

进程概述

说明

本部分笔记及源码出自 `slide/02Linux多进程开发/01 进程概述`

程序和进程

- **程序** 是包含一系列**信息**的文件，这些信息描述了如何在运行时创建一个**进程**
 - **二进制格式标识**：每个程序文件都包含用于描述可执行文件格式的元信息。内核利用此信息来解释文件中的其他信息，Linux中为ELF可执行连接格式
 - **机器语言指令**：对程序算法进行编码
 - **程序入口地址**：标识程序开始执行时的起始指令位置
 - **数据**：程序文件包含的变量初始值和程序使用的字面量值（比如字符串）
 - **符号表及重定位表**：描述程序中函数和变量的位置及名称。这些表格有多重用途，其中包括调试和运行时的符号解析（动态链接）
 - **共享库和动态链接信息**：程序文件所包含的一些字段，列出了程序运行时需要使用的共享库，以及加载共享库的动态连接器的路径名
 - 其他信息：程序文件还包含许多其他信息，用以描述如何创建进程
- **进程** 是**正在运行的程序**的实例。是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元
- 可以用**一个程序来创建多个进程**，进程是由内核定义的抽象实体，并为该实体分配用以执行程序的各项系统资源。从内核的角度看，进程由用户内存空间和一系列内核数据结构组成，其中用户内存空间包含了程序代码及代码所使用的变量，而内核数据结构则用于维护进程状态信息。记录在内核数据结构中的信息包括许多与进程相关的标识号（IDs）、虚拟内存表、打开文件的描述符表、信号传递及处理的有关信息、进程资源使用及限制、当前工作目录和大量的其他信息

单道、多道程序设计

- **单道程序**，即在计算机内存中只允许一个的程序运行
- **多道程序**设计技术是在计算机内存中同时存放几道相互独立的程序，使它们**在管理程序控制下，相互穿插运行**，两个或两个以上程序在计算机系统中同处于开始到结束之间的状态，这些程序共享计算机系统资源。**引入多道程序设计技术的根本目的是为了提高 CPU 的利用率**
- 对于一个**单 CPU 系统**来说，程序同时处于运行状态只是一种宏观上的概念，他们虽然都已经开始运行，但**就微观而言，任意时刻，CPU 上运行的程序只有一个**
- 在多道程序设计模型中，多个进程轮流使用 CPU。而当下常见 **CPU 为纳秒级**，1秒可以执行大约 10 亿条指令。由于**人眼的反应速度是毫秒级**，所以看似同时在运行

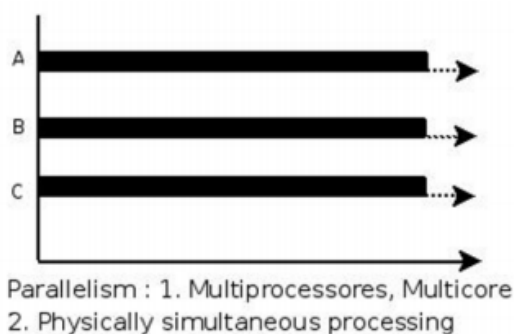
时间片

- **时间片**（`timeslice`）又称为**量子**（`quantum`）或**处理器片**（`processor slice`）是操作系统分配给每个正在运行的进程微观上的一段 CPU 时间。事实上，虽然一台计算机通常可能有多 CPU，但是同一个 CPU 永远不可能真正地同时运行多个任务。在只考虑一个 CPU 的情况下，这些进程“看起来像”同时运行的，实则是轮番穿插地运行，由于时间片通常很短（在 Linux 上为 `5ms - 800ms`），用户不会感觉到

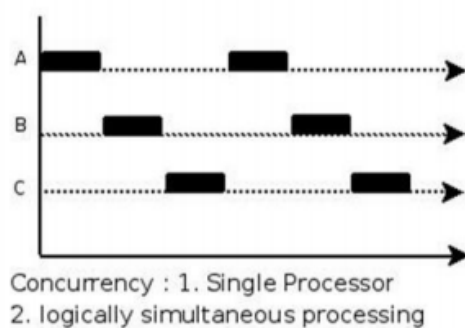
- **时间片由操作系统内核的调度程序分配给每个进程。**首先，内核会给每个进程分配相等的初始时间片，然后每个进程轮番地执行相应的时间，当所有进程都处于时间片耗尽的状态时，内核会重新为每个进程计算并分配时间片，如此往复

并行和并发

- **并行(parallel)**：指在同一时刻，有多条指令在多个处理器上同时执行
- **并发(concurrency)**：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行



并行



并发

进程控制块 (PCB)

- 为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。内核为每个进程分配一个 `PCB(Processing Control Block)` 进程控制块，维护进程相关的信息，Linux 内核的进程控制块是 `task_struct` 结构体
- 在 `/usr/src/linux-headers-xxx/include/linux/sched.h` 文件中可以查看 `struct task_struct` 结构体定义，其中 `linux-headers-xxx` 需要替换为该目录下相应的版本
- 需要掌握的 `struct task_struct` 结构体成员
 - **进程id**：系统中每个进程有唯一的 id，用 `pid_t` 类型表示，其实就是一个非负整数
 - **进程的状态**：有就绪、运行、挂起、停止等状态
 - 进程切换时需要**保存和恢复的一些CPU寄存器**
 - 描述**虚拟地址空间**的信息
 - 描述**控制终端**的信息
 - 当前工作目录 (Current Working Directory)
 - `umask` 掩码
 - 文件描述符表，包含很多指向 `file` 结构体的指针
 - 和信号相关的信息
 - 用户 id 和组 id
 - 会话 (Session) 和进程组
 - 进程可以使用的资源上限 (Resource Limit)，在Linux中可用 `ulimit -a` 查看资源上限

```
u@ubuntu:~$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 15435
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files              (-n) 1048576
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 15435
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
u@ubuntu:~$
```

进程状态

说明

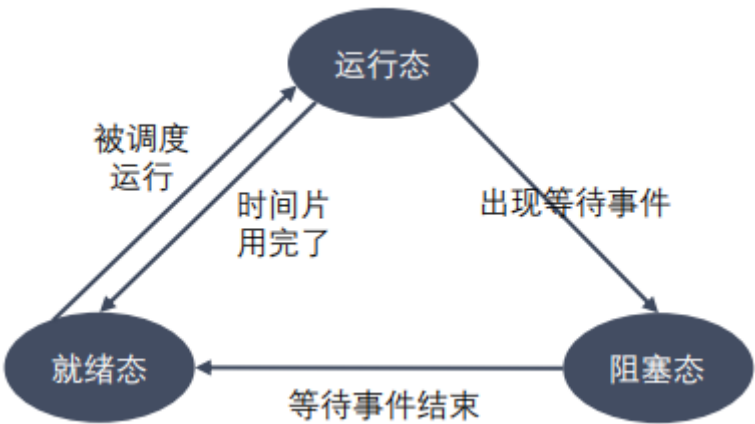
本部分笔记及源码出自 `slide/02Linux多进程开发/02 进程状态及转换`

基本概念

- 进程状态反映进程执行过程的变化，这些状态随着进程的执行和外界条件的变化而转换
- 分为 `三态模型` 和 `五态模型`

三态模型

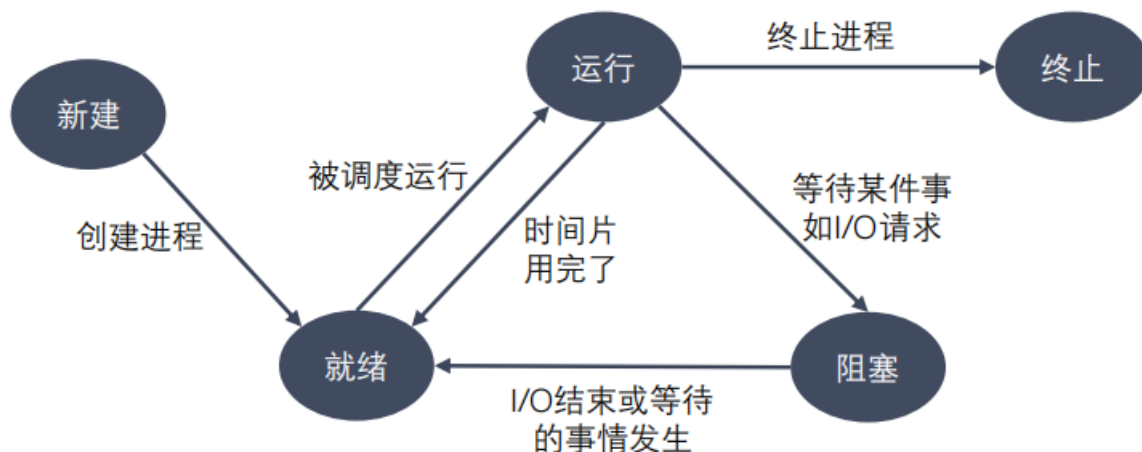
- `运行态`：进程占有处理器正在运行
- `就绪态`：进程具备运行条件，等待系统分配处理器以便运行。当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列
- `阻塞态`：又称为等待(wait)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件的完成



五态模型

- 除 `新建态` 和 `终止态`，其余三个状态与 `三态模型` 一致

- **新建态**：进程刚被创建时的状态，尚未进入就绪队列
- **终止态**：进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。进入终止态的进程以后不再执行，但依然保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程



进程相关命令

查看进程-静态

- **ps** 命令用来查看进程（静态），可以使用 **man ps** 查看使用说明

```

PS(1)                                User Commands                                PS(1)

NAME
    ps - report a snapshot of the current processes.

SYNOPSIS
    ps [options]

DESCRIPTION
    ps displays information about a selection of the active processes.  If you want a repetitive update of the selection and the displayed information, use top(1) instead.

    This version of ps accepts several kinds of options:

    1  UNIX options, which may be grouped and must be preceded by a dash.
    2  BSD options, which may be grouped and must not be used with a dash.
    3  GNU long options, which are preceded by two dashes.
  
```

- 常用参数含义
 - a: 显示终端上的所有进程，包括其他用户的进程
 - u: 显示进程的详细信息
 - x: 显示没有控制终端的进程
 - j: 列出与作业控制相关的信息
- **ps -aux** 或 **ps aux**

```

u@ubuntu:~/Desktop/Linux$ ps -aux
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2 225856  9532 ?        Ss   17:47   0:06 /sbin/init auto noprompt
root         2  0.0  0.0      0     0 ?        S    17:47   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        I<   17:47   0:00 [rcu_gp]
root         4  0.0  0.0      0     0 ?        I<   17:47   0:00 [rcu_par_gp]
root         6  0.0  0.0      0     0 ?        I<   17:47   0:00 [kworker/0:0H-kb]
root         9  0.0  0.0      0     0 ?        I<   17:47   0:00 [mm_percpu_wq]
root        10  0.0  0.0      0     0 ?        S    17:47   0:00 [ksoftirqd/0]
root        11  0.0  0.0      0     0 ?        I    17:47   0:09 [rcu_sched]
root        12  0.0  0.0      0     0 ?        S    17:47   0:00 [migration/0]
root        13  0.0  0.0      0     0 ?        S    17:47   0:00 [idle_inject/0]
root        14  0.0  0.0      0     0 ?        S    17:47   0:00 [cpuhp/0]
root        15  0.0  0.0      0     0 ?        S    17:47   0:00 [cpuhp/1]
root        16  0.0  0.0      0     0 ?        S    17:47   0:00 [idle_inject/1]
  
```

- **USER**：进程所属用户
- **PID**：进程ID
- **%CPU**：CPU使用占比

- **%MEM**：内存使用占比
- **TTY**：进程所属终端，在终端直接执行 **tty** 可查看当前 Terminal 所属终端（因为此时我还打开了另外两个终端）

```
u@ubuntu:~/Desktop/Linux$ tty
/dev/pts/3
u@ubuntu:~/Desktop/Linux$
```

