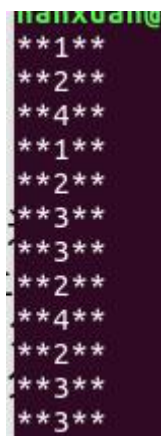


1、进程的创建实验

1) 将下面的程序编译运行，并解释现象。

```
void main(){
    int pid1=fork();
    printf("***1**\n");
    int pid2=fork();
    printf("***2**\n");
    if(pid1==0){int pid3=fork();printf("***3**\n");}
    else printf("***4**\n");
}
```

程序运行结果如下图：



根据代码对程序进行分析：

1. 在执行语句: `int pid1 = fork();`时，`fork()`通过系统调用创建一个与原来进程几乎完全相同的进程，作为原来进程的子进程，在父进程中 `fork()`返回值 `pid1` 等于新建子进程的进程 ID，子进程的 `pid1` 等于 0，因此可以用 `fork()`返回值判断父进程和子进程。

另外父子进程的执行顺序不固定，先后顺序取决与系统的调度策略，同一份代码执行结果可能会有所不同。本实验报告根据截图的实验结果进行分析。

2. `int pid1 = fork();`语句执行结束后，此时有一个父进程和一个子进程在执行，两个进程均执行 `printf` 语句打印出 `***1**`

3. 执行 `int pid2 = fork();` 时，此时原先的父子进程分别创建了一个新的子进程，四个进程均打印出 `***2**`

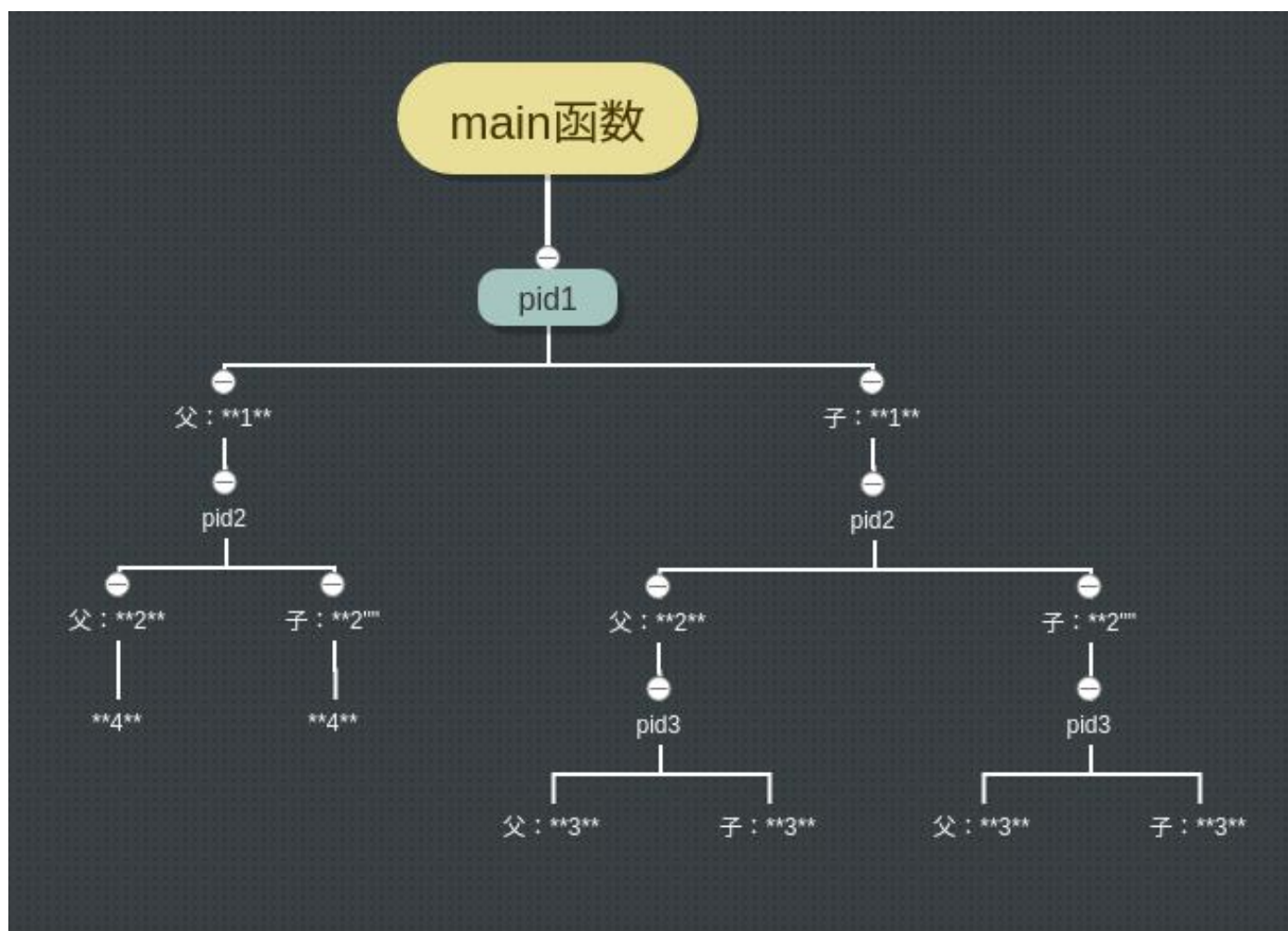
4. 对 `pid1` 的值进行判断，原先的父进程及它的最新子进程的 `pid1` 值不为 0，原先子进程和它的最新子进程的 `pid1` 等于 0。根据 `if` 判断的结果，

