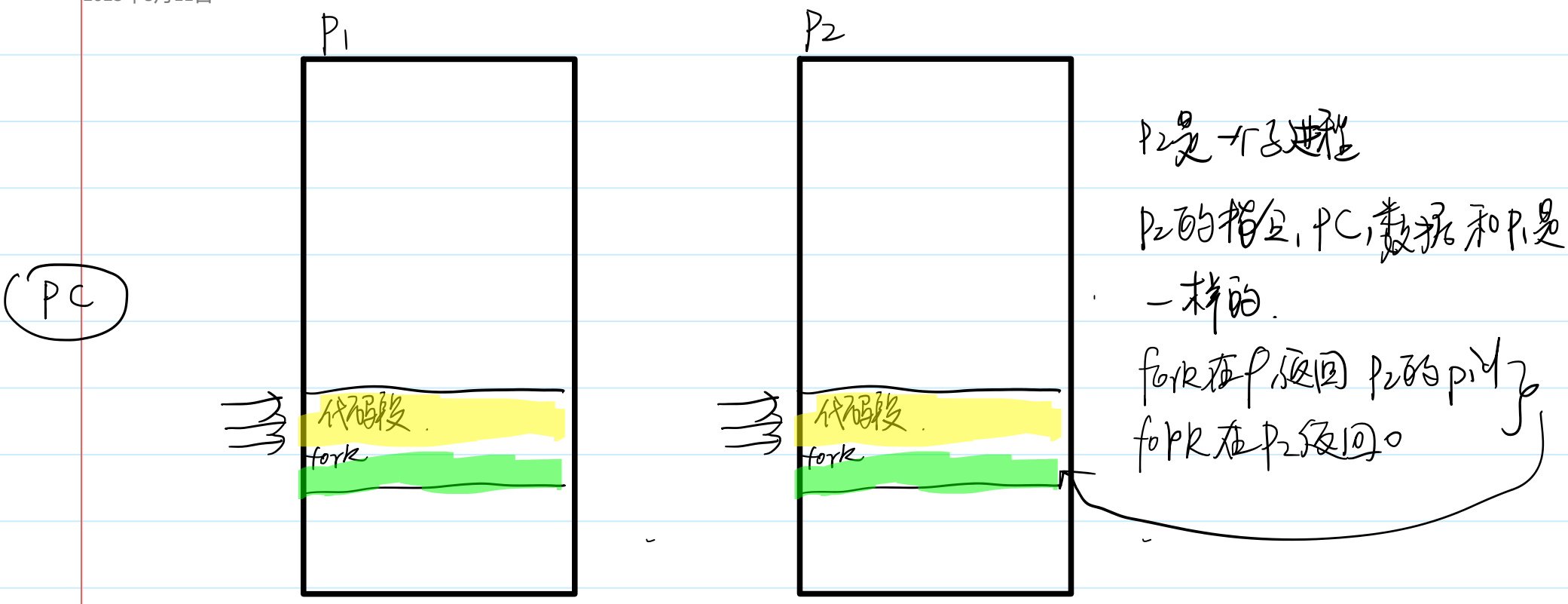
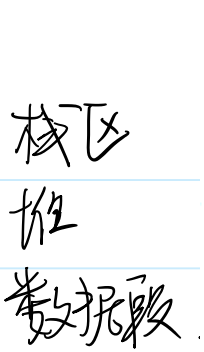
fork

2023年8月11日 9:39



用户态的数据

2023年8月11日 9:47




```c
int dataseg = 1;
int main(int argc, char *argv[])
{
    int stack = 2;
    int *pHeap = (int *)malloc(sizeof(int));
    *pHeap = 3;
    if(fork()){
        // 父进程
        sleep(1);
        printf("parent, dataseg = %d, stack = %d, heap = %d\n",
               dataseg,stack,*pHeap);
    }
    else{
        // 子进程
        printf("before child, dataseg = %d, stack = %d, heap = %d\n",
               dataseg,stack,*pHeap);
        dataseg += 3;
        stack += 3;
        *pHeap += 3;
        printf("after child, dataseg = %d, stack = %d, heap = %d\n",
               dataseg,stack,*pHeap);
    }
    return 0;
}
```

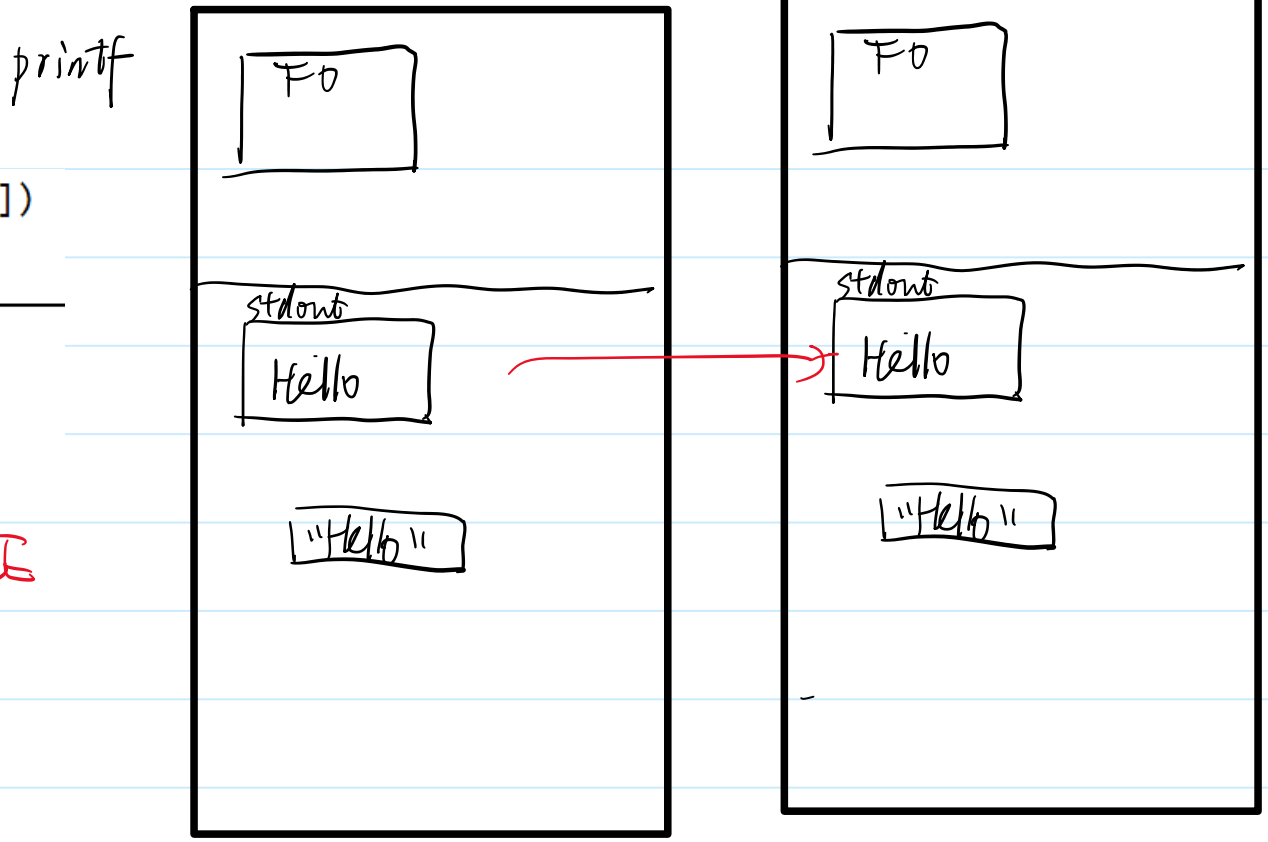
FILE 也会拷贝一份

2023年8月11日 9:56

```c
int main(int argc, char *argv[])
{
    printf("Hello");
    fork();
    printf("World\n");
    return 0;
}
```