- **STAT**：进程状态
 - D：不可中断 Uninterruptible (usually IO)
 - R：正在运行，或在队列中的进程
 - S(大写)：处于休眠状态
 - T：停止或被追踪
 - Z：僵尸进程
 - W：进入内存交换（从内核2.6开始无效）
 - X：死掉的进程
 - <：高优先级
 - N：低优先级
 - s：包含子进程
 - +：位于前台的进程组
- **START**：进程开始执行时间
- **TIME**：进程执行持续时间
- **COMMAND**：进程执行命令
- **ps -ajx 或 ps ajx**

```
u@ubuntu:~/Desktop/Linux$ ps -ajx
  PPID    PID   PGID   SID  TTY      TPGID STAT   UID    TIME COMMAND
    0      1      1      1  ?        -1 Ss      0      0:06 /sbin/init auto noprompt
    0      2      0      0  ?        -1 S       0      0:00 [kthreadd]
    2      3      0      0  ?        -1 I<      0      0:00 [rcu_gp]
    2      4      0      0  ?        -1 I<      0      0:00 [rcu_par_gp]
    2      6      0      0  ?        -1 I<      0      0:00 [kworker/0:0H-kb]
    2      9      0      0  ?        -1 I<      0      0:00 [mm_percpu_wq]
    2     10      0      0  ?        -1 S       0      0:00 [ksoftirqd/0]
    2     11      0      0  ?        -1 I       0      0:09 [rcu_sched]
    2     12      0      0  ?        -1 S       0      0:00 [migration/0]
    2     13      0      0  ?        -1 S       0      0:00 [idle_inject/0]
    2     14      0      0  ?        -1 S       0      0:00 [cpuhp/0]
    2     15      0      0  ?        -1 S       0      0:00 [cpuhp/1]
    2     16      0      0  ?        -1 S       0      0:00 [idle_inject/1]
    2     17      0      0  ?        -1 S       0      0:01 [migration/1]
    2     18      0      0  ?        -1 S       0      0:02 [ksoftirqd/1]
```

- **PPID**：该进程的父进程ID
- **PGID**：该进程所属组ID
- **SID**：该进程所属会话(session)ID，多个组构成会话

查看进程-动态

- **top**

```
top - 20:58:25 up 3:11, 1 user, load average: 0.09, 0.04, 0.01
Tasks: 338 total, 1 running, 265 sleeping, 0 stopped, 0 zombie
%Cpu(s): 1.2 us, 1.3 sy, 0.5 ni, 95.9 id, 1.0 wa, 0.0 hi, 0.1 si, 0.0 st
KiB Mem : 4001708 total, 469092 free, 1642780 used, 1889836 buff/cache
KiB Swap: 1942896 total, 1942116 free, 780 used, 2062592 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S %CPU  %MEM    TIME+  COMMAND
 13241 u        20   0 953956 73628 32772 S   5.6   1.8   0:21.21 node
 13280 u        20   0 991256 46616 30616 S   5.6   1.2   0:33.85 node
 31214 u        20   0 51320 4032 3300 R   5.6   0.1   0:00.02 top
   1 root     20   0 225856 9532 6656 S   0.0   0.2   0:07.01 systemd
   2 root     20   0      0      0 0 S   0.0   0.0   0:00.05 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
   6 root     0 -20      0      0 0 I   0.0   0.0   0:00.00 kworker/0:0H-kb
   9 root     0 -20      0      0 0 I   0.0   0.0   0:00.00 mm_percpu_wq
  10 root     20   0      0      0 0 S   0.0   0.0   0:00.65 ksoftirqd/0
  11 root     20   0      0      0 0 I   0.0   0.0   0:10.41 rcu_sched
```

- 可以在使用 top 命令时加上 -d 来指定显示信息更新的时间间隔
- 在 top 命令执行后，可以按以下按键对显示的结果进行排序
 - M：根据内存使用量排序
 - P：根据 CPU 占有率排序
 - T：根据进程运行时间长短排序
 - U：根据用户名来筛选进程
 - K：输入指定的 PID 杀死进程

杀死进程

- kill [-signal] pid
- kill -l：列出所有信号

```
u@ubuntu:~/Desktop/Linux$ 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
u@ubuntu:~/Desktop/Linux$
```

- kill -9 进程ID 等价于 kill -SIGKILL 进程ID

```
u        30501  0.2  0.2 55080 10000 S    0.0   0.0   0:00.00 python3 /usr/bin/update-manager --no-update --no-re
root    31012  0.0  0.0      0      0 ?    I   20:24  0:00 [kworker/1:1-cgr]
root    31204  0.0  0.0      0      0 ?    I   20:47  0:00 [kworker/u256:1-]
root    31209  0.0  0.0      0      0 ?    I   20:53  0:00 [kworker/u256:0-]
root    31275  0.0  0.0      0      0 ?    I   20:58  0:00 [kworker/u256:2-]
root    31710  0.0  0.0      0      0 ?    I   21:02  0:00 [kworker/0:2-cgr]
u        31798  0.5  0.1 29556 4520 pts/4  Ss+  21:03  0:00 bash
u        31806  0.0  0.0 14580  780 ?    S   21:03  0:00 sleep 180
u        31807  0.0  0.0 46776 3648 pts/3  R+   21:03  0:00 ps aux
u@ubuntu:~/Desktop/Linux$ kill -9 31798
u@ubuntu:~/Desktop/Linux$
```

- killall name：根据进程名杀死进程

进程号和相关函数

- 每个进程都由进程号来标识，其类型为 pid_t（整型），进程号的范围：0~32767。进程号总是唯一的，但可以重用。当一个进程终止后，其进程号就可以再次使用
- 任何进程（除 init 进程）都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号（PPID）
- 进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号（PGID）。默认情况下，当前的进程号会当做当前的进程组号

- 进程号和进程组相关函数
 - `pid_t getpid(void);`: 获取进程ID
 - `pid_t getppid(void);`: 获取进程的父进程ID
 - `pid_t getpgid(pid_t pid);`: 获取进程的组ID

进程创建

说明

本部分笔记及源码出自 `slide/02Linux多进程开发/03 进程创建`

进程创建：fork

- 可通过 `man 2 fork` 查看帮助

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

NAME
    fork - create a child process

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

    pid_t fork(void);
```

- `pid_t fork(void);`

```
/*
    #include <sys/types.h>
    #include <unistd.h>

    pid_t fork(void);
    函数的作用：用于创建子进程。
    返回值：
        fork()的返回值会返回两次。一次是在父进程中，一次是在子进程中。
        在父进程中返回创建的子进程的ID，
        在子进程中返回0
        如何区分父进程和子进程：通过fork的返回值。
        在父进程中返回-1，表示创建子进程失败，并且设置errno
*/

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

int main()
{
    int num = 10;

    // 创建子进程
    pid_t pid = fork();

    // 判断是父进程还是子进程
    if(pid > 0) {
        printf("pid : %d\n", pid);
```



```

// 如果大于0，返回的是创建的子进程的进程号，当前是父进程
printf("i am parent process, pid : %d, ppid : %d\n", getpid(),
getppid());

printf("parent num : %d\n", num);
num += 10;
printf("parent num += 10 : %d\n", num);
} else if(pid == 0) {
// 当前是子进程
printf("i am child process, pid : %d, ppid : %d\n",
getpid(),getppid());

printf("child num : %d\n", num);
num += 100;
printf("child num += 100 : %d\n", num);
}

// for循环
for(int i = 0; i < 3; i++) {
printf("i : %d , pid : %d\n", i , getpid());
sleep(1);
}

return 0;
}

```

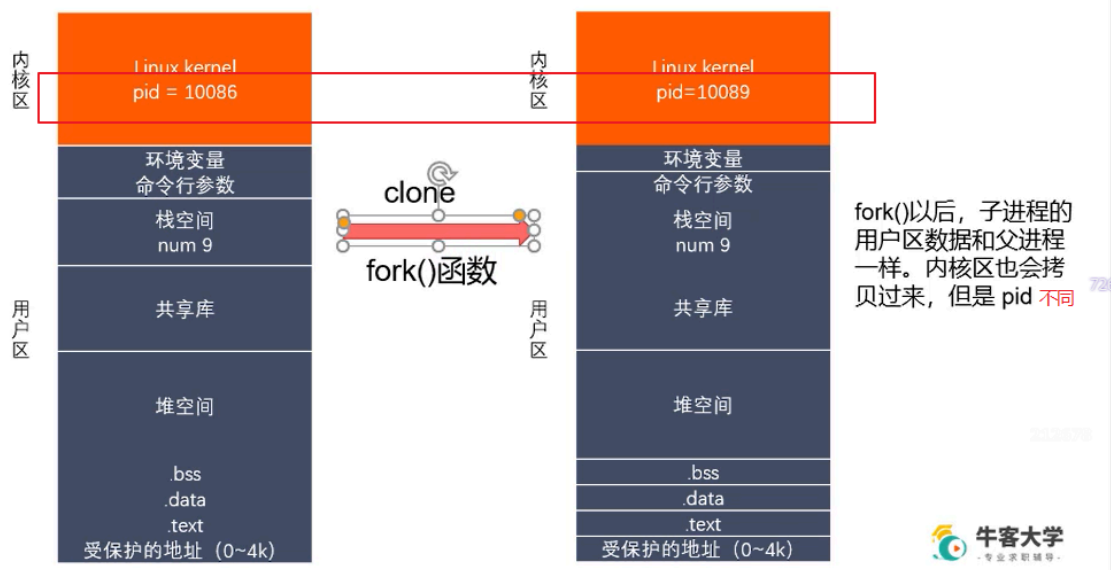
```

u@ubuntu:~/Desktop/Linux$ gcc fork.c -o fork
u@ubuntu:~/Desktop/Linux$ ./fork
pid : 32599
i am parent process, pid : 32598, ppid : 32373
parent num : 10
parent num += 10 : 20
i : 0 , pid : 32598
i am child process, pid : 32599, ppid : 32598
child num : 10
child num += 100 : 110
i : 0 , pid : 32599
i : 1 , pid : 32598
i : 1 , pid : 32599
i : 2 , pid : 32598
i : 2 , pid : 32599
u@ubuntu:~/Desktop/Linux$

```

fork工作原理

- Linux 的 `fork()` 使用是通过**写时拷贝 (copy-on-write)** 实现。写时拷贝是一种可以推迟甚至避免拷贝数据的技术
- 内核此时并不复制整个进程的地址空间，而是让**父子进程共享同一个地址空间**，只有在需要写入的**时候**才会复制地址空间，从而使各个进程拥有各自的地址空间。即**资源的复制是在需要写入的时候才会进行**，在此之前，只有以**只读方式共享**（示例程序中 `num` 的作用）
- fork之后父子进程共享文件**。fork产生的子进程与父进程有**相同的文件描述符**，指向相同的文件表，引用计数增加，共享文件偏移指针
- 使用**虚拟地址空间**，由于用的是**写时拷贝 (copy-on-write)**，下图不完全准确，但可帮助理解



父子进程关系

区别

- **fork()函数的返回值不同。**父进程中: >0 返回的是子进程的ID, 子进程中: =0
- **pcb中的一些数据不同。**pcb中存的是**当前进程的ID(pid)**, **当前进程的父ID(ppid)**和**信号集**

共同点

- 在某些状态下, 即**子进程刚被创建出来, 还没有执行任何的写数据的操作**。此时**用户区的数据和文件描述符表**父进程和子进程一样

父子进程对变量共享说明

- 刚开始的时候, 是一样的, 共享的。如果修改了数据, 不共享了
- 读时共享 (子进程被创建, 两个进程没有做任何写的操作), 写时拷贝

GDB 多进程调试

- 在以下调试程序**第10行**及**第20行**打断点, 后续说明都基于这两个断点

```

(gdb) 1
1      #include <stdio.h>
2      #include <unistd.h>
3
4      int main()
5      {
6          printf("begin\n");
7
8          if(fork() > 0) {
9
10             printf("我是父进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb)
11
12             int i;
13             for(i = 0; i < 10; i++) {
14                 printf("i = %d\n", i);
15                 sleep(1);
16             }
17
18             } else {
19
20             printf("我是子进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb)
21
22             int j;
23             for(j = 0; j < 10; j++) {
24                 printf("j = %d\n", j);
25                 sleep(1);
26             }
27
28             }
29
30             return 0;
(gdb)

```

- 打断点及查看

