第十五章 进程间通信

习题：

15-1、结果及原因如下；

(1) 当读一个写端已被关闭的管道时，在所有数据都被读取后，read返回0，以指示达到了文件结束处。(从技术方面考虑，管道的写端还有进程时，就不会产生文件的结束。可以复制一个管道的描述符，使得有多个进程对它具有写打开文件描述符。但是，通常一个管道只有一个读进程、一个写进程。下一节介绍FIFO时，我们会看到对于一个单一的FIFO常常有多个写进程。)

```
tongue aupe # grep -B1 waitpid\(pid 15-1.c
//          close(fd[1]);
        if (0 > waitpid(pid, NULL, 0)) {
```

```
tongue aupe # gcc 15-1.c -o 15-1
tongue aupe # ./15-1 /etc/resolv.conf
```

```
# Generated by dhcpcd from eth0
# /etc/resolv.conf.head can replace this line
nameserver 192.168.100.160
nameserver 8.8.8.8
nameserver 168.95.1.1
# /etc/resolv.conf.tail can replace this line
      按q没反应，ctrl+c强制结束
```

```
tongue aupe # grep -B1 waitpid\(pid 15-1.c
        close(fd[1]);
        if (0 > waitpid(pid, NULL, 0)) {
tongue aupe # gcc 15-1.c -o 15-1
tongue aupe # ./15-1 /etc/resolv.conf
```

```
# Generated by dhcpcd from eth0
# /etc/resolv.conf.head can replace this line
nameserver 192.168.100.160
nameserver 8.8.8.8
nameserver 168.95.1.1
# /etc/resolv.conf.tail can replace this line
lines 1-6/6 (END)
```
close(fd[1])注释前的正常效果

15-2、有三种情况的存在；

1、子进程读到父进程写入管道的数据，execl并退出，接着父进程退出；(正常模式)

2、子进程读到父进程写入管道的数据，父进程退出，子进程被领养，接着退出；(非期望模式)

3、父进程在管道中写入数据，父进程退出，子进程被领养，接着退出；(非期望模式)

注意：子进程一旦被领养，就会失去控制终端，即用户无法在终端上看到其输出的信息；

```
tongue aupe # grep -B1 -A3 waitpid\(pid 15-1.c
        close(fd[1]);
//          if (0 > waitpid(pid, NULL, 0)) {
//              printf("waitpid error\n"); _exit(-1);
//          }
        exit(0);
```

```
tongue aupe # grep -B1 execl\( 15-1.c
        printf("Child ppid: %d\n", getppid());
        if (0 > execl(pager, argv0, (char *)0)) {
```
为方便理解加入该行

```
tongue aupe # gcc 15-1.c -o 15-1
tongue aupe # ./15-1 /etc/resolv.conf
Child ppid: 1
```
父进程先于子进程退出

15-3、如下；

*cmdstring*由Bourne shell以下列方式执行：

`sh -c cmdstring`

```c
#include <stdio.h>
#include <unistd.h> // For pipe;

#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    FILE *fpin;
    char line[MAXLINE];
    fpin = popen(argv[1], "r");
    system(argv[1]);
    while(NULL != fgets(line, MAXLINE, fpin)) {
        if (EOF == fputs(line, stdout)) {
            printf("fputs error to stdout\n"); _exit(-1);
        }
    }
    pclose(fpin);
    return 0;
}
```

/opt/drill_ground/aupe/15-3.c [FORMAT=unix:utf-8] [TYPE=C]

```
tongue aupe # gcc 15-3.c -o 15-3
tongue aupe # ./15-3 "lsx"
sh: lsx: 未找到命令   该行由system返回
sh: lsx: 未找到命令
tongue aupe # sh -c lsx
sh: lsx: 未找到命令
```

15-4、通过返回值来判断；

(2) 如果写一个读端已被关闭的管道，则产生信号SIGPIPE。如果忽略该信号或者捕捉该信号并从其处理程序返回，则write返回-1，errno设置为EPIPE。

当用中断信号终止sleep时，pr_exit函数（见程序清单8-3）认为它正常终止。当用退出键杀死sleep进程时，会发生同样的事情。终止状态130、131又是怎样得到的呢？原来Bourne shell有一个在其文档中没有说清楚的特性，其终止状态是128加上一个信号编号，该信号终止了正在执行的命令。用交互方式使用shell可以看到这一点。

346

程序中_exit(1);是基于代码清单15-9新加的。

```
tongue aupe # egrep '^\/\/|^ *\/\/' 15-4.c; egrep -B2 'close\(fd1\[1' 15-4.c;
    // if (SIG_ERR == signal(SIGPIPE, sig_pipe)) {
    //     printf("signal error\n"); _exit(-1);
    // }
    } else {
        _exit(1);
        close(fd1[1]);
```

匹配文件中的两部分

```
tongue aupe # gcc 15-4.c -o 15-4
tongue aupe # ./15-4
a
tongue aupe # echo $?
141
tongue aupe # echo $((141-128))
13
tongue aupe # grep SIGPIPE /usr/include/bits/signum.h
#define SIGPIPE         13      /* Broken pipe (POSIX).  */
```

