

`gdb`

2023年4月27日

9:14

使用 `gdb` 调试多进程

```
set follow-fork-mode parent
```

```
(gdb) set follow-fork-mode child
```

```
(gdb) r
```

```
starting program: /home/liao/49code/2_Linux/Linuxday_13/0_gdb
```

```
[Attaching after process 48144 fork to child process 48145]
```

```
[New inferior 2 (process 48145)]
```

```
[Detaching after fork from parent process 48144]
```

```
world!
```

```
[Inferior 1 (process 48144) detached]
```

```
[Switching to process 48145]
```

```
Thread 2.1 "0_gdb" hit Breakpoint 1, main () at 0_gdb.c:4
```

```
↳      printf("Hello\n");
```

```
↳      █
```

定位段错误

2023年4月27日 9:55

1. 在gdb中启动程序

2. 触发报错 使用bt.

3. 从上向下找堆栈, 找到第一个自定义的代码.

4. 在代码处打断点, 重新运行

① 参数

② errno.

③ 内存.

进程间通信 → Inter-Processes Communication

2023年4月27日 10:11

IPC

进程间：隔离性

✓ 管道

✓ 共享内存

信号量 } 自锁.
消息队列 }

✓ 信使

套接字. socket 网络

管道

2023年4月27日 10:19

匿名管道.

→ 在文件系统中无路径.

→ 只允许有血缘关系的进程通信.

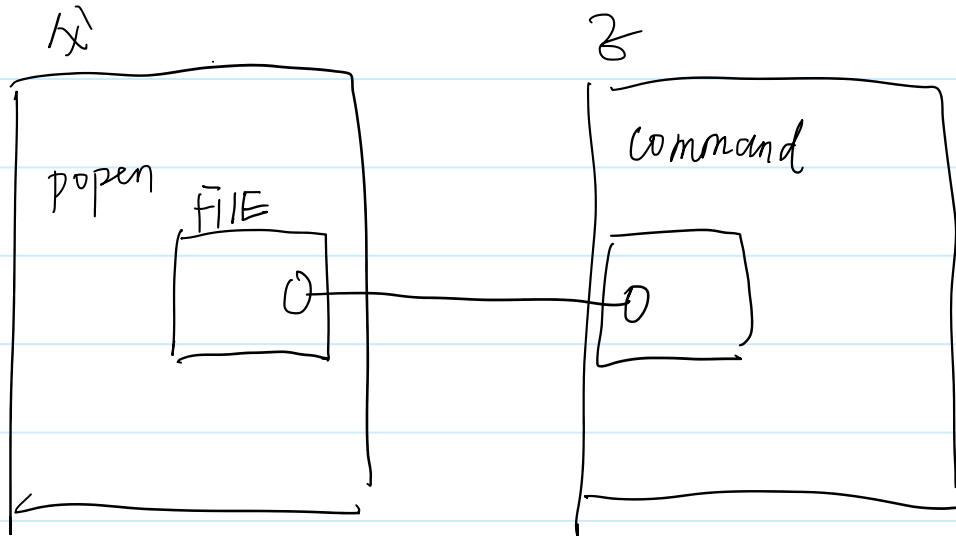
库函数

```
#include <stdio.h>
```

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

```
int pclose(FILE *stream);
```

用系统调用 (system)



"w" 父进程写端 子进程重定向 stdin
"r" 父进程读端 子进程重定向 stdout

popen的w模式

2023年4月27日 10:32

2_popen_w.c

```
1 #include <49func.h>
2 int main()
3 {
4     FILE *fp = popen("./add", "w");
5     ERROR_CHECK(fp, NULL, "popen");
6     fwrite("3 4", 1, 3, fp);
7     pclose(fp);
8     return 0;
9 }
10
```

```
int main()
{
    int lhs, rhs;
    scanf("%d%d", &lhs, &rhs);
    printf("lhs + rhs = %d\n", lhs+rhs);
    return 0;
}
```

popen的r模式

2023年4月27日 11:06

3_popen_r.c

```
1 #include <49func.h>
2 int main()
3 {
4     FILE *fp = popen("./hello", "r");
5     ERROR_CHECK(fp, NULL, "popen");
6     char buf[1024] = {0};
7     fread(buf, 1, sizeof(buf), fp);
8     printf("buf = %s\n", buf);
9     pclose(fp);
10    return 0;
11 }
```

