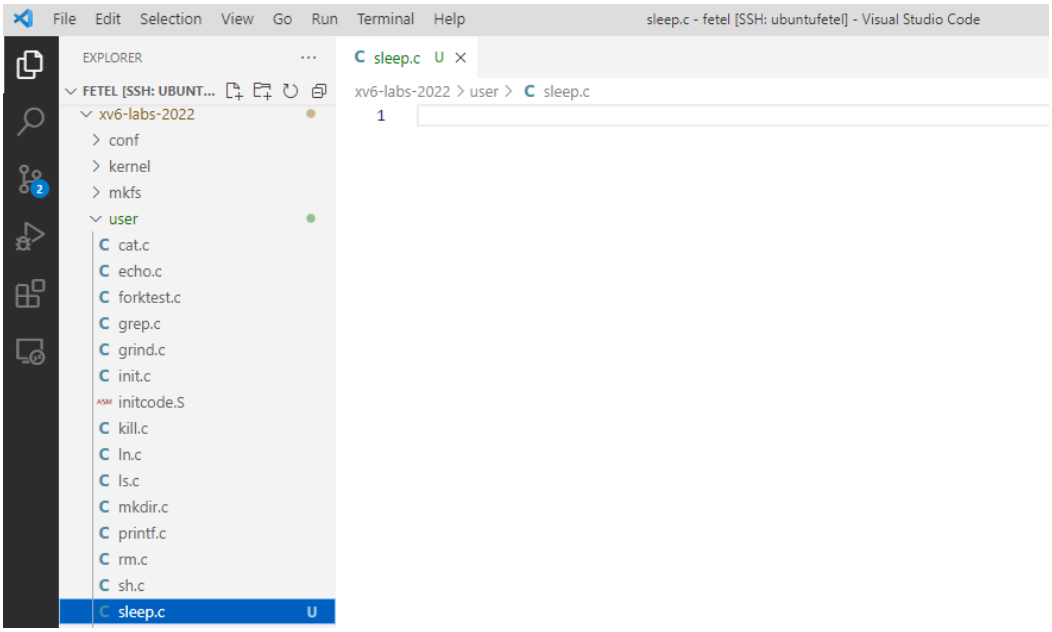
System call	Description
int fork()	Create a process, return child's PID.
int exit(int status)	Terminate the current process; status reported to wait(). No return.
int wait(int *status)	Wait for a child to exit; exit status in *status; returns child PID.
int kill(int pid)	Terminate process PID. Returns 0, or -1 for error.
int getpid()	Return the current process's PID.
int sleep(int n)	Pause for n clock ticks.
int exec(char *file, char *argv[])	Load a file and execute it with arguments; only returns if error.
char *sbrk(int n)	Grow process's memory by n bytes. Returns start of new memory.
int open(char *file, int flags)	Open a file; flags indicate read/write; returns an fd (file descriptor).
int write(int fd, char *buf, int n)	Write n bytes from buf to file descriptor fd; returns n.
int read(int fd, char *buf, int n)	Read n bytes into buf; returns number read; or 0 if end of file.
int close(int fd)	Release open file fd.
int dup(int fd)	Return a new file descriptor referring to the same file as fd.
int pipe(int p[])	Create a pipe, put read/write file descriptors in p[0] and p[1].
int chdir(char *dir)	Change the current directory.
int mkdir(char *dir)	Create a new directory.
int mknod(char *file, int, int)	Create a device file.
int fstat(int fd, struct stat *st)	Place info about an open file into *st.
int stat(char *file, struct stat *st)	Place info about a named file into *st.
int link(char *file1, char *file2)	Create another name (file2) for the file file1.
int unlink(char *file)	Remove a file.

Figure 1.2: Xv6 system calls. If not otherwise stated, these calls return 0 for no error, and -1 if there's an error.

