gdb

2024年5月17日 9:37

先一看：

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  PI
2:   y  2*PI*r
```

```
(gdb) undisplay 1
(gdb) undisplay
Delete all auto-display expressions? (y or n) y
```

```
(gdb) x/4db arr+5
0x7fffffffe164: 6       0       0       0
(gdb) x/4xb arr+5
0x7fffffffe164: 0x06    0x00    0x00    0x00
```

如何用 gdb 查看 core文件

2024年5月17日 9:56

core 文件：　　what ?

↳ 程序异常终止时的内存快照.

堆、栈、寄存器...

错误复现，恢复场景　　why ?

#1. 生成 core 文件

```
he@he-vm:~/cpp58/Linux05$ ./test
Floating point exception (core dumped)
```

```
he@he-vm:~/cpp58/Linux05$ ls
main.c  test  test.c  test_core_8_1715911726.2496
```

%e　%s　%t. timeStamp

%s 得到

SIGFPE：floating point exception（算术异常）

```
he@he-vm:~/cpp58/Linux05$ 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
```

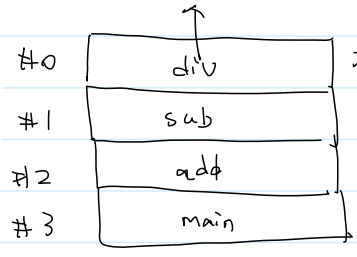
#2. 用 gdb 查看 core 文件，复现错误.

```
he@he-vm:~/cpp58/Linux05$ gdb test test_core_8_1715911726.2496
```

可执行程序　　core 文件

```
Core was generated by `./test'.
Program terminated with signal SIGFPE, Arithmetic exception.
```

```
(gdb) backtrace
#0  0x0000612af13cb158 in div (div_i=4, div_j=0) at test.c:10
#1  0x0000612af13cb19b in sub (sub_i=2, sub_j=1) at test.c:20
#2  0x0000612af13cb1e1 in add (add_i=1, add_j=0) at test.c:32
#3  0x0000612af13cb226 in main (argc=1, argv=0x7ffd8394b848) at test.c:45
```



```
(gdb) frame 3
#3  0x0000612af13cb226 in main (argc=1, argv=0x7ffd8394b848) at test.c:45
45          add(a1, b1);
```

```
(gdb) info args
argc = 1
argv = 0x7ffd8394b848
(gdb) info locals
a1 = 1
b1 = 0
c1 = 0x612af13cc02b "main function"
(gdb) info registers
rax            0x4                 4
rbx            0x0                 0
```

Makefile (了解)

2024年5月17日 10:52

#1. What

Makefile ← 解释执行 ← make

脚本文件

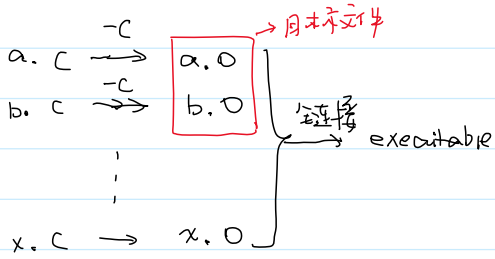
```
he@he-vm:~/cpp58/Linux05$ which make
/usr/bin/make
he@he-vm:~/cpp58/Linux05$ sudo apt install make
```

#2. Why

a. 自动编译 make

b. 增量编译

只编译新增和修改过的 .c 文件.



#3. How (如何书写 Makefile)

特点：语法要求非常严格

目标 依赖 规则 命令