创建管道的系统调用

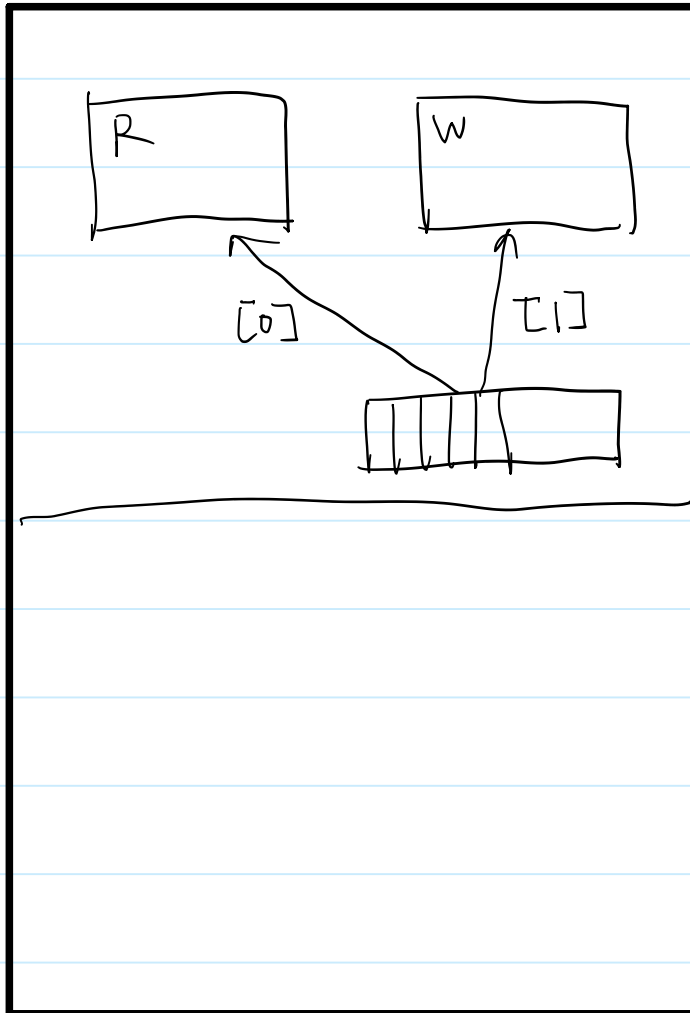
2023年4月27日

11:13

2 是一个提示, 希望传入一个长度为2的数组

pipe

```
int pipe(int pipefd[2]);
```



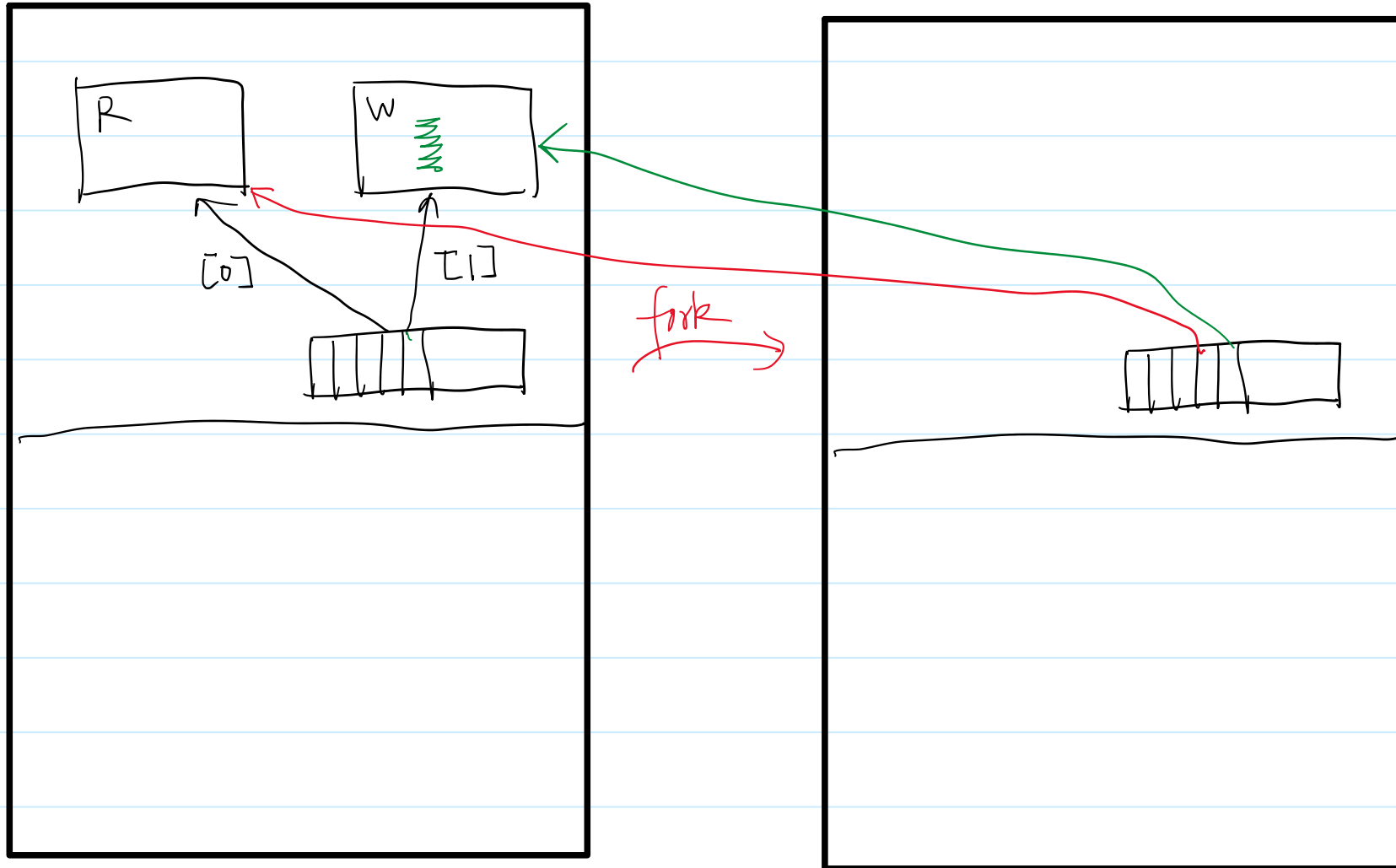
4_pipe.c

```
1 #include <49func.h>
2 int main()
3 {
4     int fds[2];
5     // pipe的参数一定是一个数组名 pipe(fds[2])
6     pipe(fds);
7     printf("fds[0] = %d, fds[1] = %d\n", fds[0], fds[1]);
8     // [0] --> 读端
9     // [1] --> 写端
10    write(fds[1], "hello", 5);
11    char buf[1024] = {0};
12    read(fds[0], buf, sizeof(buf));
13    printf("buf = %s\n", buf);
14    return 0;
15 }
```

