CPU虚拟化（~~~~~）

2022年9月16日 9:50

共享资源 { 时分共享 （CPU）
空分共享 （内存）

通过让一个进程运行一段时间，然后切换到其它进程。

代价：性能损失（上下文切换）

如何实现CPU的时分共享？

底层机制 ⟷ 如何进行上下文切换。（✓）
+
上层策略 ⟷ 调度策略 （✗）

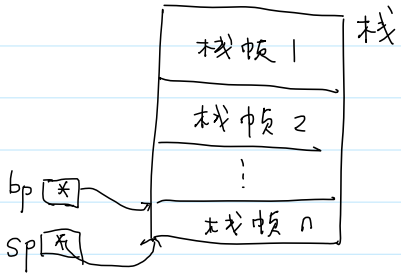
task → 任务

本质：进程（抽象）

CPU的状态：寄存器（PC、SP、BP、…）⟷（上下文）
占用的内存：进程的虚拟地址空间
持久性的存储设备：打开文件列表

下一条指令的地址
栈顶
base point

占用的资源




```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
  int eip;
  int esp;
  int ebx;
  int ecx;
  int edx;
  int esi;
  int edi;
  int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
  char *mem;                   // Start of process memory
  uint sz;                     // Size of process memory
  char *kstack;                // Bottom of kernel stack
                               // for this process
  enum proc_state state;       // Process state
  int pid;                     // Process ID
  struct proc *parent;         // Parent process
  void *chan;                  // If !zero, sleeping on chan
  int killed;                  // If !zero, has been killed
  struct file *ofile[NOFILE];  // Open files
  struct inode *cwd;           // Current directory
  struct context context;      // Switch here to run process
  struct trapframe *tf;        // Trap frame for the
                               // current interrupt
};
```

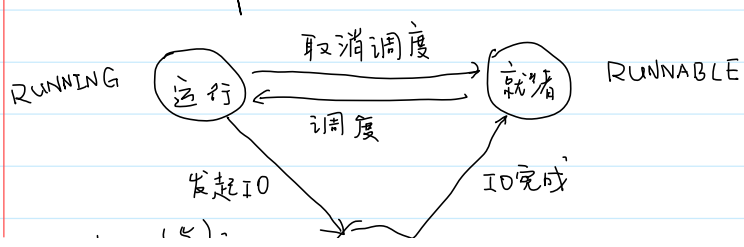
内存

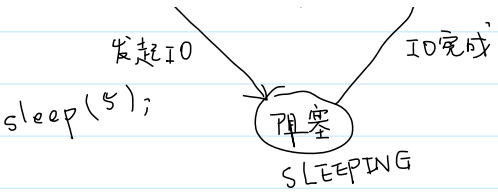
内核栈 → 内核执行态（免调用）

打开文件列表

上下文

进程的状态





并发机制：如何实现上下文切换？

指标：① 性能　不应该增加太多的额外开销
② 控制权　操作系统应该保留控制权。

表 6.1　直接运行协议（无限制）

操作系统	程序
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argc/argv 设置程序栈	
清除寄存器 执行 call main() 方法	
	执行 main() 从 main 中执行 return
释放进程的内存将进程 从进程列表中清除	

