ECF: Signals & Nonlocal Jumps & System Level I/O

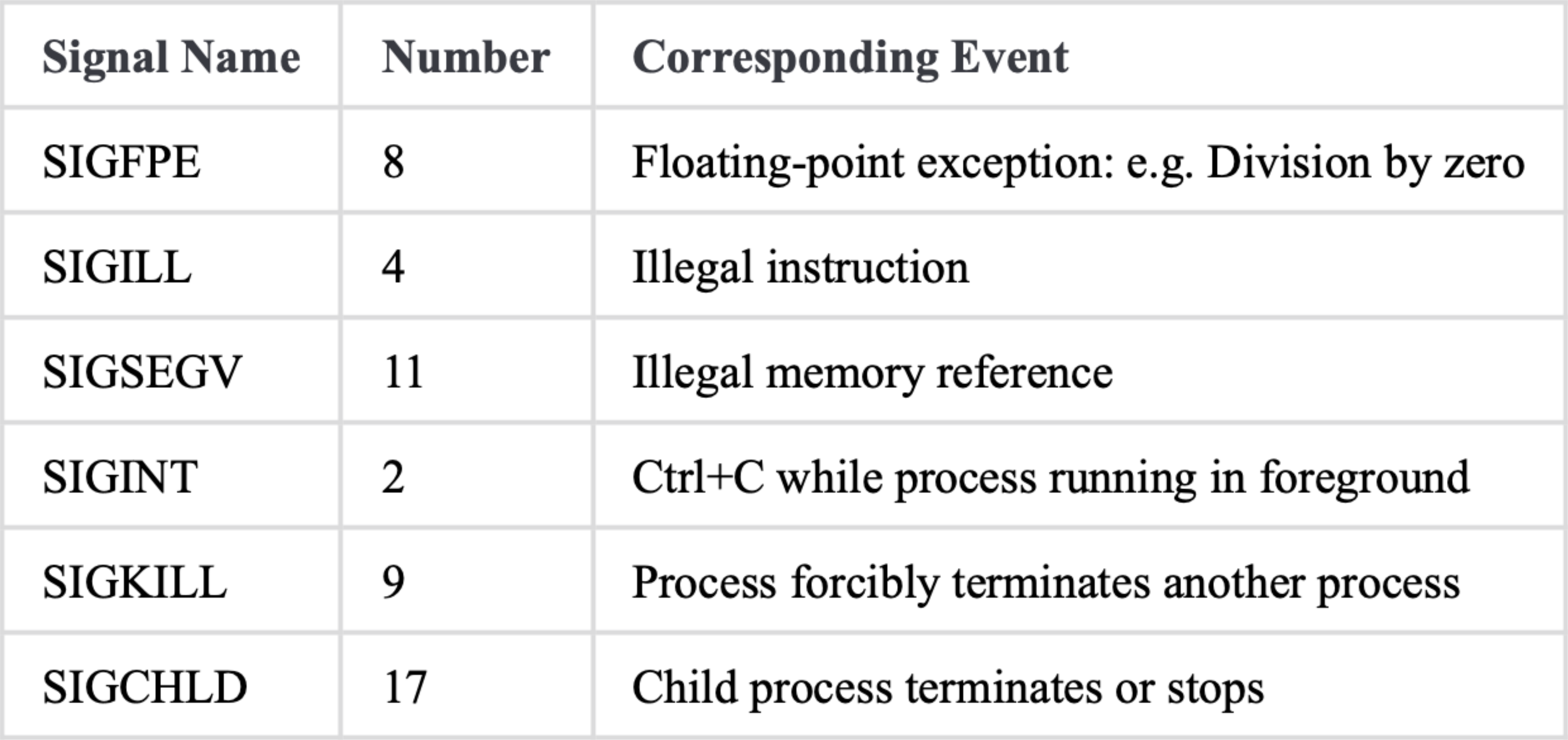
王善上 倪嘉怡 许珈铭

2023.11.29

ECF: Signals & Nonlocal Jumps (CS:APP Ch. 8.5-8.7)

倪嘉怡

Common signals



Signal Name	Number	Corresponding Event
SIGFPE	8	Floating-point exception: e.g. Division by zero
SIGILL	4	Illegal instruction
SIGSEGV	11	Illegal memory reference
SIGINT	2	Ctrl+C while process running in foreground
SIGKILL	9	Process forcibly terminates another process
SIGCHLD	17	Child process terminates or stops

- SIGKILL & SIGSTOP: can be neither caught nor ignored

Signal Transfer

1. **Sending a Signal**
   1. System event detection (e.g., divide-by-zero error, child process termination).
   2. Explicit request by a process using `kill` function.
      - A process can send a signal to itself.

2. **Receiving a Signal**
   1. Ignore.
   2. Terminate.
   3. Catch the signal with a signal handler.

Signal States and Handling

- **Pending Signal**
  - A signal that is sent but not yet received.
  - Only one pending signal of a particular type at any time.
  - Subsequent signals of the same type are not queued; they're discarded.

- **Blocking Signals**
  - A process can block certain signals.
  - Blocked signals are delivered but remain pending until unblocked.

Process Group

- Each process group has a unique Process Group ID (PGID).
- Creating a Process Group:
  - **fork()**: same process group as its parent process
  - **setpgid(pid_t pid, pid_t pgid):** set process **pid**’s process group as **pgid**
    - pid == 0: current process
    - pgid == 0: new Process Group ID set to pid (this process will become leader of the new process group)
- Process group leader: leader’s pid == pgid

Sending Signals

- **/bin/kill** –signal pid\-pgid
  - Positive: pid
  - Negative: every process in pgid
- Keyboard
  - Ctrl+C: SIGINT
  - Ctrl+Z: SIGTSTP
- int **kill**(pid_t pid, int sig);
  - pid > 0: pid
  - pid == 0: every process in process group of calling process
  - Pid < 0: every process in process group |pid|
- unsigned int **alarm**(unsigned int secs);
  - send SIGALRM to calling process in *secs* seconds

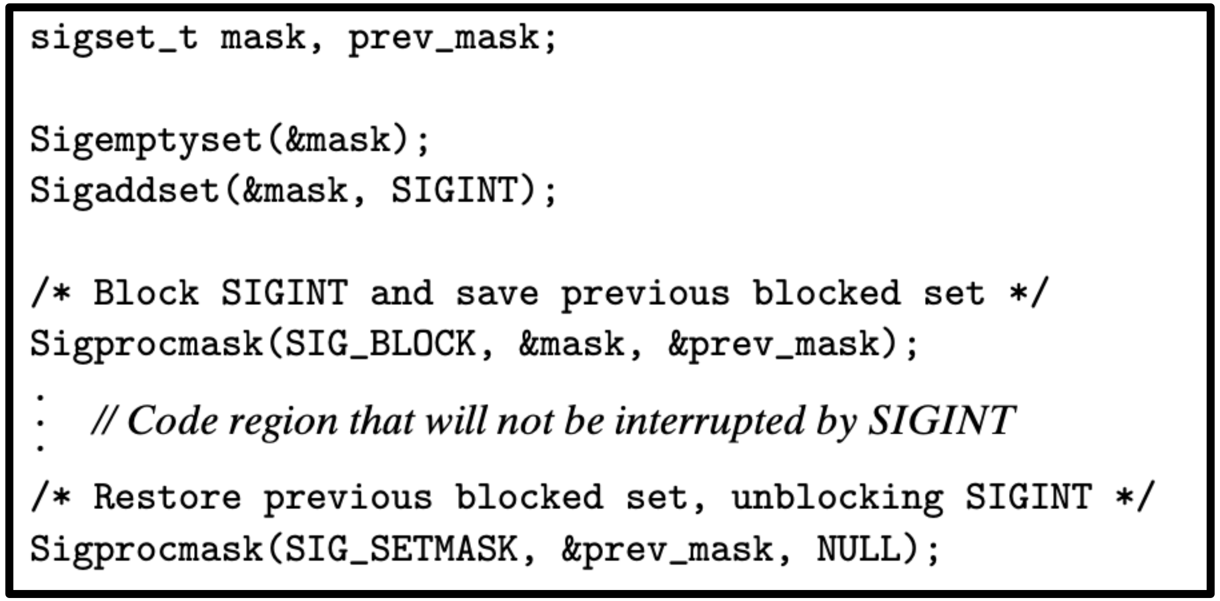
Receiving Signals

- Default action
  - terminate
  - terminate and dump core
  - stop until restarted by a SIGCONT signal
  - Ignore
- signal(int signum, sighandler_t handler);
  - handler is SIG_IGN: signals of *signum* are ignored
  - handler is SIG_DFL: action for *signum* reverts to default action
  - else: custom fn, called whenever process receives *signum*

- Signal handlers can be interrupted by other handlers

Blocking and Unblocking Signals

- sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
- how (oldset == blocked)
  - SIG_BLOCK: blocked = blocked | set
  - SIG_UNBLOCK: blocked = blocked & ~set
  - SIG_SETMASK: blocked = set



```
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set */
Sigprocmask(SIG_BLOCK, &mask, &prev_mask);
.
:   // Code region that will not be interrupted by SIGINT
.
/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

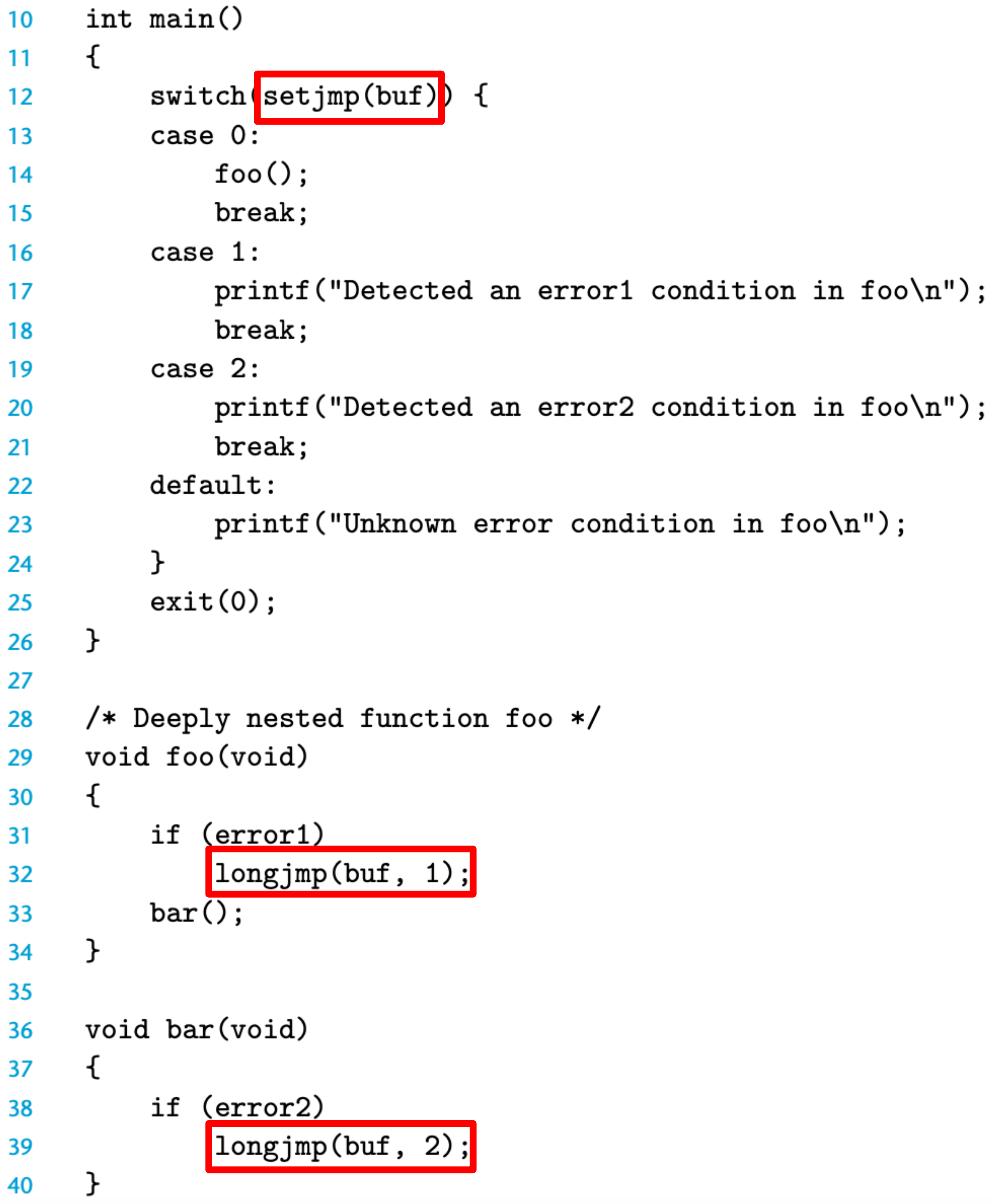
Writing signal handlers

- G1. Call only async-signal-safe functions in your handlers
  - not call *printf, sprintf, malloc, exit*
- G2. Save and restore *errno* on entry and exit
  - interfere with other parts of the program that rely on *errno*
- G3. Temporarily block signals to protect shared data.
  - prevent race conditions when writing to shared data
- Avoid the use of global variables
  - *Volatile*

- Call only asynchronous-safe functions within signal handlers *https://wiki.sei.cmu.edu/confluence/display/c/SIG30-C.+Call+only+asynchronous-safe+functions+within+signal+handlers*
- Ghosts of Unix past *https://lwn.net/Articles/414618/*

Nonlocal jumps

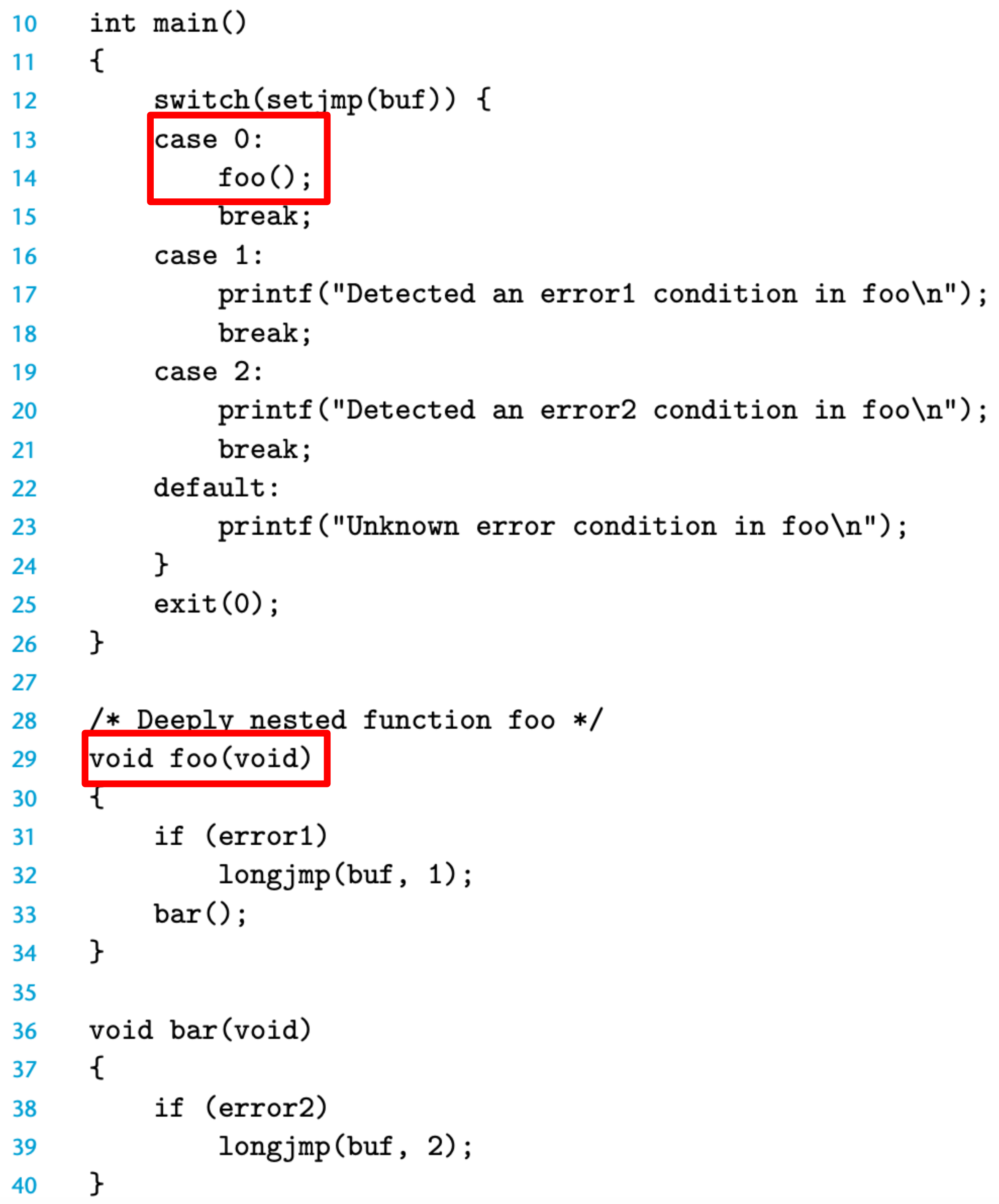
- setjmp
  - save calling environment in the *env* buffer
  - return 0
- longjmp
  - restore calling environment from the *env* buffer
  - return from the most recent *setjmp* call that initialized *env*



