进程

wait 回收子进程

作用:

1. 阻塞等待子进程退出（终止）
2. 回收子进程残留在内核的pcb
3. 获取子进程的退出状态

```c
int main(int argc, char *argv[])
{
    int status = 0;
    pid_t wpid = 0;

    pid_t pid = fork();
    if (pid == -1)
        sys_err("fork err");
    else if (pid == 0) {
        printf("I'm child pid = %d\n", getpid());
#if 1
        execl("./abnor", "abnor", NULL);
        perror("execl err");
        exit(1);
#endif
        sleep(1);
        exit(73);
    } else {
        wpid = wait(&status);  // 保存子进程退出的状态.
        if (wpid == -1)
            sys_err("wait err");
        if (WIFEXITED(status)) {            // 宏函数为真,说明子进程正常终止.
            // 获取退出码
            printf("I'm parent, pid = %d child, exit code = %d\n", wpid,
WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {  // 宏函数为真. 说明子进程被信号终止.
            // 获取信号编号
            printf("I'm parent, pid = %d child, killed by %d signal\n", wpid,
WTERMSIG(status));
        }
    }
```


waitpid 函数

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

参数:

pid：通过pid指定回收某一个子进程

>0:回收，某一个子进程

-1：回收任意子进程

0：回收与父进程同一进程组的子进程

wstatus:(传出)回收子进程状态

options:WNOHANG--指定回收方式为“非阻塞”

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
参数：
    pid：
        > 0: 通过pid指定 回收某一个子进程。
        -1: 回收任意子进程。
        0: 回收 与父进程属于同一个进程组的 子进程。
                -- 子进程创建成功后，默认，会自动加入父进程进程组。

    wstatus：（传出）回收子进程状态。
    options：WNOHANG -- 指定回收方式为 “非阻塞”。
             0 -- 指定回收方式为 “阻塞”。等同于 wait()
返回值：
    > 0: 表成功回收的进程pid
    0: 函数调用时参3指定了 WNOHANG，子进程没有结束。
    -1：失败。 errno
```

回收N个子进程

阻塞

1.

```
文件(F)  编辑(E)  查看(V)  搜索(S)  终端(T)  帮助(H)
 8
 9 void sys_err(const char *str)
10 {
11     perror(str);
12     exit(1);
13 }
14
15 int main(int argc, char *argv[])
16 {
17     int i = 0;
18     pid_t pid, wpid;
19     for(i = 0; i < 5; i++) {
20         pid = fork();                    一手v：kunlun991
21         if (pid == 0)
22             break;
23     }
24     if (5 == i) {        // 父进程
25         while ((wpid = wait(NULL))!=-1) {  // 阻塞等待子进程结束,回收
26             printf("wait child %d\n", wpid);
27         }
28
29     } else {
30         sleep(i);
31         printf("%dth child, pid = %d\n", i+1, getpid());
```

2.

```c
int main(int argc, char *argv[])
{
    int i = 0;
    pid_t pid, wpid;
    for(i = 0; i < 5; i++) {
        pid = fork();
        if (pid == 0)
            break;
    }
    if (5 == i) {       // 父进程
        /*
        while ((wpid = wait(NULL))!=-1) {  // 阻塞等待子进程结束,回收
            printf("wait child %d\n", wpid);
        } */

        while ((wpid = waitpid(-1, NULL, 0))!=-1) {  // 使用 waitpid 阻塞等待子进程结束,回收
            printf("wait child %d\n", wpid);
        }


    } else {
        sleep(i);
        printf("%dth child, pid = %d\n", i+1, getpid());
    }
```

waitpid_while.c [+] 29,3-9 82%
"waitpid_while.c" 41L, 724C

非阻塞方式