```

(gdb) b 10
Breakpoint 1 at 0x7c8: file gdb4fork.c, line 10.
(gdb) b 20
Breakpoint 2 at 0x81e: file gdb4fork.c, line 20.
(gdb) i b
Num      Type          Disp Enb Address                What
1        breakpoint    keep y   0x00000000000007c8 in main at gdb4fork.c:10
2        breakpoint    keep y   0x000000000000081e in main at gdb4fork.c:20
(gdb)

```

- 使用 GDB 调试的时候，GDB 默认只能跟踪一个进程，可以在 fork 函数调用之前，通过指令设置 GDB 调试工具跟踪父进程或者是跟踪子进程，**默认跟踪父进程**
- 查看当前跟踪的进程： `show follow-fork-mode`

```

(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
(gdb)

```

- 设置调试父进程或者子进程： `set follow-fork-mode [parent (默认) | child]`

```

(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
(gdb) set follow-fork-mode child
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
(gdb)

```

- 调试父进程，子进程循环会自动执行，完毕后需要输入 `n` 继续执行父进程

```
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "parent".
(gdb) r
Starting program: /home/u/Desktop/Linux/gdb4fork
begin
我是子进程: pid = 38236, ppid = 38232
j = 0

Breakpoint 1, main () at gdb4fork.c:10
10      printf("我是父进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb) j = 1
j = 2
j = 3
j = 4
j = 5
j = 6
j = 7
j = 8
j = 9
█
```

- 调试子进程，父进程循环会自动执行，完毕后需要输入 `n` 继续执行子进程

```
(gdb) show follow-fork-mode
Debugger response to a program call of fork or vfork is "child".
(gdb) r
Starting program: /home/u/Desktop/Linux/gdb4fork
begin
我是父进程: pid = 38000, ppid = 37925
i = 0
[New process 38004]
[Switching to process 38004]

Thread 2.1 "gdb4fork" hit Breakpoint 2, main () at gdb4fork.c:20
20      printf("我是子进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb) i = 1
i = 2
i = 3
i = 4
i = 5
i = 6
i = 7
i = 8
i = 9
█
```

- 查看调试模式: `show detach-on-fork`

```
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is on.
(gdb) █
```

- 设置调试模式: `set detach-on-fork [on | off]`
 - 默认为 on，表示调试当前进程的时候，其它的进程继续运行，如果为 off，调试当前进程的时候，其它进程被 GDB 挂起
 - 注：在设置为 off 时，执行程序会报以下错误，原因是 **gdb 8.x 版本存在 bug**

```
(gdb) set detach-on-fork off
(gdb) r
Starting program: /home/u/Desktop/Linux/gdb4fork
begin
[New process 37356]
Reading symbols from /home/u/Desktop/Linux/gdb4fork...done.
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x7c8
Cannot insert breakpoint 2.
Cannot access memory at address 0x81e
(gdb) █
```

- 以下正常执行的 gdb 版本为 v7.11.1 (截图来源于视频), 与设置为 on 的区别在于, for 循环是否打印

```
(gdb) r
Starting program: /root/Linux/hello
begin
[New process 5636]
Reading symbols from /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.23.so...done.
Reading symbols from /usr/lib/debug/lib/x86_64-linux-gnu/ld-2.23.so...done.

Thread 1.1 "hello" hit Breakpoint 1, main () at hello.c:10
10      printf("我是父进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb) n
我是父进程: pid = 5632, ppid = 5630
13      for(i = 0; i < 10; i++) {
(gdb) n
14          printf("i = %d\n", i);
(gdb) n
```

- 查看调试的进程: info inferiors, 此时调试进程为 parent, 需要执行后才会显示进程
 - 当 detach-on-fork 为 on 时, 只会显示一个进程 (因为另一个进程已经执行完毕, 销毁, 猜测)

```
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is on.
(gdb) r
Starting program: /home/u/Desktop/Linux/gdb4fork
begin
我是子进程: pid = 40962, ppid = 40958
j = 0

Breakpoint 1, main () at gdb4fork.c:10
10      printf("我是父进程: pid = %d, ppid = %d\n", getpid(), getppid());
(gdb) j = 1
j = 2
j = 3
j = 4
j = 5
j = 6
j = 7
(gdb) j = 8
j = 9
n
我是父进程: pid = 40958, ppid = 40914
13      for(i = 0; i < 10; i++) {
(gdb) info inferiors
Num  Description      Executable
* 1   process 40958      /home/u/Desktop/Linux/gdb4fork
(gdb) █
```

- 当 detach-on-fork 为 off 时, 会显示两个进程

```
(gdb) show detach-on-fork
Whether gdb will detach the child of a fork is off.
(gdb) r
Starting program: /home/u/Desktop/Linux/gdb4fork
begin
[New process 41173]
Reading symbols from /home/u/Desktop/Linux/gdb4fork...done.
Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x7c8
Cannot insert breakpoint 2.
Cannot access memory at address 0x81e
(gdb) info inferiors
Num Description Executable
* 1 process 41164 /home/u/Desktop/Linux/gdb4fork
2 process 41173 /home/u/Desktop/Linux/gdb4fork
(gdb)
```

- 切换当前调试的进程: `inferior Num`
- 使进程脱离 GDB 调试: `detach inferiors Num`

exec函数族

说明

本部分笔记及源码出自 `slide/02Linux多进程开发/04 exec函数族`

基本概念

- 可通过 `man 3 exec` 查看帮助

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

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

SYNOPSIS
    #include <unistd.h>

    extern char **environ;

    int execl(const char *path, const char *arg, ...
              /* (char *) NULL */);
    int execlp(const char *file, const char *arg, ...
              /* (char *) NULL */);
    int execl(const char *path, const char *arg, ...
              /*, (char *) NULL, char * const envp[] */);
    int execv(const char *path, char *const argv[]);
    int execvp(const char *file, char *const argv[]);
    int execvpe(const char *file, char *const argv[],
               char *const envp[]);
```

- `exec` 函数族 的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件
- `exec` 函数族的函数执行成功后不会返回，因为调用进程的实体，包括代码段，数据段和堆栈等都被新的内容取代，只留下进程 ID 等一些表面上的信息仍保持原样，颇有些神似“三十六计”中的“金蝉脱壳”。看上去还是旧的躯壳，却已经注入了新的灵魂。只有调用失败了，它们才会返回 -1，从原程序的调用点接着往下执行
- 用户区替换为 `a.out` 的内容，内核区不变



a.out

种类

- 基本组件为 `exec`，后面跟不同参数，代表不同含义
 - `l(list)`：参数地址列表，以空指针结尾
 - `v(vector)`：存有各参数地址的指针数组的地址
 - `p(path)`：按 `PATH` 环境变量指定的目录搜索可执行文件，可用 `env` 查看现有的环境变量

```
ubuntu@ubuntu:~/Desktop/Linux$ env
LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so=01;35:bd=40;33:01:cd=40;33:01:or=40;31:01:mi=00:su=37;41:sg=30;43:ca=30;41:tw=30;42:ow=34;42:st=37;44:ex=01;32:*.tar=01;31:*.tgz=01;31:*.arc=01;31:*.arj=01;31:*.taz=01;31:*.lha=01;31:*.lzh=01;31:*.lzm=01;31:*.tlz=01;31:*.txz=01;31:*.tzo=01;31:*.t7z=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.dz=01;31:*.gz=01;31:*.lrz=01;31:*.lz=01;31:*.lzo=01;31:*.xz=01;31:*.zst=01;31:*.tztst=01;31:*.bz2=01;31:*.bz=01;31:*.tbz=01;31:*.tbz2=01;31:*.tz=01;31:*.deb=01;31:*.rpm=01;31:*.jar=01;31:*.war=01;31:*.ear=01;31:*.sar=01;31:*.rar=01;31:*.alz=01;31:*.ace=01;31:*.zoo=01;31:*.cpio=01;31:*.7z=01;31:*.rz=01;31:*.cab=01;31:*.wim=01;31:*.swm=01;31:*.dwm=01;31:*.esd=01;31:*.jpg=01;31:*.jpeg=01;31:*.mjpg=01;31:*.mjpeg=01;31:*.gif=01;31:*.bmp=01;31:*.pbm=01;31:*.pgm=01;31:*.ppm=01;31:*.tga=01;31:*.xpm=01;31:*.tif=01;31:*.tiff=01;31:*.png=01;31:*.svg=01;31:*.svgz=01;31:*.mng=01;31:*.pcx=01;31:*.mov=01;31:*.mpg=01;31:*.mpeg=01;31:*.m2v=01;31:*.mkv=01;31:*.webm=01;31:*.ogm=01;31:*.mp4=01;31:*.m4v=01;31:*.mp4v=01;31:*.vob=01;31:*.qt=01;31:*.nuv=01;31:*.wmv=01;31:*.asf=01;31:*.rm=01;31:*.rmvb=01;31:*.flc=01;31:*.avi=01;31:*.fli=01;31:*.flv=01;31:*.gl=01;31:*.dl=01;31:*.xcf=01;31:*.xwd=01;31:*.yuv=01;31:*.au=00;36:*.ogg=00;36:*.opus=00;36:*.spx=00;36:*.xspf=00;36:
SSH_CONNECTION=192.168.1.100 22 192.168.1.100 22
LESSCLOSE=/usr/bin/lesspipe %s %s
LANG=en_US.UTF-8
OLDPWD=/home/u/Desktop
COLORTERM=truecolor
XDG_SESSION_ID=12
USER=u
PWD=/home/u/Desktop/Linux
HOME=/home/u
BROWSER=/home/u/.vscode-server/bin/7f6ab5485bbc088386c4386d8876667e155244e/bin/helpers/browser.sh
TERM_PROGRAM=vscode
SSH_CLIENT=192.168.1.100 22 192.168.1.100 22
TERM_PROGRAM_VERSION=1.60.2
XDG_DATA_DIRS=/usr/local/share:/usr/share:/var/lib/napd/desktop
VSCODE_IPC_HOOK_CLI=/run/user/1000/vscode-ipc-82cad088-df55-4182-bd04-98e96ee2bfd.sock
MAIL=/var/mail/u
TERM=screen-256color
SHELL=/bin/bash
SHLVL=4
LOGNAME=u
DBUS_SESSION_BUS_ADDRESS=unix:path=/run/user/1000/bus
XDG_RUNTIME_DIR=/run/user/1000
PATH=/home/u/.vscode-server/bin/7f6ab5485bbc088386c4386d8876667e155244e/bin:/home/u/.vscode-server/bin/7f6ab5485bbc088386c4386d8876667e155244e/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
LESSOPEN=| /usr/bin/lesspipe %s
_=/usr/bin/env
ubuntu@ubuntu:~/Desktop/Linux$
```

- `e(environment)`：存有环境变量字符串地址的指针数组的地址，增加新的环境变量
- 说明：下列示例程序除核心代码外，保持一致，初始包含文件有

```
u@ubuntu:~/Desktop/Linux$ ls
hello hello.c
u@ubuntu:~/Desktop/Linux$
```

- `int execl(const char *path, const char *arg, .../* (char *) NULL */);`
 - `path`：需要指定的执行的文件的路径或者名称

- `arg`: 是执行可执行文件所需要的参数列表。第一个参数一般没有什么作用, 为了方便, 一般写的是执行的程序的名称, 从第二个参数开始往后, 就是程序执行所需要的参数列表, 参数最后需要以NULL结束 (哨兵)
- code

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

int main() {

    // 创建一个子进程, 在子进程中执行exec函数族中的函数
    pid_t pid = fork();