fork配合pipe

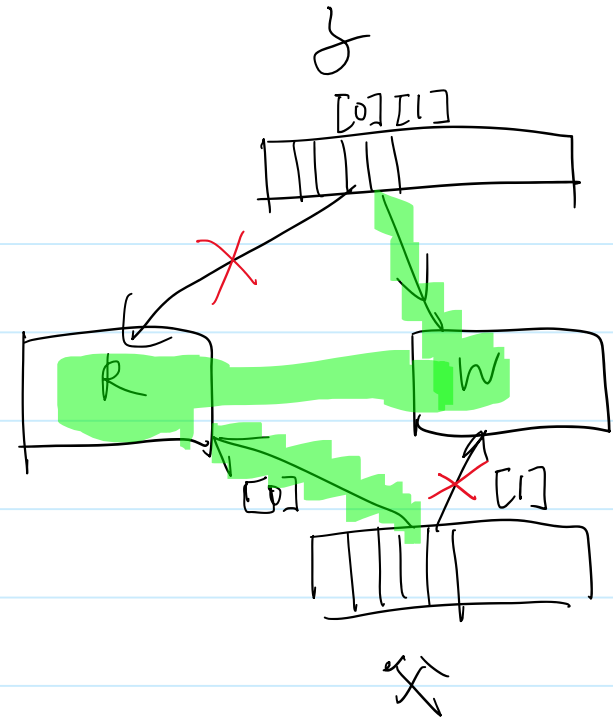
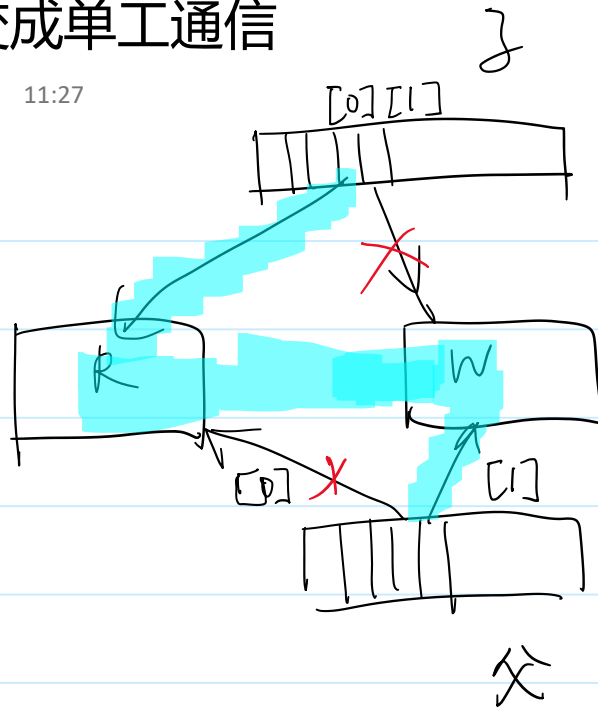
2023年4月27日 11:23

先pipe再fork



把管道变成单工通信

2023年4月27日 11:27

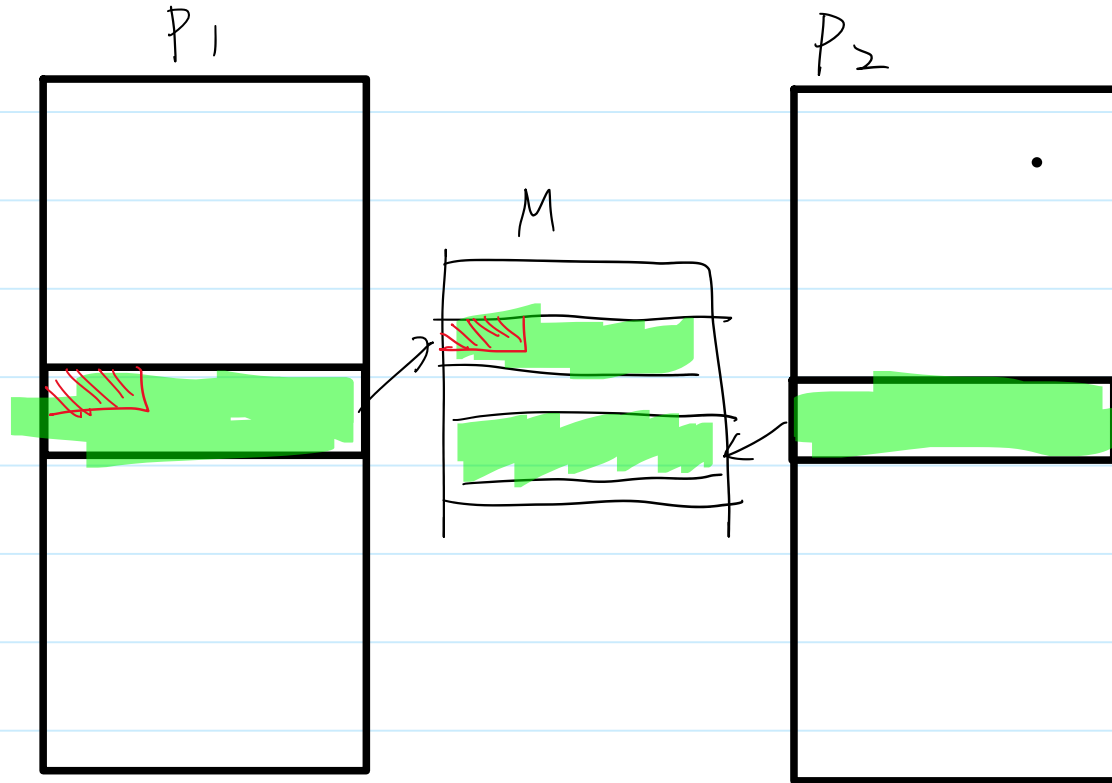


实现双工通信 采用多条管道的策略.

```
{  
    int fds[2];  
    // pipe的参数一定是一个数组名 pipe(fds[2])  
    pipe(fds);  
    printf("fds[0] = %d, fds[1] = %d\n",fds[0], fds[1]);  
    if(fork() == 0){  
        //父向子发送数据  
        // 子关闭写端  
        close(fds[1]);  
        char buf[1024] = {0};  
        read(fds[0],buf,sizeof(buf));  
        printf("buf = %s\n",buf);  
        close(fds[0]);  
        exit(0);  
    }  
    else{  
        // 父关闭读端  
        close(fds[0]);  
        sleep(10);  
        write(fds[1],"hello",5);  
        wait(NULL);  
        close(fds[1]);  
        exit(0);  
    }  
    return 0;  
}
```