3. Thực hành

Chuyển thư mục làm việc vào thư mục source code của xv6 và tạo branch mới lab4 từ branch origin/util, sau đó checkout branch lab4

Tạo file sleep.c trong thư mục user và mở file trên VSCode



Thêm nội dung bên dưới vào file và save lại

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

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

    int num = atoi(argv[1]);
    sleep(num);

    exit(0);
}
```

Tham số truyền vào chương trình là số clock tick, chương trình sẽ ngưng hoạt động trong khoảng thời gian bằng với số clock tick này. Lưu ý là tham số truyền vào dưới dạng string, hàm atoi chuyển từ string sang số nguyên trước khi gọi đến system call sleep

Mở Makefile trên VSCode và thêm tên chương trình _sleep vào cuối variable UPROGS, sau đó save lại. Lưu ý là tên chương trình phải tương ứng với tên file source code (sleep.c → _sleep)

```
174  UPROGS=\
175      $U/_cat\
176      $U/_echo\
177      $U/_forktest\
178      $U/_grep\
179      $U/_init\
180      $U/_kill\
181      $U/_ln\
182      $U/_ls\
183      $U/_mkdir\
184      $U/_rm\
185      $U/_sh\
186      $U/_stressfs\
187      $U/_usertests\
188      $U/_grind\
189      $U/_wc\
190      $U/_zombie\
191      $U/_sleep\
```

Gõ command make qemu để build và khởi động emulator. Thực thi ls trên shell của xv6

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2227
xargstest.sh   2 3 93
cat            2 4 32328
echo           2 5 31200
forktest       2 6 15304
grep           2 7 35680
init           2 8 31992
kill           2 9 31200
ln             2 10 31112
ls             2 11 34232
mkdir          2 12 31240
rm             2 13 31224
sh             2 14 53472
stressfs       2 15 32096
usertests      2 16 180584
grind          2 17 47296
wc             2 18 33320
zombie         2 19 30768
sleep          2 20 31192
console        3 21 0
$
```

Gõ command sleep 30. Chương trình sẽ ngưng trong một lúc (30 clock ticks) trước khi kết thúc thực thi

```
$ sleep 30
$
```

BÁO CÁO THỰC HÀNH

Sinh viên: .....

MSSV: ..... Nhóm: .....

Lưu ý:

- Trước khi làm các bài tập, SV nên xem source code các user app có sẵn trong thư mục user: user/echo.c, user/grep.c, user/rm.c, …
- Library function mà user app có thể dùng rất hạn chế. Danh sách các library function được khai báo trong file user/user.h và source code của chúng được định nghĩa trong user/ulib.c, user/printf.c, và user/umalloc.c.
- Danh sách các system call được khai báo trong file user/user.h. File user/usys.S (được tạo tự động khi build) là code assembly gọi đến system call để chuyển từ user app (user space) sang kernel code (kernel space)
- Hàm main() cần gọi đến system call exit() khi kết thúc thực thi
- Để build và thêm chương trình vào file system, thêm tên nó vào variable UPROGS trong Makefile
- File system của xv6 được lưu trong file fs.img. File fs.img không thay đổi đối với mỗi lần build trừ khi thực thi command make clean

Bài 1: Viết user app **pingpong** trên xv6:

- Gồm hai process: process cha và process con
- Process cha gửi 1 byte đến process con, process con in ra stdout “<pid>: received ping”
- Process con gửi 1 byte đến process cha và exit, process cha in ra stdout “<pid>: received pong” và exit
- File source code của chương trình được lưu tại user/pingpong.c

Ví dụ:

```
$ pingpong
4: received ping
3: received pong
$
```

Gợi ý:

- Dùng pipe() để tạo ra một pipe
- Dùng fork() để tạo process con
- Dùng read() và write() để đọc ghi pipe
- Dùng getpid() để tìm process ID

Bài 2: Viết user app **find** trên xv6:

- Nhận tham số <P> là đường dẫn của một thư mục và <N> là tên file cần tìm
- In ra stdout đường dẫn của tất cả các file có tên giống với <N> trong thư mục <P>
- File source code của chương trình được lưu tại user/find.c