内核　Code　上层应用程序（任务）

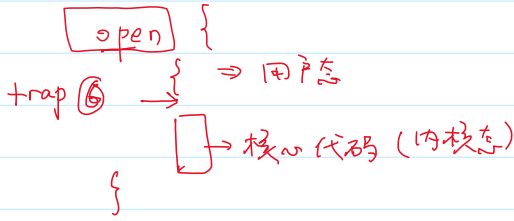
单CPU

优点：简单、快
缺点：没有控制权、不安全

如何限制应用程序的权限。（不能够访问非法的内存空间和执行一些特权指令）

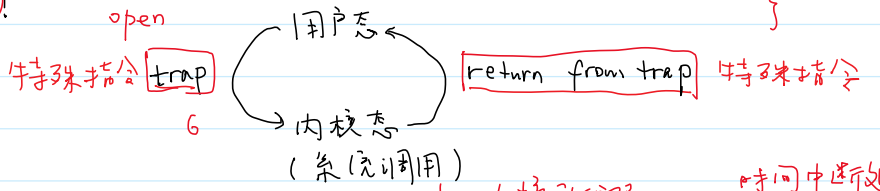
需要硬件的帮忙。

CPU的模态（模式）
用户态：应用程序　（不能够访问非法的内存空间和执行一些特权指令）
内核态：操作系统　（可以访问机器的所有资源）



问题：应用程序如何执行特权操作？

系统调用！



↳ 内核的代码

操作系统在启动时会设置陷阱表（里面放的是系统调用的位置）

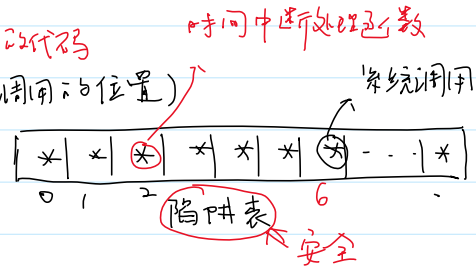
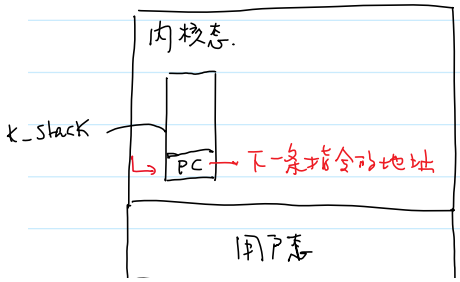


表 6.2　受限直接运行协议

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住系统调用处理程序的地址	
操作系统@运行（内核模式）	硬件	程序（应用模式）
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argv 设置程序栈		
用寄存器/程序计数器填充内核栈 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到 main	
		运行 main

PC → 用寄存器/程序计数器填充内核栈 PC
PC ↓ return from trap　从内核栈恢复寄存器 转向用户模式 跳到 main　PC



return from trap	从内核栈恢复寄存器 转向用户模式 跳到 main	pc
		运行 main …… 调用系统调用 陷入操作系统 (trap) 6
	将寄存器保存到内核栈 转向内核模式 跳到陷阱处理程序	pc while (1);
处理陷阱 做系统调用的工作 从陷阱返回 return from trap		
	从内核栈恢复寄存器 转向用户模式 跳到陷阱之后的程序计数器	
		……从 main 返回 陷入（通过 exit()）
释放进程的内存将进程 从进程列表中清除		

用户态

用户态 应用程序　内核态 操作系统

trap（保存上下文）

系统调用

return from trap（恢复上下文）

（上下文切换）

缺陷：控制权不是主动掌握在操作系统手里

问题：如何实现进程间的切换？

① 协作方式：等待系统调用，yield()　主动让

② 非协作式（抢占方式）：操作系统进行控制　（需要硬件的帮助）

yield

时钟中断：（时钟设备 每隔几毫秒可以产生一次时钟中断）

时间片（是时钟中断间隔的整数倍）　分配给进程运行的时间

变量

用户态 应用程序　内核态 操作系统

产生中断

中断处理函数

表 6.3　受限直接执行协议（时钟中断）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 系统调用处理程序 时钟处理程序	
启动中断时钟		
	启动时钟 每隔 x ms 中断 CPU	
操作系统@运行（内核模式）	硬件	程序（应用模式）
		进程 A……
	时钟中断 将寄存器（A）保存到内核栈（A） 转向内核模式 跳到陷阱处理程序	
处理陷阱 调用 switch()例程 将寄存器（A）保存到进程结构（A） 将进程结构（B）恢复到寄存器（B） 从陷阱返回（进入 B）		
	从内核栈（B）恢复寄存器（B） 转向用户模式 跳到 B 的程序计数器	
		进程 B……

内核栈　B　CPU

保存部分寄存器、PC

context

PC

进程之间的切换

1. 进程之间是隔离的（感知到内核和其他进程存在）

1. 进程之间是隔离（感知不到内核和其他进程是存在）

2. 进程是资源分配最小单位。

结论

2024年5月21日 14:26

① 进程是资源分配最小单位.

任务 ← 分配资源

② 进程是隔离

③ 上下文切换

调用态/内核调用

切换进程.