15-5、基于程序清单15-9修改的部分；

```
    } else if (0 < pid) {
        close(fd1[0]);
        close(fd2[1]);
        fp1_1 = fdopen(fd1[1], "w");
        fp2_0 = fdopen(fd2[0], "r");
        setvbuf(fp1_1, NULL, _IONBF, 0);
        // fcntl(fp2_0->_fileno, F_SETFL, O_NONBLOCK);      // 若用fread请开启此行;
        while (NULL != fgets(line, MAXLINE, stdin)) {
            n = strlen(line);
            if (n != fwrite(line, sizeof(char), n, fp1_1)) {
                printf("write error to pipe\n"); _exit(-1);
            }
            // 使用fread，关于读取多少返回的大小，无法提前获知;
            // 所以需要使用非阻塞模式，并且在读前等待些许时间，
            // 好让缓冲区在读前有时间被写入数据;
            // 若不然，fread将会一直阻塞，指到读取MAXLINE字节数据才返回;
            // sleep(1);    // 若用fread请开启此行;
            // if (0 > (n = fread(line, sizeof(char), MAXLINE, fp2_0))) {
            // 建议使用fgets，阻塞读取，直至返回1行;
            if (NULL == fgets(line, MAXLINE, fp2_0)) {
            // if (0 > (n = read(fd2[0], line, MAXLINE))) {
                printf("read error from pipe\n"); _exit(-1);
            }
            if (0 == n) {
/opt/drill_ground/aupe/15-5.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=031/84(3
```

```
tongue aupe # gcc -g 15-5.c -o 15-5
tongue aupe # ./15-5
1+2
3
3+3
6
^C
```

15-6、第一个问题原因如下图红字描述，第二个问题方案在第二图；

```
if ((fp = popen("/bin/true", "r")) == NULL)
    ...
if ((rc = system("sleep 100")) == -1)
    ...
if (pclose(fp) == -1)
    ...
```

若不用waitpid，pclose无法专门针对popen创建的子进程做阻塞等待操作，很可能等到的结束进程是system所创建的

```
    childpid[fileno(fp)] = pid; /* remember child pid for this fd */
    return(fp);
}
int
pclose(FILE *fp)
{
    int     fd, stat;
    pid_t   pid;

    if (childpid == NULL) {
        errno = EINVAL;
        return(-1);     /* popen() has never been called */
    }

    fd = fileno(fp);
    if ((pid = childpid[fd]) == 0) {
        errno = EINVAL;
        return(-1);     /* fp wasn't opened by popen() */
    }

    childpid[fd] = 0;
    if (fclose(fp) == EOF)
        return(-1);

    while (waitpid(pid, &stat, 0) < 0)
        if (errno != EINTR)
            return(-1); /* error other than EINTR from waitpid() */

    return(stat);   /* return child's termination status */
}
```

因为wait的返回值是所等到结束子进程的pid，所以可以根据wait返回的pid与之前记录的子进程pid做比较，若相等则返回，否则继续等待，直至达到预期

15-7、父子进程间用两个管道打通，值得注意的是，由于在下面select、poll中，写是一直可用的，故而可能会出现父进程while循环写A管道n次，子进程才可能由select返回一次，读取A管道数据，并写入B管道，父进程再等待从B管道读出数据；

```c
#include <stdio.h>
#include <unistd.h>     // For pipe;
#include <string.h>     // For bzero;
#include <sys/select.h>

#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    int fd0[2], fd1[2], fd2[2], rst_n, fd_counter, w_counter = 1, w_cycle = 3, i, n;
    fd_set rset, wset;
    char line_buf[MAXLINE];
    bzero(line_buf, MAXLINE);
    pid_t pid;
    if (2 <= argc) {
        w_cycle = atoi(argv[1]);
    }
    // select中，单个读、写文件描述符的关闭不具代表性，故多添加一个；
    pipe(fd0);
    pipe(fd1);
    pipe(fd2);
```

```c
    if (0 < (pid = fork())) {
        close(fd1[1]);
        close(fd2[0]);
        FD_ZERO(&rset);
        FD_ZERO(&wset);
        FD_SET(fd1[0], &rset);
        FD_SET(fd0[1], &wset);
        FD_SET(fd2[1], &wset);
        fd_counter = fd2[1] + 1;
        sleep(1);
        while(1) {
            rst_n = select(fd_counter, NULL, &wset, NULL, NULL);
            if (0 < rst_n) {
                for(i=0; i < fd_counter; i++) {
                    if (FD_ISSET(i, &wset) && i != fd0[1]) {
                        n = write(i, "something...", sizeof("something..."));
                        printf("write: something...\n");
                    }
                }
            } else if (0 == rst_n) {
                printf("Interrupted due to timeout!\n"); break;
            } else {
                fprintf(stderr, "Interrupted due to a signal!\n"); break;
            }
```

```
            rst_n = select(fd_counter, &rset, NULL, NULL, NULL);
            if (0 < rst_n) {
                for(i=0; i < fd_counter; i++) {
                    if (FD_ISSET(i, &rset)) {
                        bzero(line_buf, MAXLINE);
                        n = read(i, line_buf, MAXLINE);
                        printf("read: %s\n", line_buf);
                    }
                }
            }
            w_counter++;
            if (w_cycle <= w_counter) {
                close(fd2[1]);
                printf("close fd2[1] w_counter: %d\n", w_counter);
            }
        }
        waitpid(pid, NULL, 0);
```

```
    } else {
        close(fd1[0]);
        close(fd2[1]);
        FD_ZERO(&rset);
        FD_SET(fd0[0], &rset);
        FD_SET(fd2[0], &rset);
        fd_counter = fd2[0] + 1;
        while(1) {
            rst_n = select(fd_counter, &rset, NULL, NULL, NULL);
            if (0 < rst_n) {
                for(i=0; i < fd_counter; i++) {
                    if (FD_ISSET(i, &rset)) {
                        n = read(i, line_buf, MAXLINE);
                        if (0 == n) {
                            fprintf(stderr, "Child: Pipeline peer side Close\n"); break;
                        }
                        n = write(fd1[1], line_buf, n);
                    }
                }
            } else if (0 == rst_n) {
                printf("Child: Interrupted due to timeout!\n");
            } else {
                fprintf(stderr, "Child: Interrupted due to a signal!\n"); break;
            }
```

```
            w_counter++;
            if (w_cycle+1 <= w_counter) {
                sleep(1);
                close(fd2[0]);
                printf("Child: Close fd2[0] w_counter: %d\n", w_counter);
            }
        }
    }
    return 0;
}
/opt/drill_ground/aupe/15-7.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW
```

```
tongue aupe # gcc -g 15-7.c -o 15-7
tongue aupe # ./15-7
write: something...
read: something...
write: something...
read: something...
close fd2[1] w_counter: 3
Interrupted due to a signal!
Child: Pipeline peer side Close
Child: Close fd2[0] w_counter: 4
Child: Interrupted due to a signal!
```