```c
        /*
        while ((wpid = wait(NULL))!=-1) {  // 阻塞等待子进程结束,回收
            printf("wait child %d\n", wpid);
        } */
        /*
        while ((wpid = waitpid(-1, NULL, 0))!=-1) {  // 使用 waitpid 阻塞等待子进程结束,回收
            printf("wait child %d\n", wpid);
        } */

        while ((wpid = waitpid(-1, NULL, WNOHANG))!=-1) {  // 使用 waitpid 非 阻塞回收子进程
            if (wpid > 0) {
                printf("wait child %d\n", wpid);        // 正常回收一个子进程
            } else if (wpid == 0) {
                sleep(1);
                continue;
            }
        }
        printf("catch All child  finish\");

    } else {
        sleep(i);
        printf("%dth child, pid = %d\n", i+1, getpid());
    }

```

waitpid_while.c [+] 42,35-41 92%
-- 插入 --

总结

一次wait.waitpid调用，只能回收一个子进程

回收N个只能放在循环中

进程间通信IPC

- 进程间通信原理：借助多个进程共同使用同一个内核，借助内核传递数据

进程间通信的方法

1. 管道：最简单
2. 信号：开销小
3. mmap映射：速度快，非血缘关系间
4. socket（本地套接字）：稳定性好

管道pipe

- 实现原理：Linux内核使用环形队列机制，借助缓冲区（4K）实现。
- 特质：
    1. 本质：伪文件（实为内核缓冲区）
    2. 用于进程间通信，一个读端一个写端
    3. 规定，数据从写端流入，读端流出
- 局限性：

1. 自己写，不能自己读
2. 管道中的数据，读走没，不能反复读取
3. 半双工通信
4. 必须应用于血缘关系进程间

使用的函数

```
函数调用成功，自动创建匿名管道，返回两个文件描述符，无需open，但需要手动close
 pipe(int pipifd[2])
0]:管道读端  r
1]:管道写端  w
l值：
 成功：0
 失败：-1、0
```

- 父子进程 管道通信ipc实例：

管道读写行为

读管道：

- 管道有数据，read返回实际读到的字节数
- 无数据 1）无写端，read返回0 2）有写端，阻塞等待

写管道：

- 无读端，异常终止（SIGPIPE信号）
- 有读端 1）管道已满， 阻塞等待

    2）管道未满，返回实际写出字节数

父子进程ls -wc-l

```c
int main(int argc, char *argv[])
{
    pid_t pid;
    int fd[2];

    // 先创建pipe
    pipe(fd);
    pid = fork();           // ls | wc -l

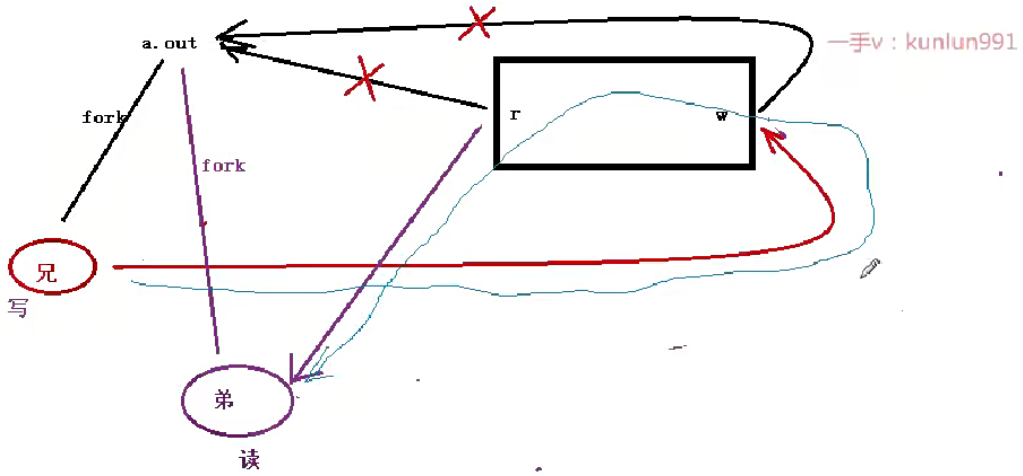
    if (pid == 0) {  // 子进程 实现 wc -l
        close(fd[1]);       // 子进程读管道,关闭写端.
        dup2(fd[0], STDIN_FILENO);      // 让 wc 从管道的读端,读数据.
        execlp("wc", "wc", "-l", NULL);

