孤儿进程 / 僵尸进程

2024年5月22日 9:57

#1. 孤儿进程：子进程存活，父进程终止了.

```
orphen.c                                                    buffers
#include <func.h>

int main(int argc, char* argv[])
{
    pid_t pid = fork();
    switch(pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        sleep(2);
        printf("pid = %d, ppid = %d\n", getpid(), getppid())
        exit(0);
    default:
        // 父进程
        printf("Parent: pid = %d, childPid = %d\n", getpid()
        exit(0);
    }
    return 0;
}

he@he-vm:~/cpp58/2_Linux/Linux09 (master)$ ./orphen
Parent: pid = 12423, childPid = 12424
he@he-vm:~/cpp58/2_Linux/Linux09 (master)$ pid = 12424, ppid = 1
```

孤儿进程会被1号进程收养

→ init进程.

↓

for(;;){

wait(...);

}

僵尸进程：子进程死亡时，有一些信息会保存在内核（pid，退出状态，CPU时间...）

方便父进程以后查看这些信息.

并且会给父进程发送 SIGCHLD 信号

↕

父进程默认会忽略信号.

↕

如何给僵尸进程收尸：wait / waitpid

wait

2024年5月22日 10:11

```
WAIT(2)                          Linux Programmer's Manual

NAME
       wait, waitpid, waitid - wait for process to change state
```

```
SYNOPSIS
       #include <sys/types.h>
       #include <sys/wait.h>

       pid_t wait(int *wstatus);

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

子进程的终止状态信息

```
RETURN VALUE
       wait(): on success, returns the process ID of the terminated child; on error, -1 is returned.
```

成功，终止子进程的pid.

失败：-1，设置errno。

wait

WIFEXITED(wstatus) ⇒ 子进程是否正常终止.

WEXITSTATUS(wstatus) ⇒ 获取正常终止的退出状态码　_exit(status)

WIFSIGNALED(wstatus) ⇒ 子进程是否异常终止

WTERMSIG(wstatus) ⇒ 获取导致异常终止的信号

WCOREDUMP(wstatus) ⇒ 是否能够产生core文件，

#ifdef WCOREDUMP ... #endif.

test_wait.c

```c
#include <func.h>

void print_wstatus(int status) {
    if (WIFEXITED(status)) {
        int exit_code = WEXITSTATUS(status);
        printf("exit_code = %d", exit_code);
    } else if (WIFSIGNALED(status)) {
        int signo = WTERMSIG(status);
        printf("term_sig = %d", signo);
#ifdef WCOREDUMP
        if (WCOREDUMP(status)) {
            printf(" (core dump)");
        }
#endif
    }
    printf("\n");
}

int main(int argc, char* argv[])
{
    pid_t pid = fork();
    switch (pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        printf("CHILD: pid = %d\n", getpid());
        // sleep(2);
        return 123;
    default:
        // 父进程
        int status; // 保存子进程的终止状态信息，位图。
        pid_t childPid = wait(&status); // 阻塞点：一直等待，直到有子进程终止
        if (childPid > 0) {
            printf("PARENT: %d terminated\n", childPid);
            print_wstatus(status);
        }
        exit(0);
    }
    return 0;
}
```

waitpid

2024年5月22日 11:01

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

pid:

- >0 : 等待指定的子进程
- -1 : 等待任意子进程.
- 0 : 等待同进程组的子进程
- <-1: 等待指定进程组 |pid| 中的子进程.

options:

- **WNOHANG** 不阻塞.
- **WUNTRACED** stopped.
- **WCONTINUED** continue

wait(&status) 等价于 waitpid(-1, &status, 0);

waitpid(): on success, returns the process ID of the child whose state has changed; if **WNOHANG** was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

成功: 子进程的 pid

如果设置了WNOHANG, 并且没有子进程修改状态, 返回0.

失败: -1

环境变量 env

2024年5月22日 11:17