select中，管道写关闭的反应

select中，管道读关闭的反应

```c
#include <stdio.h>
#include <unistd.h>     // For pipe;
#include <string.h>     // For bzero;
#include <poll.h>

#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    int fd0[2], fd1[2], fd2[2], flag = 1, rst_n, fd_counter, w_counter = 1, w_cycle = 2, i, n;
    char line_buf[MAXLINE];
    bzero(line_buf, MAXLINE);
    struct pollfd pfdarr[1024];
    pid_t pid;
    if (2 <= argc) {
        w_cycle = atoi(argv[1]);
    }
    // poll中，单个读、写文件描述符的关闭不具代表性，故多添加一个；
    pipe(fd0);
    pipe(fd1);
    pipe(fd2);
    if (0 < (pid = fork())) {
        close(fd1[1]);
        close(fd2[0]);
        pfdarr[0].fd = fd1[0];
        pfdarr[0].events = POLLIN;
        pfdarr[1].fd = fd0[1];
        pfdarr[1].events = POLLOUT;
        pfdarr[2].fd = fd2[1];
        pfdarr[2].events = POLLOUT;
        fd_counter = 2 + 1;
        sleep(1);
        while(flag) {
            rst_n = poll(pfdarr, fd_counter, -1);
            printf("Parent: rst_n: %d\n", rst_n);
            if (0 < rst_n) {
                for(i=0; i < fd_counter; i++) {
                    printf("Parent: i: %d, fd: %d, revents: %d\n", i, pfdarr[i].fd, pfdarr[i].revents);
                    switch(pfdarr[i].revents) {
                        case POLLIN:
                        case POLLRDNORM:
                        case POLLRDBAND:
                        case POLLPRI: {
                                bzero(line_buf, MAXLINE);
                                n = read(pfdarr[i].fd, line_buf, MAXLINE);
                                printf("Parent: read: %s\n", line_buf);
```