    } else if (pid > 0) {
        close(fd[0]);       // 父进程写管道,关闭读端.
        dup2(fd[1], STDOUT_FILENO);// 将 写出到 屏幕的ls 结果,写入到 管道写端.
        execlp("ls", "ls", NULL);
    }

```

兄弟进程ls-wc-l

阻塞原因：



正确代码：

```
        pid_t pid;

        pipe(fd);

        for (i = 0; i < 2; i++)
                if ((pid = fork()) == 0) {
                        break;
                }
        if (i == 0) {           // 兄          ls
                close(fd[0]);
                dup2(fd[1], STDOUT_FILENO);
                execlp("ls", "ls", NULL);
        } else if (i == 1) {            // 弟   wc -l
                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                execlp("wc", "wc","-l",NULL);

        } else {                // 父
                close(fd[0]);
                close(fd[1]);
                for (i = 0; i<2; i++)
                        wait(NULL);
        }

        return 0;
}
```

管道缓冲区

- 命令查询

```
itcast@itcast:~/bj_40/IPC_test/pipe/test$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 31641
max locked memory       (kbytes, -l) 16384
max memory size         (kbytes, -m) unlimited
open files                      (-n) 65535
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 31641
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
itcast@itcast:~/bj_40/IPC_test/pipe/test$
```

- 函数查询


```
long fpathconf(int fd, int name);
参1 ：传 fd[0]/fd[1] 都可以!
参2：传 _PC_PIPE_BUF 宏。
```

管道优劣

- 优点：简单。
- 缺点：
  1. 只能单向通信，实现双向通信，需要两个管道
  2. 只能应用于父子，兄弟（有公共祖先）。无血缘关系进程，后来用fifo替代

命名管道fifo

- 命令创建：mkfifo 管道名
- 函数创建：mkfifo
- ```
  mkfifo (const char* pathname,mode_t mode)
  ```
- 可用于无血缘关系的进程间通信
- 数据一次性读取
- 读端：以O_RDONLY打开fifo管道
- 写端：以O_WRONLY/O_RDWR打开同一个fifo管道

mmap

文件进程间通信

- 有无血缘关系的进程，都可以使用同一个文件来实现进程通信

建立、释放映射区

建立:
  ```

- mmap 借助文件映射，创建共享内存映射区。

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
参数：
    addr：指定映射区的首地址。通常传NULL，表示让系统自动分配。
    length：共享内存映射区的大小。（<=文件的实际大小。）
    prot：共享内存映射区的读写属性。PROT_READ、PROT_WRITE、PROT_READ|PROT_WRITE
    flags：标注共享内存映射区的共享属性：
            MAP_SHARED：  对共享内存所做的修改，会反应到物理磁盘文件上。 IPC专用！
            MAP_PRIVATE：对共享内存所做的修改，不会反应到物理磁盘文件上。
    fd: 用来创建共享内存映射区的那个文件的 文件描述符。
    offset：默认0，表示映射文件全部！偏移位置。必须是4k整数倍。
返回值：
    成功：映射区的首地址。
    失败：MAP_FAILED (void *(-1)), errno
```

释放:

- munmap 释放共享内存映射

```
int munmap(void *addr, size_t length);
参1：mmap() 函数的返回值。
参2：共享内存映射区大小
返回值：
    成功：0
    失败：-1, errno
```

mmap建立映射区

```
int main(int argc, char *argv[])
{
    int fd = open("test.mmap", O_RDWR|O_CREAT|O_TRUNC, 0644);

    ftruncate(fd, 4);

    char *memp;                                          一手v：kunlun991
    memp = mmap(NULL, 4, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (memp == MAP_FAILED) {
        sys_err("mmap error");
    }

    strcpy(memp, "xxx");        // 写 mmap --- 文件中
    printf("%s\n", memp);       // 读 mmap

    if (munmap(memp, 4) == -1)
        sys_err("munmap error");

    close(fd);
ap.c
```

mmap使用注意事项