和进程相关的常用命令

2024年5月21日 14:29

```
PS(1)

NAME
       ps - report a snapshot of the current processes.
```

远程控制终端

```
he@he-vm:~$ ps
    PID TTY          TIME CMD
   1947 pts/0    00:00:00 bash
   3121 pts/0    00:00:00 ps
```

→ 显示和该终端关联的进程

状态

```
he@he-vm:~$ ps x
    PID TTY      STAT   TIME COMMAND
   1634 ?        Ss     0:00 /lib/systemd/systemd --user
   1635 ?        S      0:00 (sd-pam)
   1641 ?        Ssl    0:00 /usr/bin/pipewire
   1642 ?        Ssl    0:00 /usr/bin/pipewire-media-session
```

→ 显示和用户关联的进程

Deep Sleep

```
D    uninterruptible sleep (usually IO)
I    Idle kernel thread
R    running or runnable (on run queue)
S    interruptible sleep (waiting for an event to complete)
T    stopped by job control signal
t    stopped by debugger during the tracing
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

Sleep ← S

僵尸 ← Z

```
For BSD formats and when the stat keyword is used, additional characters may be displayed:

<    high-priority (not nice to other users)
N    low-priority (nice to other users)
L    has pages locked into memory (for real-time and custom IO)
s    is a session leader
l    is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
+    is in the foreground process group
```

前台进程组

```
he@he-vm:~$ ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.2 166708 11612 ?        Ss   09:24   0:02 /sbin/init splash
root           2  0.0  0.0      0     0 ?        S    09:24   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   09:24   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   09:24   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   09:24   0:00 [slub_flushwq]
root           6  0.0  0.0      0     0 ?        I<   09:24   0:00 [netns]
```

起始时间

→ 显示所有用户相关的进程

VSZ virtual memory size of the process in KiB (1024-byte units).

swap area (交换区)

RSS resident set size, the non-swapped physical memory that a task has used (in kilobytes).

常驻主内存的大小

TOP(1) → 类似windows的任务管理器，（每3秒统计进程的信息）

NAME

top - display Linux processes

当前时间　　平均负载，1min　5min　15min　　<1 低　>1 高

```
top - 14:55:57 up  5:31,  2 users,  load average: 0.15, 0.15, 0.16
Tasks: 309 total,   1 running, 308 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.1 us,  0.0 sy,  0.0 ni, 99.9 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   3870.6 total,   1645.5 free,   1020.2 used,   1204.9 buff/cache
MiB Swap:   3898.0 total,   3898.0 free,      0.0 used.   2566.2 avail Mem
```

进程 CPU　内存　交换区

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
   3574 he        20   0  143276  37320  29000 S   0.3   0.9   0:01.52 vmtoolsd
      1 root      20   0  166708  11612   8028 S   0.0   0.3   0:02.67 systemd
      2 root      20   0       0      0      0 S   0.0   0.0   0:00.03 kthreadd
      3 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_gp
      4 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 rcu_par_gp
      5 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 slub_flushwq
      6 root       0 -20       0      0      0 I   0.0   0.0   0:00.00 netns
```

打印进程树

```
he@he-vm:~$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─VGAuthService
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─avahi-daemon───avahi-daemon
        ├─bluetoothd
        ├─colord───2*[{colord}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─gdm3─┬─gdm-session-wor─┬─gdm-wayland-ses─┬─gnome-session-b───2*[{gnome-session-b}]
        │      │                 │                 └─2*[{gdm-wayland-ses}]
        │      │                 └─2*[{gdm-session-wor}]
        │      └─2*[{gdm3}]
```

前台进程 VS. 后台进程

2024年5月21日 15:03

```
he@he-vm:~$ xlogo
```

前台进程：受终端的控制

ctrl + C ：终止进程 SIGINT

ctrl + Z ：停止进程 （植物人） SIGTSTP

ctrl + \ ：退出进程 SIGQUIT

关闭终端，所有的前台进程都会终止。

```
he@he-vm:~$ xlogo &
[1] 4564
```

后台进程。

```
he@he-vm:~$ jobs
[1]+  Running                 xlogo &
```

```
he@he-vm:~$ fg 1
xlogo
^C
```