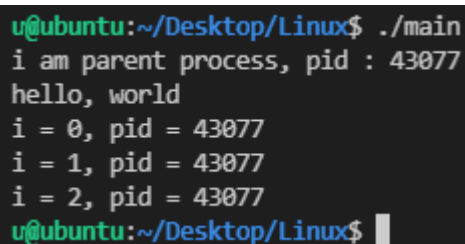
    if(pid > 0) {
        // 父进程
        printf("i am parent process, pid : %d\n", getpid());
        // 如果不加这句, 会存在孤儿进程, 输出异常
        sleep(1);
    } else if(pid == 0) {
        // 子进程
        // 调用自己写的可执行程序
        execl("/home/u/Desktop/Linux/hello", "hello", NULL);

        // 调用系统进程
        // execl("/bin/ps", "ps", "aux", NULL);
        perror("execl");
        printf("i am child process, pid : %d\n", getpid());
    }

    for(int i = 0; i < 3; i++) {
        printf("i = %d, pid = %d\n", i, getpid());
    }

    return 0;
}
```

- output



```
u@ubuntu:~/Desktop/Linux$ ./main
i am parent process, pid : 43077
hello, world
i = 0, pid = 43077
i = 1, pid = 43077
i = 2, pid = 43077
u@ubuntu:~/Desktop/Linux$
```

- 说明: 可以看到, 子进程的内容 (用户区) 被替换, 打印的是 `hello` 中的内容
- `int execlp(const char *file, const char *arg, ... /* (char *) NULL */);`
 - 会到环境变量中查找指定的可执行文件, 如果找到了就执行, 找不到就执行不成功
 - `file`: 只需要提供名称 (不需要提供路径)
 - code


```

#include <unistd.h>
#include <stdio.h>

int main() {

    // 创建一个子进程，在子进程中执行exec函数族中的函数
    pid_t pid = fork();

    if(pid > 0) {
        // 父进程
        printf("i am parent process, pid : %d\n", getpid());
        sleep(1);
    } else if(pid == 0) {
        // 子进程
        execlp("ps", "ps", "aux", NULL);

        printf("i am child process, pid : %d\n", getpid());
    }

    for(int i = 0; i < 3; i++) {
        printf("i = %d, pid = %d\n", i, getpid());
    }

    return 0;
}

```

◦ output

```

root    42317  0.2  0.0      0      0 ?      I   06:22   0:29 [kworker/0:2-eve]
root    42354  0.0  0.0      0      0 ?      I   06:57   0:06 [kworker/1:0-mpt]
root    42434  0.0  0.0      0      0 ?      I   08:01   0:00 [kworker/1:2-cgr]
root    42478  0.0  0.0      0      0 ?      I   08:49   0:00 [kworker/u256:1-]
root    42612  0.0  0.0      0      0 ?      I   09:02   0:00 [kworker/0:1-cgr]
root    43029  0.0  0.0      0      0 ?      I   09:21   0:00 [kworker/u256:0-]
u       43079  0.1  0.3 4820968 15068 ?      Sl  09:26   0:00 /home/u/.vscode-server/extensions/
root    43113  0.0  0.0      0      0 ?      I   09:27   0:00 [kworker/u256:2-]
u       43115  0.0  0.0   14580   736 ?      S   09:27   0:00 sleep 180
u       43125  0.0  0.0    4512   728 pts/5    S+  09:29   0:00 ./main
u       43126  0.0  0.0   46776  3560 pts/5    R+  09:29   0:00 ps aux
i = 0, pid = 43125
i = 1, pid = 43125
i = 2, pid = 43125
u@ubuntu:~/Desktop/Linux$

```

- `int execl(const char *path, const char *arg, .../*, (char *) NULL, char * const envp[] */);`

- `envp`: 添加路径至环境变量, 注意以 `NULL` 结尾, 否则报 `execl: Bad address`
- code

```

#include <unistd.h>
#include <stdio.h>

int main() {

    // 创建一个子进程，在子进程中执行exec函数族中的函数
    pid_t pid = fork();

```

```

if(pid > 0) {
    // 父进程
    printf("i am parent process, pid : %d\n",getpid());
    sleep(1);
}else if(pid == 0) {
    // 子进程
    // 需要已NULL结尾, 否则报 execle: Bad address 错误
    char* envp[] = {"/home/u/Desktop/Linux/", NULL};
    execle("/home/u/Desktop/Linux/hello", "hello", NULL, envp);
    perror("execle");
    printf("i am child process, pid : %d\n", getpid());
}

for(int i = 0; i < 3; i++) {
    printf("i = %d, pid = %d\n", i, getpid());
}

return 0;
}

```

◦ output

```

u@ubuntu:~/Desktop/Linux$ gcc main.c -o main
u@ubuntu:~/Desktop/Linux$ ./main
i am parent process, pid : 44124
hello, world
i = 0, pid = 44124
i = 1, pid = 44124
i = 2, pid = 44124
u@ubuntu:~/Desktop/Linux$

```

- `int execev(const char *path, char *const argv[]);`

- `argv`: 将运行参数都写在数组中

- code

```

#include <unistd.h>
#include <stdio.h>

int main() {

    // 创建一个子进程, 在子进程中执行exec函数族中的函数
    pid_t pid = fork();

    if(pid > 0) {
        // 父进程
        printf("i am parent process, pid : %d\n",getpid());
        sleep(1);
    }else if(pid == 0) {
        // 子进程
        char* argv[] = {"hello", NULL};
        execev("/home/u/Desktop/Linux/hello", argv);
        perror("execev");
        printf("i am child process, pid : %d\n", getpid());
    }
}

```

```

    }

    for(int i = 0; i < 3; i++) {
        printf("i = %d, pid = %d\n", i, getpid());
    }

    return 0;
}

```

◦ output

```

u@ubuntu:~/Desktop/Linux$ gcc main.c -o main
u@ubuntu:~/Desktop/Linux$ ./main
i am parent process, pid : 44167
hello, world
i = 0, pid = 44167
i = 1, pid = 44167
i = 2, pid = 44167
u@ubuntu:~/Desktop/Linux$

```

- `int execvp(const char *file, char *const argv[]);`
- `int execvpe(const char *file, char *const argv[], char *const envp[]);`
- `int execve(const char *filename, char *const argv[], char *const envp[]);`

进程控制

说明

本部分笔记及源码出自 `slide/02Linux多进程开发/05 进程控制`

进程退出

- 标准C库: `exit()`
- Linux系统: `_exit()`
- 区别

```

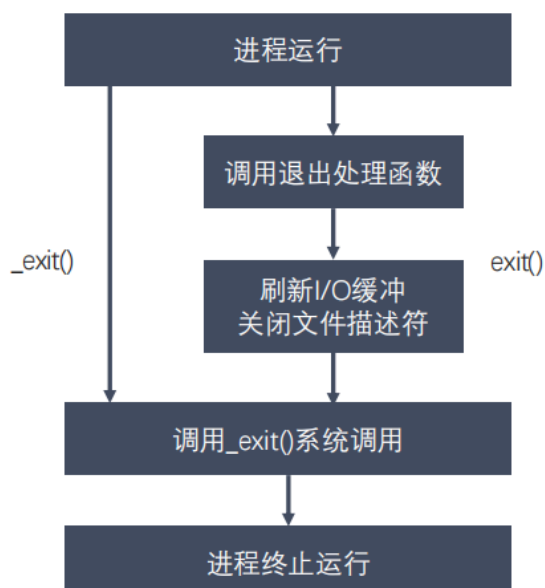
#include <stdlib.h>

void exit(int status);

#include <unistd.h>

void _exit(int status);

```



- 程序说明

◦ `exit()`

```
u@ubuntu:~/Desktop/Linux$ cat exit.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {

    printf("hello\n");
    printf("world");
    exit(0);
    return 0;
}u@ubuntu:~/Desktop/Linux$ gcc exit.c -o exit
u@ubuntu:~/Desktop/Linux$ ./exit
hello
worldu@ubuntu:~/Desktop/Linux$
```

◦ `_exit()`

```
u@ubuntu:~/Desktop/Linux$ cat exit.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {

    printf("hello\n");
    printf("world");
    _exit(0);
    return 0;
}u@ubuntu:~/Desktop/Linux$ gcc exit.c -o exit
u@ubuntu:~/Desktop/Linux$ ./exit
hello
u@ubuntu:~/Desktop/Linux$
```

- 原因：调用 `_exit` 时没有刷新缓冲区，所以 `world` 还留在缓冲区中，没有被输出，`\n` 会刷新缓冲区

孤儿进程

- 父进程运行结束，但子进程还在运行（未运行结束），这样的子进程就称为 孤儿进程（Orphan Process）
- 每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 `init`，而 `init` 进程会循环地 `wait()` 它的已经退出的子进程。
- 孤儿进程并不会有什么危害

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main()
{
    // 创建子进程
    pid_t pid = fork();
    // 判断是父进程还是子进程
```

```

    if(pid > 0) {
        printf("i am parent process, pid : %d, ppid : %d\n", getpid(),
getppid());
    } else if(pid == 0) {
        sleep(1);
        // 当前是子进程
        printf("i am child process, pid : %d, ppid : %d\n", getpid(),getppid());
    }
    // for循环
    for(int i = 0; i < 3; i++) {
        printf("i : %d , pid : %d\n", i , getpid());
    }

    return 0;
}

```

```

u@ubuntu:~/Desktop/Linux$ gcc orphan.c -o orphan
u@ubuntu:~/Desktop/Linux$ ./orphan
i am parent process, pid : 44858, ppid : 39381
i : 0 , pid : 44858
i : 1 , pid : 44858
i : 2 , pid : 44858
u@ubuntu:~/Desktop/Linux$ i am child process, pid : 44859, ppid : 1
i : 0 , pid : 44859
i : 1 , pid : 44859
i : 2 , pid : 44859

```

僵尸进程

- 每个进程结束之后，都会释放自己地址空间中的用户区数据，内核区的 PCB 没有办法自己释放掉，需要父进程去释放
- 进程终止时，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成 僵尸（Zombie）进程
- **僵尸进程不能被 kill -9 杀死**，这样就会导致一个问题，如果父进程不调用 `wait()` 或 `waitpid()` 的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免
- 示例
 - code

```

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

int main() {

    // 创建子进程
    pid_t pid = fork();

    // 判断是父进程还是子进程
    if(pid > 0) {
        while(1) {

```

```

        printf("i am parent process, pid : %d, ppid : %d\n",
getpid(), getppid());
        sleep(1);
    }

    } else if(pid == 0) {
        // 当前是子进程
        printf("i am child process, pid : %d, ppid : %d\n",
getpid(),getppid());

    }

    // for循环
    for(int i = 0; i < 3; i++) {
        printf("i : %d , pid : %d\n", i , getpid());
    }

    return 0;
}

```

- 僵尸进程ID: 45161, 可以通过杀死父进程45160, 从而使僵尸进程变为孤儿进程, 让init领养进行释放

```

u      43327  0.0  0.5 4820968 22164 ?      Sl   09:42   0:04 /home/u/.vscode-
root   43982  0.0  0.0      0      0 ?      I    09:48   0:00 [kworker/0:0-cgr
root   44230  0.0  0.0      0      0 ?      I    10:00   0:00 [kworker/1:1-cgr
root   44742  0.0  0.0      0      0 ?      I    10:36   0:00 [kworker/u256:2-
root   44746  0.0  0.0      0      0 ?      I    10:42   0:00 [kworker/u256:0-
root   44950  0.0  0.0      0      0 ?      I    10:58   0:00 [kworker/u256:1-
root   44952  0.0  0.0      0      0 ?      I    11:00   0:00 [kworker/1:2-cgr
u      44953  0.0  0.0   14580    852 ?      S    11:00   0:00 sleep 180
u      45041  0.1  0.3 4821100 13012 ?      Sl   11:01   0:00 /home/u/.vscode-
u      45063  0.6  0.1   29680   5060 pts/1   Ss   11:01   0:00 /bin/bash
u      45160  0.0  0.0   4512    716 pts/5   S+   11:02   0:00 ./zombie
u      45161  0.0  0.0      0      0 pts/5   Z+   11:02   0:00 [zombie] <defunc
u      45166  0.0  0.0    4632    864 ?      S    11:02   0:00 /bin/sh -c "/hom
u      45167  0.0  0.0   19996   3316 ?      S    11:02   0:00 /bin/bash /home/
u      45172  0.0  0.0   14580    792 ?      S    11:02   0:00 sleep 1
u      45173  0.0  0.0   46776   3620 pts/1   R+   11:02   0:00 ps aux
u@ubuntu:~/Desktop/Linux$