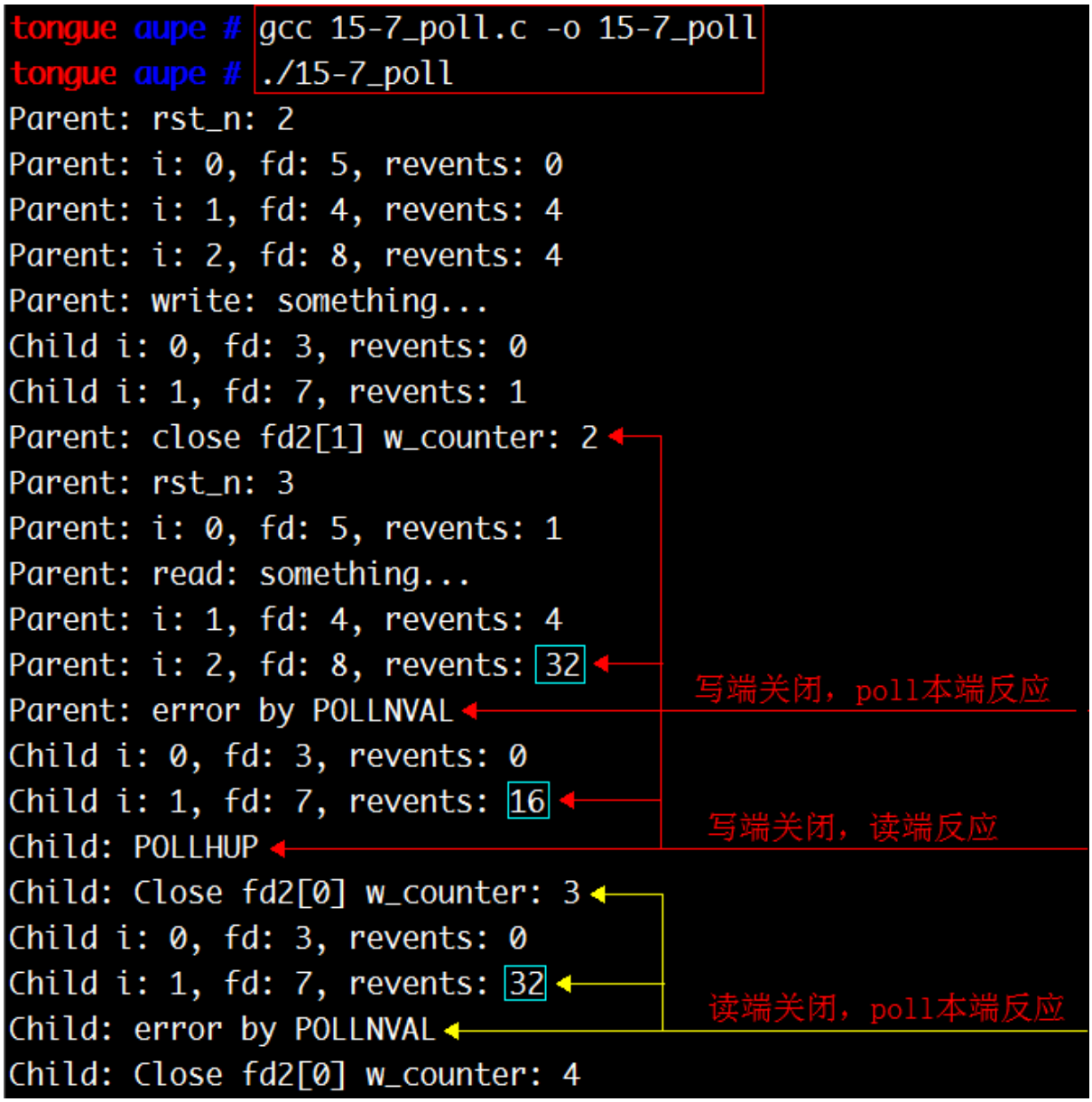
```
                    }
                    break;
                case POLLOUT:
                case POLLWRNORM:
                case POLLWRBAND: {
                        if (fd2[1] == pfdarr[i].fd) {
                            n = write(pfdarr[i].fd, "something...", sizeof("something..."));
                            printf("Parent: write: something...\n");
                            // 等待切换到子进程读;
                            sleep(1);
                        }
                    }
                    break;
                case POLLERR: {
                        printf("Parent: error by POLLERR\n");
                        flag = 0;
                    }
                    break;
                case POLLHUP: {
                        printf("Parent: POLLHUP\n");
                    }
                    break;
                case POLLNVAL: {
                        printf("Parent: error by POLLNVAL\n");
                        flag = 0;
```

```
                        }
                        break;
                    default:
                        continue;
                }
            }
        } else if (0 == rst_n) {
            printf("Parent: Interrupted due to timeout!\n"); break;
        } else {
            fprintf(stderr, "Parent: Interrupted due to a signal!\n"); break;
        }
        w_counter++;
        if (w_cycle <= w_counter && flag) {
            close(fd2[1]);
            printf("Parent: close fd2[1] w_counter: %d\n", w_counter);
        }
    }
    waitpid(pid, NULL, 0);
```

```
    } else {
        close(fd1[0]);
        close(fd2[1]);
        pfdarr[0].fd = fd0[0];
        pfdarr[0].events = POLLIN;
        pfdarr[1].fd = fd2[0];
        pfdarr[1].events = POLLIN;
        fd_counter = 1 + 1;
        while(flag) {
            rst_n = poll(pfdarr, fd_counter, -1);
            if (0 < rst_n) {
                for(i=0; i < fd_counter; i++) {
                    printf("Child i: %d, fd: %d, revents: %d\n", i, pfdarr[i].fd, pfdarr[i].revents);
                    switch(pfdarr[i].revents) {
                        case POLLIN:
                        case POLLRDNORM:
                        case POLLRDBAND:
                        case POLLPRI: {
                                bzero(line_buf, MAXLINE);
                                n = read(pfdarr[i].fd, line_buf, MAXLINE);
                                if (0 == n) {
                                    fprintf(stderr, "Child: Pipeline peer side Close\n"); break;
                                }
                                n = write(fd1[1], line_buf, n);
```

```
                // 等待切换到父进程读;
                sleep(1);
            }
            break;
        case POLLOUT:
        case POLLWRNORM:
        case POLLWRBAND: {
                printf("Child: can wrint\n");
            }
            break;
        case POLLERR: {
                printf("Child: error by POLLERR\n");
                flag = 0;
            }
            break;
        case POLLHUP: {
                printf("Child: POLLHUP\n");
            }
            break;
        case POLLNVAL: {
                printf("Child: error by POLLNVAL\n");
                flag = 0;
            }
            break;
```

```
                        default:
                            continue;
                    }
                }
            } else if (0 == rst_n) {
                printf("Child: Interrupted due to timeout!\n");
            } else {
                fprintf(stderr, "Child: Interrupted due to a signal!\n"); break;
            }
            w_counter++;
            if (w_cycle+1 <= w_counter) {
                sleep(1);
                close(fd2[0]);
                printf("Child: Close fd2[0] w_counter: %d\n", w_counter);
            }
        }
    }
    return 0;
}
/opt/drill_ground/aupe/15-7_poll.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=1
```


tongue aupe # gcc 15-7_poll.c -o 15-7_poll
tongue aupe # ./15-7_poll
Parent: rst_n: 2
Parent: i: 0, fd: 5, revents: 0
Parent: i: 1, fd: 4, revents: 4
Parent: i: 2, fd: 8, revents: 4
Parent: write: something...
Child i: 0, fd: 3, revents: 0
Child i: 1, fd: 7, revents: 1
Parent: close fd2[1] w_counter: 2
Parent: rst_n: 3
Parent: i: 0, fd: 5, revents: 1
Parent: read: something...
Parent: i: 1, fd: 4, revents: 4
Parent: i: 2, fd: 8, revents: 32
Parent: error by POLLNVAL
写端关闭，poll本端反应
Child i: 0, fd: 3, revents: 0
Child i: 1, fd: 7, revents: 16
Child: POLLHUP
写端关闭，读端反应
Child: Close fd2[0] w_counter: 3
Child i: 0, fd: 3, revents: 0
Child i: 1, fd: 7, revents: 32
Child: error by POLLNVAL
读端关闭，poll本端反应
Child: Close fd2[0] w_counter: 4