原先的父进程及它的最新子进程均打印出 `***4**`，

原先子进程和它的子进程均执行 `int fork3 = fork();`各自创建了一个新的子进程，

原先子进程、它的子进程和它们创建的子进程，这四个进程均打印出 `***3**`

程序执行流程如下：



截图的运行结果的数字顺序是：1 2 4 1 2 3 3 2 4 2 3 3

根据上图可知，该数字顺序可分为：

最原先的父进程：1 2 4

最原先的父进程在 pid1 产生的子进程：1 2 3

最原先的父进程在 pid1 产生的子进程在 pid3 产生的子进程：3

最原先的父进程在 pid2 产生的子进程：2 4

最原先的父进程在 pid1 产生的子进程在 pid2 产生的子进程：2 3

最原先的父进程在 pid1 产生的子进程在 pid2 产生的子进程在 pid3 产生的子进程：3

2) 编写一段程序，使用系统调用 `fork()` 创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符；父进程显示字符“a”；子进程分别显示字符“b”和字符“c”。试观察记录屏幕上的显示结果，并分析原因。

编写的代码如图：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <sys/types.h>
4  #include <unistd.h>
5
6  int main() {
7      int pid1 = fork();
8      if (pid1 != 0) {
9          int pid2 = fork();
10         if (pid2 != 0)
11             printf("a\n");
12         else
13             printf("c\n");
14     }
15     else {
16         printf("b\n");
17     }
18 }
```

多次执行代码的结果如图：

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
a
c
b
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
a
b
c
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
a
b
c
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
b
a
c
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
a
c
b
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./e
b
a
c
```

执行多程序后可以看到 a、b、c 的打印顺序并不固定。

分析：在第一次 fork() 后，子进程的 pid1 等于 0，打印出 b；父进程 pid1 不等于 0，进行第 2 次 fork()，第 2 次 fork() 中产生的新子进程的 pid2 等于 0，打印出 c，父进程打印出 a；父子进程的执行顺序不固定，先后顺序取决于系统的调度策略，因此同一份代码执行结果可能会有所不同。

3) 下面程序将在屏幕上输出的字符‘X’、数字“1”和“0”各多少个？为什么？

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    int i, a=0;
    pid_t pid;
    if((pid=fork()))a=1;
    for(i=0; i<2; i++){
        printf("X");
    }
    if(pid==0)printf("%d\n",a);
    return 0;
}
```

根据代码分析，程序应该有 4 个 X，0 个 1,1 个 0

验证：程序运行结果如下：



```
XXXX0
```

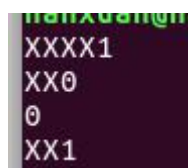
分析：在第一次 fork()后，pid 不为 0 的是父进程，父进程的 a=1,在循环中父进程打印 2 次 X，子进程也打印 2 个 X，子进程的 pid 等于 0，因此子进程打印出 a 值为 0。所以总共打印出 4 个 X，一个 0。

4) 如果将上面 main 函数修改如下，则屏幕上输出的字符‘X’、数字“1”和“0”各多少个？为什么？

```
int main(void)
{
    int i, a=0;
    pid_t pid[2];
    for(i=0; i<2; i++){
        if((pid[i]=fork()))a=1;
        printf("X");
    }
    if(pid[0]==0)printf("%d\n",a);
    if(pid[1]==0)printf("%d\n",a);
    return 0;
}
```

从代码分析，一开始我认为是打印出 6 个 X，2 个 1,2 个 0

验证：运行结果如图：