```

- 释放后

```

root   42317  0.2  0.0      0      0 ?      I    06:22   0:41 [kworker/0:2-eve
root   42354  0.0  0.0      0      0 ?      I    06:57   0:09 [kworker/1:0-mpt
u      43327  0.0  0.5 4820968 22164 ?      Sl   09:42   0:04 /home/u/.vscode-
root   43982  0.0  0.0      0      0 ?      I    09:48   0:00 [kworker/0:0-cgr
root   44230  0.0  0.0      0      0 ?      I    10:00   0:00 [kworker/1:1-cgr
root   44742  0.0  0.0      0      0 ?      I    10:36   0:00 [kworker/u256:2-
root   44746  0.0  0.0      0      0 ?      I    10:42   0:00 [kworker/u256:0-
root   44950  0.0  0.0      0      0 ?      I    10:58   0:00 [kworker/u256:1-
root   44952  0.0  0.0      0      0 ?      I    11:00   0:00 [kworker/1:2-cgr
u      45041  0.0  0.3 4821100 13012 ?      Sl   11:01   0:00 /home/u/.vscode-
u      45063  0.1  0.1   29680   5060 pts/1   Ss   11:01   0:00 /bin/bash
u      45549  0.0  0.0   14580    748 ?      S    11:03   0:00 sleep 180
u      45762  0.0  0.0    4632    876 ?      S    11:04   0:00 /bin/sh -c "/hom
u      45763  0.0  0.0   19996   3332 ?      S    11:04   0:00 /bin/bash /home/
u      45766  0.0  0.0   14580    736 ?      S    11:04   0:00 sleep 1
u      45767  0.0  0.0   46776   3572 pts/1   R+   11:04   0:00 ps aux
u@ubuntu:~/Desktop/Linux$

```

进程回收

基本概念

- 在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内存等。但是仍然为其保留一定的信息，这些信息主要指进程控制块PCB的信息（包括进程号、退出状态、运行时间等）
- 父进程可以通过调用 `wait` 或 `waitpid` 得到它的退出状态同时彻底清除掉这个进程，查看帮助：
`man 2 wait`
- `wait()` 和 `waitpid()` 函数的功能一样，区别在于
 - `wait()` 函数会阻塞
 - `waitpid()` 可以设置是否阻塞，`waitpid()` 还可以指定等待哪个子进程结束
- 注意：一次 `wait` 或 `waitpid` 调用只能清理一个子进程，清理多个子进程应使用循环

退出信息相关宏函数

- `WIFEXITED(status)`：非0，进程正常退出
- `WEXITSTATUS(status)`：如果上宏为真，获取进程退出的状态（`exit`的参数）
- `WIFSIGNALED(status)`：非0，进程异常终止
- `WTERMSIG(status)`：如果上宏为真，获取使进程终止的信号编号
- `WIFSTOPPED(status)`：非0，进程处于暂停状态
- `WSTOPSIG(status)`：如果上宏为真，获取使进程暂停的信号的编号
- `WIFCONTINUED(status)`：非0，进程暂停后已经继续运行

wait()

- 可通过 `man 2 wait` 查看帮助

```
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 wait(int *wstatus);`
 - 功能：等待任意一个子进程结束，如果任意一个子进程结束了，此函数会回收子进程的资源
 - 参数
 - `int *wstatus`：进程退出时的状态信息，传入的是一个int类型的地址，传出参数。
 - 返回值
 - 成功：返回被回收的子进程的id
 - 失败：-1 (所有的子进程都结束，调用函数失败)
- 其他说明

- 调用wait函数的进程会被挂起（阻塞），直到它的一个子进程退出或者收到一个不能被忽略的信号时才被唤醒（相当于继续往下执行）
- 如果没有子进程了，函数立刻返回，返回-1；如果子进程都已经结束了，也会立即返回，返回-1

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
    // 有一个父进程，创建5个子进程（兄弟）
    pid_t pid;

    // 创建5个子进程
    for(int i = 0; i < 5; i++) {
        pid = fork();
        // 避免嵌套重复生成子进程
        if(pid == 0) {
            break;
        }
    }

    if(pid > 0) {
        // 父进程
        while(1) {
            printf("parent, pid = %d\n", getpid());
            // int ret = wait(NULL);
            int st;
            int ret = wait(&st);

            if(ret == -1) {
                break;
            }

            if(WIFEXITED(st)) {
                // 是不是正常退出
                printf("退出的状态码: %d\n", WEXITSTATUS(st));
            }
            if(WIFSIGNALED(st)) {
                // 是不是异常终止
                printf("被哪个信号干掉了: %d\n", WTERMSIG(st));
            }

            printf("child die, pid = %d\n", ret);

            sleep(1);
        }
    } else if (pid == 0){
        // 子进程
        while(1) {
            printf("child, pid = %d\n", getpid());
```

```

        sleep(1);
    }

    exit(0);
}

return 0; // exit(0)
}

```

- 程序开始执行

```

child, pid = 47548
child, pid = 47549
child, pid = 47551
child, pid = 47547
child, pid = 47548
child, pid = 47551
child, pid = 47550
child, pid = 47549
child, pid = 47547
child, pid = 47550
child, pid = 47548
child, pid = 47551
child, pid = 47549
root 47090 0.0 0.0 0 0 ? I 14:57 0:00 [kworker/u256:0-
root 47431 0.0 0.0 0 0 ? I 15:02 0:00 [kworker/u256:1-
root 47433 0.0 0.0 0 0 ? I 15:03 0:00 [kworker/1:2-cgr
u 47434 0.0 0.0 14580 792 ? S 15:03 0:00 sleep 180
u 47475 0.1 0.3 4821100 14604 ? Sl 15:05 0:00 /home/u/.vscode-
u 47500 0.2 0.1 29548 4956 pts/1 Ss 15:05 0:00 /bin/bash
u 47546 0.0 0.0 4512 764 pts/5 S+ 15:06 0:00 ./wait
u 47547 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47548 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47549 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47550 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47551 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47582 0.0 0.0 46776 3720 pts/1 R+ 15:06 0:00 ps aux
u@ubuntu:~/Desktop$

```

- 通过命令杀死子进程: `kill -9 47548`

```

child, pid = 47547
child die, pid = 47548
child, pid = 47551
child, pid = 47550
child, pid = 47547
child, pid = 47549
parent, pid = 47546
child, pid = 47551
child, pid = 47550
u 47475 0.1 0.3 4821100 14604 ? Sl 15:05 0:00 /home/u/.vscode-
u 47500 0.2 0.1 29548 4956 pts/1 Ss 15:05 0:00 /bin/bash
u 47546 0.0 0.0 4512 764 pts/5 S+ 15:06 0:00 ./wait
u 47547 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47548 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47549 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47550 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47551 0.0 0.0 4512 72 pts/5 S+ 15:06 0:00 ./wait
u 47582 0.0 0.0 46776 3720 pts/1 R+ 15:06 0:00 ps aux
u@ubuntu:~/Desktop$ kill -9 47548
u@ubuntu:~/Desktop$

```

waitpid()

- 可通过 `man 2 wait` 查看帮助

```

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);

    int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
    /* This is the glibc and POSIX interface; see
       NOTES for information on the raw system call. */

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```

- `pid_t waitpid(pid_t pid, int *wstatus, int options);`
 - 功能: 回收指定进程号的子进程, 可以设置是否阻塞
 - 参数
 - `pid`
 - `pid > 0`: 回收某个子进程的pid
 - `pid = 0`: 回收当前进程组的所有子进程
 - `pid = -1`: 回收所有的子进程, 相当于 `wait()` (最常用)
 - `pid < -1`: 某个进程组的组id的绝对值, 回收指定进程组中的子进程
 - `options`: 设置阻塞或者非阻塞
 - `0`: 阻塞

- WNOHANG : 非阻塞
- 返回值
 - > 0 : 返回子进程的id
 - 0 : options=WNOHANG, 表示还有子进程活着
 - -1 : 错误, 或者没有子进程了

```
#include <sys/types.h>
#include <sys/wait.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main() {

    // 有一个父进程, 创建5个子进程 (兄弟)
    pid_t pid;

    // 创建5个子进程
    for(int i = 0; i < 5; i++) {
        pid = fork();
        if(pid == 0) {
            break;
        }
    }

    if(pid > 0) {
        // 父进程
        while(1) {
            printf("parent, pid = %d\n", getpid());
            sleep(1);

            int st;
            // int ret = waitpid(-1, &st, 0);
            int ret = waitpid(-1, &st, WNOHANG);

            if(ret == -1) {
                break;
            } else if(ret == 0) {
                // 说明还有子进程存在
                continue;
            } else if(ret > 0) {

                if(WIFEXITED(st)) {
                    // 是不是正常退出
                    printf("退出的状态码: %d\n", WEXITSTATUS(st));
                }
                if(WIFSIGNALED(st)) {
                    // 是不是异常终止
                    printf("被哪个信号干掉了: %d\n", WTERMSIG(st));
                }

                printf("child die, pid = %d\n", ret);
            }
        }
    }
}
```

```

    }

    } else if (pid == 0){
        // 子进程
        while(1) {
            printf("child, pid = %d\n",getpid());
            sleep(1);
        }
        exit(0);
    }

    return 0;
}

```

```

child, pid = 49095
child, pid = 49094
被哪个信号干掉了: 9
child die, pid = 49093
parent, pid = 49090
child, pid = 49091
child, pid = 49092
child, pid = 49095
parent, pid = 49090
child, pid = 49094
child, pid = 49091
child, pid = 49092
child, pid = 49095
parent, pid = 49090
child, pid = 49094
child, pid = 49091
child, pid = 49092
child, pid = 49095
parent, pid = 49090
child, pid = 49094

```

root	49020	0.0	0.0	0	0 ?	I	15:28	0:00	[kworker/1:0-cg]
root	49032	0.0	0.0	0	0 ?	I	15:34	0:00	[kworker/u256:0-
root	49038	0.0	0.0	0	0 ?	I	15:39	0:00	[kworker/u256:2-
u	49067	0.0	0.0	14580	784 ?	S	15:42	0:00	sleep 180
u	49090	0.0	0.0	4512	812 pts/5	S+	15:43	0:00	./waitpid
u	49091	0.0	0.0	4512	72 pts/5	S+	15:43	0:00	./waitpid
u	49092	0.0	0.0	4512	72 pts/5	S+	15:43	0:00	./waitpid
u	49093	0.0	0.0	4512	72 pts/5	S+	15:43	0:00	./waitpid
u	49094	0.0	0.0	4512	72 pts/5	S+	15:43	0:00	./waitpid
u	49095	0.0	0.0	4512	72 pts/5	S+	15:43	0:00	./waitpid
u	49096	0.1	0.3	4821100	14764 ?	Sl	15:43	0:00	/home/u/.vscode-
u	49169	0.1	0.1	29548	4664 pts/1	Ss	15:43	0:00	/bin/bash
u	49358	0.0	0.0	46776	3672 pts/1	R+	15:43	0:00	ps aux