```
EXEC(3)                    Linux Programmer's Manual

NAME
       execl, execlp, execle, execv, execvp, execvpe - execute a file
```

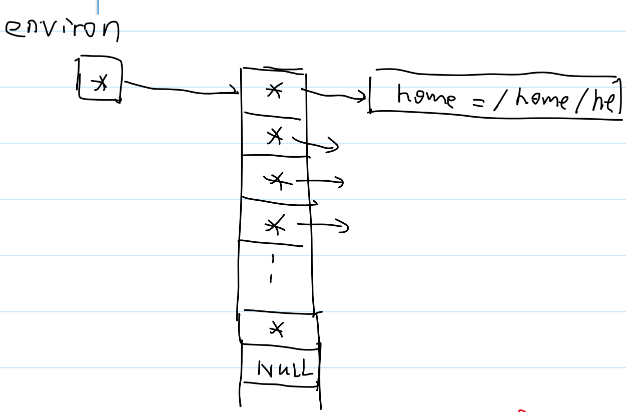
```
SYNOPSIS
       #include <unistd.h>

       extern char **environ;

       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[]);
```

外部变量

链接

环境变量

⇒ 执行程序



子进程会继承父进程环境变量

```c
echoall.c
#include <func.h>

int main(int argc, char* argv[])
{
    // 打印命令行参数
    for(int i = 0; i < argc; i++) {
        puts(argv[i]);
    }

    printf("-------------------------------------\n");
    // 打印环境变量
    extern char** environ; // 声明外部变量
    char** curr = environ;
    while (*curr) {
        puts(*curr);
        curr++;
    }
    return 0;
}
```

exec函数簇

2024年5月22日 11:31

```
SYNOPSIS
       #include <unistd.h>

       extern char **environ;                    路径名

       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[]);
```

L (list)：命令行参数以可变长参数指定，并且以NULL结尾

p (path)：会根据PATH环境变量查找可执行程序

e (environment)：会替换环境变量

v (vector)：命令行参数以数组的形式指定，并且以NULL结尾

```
RETURN VALUE
       The exec() functions return only if an error has occurred.  The re-
       turn value is -1, and errno is set to indicate the error.
```

成功：不返回

失败：-1，设置errno

```
test_exec.c
#include <func.h>


char* new_env[] = {"user=he", "aaa=hello", NULL};
char* args[] = {"./echoall", "aaa", "bbb", "ccc", NULL};
int main(int argc, char* argv[])
{
    printf("BEGIN\n");


    // execlp("echoall", "./echoall", "hello", "world", NULL);
    // execl("echoall", "./echoall", "hello", "world", NULL);
    // execle("echoall", "./echoall", "hello", "world", NULL, new_env);
    execve("echoall", args, new_env);
    printf("END\n"); // 看不到
    return 0;
}
```

exec原理

2024年5月22日 14:28

test_exec2.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    printf("pid = %d, ppid = %d\n", getpid(), getppid());

    // 执行程序
    execl("echoall", "./echoall", "aaa", "bbb", NULL);

    error(1, errno, "execl");
    return 0;
}
```

没有创建新的进程

```
he@he-vm:~/cpp58/2_Linux/Linux09 (master)$ ./test_exec2
pid = 15792, ppid = 12354
pid = 15792, ppid = 12354
Arguments:      ./echoall
    aaa
    bbb
Environments:      SHELL=/bin/bash
    LC_ADDRESS=zh_CN.UTF-8
    LC_NAME=zh_CN.UTF-8
```

从新可执行程序的第一行开始执行

原理：① 清除进程的代码段、数据段、堆、栈、上下文

② 加载新的可执行程序（设置代码段，数据段）

③ 从新可执行程序的main函数第一行开始执行

system

2024年5月22日 14:36

```
SYSTEM(3)        Linux Programmer's Manual

NAME
       system - execute a shell command

SYNOPSIS
       #include <stdlib.h>

       int system(const char *command);
```

```
test_system.c
#include <func.h>

int main(int argc, char* argv[])
{
    system("top");
    return 0;
}
```

```
)$ ps x | grep "top"

16047 pts/0    S+     0:00 sh -c top
```