```
XXXX1
XX0
0
XX1
```

结果是 8 个 X，在网上查资料后才了解原因。

分析：

出现 8 个 X 主要和两个方面相关：

1) `printf` 的缓冲机制

`printf` 是一个行缓冲函数，只有在一定条件下才会刷新缓冲区输出数据：

1.缓冲区已满

2.检测到输入的字符有 `"\n"`或`"\r"`

3.调用 `fflush` 函数刷新缓冲区

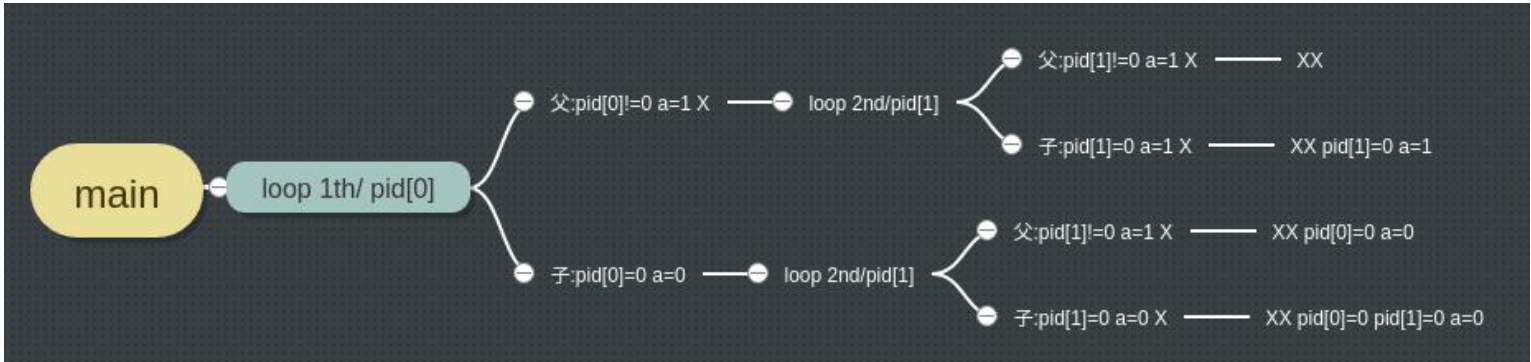
4.调用 `scanf` 函数从缓冲区读取数据

5.程序运行结束（执行 `printf` 的进程或线程结束的时候会主动调用 `fflush` 来刷新缓冲区）

因此在第一次循环结束时，由于程序运行不满足上述任何一个条件，因此此时缓冲区数据是没有被刷新，因此数据并没有输出。

2) `fork()` 函数是将进程的当前情况拷贝一份，因此第 2 次循环中产生的 2 个子进程会分别拷贝各自父进程缓冲区的数据，因此多了两个 X,所以是 8 个 X 不是 6 个。

程序执行流程如图：



2.信号处理实验

(a)编制一段程序，使用系统调用 `fork()` 创建两个子程序，再用系统调用 `signal()` 让父进程捕捉键盘上来的中断信号(即按 `Ctrl C` 键)，当捕捉到中断信号后，父进程调用 `kill()` 向两个子进程发出信号，子进程捕捉到信号后，分别输出下面信息后终止：`child process 1 is killed by parent!``child process 2 is killed by parent!`父进程等待两个子进程终止后，输出以下信息后终止：`parent process is killed`

(b)在上述 (a) 中的程序中增加语句 `signal(SIGINT, SIG_IGN)` 和 `signal(SIGQUIT, SIG_IGN)`，观察执行结果并分析原因。这里 `signal(SIGINT, SIG_IGN)` 和 `signal(SIGQUIT, SIG_IGN)` 分别为忽略“`Ctrl Z`”键信号以及忽略“`Ctrl C`”中断信号。

```
void waiting();
void stop();
int wait_mark;

int main() {
    int p1, p2;
    while ((p1 = fork()) == -1);

    if (p1 > 0) {
        while ((p2 = fork()) == -1);

        if (p2 > 0) {
            wait_mark = 1;
            signal(SIGSTOP, SIG_IGN);
            signal(SIGINT, stop);
            waiting();
            kill(p1, 16);
            kill(p2, 17);
            wait(0);
            wait(0);
            printf("parent process is killed!\n");
            exit(0);
        } else {
            wait_mark = 1;
            signal(17, stop);
            waiting();
            printf("child process 2 is killed by parent!\n");
            exit(0);
        }
    } else {
        wait_mark = 1;
        signal(16, stop);
        waiting();
        printf("child process 1 is killed by parent!\n");
        exit(0);
    }
}