共享内存

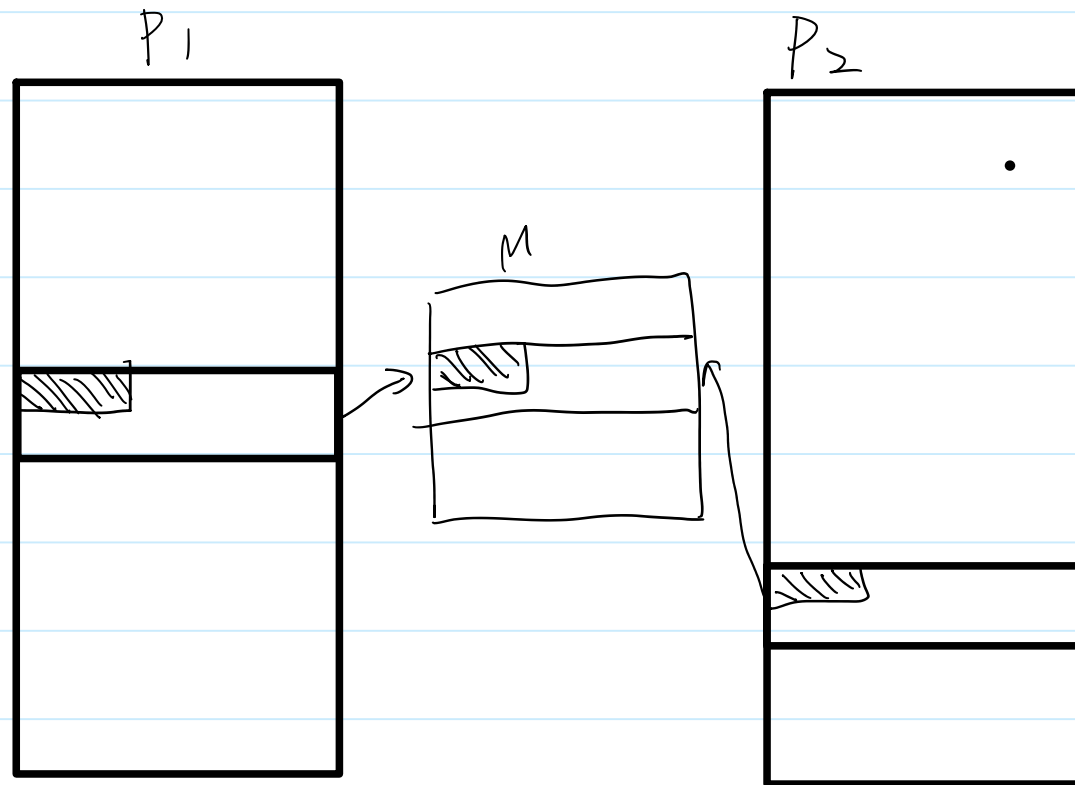
2023年4月27日 11:40



特殊设计

2023年4月27日 11:43

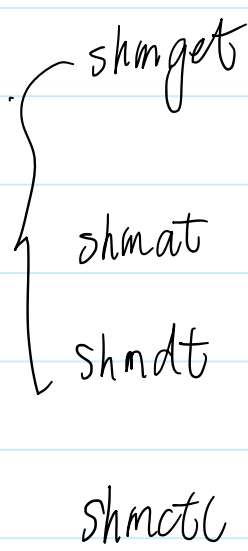
让不同进程的虚拟页，映射到同一个页框



shm system V 共享内存系统调用

2023年4月27日 11:48

| | |
|-----------------|--|
| shmat (2) | - System V shared memory operations |
| shmat (3posix) | - XSI shared memory attach operation |
| shmctl (2) | - System V shared memory control |
| shmctl (3posix) | - XSI shared memory control operations |
| shmdt (2) | - System V shared memory operations |
| shmdt (3posix) | - XSI shared memory detach operation |
| shmget (2) | - allocates a System V shared memory segment |
| shmget (3posix) | - get an XSI shared memory segment |
| shmop (2) | - System V shared memory operations |

shmget 创建 or 获取
shmat 把共享内存加载到地址空间内
shmdt 释放虚拟内存
shmctl 管理

shmget 创建or获取

2023年4月27日

14:30

`int shmget(key_t key, size_t size, int shmflg);`

共享内存标识 大小 属性 IPC_CREAT

低9位 - 权限

id. → 第一次 shmget 创建共享内存, 内存中一开始每个bit都是0.