```c
int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

Nonlocal jumps

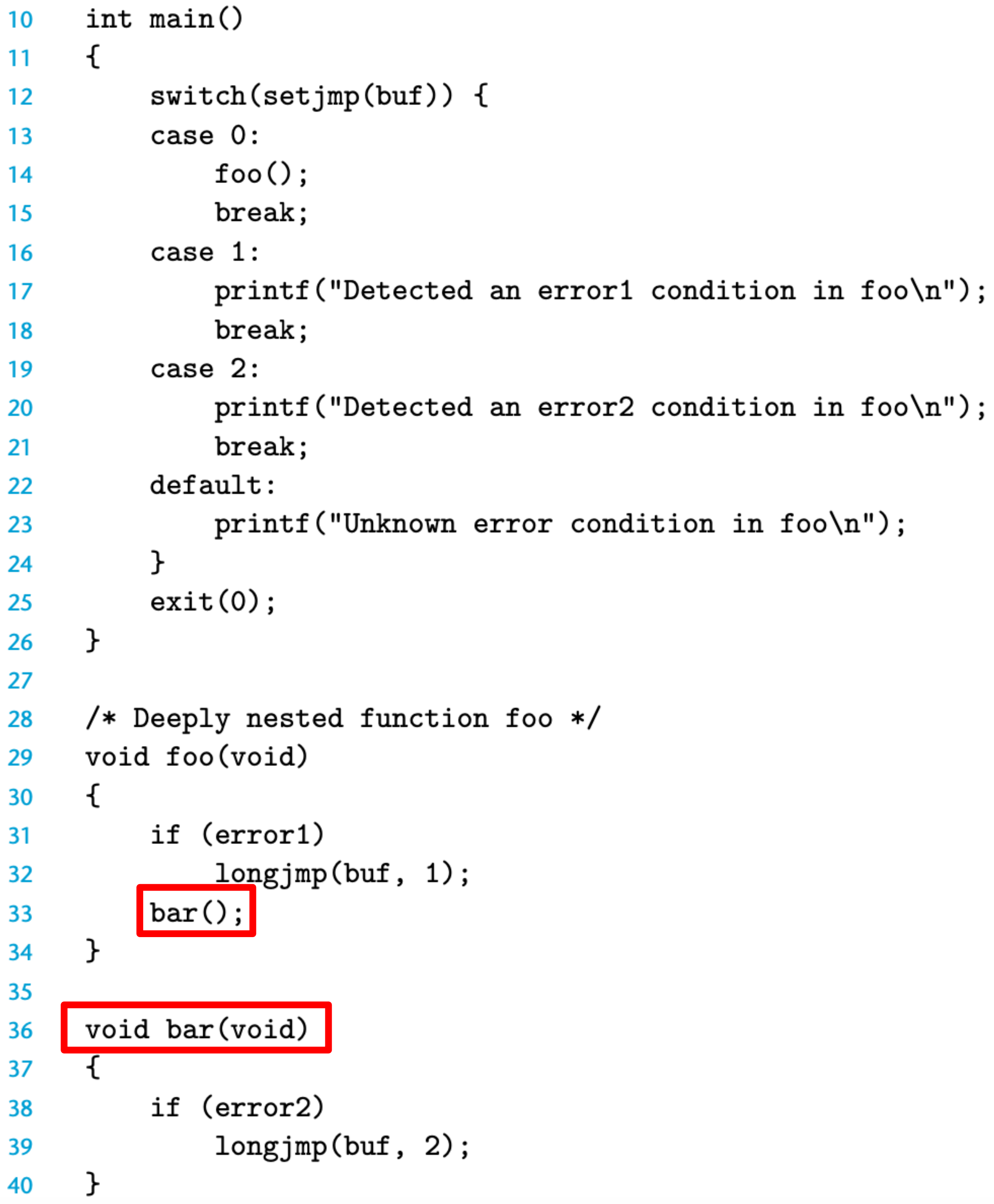
- setjmp
  - save calling environment in the *env* buffer
  - return 0
- longjmp
  - restore calling environment from the *env* buffer
  - return from the most recent *setjmp* call that initialized *env*



```
int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

Nonlocal jumps

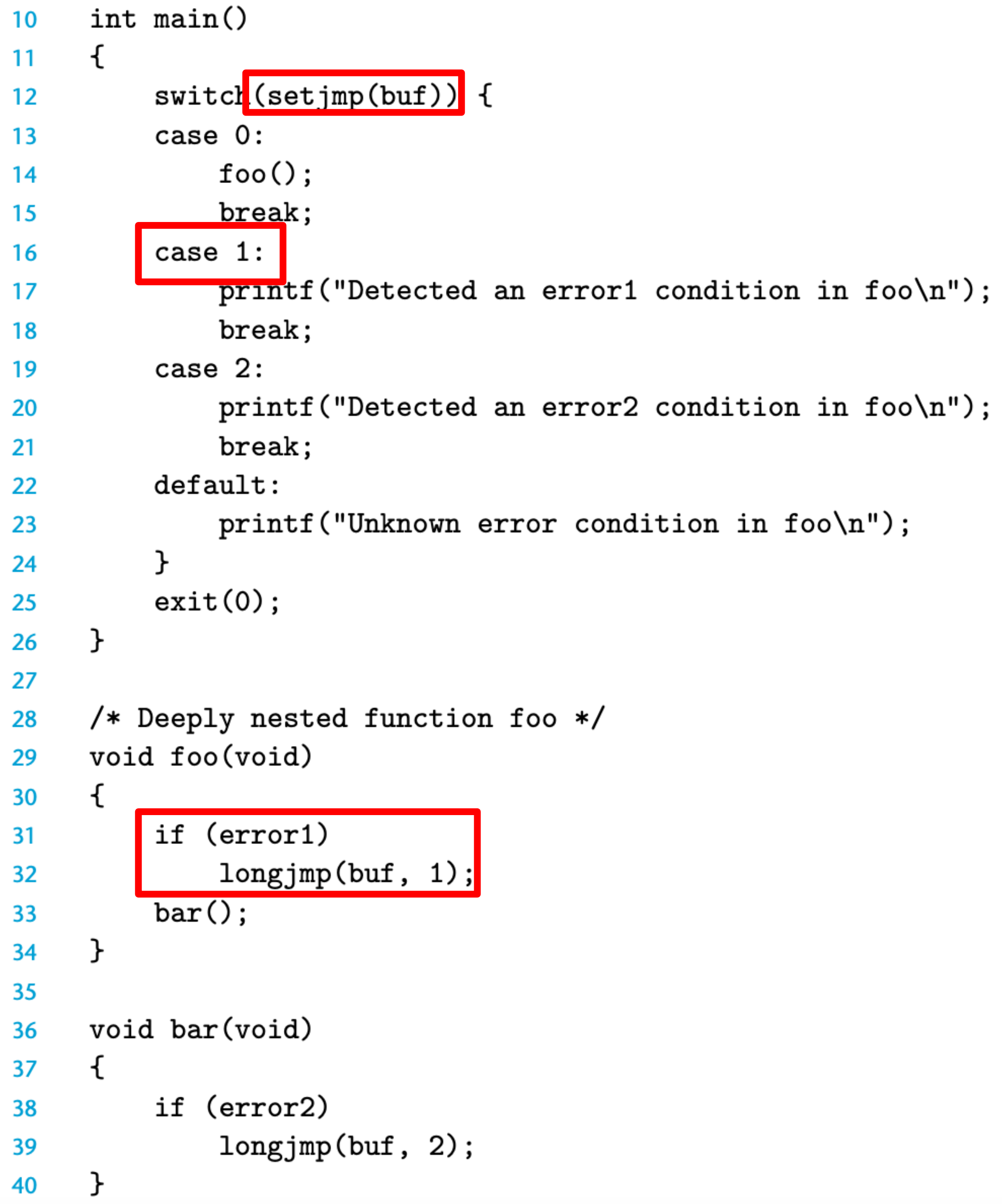
- setjmp
  - save calling environment in the *env* buffer
  - return 0
- longjmp
  - restore calling environment from the *env* buffer
  - return from the most recent *setjmp* call that initialized *env*



```
int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

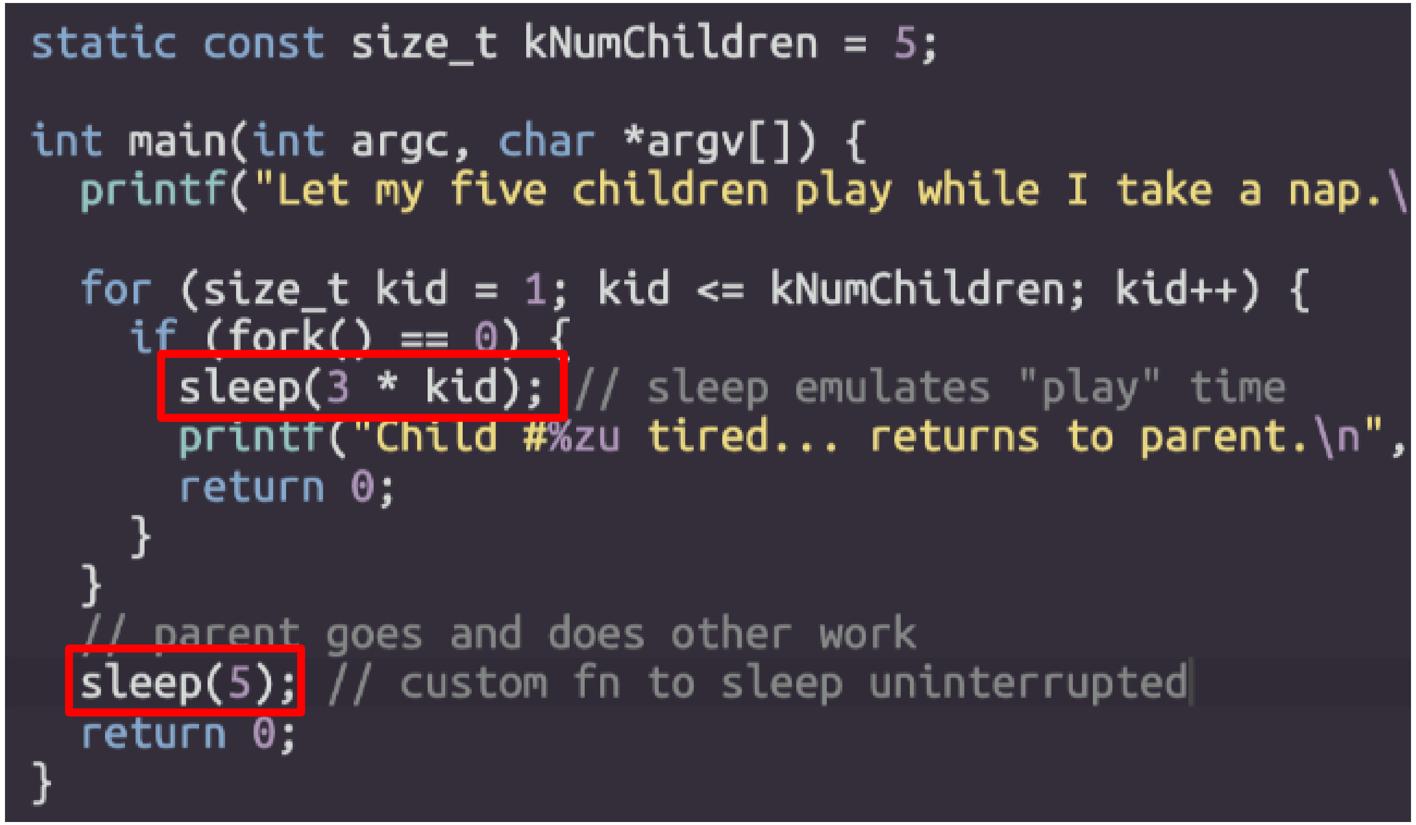
Nonlocal jumps

- setjmp
  - save calling environment in the *env* buffer
  - return 0
- longjmp
  - restore calling environment from the *env* buffer
  - return from the most recent *setjmp* call that initialized *env*

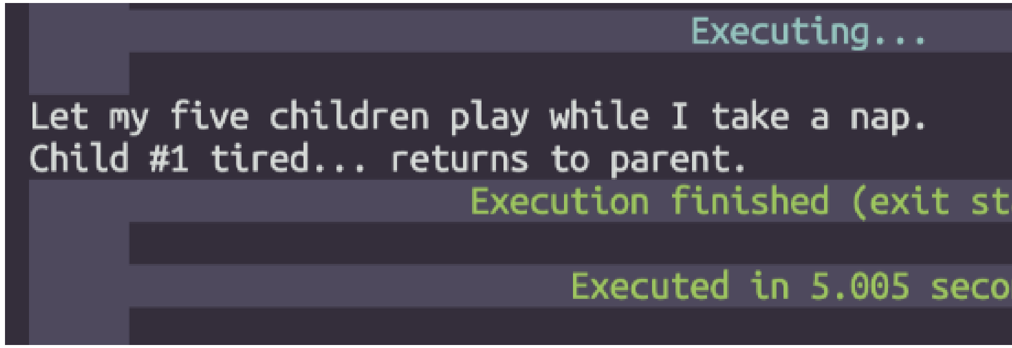


```
int main()
{
    switch(setjmp(buf)) {
    case 0:
        foo();
        break;
    case 1:
        printf("Detected an error1 condition in foo\n");
        break;
    case 2:
        printf("Detected an error2 condition in foo\n");
        break;
    default:
        printf("Unknown error condition in foo\n");
    }
    exit(0);
}

/* Deeply nested function foo */
void foo(void)
{
    if (error1)
        longjmp(buf, 1);
    bar();
}

void bar(void)
{
    if (error2)
        longjmp(buf, 2);
}
```

Disney example (cs110)

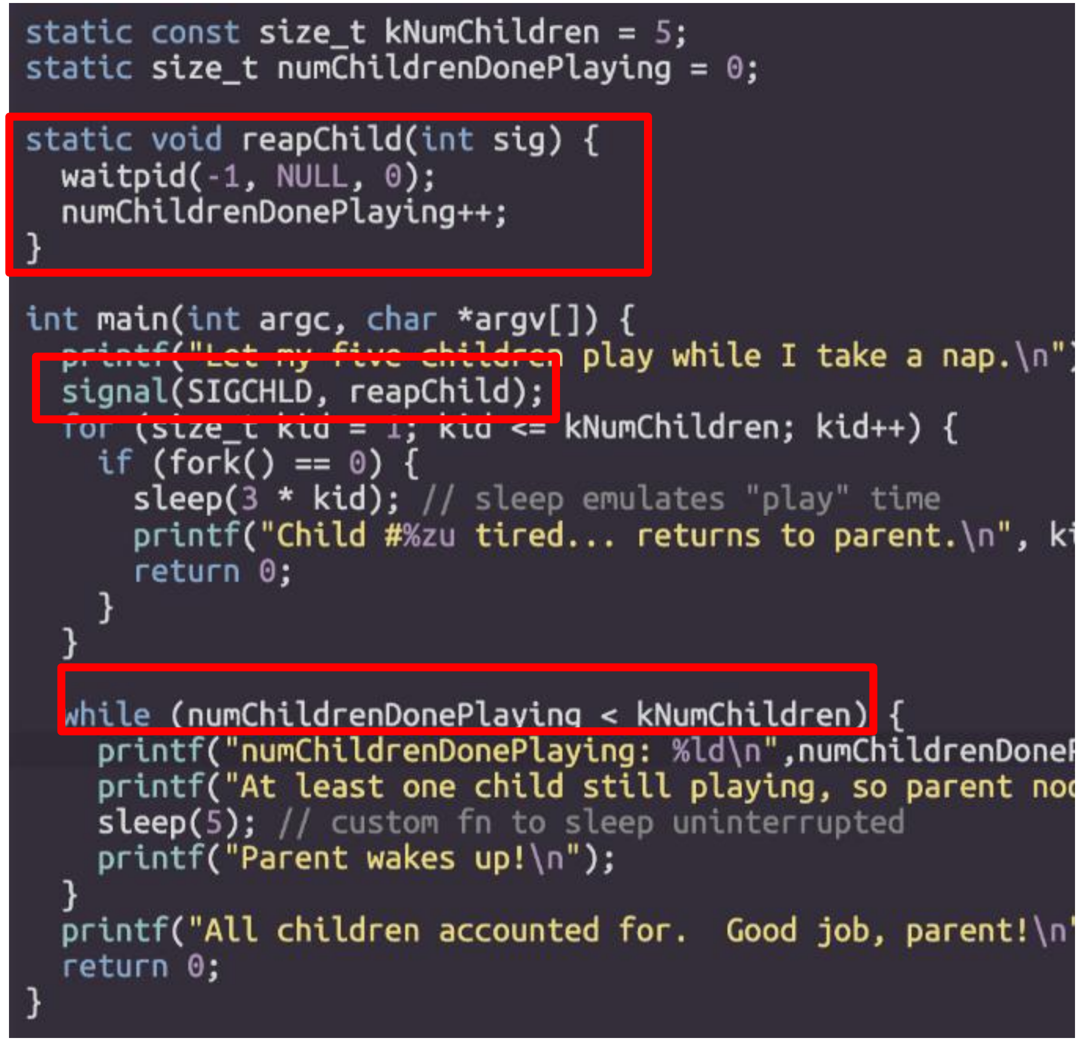


```
static const size_t kNumChildren = 5;

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\

  for (size_t kid = 1; kid <= kNumChildren; kid++) {
    if (fork() == 0) {
      sleep(3 * kid); // sleep emulates "play" time
      printf("Child #%zu tired... returns to parent.\n",
      return 0;
    }
  }
  // parent goes and does other work
  sleep(5); // custom fn to sleep uninterrupted
  return 0;
}
```