void waiting() {
    while(wait_mark != 0);
}

void stop() {
    wait_mark = 0;
}
```

代码运行结果：


```
nanxuan@nanxuan-Lenovo-G40-70:~/Desktop$ ./t
^Cparent process is killed!
```

没有出现想要的 child process 1 is killed by parent!child process 2 is killed by parent!

原因分析：由于用户按下 Ctrl+C 的时候，此时存在三个进程，三个进程都会捕捉到信号 SIGINT，父进程对 SIGINT 做了处理，但是两个子进程没有对它进行处理，因此两个子进程都会提前终止。

* 分别在两个子程序的 *signal* 语句前增加语句 `signal(SIGINT, SIG_IGN)`:

```
nanxuan@nanxuan-Lenovo-G40-70M:~/Desktop$ ./t
^Cchild process 1 is killed by parent!
child process 2 is killed by parent!
parent process is killed!
```

原因分析：SIGINT 是可捕捉信号，子进程捕捉到 SIGINT 后对它处理方式是忽略，因此子进程没有受到影响，可以接收到父进程发出的 kill 信号，打印出信息后自行终止。

增加语句 `signal(SIGQUIT, SIG_IGN)`:

```
nanxuan@nanxuan-Lenovo-G40-70M:~/Desktop$ ./t
^Cchild process 1 is killed by parent!
child process 2 is killed by parent!
parent process is killed!
```

```
nanxuan@nanxuan-Lenovo-G40-70M:~/Desktop$ ./t
^Z
[24]+  Stopped                  ./t
```

原因分析：

SIGQUIT 是不可捕捉、不可忽略的信号，因此用户按下 Ctrl+Z 的时候，发出的 SIGQUIT 会让三个进程终止

3.进程间共享内存实验

完成课本第三章的练习 3.10 的程序

代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/shm.h>
#include <errno.h>
#include <signal.h>
#include <unistd.h>
#include <ctype.h>
#include <string.h>

#define SIZE 1024
extern int errno;          // linux下捕获错误
```

判断命令行输入参数的个数，是否为 2：

```
int main(int argc, char* argv[]) {
    if (argc != 2) {
        printf("Usage: ./a.out [number]\n");
        exit(1);
    }
}
```

对第 2 个参数进行检查是否都是数字：

```
for (int i = 0; i < strlen(argv[1]); i++) {
    if (!isdigit(argv[1][i])) {
        printf("Please enter a positive number!\n");
        exit(1);
    }
}
```

将字符串参数转换成数字并判断是否为 0：

```
int num = atoi(argv[1]);
if (num == 0) {
    printf("Please enter a positive number!\n");
    exit(1);
}
```


生成一个新的 IPC 对象，IPC_CREAT|IPC_EXCL 可确保生成的 IPC 对象是新建的，如果已存在一个 IPC 对象，则会报错；加上 0600 作为校验是因为我用的是 Ubuntu 系统，不如此校验，不能顺利获得共享空间的值

```
int shmid;
char *shmptr;
key_t key;
// 生成一个整数IPC，确保父子进程的ICP相同
if ((key = ftok("/dev/null", 1)) < 0) {
    printf("ftok error:%s\n", strerror(errno));
    return -1;
}
// 创建一个新的IPC对象
if ((shmid = shmget(key, SIZE, 0600|IPC_CREAT|IPC_EXCL)) < 0) {
    printf("shmget error:%s\n", strerror(errno));
    return -1;
}
```

生成子进程，判断是否生成成功：

```
pid_t fork1 = fork();

if (fork1 < 0) {
    printf("fork error:%s\n", strerror(errno));
    return -1;
}
```

对于子进程，

先连接共享内存为 shmid 的共享地址，连接成功后把共享内存区对象映射到调用进程的地址空间，随后可像本地空间一样访问

再根据获取的数字 num，生成有 num 个数字的斐波那契数列，把包含生成的数列的格式化字符串 format_string 放进共享内存区中，最后子进程输出“child Done!”后退出

```

}
else if (fork1 == 0) {
    if ((shmptr = (char*)shmat(shmid, 0, 0)) == (void*)-1) {
        printf("child shmat error:%s\n", strerror(errno));
        return -1;
    }

    char format_string[SIZE];
    if (num == 1) {
        sprintf(format_string, "%d", 0);
    }
    else if (num == 2) {
        sprintf(format_string, "%d %d", 0, 1);
    }
    else {
        int fib0 = 0, fib1 = 1;
        sprintf(format_string, "%d %d ", 0, 1);
        for (int i = 2; i < num; i++) {
            int format_string_length = strlen(format_string);
            if (i % 2 == 0) {
                fib0 += fib1;
                sprintf(format_string+format_string_length, "%d ", fib0);
            }
            else {
                fib1 += fib0;
                sprintf(format_string+format_string_length, "%d ", fib1);
            }
        }
    }

    int format_string_length = strlen(format_string);
    sprintf(format_string+format_string_length, "\n");
    memcpy(shmptr, format_string, sizeof(format_string));
    puts("child Done!\n");
    exit(0);
}