① printf 不加换行 拷贝到 FILE

② FILE 在用户态，在 fork 时子进程会拷贝一份。

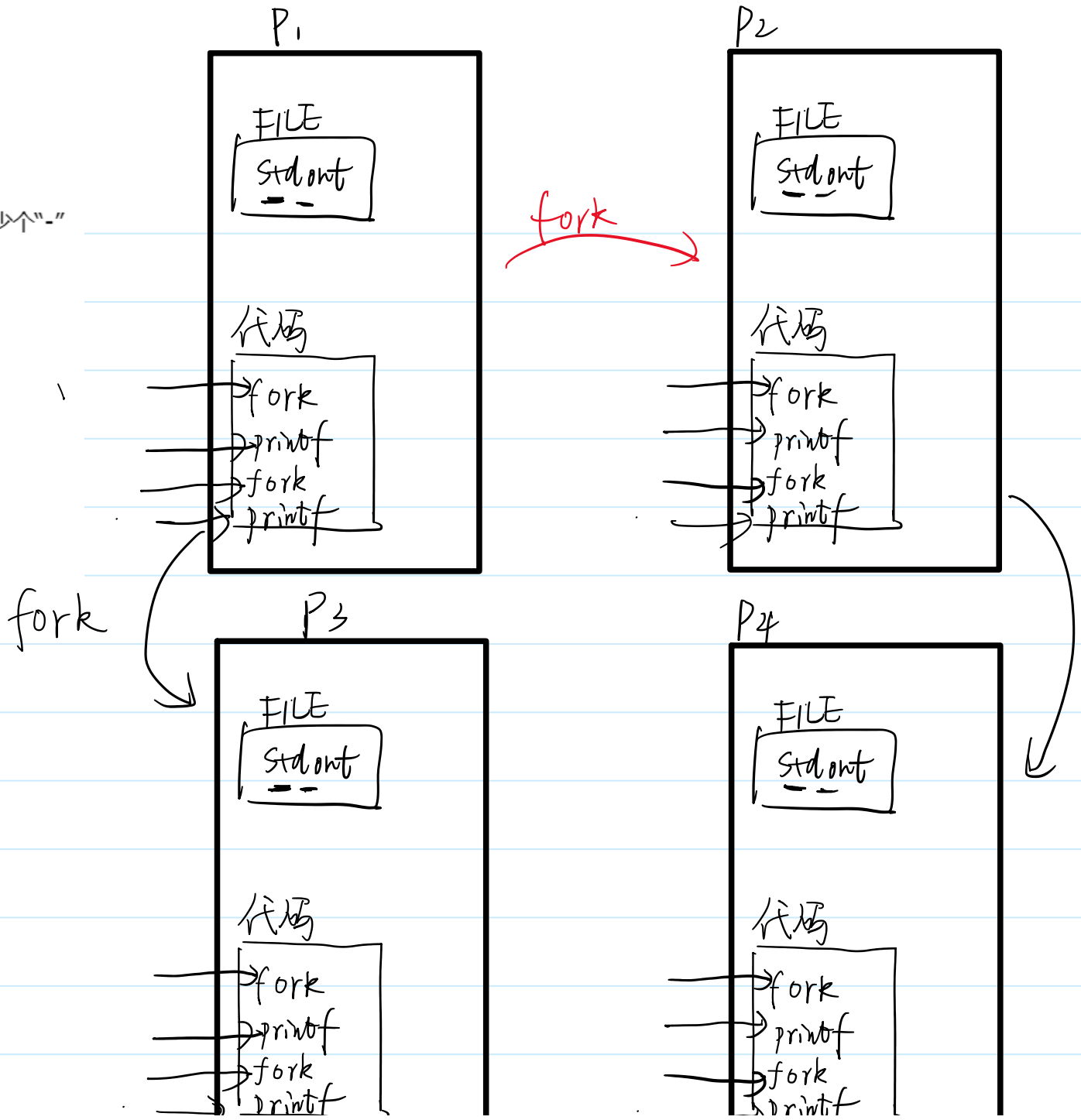


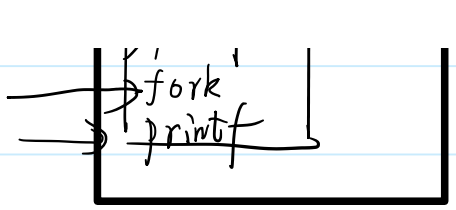
作业

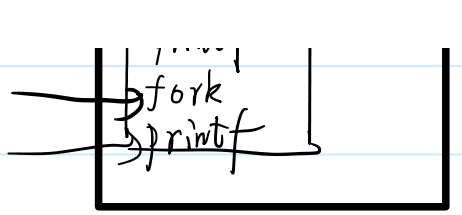
2023年8月11日 10:06

(1). 请问下面的程序一共输出多少个"-"

```
int main()
{
   fork();
   printf("-");
   fork();
   printf("-");
   return 0;
}
```