```
Makefile
1  main: main.o add.o sub.o mul.o div.o
2      gcc main.o add.o sub.o mul.o div.o -o main -Wall -g
3  main.o: main.c algs.h
4      gcc -c main.c
5  add.o: add.c algs.h
6      gcc -c add.c
7  sub.o: sub.c algs.h
8      gcc -c sub.c
9  mul.o: mul.c algs.h
10     gcc -c mul.c
11 div.o: div.c algs.h
12     gcc -c div.c
```

① 目标不存在

② 依赖比目标更新

执行命令。

由规则组成

定义了文件之间的依赖关系

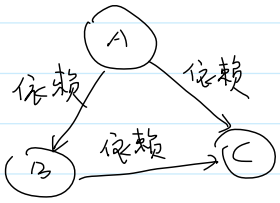
```
he@he-vm:~/cpp58/Linux05/makefile$ make
gcc -c main.c -Wall -g
gcc -c add.c -Wall -g
gcc -c sub.c -Wall -g
gcc -c mul.c -Wall -g
gcc -c div.c -Wall -g
gcc main.o add.o sub.o mul.o div.o -o main
```

⇒ 自动编译

```
he@he-vm:~/cpp58/Linux05/makefile$ make
gcc -c add.c -Wall -g
gcc -c sub.c -Wall -g
gcc main.o add.o sub.o mul.o div.o -o main
```

⇒ 增量编译

#4. 工作原理

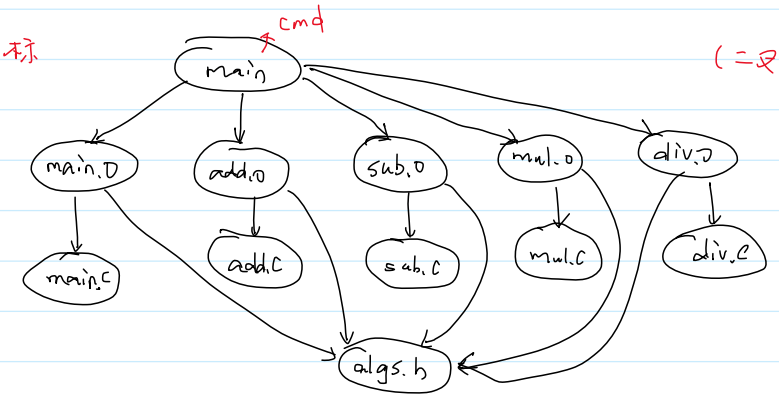


有向无环图（Directed Acyclic Graph，DAG）

Makefile

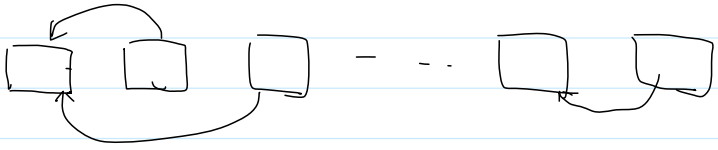
```
1  main: main.o add.o sub.o mul.o div.o
2      gcc main.o add.o sub.o mul.o div.o -o main -Wall -g
3  main.o: main.c algs.h
4      gcc -c main.c
5  add.o: add.c algs.h
6      gcc -c add.c
7  sub.o: sub.c algs.h
8      gcc -c sub.c
9  mul.o: mul.c algs.h
10     gcc -c mul.c
11 div.o: div.c algs.h
12     gcc -c div.c
```

make [目标] → 要执行的目标



（二叉树后序遍历 DFS）

（拓扑排序） DFS



保证依赖关系

伪目标

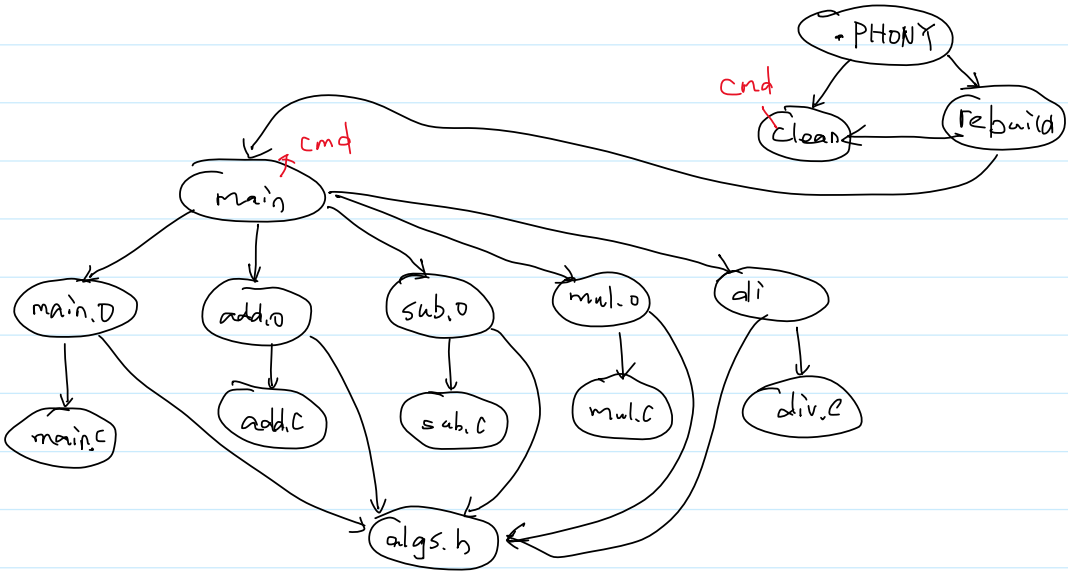
2024年5月17日 11:37