把后台任务拉到前台

获取进程的标识

2024年5月21日 15:15

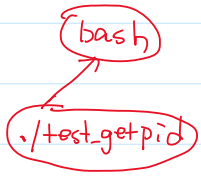
```
GETPID(2)          Linux Programmer's Manual

NAME
       getpid, getppid - get process identification

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t getpid(void);
       pid_t getppid(void);

ERRORS
       These functions are always successful.
```

parent

⇒ 当系统调用返回时，一定执行成功了。

```
test_getpid.c
 1 #include <func.h>
 2
 3 int main(int argc, char* argv[])
 4 {
 5     // ./test_getpid
 6     printf("pid = %d\n", getpid());
 7     printf("ppid = %d\n", getppid());
 8
 9     sleep(10);
10     return 0;
11 }
```

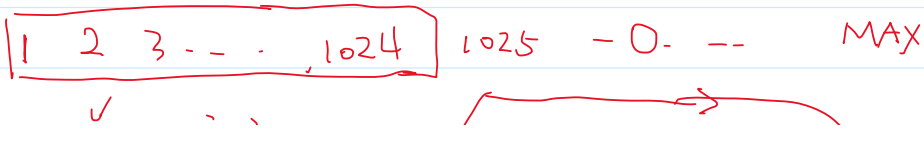
./test_getpid

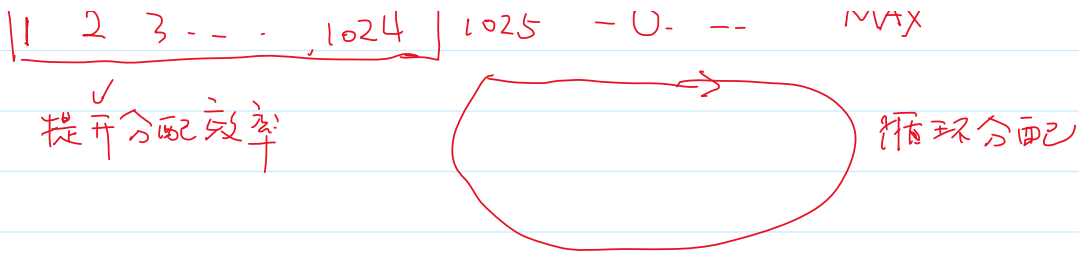



```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./test_getpid
pid = 5280
ppid = 5051
```

```
he@he-vm:~$ ps x
5051 pts/0    Ss     0:00 -bash
5280 pts/0    S+     0:00 ./test_getpid
```

Linux 进程 id 的分配策略




1 2 3 . . . 1024 1025 . . . MAX
提升分配效率
循环分配

进程的基本操作

2024年5月21日 15:51

创建进程：fork()

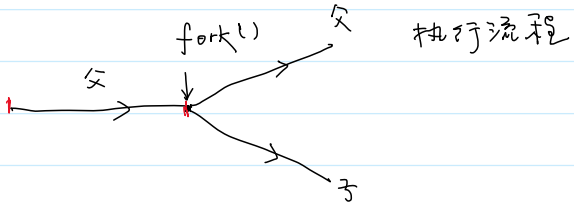
终止进程：exit()，_exit()，abort()，wait()，waitpid()

执行程序：exec函数族。

创建进程:fork()

2024年5月21日 15:54

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

NAME
       fork - create a child process

SYNOPSIS
       #include <sys/types.h>
       #include <unistd.h>

       pid_t fork(void);
```




```
RETURN VALUE
       On success, the PID of the child process is returned in the parent, and 0 is returned in
       the child.  On failure, -1 is returned in the parent, no child process is  created,  and
       errno is set appropriately.
```

成功:

父 ← 子进程的pid

子 ← 0

失败: 父 ← -1, 并且不会创建子进程, 设置errno

test_fork.c buffers

```c
#include <func.h>

int main(int argc, char* argv[])
{
    // ./test_fork
    printf("BEGIN\n");

    // idiom
    pid_t pid = fork();

    switch (pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        printf("I am a baby\n");
        printf("child: pid = %d, ppid = %d\n", getpid(), getppid());
        break;
    default:
        // 父进程
        printf("Who's your daddy?\n");
        printf("parent: pid = %d, childpid = %d\n", getpid(), pid);
        break;
    }

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

← 父子进程都是从 fork() 返回! 子进程不会执行前面的代码.