↳ shell:

sh, bash, zsh, csh, ksh, ...

```
test_system.c                                              buffers
#include <func.h>

int my_system(const char* cmd) {
    pid_t pid = fork();  ①
    switch(pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程执行新的可执行程序
        execlp("sh", "sh", "-c", cmd, NULL);
        error(1, errno, "exelp");
    default:
        // 父进程
        waitpid(pid, NULL, 0);
    }
}

int main(int argc, char* argv[])
{
    // system("top");
    my_system("top");
    return 0;
}
```

① 先 fork()

模仿法: ② 子进程执行新的可执行程序

③ 父进程等待子进程结束

作业题. 简易的shell (命令行解释器)

```
for ( ; ; ) {
    读取用户输入的命令
    pid_t pid = fork()
    子进程执行命令
    父进程等待子进程结束
}
```

子进程执行命令

父进程等待子进程结束

进程之间的通信 (IPC)

2024年5月22日 14:53

InterProcess Communication

电话： 网络中TCP通信

邮件、短信， 网络中UDP通信， 消息队列

流水线． 管道 cmd1 | cmd2

讲课、开会， 共享内存 + 信号量

System V 分支

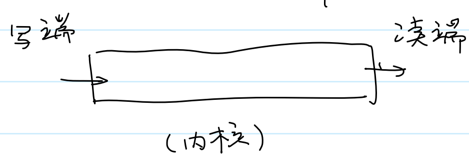
应急措施： 信号

⋮

有名管道 (fifo, named pipe)

2024年5月22日 15:04

管道：内核管理一个数据结构



半双工通信：A → B

全双工通信：A ⇄ B

```
MKFIFO(1)                User Commands

NAME
       mkfifo - make FIFOs (named pipes)
```

```
$ mkfifo pipe1
he@he-vm:~/cpp58/2_Linux/Linux09/IPC (master)$ ls -l
total 0
prw-rw-r-- 1 he he 0  5月 22 15:08 pipe1
```

← 操作文件一样操作有名管道

p1.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    int fd = open("pipe1", O_WRONLY);
    if (fd == -1) {
        error(1, errno, "open pipe1");
    }

    printf("Established\n");

    sleep(5);

    write(fd, "Hello from FIFO\n", 16);
    return 0;
}
```

→ 阻塞点：管道需要读端和写端都就绪，open 才会返回。

p2.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    int fd = open("pipe1", O_RDONLY);
    if (fd == -1) {
        error(1, errno, "open pipe1");
    }

    printf("Established\n");

    char buf[1024];
    read(fd, buf, sizeof(buf));

    printf("p2: %s\n", buf);
    return 0;
}
```

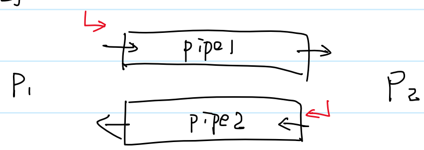
→ 阻塞点：当写端写入数据时，read 才会返回。

```
he@he-vm:~/cpp58/2_Linux/Linux09/IPC (master)$ ps x | grep "./p1"
  17011 pts/0    S+     0:00 ./p1
```

#? 能不能用管道实现全双工通信?

#2. 能不能用管道实现全双工通信?

可以的



点对点聊天系统

p1.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    int fd1 = open("pipe1", O_WRONLY);  ← 阻塞
    if (fd1 == -1) {
        error(1, errno, "open pipe1");
    }

    int fd2 = open("pipe2", O_RDONLY);
    if (fd2 == -1) {
        error(1, errno, "open pipe2");
    }

    printf("Established\n");
    return 0;
}
```

```
17347 pts/0    S+    0:00 ./p1
17348 pts/2    S+    0:00 ./p2
```

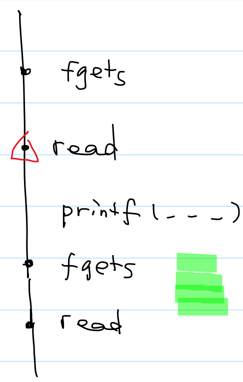
死锁

p2.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    int fd2 = open("pipe2", O_WRONLY);  ← 阻塞
    if (fd2 == -1) {
        error(1, errno, "open pipe1");
    }

    int fd1 = open("pipe1", O_RDONLY);
    if (fd1 == -1) {
        error(1, errno, "open pipe2");
    }

    printf("Established\n");
    return 0;
}
```