```

u@ubuntu:~/Desktop$ kill -9 49093
u@ubuntu:~/Desktop$

```

进程间通信之管道及内存映射

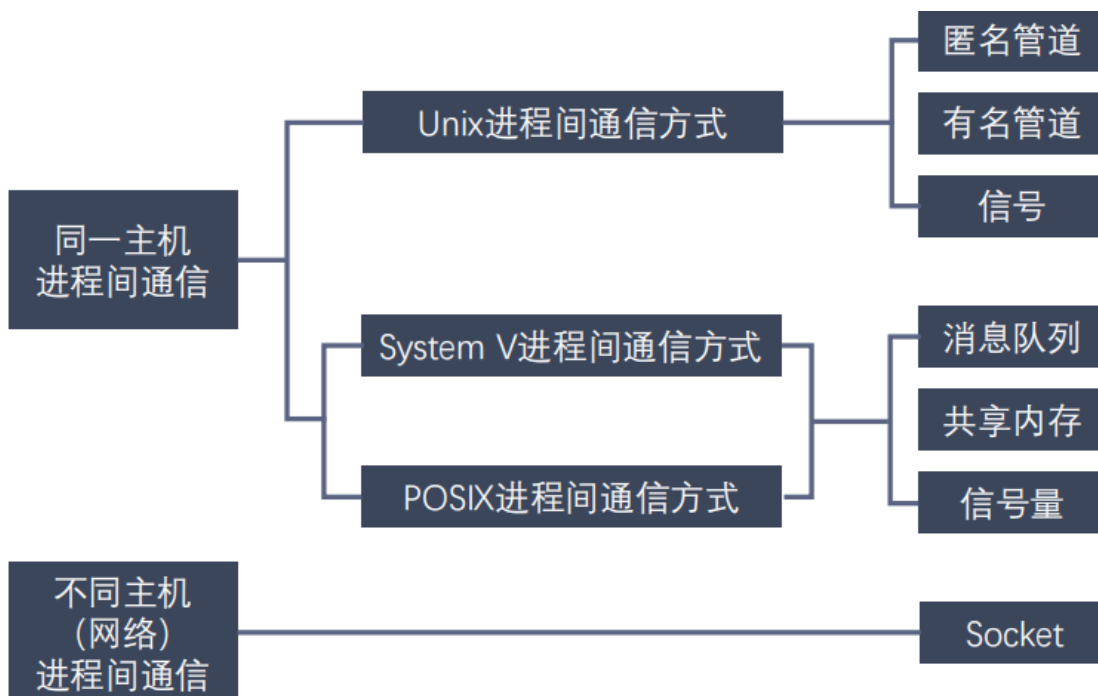
说明

本部分笔记及源码出自 [slide/02Linux多进程开发/06 进程间通信之管道及内存映射](#)

进程间通讯概念

- 进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源
- 但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要 进程间通信（IPC: Inter Processes Communication）
- 进程间通信的目的
 - 数据传输：一个进程需要将它的数据发送给另一个进程
 - 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）
 - 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制
 - 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变

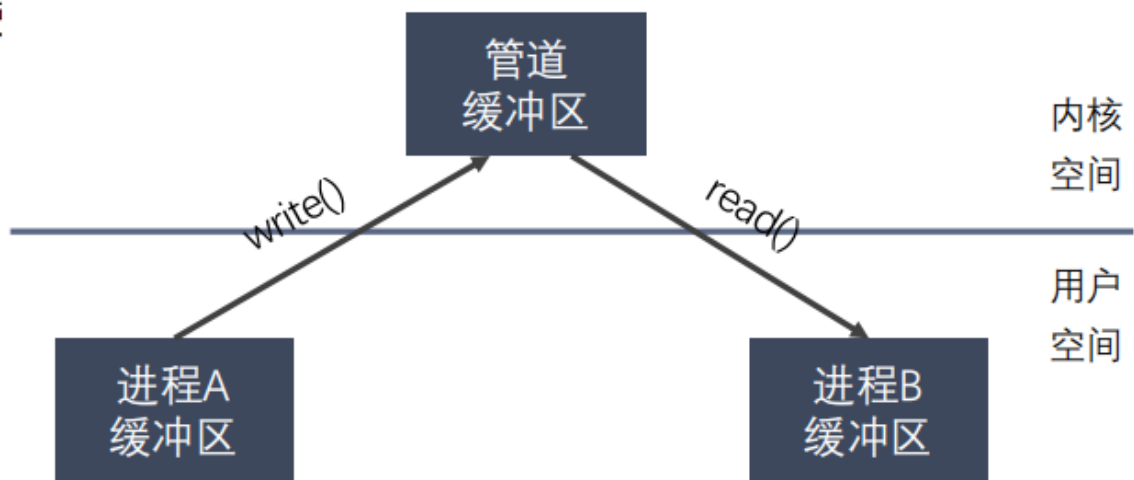
Linux 进程间通信的方式



管道

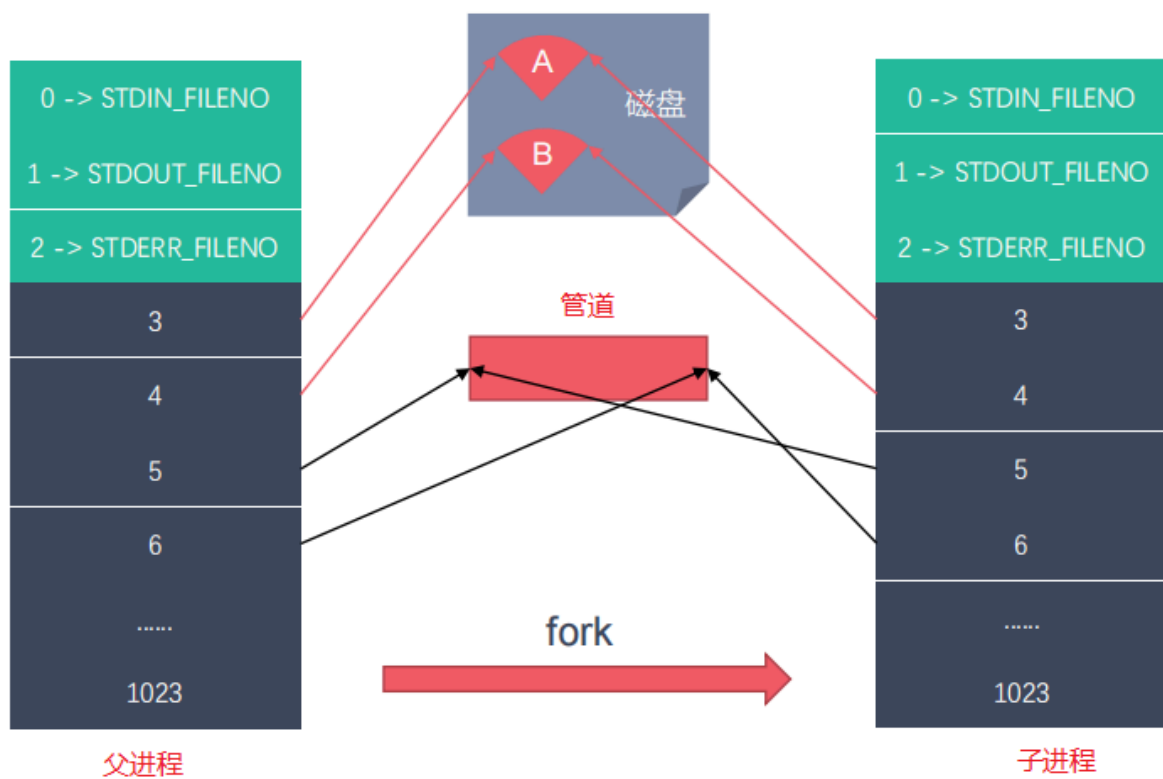
管道特点

- 管道其实是一个在**内核内存中维护的缓冲器**，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同
- 管道拥有文件的特质：读操作、写操作
 - **匿名管道**没有文件实体
 - **有名管道**有文件实体，但不存储数据。可以按照操作文件的方式对管道进行操作
- **一个管道是一个字节流**，使用管道时不存在消息或者消息边界的概念，从管道读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少
- 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺序是完全一样的
- 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，管道是**半双工**的
- 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据，**在管道中无法使用 lseek() 来随机的访问数据**
- **匿名管道**只能在**具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）**之间使用

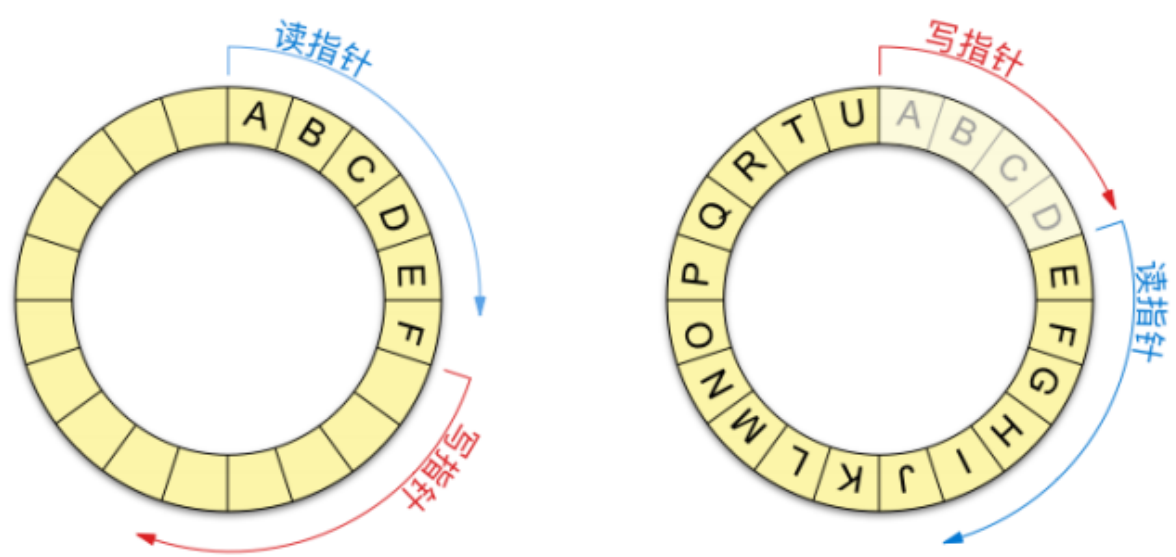


管道实现进程通信的原理

- 管道相当于一个中间媒介，共享数据



管道的数据结构



匿名管道

概念及使用

- 管道也叫无名(匿名)管道，它是 UNIX 系统 IPC (进程间通信) 的最古老形式，所有的 UNIX 系统都支持这种通信机制
- 统计一个目录中文件的数目命令: `ls | wc -l`，为了执行该命令，shell 创建了两个进程来分别执行 `ls` 和 `wc`



- 查看帮助: `man 2 pipe`
- 创建匿名管道: `int pipe(int pipefd[2]);`
- 查看管道缓冲大小命令: `ulimit -a` (共8个, 每个521byte, 即4k)


```

u@ubuntu:~/Desktop/Linux$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority     (-e) 0
file size               (blocks, -f) unlimited
pending signals         (-i) 15435
max locked memory       (kbytes, -l) 65536
max memory size         (kbytes, -m) unlimited
open files              (-n) 1048576
pipe size               (512 bytes, -p) 8
POSIX message queues    (bytes, -q) 819200
real-time priority      (-r) 0
stack size              (kbytes, -s) 8192
cpu time                (seconds, -t) unlimited
max user processes      (-u) 15435
virtual memory          (kbytes, -v) unlimited
file locks              (-x) unlimited
u@ubuntu:~/Desktop/Linux$

```

- 查看管道缓冲大小函数：`long fpathconf(int fd, int name);`

创建匿名管道

- `int pipe(int pipefd[2])`
 - 功能：创建一个匿名管道，用来进程间通信。
 - 参数：`int pipefd[2]` 这个数组是一个传出参数。
 - `pipefd[0]` 对应的是管道的读端
 - `pipefd[1]` 对应的是管道的写端
 - 返回值：成功 0，失败 -1
- 注意
 - 管道默认是阻塞的：如果管道中没有数据，read阻塞，如果管道满了，write阻塞
 - 匿名管道只能用于具有关系的进程之间的通信（父子进程，兄弟进程）
- 实现子进程发送数据给父进程，父进程读取到数据输出
 - 管道应在子进程创建前生成，否则父子进程不一定对应同一个管道
 - 单向发送时
 - 由于读写顺序不定，看起来像自己写自己读