↵ 子进程

↵ 父进程

← 父,子进程

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./test_fork
BEGIN 一次!
Who's your daddy?
parent: pid = 5672, childpid = 5673
BYE BYE
I am a baby
child: pid = 5673, ppid = 5672
BYE BYE 两次
```

到底是父进程先执行，还是子进程先执行，这是不确定的。

（不能1假定到底是谁先执行）

#2. fork的原理

createProcess 重量级

fork 轻量级

1. 复制父进程 proc结构体，修改相应的信息。

pid

ppid

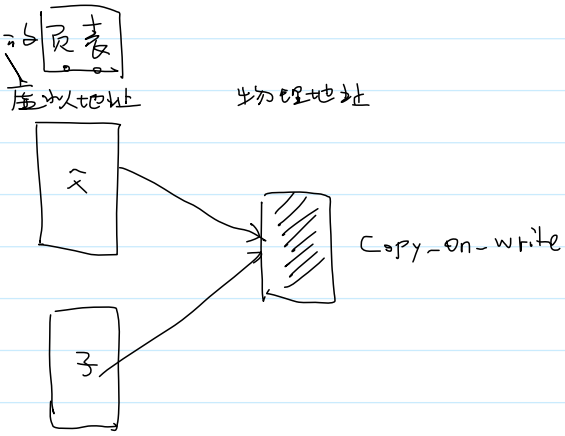
!

→ 继承父进程的属性（context）

↳ SP

↳ PC → 下一条指令的位置

2. 复制父进程的页表



#3.

代码段：父子进程是享 （不能修改）

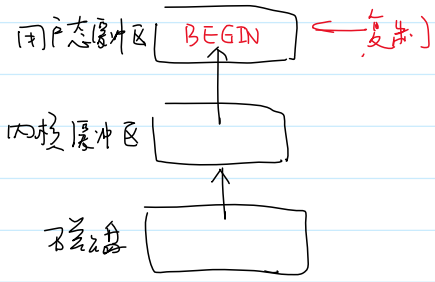
a、栈、堆、数据段 （父子私有的）

fork1.c buffers

```c
#include <func.h>

int g_value = 10;  // 数据段
int main(int argc, char* argv[])
{
    int l_value = 20;   // 栈
    int* d_value = (int*)malloc(sizeof(int)); // 堆
    *d_value = 30;

    pid_t pid = fork();
    switch (pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        g_value += 100;
        l_value += 100;
        *d_value += 100;
        printf("g_value = %d, l_value = %d, d_value = %d\n", g_value, l_value, *d_value);
        exit(0);
    default:
        // 父进程
        sleep(2);
        printf("g_value = %d, l_value = %d, d_value = %d\n", g_value, l_value, *d_value);
        exit(0);
    }
    return 0;
}
```

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./fork1
g_value = 110, l_value = 120, d_value = 130
g_value = 10, l_value = 20, d_value = 30
```

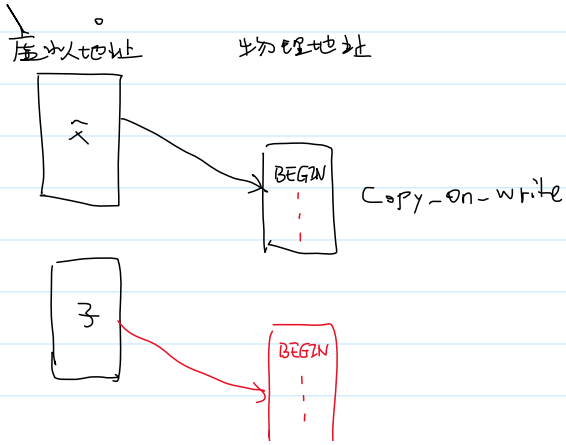
b. 用户态缓冲区. (标准流) (父子进程私有的)

