作业讲解

2024年5月23日 9:47

https://notion-next-lovat-ten.vercel.app/

```c
// 解析命令行输入，分离命令和参数
void parseInput(char* input, char* args[]) {
    int i = 0;
    args[i] = strtok(input, " \n");
    while (args[i] != NULL) {
        i++;
        args[i] = strtok(NULL, " \n");
    }
}
```

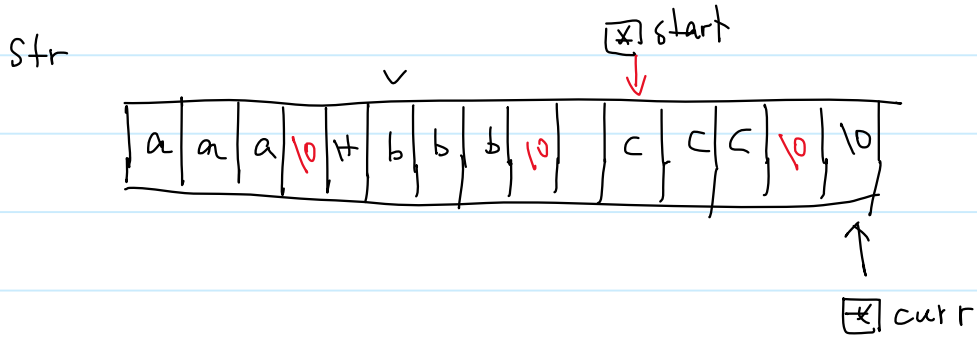
strtok

↳ token

SYNOPSIS

```c
#include <string.h>

char *strtok(char *str, const char *delim);
```

↳ 传入传出参数　　↳ 分割符 " \t\n"



static

broken_pipe1.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 world\n", 11);

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

broken_pipe2.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");

    close(fd);
    return 0;
}

```

无名管道

2024年5月23日 11:44

在文件系统上是没有名字的，只能用于有血缘关系的进程之间通信

⇓

一般是父子关系

PIPE(2) Linux Programmer's Manual

NAME

pipe, pipe2 - create pipe

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

⇓ 传参参数

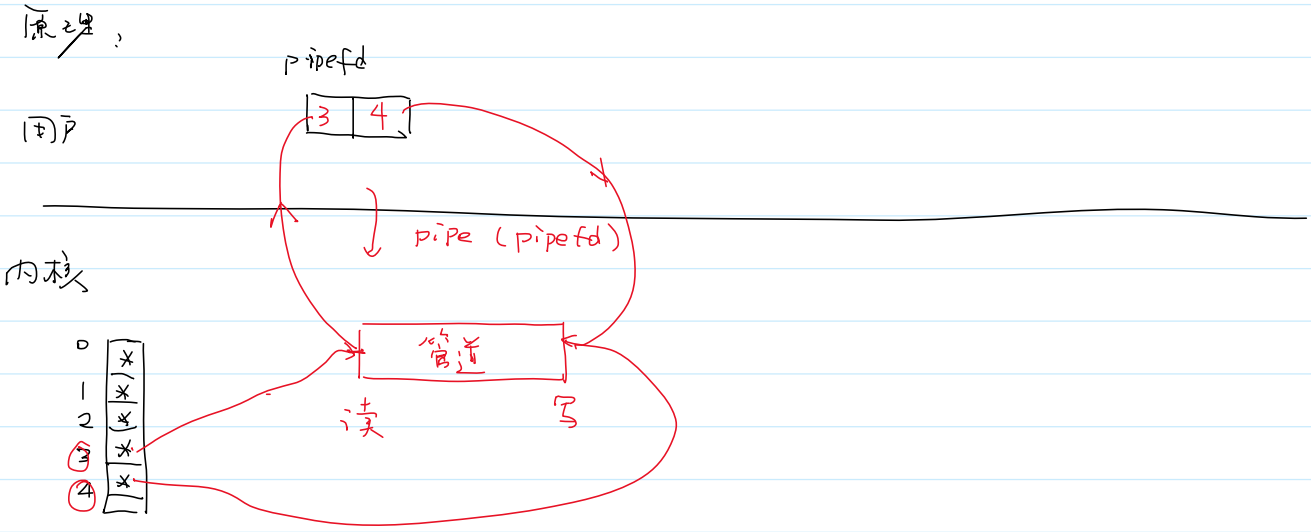
↳ 提示作用.

RETURN VALUE

On success, zero is returned. On error, -1 is returned, errno is set appropriately, and pipefd is left unchanged.

成功：0

失败：-1，设置 errno.



test_pipe1.c buffers