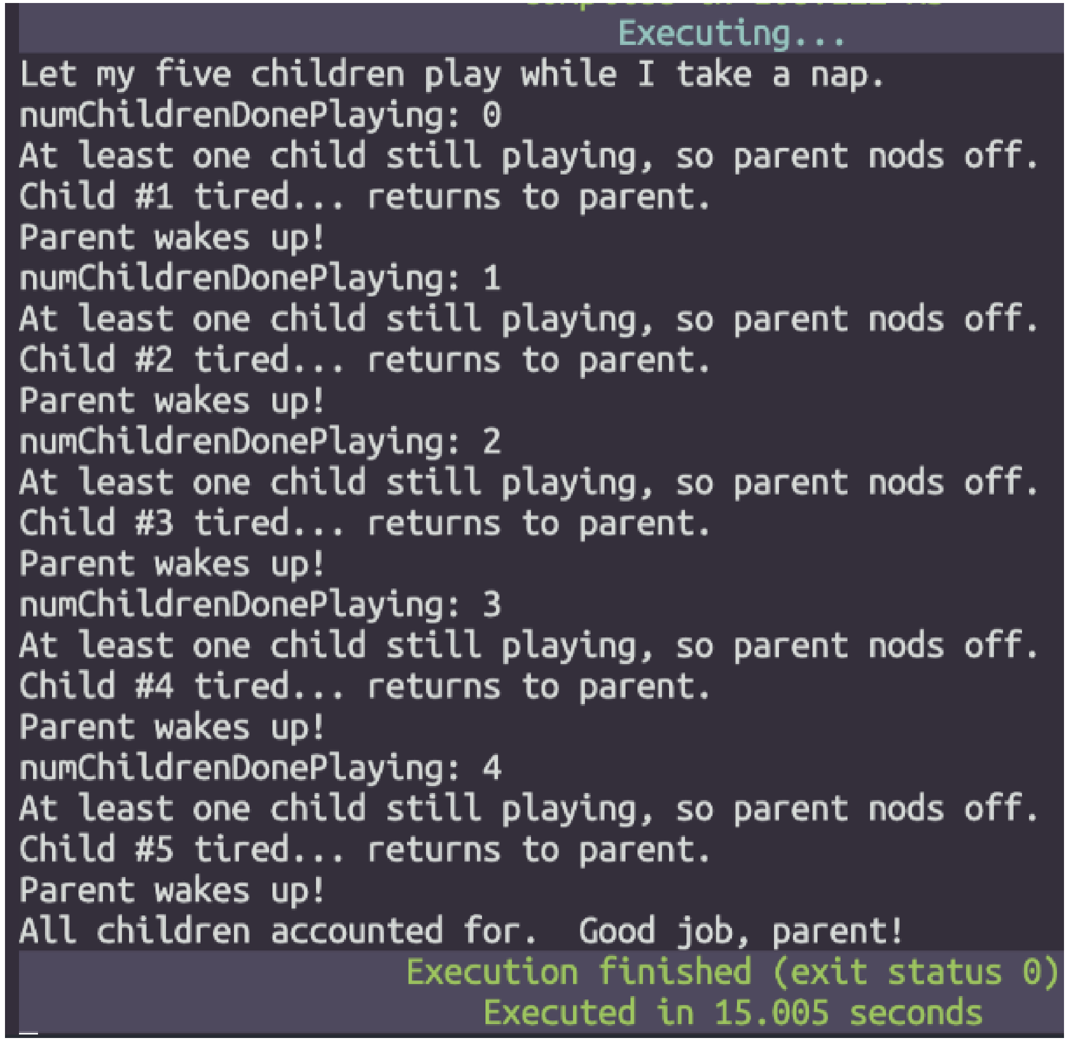


```
                             Executing...

Let my five children play while I take a nap.
Child #1 tired... returns to parent.
                      Execution finished (exit st

                         Executed in 5.005 seco
```

Disney example (cs110)

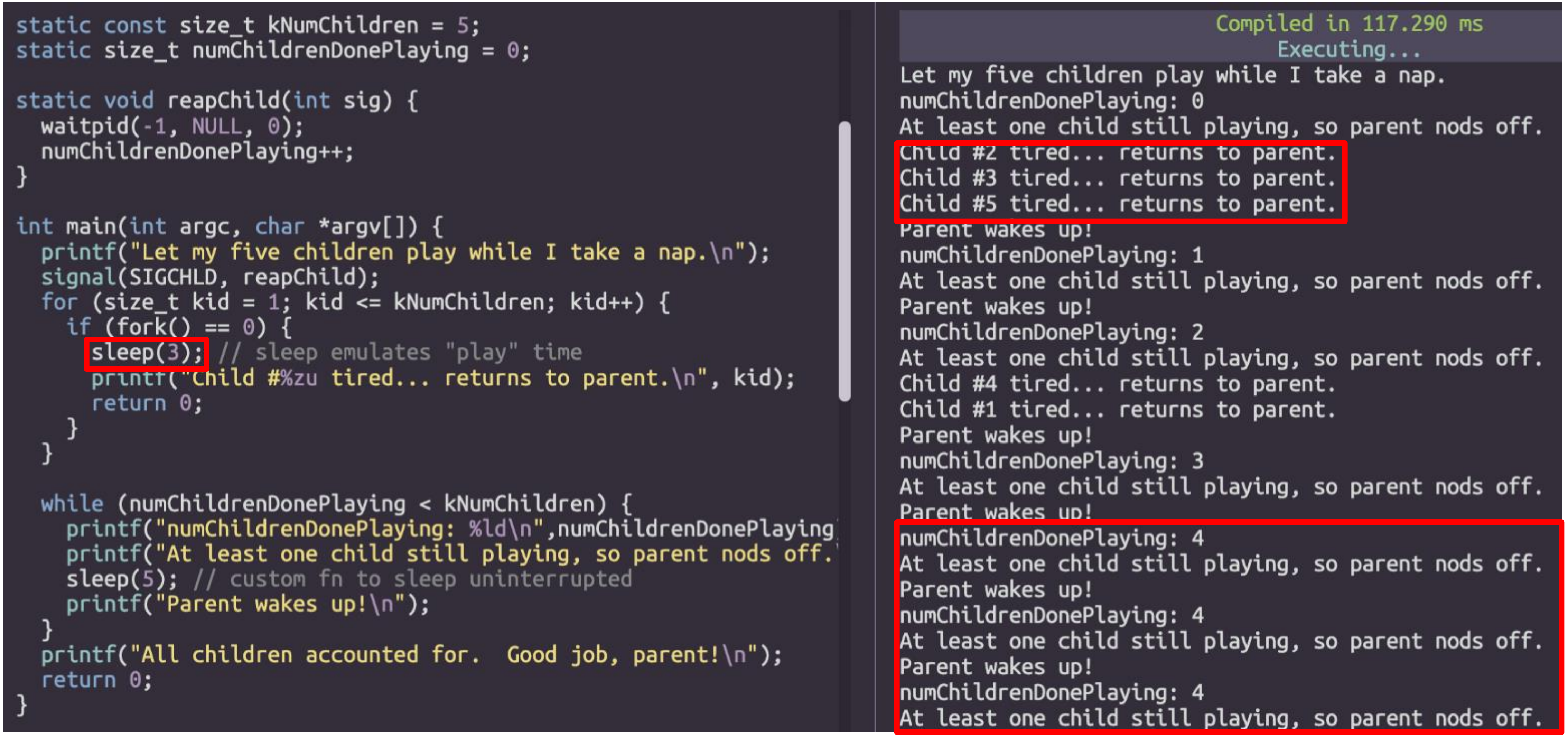


```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  waitpid(-1, NULL, 0);
  numChildrenDonePlaying++;
}

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\n")
  signal(SIGCHLD, reapChild);
  for (size_t kid = 1; kid <= kNumChildren; kid++) {
    if (fork() == 0) {
      sleep(3 * kid); // sleep emulates "play" time
      printf("Child #%zu tired... returns to parent.\n", ki
      return 0;
    }
  }

  while (numChildrenDonePlaying < kNumChildren) {
    printf("numChildrenDonePlaying: %ld\n",numChildrenDoneP
    printf("At least one child still playing, so parent noc
    sleep(5); // custom fn to sleep uninterrupted
    printf("Parent wakes up!\n");
  }
  printf("All children accounted for.  Good job, parent!\n'
  return 0;
}
```



```
                    Executing...
Let my five children play while I take a nap.
numChildrenDonePlaying: 0
At least one child still playing, so parent nods off.
Child #1 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 1
At least one child still playing, so parent nods off.
Child #2 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 2
At least one child still playing, so parent nods off.
Child #3 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 3
At least one child still playing, so parent nods off.
Child #4 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 4
At least one child still playing, so parent nods off.
Child #5 tired... returns to parent.
Parent wakes up!
All children accounted for.  Good job, parent!
                         Execution finished (exit status 0)
                             Executed in 15.005 seconds
```

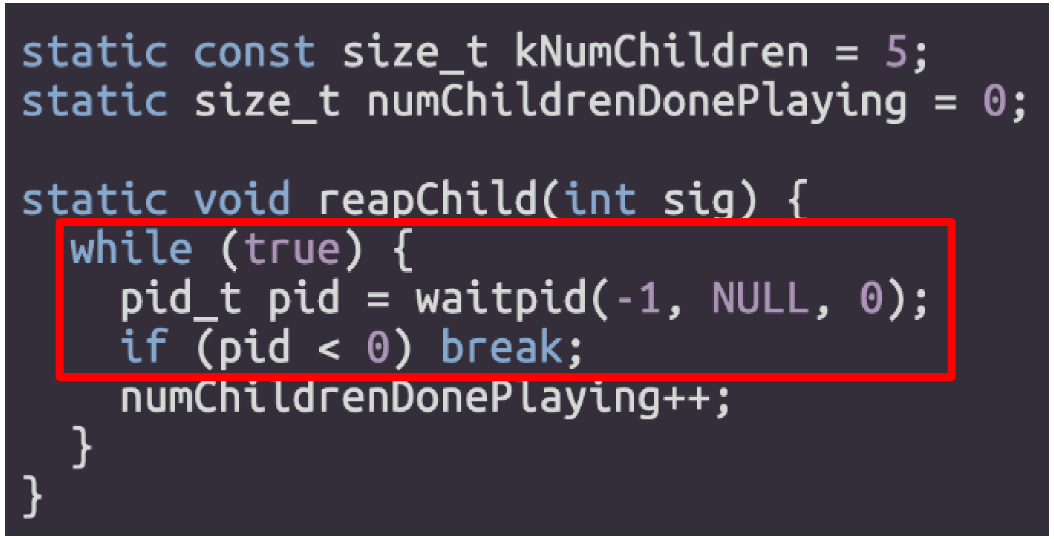
Disney example (cs110)



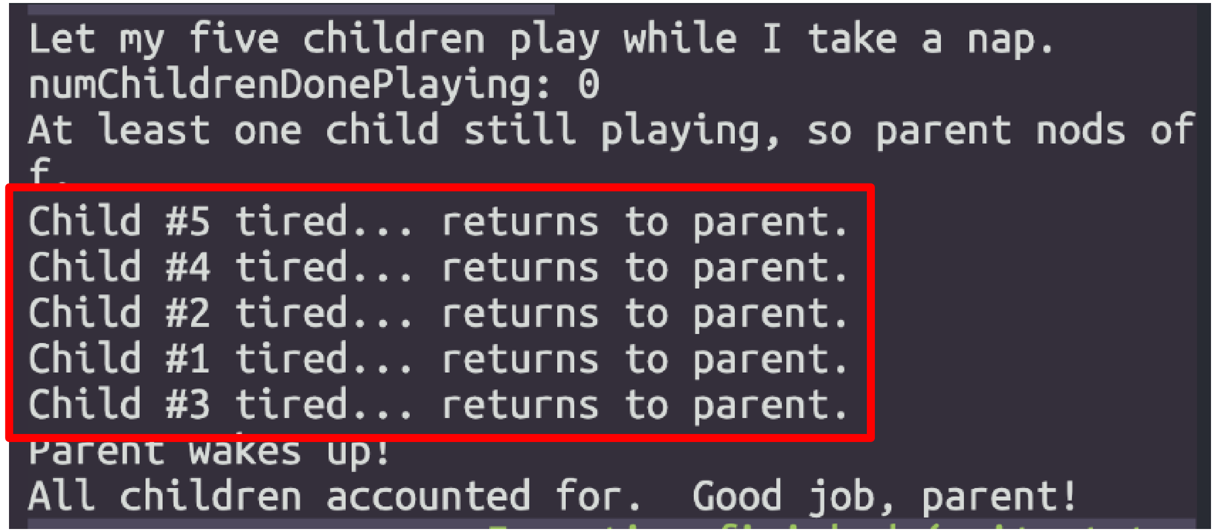
```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  waitpid(-1, NULL, 0);
  numChildrenDonePlaying++;
}

int main(int argc, char *argv[]) {
  printf("Let my five children play while I take a nap.\n");
  signal(SIGCHLD, reapChild);
  for (size_t kid = 1; kid <= kNumChildren; kid++) {
    if (fork() == 0) {
      sleep(3); // sleep emulates "play" time
      printf("Child #%zu tired... returns to parent.\n", kid);
      return 0;
    }
  }

  while (numChildrenDonePlaying < kNumChildren) {
    printf("numChildrenDonePlaying: %ld\n",numChildrenDonePlaying
    printf("At least one child still playing, so parent nods off.
    sleep(5); // custom fn to sleep uninterrupted
    printf("Parent wakes up!\n");
  }
  printf("All children accounted for.  Good job, parent!\n");
  return 0;
}
```

```
                                   Compiled in 117.290 ms
                                        Executing...
Let my five children play while I take a nap.
numChildrenDonePlaying: 0
At least one child still playing, so parent nods off.
Child #2 tired... returns to parent.
Child #3 tired... returns to parent.
Child #5 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 1
At least one child still playing, so parent nods off.
Parent wakes up!
numChildrenDonePlaying: 2
At least one child still playing, so parent nods off.
Child #4 tired... returns to parent.
Child #1 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 3
At least one child still playing, so parent nods off.
Parent wakes up!
numChildrenDonePlaying: 4
At least one child still playing, so parent nods off.
Parent wakes up!
numChildrenDonePlaying: 4
At least one child still playing, so parent nods off.
Parent wakes up!
numChildrenDonePlaying: 4
At least one child still playing, so parent nods off.
```

Disney example (cs110)



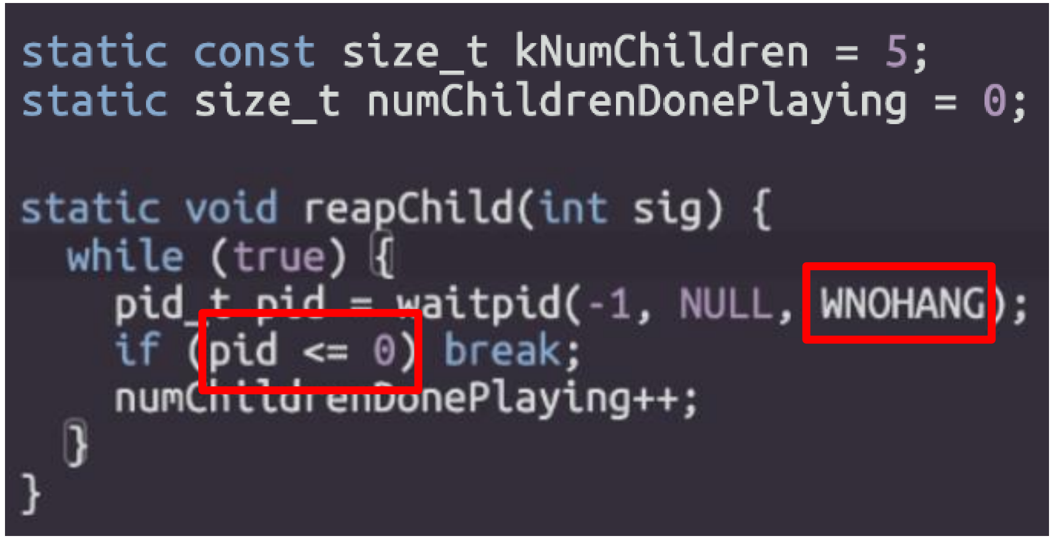
```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, 0);
    if (pid < 0) break;
    numChildrenDonePlaying++;
  }
}
```



```
Let my five children play while I take a nap.
numChildrenDonePlaying: 0
At least one child still playing, so parent nods of
f.
Child #5 tired... returns to parent.
Child #4 tired... returns to parent.
Child #2 tired... returns to parent.
Child #1 tired... returns to parent.
Child #3 tired... returns to parent.
Parent wakes up!
All children accounted for.  Good job, parent!
```