fork
printf

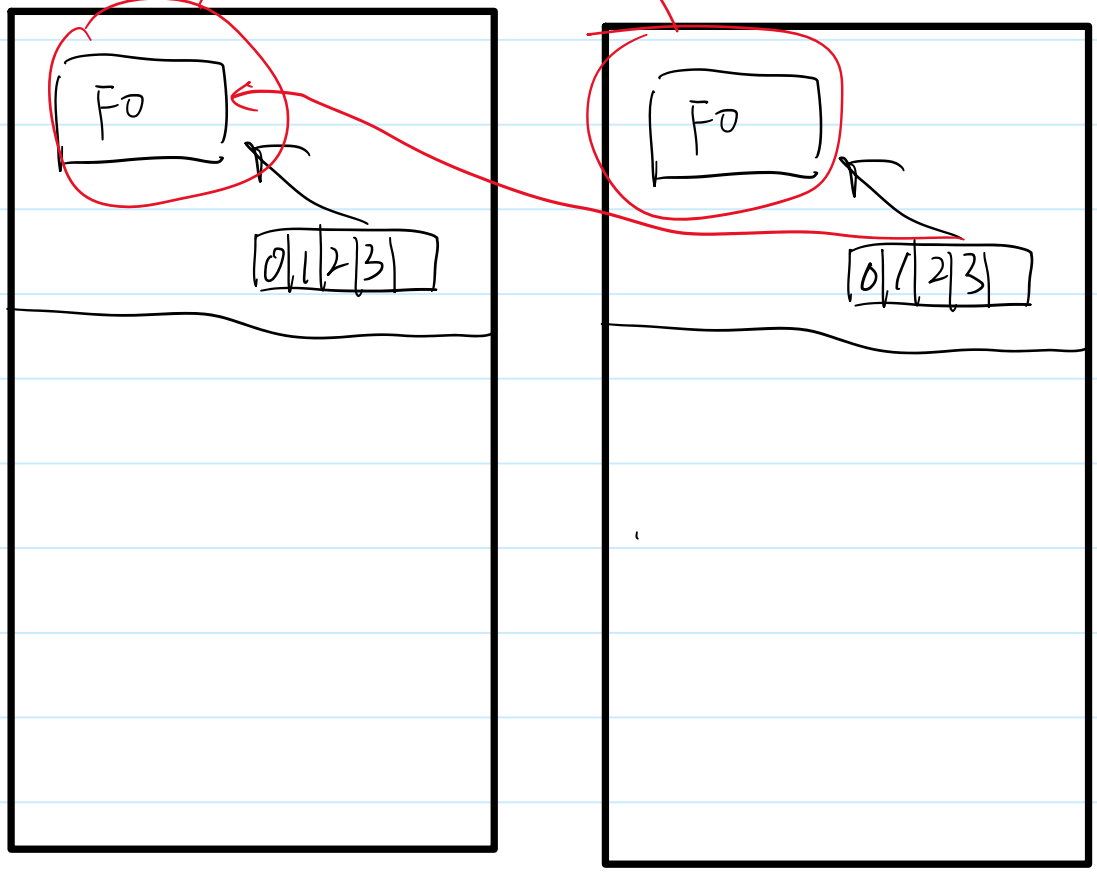

fork
printf

文件对象

2023年8月11日 10:11

先创建一个文件对象(open)
再fork.

```c
int main(int argc, char *argv[])
{
    int fd = open("file1",O_RDWR);
    if(fork()){
        sleep(1);
        write(fd,"world",5);
    }
    else{
        write(fd,"hello",5);
    }
    return 0;
}
```

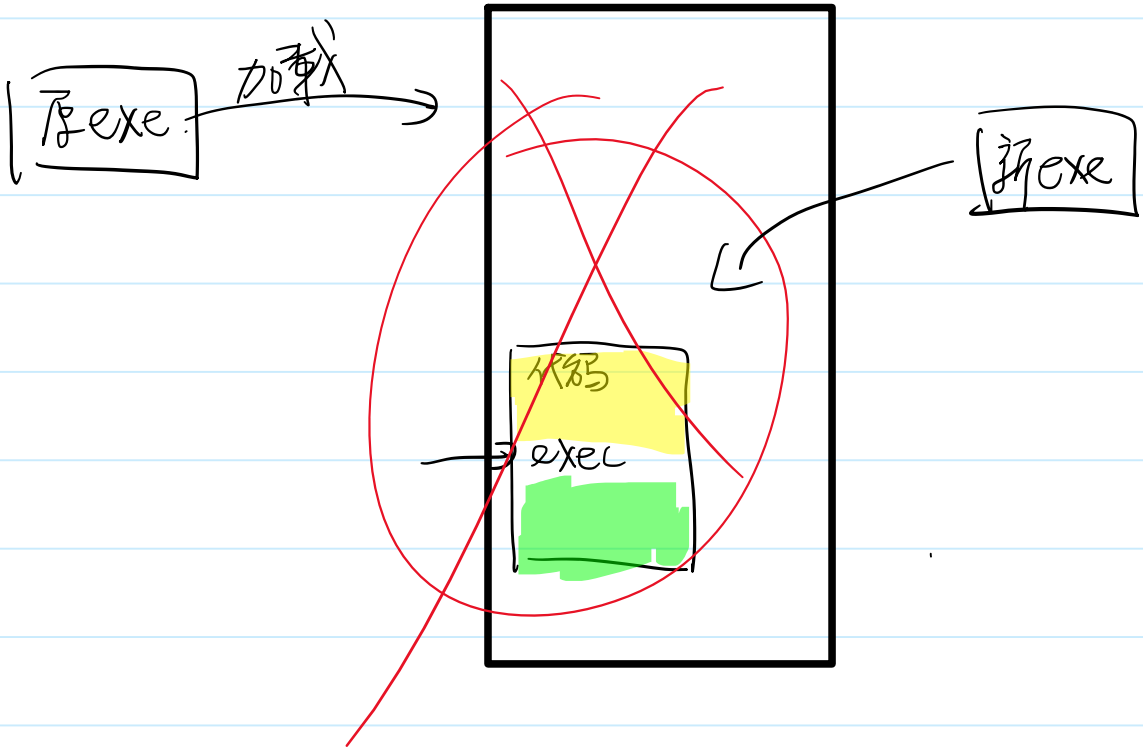


只有fork之前FO共享的
索引数组独立的.
fork之后的FO都独立的.

exec函数族

2023年8月11日 10:28

execute - 进程运行过程 加载另一个可执行程序的指令和数据 "等会"



具体的一些函数

2023年8月11日 10:33

l list 可变参数

v vector 数组.

```
int execl(const char *pathname, const char *arg, ...
                /* (char  *) NULL */);
int execlp(const char *file, const char *arg, ...
                /* (char  *) NULL */);
int execle(const char *pathname, const char *arg, ...
                /*, (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
                char *const envp[]);
```

数组，每一个元素 char*

pathname. 指名 exe 在文件系统的路径.

arg 组织命令的参数.