```c
#include <func.h>

int main(int argc, char* argv[])
{
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        error(1, errno, "pipe");
    }

    printf("pipefd[0] = %d, pipefd[1] = %d\n", pipefd[0], pipefd[1]);

    char buf[1024];

    write(pipefd[1], "Hello from pipe\n", 16);

    read(pipefd[0], buf, 1024);
    puts(buf);

    return 0;
}
```

#2.

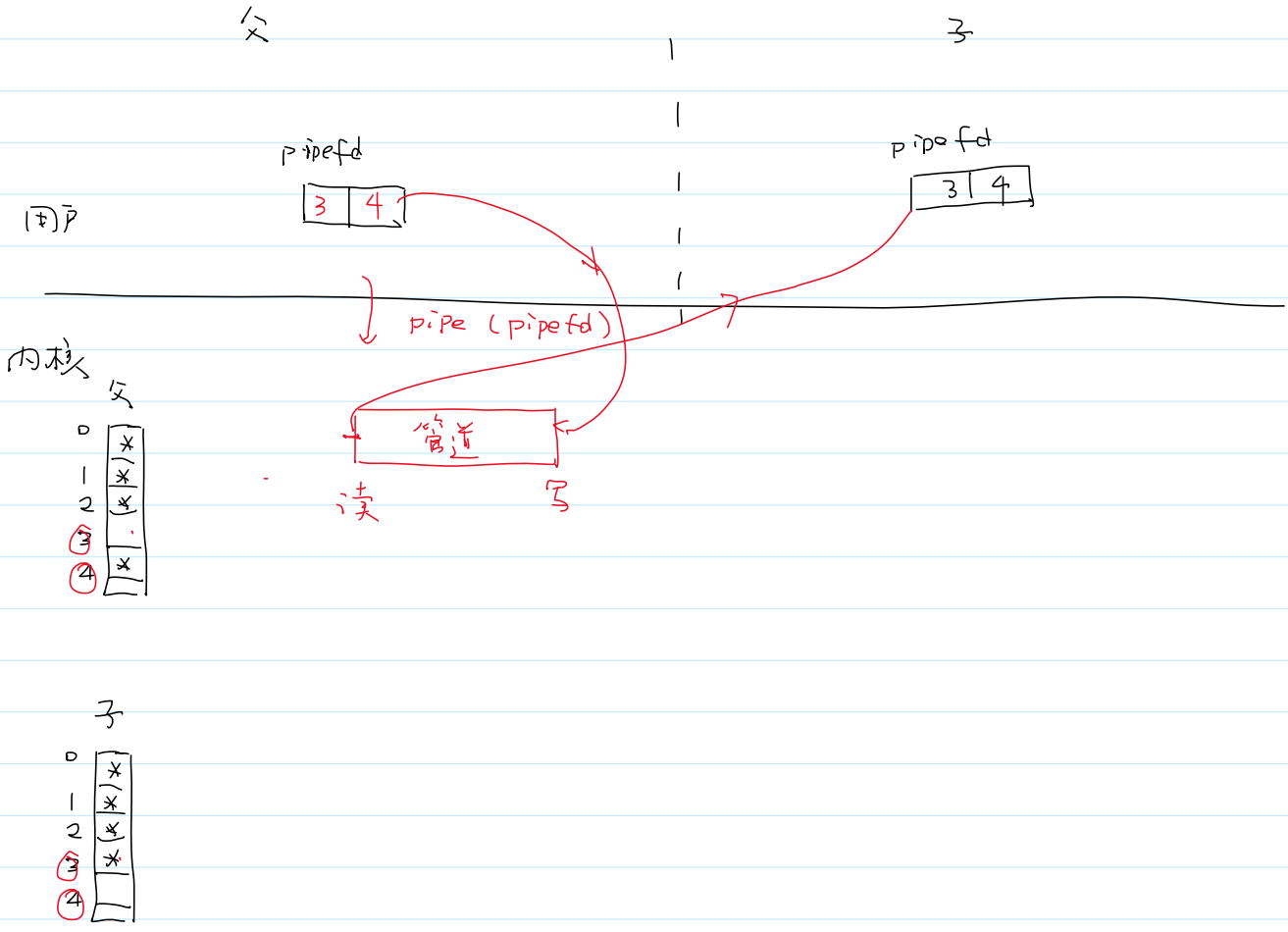
惯用法

① 先 pipe()

② 后 fork()

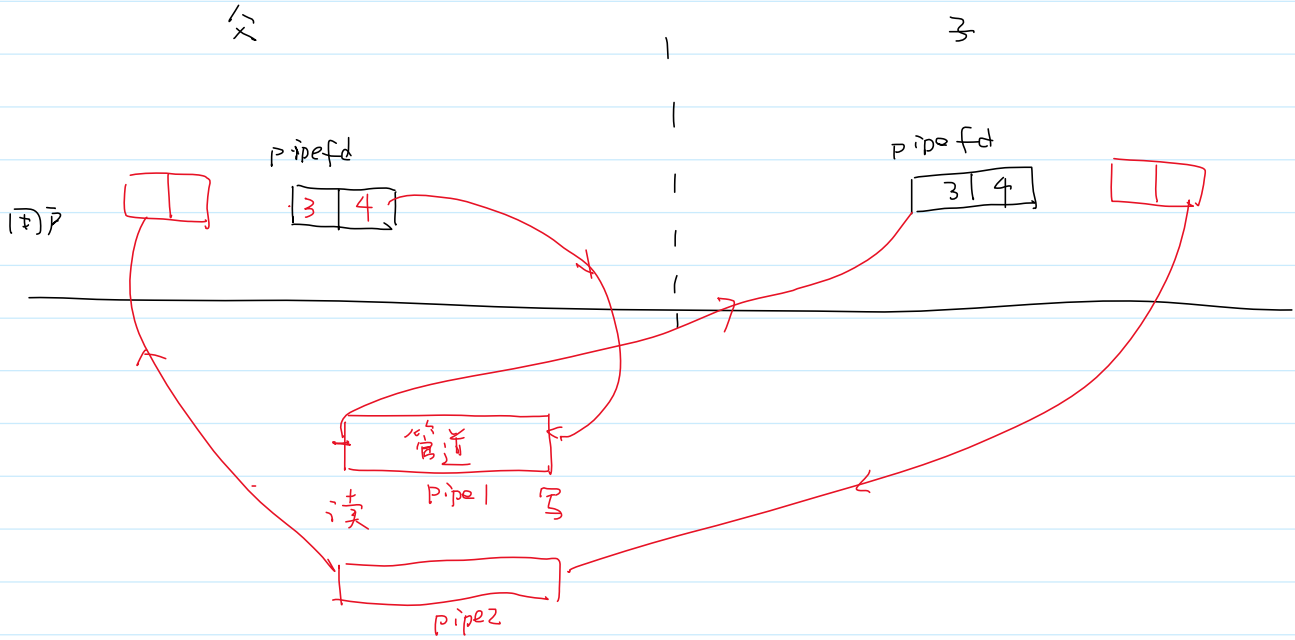
③ 父进程关闭管道一端

④ 子进程关闭管道另一端.



test_pipe2.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    // 1. 先 pipe()
    int pipefd[2];
    if (pipe(pipefd) == -1) {
        error(1, errno, "pipe");
    }

    // 2. 后 fork()
    char buf[1024];

    switch (fork()) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 3. 子进程关闭管道的另一端
        close(pipefd[1]);
        read(pipefd[0], buf, 1024);
        printf("Child: %s\n", buf);
        exit(0);
    default:
        // 4. 父进程关闭管道的一端
        close(pipefd[0]);
        sleep(2);
        write(pipefd[1], "Hello from parent", 18);
        exit(0);
    }

    return 0;
}
```