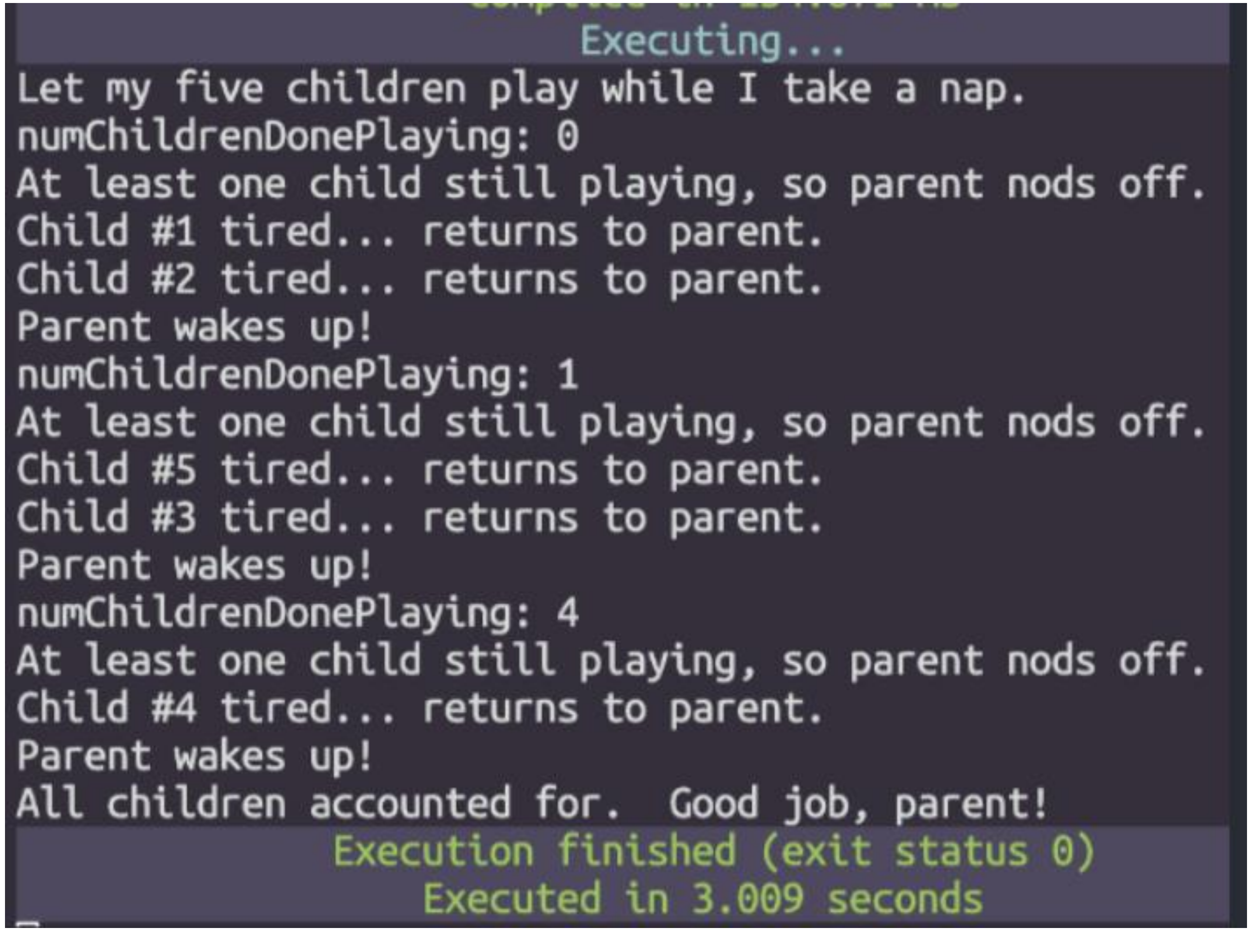
while (waitpid(-1, NULL, 0) >= 0)

Disney example (cs110)



```
static const size_t kNumChildren = 5;
static size_t numChildrenDonePlaying = 0;

static void reapChild(int sig) {
  while (true) {
    pid_t pid = waitpid(-1, NULL, WNOHANG);
    if (pid <= 0) break;
    numChildrenDonePlaying++;
  }
}
```

- while (waitpid(-1, NULL, WNOHANG) > 0)
- WNOHANG: return immediately without blocking if no child processes exit



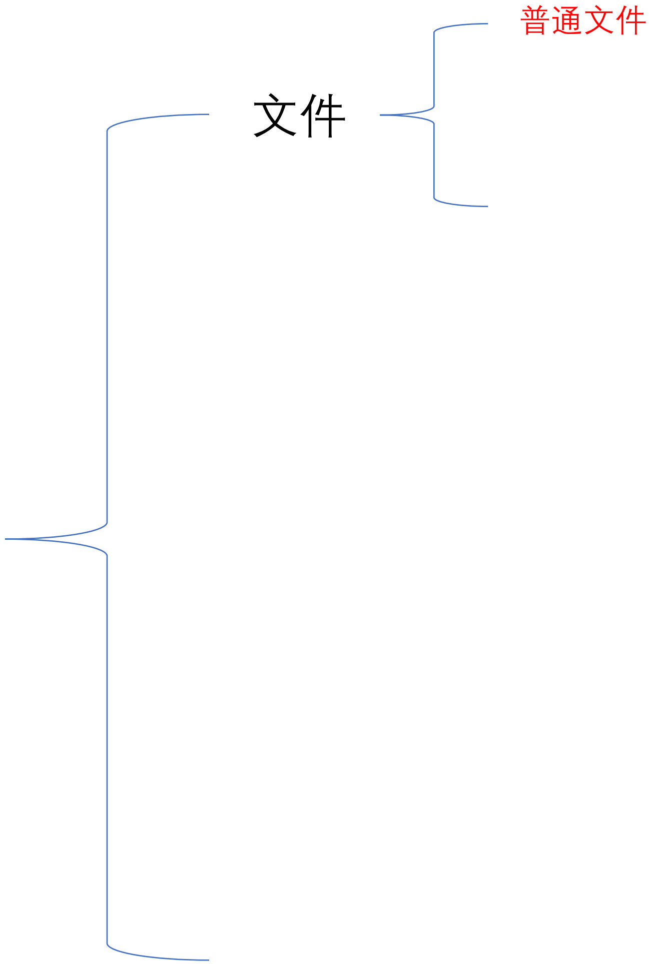
```
                         Executing...
Let my five children play while I take a nap.
numChildrenDonePlaying: 0
At least one child still playing, so parent nods off.
Child #1 tired... returns to parent.
Child #2 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 1
At least one child still playing, so parent nods off.
Child #5 tired... returns to parent.
Child #3 tired... returns to parent.
Parent wakes up!
numChildrenDonePlaying: 4
At least one child still playing, so parent nods off.
Child #4 tired... returns to parent.
Parent wakes up!
All children accounted for.  Good job, parent!
                Execution finished (exit status 0)
                  Executed in 3.009 seconds
```

Disney example (cs110)

- **sigwait** instead of signal handlers
- https://slides.com/troccoli/cs110-win2122-lecture-12/fullscreen#/

System Level I/O
(CS:APP Ch. 10.1-10.11)

王善上

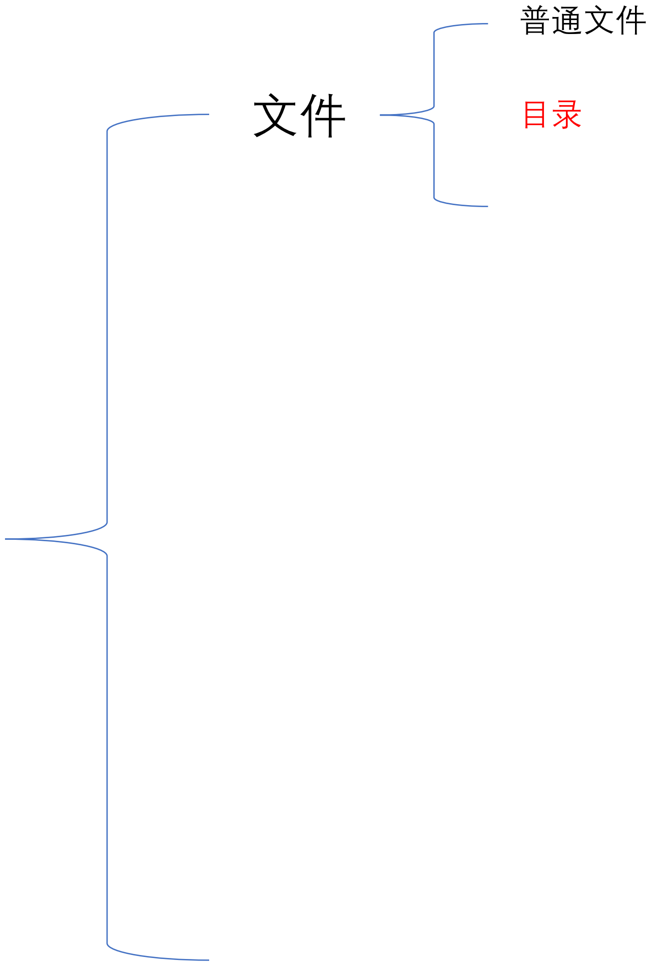


文件

普通文件

普通文件Regular Files

- 文本文件：
  - 只含ascii或unicode编码
  - 每个文件是一个文本行的序列，每个文本行是个字符的序列
  - 每一行以‘\n’(0xa)结尾（补充：windows以‘\r’ ‘\n’结尾）
- 二进制文件：
  - Else
  - 例子：object files, jpeg images…
- Linux内核不区分文本文件和二进制文件



文件

普通文件

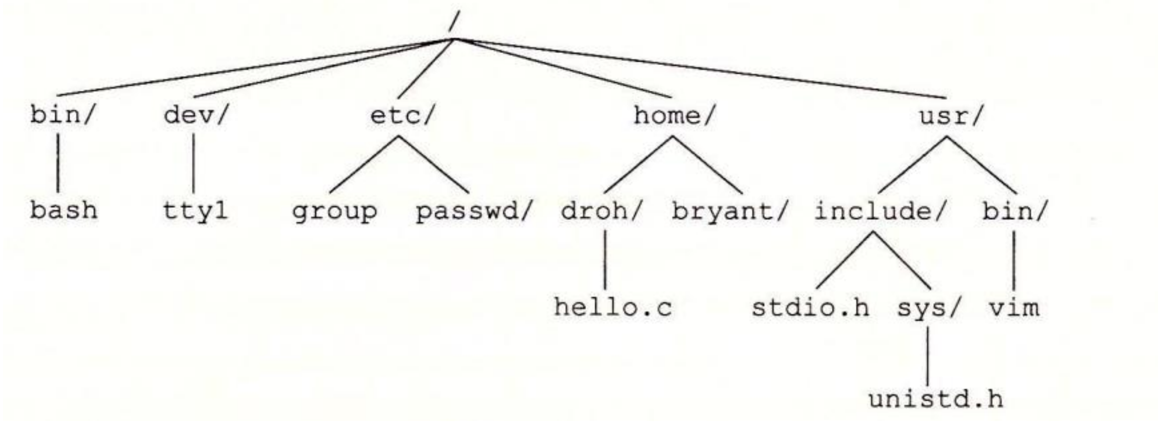
目录

目录directory

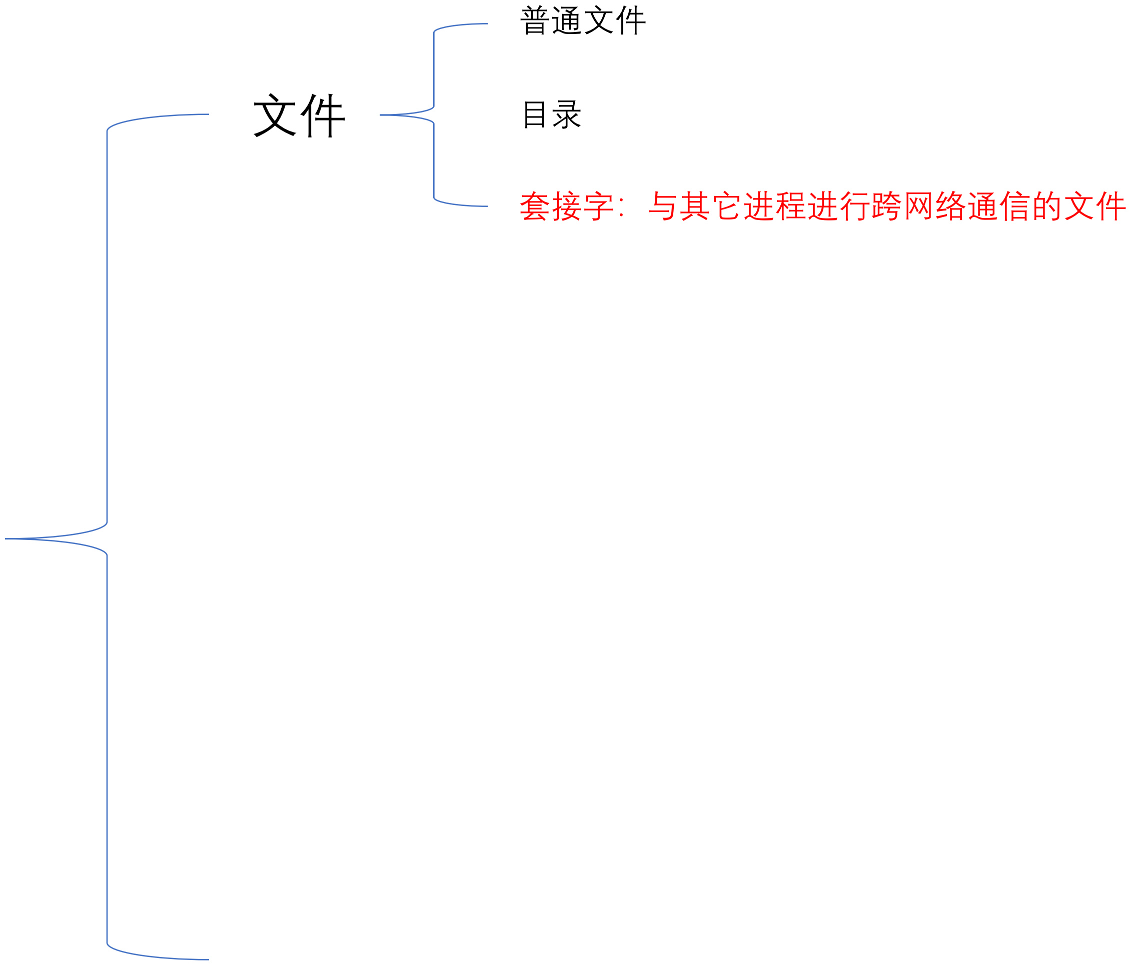
- 包含一组链接的文件
  - 链接：把一个文件名映射到一个文件
- 至少包含两个链接：
  - .：指向自己
  - ..：指向父目录

- 指令
  - mkdir(make directory)
  - ls(list)
  - rmdir(remove directory)
    - 补充：删除非空目录（并递归删除其指向的所有文件）：rm -r