`void *shmat(int shmid, const void *shmaddr, int shmflg);`

`int shmdt(const void *shmaddr);`

```
int main()
{
    int shmid = shmget(0x1235, 4096, IPC_CREAT | 0600);
    ERROR_CHECK(shmid, -1, "shmget");
    char *p = (char *)shmat(shmid, NULL, 0);
    printf("p = %s\n", p);
    shmdt(p);
    return 0;
}
```

共享内存的删除

2023年4月27日 15:00

```
[liao@ubuntu Linuxday 13]$ ipcrm -a  
[liao@ubuntu Linuxday 13]$ ipcs
```

```
----- Message Queues -----  
key          msqid      owner          perms  
  
----- Shared Memory Segments -----  
key          shmid      owner          perms  
0x00000000  3              liao          600  
  
----- Semaphore Arrays -----  
key          semid      owner          perms
```

标记删除

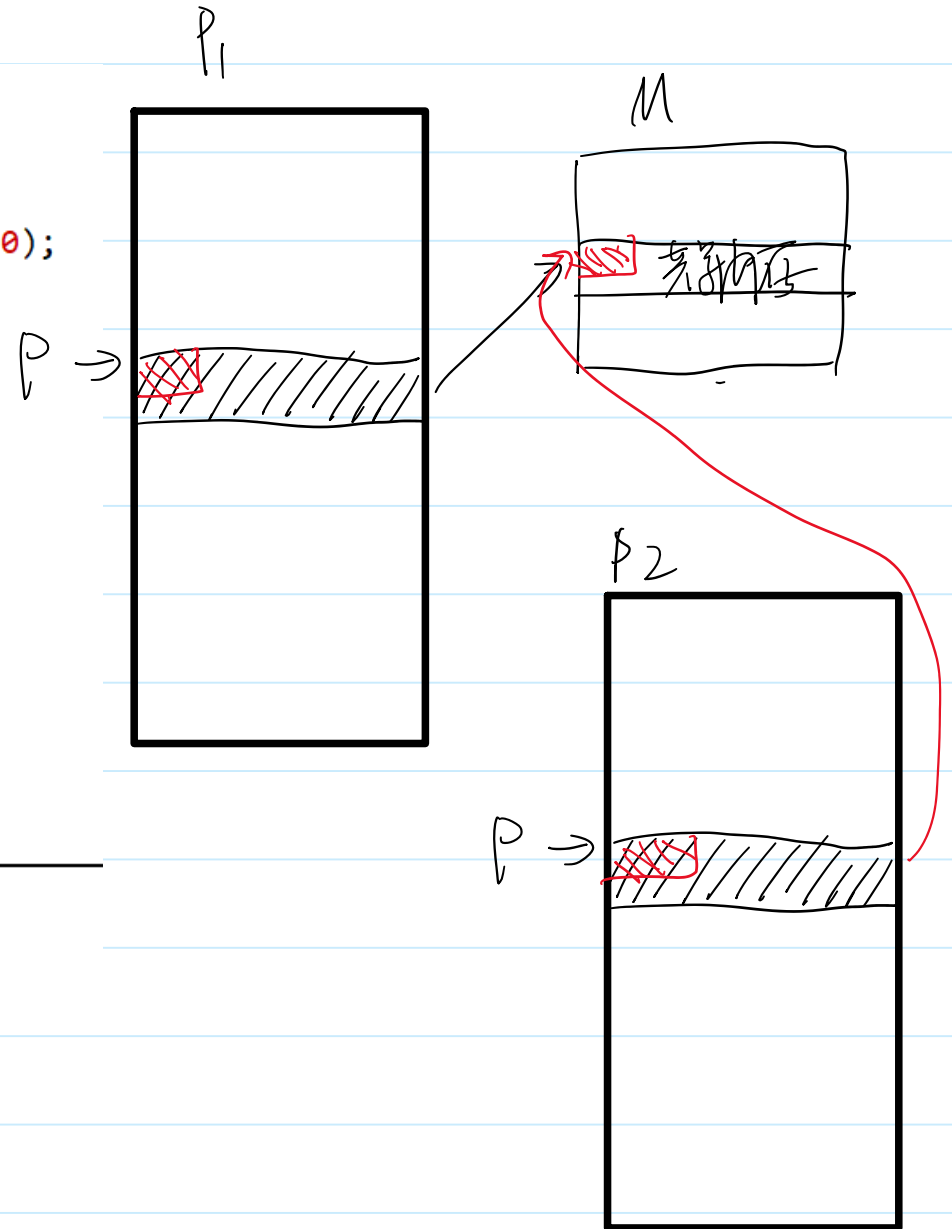
| used-bytes | messages | |
|------------|----------|--------|
| bytes | nattch | status |
| 4096 | 1 | dest |



私有共享内存

2023年4月27日 15:01

```
#include <sys/types.h>
#define NUM 10000
int main()
{
    int shmid = shmget(IPC_PRIVATE, 4096, IPC_CREAT | 0600);
    ERROR_CHECK(shmid, -1, "shmget");
    int *p = (int *)shmat(shmid, NULL, 0);
    p[0] = 0;
    if(fork() == 0){
        for(int i = 0; i < NUM; ++i){
            ++p[0];
        }
    }
    else{
        for(int i = 0; i < NUM; ++i){
            ++p[0];
        }
        wait(NULL);
        printf("p[0] = %d\n", p[0]);
    }
    return 0;
}
```