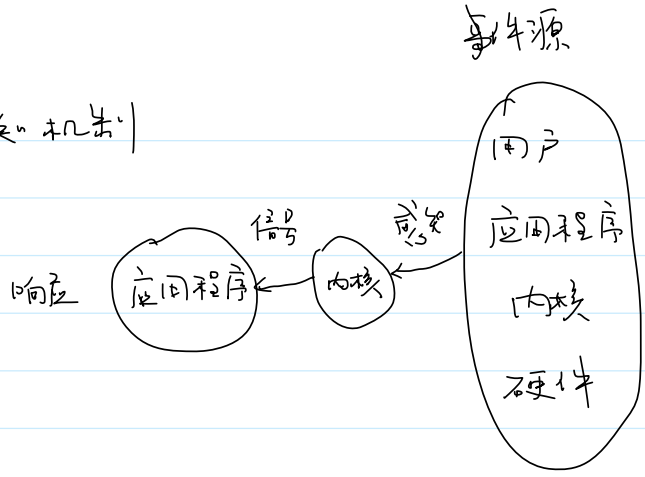



```
he@he-vm:~/cpp58/2_Linux/Linux10 (master)$ ./test_pipe3
from paretn: Homework done?
from child: Done!
```

信号

2024年5月23日 14:58

事件通知机制



select, I/O事件通知机制.

1. 事件源

硬件:

访问非法内存　　SIGSEGV　　(段错误)
　　segment Volation

执行非法的指令　　SIGILL
　　illegal

除0　　SIGFPE　　(算术异常)
　　float point exception

内核:

读端关闭:写管道　　SIGPIPE

应用程序:

abort()　　SIGABRT

子进程终止　　SIGCHLD

用户

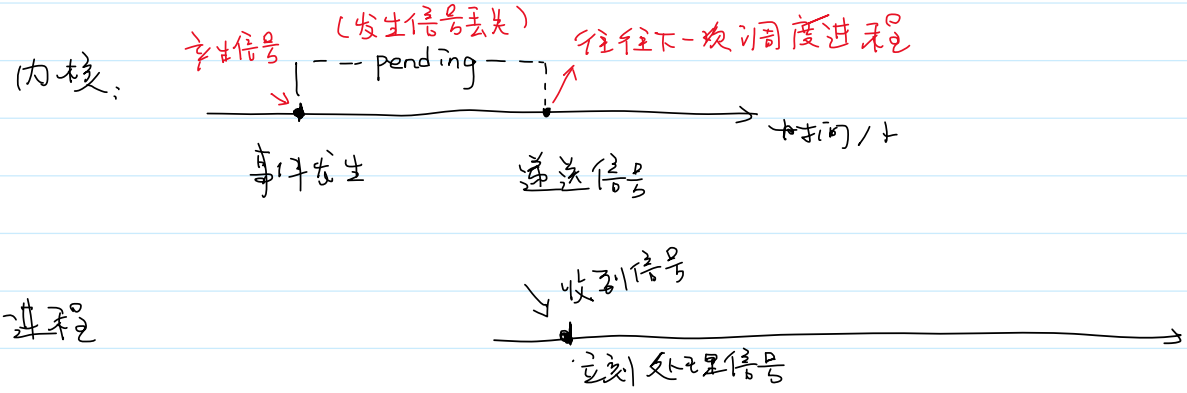
ctrl + C　　SIGINT

ctrl + \　　SIGQUIT

ctrl + z　　SIGTSTP

kill命令

#2. 内核会感知事件，并给进程发送相应的信号.



#3. `$ man 7 signal`

Signal dispositions (默认的处理方式)

Term

Ign 忽略.

Core 终止进程，并产生 core 文件

Stop

Cont

Standard signals

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from **abort**(3)
SIGALRM	P1990	Term	Timer signal from **alarm**(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for **SIGCHLD**
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for **SIGPWR**
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for **SIGABRT**
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see **pipe**(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4);

Ctrl+C

Ctrl+\

Ctrl+Z

不同的信号代表发生了不同的事件.

			synonym for **SIGIO**
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also **seccomp**(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with **SIGSYS**
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see **setrlimit**(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see **setrlimit**(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