```
1 main: main.o add.o sub.o mul.o div.o
2     gcc main.o add.o sub.o mul.o div.o -o main
```

→ 命令

```
14 .PHONY: clean
15 clean:
16     rm -rf *.o main
```

⇒ 伪目标



Makefile

```
1 main: main.o add.o sub.o mul.o div.o
2     gcc main.o add.o sub.o mul.o div.o -o main
3 main.o: main.c algs.h
4     gcc -c main.c -Wall -g
5 add.o: add.c algs.h
6     gcc -c add.c -Wall -g
7 sub.o: sub.c algs.h
8     gcc -c sub.c -Wall -g
9 mul.o: mul.c algs.h
10    gcc -c mul.c -Wall -g
11 div.o: div.c algs.h
12    gcc -c div.c -Wall -g
13
14 .PHONY: clean rebuild
15 clean:
16    rm -rf *.o main
17 rebuild: clean main
```

库文件 (了解)

2024年5月17日 14:26

1. what

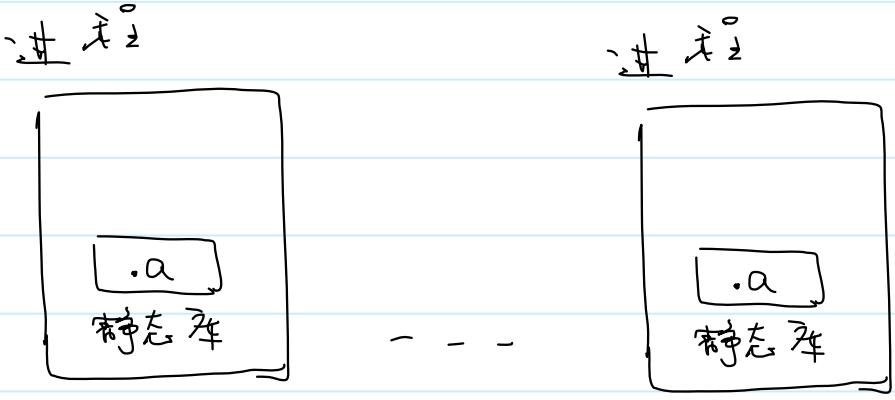
目标文件 (*.o) 集合

	windows	Linux
静态库	.lib	.a
动态库	.dll (dynamic link library)	.so

2. 区别

静态库

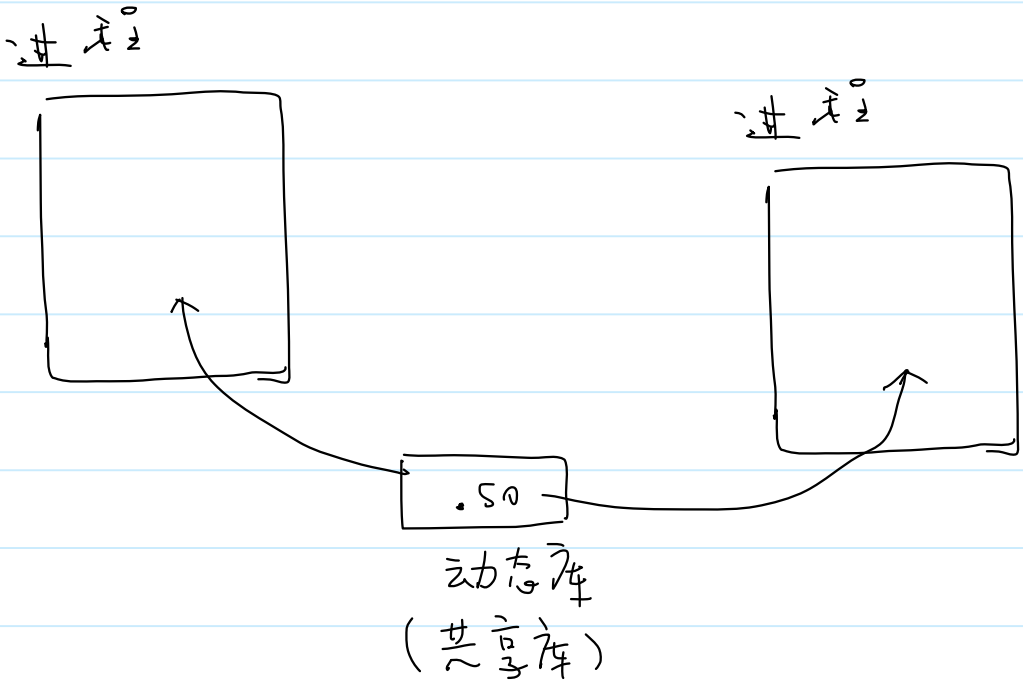
1. 静态库对函数的链接是在链接阶段完成的。
2. 程序在运行时，与静态库再无瓜葛。移植方便。
3. 浪费空间，每一个进程中都有静态库的一个副本。
4. 对程序的更新，部署，发布不友好(需要所有用户重新下载安装新的可执行程序)。



动态库:

1. 动态库对函数的链接是在运行时完成的。
2. 动态库可以在进程之间共享(所以，动态库又被称为共享库)

1. 动态库对函数的链接是在运行时完成的。
2. 动态库可以在进程之间共享(所以，动态库又被称为共享库)。
3. 对程序的更新，部署，发布友好(因为，我们只要更新动态库就好了)。
4. 程序在运行时，依赖动态库。不方便移植。



制作库

① 静态库