```

child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015
child recv : hello,i am child, pid : 56015

```

- 解决方法：关闭不需要的端口（即代码中的 `close(pipefd[1]);`）

The diagram shows a vertical array of file descriptors (fd[0], fd[1], etc.). fd[0] is connected to a 'tty' device. fd[1] is also connected to a 'tty' device. fd[2] is connected to a 'tty' device. fd[3] is labeled 'fd[0]=3' and is connected to the '读端' (read end) of a '管道' (pipe). fd[4] is labeled 'fd[1]=4' and is connected to the '写端' (write end) of the same '管道'.

- [illegible]

- ```
/*
#include <unistd.h>
int pipe(int pipefd[2]);
 功能：创建一个匿名管道，用来进程间通信。
 参数：int pipefd[2] 这个数组是一个传出参数。
 pipefd[0] 对应的是管道的读端
 pipefd[1] 对应的是管道的写端
 返回值：
 成功 0
 失败 -1
*/
```

管道默认是阻塞的：如果管道中没有数据，**read**阻塞，如果管道满了，**write**阻塞

注意：匿名管道只能用于具有关系的进程之间的通信（父子进程，兄弟进程）

```
*/

// 子进程发送数据给父进程，父进程读取到数据输出
#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {

 // 在fork之前创建管道
 int pipefd[2];
 int ret = pipe(pipefd);
 if(ret == -1) {
 perror("pipe");
 exit(0);
 }

 // 创建子进程
 pid_t pid = fork();
 if(pid > 0) {
 // 父进程
 printf("i am parent process, pid : %d\n", getpid());

 // 关闭写端
 close(pipefd[1]);

 // 从管道的读取端读取数据
 char buf[1024] = {0};
 while(1) {
 int len = read(pipefd[0], buf, sizeof(buf));
 printf("parent recv : %s, pid : %d\n", buf, getpid());

 // 向管道中写入数据
 //char * str = "hello,i am parent";
 //write(pipefd[1], str, strlen(str));
 //sleep(1);
 }
 } else if(pid == 0){
 // 子进程
 printf("i am child process, pid : %d\n", getpid());
 // 关闭读端
 close(pipefd[0]);
 char buf[1024] = {0};
 while(1) {
 // 向管道中写入数据
 char * str = "hello,i am child";
 write(pipefd[1], str, strlen(str));
 sleep(1);

 // int len = read(pipefd[0], buf, sizeof(buf));
 }
 }
}
```

```

 // printf("child recv : %s, pid : %d\n", buf, getpid());
 // bzero(buf, 1024);
 }

}

return 0;
}

```

```

u@ubuntu:~/Desktop/Linux$ gcc pipe.c -o pipe
u@ubuntu:~/Desktop/Linux$./pipe
i am parent process, pid : 50483
i am child process, pid : 50484
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
parent recv : hello,i am child, pid : 50483
^C
u@ubuntu:~/Desktop/Linux$

```

## 实例：自建管道实现shell命令(ps aux)

- 思路
  - 子进程：实现 ps aux，子进程结束后，将数据发送给父进程
  - 父进程：获取到数据并打印
  - pipe()->fork()->execlp()<在此之前，输出为文件描述符重定向->->打印
- code

```

/*
 实现 ps aux | grep xxx 父子进程间通信

 子进程： ps aux，子进程结束后，将数据发送给父进程
 父进程：获取到数据，过滤

 pipe()
 execlp()
 子进程将标准输出 stdout_fileno 重定向到管道的写端。 dup2

*/

#include <unistd.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <wait.h>

int main() {

 // 创建一个管道
 int fd[2];
 int ret = pipe(fd);

```

```

if(ret == -1) {
 perror("pipe");
 exit(0);
}

// 创建子进程
pid_t pid = fork();

if(pid > 0) {
 // 父进程
 // 关闭写端，必须要有，否则程序不会结束
 close(fd[1]);
 // 从管道中读取
 char buf[1024] = {0};

 int len = -1;
 while((len = read(fd[0], buf, sizeof(buf) - 1)) > 0) {
 // 过滤数据输出
 printf("%s", buf);
 memset(buf, 0, 1024);
 }

 wait(NULL);
} else if(pid == 0) {
 // 子进程
 // 关闭读端
 close(fd[0]);

 // 文件描述符的重定向 stdout_fileno -> fd[1]
 dup2(fd[1], STDOUT_FILENO);
 // 执行 ps aux
 execvp("ps", "ps", "aux", NULL);
 perror("execvp");
 exit(0);
} else {
 perror("fork");
 exit(0);
}

return 0;
}

```

- 未解决: `./ipc | wc -c` 比 `ps aux | wc -c` 统计的进程数不同

## 设置管道非阻塞

```

int flags = fcntl(fd[0], F_GETFL); // 获取原来的flag
flags |= O_NONBLOCK; // 修改flag的值
fcntl(fd[0], F_SETFL, flags); // 设置新的flag

```

## 读写特点总结

- 读管道
  - 管道中有数据，read返回实际读到的字节数
  - 管道中无数据
    - 写端被全部关闭，read返回0（相当于读到文件的末尾）
    - 写端没有完全关闭，read阻塞等待
- 写管道
  - 管道读端全部被关闭，进程异常终止（进程收到 SIGPIPE 信号）
  - 管道读端没有全部关闭：
    - 管道已满，write阻塞
    - 管道没有满，write将数据写入，并返回实际写入的字节数

## 有名管道

### 概念及使用

- 匿名管道，由于没有名字，只能用于亲缘关系的进程间通信。为了克服这个缺点，提出了有名管道（FIFO），也叫命名管道、FIFO文件
- 有名管道（FIFO）不同于匿名管道之处在于它提供了一个路径名与之关联，以FIFO的文件形式存在于文件系统中，并且其打开方式与打开一个普通文件是一样的，这样即使与FIFO的创建进程不存在亲缘关系的进程，只要可以访问该路径，就能够彼此通过FIFO相互通信，因此，通过FIFO不相关的进程也能交换数据
- 一旦打开了FIFO，就能在它上面使用与操作匿名管道和其他文件的系统调用一样的I/O系统调用了（如read()、write()和close()）。与管道一样，FIFO也有一个写入端和读取端，并且从管道中读取数据的顺序与写入的顺序是一样的。FIFO的名称也由此而来：先入先出
- 有名管道（FIFO）和匿名管道（pipe）有一些特点是相同的，不一样的地方在于
  - FIFO在文件系统中作为一个特殊文件存在，但FIFO中的内容却存放在内存中
  - 当使用FIFO的进程退出后，FIFO文件将继续保存在文件系统中以便以后使用
  - FIFO有名字，不相关的进程可以通过打开有名管道进行通信
- 可使用man fifo查看帮助

### 创建有名管道

- shell命令创建：mkfifo 文件名，可通过man 1 mkfifo查看帮助

```
u@ubuntu:~/Desktop/Linux$ ll
total 12
drwxrwxr-x 2 u u 4096 Oct 3 16:06 ./
drwxr-xr-x 4 u u 4096 Oct 2 17:20 ../
-rwxrwxr-x 1 u u 95 Oct 2 18:52 delete.sh*
u@ubuntu:~/Desktop/Linux$ mkfifo test1
u@ubuntu:~/Desktop/Linux$ ll
total 12
drwxrwxr-x 2 u u 4096 Oct 3 16:06 ./
drwxr-xr-x 4 u u 4096 Oct 2 17:20 ../
-rwxrwxr-x 1 u u 95 Oct 2 18:52 delete.sh*
prw-rw-r-- 1 u u 0 Oct 3 16:06 test1|
u@ubuntu:~/Desktop/Linux$
```

- 函数创建: `int mkfifo(const char *pathname, mode_t mode);`, 可通过 `man 3 mkfifo` 查看帮助

```
#include <sys/types.h>
#include <sys/stat.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main()
{
 // 判断文件是否存在
 int ret = access("test", F_OK);
 // 不存在则创建
 if (ret == -1) {
 printf("管道不存在, 创建管道...\n");
 ret = mkfifo("test", 0664);
 if (ret == -1) {
 perror("mkfifo");
 exit(0);
 }
 }

 return 0;
}
```

```
u@ubuntu:~/Desktop/Linux$ ll
total 16
drwxrwxr-x 2 u u 4096 Oct 3 16:07 ./
drwxr-xr-x 4 u u 4096 Oct 2 17:20 ../
-rwxrwxr-x 1 u u 95 Oct 2 18:52 delete.sh*
-rw-rw-r-- 1 u u 411 Oct 3 16:14 mkfifo.c
u@ubuntu:~/Desktop/Linux$ gcc mkfifo.c -o mkfifo
u@ubuntu:~/Desktop/Linux$./mkfifo
管道不存在, 创建管道...
u@ubuntu:~/Desktop/Linux$ ll
total 28
drwxrwxr-x 2 u u 4096 Oct 3 16:14 ./
drwxr-xr-x 4 u u 4096 Oct 2 17:20 ../
-rwxrwxr-x 1 u u 95 Oct 2 18:52 delete.sh*
-rwxrwxr-x 1 u u 8472 Oct 3 16:14 mkfifo*
-rw-rw-r-- 1 u u 411 Oct 3 16:14 mkfifo.c
prw-rw-r-- 1 u u 0 Oct 3 16:14 test|
u@ubuntu:~/Desktop/Linux$
```

## 实例：两进程通过有名管道通信（单一发送）

- 写端

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>
```



```

// 向管道中写数据
int main()
{
 // 1.判断文件是否存在
 int ret = access("test", F_OK);
 if(ret == -1) {
 printf("管道不存在, 创建管道\n");

 // 2.创建管道文件
 ret = mkfifo("test", 0664);

 if(ret == -1) {
 perror("mkfifo");
 exit(0);
 }
 }

 // 3.以只写的方式打开管道
 int fd = open("test", O_WRONLY);
 if(fd == -1) {
 perror("open");
 exit(0);
 }

 // 写数据
 for(int i = 0; i < 100; i++) {
 char buf[1024];
 sprintf(buf, "hello, %d\n", i);
 printf("write data : %s\n", buf);
 write(fd, buf, strlen(buf));
 sleep(1);
 }

 close(fd);

 return 0;
}

```

- 读端

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>

// 从管道中读取数据
int main()
{
 // 1.打开管道文件
 int fd = open("test", O_RDONLY);
 if(fd == -1) {
 perror("open");
 }
}

```