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

用户自定义信号处理函数

信号的执行流程

2024年5月23日 15:53

```
SIGNAL(2)                                                    Linux Prog

NAME
       signal - ANSI C signal handling

SYNOPSIS
   I   #include <signal.h>

       typedef void (*sighandler_t)(int);

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

SIG_IGN 忽略

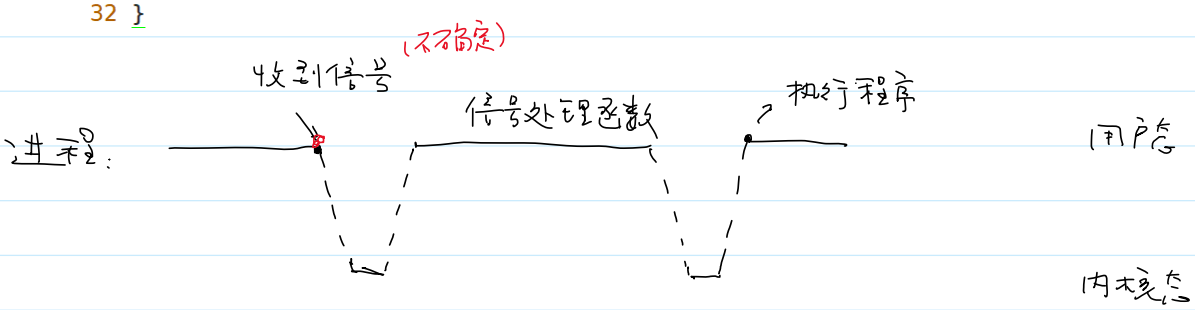
SIG_DFL 默认行为

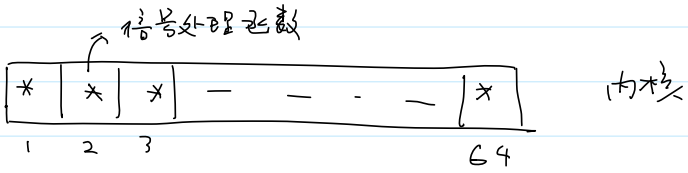
RETURN VALUE

I **signal**() returns the previous value of the signal handler, or **SIG_ERR** on error.
In the event of an error, errno is set to indicate the cause.

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

void handler(int signo) {
    switch (signo) {
    case SIGINT:
        printf("Caught SIGINT\n");
        break;
    case SIGTSTP:
        printf("Caught SIGTSTP\n");
        break;
    default:
        printf("Unknown %d\n", signo);
    }
}

int main(int argc, char* argv[])
{
    // 注册信号处理函数 (捕获信号)
    sighandler_t oldhandler = signal(SIGINT, handler);
    if (oldhandler == SIG_ERR) {
        error(1, errno, "signal %d", SIGINT);
    }

    oldhandler = signal(SIGTSTP, handler);
    if (oldhandler == SIG_ERR) {
        error(1, errno, "signal %d", SIGTSTP);
    }

    for(;;) {
    }
    return 0;
}
```

```
31      return 0;
32  }
```





信号的特点：

① 不稳定

② 异步的 （什么时候收到信号是不确定，收到信号后，会立刻马上执行信号处理函数）

③ 不同系统关于信号的语义也不一样。

注册信号处理函数

2024年5月23日 16:24

SIGNAL(2) Linux Prog

NAME

signal - ANSI C signal handling ↔ 注册信号处理函数

SYNOPSIS

```
#include <signal.h>

typedef void (*sighandler_t)(int);

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

别名 typedef 类型 别名

函数指针

SIG_IGN 忽略

SIG_DFL 默认行为

RETURN VALUE

signal() returns the previous value of the signal handler, or **SIG_ERR** on error. In the event of an error, errno is set to indicate the cause.

成功：先前的信号处理函数

失败：SIG_ERR，设置errno

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

test_signal2.c

```c
#include <func.h>

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

    // 忽略 SIGINT 信号
    sighandler_t oldhandler = signal(SIGINT, SIG_IGN);
    if (oldhandler == SIG_ERR) {
        error(1, errno, "signal SIGINT");
    }

    sleep(10);

    printf("Wake up\n");

    signal(SIGINT, SIG_DFL);

    for(;;) {

    }
    return 0;
}
```

test_signal3.c buffers

```c
#include <func.h>

void handler(int signo) {
    switch (signo) {
    case SIGKILL:
        printf("Caught SIGKILL\n");
        break;
    case SIGSTOP:
        printf("Caught SIGSTOP\n");
        break;
    }
}

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

    sighandler_t oldhandler = signal(SIGKILL, handler);
    if (oldhandler == SIG_ERR) {
        error(0, errno, "signal SIGKILL");
    }


    oldhandler =  signal(SIGSTOP, handler);
    if (oldhandler == SIG_ERR) {
        error(0, errno, "signal SIGSTOP");
    }

    for(;;) {

    }

    return 0;
}
```

发送信号

2024年5月23日 16:25

① kill 命令

```
KILL(1)                          User Commands

NAME
       kill - send a signal to a process
```

kill -SIGkill pid...