```
$ ar crsv libalgs.a add.o sub.o mul.o div.o

$ gcc main.c -o main -lalgs
```

② 动态库

```
$ gcc -c *.c -fpic

$ gcc -shared add.o sub.o mul.o div.o -o libalgs.so
```

```
$ gcc main.c -o main -lalgs
```

课程大纲

2024年4月22日 14:32

C语言/数据结构/算法 (15天)

C语言基础, (奇怪的, 有点知道的 就得巨大的努力) (9天)

数据结构和算法 (时间复杂度) (6天)

动态数组、链表、栈、队列、哈希表、位图、二叉树 (排序、查找)

↳ 分治算法.

词法解析器.

Linux 系统编程

Shell 命令. (4天)

文件

目录: 递归地处理目录 (递归复制、递归地删除、递归地打印目录)

普通文件, 对文件描述符的操作

进程

基本操作, 创建进程、终止进程、执行程序.

进程间的通信. (管道、信号...)

线程

基本操作, 创建、终止、等待、游离, join, 清理...

同步 { 1. 互斥地访问资源. → 锁
2. 等待某个条件成立 → 条件变量

模型: 生产者消费者模型.

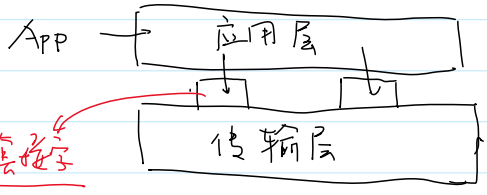
线程池

(30天)

网络

降低复杂度!

TCP/IP

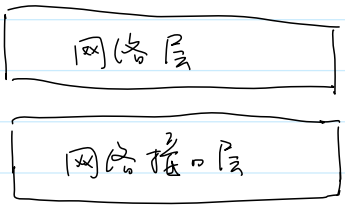


① socket 套接字

TCP: 可靠的、有连接的、全双工、流式 (电话)

UDP: 不可靠的、无连接的、数据报 (写信)

② IO多路复用模型

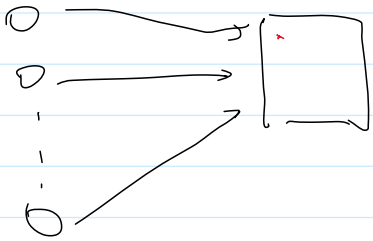


IP

select

poll

epoll



getcwd

2024年5月17日 15:45

GETCWD(3) → 库函数

NAME

getcwd, getwd, get_current_dir_name - get current working directory

SYNOPSIS

```
#include <unistd.h>
```

→ 它含的头文件

```
char *getcwd(char *buf, size_t size);
```

指针类型的参数.

buf: char * 传出参数, 可以一定会修改 buf 指向对象.

const char * 传入参数. 不会修改指针指向的对象

指针类型的返回值:

栈

数据段、代码段

堆: 负责 free

RETURN VALUE

On success, these functions return a pointer to a string containing the pathname of the current working directory. In the case of **getcwd**() and **getwd**() this is the same value as *buf*.

On failure, these functions return NULL, and *errno* is set to indicate the error. The contents of the array pointed to by *buf* are undefined on error.

DESCRIPTION

As an extension to the POSIX.1-2001 standard, glibc's **getcwd**() allocates the buffer dynamically using **malloc**(3) if *buf* is NULL. In this case, the allocated buffer has the length *size* unless *size* is zero, when *buf* is allocated as big as necessary. The caller should **free**(3) the returned buffer.

test_getcwd.c

```c
#include <func.h>

int main(void)
{
    // ./test_getcwd
    // char cwd[20];

    char* cwd;
    if ((cwd = getcwd(NULL, 0)) == NULL) {
        // 错误处理
        perror("getcwd");
        exit(1);
    }

    // getcwd一定成功
    puts(cwd);
    free(cwd);
    return 0;
}
```

error

2024年5月17日 16:29

ERROR(3)

NAME

error reporting functions

SYNOPSIS

#include <error.h>

void error(int status**, int** errnum**, const char** *format**, ...);**

0
非0：exit(status)
↳ errno

It flushes stdout, and then outputs to stderr the program name, a colon and a space, the message specified by the **printf**(3)-style format string format, and, if errnum is nonzero, a second colon and a space followed by the string given by strerror(errnum). Any arguments required for format should follow format in the argument list. The output is terminated by a newline character.

If status has a nonzero value, then **error**() calls **exit**(3) to terminate the program using the given value as the exit status.

test_error.c

```c
#include <func.h>

int main(void)
{
    char cwd[20];

    if ((getcwd(NULL, 20)) == NULL) {
        // 错误处理
        error(1, errno, "getcwd");
    }