15-8、结果正常；

```c
#include <stdio.h>
#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    char line_buf[MAXLINE];
    FILE *fpin;
    if (2 > argc) {
        printf("You need cmdstring!\n");
        return 0;
    }
    fpin = popen(argv[1], "r");
    while(fgets(line_buf, MAXLINE, fpin)) {
        fputs(line_buf, stderr);
    }
    pclose(fpin);
    return 0;
}
```

```
tongue aupe # gcc 15-8.c -o 15-8
tongue aupe # ./15-8 "head -1 /etc/fstab"
# /etc/fstab: static file system information.
```

15-9、被执行的cmdstring命令，因直接终止，导致其仍在缓存区中的数据因没被刷新，而不被输出；

```c
#include <stdio.h>

int main(int argc, const char *argv[]) {
    printf("%d\n", getpid());
    if (2 <= argc) {
        _Exit(-1);
    }
    return 0;
}
```

/opt/drill_ground/aupe/getpid.c [FORMAT=

```c
#include <stdio.h>

#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    char *rst_c;
    char line_buf[MAXLINE];
    FILE *fpin;
    printf("pid: %d\n", getpid());
    fpin = popen(argv[1], "r");
    while((rst_c = fgets(line_buf, MAXLINE, fpin))) {
        printf("fgets returned %s", rst_c);
        fputs(line_buf, stdout);
    }
    printf("fgets returned %s\n", rst_c);
    pclose(fpin);
    return 0;
}
```

/opt/drill_ground/aupe/15-9.c [FORMAT=unix:utf-8] [TY

```
tongue aupe # gcc 15-9.c -o 15-9
tongue aupe # ./15-9 "./getpid x"
pid: 15003
fgets returned (null)
tongue aupe # ./15-9 "./getpid"
pid: 15005
fgets returned 15006
15006
fgets returned (null)
```

15-10、注意以读写方式打开时，没有用到O_NONBLOCK标记，该题是对415页FIFO章节内容的回顾或验证；

```c
#include <stdio.h>
#include <string.h>
#include <fcntl.h>      // For O_NONBLOCK...;
#include <sys/stat.h>   // For mkfifo;

#define MAXLINE 1024

int main(int argc, const char *argv[]) {
    int fifd, flag = 1, cycle = 3, cyc_counter = 0, n; // fifd for fifofd;
    pid_t pid;
    char *ffpath = "./a.fifo";
    char line_buf[MAXLINE];
    if (2 > argc) {
        printf("Example format: ./15-10 <RD|WR|RDWR> [1-9..]\n"); return 0;
    } else if (3 <= argc) {
        cycle = atoi(argv[2]);
    }
```

```
    mkfifo(ffpath, 0600);
    if (!strcmp("RD", argv[1])) {
        fffd = open(ffpath, O_RDONLY | O_NONBLOCK);
    } else if (!strcmp("WR", argv[1])) {
        fffd = open(ffpath, O_WRONLY | O_NONBLOCK);
    } else if (!strcmp("RDWR", argv[1])) {
        fffd = open(ffpath, O_RDWR);
    } else {
        printf("Example format: ./15-10 <RD|WR|RDWR> [1-9..]\n"); return 0;
    }
    printf("open return of fffd: %d\n", fffd);
    while(flag) {
        bzero(line_buf, MAXLINE);
        n = write(fffd, "Parent send", strlen("Parent send"));
        printf("write return: %d\n", n);
        n = read(fffd, line_buf, MAXLINE);
        printf("read return: %d\n", n);
        printf("Parend read: %s\n", line_buf);
        cyc_counter++;
        if (cycle <= cyc_counter) {
            flag = 0;
        }
    }
    return 0;
}
/opt/drill_ground/aupe/15-10.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=
```

```
tongue aupe # gcc 15-10.c -o 15-10
tongue aupe # ./15-10 RD 1
open return of fffd: 3
write return: -1
read return: 0
Parend read:
tongue aupe # ./15-10 WR 1
open return of fffd: -1
write return: -1
read return: -1
Parend read:
tongue aupe # ./15-10 RDWR 1
open return of fffd: 3
write return: 11
read return: 11
Parend read: Parent send
```

15-11、须知道队列id、内容字段大小即可；

```c
#include <stdio.h>
#include <sys/ipc.h>     // For ftok;
#include <sys/msg.h>     // For msgget;
#include <string.h>      // For memcpy;
#include <errno.h>       // For errno;

#define MSGCNTLEN 128    // message content length;

struct msgsrt {
    long mtype;
    char mtext[MSGCNTLEN];
};

int main(int argc, const char *argv[]) {
    char *pjt_path = "/dev/zero"; // Project path;
    int queue_id, flag = 1, i = 0;
    pid_t pid;
    struct msgsrt msg;
    key_t queue_key = ftok(pjt_path, 1);
    queue_id = msgget(queue_key, IPC_CREAT);
    msgctl(queue_id, IPC_RMID, NULL);
    queue_id = msgget(queue_key, IPC_CREAT);
    printf("queue_id: %d\n", queue_id);
    printf("errno: %d %s\n", errno, strerror(errno));
    if (0 < (pid = fork())) {
        msg.mtype = 1;
        char cnt[] = "Counter ";
        int cnt_len = sizeof(cnt);
        while (i < 9) {
            i++;
            memcpy(msg.mtext, cnt, cnt_len);
            msg.mtext[cnt_len-1] = i+48;
            msg.mtext[cnt_len] = '\0';
            msgsnd(queue_id, &msg, MSGCNTLEN, 0);
        }
    } else {
        while (flag) {
            sleep(1);
            if (-1 == msgrcv(queue_id, &msg, MSGCNTLEN, 1, MSG_NOERROR | IPC_NOWAIT)) {
                break;
            }
            printf("read msg.mtext: %s\n", msg.mtext);
        }
    }
    return 0;
}
```