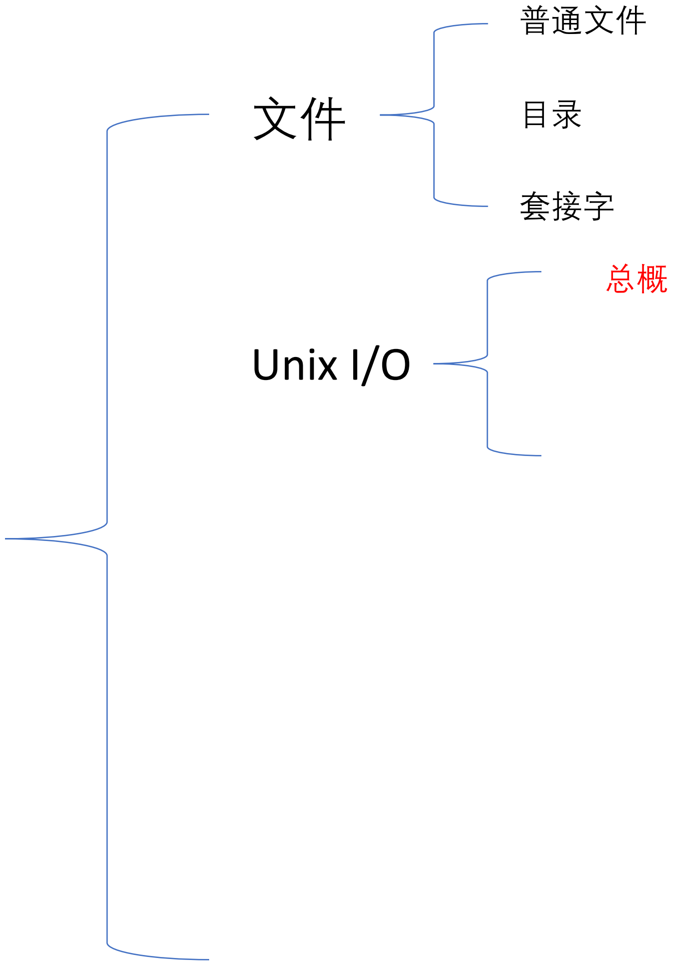
目录层次结构



- 路径名：
  - 绝对路径名：从根节点往下走
  - 相对路径名：从当前节点开始走



- 文件
  - 普通文件
  - 目录
  - 套接字：与其它进程进行跨网络通信的文件



文件

普通文件

目录

套接字

Unix I/O

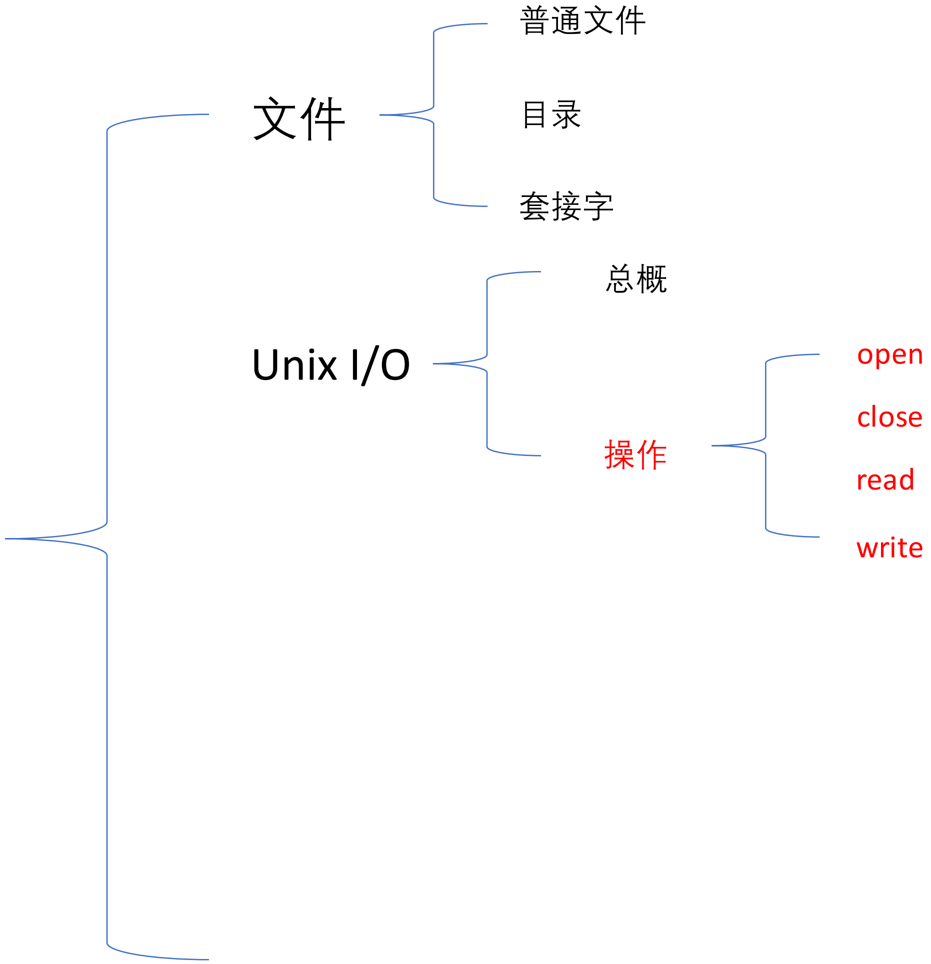
总概

总概

- 所有I/O设备->文件
- 所有输入输出->文件读写

- 大多数文件含有的属性：文件位置k（计数器，记录读写到哪了，从零开始，可用seek改变k值）
    - shell,网络套接字没有文件位置

- 操作：open,read,write,close,…



- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write

Open

- int open(char *filename, int flags, mode_t mode);
- 人话：open(路径名，打开模式，权限)

- 打开模式：O_RDONLY , O_WRONLY , O_RDWR , O_CREAT , O_TRUNC , O_APPEND
    - ps: truncated：截断（人话：覆盖）
    - pps:实际为2的幂，可用位运算组合

- 权限：默认为0，当open创建文件时会用到，给文件权限赋为 mode & ~umask（函数umask(x)）

Open返回值

- 正常情况：
- 返回一个正数：文件描述符
    - 给打开的文件一个代号，之后对该文件操作会用到
    - 每个运行的进程都有三个特定的文件描述符：0:stdin,1:stdout,2:strerr

- 非正常情况：-1
- 如文件不存在，没有访问权限等

Close

- int close(int fd)；
- fd:file descripter，文件描述符

- 返回值：
  - 成功为0
  - 失败为-1（关掉已经关掉的文件）

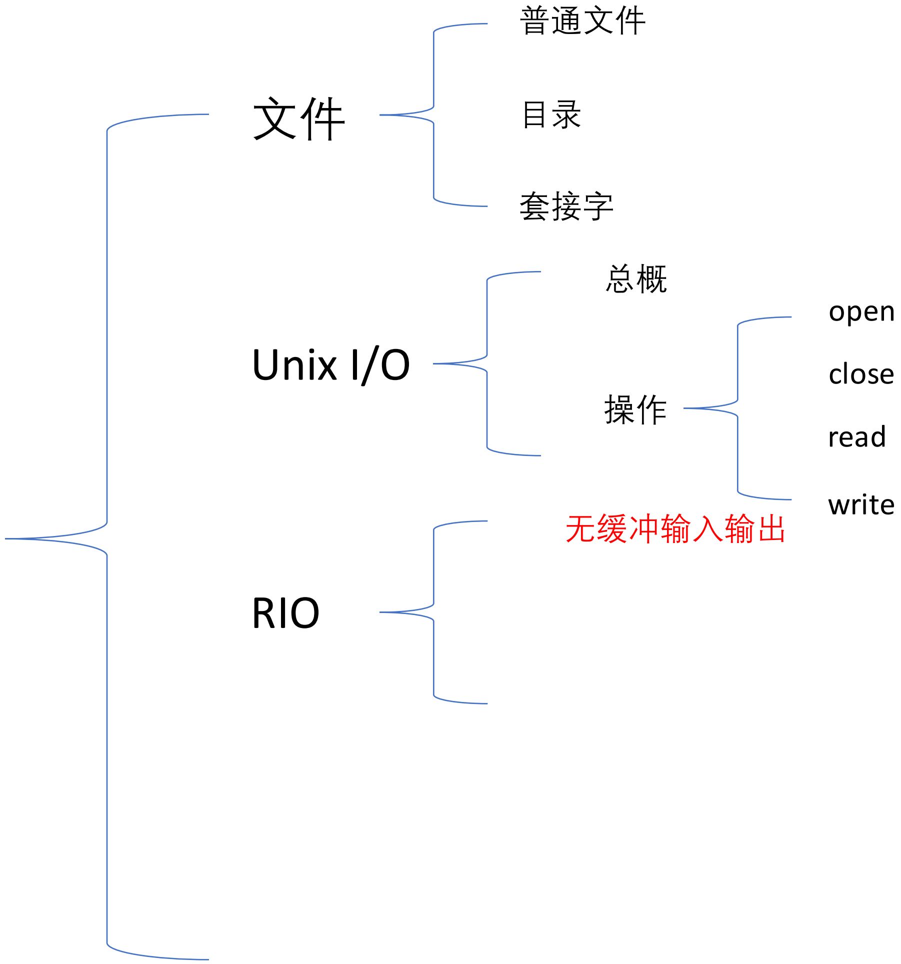
Read/Write

- ssize_t read(int fd, void *buf, size_t n)
- 人话：read(fd，读到哪，最多读多少)
  - ps: ssize_t & size_t

- 返回值：
  - 正：成功读写字节数
  - 0：EOF
  - 负：出错

Short count

- 一个例子：100B的文件，循环读入，每次读70B，每次返回多少？
- 还可能出现于：终端读写，网络读写（socket内部缓冲约束，较长的网络延迟）
- 需要反复调用read, write来保证真正读写完了
- 于是有了RIO包



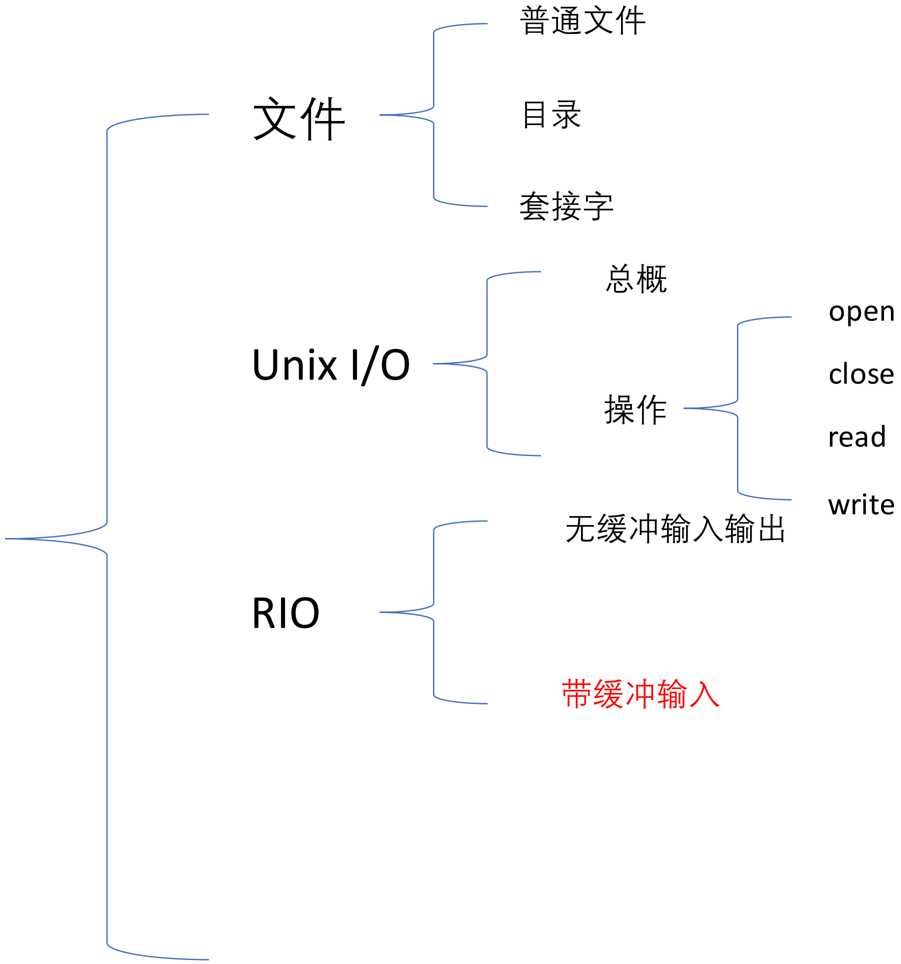
- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
  - 无缓冲输入输出
- RIO

无缓冲输入输出

- 封装，允许被中断的系统调用，可处理short count

- ssize_t rio_readn(int fd, void *buf, size_t n);
- ssize_t rio_writen(int fd, void *buf, size_t n);

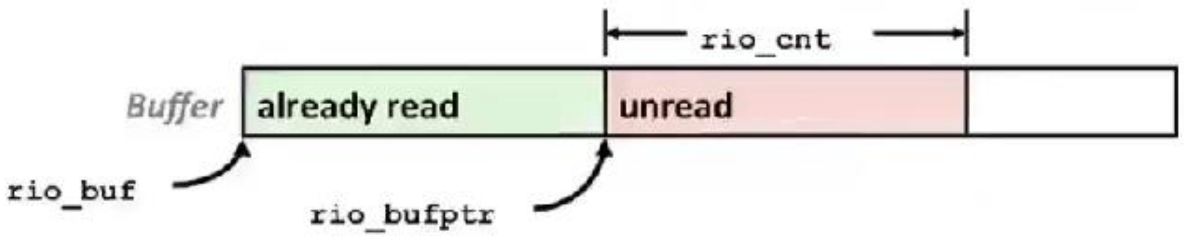
- 用法与read,write相同

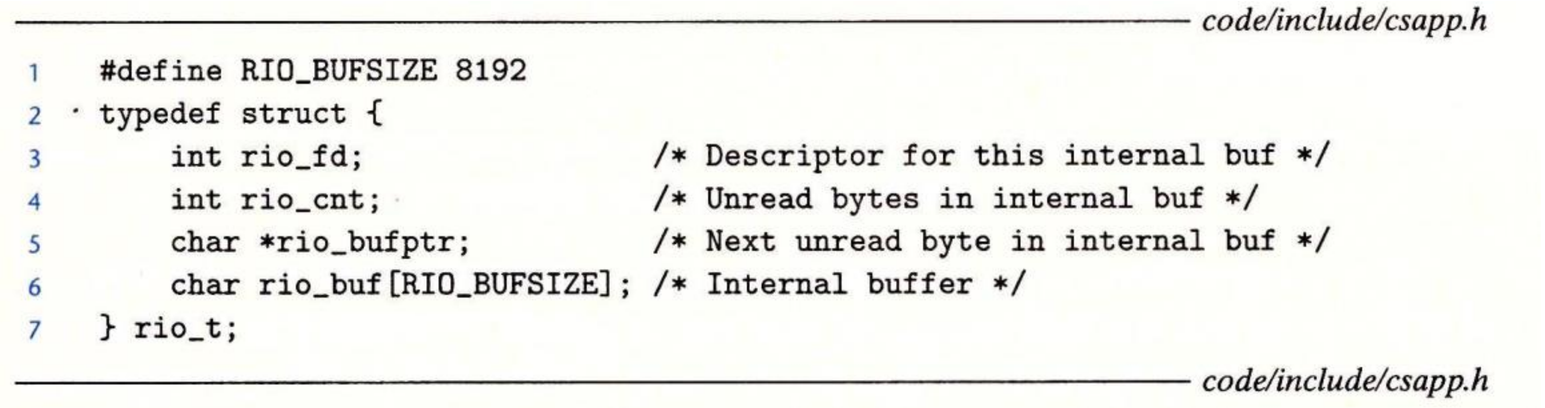


- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
- RIO
  - 无缓冲输入输出
  - 带缓冲输入

带缓冲输入

- 举一个带有四川特色的例子





code/include/csapp.h

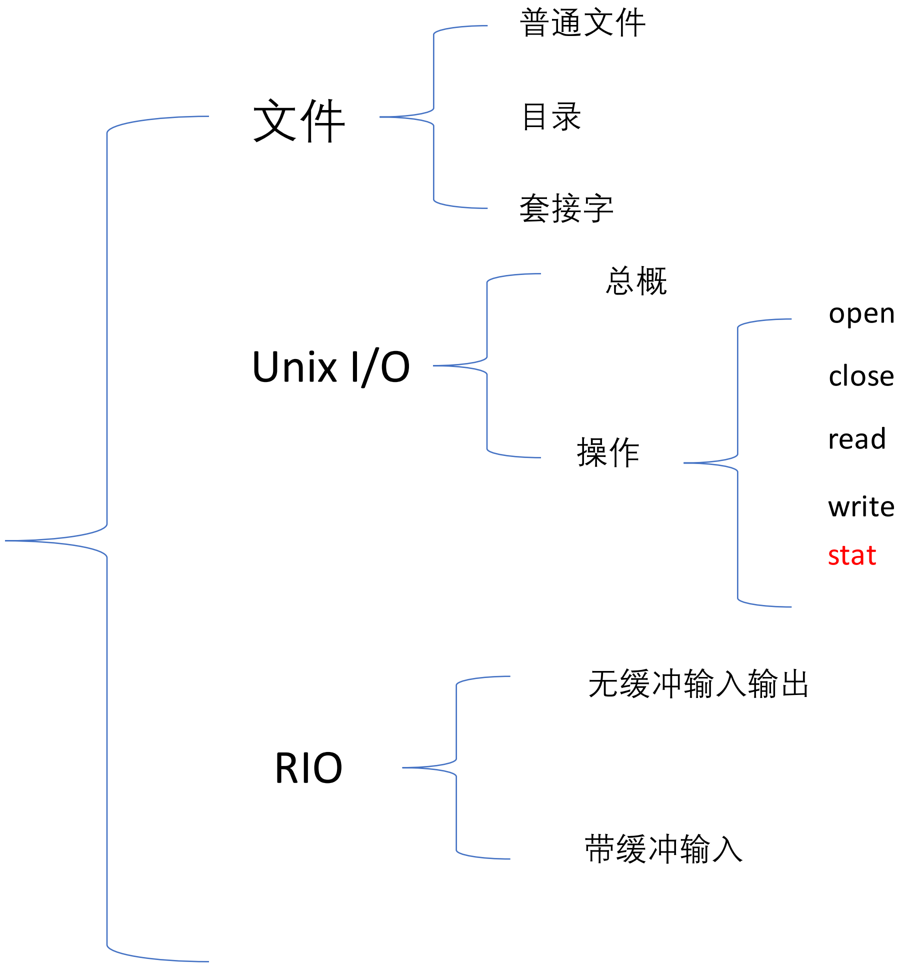
```c
#define RIO_BUFSIZE 8192
typedef struct {
    int rio_fd;                /* Descriptor for this internal buf */
    int rio_cnt;               /* Unread bytes in internal buf */
    char *rio_bufptr;          /* Next unread byte in internal buf */
    char rio_buf[RIO_BUFSIZE]; /* Internal buffer */
} rio_t;
```

code/include/csapp.h

带缓冲输入

- 初始化：
- void rio_readinitb(rio_t *rp, int fd)