```

对于父进程，连接共享内存为 shmid 的共享地址，等待子进程结束后，输出共享内存中的字符串，然后删除共享区域，并输出 parent Done!后结束进程：

```

else {
    if ((shmptr = (char*)shmat(shmid, 0, 0)) == (void*)-1) {
        printf("parent shmat error:%s\n", strerror(errno));
        return -1;
    }

    wait(0);
    puts("parent Output:");
    puts(shmptr);

    if ((shmctl(shmid, IPC_RMID, 0) < 0)) {
        printf("parent shmctl error:%s\n", strerror(errno));
        return -1;
    }
    printf("parent Done!\n");
}
}

```

运行结果如下:

```

nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6
Usage: ./a.out [number]
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 -1
Please enter a positive number!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 0
Please enter a positive number!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 1
child Done!

parent Output:
0

parent Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 2
child Done!

parent Output:
0 1

parent Done!

```

```

nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 11
child Done!

parent Output:
0 1 1 2 3 5 8 13 21 34 55

parent Done!

```

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 7
child Done!

parent Output:
0 1 1 2 3 5 8

parent Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 8
child Done!

parent Output:
0 1 1 2 3 5 8 13

parent Done!
```

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 12
child Done!

parent Output:
0 1 1 2 3 5 8 13 21 34 55 89

parent Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 13
child Done!

parent Output:
0 1 1 2 3 5 8 13 21 34 55 89 144

parent Done!
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 15
child Done!

parent Output:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377

parent Done!
```

```
nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./ex6 23
child Done!

parent Output:
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946 17711

parent Done!
```

4.实现 shell 的提示

完成课本上第三章的项目：实现 **shell**。除此之外满足下面要求：

在 **shell** 下，按 **ctrl+C** 时不会终止 **shell**；

实现程序的后台运行。

代码截图如下：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <sys/wait.h>
5  #include <unistd.h>
6  #include <signal.h>
7
8  /* 每次输入的命令规定不超过80个字符 */
9  #define MAX_LINE 80
10 #define HISTORY_NUM 11
11
12 int isGetHistoryList = 0;    // 判读是否按下Ctrl+C
13 char tmpInputBuffer[MAX_LINE]; // 存放输入命令的字符数组
14
```

一个长度为 11 的循环队列，可以存放最近的 10 个历史命令记录：

```
typedef struct
{
    char data[HISTORY_NUM][MAX_LINE];
    int front;
    int rear;
} HistoryQueue;
HistoryQueue history_queue;    // 存放历史命令
```

```
// 初始化历史命令队列
int initHistoryQueue(HistoryQueue *Q) {
    Q->front = Q->rear = 0;
    return 1;
}
// 判断是否为空
int isEmptyQueue(HistoryQueue Q) {
    return Q.front == Q.rear;
}
// 判断是否队列已满
int isFullQueue(HistoryQueue Q) {
    return (Q.rear+1)%HISTORY_NUM == Q.front;
}
```



```

}
// 把命令存入队列
int enqueue(HistoryQueue *Q, char *command) {
    if (isFullQueue(*Q))
        Q->front = (Q->front + 1) % HISTORY_NUM;
    memset(Q->data[Q->rear], '\0', sizeof(Q->data[Q->rear]));
    strncpy(Q->data[Q->rear], command, strlen(command));
    Q->rear = (Q->rear + 1) % HISTORY_NUM;
    return 1;
}

```

按下 Ctrl+C 获得历史命令记录:

```

void getHistoryList() {
    isGetHistoryList = 1;
    if (!isEmptyQueue(history_queue)) {
        putchar('\n');
        int index = history_queue.front;
        for (int i = 0; index != history_queue.rear; i++) {
            printf("[%d] - %s\n", i, history_queue.data[index]);
            index = (index + 1) % HISTORY_NUM;
        }
    }
    fflush(stdout);
}

```

输入 r x 时, 获取最近和 x 匹配的命令:

```

char *getHistoryData(char *x) {
    if (!isEmptyQueue(history_queue)) {
        int index = (history_queue.rear - 1) % HISTORY_NUM;
        // 从后往前找匹配命令
        for (int i = 0; index != (history_queue.front - 1) % HISTORY_NUM; i++)
            if (strncmp(history_queue.data[index], x, strlen(x)) == 0) {
                return history_queue.data[index];
            }
        index = (index - 1) % HISTORY_NUM;
    }
    return "";
}

```

获得最后一条命令:


```
char *getLastCommand() {
    if (isEmptyQueue(history_queue))
        return "";
    int index = (history_queue.rear - 1) % HISTORY_NUM;
    return history_queue.data[index];
}
```

对输入命令进行分解：

先把命令复制进全局变量 tmpInputBuffer

```
// 对输入命令进行分解
void splitInputBuffer(char inputBuffer[], int length, char *args[], int *background) {
    memset(tmpInputBuffer, '\0', MAX_LINE);
    strncpy(tmpInputBuffer, inputBuffer, strlen(inputBuffer));
}
```

检查每一个字符，并将分解的字符串数组放进 args：

```
int start = -1; /* 命令的第一个字符位置 */
int ct = 0; /* 下一个参数存入args[]的位置 */
// 检查inputbuffer每一个字符
for (int i = 0; i < length; i++) {
    switch(inputBuffer[i]) {
        case ' ':
        case '\t': /* 字符为分割参数的空格或制表符(tab) '\t' */
            if (start != -1) {
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            start = -1;
            break;
        case '\n': /* 命令行结束 */
            if (start != -1) {
                args[ct] = &inputBuffer[start];
                ct++;
            }
            inputBuffer[i] = '\0';
            args[ct] = NULL;
            break;
        default:
            if (start == -1)
                start = i;
            if (inputBuffer[i] == '&') {
                *background = 1;
                inputBuffer[i] = '\0';
            }
    }
}
args[ct] = NULL; /* 命令字符串 \0 */
```

```

    }
    args[ct] = NULL; /* 命令字符数 > 80 */
    // 如果用户按下回车('\n'), 则将args第一个元素设为空字符, 避免段错误
    if (args[0] == NULL) {
        args[0] = "";
        args[1] = NULL;
    }
}

```

接收用户输入的指令并分解命令:

```

void setup(char inputBuffer[], char *args[], int *background) {
    int length; /* 命令的字符数目 */

    memset(inputBuffer, '\0', MAX_LINE);
    /* 读入命令行字符, 存入inputBuffer */
    length = read(STDIN_FILENO, inputBuffer, MAX_LINE);

    if (length == 0) /* 输入ctrl+d, 结束shell程序 */
        exit(0);
    if (length < 0) {
        perror("error reading the command");
        exit(-1);
    }

    splitInputBuffer(inputBuffer, length, args, background);
}

```

main 函数:

先对历史命令记录队列进行初始化以及声明其他变量:

```

int main() {
    initHistoryQueue(&history_queue); /* 初始化历史命令记录队列 */

    char inputBuffer[MAX_LINE]; /* 这个缓存用来存放输入的命令 */
    int background; /* ==1时, 表示在后台运行命令, 即在命令后加上'&' */
    char *args[MAX_LINE/2+1]; /* 命令最多40个参数 */
    pid_t pid;
}

```

进入接收用户命令的循环：

如果接收到 Ctrl+C，打印出历史命令记录后，isGetHistoryList 被设置为 1，因此 continue，不执行后续代码，继续循环：

```
while(1) {
    background = 0;
    isGetHistoryList = 0;

    printf(" COMMAND->");
    fflush(stdout); // 输出缓冲区内容

    signal(SIGINT, getHistoryList);

    if (isGetHistoryList != 0) {
        continue;
    }
}
```

如果用户不是输入 Ctrl+C：

如果用户是输入 “r” 或 “r x” 格式的命令：

```
// 接收用户命令并分解
setup(inputBuffer, args, &background);
// 判断命令是不是 "r" 开头的
if (strcmp(args[0], "r") == 0) {
    // 对inputBuffer清零
    memset(inputBuffer, '\0', strlen(inputBuffer));
    // 输入 "r x" 格式时，
    if (args[1] != NULL) {
        // 寻找可以匹配的最近的命令
        char *oldCommand = getHistoryData(args[1]);
        strncpy(inputBuffer, oldCommand, strlen(oldCommand));
    }
    else { // 输入 "r" 时，
        // 获取最后一条命令
        char *oldCommand = getLastCommand();
        puts(oldCommand);
        strncpy(inputBuffer, oldCommand, strlen(oldCommand));
    }
    if (inputBuffer == NULL)
        continue;
    else {
        // 分解 inputBuffer
        splitInputBuffer(inputBuffer, strlen(inputBuffer), args, &background);
    }
}
```