/opt/drill_ground/aupe/15-11.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=046/46(100%)

```c
#include <stdio.h>
#include <sys/ipc.h>    // For ftok;
#include <sys/msg.h>    // For msgget;

#define MSGCNTLEN 128   // message content length;

struct msgsrt {
    long mtype;
    char mtext[MSGCNTLEN];
};

int main(int argc, const char *argv[]) {
    char *pjt_path = "/dev/zero"; // Project path;
    int queue_id, flag = 1, i = 0;
    struct msgsrt msg;
    key_t queue_key = ftok(pjt_path, 1);
    queue_id = msgget(queue_key, 0);
    sleep(3);
    while (i < 5) {
        i++;
        msgrcv(queue_id, &msg, MSGCNTLEN, 1, MSG_NOERROR);
        printf("other read msg.mtext: %s\n", msg.mtext);
    }
    return 0;
}
</drill_ground/aupe/15-11_other_reader.c [FORMAT=unix:utf-
```

```
tongue aupe # gcc 15-11.c -o 15-11
tongue aupe # gcc 15-11_other_reader.c -o 15-11_other_reader
tongue aupe # ./15-11; ./15-11_other_reader
queue_id: 983040
errno: 0 Success
read msg.mtext: Counter 1
read msg.mtext: Counter 2
other read msg.mtext: Counter 3
other read msg.mtext: Counter 4
other read msg.mtext: Counter 5
other read msg.mtext: Counter 6
other read msg.mtext: Counter 7
tongue aupe # read msg.mtext: Counter 8
read msg.mtext: Counter 9
```

15-12、目的使读者理解队列标识符产生的规律；

```c
#include <stdio.h>
#include <sys/ipc.h>     // For ftok;
#include <sys/msg.h>     // For msgget;
#include <string.h>      // For memcpy;

#define MSGCNTLEN 128    // message content length;

struct msgsrt {
    long mtype;
    char mtext[MSGCNTLEN];
};

int main(int argc, const char *argv[]) {
    char *pjt_path = "/dev/zero"; // Project path;
    int queue_id, i = 1;
    pid_t pid;
    struct msgsrt msg;
    key_t queue_key = ftok(pjt_path, 1);
    char cnt[] = "content", cmdstr[4096];
    int cnt_len = sizeof(cnt), cycle = 10;
    if (2 <= argc) {
        cycle = atoi(argv[1]);
    }
    msg.mtype = 1;
    memcpy(msg.mtext, cnt, cnt_len);
    system("ipcrm -a");
    while (i <= cycle) {
        i++;
        if (i <= 5) {
            queue_id = msgget(queue_key, IPC_CREAT);
            msgctl(queue_id, IPC_RMID, NULL);
        } else {
            queue_id = msgget(IPC_PRIVATE, IPC_CREAT);
            msgsnd(queue_id, &msg, MSGCNTLEN, 0);
        }
        sprintf(cmdstr, "echo -n \"queue_id: %X\t\"; echo \"obase=2; ibase=16; %X\" | bc", queue
_id, queue_id);
        system(cmdstr);
        // sleep(1);
    }
    return 0;
}
```

/opt/drill_ground/aupe/15-12.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=041/41(100%)]

每个内核中的IPC结构（消息队列、信号量或共享存储段）都用一个非负整数的标识符(identifier）加以引用。例如，为了对一个消息队列发送或取消息，只需要知道其队列标识符。与文件描述符不同，IPC标识符不是小的整数。当一个IPC结构被创建，以后又被删除时，与这种结构相关的标识符连续加1，直至达到一个整型数的最大正值，然后又回转到0。

下图中IPC结构标识符范围是从0~7FFF，记录当前队列长度是一个12位长的正整数；
图中黄色框为当前IPC结构标志值，绿色框为当前IPC队列长度；
IPC结构标志值从0开始，下图之所以从1开始，是应为在此之前产生过一个IPC结构标志，并且已被删除；

```
tongue aupe # ./15-12 10
queue_id: 8000   1000000000000000
queue_id: 10000  1000000000000000
queue_id: 18000  1100000000000000
queue_id: 20000  1000000000000000
queue_id: 28000  1010000000000000
queue_id: 30001  1100000000000001
queue_id: 38002  1110000000000010
queue_id: 40003  1000000000000011
queue_id: 48004  1001000000000100
queue_id: 50005  1010000000000101
```

奇怪：

```
queue_id: 3EC60FAC      11111011000110000011111010110O
queue_id: FFFFFFFF      11111111111111111111111111111111
tongue aupe # ./15-12 4018          有点莫名其妙哦
```

```
tongue aupe # ipcs -l

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 32768
max total shared memory (kbytes) = 8388608
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767

------ Messages Limits --------
max queues system wide = 4013
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

这里解释了上面的4018值，
4018=4013+5

```
tongue aupe # ipcs -u

------ Shared Memory Status --------
segments allocated 0
pages allocated 0
pages resident  0
pages swapped   0
Swap performance: 0 attempts     0 successes