- 读入：
- ssize_t rio_readnb(rio_t *rp, void *buf, size_t n)
  - 用法同read
- ssize_t rio_readlineb(rio_t *rp, void *buf, size_t n)
  - 读入一个文本行（类似字符串，会以NULL结尾，因此最多只能读n-1）
- 带缓冲的两个可以交叉使用，但带缓冲读入和无缓冲读入不能



文件

普通文件

目录

套接字

Unix I/O

总概

操作

open

close

read

write

stat

RIO

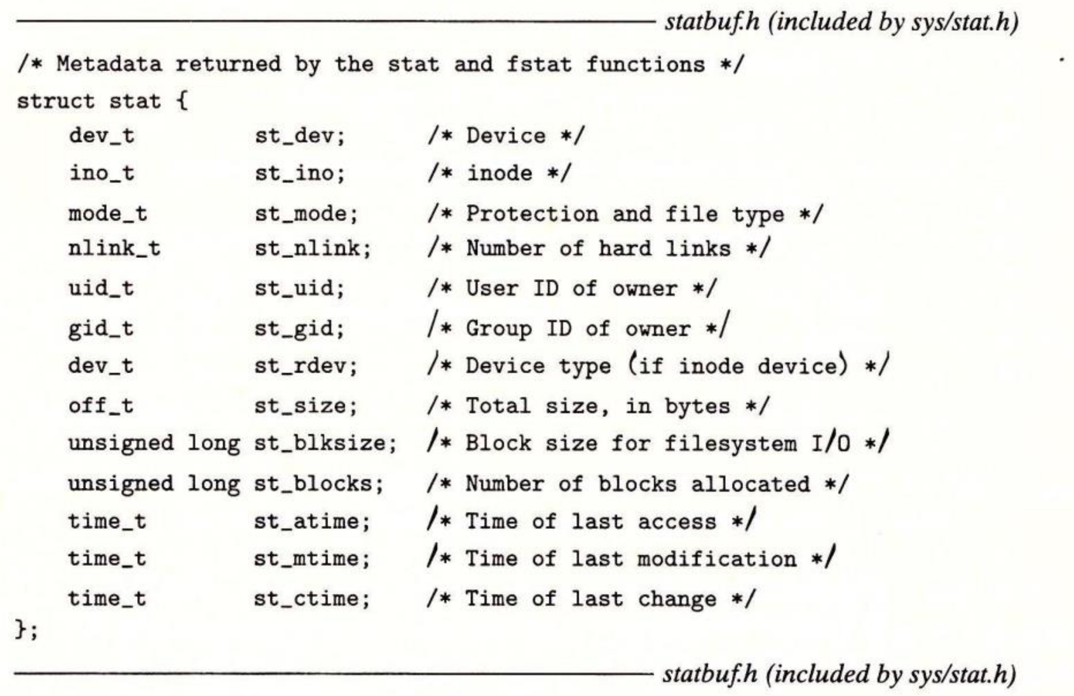
无缓冲输入输出

带缓冲输入

读取文件元数据

- 元数据：

- int stat(char *filename, stat *buf)
- int fstat(int fd, stat *buf)
- 人话：stat(路径名，用于存信息的stat的指针)



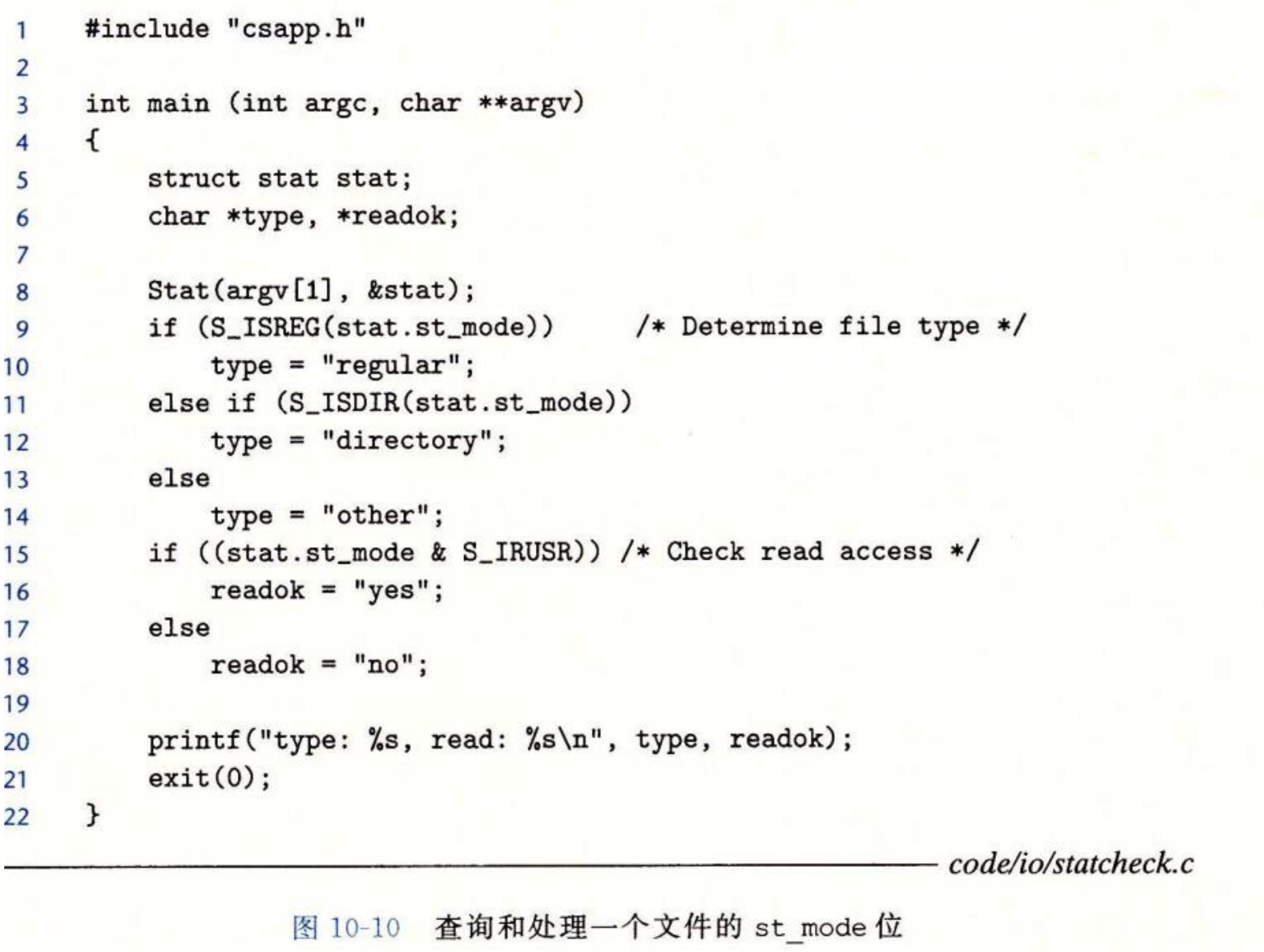
statbuf.h (included by sys/stat.h)

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* Device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* Protection and file type */
    nlink_t       st_nlink;    /* Number of hard links */
    uid_t         st_uid;      /* User ID of owner */
    gid_t         st_gid;      /* Group ID of owner */
    dev_t         st_rdev;     /* Device type (if inode device) */
    off_t         st_size;     /* Total size, in bytes */
    unsigned long st_blksize;  /* Block size for filesystem I/O */
    unsigned long st_blocks;   /* Number of blocks allocated */
    time_t        st_atime;    /* Time of last access */
    time_t        st_mtime;    /* Time of last modification */
    time_t        st_ctime;    /* Time of last change */
};
```

statbuf.h (included by sys/stat.h)

图 10-9 stat 数据结构

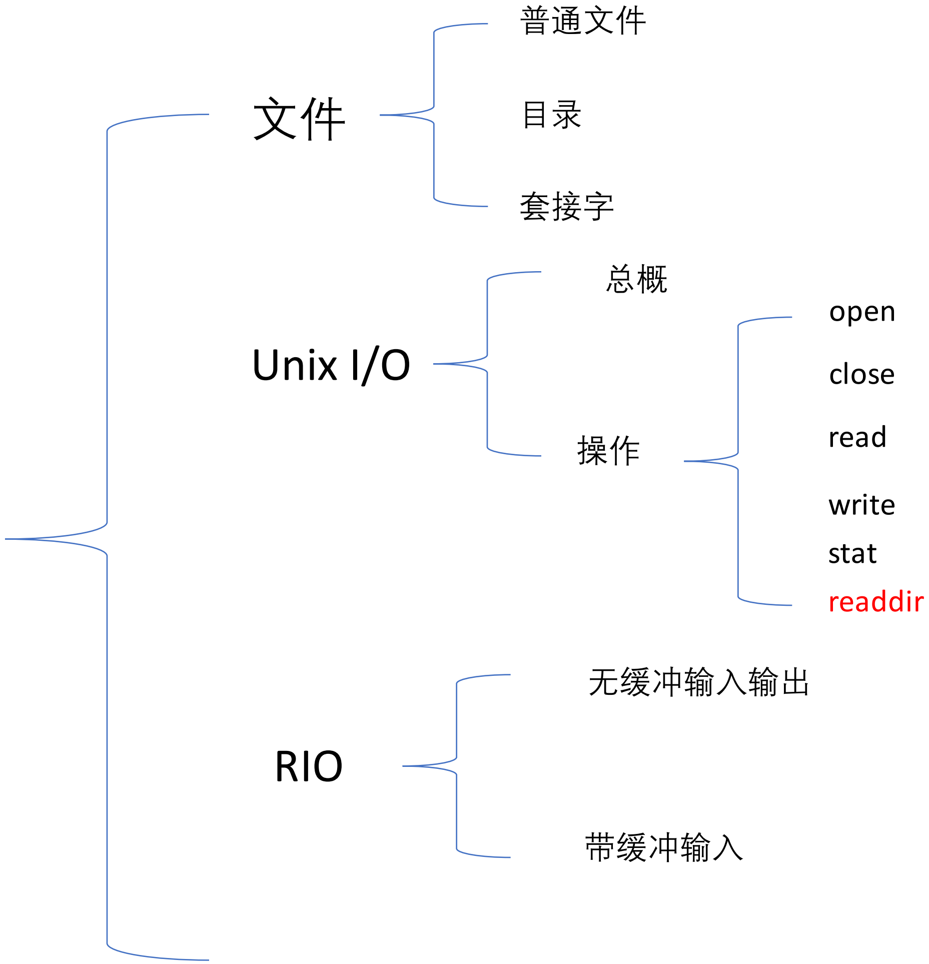
读取文件元数据



```c
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;

    Stat(argv[1], &stat);
    if (S_ISREG(stat.st_mode))     /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else
        type = "other";
    if ((stat.st_mode & S_IRUSR)) /* Check read access */
        readok = "yes";
    else
        readok = "no";