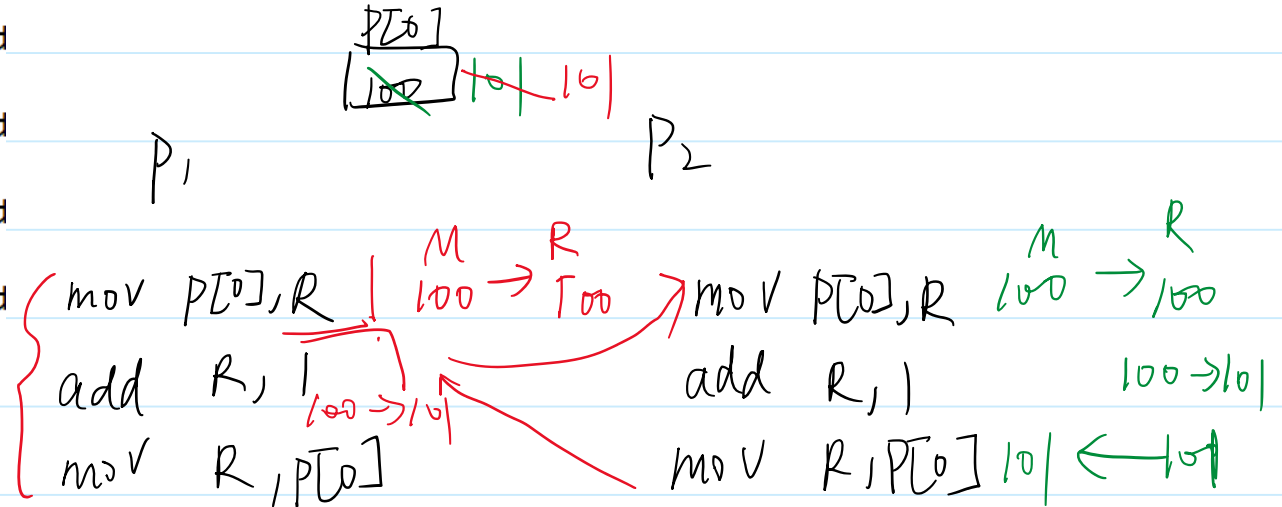
竞争条件

2023年4月27日

15:12

两个并发的进程访问共享资源

```
[liao@ubuntu Linuxday 13]$ ./8_add
p[0] = 16176037
[liao@ubuntu Linuxday 13]$ ./8_add
p[0] = 17988082
[liao@ubuntu Linuxday 13]$ ./8_add
p[0] = 20000000
[liao@ubuntu Linuxday 13]$ ./8_add
p[0] = 18825142
```