------ Semaphore Status --------
used arrays = 0
allocated semaphores = 0

------ Messages Status --------
allocated queues = 4013
used headers = 4013
used space = 513664 bytes
```

15-13、共享存储区是一块空白、自由的内存区域，该区域的使用全由使用者来规划，若要满足该题所隐喻的现实中某种实际需求。那么我们可以做如下使用示例：

假设我们要共享存储，最大100个某种单一结构的对象，并给这些对象建立列表，以便检索。每个对象结构固定大小128字节，列表中的元素每个24字节(8prev,8next,8obj_ptr)。

首先开辟一个15200=(128+24)*100字节大小的共享存储区，共享区起始地址至其后2400字节为列表存放专用区，起始地址2400字节之后空间为具体对象存放区。列表中的所有指针地址，均为以共享区起始地址为参考的偏移量。

15-14、如图，产生该图的程序由15-16提供；

```
parent current | i:0 | update() return: 0 | share value:  1|
child  current | i:1 | update() return: 1 | share value:  2|
parent current | i:2 | update() return: 2 | share value:  3|
child  current | i:3 | update() return: 3 | share value:  4|
parent current | i:4 | update() return: 4 | share value:  5|
child  current | i:5 | update() return: 5 | share value:  6|
parent current | i:6 | update() return: 6 | share value:  7|
child  current | i:7 | update() return: 7 | share value:  8|
parent current | i:8 | update() return: 8 | share value:  9|
child  current | i:9 | update() return: 9 | share value: 10|
```

15-15、与15-16合并；

15-16、包含了15-15所问的内容，打印了15-14所问内容。信号量值得注意的地方；

说　明	典　型　值			
	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9
任一信号量的最大值	32 767	32 767	32 767	32 767
任一信号量的最大的终止时调整值	16 384	32 767	16 384	16 384
系统中信号量集的最大数	10	128	87 381	10
系统中信号量的最大数	60	32 000	87 381	60
每个信号量集中的最大信号量数	60	250	87 381	25
系统中undo结构的最大数	30	32 000	87 381	30
每个undo结构中的最大undo项数	10	32	10	10
每个semop调用中的最大操作项数	100	32	100	10

对集合中每个成员的操作由相应的sem_op值规定。此值可以是负值、0或正值。（下面的讨论将提到信号量的undo标志。此标志对应于相应sem_flg成员的SEM_UNDO位。）

(1) 最易于处理的情况是sem_op为正。这对应于进程释放占用的资源数。sem_op值加到信号量的值上。如果指定了undo标志，则也从该进程的此信号量调整值中减去sem_op。

(2) 若sem_op为负，则表示要获取由该信号量控制的资源。

如若该信号量的值大于或等于sem_op的绝对值（具有所需的资源），则从信号量值中减去sem_op的绝对值。这保证信号量的结果值大于或等于0。如果指定了undo标志，则sem_op的绝对值也加到该进程的此信号量调整值上。

```c
#include <stdio.h>
#include <sys/sem.h>    // For semget...;
#include <sys/shm.h>    // For shmget...;
#include <string.h>     // For strerror;
#include <errno.h>

#define SIZE sizeof(long)

union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                (Linux-specific) */
};

static int update(long *ptr) {
    return((*ptr)++);
}

int main(int argc, const char *argv[]) {
    int shm_id, sem_id, i, counter, nloops = 10;
    char flag = 'p';
    pid_t pid;
    union semun seun;
    struct sembuf sebf[1];
    void *shm_area;
    if (2 <= argc) {
        nloops = atoi(argv[1]);
    }
    sebf[0].sem_num = 0;
    sebf[0].sem_flg = SEM_UNDO;
```

```
    sem_id = semget(IPC_PRIVATE, 1, IPC_CREAT);
    if ('p' == flag) {
        seun.val = 0;   // 父进程先执行;
    } else {
        seun.val = 2;   // 子进程先执行;
    }
    // 给第一个信号量赋初值0, 序数从0开始;
    semctl(sem_id, 0, SETVAL, seun);
    printf("the semid_ds[0].semval is %d\n", semctl(sem_id, 0, GETVAL));
    shm_id = shmget(IPC_PRIVATE, SIZE, IPC_CREAT);
    if (0 > (pid = fork())) {
        fprintf(stderr, "fork error"); _Exit(-1);
    } else if (0 < pid) {
        shm_area = shmat(shm_id, 0, 0);
        for (i=0; i<nloops; i+=2) {
            // SEM_UNDO仅执行一次, 不然每次都会给 信号量调整值中累加;
            if (2==i) {
                sebf[0].sem_flg = 0;
            }
            // 注释中包含@符的, 表示可与其上语句替换,
            // 实现资源量分配、限制作用;
            sebf[0].sem_op = 0;
            // @ sebf[0].sem_op = -1;
            semop(sem_id, sebf, 1);
            if (i != (counter = update((long *)shm_area))) {
                fprintf(stderr, "%s\n", strerror(errno));
                fprintf(stderr, "parent: expected %d, got %d", i, counter); _Exit(-1);
            }
            printf("%c[7;32mparent current | i:%d | update() return: %d | share value: "\
                    "%2ld|%c[0m\n", 27, i, counter, *(long *)shm_area, 27);
            sebf[0].sem_op = 2;
            // @ sebf[0].sem_op = 1;
            semop(sem_id, sebf, 1);
        }
        shmdt(shm_area);
    } else {
        shm_area = shmat(shm_id, 0, 0);
        for (i=1; i<nloops+1; i+=2) {
            // SEM_UNDO仅执行一次, 不然每次都会给 信号量调整值中累加;
            if (3==i) {
                sebf[0].sem_flg = 0;
            }
            sebf[0].sem_op = -1;
            semop(sem_id, sebf, 1);
            if (i != (counter = update((long *)shm_area))) {
                fprintf(stderr, "%s\n", strerror(errno));
                fprintf(stderr, "child: expected %d, got %d", i, counter); _Exit(-1);
            }
            printf("%c[7;33mchild  current | i:%d | update() return: %d | share value: "\
                    "%2ld|%c[0m\n", 27, i, counter, *(long *)shm_area, 27);
            sebf[0].sem_op = -1;
            // @ sebf[0].sem_op = 1;
            semop(sem_id, sebf, 1);
        }
        shmdt(shm_area);
    }
    waitpid(pid, NULL, 0);
    return 0;
}
/opt/drill_ground/aupe/15-16.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=091/91(100%)]
```