    // getcwd一定成功
    puts(cwd);
    return 0;
}
```

```
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$ ./test_error
./test_error: getcwd: Numerical result out of range
```

\n

可执行程序名　格式串中的内容　strerror(errnum)

chdir

2024年5月17日 16:57

改变当前工作目录

`chdir` 函数可以改变当前工作目录。

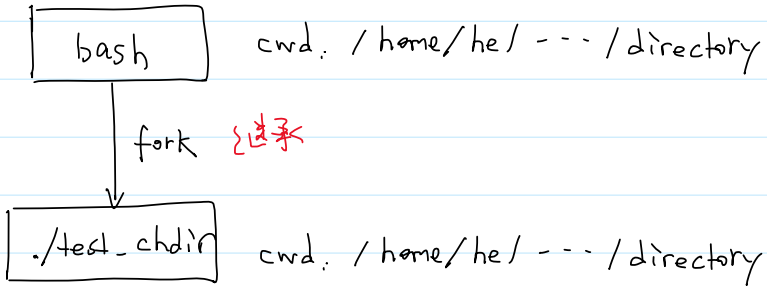
```
#include <unistd.h>

int chdir(const char *path);
参数
    path: 改变后的路径。
返回值
    成功: 返回0。
    失败: 返回-1, 并设置errno。
```

```
test_chdir.c+                                                    buffers
 1 #include <func.h>
 2
 3 int main(int argc, char* argv[])
 4 {
 5     // ./test_chdir path
 6     if (argc != 2) {
 7         error(1, 0, "Usage: %s path", argv[0]);
 8     }
 9
10     char cwd[128];
11     getcwd(cwd, 128);
12     puts(cwd);
13
14     // 惯用法：切换当前工作目录
15     if (chdir(argv[1]) == -1) {
16         error(1, errno, "chdir %s", argv[1]);
17     }
18
19     getcwd(cwd, 128);
20     puts(cwd);
21     return 0;
22 }
```

bash

```
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$ ./test_chdir ~
/home/he/cpp58/2_Linux/Linux05/directory
/home/he
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$
```

当前工作目录是进程的属性




```
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$ which cd
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$
```

内置命令

mkdir

2024年5月17日 17:14

```
#include <sys/stat.h>
#include <sys/types.h>

int mkdir(const char *pathname, mode_t mode);
参数
    pathname: 要创建目录的路径
    mode: 目录的权限位，会受文件创建掩码umask的影响，实际的权限为(mode & ~umask & 0777)
返回值
    成功: 返回0
    失败: 返回-1，并设置errno
```

test_mkdir.c+

```c
#include <func.h>

int main(int argc, char* argv[])
{
    // ./test_mkdir dir mode(八进制)
    // 参数校验
    if (argc != 3) {
        error(1, 0, "Usage: %s dir mode", argv[0]);
    }

    // 参数类型转换
    mode_t mode;
    sscanf(argv[2], "%o", &mode);
    int err = mkdir(argv[1], mode);
    if (err) {
        error(1, errno, "mkdir %s", argv[1]);
    }

    return 0;
}
```

```
$ gcc -E test_mkdir.c -o test_mkdir.i
$ grep -nE "mode_t" test_mkdir.i
130:typedef unsigned int __mode_t;
1231:typedef __mode_t mode_t;
```

```
$ ./test_mkdir dir 756
he@he-vm:~/cpp58/2_Linux/Linux05/directory (master)$ ls -l
total 316
drwxr-xr-- 2 he he   4096  5月 17 17:25 dir
```

rmdir

2024年5月17日 17:28

删除空目录

`rmdir` 可以删除空目录。