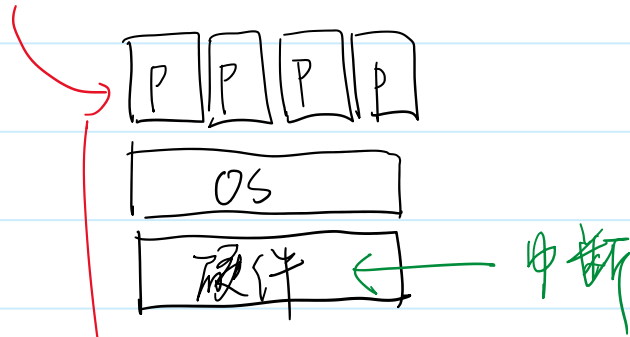


信号

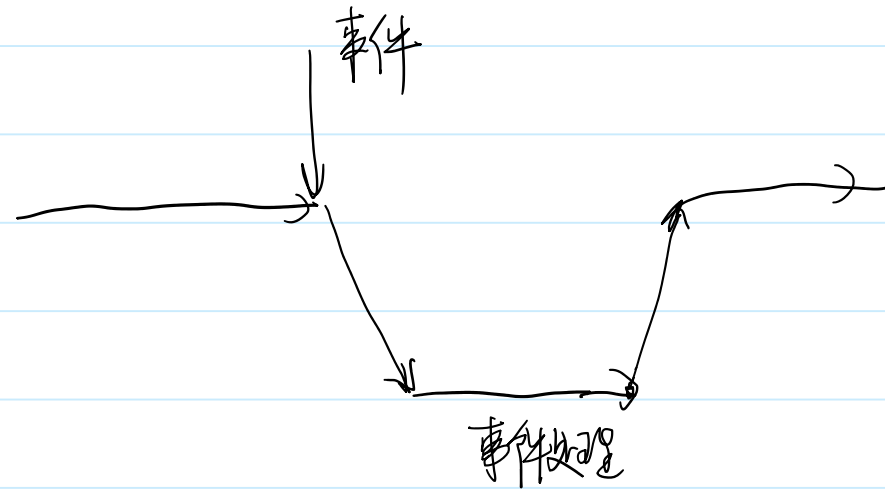
2023年4月27日

15:51

软件层面上的事件机制



信号的目标是进程



信号的产生

2023年4月27日 16:03

底层原理

修改目标进程的 task_struct 的 signal 字段

硬件 ctrl+c 异常

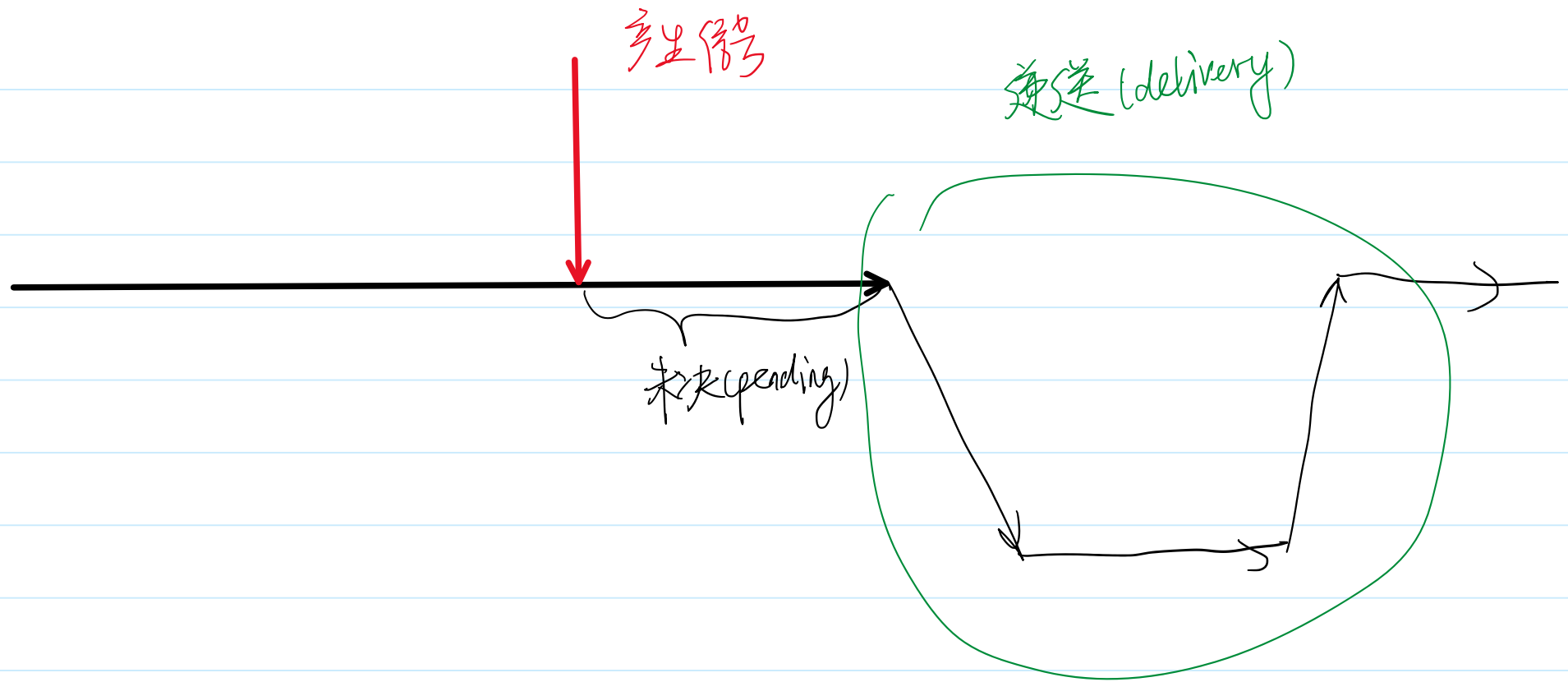
硬件 除0 同步

软件 kill 异常

软件 abort 同步

信号的处理

2023年4月27日 16:10



信号的分类

2023年4月27日 16:15

```
[liao@ubuntu Linuxday 13]$ kill -l
```

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX

Handwritten notes: Ctrl+C points to SIGINT, Ctrl+Z points to SIGQUIT.

```
$ man 7 signal
```

默认递送行为

2023年4月27日 16:21

Term Default action is to terminate the process.

Ign Default action is to ignore the signal.

Core Default action is to terminate the process and dump core (see `core(5)`).

Stop Default action is to stop the process. → Ctrl+C

Cont Default action is to continue the process if it is currently stopped.

Program received signal SIGSEGV, Segmentation fault.
__strlen_avx2 () at ../sysdeps/x86_64/multiarch/strlen-avx2.S:65
65 ../sysdeps/x86_64/multiarch/strlen-avx2.S: No such file or directory.

生成core文件

更换信号的递送行为

2023年4月27日

16:27

注册

The signals **SIGKILL** and SIGSTOP cannot be caught, blocked, or ignored.

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);
```

函数指针.

```
sighandler_t signal(int signum, sighandler_t handler);
```

} 简介

```
void (*signal(int sig, void (*func)(int)))(int);
```

`int *p[5];` → 长度为5的数组, 成员是int

`int (*p)[5];` → 指针, 其基型是 `int[5]`

`int *func(int);` → `func` 是一个函数. 返回类型是 `int *`

`int (*pf)(int);` → `pf` 是一个指针, 其基型是函数.

回调函数

2023年4月27日

16:39

我

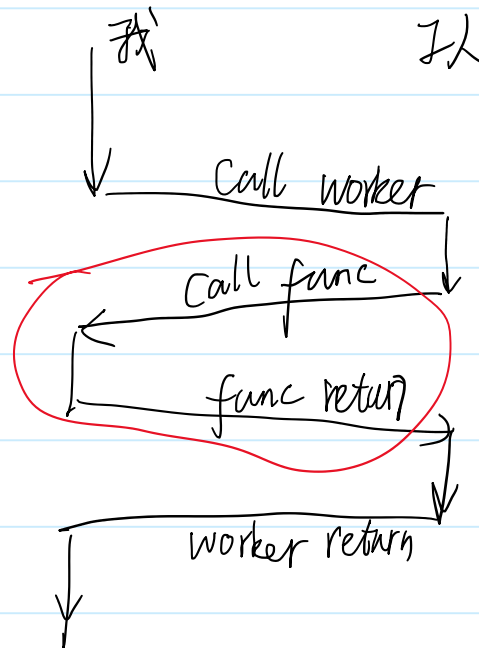
```
void func1() { ... }

int main() {
    worker (func);
}
```

工人

```
void work (func) {
    ...
    func();
}
```

回调函数
callback



基于函数指针实现
用户去设计函数的定义
被调函数为回调函数。

注册信号

2023年4月27日 17:13

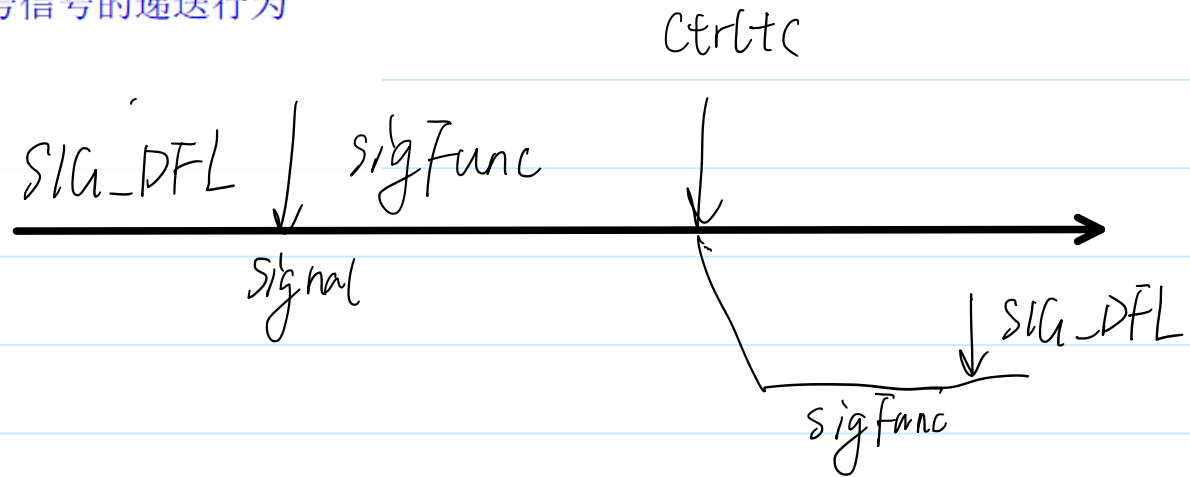
修改信号的递送行为.