p1.c

```c
#include <func.h>

#define MAXLINE 256

int main(int argc, char* argv[])
{
    int fd1 = open("pipe1", O_WRONLY);
    if (fd1 == -1) {
        error(1, errno, "open pipe1");
    }

    int fd2 = open("pipe2", O_RDONLY);
    if (fd2 == -1) {
        error(1, errno, "open pipe2");
    }

    printf("Established\n");

    char recvline[MAXLINE];
    char sendline[MAXLINE];

    while (fgets(sendline, MAXLINE, stdin) != NULL) {   // 阻塞点1
        write(fd1, sendline, strlen(sendline));
        read(fd2, recvline, MAXLINE);   // 阻塞点2
        printf("from p2: %s\n", recvline);
    }

    close(fd1);
    close(fd2);

    return 0;
}
```



p2.c buffers

```c
#include <func.h>

#define MAXLINE 256

int main(int argc, char* argv[])
{
    int fd1 = open("pipe1", O_RDONLY);
    if (fd1 == -1) {
        error(1, errno, "open pipe1");
    }

    int fd2 = open("pipe2", O_WRONLY);
    if (fd2 == -1) {
        error(1, errno, "open pipe2");
    }

    printf("Established\n");

    char recvline[MAXLINE];
    char sendline[MAXLINE];

    while (fgets(sendline, MAXLINE, stdin) != NULL) {
        write(fd2, sendline, strlen(sendline));
        read(fd1, recvline, MAXLINE);
        printf("from p1: %s\n", recvline);
    }

    close(fd1);
    close(fd2);

    return 0;
}
```

原因：一个执行流程有多个阻塞点

解决办法：

一个执行流程最多只有一个阻塞点

⇓

线程。

5种I/O模型

2024年5月22日 16:09

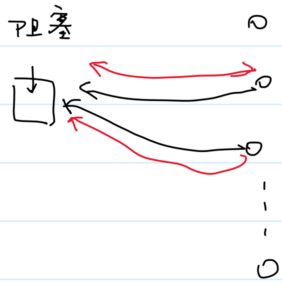
5种 I/O 模型、

阻塞I/O.

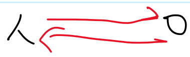
非阻塞I/O：（轮询）.

I/O多路复用.（监听多个I/O事件） 将多个阻塞点，变成一个阻塞点

select
poll
epoll



信号驱动 I/O：



异步 I/O：

select

2024年5月22日 16:20

SELECT(2) Linux Programmer's Manual SELECT(2)

NAME

select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing

同步的 I/O 多路复用

SYNOPSIS

传入传出参数

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

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

NULL，不限期阻塞

{0,0}：不阻塞

ndfs：监听的最大文件描述符 +1

readfds：传入(调用时)，表示对哪些文件描述符读事件感兴趣。

传出(返回时)，读事件已就绪的文件描述符

writefds：

exceptfds：

timeout：超时时间，最多阻塞的时间长度

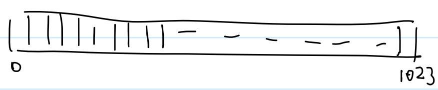
传入：超时时间

传出：还剩多长时间

NULL：不限期阻塞。

{0,0}：不阻塞。

fd_set：1024的位图




```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};
```

RETURN VALUE

On success, **select**() and **pselect**() return the number of file descriptors contained in the three returned descriptor sets (that is, the total number of bits that are set in *readfds*, *writefds*, *exceptfds*). The return value may be zero if the timeout expired before any file descriptors became ready.