```
tongue aupe # gcc -g 15-16.c -o 15-16
tongue aupe # ./15-16
the semid_ds[0].semval is 0
parent current | i:0 | update() return: 0 | share value:  1|
child  current | i:1 | update() return: 1 | share value:  2|
parent current | i:2 | update() return: 2 | share value:  3|
child  current | i:3 | update() return: 3 | share value:  4|
parent current | i:4 | update() return: 4 | share value:  5|
child  current | i:5 | update() return: 5 | share value:  6|
parent current | i:6 | update() return: 6 | share value:  7|
child  current | i:7 | update() return: 7 | share value:  8|
parent current | i:8 | update() return: 8 | share value:  9|
child  current | i:9 | update() return: 9 | share value: 10|
```

15-17、用比记录锁更直接的pwrite、pread来实现；

```c
#include <stdio.h>
#include <sys/sem.h>    // For semget...;
#include <sys/shm.h>    // For shmget...;
#include <fcntl.h>

#define SIZE sizeof(long)

union semun {
    int              val;    /* Value for SETVAL */
    struct semid_ds *buf;    /* Buffer for IPC_STAT, IPC_SET */
    unsigned short  *array;  /* Array for GETALL, SETALL */
    struct seminfo  *__buf;  /* Buffer for IPC_INFO
                                (Linux-specific) */
};

static int update(long *ptr) {
    return((*ptr)++);
}

int main(int argc, const char *argv[]) {
    int fd, shm_id, i, counter, nloops = 10;
    char path[BUFSIZ] = "./tmpfile", flag = 'p';
    pid_t pid;
    void *shm_area;
    fd = open( path, O_CREAT | O_RDWR, S_IRUSR | S_IWUSR);
    unlink( path);
    pwrite( fd, &flag, sizeof(flag), 0);
    shm_id = shmget(IPC_PRIVATE, SIZE, IPC_CREAT);
    if (2 <= argc) {
        nloops = atoi(argv[1]);
    }
```

```
    if (0 > (pid = fork())) {
        fprintf(stderr, "fork error"); _Exit(-1);
    } else if (0 < pid) {
        shm_area = shmat(shm_id, 0, 0);
        for (i=0; i<nloops; i+=2) {
            // 下面while代码块即类 WAIT_CHILD();
            while( 'p' != flag) {
                if (-1 == pread( fd, &flag, sizeof(flag), 0)) {
                    printf("Error by child read!\n");
                }
            }
            if (i != (counter = update((long *)shm_area))) {
                fprintf(stderr, "parent: expected %d, got %d", i, counter); _Exit(-1);
            }
            printf("%c[7;32mparent current | i:%d | update() return: %d | share value: "\
                    "%2ld|%c[0m\n", 27, i, counter, *(long *)shm_area, 27);
            // 下面代码块即类 TELL_CHILD();
            flag = 'c';
            if (-1 == pwrite( fd, &flag, sizeof(flag), 0)) {
                printf("Error by child write!\n");
            }
        }
        shmdt(shm_area);
    } else {
        shm_area = shmat(shm_id, 0, 0);
        for (i=1; i<nloops+1; i+=2) {
            // 下面while代码块即类 WAIT_PARENT();
            while( 'c' != flag) {
                if (-1 == pread( fd, &flag, sizeof(flag), 0)) {
                    printf("Error by child read!\n");
                }
            }
            if (i != (counter = update((long *)shm_area))) {
                fprintf(stderr, "child: expected %d, got %d", i, counter); _Exit(-1);
            }
            printf("%c[7;33mchild  current | i:%d | update() return: %d | share value: "\
                    "%2ld|%c[0m\n", 27, i, counter, *(long *)shm_area, 27);
            // 下面代码块即类 TELL_PARENT();
            flag = 'p';
            if (-1 == pwrite( fd, &flag, sizeof(flag), 0)) {
                printf("Error by child write!\n");
            }
        }
        shmdt(shm_area);
    }
    waitpid(pid, NULL, 0);
    return 0;
}
/opt/drill_ground/aupe/15-17.c [FORMAT=unix:utf-8] [TYPE=C] [COL=001] [ROW=079/79(100%)]
```

```
tongue aupe # gcc 15-17.c -o 15-17
tongue aupe # ./15-17
parent current | i:0 | update() return: 0 | share value:  1|
child  current | i:1 | update() return: 1 | share value:  2|
parent current | i:2 | update() return: 2 | share value:  3|
child  current | i:3 | update() return: 3 | share value:  4|
parent current | i:4 | update() return: 4 | share value:  5|
child  current | i:5 | update() return: 5 | share value:  6|
parent current | i:6 | update() return: 6 | share value:  7|
child  current | i:7 | update() return: 7 | share value:  8|
parent current | i:8 | update() return: 8 | share value:  9|
child  current | i:9 | update() return: 9 | share value: 10|
```

两种同步机制的简单比较：

```
real    0m39.017s
user    0m0.610s
sys     0m2.600s
tongue aupe # time ./15-16 100000
```

```
real    0m35.212s
user    0m4.940s
sys     0m30.730s
tongue aupe # time ./15-17 100000
```