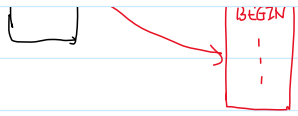


fork2.c buffers

```c
#include <func.h>

int main(int argc, char* argv[])
{
    printf("BEGIN:"); // stdout是行缓冲

    pid_t pid = fork();
    switch (pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        printf("I am a baby\n");
        exit(0);
    default:
        // 父进程
        sleep(2);
        printf("Who's your daddy?\n");
        exit(0);
    }
    return 0;
}
```

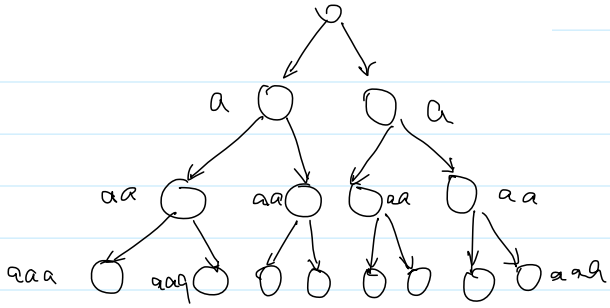
```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./fork2
BEGIN:I am a baby
BEGIN:Who's your daddy?
```





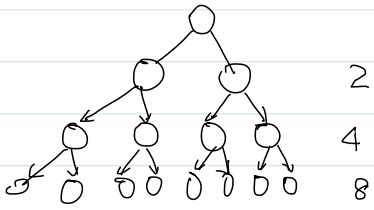
homework.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    for(int i = 0; i < 3; i++) {
        fork();
        printf("a");
    }
    return 0;
}
```



homework.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    for(int i = 0; i < 3; i++) {
        fork();
        printf("a\n");
    }
    return 0;
}
```



C. 打开文件 (共享的)

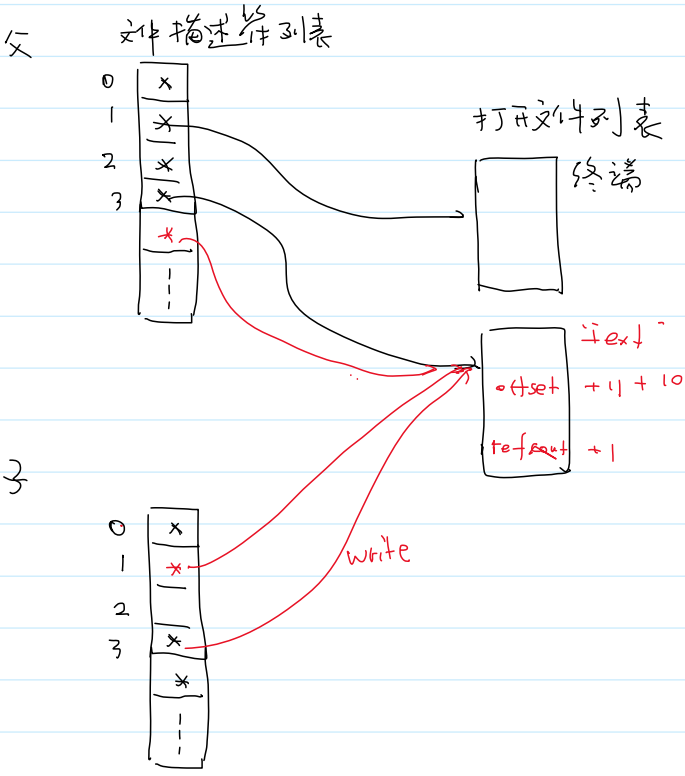
文件描述符列表 (私有的)