```

}
// 把命令放进历史信息队列
enqueue(&history_queue, tmpInputBuffer);
// 产生子进程
pid = fork();
if (pid < 0) {
    printf("fork error\n");
    exit(1);
}
else if (pid == 0) {
    execvp(args[0], args);
    exit(0);
}
else {
    if (background == 0)
        wait(0);
}
}

```

运行结果如下图：

先输入超过 10 条指令：

```

nanxuan@nanxuan-Lenovo-G40-70m:~/Desktop/os$ ./t
COMMAND->ls
ash1    ash2.c  os.docx  shm2    t        tmp.c  实验课第1讲-实验1指导课件.ppt
ash1.c  ex6.c   os.txt   shm2.c  test.c   why.c
COMMAND->ls -l
total 744
-rwxrwxr-x 1 nanxuan nanxuan  9232 4月 16 15:30 ash1
-rw-rw-r-- 1 nanxuan nanxuan  1703 4月 16 15:30 ash1.c
-rw-rw-r-- 1 nanxuan nanxuan   297 4月 16 15:27 ash2.c
-rw-rw-r-- 1 nanxuan nanxuan  2896 4月 16 11:04 ex6.c
-rw-rw-r-- 1 nanxuan nanxuan 498783 4月 16 11:10 os.docx
-rwxrwxrwx 1 nanxuan nanxuan  3974 4月 16 11:55 os.txt
-rwxrwxr-x 1 nanxuan nanxuan 13376 4月 16 09:30 shm2
-rw-rw-r-- 1 nanxuan nanxuan  2309 4月 16 00:09 shm2.c
-rwxrwxr-x 1 nanxuan nanxuan 13968 4月 16 23:30 t
-rw-rw-r-- 1 nanxuan nanxuan  5620 4月 16 22:06 test.c
-rw-rw-r-- 1 nanxuan nanxuan   911 4月 16 21:58 tmp.c
-rw-rw-r-- 1 nanxuan nanxuan   557 4月 16 20:05 why.c
-rwxrwxrwx 1 nanxuan nanxuan 177664 4月  5 16:45 实验课第1讲-实验1指导课件.ppt
COMMAND->ls -a
.      ash1.c  ~/.lock.os.docx#  shm2    test.c  实验课第1讲-实验1指导课件.ppt
..     ash2.c  os.docx          shm2.c  tmp.c
ash1   ex6.c   os.txt          t        why.c
COMMAND->

```

```
ash1 ash2.c ex6.c os.txt shm2.c test.c why.c
COMMAND->gcc ex6.c -o e
COMMAND->rm e
COMMAND->gcc ex6.c -o e
COMMAND->ls
ash1 ash2.c ex6.c os.txt shm2.c test.c why.c
ash1.c e os.docx shm2 t tmp.c 实验课第1讲-5
COMMAND->rm
rm: missing operand
Try 'rm --help' for more information.
COMMAND->rm e
COMMAND->
```

```
COMMAND->rm
rm: missing operand
Try 'rm --help' for more information.
COMMAND->rm e
COMMAND->less why.c
COMMAND->cat why.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
```

按下 Ctrl+C，可以显示最近的 10 条命令：

```
COMMAND->^C
[0] - ls -l
[1] - ls -a
[2] - gcc ex6.c -o e
[3] - rm e
[4] - gcc ex6.c -o e
[5] - ls
[6] - rm
[7] - rm e
[8] - less why.c
[9] - cat why.c
```

输入 r, 执行上一条命令:

```
FWXFWXFWX 1 nanxuan nanxuan 177004 473 5 10.43 实验课第1讲-实验1指导课件.ppt
COMMAND->ls -a
.      ash1.c  ~/.lock.os.docx# shm2    test.c  实验课第1讲-实验1指导课件.ppt
..     ash2.c  os.docx          shm2.c  tmp.c
ash1   ex6.c   os.txt           t       why.c
COMMAND->r
ls -a
.      ash1.c  ~/.lock.os.docx# shm2    test.c  实验课第1讲-实验1指导课件.ppt
..     ash2.c  os.docx          shm2.c  tmp.c
ash1   ex6.c   os.txt           t       why.c
COMMAND->less why.c
```

输入 less why.c 再输入 cat why.c

```
ash1 ex6.c os.txt
COMMAND->less why.c
COMMAND->cat why.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