Ví dụ:

```
$ echo helloo > b
$ mkdir a
$ echo he > a/b
$ mkdir a/aa
$ echo hhello > a/aa/b
$ find . b
./b
./a/b
./a/aa/b
$
```

Gợi ý:

- Xem file user/ls.c để biết cách đọc thư mục
- Dùng đệ quy để tìm file trong tất cả các thư mục con trừ "." và ".."
- Command `find . b` tương đương với command `find . -name b` trên Linux

Bài 3: Viết user app **xargs** trên xv6:

- Nhận tham số <CMD> là một command
- Thực thi <CMD> với tham số là mỗi dòng đọc được từ stdin
- File source code của chương trình được lưu tại user/xargs.c

Ví dụ:

```
$ echo hello too | xargs echo bye
bye hello too
$
$ find . b | xargs grep hello
helloo
hhello
$
```

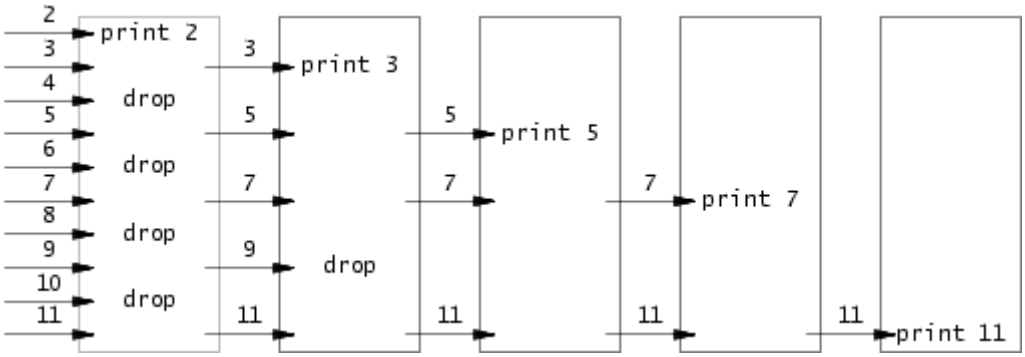
Gợi ý:

- Dùng fork() và exec() để tạo process con để thực thi <CMD>. Dùng wait() để chờ cho process con thực thi xong. Lặp lại các bước trên cho mỗi dòng đọc từ stdin
- Khi đọc từ stdin, kí tự "\n" báo hiệu kết thúc 1 dòng
- Command `find . b | xargs grep hello` tương đương với command `find . -name b | xargs -n 1 grep hello` trên Linux

Bài 4:** Viết user app **primes** trên xv6:

- Tạo ra process con P1 và gửi cho P1 các số từ 2 đến 35
- P1 in ra stdout số đầu tiên mà nó nhận được (số 2), bỏ qua các số là bội số của 2 và gửi các số còn lại cho process con P2 do P1 tạo ra
- P2 in ra stdout số đầu tiên mà nó nhận được (số 3), bỏ qua các số là bội số của 3 và gửi các số còn lại cho process con P3 do P2 tạo ra
- Chuỗi pipeline của các process tiếp diễn như trên

- Kết quả in ra Terminal là các số nguyên tố từ 2 đến 35
- File source code của chương trình được lưu tại user/primes.c



Ví dụ:

```
$ primes
prime 2
prime 3
prime 5
prime 7
prime 11
prime 13
prime 17
prime 19
prime 23
prime 29
prime 31
$
```

Gợi ý:

- Đóng các file descriptor mà process không cần, nếu không xv6 sẽ cạn kiệt tài nguyên trước khi đạt đến số 35
- Process cha nên đợi cho đến khi tất cả các process con P1, P2,… kết thúc
- Số nguyên gửi đến các process thông qua các pipe nên là kiểu int 4 bytes
- read() trả về 0 nếu write-side của một pipe đã đóng