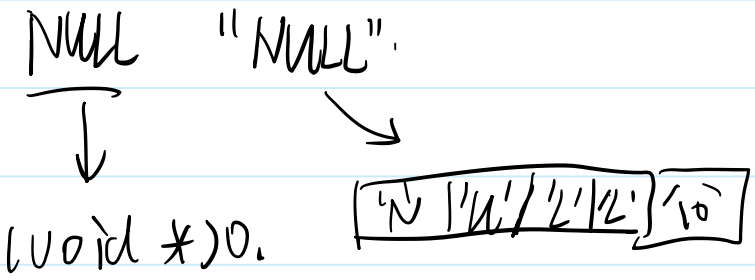
2023年8月11日 11:10

旧exe

```c
int main(int argc, char *argv[])
{
    printf("Hello\n");
    printf("pid = %d\n", getpid());
    // my_add 可执行程序的路径
    // ./my_add 3 4 命令行参数
    execl("my_add",
          "./my_add","3","4",
          NULL); //NULL提示execl参数个数
    printf("World\n");
    return 0;
}
```

新exe.

```c
int main(int argc, char *argv[])
{
    // ./my_add 3 4
    printf("my add pid = %d\n", getpid());
    ARGS_CHECK(argc,3);
    int lhs = atoi(argv[1]);
    // 字符串->整数 array to integer
    int rhs = atoi(argv[2]);
    printf("lhs + rhs = %d\n", lhs+rhs);
    return 0;
}
```



exec的过程

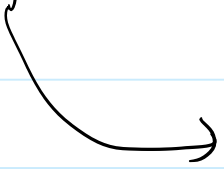
2023年8月11日 11:14

① 清空堆、栈

② 用新exe的代码和数据去取代原进程代码段和数据段

③ PC重置为新代码段开始

create_process：创建子进程，让子进程执行命令



if(!fork()){

exec

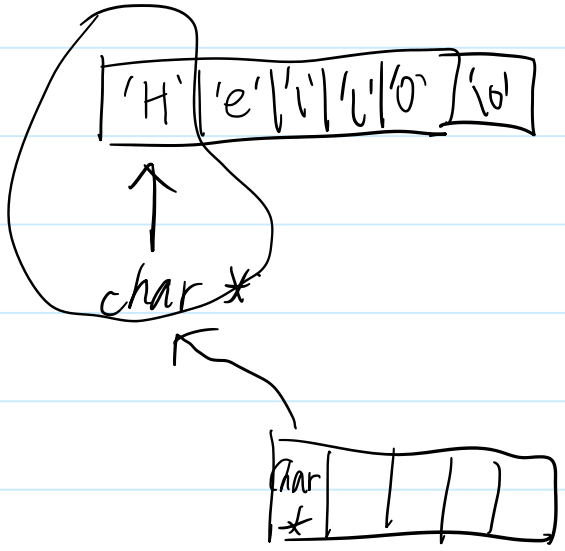
}.

execv

2023年8月11日 11:19

把命令行参数组织成数组.

```
int execv(const char *pathname, char *const argv[]);
```




```c
int main(int argc, char *argv[])
{
    printf("Hello\n");
    printf("pid = %d\n", getpid());
    // my_add 可执行程序的路径
    // ./my_add 3 4 命令行参数
    char *args[] = {"./my_add","4","5",NULL};
    execv("my_add",args);
    printf("World\n");
    return 0;
}
```

进程终止的故事

2023年8月11日 11:31

```
./5_fork
parent, pid = 8869, ppid = 7384
child, pid = 8870, ppid = 8869
```

```
liao:LinuxDay14$ ./5_fork
parent, pid = 8871, ppid = 7384
liao:LinuxDay14$ child, pid = 8872, ppid = 1
```

进程具有并发性

①命令行提示符
②子进程的ppid.

```c
int main(int argc, char *argv[])
{
    if(fork()){
        printf("parent, pid = %d, ppid = %d\n",
               getpid(), getppid());
        //sleep(1);
    }
    else{
        printf("child, pid = %d, ppid = %d\n",
               getpid(), getppid());
    }
    return 0;
}
```

资源回收

2023年8月11日 11:37

task_struct

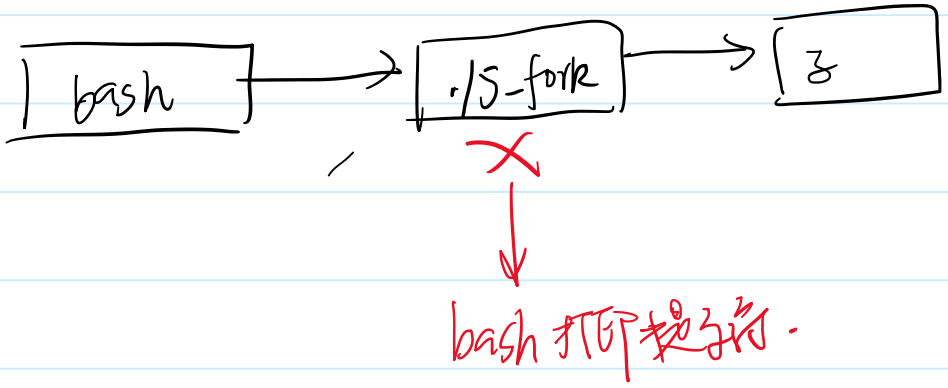
子进程终止，资源由父进程回收

父进程在子进程之前终止

孤儿进程

Orphan

(ppid → 1)



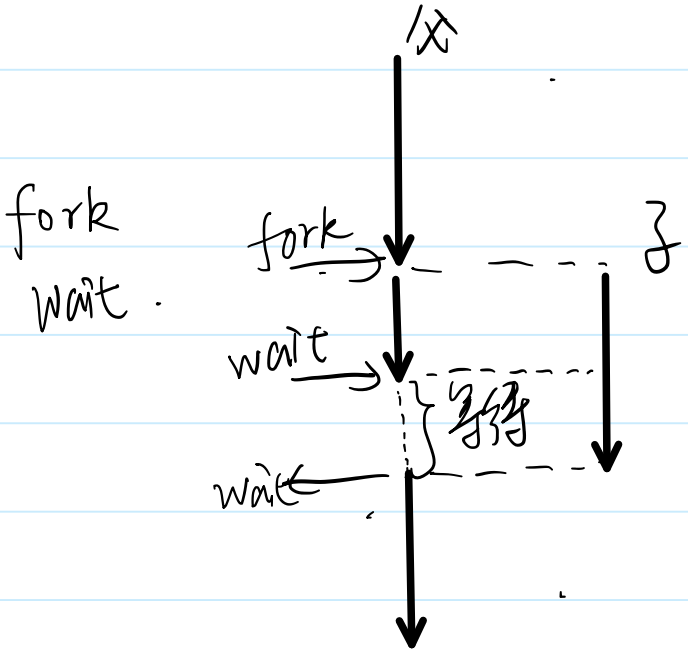
wait

2023年8月11日 11:46

```
pid_t wait(int *wstatus);
```

传出信息参数.