    printf("type: %s, read: %s\n", type, readok);
    exit(0);
}
```

code/io/statcheck.c

图 10-10　查询和处理一个文件的 st_mode 位



- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
    - stat
    - readdir
- RIO
  - 无缓冲输入输出
  - 带缓冲输入

读取目录内容

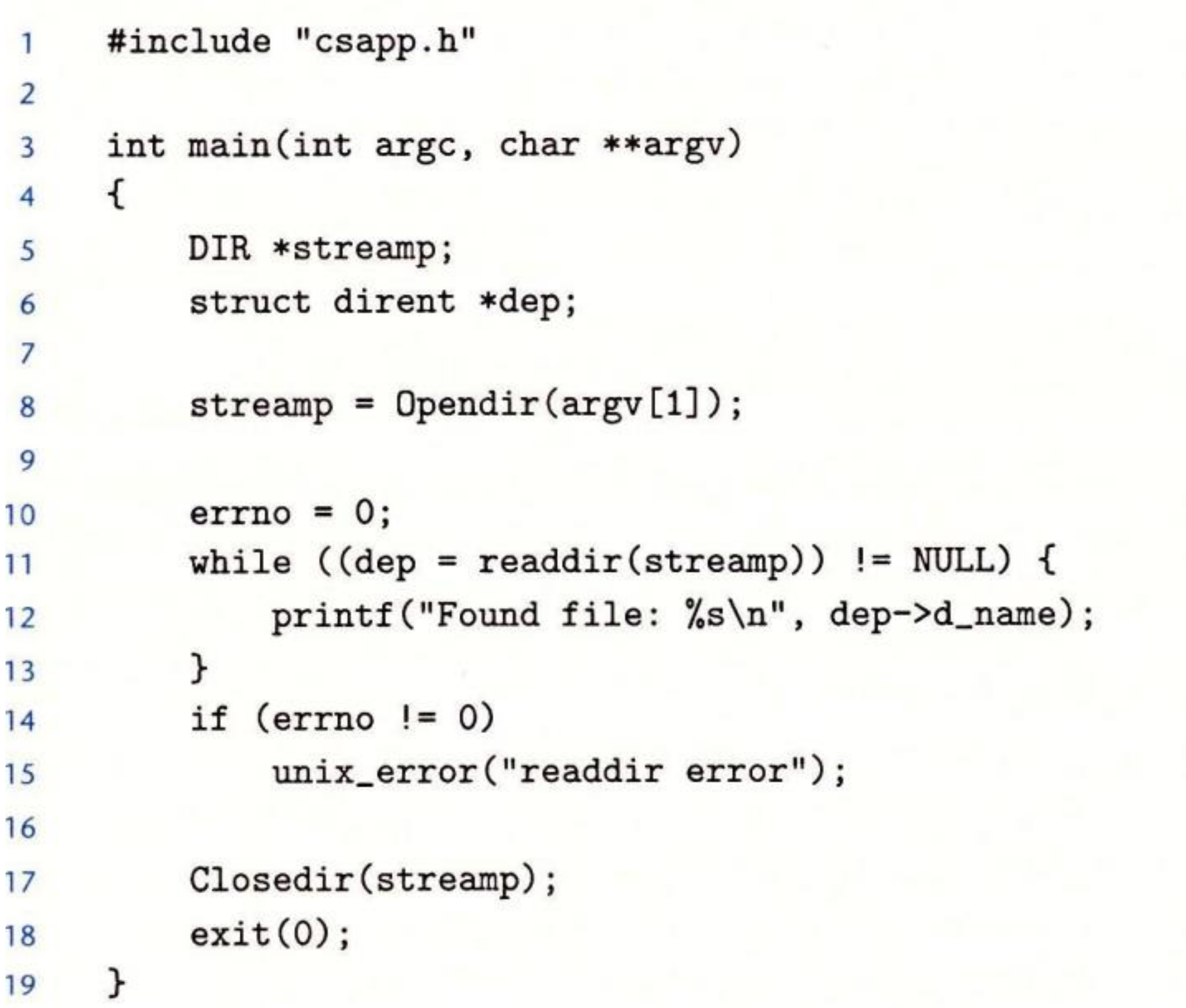
- DIR *opendir(路径名)
- 返回指向目录流的指针，出错则为NULL

- dirent *readdir(DIR *dirp)
- 返回指向下一个目录项的指针，若无更多目录项或出错则为NULL

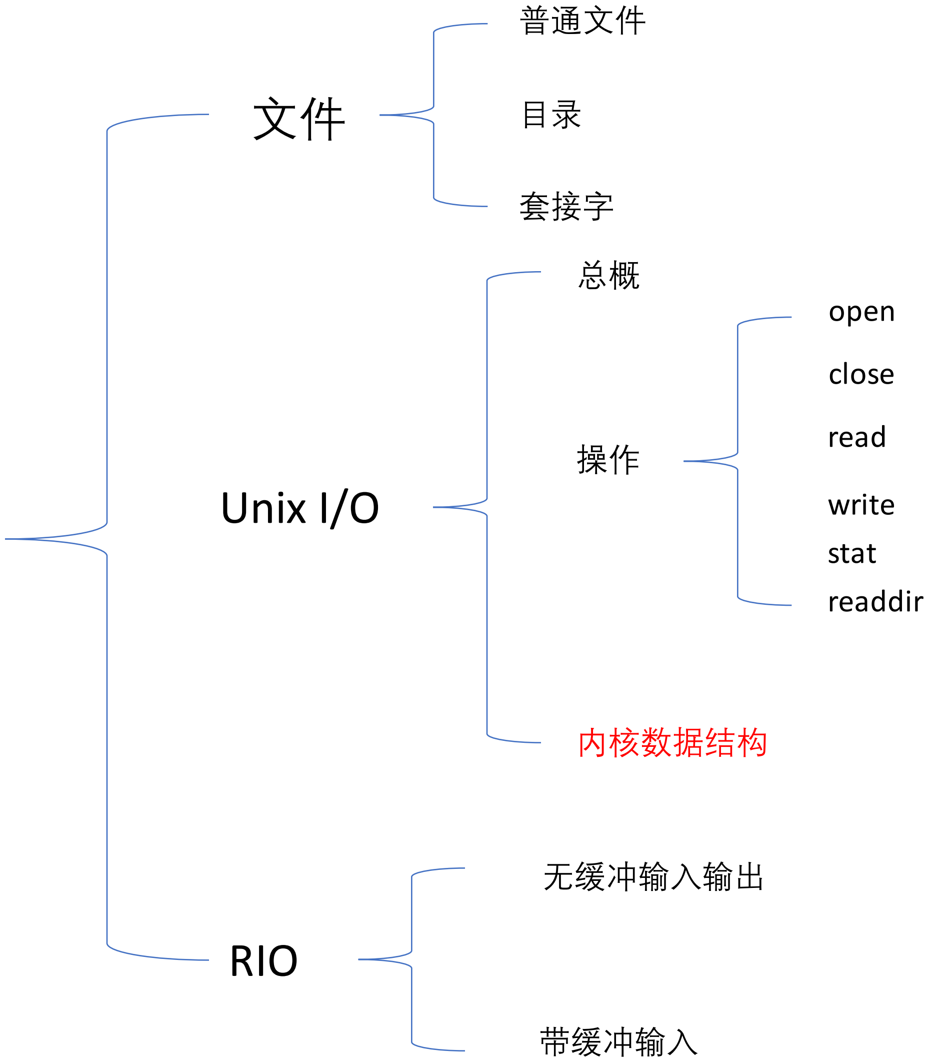
- struct dirent{ino_t d_ino;char d_name[256];};
- 分别为文件位置，文件名

- int closedir(DIR *dirp)
- 关闭该目录流并释放所有资源

读取目录内容

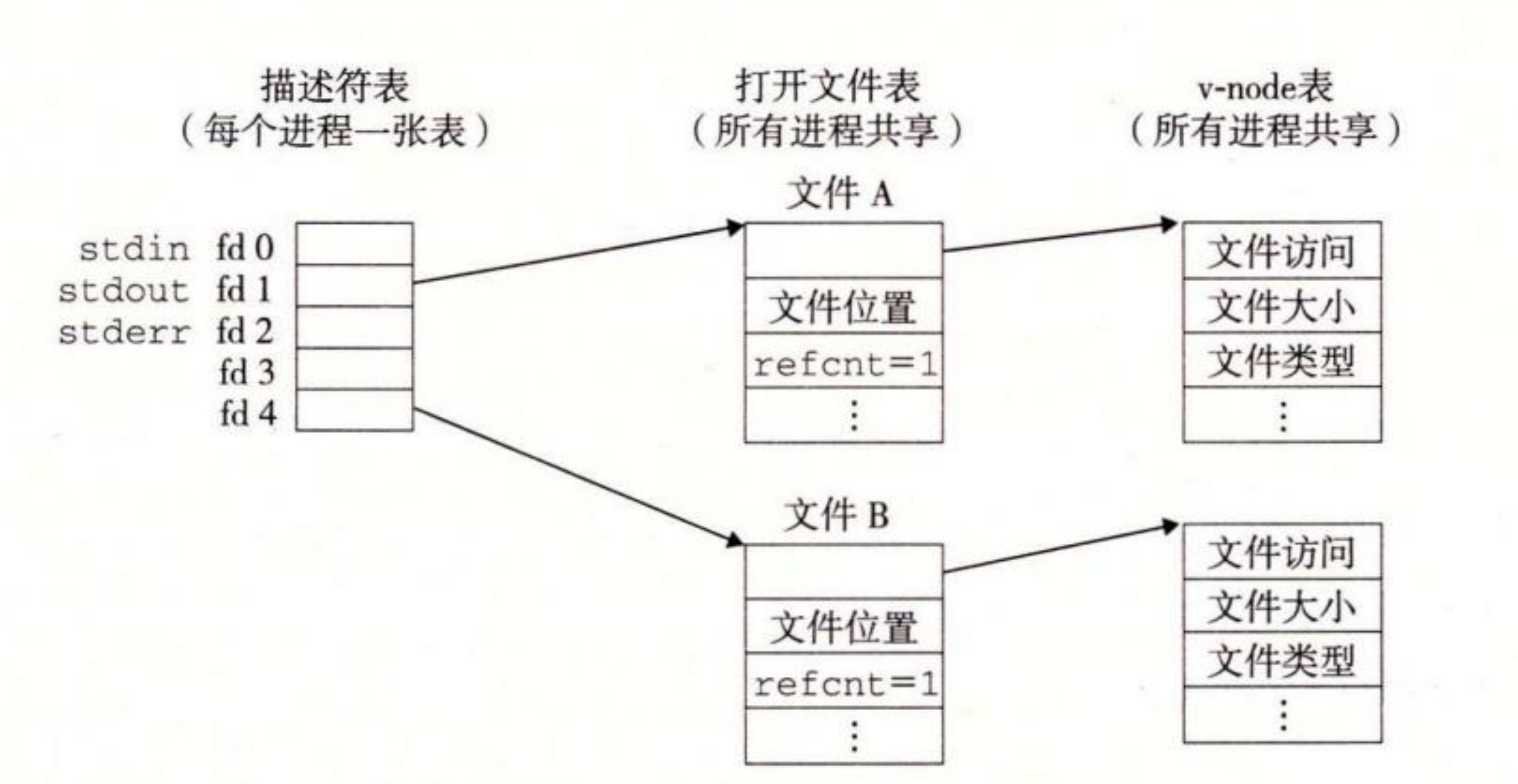


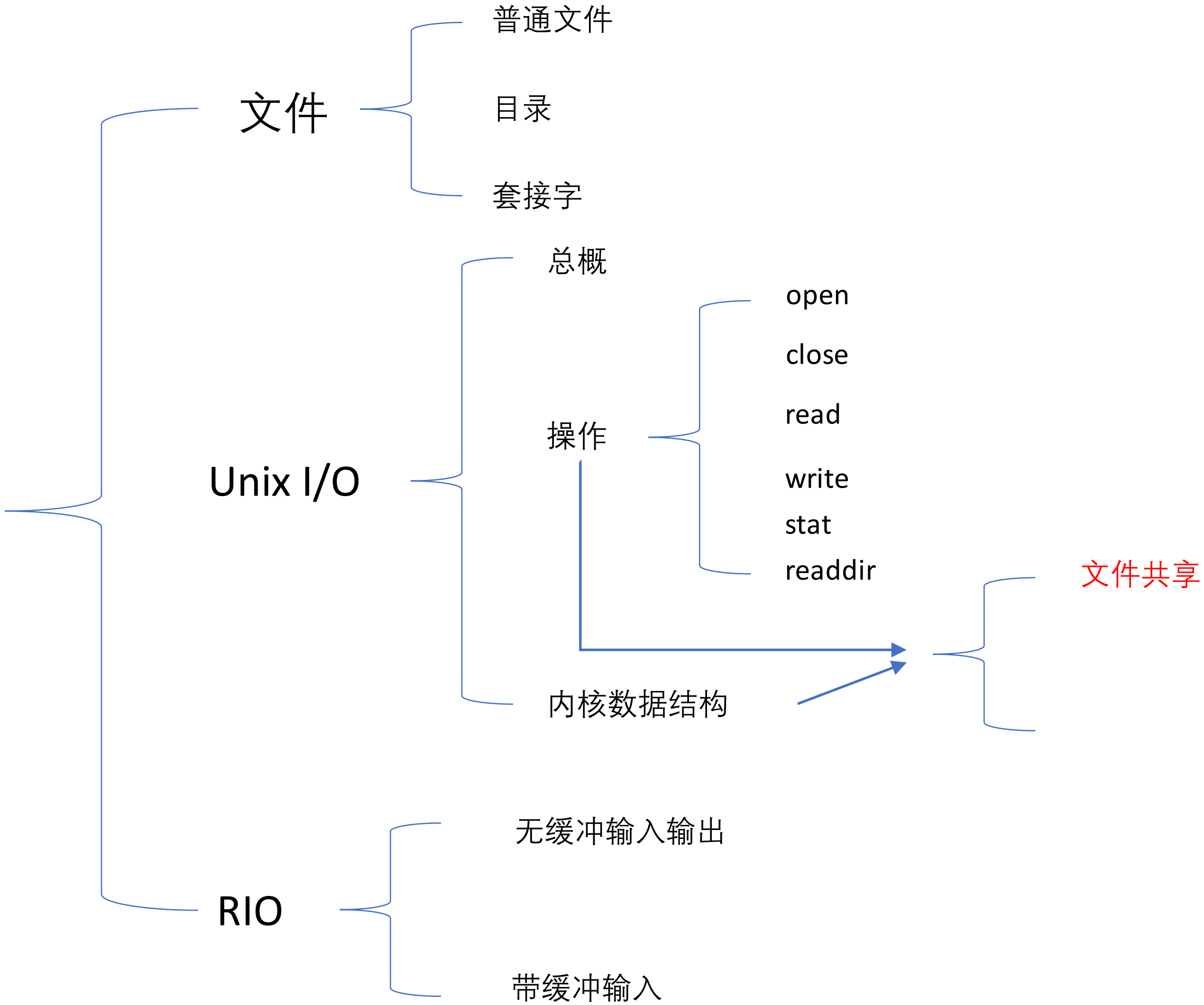
```c
#include "csapp.h"

int main(int argc, char **argv)
{
    DIR *streamp;
    struct dirent *dep;

    streamp = Opendir(argv[1]);

    errno = 0;
    while ((dep = readdir(streamp)) != NULL) {
        printf("Found file: %s\n", dep->d_name);
    }
    if (errno != 0)
        unix_error("readdir error");