```
9_signal.c
1 #include <49func.h>
2 void sigFunc(int signum){
3     printf("signum = %d\n",signum);
4 }
5 int main()
6 {
7     sleep(10);
8     printf("sleep over!\n");
9     signal(2,sigFunc);//修改2号信号的递送行为
10    while(1);
11    return 0;
12 }
```

在递送过程中注册

2023年4月27日 17:28

```
#include <49func.h>
void sigFunc(int signum){
    printf("signum = %d\n",signum);
    signal(SIGINT,SIG_DFL);
}
int main()
{
    //sleep(10);
    //printf("sleep over!\n");
    → signal(2,sigFunc); //修改2号信号的递送行为
    while(1);
    return 0;
}
```



如果递送的时间比较长

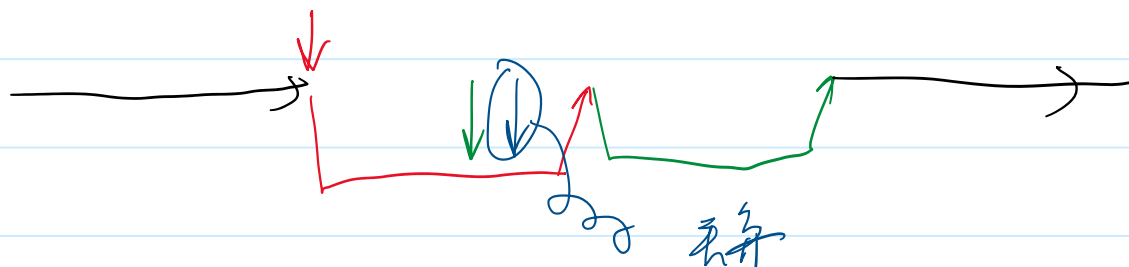
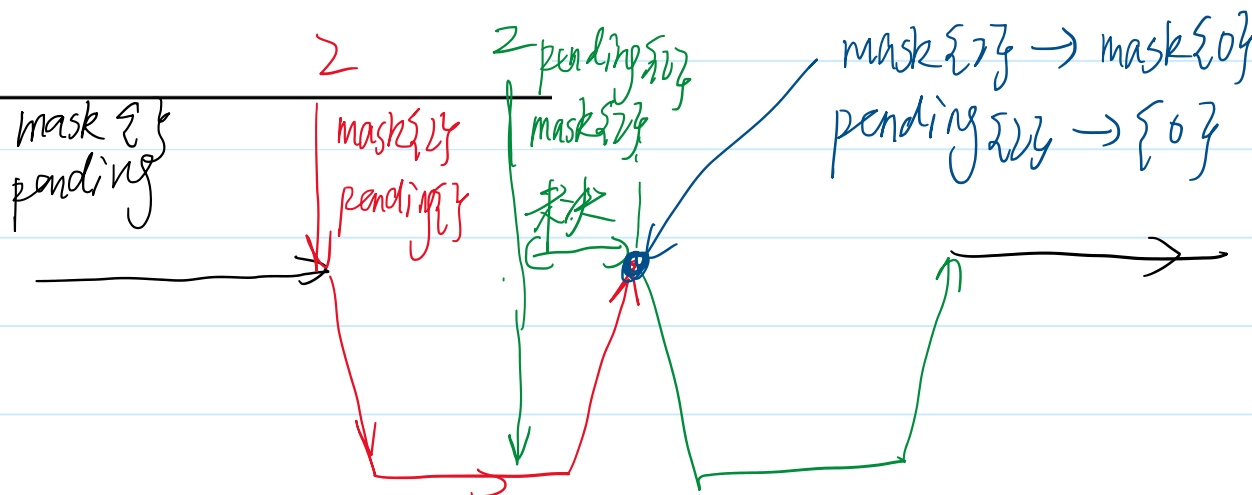
2023年4月27日 17:35

```
#include <49func.h>
void sigFunc(int signum){
    printf("before, signum = %d\n", signum);
    sleep(5);
    printf("after , signum = %d\n", signum);
}
int main()
{
    signal(SIGINT, sigFunc);
    while(1){
        sleep(1);
    }
    return 0;
}
```

^Cbefore, signum = 2
^Cafter , signum = 2
before, signum = 2
after , signum = 2
□

^Cbefore, signum = 2
^C^Cafter , signum = 2
before, signum = 2
after , signum = 2
■

递送信号中, 会在mask中临时增加



after , signum = 2

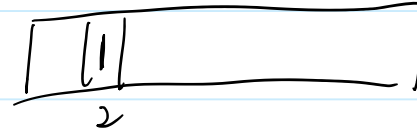
子群

mask 和 pending

2023年4月27日 17:44

位图.

mask



mask. 产生某个信号时, 是否阻塞之.

pending. 未决信号集

在阻塞某个信号期间, 产生了后续信号, 移入 pending

当 mask 改变时, 某个信号解除阻塞, 观察 pending, 若有后续信号, 就取出并递送.