```
he@he-vm:~$ kill -9 7438
```

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

NAME
       kill - send signal to a process
```

权限。

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

       int kill(pid_t pid, int sig);
```

pid。 和 waitpid 的 pid 含义一样

>0：给指定的进程发送信号。

=0：给同进程组的所有进程发送信号。

-1：给所有能够发送信号的进程发送信号（除了init (1)）

<-1：给指定进程组的进程发送信号。

```
RETURN VALUE
       On  success  (at least one signal was sent), zero is returned.  On
       error, -1 is returned, and errno is set appropriately.
```

成功：0

失败：-1，设置 errno。

test_kill.c buffers

```c
#include <func.h>

int main(int argc, char* argv[])
{
    // ./test_kill signo pid...
    if (argc < 3) {
        error(1, 0, "Usage: %s signo pid...", argv[0]);
    }

    int signo;
    sscanf(argv[1], "%d", &signo);

    for (int i = 2; i < argc; i++) {
        pid_t pid;
        sscanf(argv[i], "%d", &pid);

        if (kill(pid, signo) == -1) {
            error(0, errno, "kill(%d, %d)", pid, signo);
        }
    }
    return 0;
}
```

RAISE(3) Linux Programmer's Manual

NAME

raise - send a signal to the caller

SYNOPSIS

```c
#include <signal.h>

int raise(int sig);
```

⇓ 等价

```c
kill(getpid(), sig);
```

```c
int err = raise(signo);
if (err) {
    ...
}
```

RETURN VALUE

raise() returns 0 on success, and nonzero for failure.

成功为 0

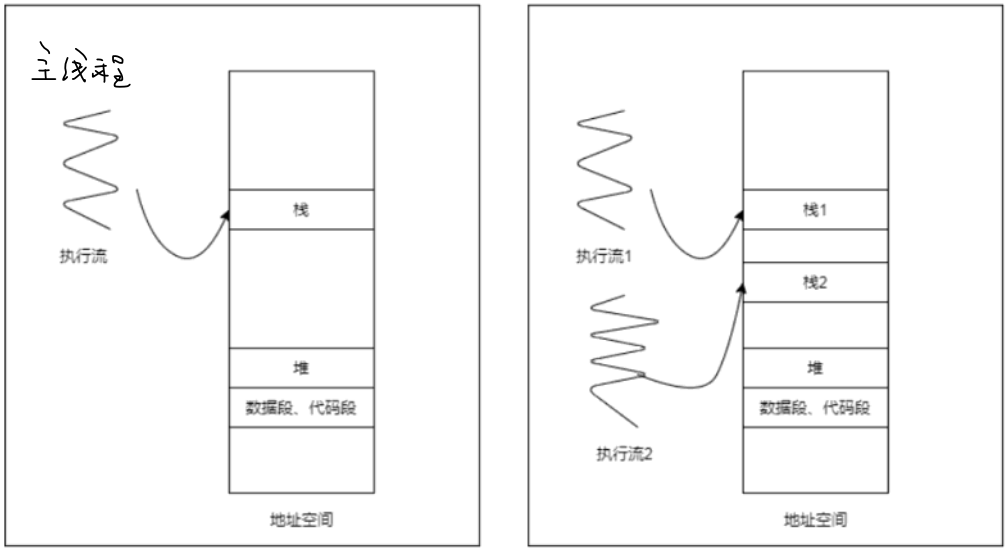
失败：非 0

线程

2024年5月23日 17:25

1. what, 什么是线程.

一条执行流程



主线程, 8M

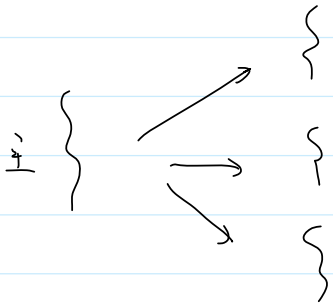
其它线程, 2M

引入线程.

进程是资源分配的最小单位.

线程是调度的最小单位.

线程共享进程的所有资源.



为什么要引入线程?

2024年5月23日 17:35

① 进程之间切换，（CPU的高速缓存，TLB失效），开销大

同进程线程之间切换，开销小.

② 进程之间通信，需要打破隔离壁障，

线程之间通信，无锁编程.

③ 进程创建和销毁比较耗时.

线程创建和销毁要求量很多

⋮

线程的基本操作

2024年5月23日 17:43

① 获取线程的标识。

② 创建线程。

③ 终止线程

④ 等待线程

⑤ 线程清理

⋮

获取线程的标识

2024年5月23日 17:44

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

NAME
       pthread_self - obtain ID of the calling thread

SYNOPSIS
       #include <pthread.h>

       pthread_t pthread_self(void);

       Compile and link with -pthread.
```