在输入 cat why.c 后再输入 rl 获取最近 l 开头的命令:

```
#define MAX_LINE 80
int main() {
    char inputBuffer[MAX_LINE];
    char tmpInputBuffer[MAX_LINE];

    while (1) {
        read(STDIN_FILENO, inputBuffer, MAX_LINE);
        puts("test1:puts(inputBuffer)");
        puts(inputBuffer);
        puts("OK!");

        memset(tmpInputBuffer, '\0', MAX_LINE);

        puts("test2:puts(inputBuffer)");
        puts(inputBuffer);
        puts("OK!");
    }
}
COMMAND->r l
```

最近匹配 less why.c 的命令是: less why.c, 命令成功执行:

nanxuan@nanxuan-Lenovo-G40-70m: ~/Desktop/os

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include <signal.h>

#define MAX_LINE 80
int main() {
    char inputBuffer[MAX_LINE];
    char tmpInputBuffer[MAX_LINE];

    while (1) {
        read(STDIN_FILENO, inputBuffer, MAX_LINE);
        puts("test1:puts(inputBuffer)");
        puts(inputBuffer);
        puts("OK!");

        memset(tmpInputBuffer, '\0', MAX_LINE);

        puts("test2:puts(inputBuffer)");
        puts(inputBuffer);
        puts("OK!");
    }
}
```

why.c

再输入 ls，再输入 r l，匹配最近输入的 ls 命令：

```
COMMAND->r l
COMMAND->ls
ash1    ash2.c  os.docx  shm2     t        tmp.c    实验课第1讲-实验1指导课件.ppt
ash1.c  ex6.c      os.txt   shm2.c   test.c   why.c
COMMAND->r l
ash1    ash2.c  os.docx  shm2     t        tmp.c    实验课第1讲-实验1指导课件.ppt
ash1.c  ex6.c      os.txt   shm2.c   test.c   why.c
```

输入 rm ash1 后，再输入 r r：

```
ash1.c  ex6.c      os.txt   shm2.c   test.c   why.c
COMMAND->rm ash1
COMMAND->r r
rm: cannot remove 'ash1': No such file or directory
COMMAND->
```

Ctrl+C 获取最近的命令，可以看到 r 和 rl 等执行历史命令，在历史均有保存到历史记录中：

```
COMMAND->^C
[0] - ls -a
[1] - less why.c
[2] - cat why.c
[3] - less why.c
[4] - cat why.c
[5] - less why.c
[6] - ls
[7] - ls
[8] - rm ash1
[9] - rm ash1
```

```
COMMAND->ls
ash1.c  ex6.c  shm2    t      tmp.c  实验课第1讲-实验1指导课件.ppt
ash2.c  os.txt  shm2.c  test.c  why.c
COMMAND->r
ls
ash1.c  ex6.c  shm2    t      tmp.c  实验课第1讲-实验1指导课件.ppt
ash2.c  os.txt  shm2.c  test.c  why.c
COMMAND->^C
[0] - ls
[1] - ls -a
[2] - ls -l
[3] - cat tmp.c
[4] - ls -l
[5] - ls -l
[6] - ls
[7] - ls
```

后台运行效果如下：

因为父、子进程同时运行，也就是父进程可以不需要等待子进程完成就可以接收用户的输入，也就是会在子进程完成之前就打印出 COMMAND->可供用户输入，可以看到在下面的例子中子进程还未列出所有文件，父进程就打印出 COMMAND->供用户输入

```
COMMAND->ls -l&
COMMAND->total 244
-rw-rw-r-- 1 nanxuan nanxuan 1703 4月 16 15:30 ash1.c
-rw-rw-r-- 1 nanxuan nanxuan 297 4月 16 15:27 ash2.c
-rw-rw-r-- 1 nanxuan nanxuan 2896 4月 16 11:04 ex6.c
-rwxrwxrwx 1 nanxuan nanxuan 3974 4月 16 11:55 os.txt
-rwxrwxr-x 1 nanxuan nanxuan 13376 4月 16 09:30 shm2
-rw-rw-r-- 1 nanxuan nanxuan 2309 4月 16 00:09 shm2.c
-rwxrwxr-x 1 nanxuan nanxuan 13968 4月 17 09:47 t
-rw-rw-r-- 1 nanxuan nanxuan 5905 4月 16 23:46 test.c
-rw-rw-r-- 1 nanxuan nanxuan 911 4月 16 21:58 tmp.c
-rw-rw-r-- 1 nanxuan nanxuan 557 4月 16 20:05 why.c
-rwxrwxrwx 1 nanxuan nanxuan 177664 4月 5 16:45 实验课第1讲-实验1指导课件.ppt
```