父进程调用wait回收资源., 若子进程未终止, wait陷入等待.




```c
#include <52func.h>
int main(int argc, char *argv[])
{
    if(fork()){
        printf("parent, pid = %d, ppid = %d\n",
                getpid(), getppid());
        //sleep(1);
        wait(NULL);
    }
    else{
        printf("child, pid = %d, ppid = %d\n",
                getpid(), getppid());
    }
    return 0;
}
```

子进程已经终止，父进程没有终止，也没有wait

2023年8月11日 11:54

```
0 S liao        9233    7384  0  80   0 -   624 hrtime 11:56 pts/0    00:00:00 ./6_zombie
1 Z liao        9234    9233  0  80   0 -     0 -      11:56 pts/0    00:00:00 [6_zombie] <defunct>
```

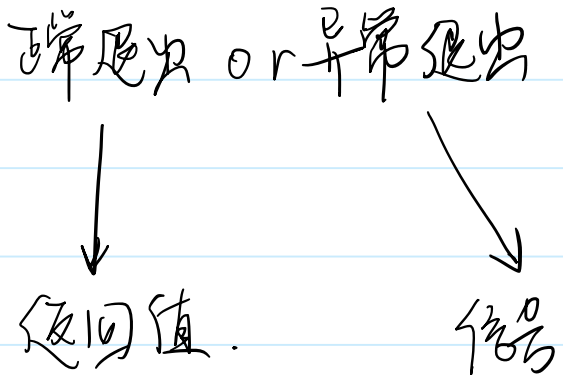
```c
int main(int argc, char *argv[])
{
    if(fork()){
        printf("I am parent\n");
        while(1){
            sleep(1);
        }
    }
    else{
        printf("I am child\n");
    }
    return 0;
}
```

僵尸进程

父进程用wait获取子进程的退出状态

2023年8月11日 14:32

```
pid_t wait(int *wstatus);
```




```
WIFEXITED(wstatus)
        returns true i

WEXITSTATUS(wstatus)
        returns the ex
        specified  in
        only if WIFEXI

WIFSIGNALED(wstatus)
        returns true i

WTERMSIG(wstatus)
        returns the nu
        returned true.

WCOREDUMP(wstatus)
```

2023年8月11日 14:40

```c
int main(int argc, char *argv[])
{
    if(fork()){
        printf("I am parent!\n");
        int status;
        wait(&status);
        if(WIFEXITED(status)){
            printf("normal exit! return value = %d\n", WEXITSTATUS(status));
        }
        else if(WIFSIGNALED(status)){
            printf("abnormal quit! signal = %d\n", WTERMSIG(status));
        }
    }
    else{
        printf("I am child!\n");
        //return 123;
        while(1){
            sleep(1);
        }
    }
    return 0;
}
```

status 32bit { 正常/异常
返回值 < 32bit

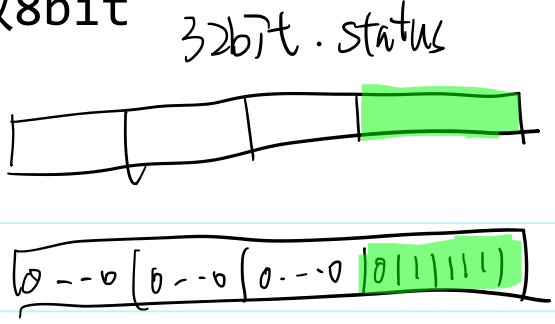
子进程返回 32bit

父进程获取子进程的返回值最多获取8bit

2023年8月11日 14:45

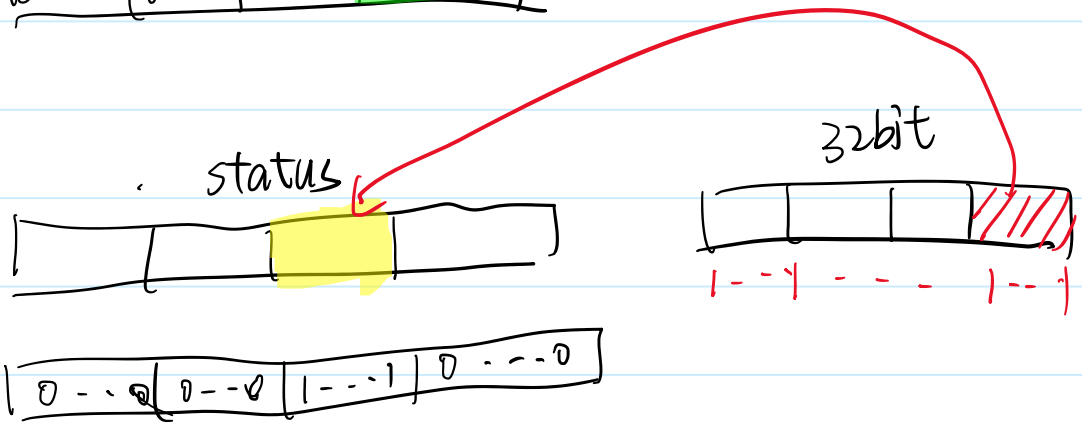
WIFEXITED.

```
(((
t.c"
status
t.c" 3 4
) & 0x7f) == 0)
```



WEXITSTATUS

```
(((

status

) & 0xff00) >> 8)
```



回收指定子进程的资源

2023年8月11日 14:52

```
WNOHANG     return immediately if no child has exited.
```

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

waitpid(-1, &status, WNOHANG)

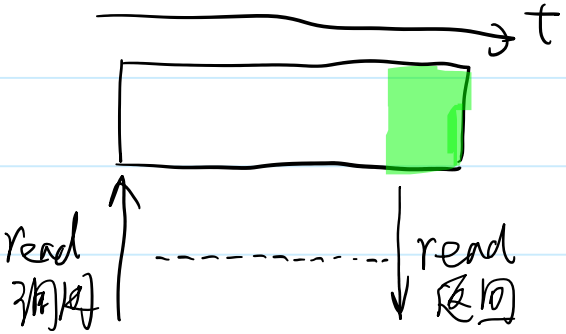
① 子进程已终止 回收资源.

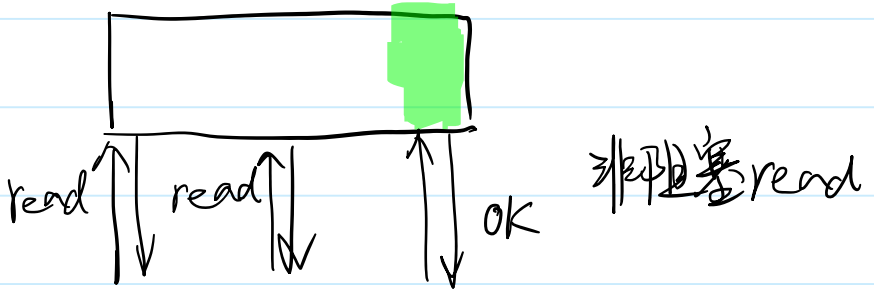
② 未终止. 直接返回

父通过fork的返回值知道了进程pid.