成功：就绪事件的数目

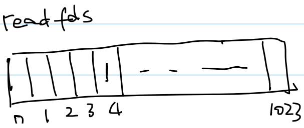
如果超时，返回0。

失败：-1，设置errno

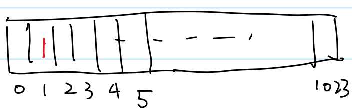
#2. 原理

用户空间

readfds



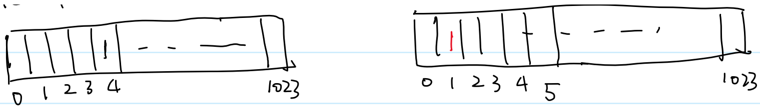
writefds



NULL

timeout {5, 1000}

用户空间



nfds =

NULL

timeout {5, 1000}

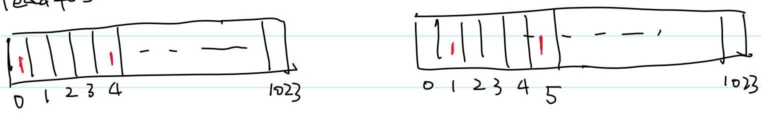
内核

select (6, &readfds, &writefds, NULL, &timeout)

提示效率

readfds

write-ds




```
typedef struct
  {

    __fd_mask fds_bits[1024 / (8 * (int) sizeof (__fd_mask))];

  } fd_set;
```

T

长度

$$\frac{1024}{8 * sizeof(T)} * sizeof(T) * 8$$

```
void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

当管道的写端关闭时，读端会读到EOF（读事件就绪）

int n = read (fd, recvline, MAXLINE);

成功 — EOF, 0

select_p1.c

```c
#include <func.h>

#define MAXLINE 256

int main(int argc, char* argv[])
{
    int fd1 = open("pipe1", O_WRONLY);
    if (fd1 == -1) {
        error(1, errno, "open pipe1");
    }

    int fd2 = open("pipe2", O_RDONLY);
    if (fd2 == -1) {
        error(1, errno, "open pipe2");
    }

    printf("Established\n");

    char recvline[MAXLINE];
    char sendline[MAXLINE];

    fd_set mainfds; // 局部变量
    FD_ZERO(&mainfds);   // 将所有的位置为0
    FD_SET(STDIN_FILENO, &mainfds);
    int maxfds = STDIN_FILENO;

    FD_SET(fd2, &mainfds);
    if (fd2 > maxfds) {
        maxfds = fd2;
    }


    for (;;) {
        fd_set readfds = mainfds;   // 结构体的复制

        // struct timeval timeout = {5, 0};
        // int events = select(maxfds + 1, &readfds, NULL, NULL, &timeout);
        int events = select(maxfds + 1, &readfds, NULL, NULL, NULL);
        switch (events) {
        case -1:
            error(1, errno, "select");
        case 0:
            // 超时
            printf("TIMEOUT\n");
            continue;
        default:
            // 打印 timeout 的值
            /* printf("timeout: tv_sec = %ld, tv_usec = %ld\n", */
            /*        timeout.tv_sec, timeout.tv_usec); */

            // STDIN_FILENO 就绪
            if (FD_ISSET(STDIN_FILENO, &readfds)) {
                // 一定不会阻塞
                fgets(sendline, MAXLINE, stdin);
                // memset(sendline, 0, MAXLINE);
                write(fd1, sendline, strlen(sendline) + 1); // +1: for '\0'
            }
            // pipe2就绪
            if (FD_ISSET(fd2, &readfds)) {
                // 一定不会阻塞
                int nbytes = read(fd2, recvline, MAXLINE);
                if (nbytes == 0) {
                    // 管道的写端关闭了
                    goto end;
                } else if (nbytes == -1) {
                    error(1, errno, "read pipe2");
                }
                printf("from p2: %s", recvline);
            }
        }
    }

end:
    close(fd1);
    close(fd2);

    return 0;
}
```

预告

2024年5月22日 21:33

1、讲作业

2. 无名管道 pipe().

3、信号 (基本用法)

4. 线程的基本操作

pthread_create

exit

join

detach

⋮