gg 二 对齐 G

fork3.c

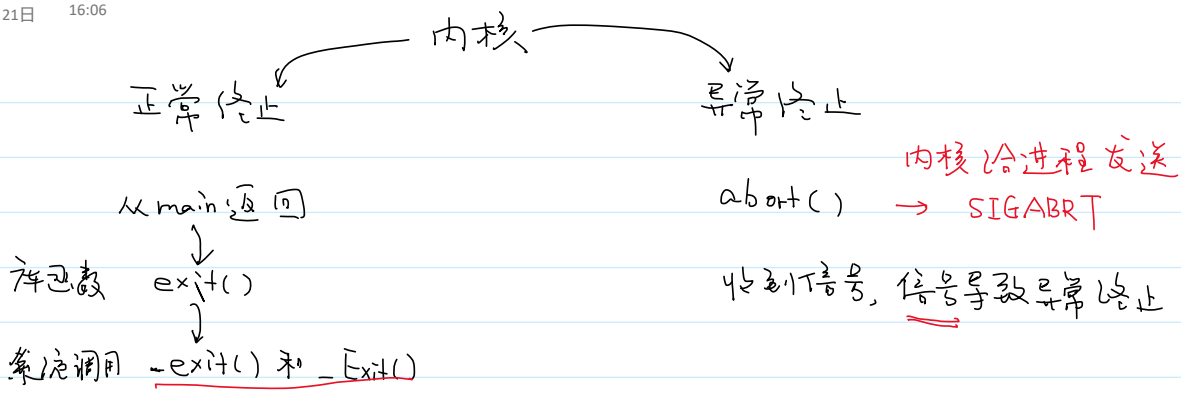
```c
#include <func.h>

int main(int argc, char* argv[])
{
    int fd = open("text", O_RDWR | O_CREAT | O_TRUNC, 0666);
    printf("pos: %ld\n", lseek(fd, 0, SEEK_CUR));

    pid_t pid = fork();
    int newfd;
    switch (pid) {
    case -1:
        error(1, errno, "fork");
    case 0:
        // 子进程
        write(fd, "Hello world", 11);

        close(STDOUT_FILENO);
        close(STDERR_FILENO);
        newfd = dup(fd);    // newfd=1
        printf("newfd = %d\n", newfd);
        exit(0);
    default:
        // 父进程
        sleep(2);
        printf("pos: %ld\n", lseek(fd, 0, SEEK_CUR));

        newfd = dup(fd);
        printf("newfd = %d\n", newfd);  // newfd = 4
        exit(0);
    }
    return 0;
}
```

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./fork3
pos: 0
pos: 21
newfd = 4
```



终止进程

2024年5月21日 16:06



执行成功：exit(0);
执行失败：exit(1);
} 用户

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ 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
```

正常终止

2024年5月21日 17:36

#1. exit().

① 执行 atexit() 注册的函数

② 刷新用户态缓冲区

③ 调用 _exit()，正常终止进程。

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

NAME
       atexit - register a function to be called at normal process termination

SYNOPSIS
       #include <stdlib.h>

       int atexit(void (*function)(void));
```

注册　　函数指针

标识符

```
void (*function)(void)
```

```
RETURN VALUE
       The  atexit()  function returns the value 0 if successful; otherwise it
       returns a nonzero value.
```

成功，0

失败，非0

test_exit.c

```c
#include <func.h>

// 执行一些资源清理操作
void func(void) {
    printf("I am going to die...");
}

int main(int argc, char* argv[])
{
    // 调用atexit()注册函数
    int err = atexit(func);
    if (err) {
        error(1, 0, "atexit");
    }

    // ...
    printf("Hello world");

    exit(123);
}
```

← 注册，不会执行 func

← 执行 func

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./test_exit
Hello worldI am going to die...he@he-vm:~/cpp58/2_Linux
```

#2. _exit()

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

NAME
       _exit, _Exit - terminate the calling process

SYNOPSIS
       #include <unistd.h>

       void _exit(int status);
```

→ 退出状态码.

→ 导致进程正常终止.

```
RETURN VALUE
       These functions do not return.
```

test__exit.c

```c
#include <func.h>

// 执行一些资源清理操作
void func(void) {
    printf("I am going to die...");
}

int main(int argc, char* argv[])
{
    // 调用atexit()注册函数
    int err = atexit(func);
    if (err) {
        error(1, 0, "atexit");
    }

    // ...
    printf("Hello world");

    exit(123);
}
```

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./test__exit
```

异常终止

2024年5月21日 17:54

abort()

↳ 内核就会给该进程发送 SIGABRT 信号

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

NAME
       abort - cause abnormal process termination


SYNOPSIS
       #include <stdlib.h>

       void abort(void);