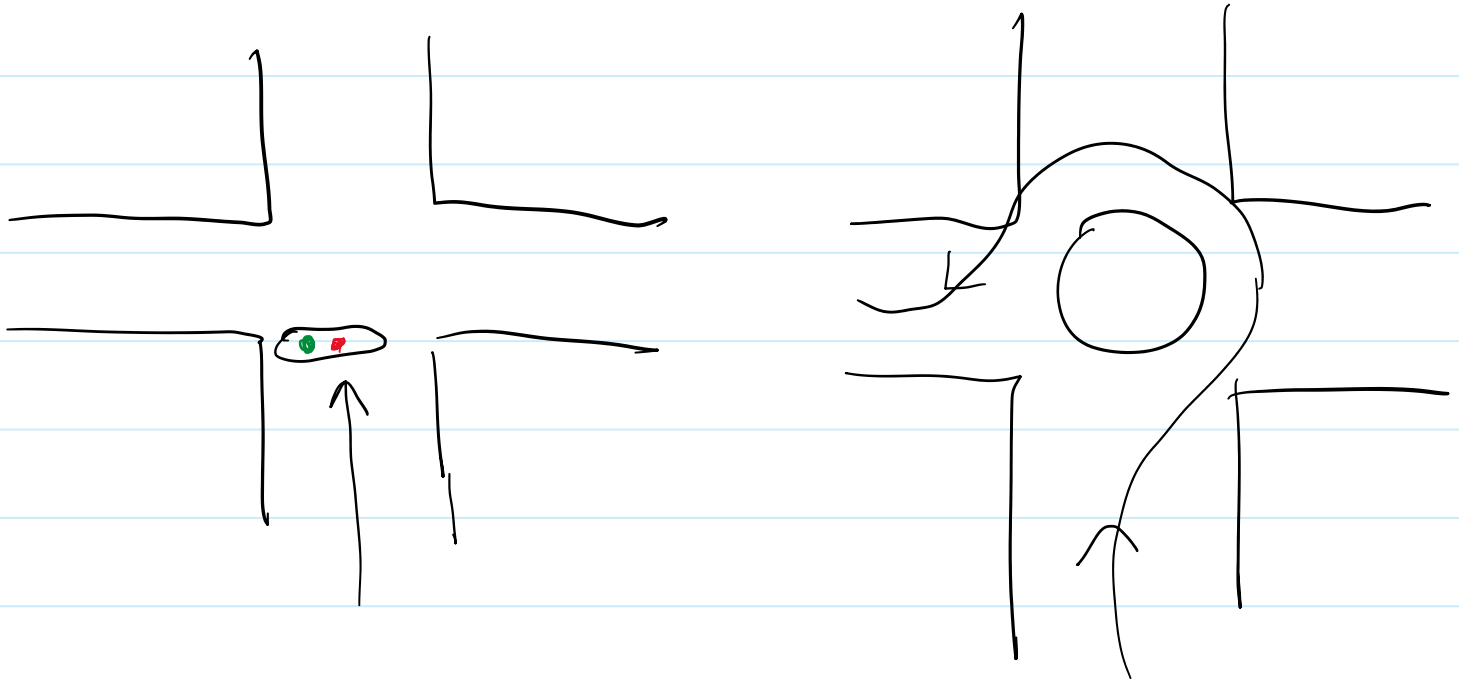
```
-1      meaning wait for any child process.
```

wait 和 waitpid.





2023年8月11日 15:03



非阻塞回收资源

2023年8月11日 15:13

```c
if(fork()){
    int status;
    while(1){
        int ret = waitpid(-1,&status,WNOHANG);//以非阻塞的方式回收资源
        if(ret == 0){
            printf("child has not dead yet!\n");
            sleep(2);
        }
        else{
            break;
        }
    }
    if(WIFEXITED(status)){
        printf("normal exit! ret val = %d\n", WEXITSTATUS(status));
    }
    else if(WIFSIGNALED(status)){
        printf("abnormal quit! signal = %d\n", WTERMSIG(status));
    }
}
```

进程的终止

2023年8月11日 15:15

1、在main函数中执行return.

```
int main(int argc, char *argv[])
{
    printf("Hello\n");
    return -1;
    printf("World\n");
}
```

```
liao:LinuxDay14$ echo $?
255
```

main函数return/exit，先清理所有文件流，再终止.

2、在任何位置 exit.

```
NAME
       exit - cause normal process termination

SYNOPSIS
       #include <stdlib.h>

       void exit(int status);
```

```
int func(){
    //return -1;
    exit(-1);
}
int main(int argc, char *argv[])
{
    printf("Hello\n");
    func();
    printf("World\n");
}
```

_exit _Exit

2023年8月11日 15:26

```
#include <unistd.h>

void _exit(int status);

#include <stdlib.h>

void _Exit(int status);
```

3. 在任何位置调用，直接终止进程

```
int func(){
    //return -1;
    _exit(-1);
}
int main(int argc, char *argv[])
{
    printf("Hello");
    func();
    printf("World\n");
}
```

异常退出

2023年8月11日 15:29

1. "他杀"：收到一个信号.

kill -9. pid. / Ctrl+C. / write已关闭pipe.

2. "自杀"：

```
void abort(void);
```

```c
if(fork()){
    printf("I am parent!\n");
    int status;
    wait(&status);
    if(WIFEXITED(status)){
        printf("normal exit! return value = %d\n", WEXITSTATUS(status));
    }
    else if(WIFSIGNALED(status)){
        printf("abnormal quit! signal = %d\n", WTERMSIG(status));
    }
}
else{
    printf("I am child!\n");
    abort();
}
```

os如何管理多进程

2023年8月11日 15:54

进程组：进程的集合。一个进程属于一个进程组。

组ID：和进程组长PID相同。进程组长终止，组ID不变

父进程 ⟶ 子进程 在子进程刚建，子进程的组ID和父进程的组ID一样。

↓

可以修改。

组长进程不允许更改组ID

获取本进程的组ID

2023年8月11日 15:59

```
pid_t getpgid(pid_t pid);
```

pid填0，获取本进程.

```
int main(int argc, char *argv[])
{
    if(fork()){
        printf("parent, pid = %d, pgid = %d\n",
                getpid(),getpgid(0));
        wait(NULL);
    }
    else{
        printf("child, pid = %d, pgid = %d\n",
                getpid(),getpgid(0));
    }
    return 0;
}
```

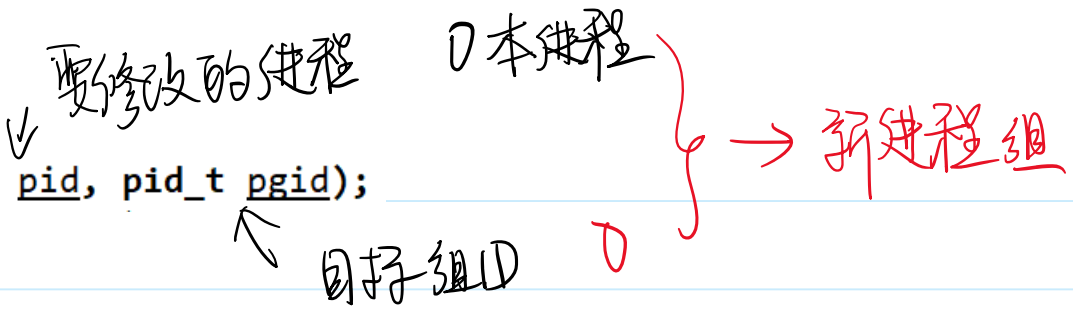
```
liao:LinuxDay14$ ./11_getpgid
parent, pid = 10793, pgid = 10793
child, pid = 10794, pgid = 10793
```

① 子进程刚创建时，pgid和父一样.