    Closedir(streamp);
    exit(0);
}
```



- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
    - stat
    - readdir
  - 内核数据结构
- RIO
  - 无缓冲输入输出
  - 带缓冲输入

打开文件的内核数据结构





- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
    - stat
    - readdir
  - 内核数据结构
- 文件共享
- RIO
  - 无缓冲输入输出
  - 带缓冲输入

文件共享

- 可以打开多次同一个文件，实现从多个不同文件位置进行读取

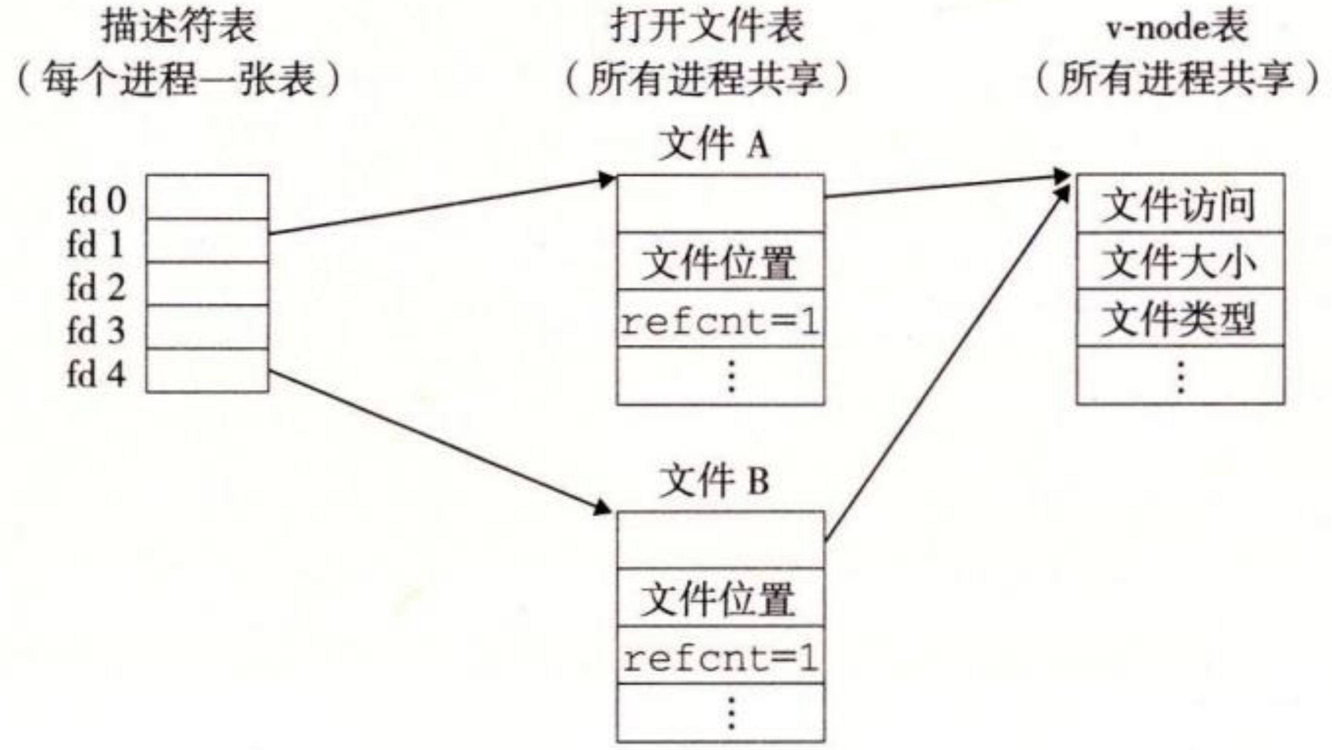
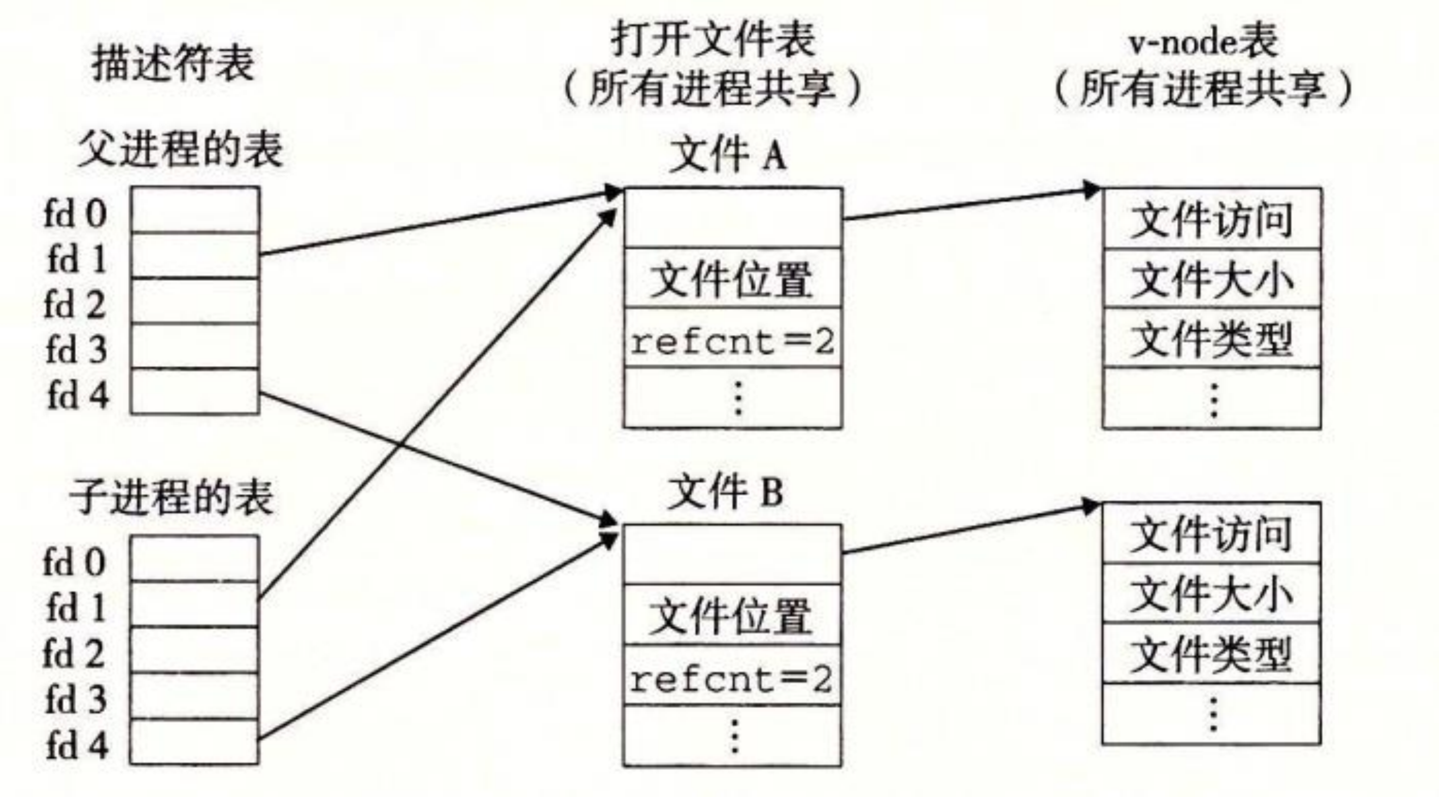
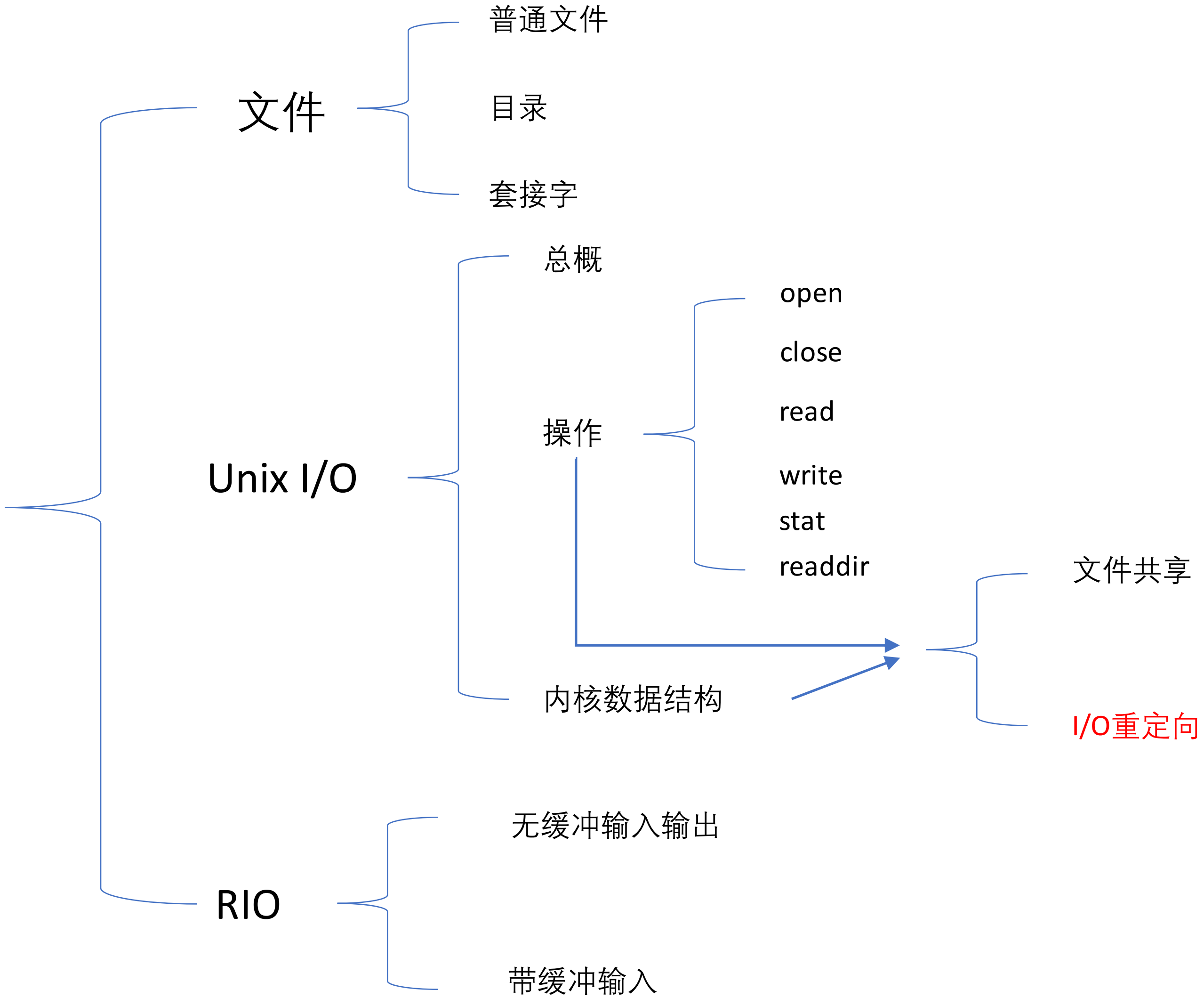


图 10-13　文件共享。这个例子展示了两个描述符通过两个打开文件表表项共享同一个磁盘文件

文件共享

- fork，子进程的描述符表为父进程的复制，所以与父进程指向相同的打开文件表，实现文件共享





- 文件
  - 普通文件
  - 目录
  - 套接字
- Unix I/O
  - 总概
  - 操作
    - open
    - close
    - read
    - write
    - stat
    - readdir
  - 内核数据结构
- 文件共享
- I/O重定向
- RIO
  - 无缓冲输入输出
  - 带缓冲输入

I/O重定向

- int dup2(int oldfd,int newfd);返回值：newfd

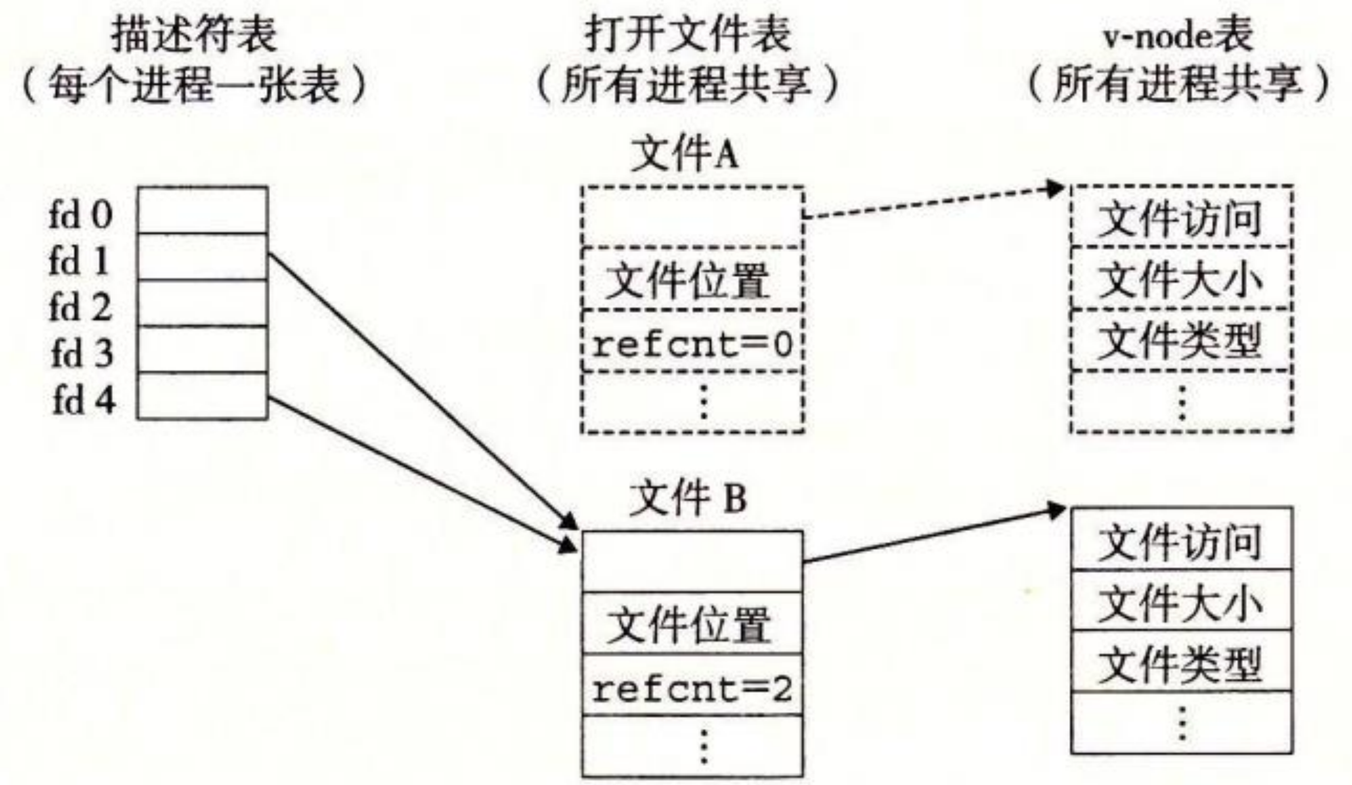
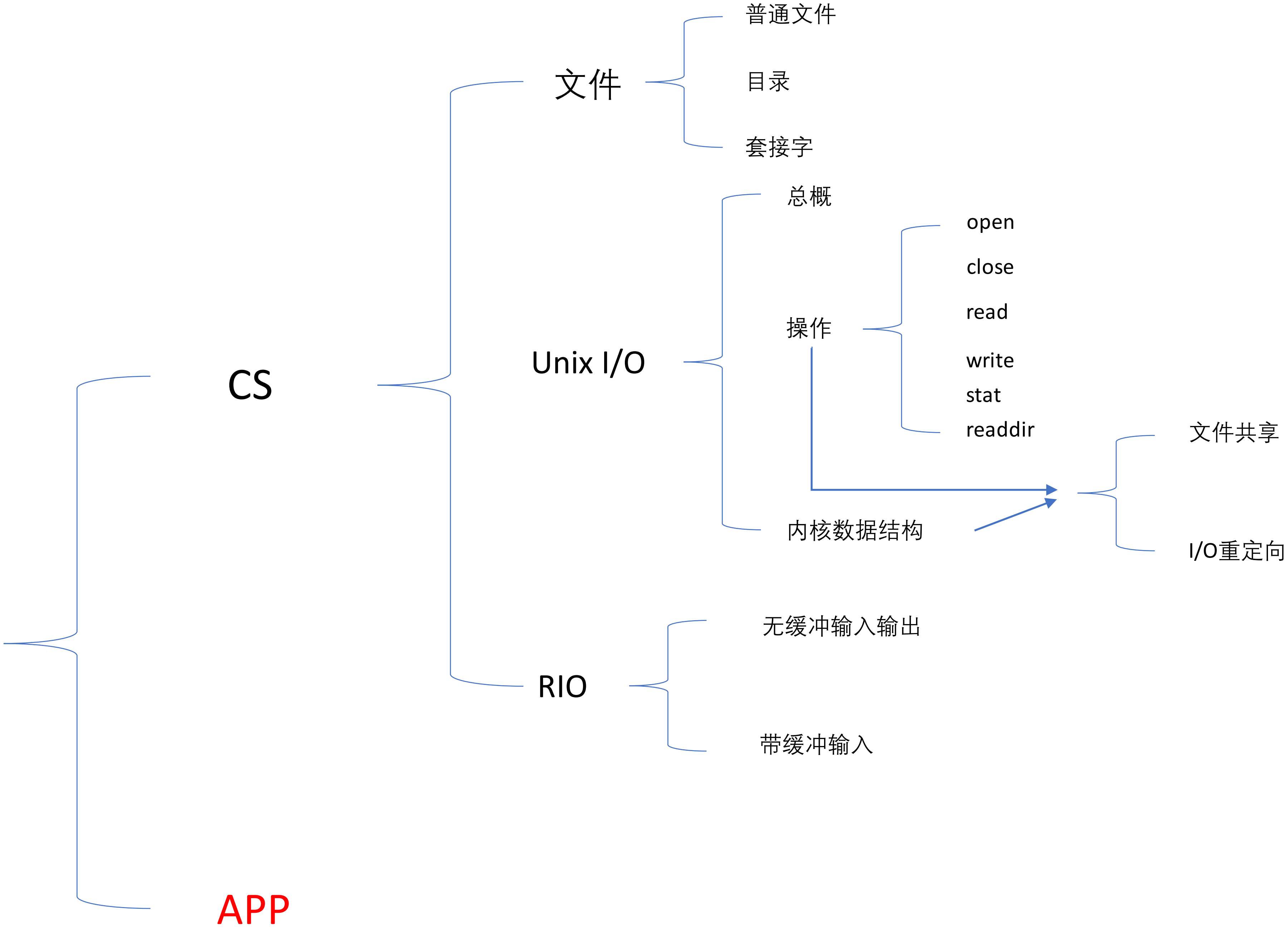


图 10-15　通过调用 dup2(4,1)重定向标准输出之后的内核数据结构。初始状态如图 10-12 所示

I/O重定向（补充）

- int dup(int fd);
- 返回当前可用的最小描述符fdx，并使fdx指向fd指向的文件

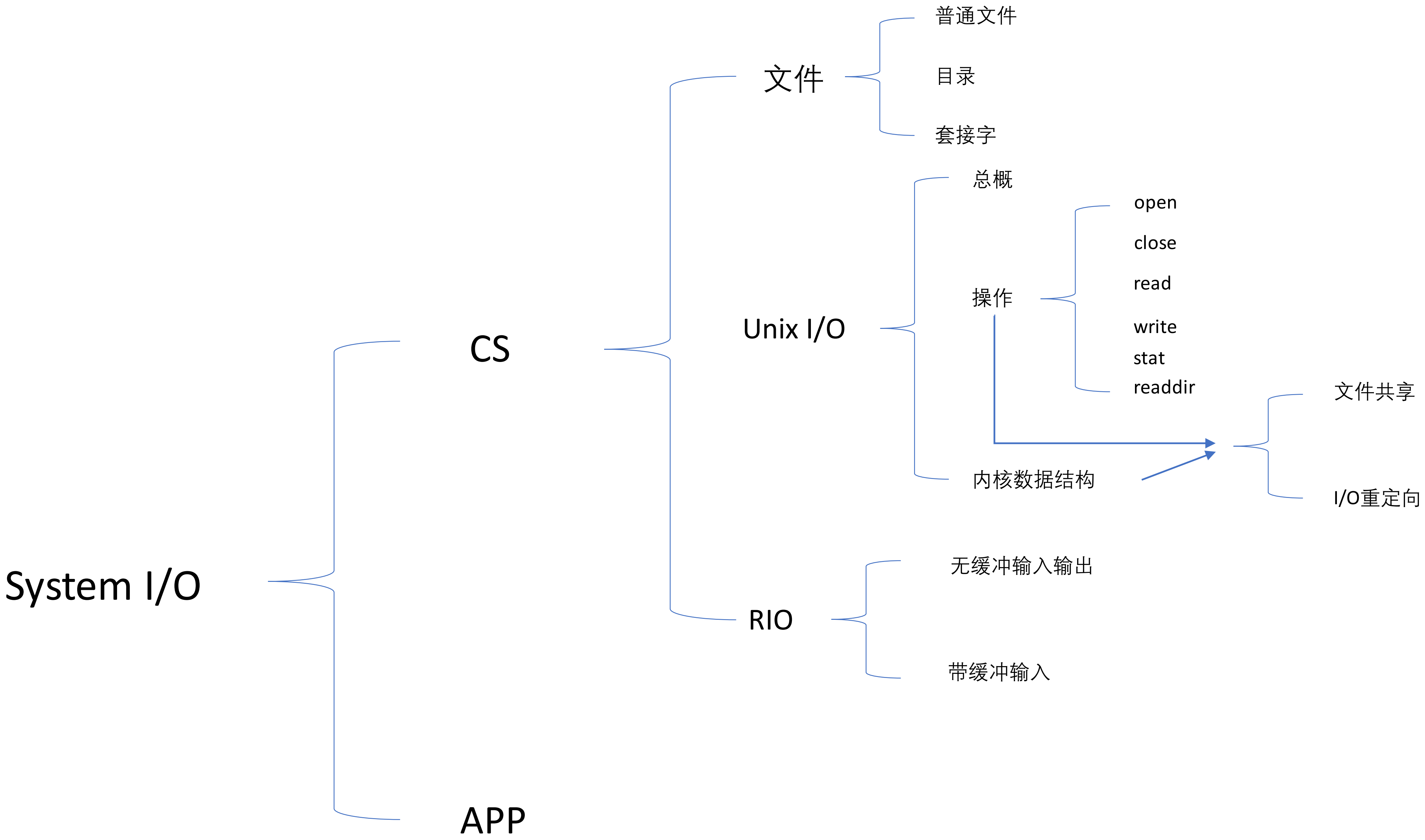


- CS
  - 文件
    - 普通文件
    - 目录
    - 套接字
  - Unix I/O
    - 总概
    - 操作
      - open
      - close
      - read
      - write
      - stat
      - readdir
    - 内核数据结构
      - 文件共享
      - I/O重定向
  - RIO
    - 无缓冲输入输出
    - 带缓冲输入
- APP

APP

- Unix I/O：最底层最基础
- RIO：将Unix I/O加以封装，并解决了中断、不足值问题，并加入缓冲区优化效率。
- 标准IO：提供更完整的带缓冲的替代品

- G1:如果可以的话，标准IO是首选
- G2:不要使用scanf和rio_readlineb来读二进制文件（0xa）
- G3:对网络套接字使用RIO函数（存在标准IO与网络文件不相兼容的限制）
- (G4:这章的IO内核数据结构和signal结合出题，加入race的因素，可以出的很花，面向考试学习的话这章的IO内核数据结构是重点）



System I/O

CS

文件

普通文件

目录

套接字

Unix I/O

总概

操作

open

close

read

write

stat

readdir

内核数据结构

文件共享

I/O重定向

RIO

无缓冲输入输出

带缓冲输入

APP

Practice

王善上

The End

Rings & modes

许珈铭

Processes & threads & coroutines

许珈铭