RETURN VALUE
       The abort() function never returns.
```

test_abort.c

```c
#include <func.h>

// 执行一些资源清理操作
void func(void) {
    printf("I am going to die...");
}

int main(int argc, char* argv[])
{
    // 调用atexit()注册函数
    int err = atexit(func);
    if (err) {
        error(1, 0, "atexit");
    }

    // ...
    printf("Hello world");

    abort();

    printf("You cannot see me!\n");
}
```

```
he@he-vm:~/cpp58/2_Linux/Linux08 (master)$ ./test_abort
Aborted (core dumped)
```

Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from **abort**(3)

Core Default action is to terminate the process and dump core (see **core**(5)).

常见问题

2024年5月21日 21:21

作业1. 复制文件空洞

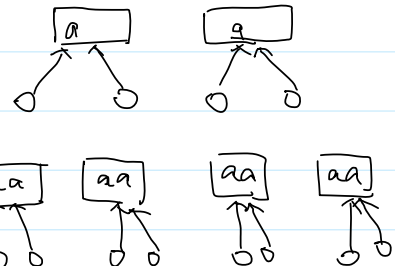
```c
#include <func.h>

#define PAGESIZE 4096

int main(int argc, char* argv[])
{
    // ./copy_holes src dst
    if (argc != 3) {
        error(1, 0, "Usage: %s src dst", argv[0]);
    }

    // 1.打开文件
    int srcfd = open(argv[1], O_RDONLY);
    if (srcfd == -1) {
        error(1, errno, "open %s", argv[1]);
    }

    int dstfd = open(argv[2], O_WRONLY | O_CREAT | O_TRUNC, 0666);
    if (dstfd == -1) {
        error(1, errno, "open %s", argv[2]);
    }

    // 2. 设置文件大小
    off_t fsize = lseek(srcfd, 0, SEEK_END);
    ftruncate(dstfd, fsize);
    lseek(srcfd, 0, SEEK_SET);

    // 3. 复制文件
    char empty[PAGESIZE] = {'\0'};
    char buf[PAGESIZE];

    int n;
    while ((n = read(srcfd, buf, PAGESIZE)) > 0) {
        if (memcmp(buf, empty, n) == 0) {
            lseek(dstfd, n, SEEK_CUR);
        } else {
            write(dstfd, buf, n);
        }
    }

    close(srcfd);
    close(dstfd);
    return 0;
}
```

作业2.

homework.c

```c
#include <func.h>

int main(int argc, char* argv[])
{
    for(int i = 0; i < 3; i++) {
        fork();
        printf("a");
    }
    return 0; → exit(0)
}
```

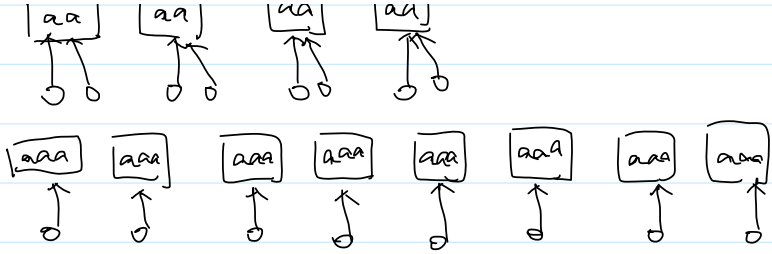
```
fork();
printf("a");
fork();
printf("a");
```



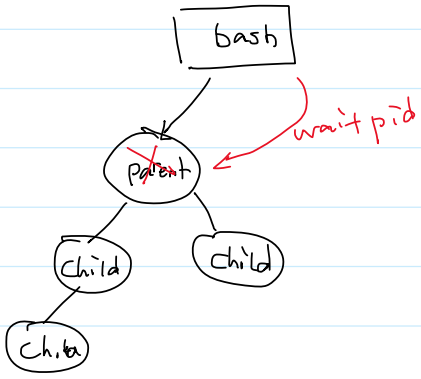
printf("a");

fork();

printf("a");



#3. aaa--a 提示符 $ aaa--a



预告

2024年5月21日 21:35

明天下午补课、

① wait / waitpid

② exec 函数族

③ 有名管道 / 匿名管道

④ select