② 通过bash启动的进程是一个进程组的组长.

更换组ID

2023年8月11日 16:05




```
int setpgid(pid_t pid, pid_t pgid);
```

```c
int main(int argc, char *argv[])
{
    if(fork()){
        printf("parent, pid = %d, pgid = %d\n",
               getpid(),getpgid(0));
        wait(NULL);
    }
    else{
        printf("before child, pid = %d, pgid = %d\n",
               getpid(),getpgid(0));
        setpgid(0,0);
        printf("after child, pid = %d, pgid = %d\n",
               getpid(),getpgid(0));
    }
    return 0;
}
```

会话 session

2023年8月11日 16:09

会话是进程组的集合.

会话首进程

一个终端对应一个会话. { 前台进程组 最多只有一个.
后台进程组

终端关闭了,与之相连的会话内的所有进程都收到 断开信号.

Ctrl+C Ctrl+\ Ctrl+Z

```c
int main(int argc, char *argv[])
{
    if(fork()){
        printf("parent, pid = %d, pgid = %d\n",
               getpid(),getpgid(0));
        wait(NULL);
    }
    else{
        setpgid(0,0);
        printf("child, pid = %d, pgid = %d\n",
               getpid(),getpgid(0));
        while(1);
    }
    return 0;
}
```

获取sid 新建会话

2023年8月11日 16:18

```
NAME
       getsid - get session ID

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t getsid(pid_t pid);
```

```c
int main(int argc, char *argv[])
{
    printf("sid = %d\n",getsid(0));
    return 0;
}
```

```
setsid - creates a session and sets the process group ID

S
#include <sys/types.h>
#include <unistd.h>

pid_t setsid(void);

TION
setsid()  creates  a  new  session if the calling process is not a process group leader.
```

通过bash启动的进程，不能新建会话

守护进程 daemon

2023年8月11日 16:29

命名以d为后缀

脱离启动环境.

① 创建新会话

② cwd., umask.

③ 断开文件.

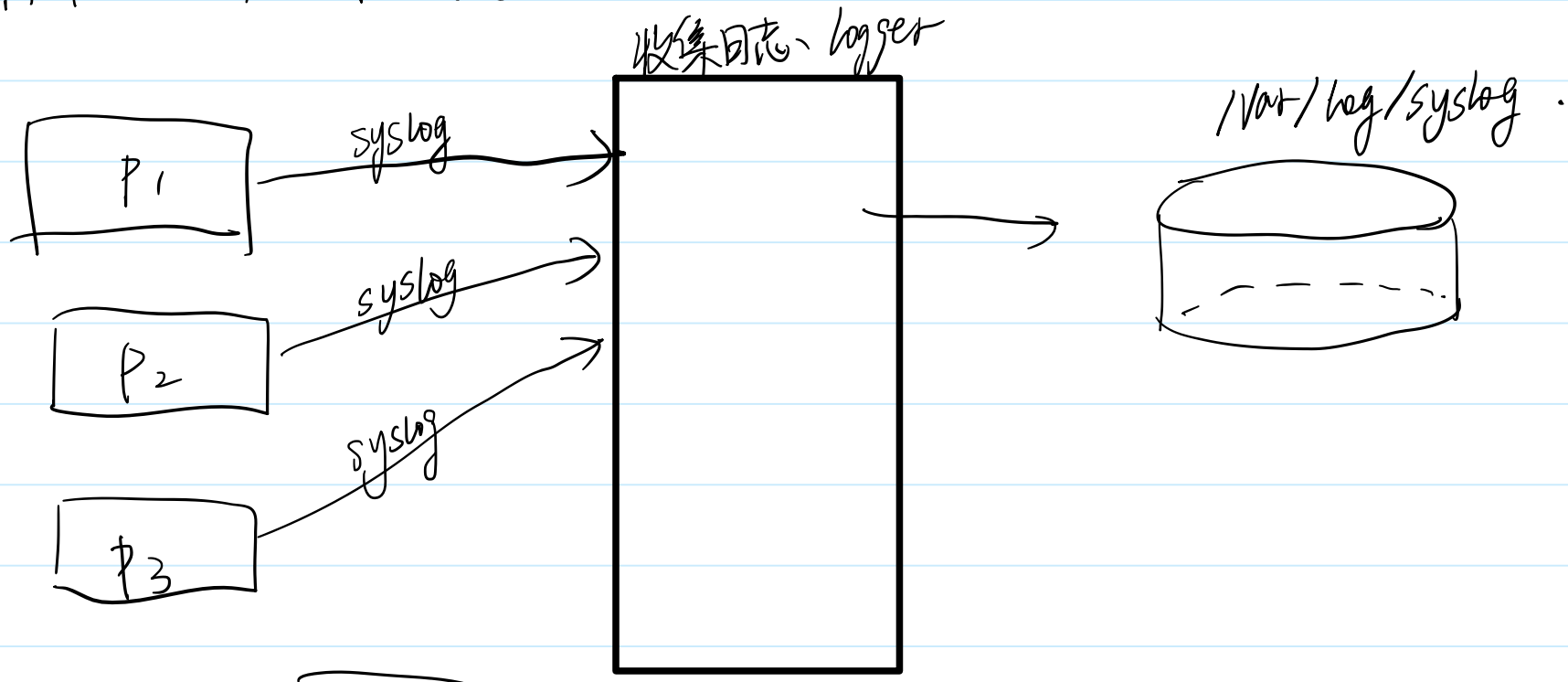
2023年8月11日 16:37

```
void Daemon(){
    // 新建会话
    if(fork()){
        exit(0); //父进程是一个组长，不能新建会话，直接终止
    }
    setsid();
    // 修改和启动环境相关的属性
    chdir("/");
    umask(0);
    // 关闭之前打开的文件
    for(int i = 0; i < 3; ++i){
        close(i);
    }
}
int main(int argc, char *argv[])
{
    Daemon();
    for(int i = 0; i < 100; ++i){
        syslog(LOG_INFO,"i = %d\n", i);
        sleep(2);
    }
    return 0;
}
```

日志系统

2023年8月11日 16:37

操作系统自带的日志系统



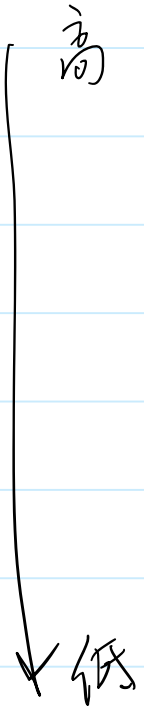

```
void syslog(int priority, const char *format, ...);
```

和printf一样的

日志的优先级别

2023年8月11日 16:47

LOG_EMERG	system is unusable
LOG_ALERT	action must be taken immediately
LOG_CRIT	critical conditions
LOG_ERR	error conditions
LOG_WARNING	warning conditions
LOG_NOTICE	normal, but significant, condition
LOG_INFO	informational message
LOG_DEBUG	debug-level message



gdb的使用

2023年8月11日 17:09

常规命令. gdb a.out　　gdb --args a.out 1 2 3

r/b/n/s/c/l

bt. 展示栈帧

① . 在gdb内重新启动一遍程序

② bt. 调用栈. , 从上往下. 去找到自己写的代码.