-lpthread

```
RETURN VALUE
       This function always succeeds, returning the calling thread's ID.
```

Makefile

```
1  Srcs := $(wildcard *.c)
2  Outs := $(patsubst %.c, %, $(Srcs))
3
4  CC := gcc
5  CFLAGS = -Wall -g -lpthread
6
7  ALL: $(Outs)
8
9  %: %.c
10     $(CC) $< -o $@ $(CFLAGS)
11
12 .PHONY: clean rebuild ALL
13
14 clean:
15     $(RM) $(Outs)
16 rebuild: clean ALL
```

```
l648:typedef unsigned long int pthread_t;
```

test_pthread_self.c

```
1  #include <func.h>
2
3  int main(int argc, char* argv[])
4  {
5      pthread_t tid = pthread_self();
6
7      printf("tid = %lu\n", tid);
8      return 0;
9  }
```

```
he@he-vm:~/cpp58/2_Linux/Linux10/pthread (master)$ ./test_pthread_self
tid = 136581611161408
```

创建线程

2024年5月23日 17:53

```
PTHREAD_CREATE(3)          Linux Programmer'

NAME
       pthread_create - create a new thread


SYNOPSIS
       #include <pthread.h>

       int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                          void *(*start_routine) (void *), void *arg);

       Compile and link with -pthread.
```

传出参数

返回任值.

任意参数

pthread 库设计原则：

返回值是 int，表示调用成功或失败

成功：0

失败，为错误码，不会设置 errno

thread：返回时，存放创建线程 ID

attr：线程属性，一般填 NULL，表示采用默认属性

start_routine：线程的入口函数

arg：线程的入口函数的参数

test_pthread_create.c

```c
#include <func.h>

void print_ids(const char* prefix) {
    printf("%s: ", prefix);
    printf("pid = %d, ppid = %d", getpid(), getppid());
    printf("tid = %lu\n", pthread_self());
}

void* start_routine(void* args) {
    print_ids("new_thread");

    return NULL;
}

int main(int argc, char* argv[])
{
    // 主线程
    print_ids("main");

    pthread_t tid;
    int err = pthread_create(&tid, NULL, start_routine, NULL);
    if (err) {
        error(1, err, "pthread_create");
    }

    printf("main: new thread = %lu\n", tid);
```

子线程的执行流程

主线程的执行流程

```
24      }
25
26      printf("main: new_thread = %lu\n", tid);
27
28      // 注意事项：当主线程终止时，整个进程就终止了。
29      sleep(2);
30      return 0;
31  }
```

```
he@he-vm:~/cpp58/2_Linux/Linux10/pthread (master)$ ./test_pthread_create
main: pid = 8908, ppid = 7770, tid = 130354905302848
main: new_thread = 130354901939776
new_thread: pid = 8908, ppid = 7770, tid = 130354901939776
```

预告

2024年5月23日 21:27

线程的基本操作：

pthread_create

pthread_exit

pthread_join

pthread_detach

pthread_cancel

pthread_cleanup_push

pthread_cleanup_pop

线程的同步：

互斥：互斥锁、

死锁

等待条件成立：条件变量