```
#include <unistd.h>

int rmdir(const char *pathname);
参数
    pathname: 要删除的目录
返回值
    成功: 返回0
    失败: 返回-1, 并设置errno
```

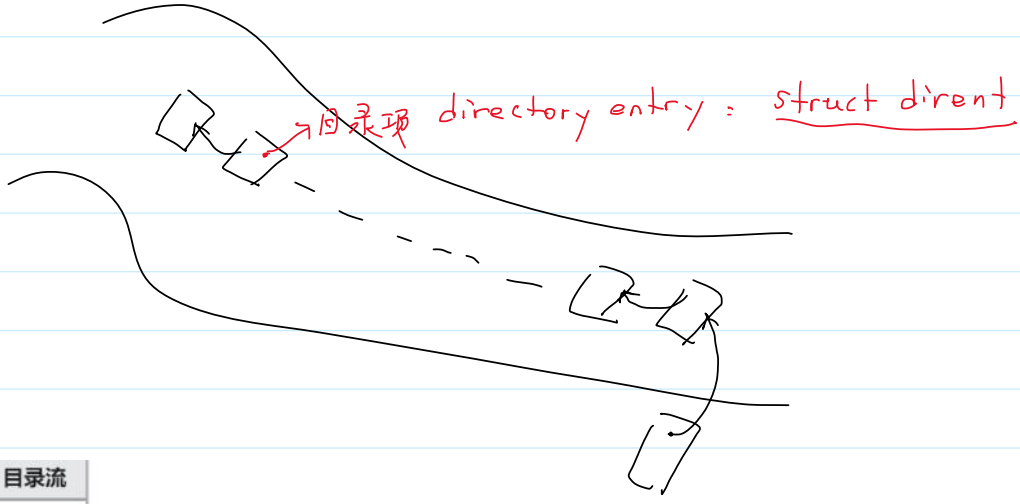
test_rmdir.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    // ./test_rmdir dir
    if (argc != 2) {
        error(1, 0, "Usage: %s dir", argv[0]);
    }

    if (rmdir(argv[1]) == -1) {
        error(1, errno, "rmdir %s", argv[1]);
    }
    return 0;
}
```

目录流

2024年5月17日 17:34

流模型



文件流	目录流
fopen	opendir
fclose	closedir
fread	readdir
fwrite	×
ftell	telldir
fseek	seekdir
rewind	rewinddir

→读取下一个目录项

X

`readdir` 读目录流，得到指向下一个目录项的指针。

```
#include <dirent.h>

struct dirent* readdir(DIR *dirp);
```

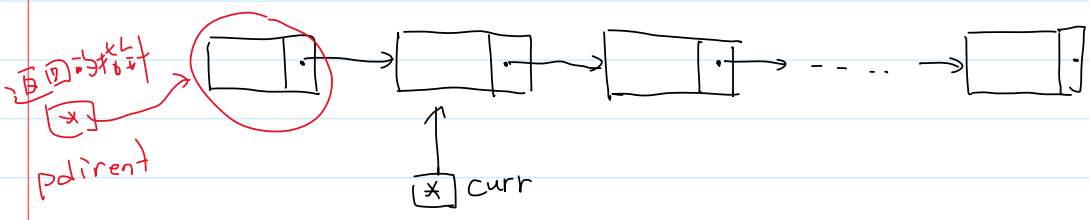
参数

dirp：指向目录流的指针

返回值

成功：返回指向结构体dirent的指针；如果读到流的末尾，返回NULL，不改变errno的值。

失败：返回NULL，并设置errno



结构体dirent的定义如下： // dirent: directory entry

```
struct dirent {
    ino_t   d_ino;              /* inode编号 */
    off_t   d_off;
    unsigned short d_reclen;    /* 结构体的长度(d_name在有些实现上是一个可变长数组) */
    unsigned char  d_type;      /* 文件的类型 */
    char           d_name[256]; /* 文件名 */
};
```

(inode, name)

d_type的可选值如下：

```
DT_BLK      This is a block device.
DT_CHR      This is a character device.
DT_DIR      This is a directory.
DT_FIFO     This is a named pipe (FIFO).
DT_LNK      This is a symbolic link.
DT_REG      This is a regular file.
DT_SOCK     This is a UNIX domain socket.
DT_UNKNOWN  The file type could not be determined.
```

预告

2024年5月17日 22:33

① 递归处理目录

② open, close, read, write, lseek