③ 定位问题 a. 指针的 *、[]、->

b. 字符数组越界. char buf[512]; read(fd, buf, 40960);

x

多进程下使用gdb

2023年8月11日 17:23

多进程下任何调试器都不好用，并发问题一般靠打日志解决.

```
(gdb) set follow-fork-mode child
(gdb) r
Starting program: /home/liao/cpp52/2_Linux/LinuxDay14/10_abort
[Attaching after process 13104 fork to child process 13105]
[New inferior 2 (process 13105)]
[Detaching after fork from parent process 13104]
I am parent!
[Inferior 1 (process 13104) detached]
[Switching to process 13105]
```

→ parent

valgrind.

进程间通信 → InterProcess Communication IPC

2023年8月11日 17:31

部分地打破 隔离性

管道 { 有名管道
不名管道

共享内存.

信号量 } 不详. 自学.

消息队列

信号 ☆

网络.

无名管道

2023年8月11日 17:35

在文件系统中不存在. ⟹ 只适用于父子进程间.

库函数.

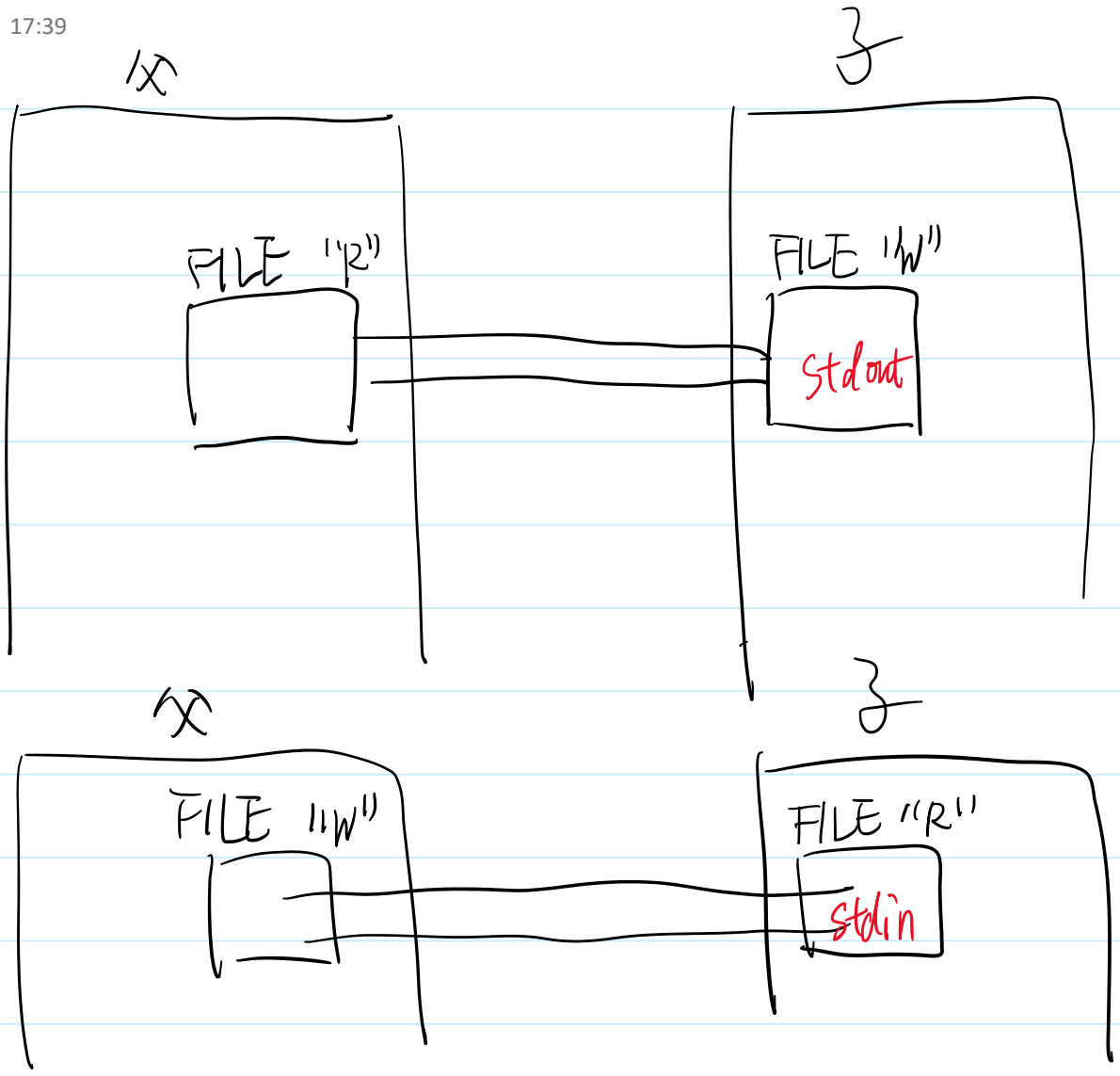
```
NAME
       popen, pclose - pipe stream to or from a process

SYNOPSIS
       #include <stdio.h>

       FILE *popen(const char *command, const char *type);

       int pclose(FILE *stream);
```

① 根据 command, 启动一个子进程.

② 父子连接一条管道.

③ type ⇒ "r", 父可读, 子可写.
⇒ "w", 父可写, 子可读

2023年8月11日 17:39



popen的w模式

2023年8月11日 17:49

```c
int main(int argc, char *argv[])
{
    FILE *fp = popen("./command_stdin","w");
    ERROR_CHECK(fp,NULL,"popen");
    fwrite("5 6",1,3,fp);
    pclose(fp);
    return 0;
}
```

```c
int main(int argc, char *argv[])
{
    int i,j;
    scanf("%d%d",&i,&j);
    printf("child, i + j = %d\n", i+j);
    return 0;
}
```

popen的r模式

2023年8月11日 17:50

```c
int main(int argc, char *argv[])
{
    FILE *fp = popen("./command_stdout","r");
    ERROR_CHECK(fp,NULL,"popen");
    char buf[4096] = {0};
    fread(buf,1,sizeof(buf),fp);
    buf[1] = 'E';
    printf("buf = %s\n", buf);
    pclose(fp);
    return 0;
}
```

```c
int main(int argc, char *argv[])
{
    printf("Hello world!\n");
    return 0;
}
```