```

 exit(0);
 }

 // 读数据
 while(1) {
 char buf[1024] = {0};
 // 这里不能写strlen(buf) 因为这里的含义是每次按固定长度读取，最开始
 strlen(buf)=0
 int len = read(fd, buf, sizeof(buf));
 if(len == 0) {
 printf("写端断开连接了...\n");
 break;
 }
 printf("recv buf : %s\n", buf);
 }

 close(fd);

 return 0;
}

```

- 运行

- 当写端开始写数据，但读端没有启动时，写端阻塞

|                                                               |                                          |
|---------------------------------------------------------------|------------------------------------------|
| <pre> u@ubuntu:~/Desktop/Linux\$ ./write 管道不存在，创建管道... </pre> | <pre> u@ubuntu:~/Desktop/Linux\$  </pre> |
|---------------------------------------------------------------|------------------------------------------|

- 当读端开始读数据，但写端没有启动时，读端阻塞

|                                          |                                                |
|------------------------------------------|------------------------------------------------|
| <pre> u@ubuntu:~/Desktop/Linux\$  </pre> | <pre> u@ubuntu:~/Desktop/Linux\$ ./read </pre> |
|------------------------------------------|------------------------------------------------|

- 两端都启动时，正常输出（无关哪个先启动）

|                                                                                                 |                                                                                              |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| <pre> u@ubuntu:~/Desktop/Linux\$ ./write buf : hello, this is 0  buf : hello, this is 1  </pre> | <pre> u@ubuntu:~/Desktop/Linux\$ ./read recv hello, this is 0  recv hello, this is 1  </pre> |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|

- 先关闭读端

|                                                                                                                                                     |                                                                                                                             |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|
| <pre> u@ubuntu:~/Desktop/Linux\$ ./write buf : hello, this is 0  buf : hello, this is 1  buf : hello, this is 2  u@ubuntu:~/Desktop/Linux\$  </pre> | <pre> u@ubuntu:~/Desktop/Linux\$ ./read recv hello, this is 0  recv hello, this is 1  ^C u@ubuntu:~/Desktop/Linux\$  </pre> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------|

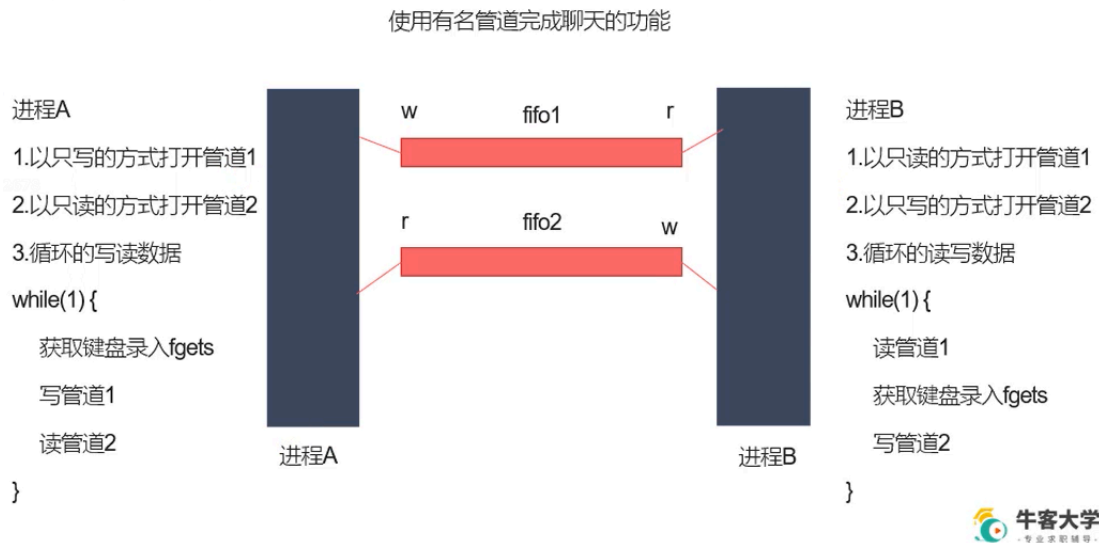
- 先关闭写端

|                                                                                                                                                        |                                                                                                                                                            |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre> u@ubuntu:~/Desktop/Linux\$ ./write buf : hello, this is 0  buf : hello, this is 1  buf : hello, this is 2  ^C u@ubuntu:~/Desktop/Linux\$  </pre> | <pre> u@ubuntu:~/Desktop/Linux\$ ./read recv hello, this is 0  recv hello, this is 1  recv hello, this is 2  写端断开连接了... u@ubuntu:~/Desktop/Linux\$  </pre> |
|--------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 实例：简易版聊天功能（连续发送）

- 功能：两个进程相互发送数据及接收数据，能够连续发送及接收
- 思路
  - 由于两个进程并没有亲缘关系，所以只能使用有名管道实现

- 需要两个管道
  - 一个管道用于进程A的写与进程B的读
  - 一个管道用于进程B的写与进程A的读
- 需要父子进程，实现连续发送及接收
  - 父进程负责写入数据到管道
  - 子进程负责从管道读取数据
- 流程（不包含父子进程，即下图所示流程不能实现连续发送功能）



- 进程A

```
/*
chatA
1. 读、写数据分开，用两个管道
 1. fifo1用于进程A写及进程B读
 2. fifo2用于进程B写及进程A读
2. 连续发送及接收信息，使用两个进程
 1. 父进程用于写数据
 2. 子进程用于读数据
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main()
{
 // 判断写管道是否存在，不存在则创建
 int ret = access("fifo1", F_OK);
 if (ret == -1) {
 printf("fifo1不存在，创建...\n");
 ret = mkfifo("fifo1", 0664);
 if (ret == -1) {
 perror("mkfifo");
 exit(-1);
 }
 }
}
```

```

 }
}

// 判断读管道是否存在，不存在则创建
ret = access("fifo2", F_OK);
if (ret == -1) {
 printf("fifo2不存在, 创建...\n");
 ret = mkfifo("fifo2", 0664);
 if (ret == -1) {
 perror("mkfifo");
 exit(-1);
 }
}

// 创建进程
pid_t pid = fork();
char buf[1024];
if (pid > 0) {
 // 父进程
 // 打开写管道
 // 打开一次，否则系统可能会崩
 int fdw = open("fifo1", O_WRONLY);
 while (1) {
 // 从键盘读取输入
 printf("[chata]please input: \n");
 fgets(buf, sizeof(buf), stdin);
 write(fdw, buf, strlen(buf));
 // 清空数组
 memset(buf, 0, sizeof(buf));
 }
 close(fdw);
} else if (pid == 0) {
 // 子进程
 // 打开读管道
 // 打开一次，否则系统可能会崩
 int fdr = open("fifo2", O_RDONLY);
 while (1) {
 char buf[1024];
 int len = read(fdr, buf, sizeof(buf));
 if (len == 0) {
 printf("[chata]写端断开连接了...\n");
 break;
 }
 printf("[chata]recv : %s", buf);
 // 清空数组
 memset(buf, 0, sizeof(buf));
 }
 close(fdr);
} else {
 perror("fork");
 exit(-2);
}

return 0;
}

```

- 进程B

```

/*
chatB
1. 读、写数据分开，用两个管道
 1. fifo1用于进程A写及进程B读
 2. fifo2用于进程B写及进程A读
2. 连续发送及接收信息，使用两个进程
 1. 父进程用于写数据
 2. 子进程用于读数据
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main()
{
 // 判断写管道是否存在，不存在则创建
 int ret = access("fifo1", F_OK);
 if (ret == -1) {
 printf("fifo1不存在，创建...\n");
 ret = mkfifo("fifo1", 0664);
 if (ret == -1) {
 perror("mkfifo");
 exit(-1);
 }
 }

 // 判断读管道是否存在，不存在则创建
 ret = access("fifo2", F_OK);
 if (ret == -1) {
 printf("fifo2不存在，创建...\n");
 ret = mkfifo("fifo2", 0664);
 if (ret == -1) {
 perror("mkfifo");
 exit(-1);
 }
 }

 // 创建进程
 pid_t pid = fork();
 char buf[1024] = { 0 };
 if (pid > 0) {
 // 父进程
 // 打开写管道
 // 打开一次，否则系统可能会崩
 int fdw = open("fifo2", O_WRONLY);
 while (1) {
 // 从键盘读取输入
 printf("[chatB]please input: \n");
 fgets(buf, sizeof(buf), stdin);
 write(fdw, buf, strlen(buf));
 // 清空数组

```

```

 memset(buf, 0, sizeof(buf));
 }
 close(fdw);
} else if (pid == 0) {
 // 子进程
 // 打开读管道
 // 打开一次，否则系统可能会崩
 int fdr = open("fifo1", O_RDONLY);
 while (1) {
 char buf[1024];
 int len = read(fdr, buf, sizeof(buf));
 if (len == 0) {
 printf("[chatB] 写端断开连接了...\n");
 break;
 }
 printf("[chatB]recv : %s", buf);
 // 清空数组
 memset(buf, 0, sizeof(buf));
 }
 close(fdr);
} else {
 perror("fork");
 exit(-2);
}

return 0;
}

```

- 运行结果

```

u@ubuntu:~/Desktop/Linux$./a
fifo1不存在, 创建...
fifo2不存在, 创建...
[chatA]please input:
123
[chatA]please input:
[chatA]recv : abc
111
[chatA]please input:
222
[chatA]please input:
333
[chatA]please input:
[chatA] 写端断开连接了...
^C

u@ubuntu:~/Desktop/Linux$./b
[chatB]please input:
[chatB]recv : 123
abc
[chatB]please input:
[chatB]recv : 111
[chatB]recv : 222
[chatB]recv : 333
^C
u@ubuntu:~/Desktop/Linux$

```

- 存在的问题:

- 乱码
- 一个进程结束后，另一个还未结束，需要手动关闭

## 读写特点总结

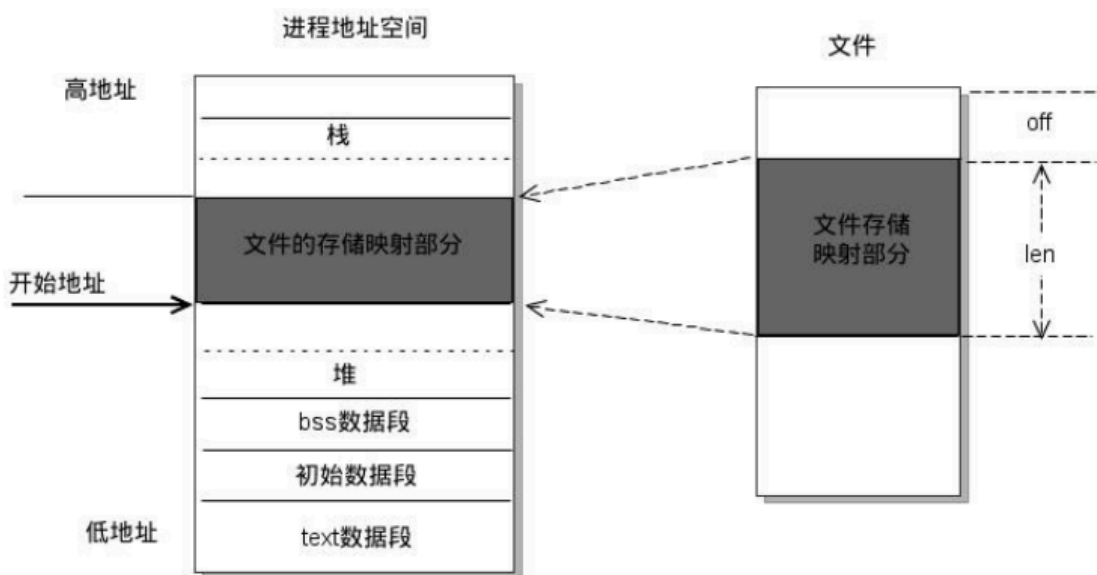
- 读管道
  - 管道中有数据，`read` 返回实际读到的字节数
  - 管道中无数据:
    - 管道写端被全部关闭，`read` 返回0，（相当于读到文件末尾）
    - 写端没有全部被关闭，`read` 阻塞等待
- 写管道
  - 管道读端被全部关闭，进行异常终止（收到一个 `SIGPIPE` 信号）

- 管道读端没有全部关闭：
  - 管道已经满了，`write` 会阻塞
  - 管道没有满，`write` 将数据写入，并返回实际写入的字节数

## 内存映射

### 概念

- 内存映射（Memory-mapped I/O）是将**磁盘文件的数据映射到内存**，用户通过修改内存就能修改磁盘文件



- 内存映射相关系统调用，使用 `man 2 mmap` 查看帮助
  - `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
    - 功能：将一个文件或者设备的数据映射到内存中
    - 参数
      - `addr`：设置为 `NULL` 时，由内核指定（推荐做法）
      - `length`：要映射的数据的长度，这个值**不能为0**。**建议使用文件的长度**，获取文件的长度：`stat`，`lseek`
      - `prot`：对申请的内存映射区的操作权限
        - `PROT_EXEC`：可执行的权限
        - `PROT_READ`：读权限
        - `PROT_WRITE`：写权限
        - `PROT_NONE`：没有权限
      - `flags`
        - `MAP_SHARED`：映射区的数据会自动和磁盘文件进行同步，进程间通信，必须要设置这个选项
        - `MAP_PRIVATE`：不同步，内存映射区的数据改变了，对原来的文件不会修改，会重新创建一个新的文件。（`copy on write`）
      - `fd`：需要映射的那个文件的文件描述符，通过 `open` 得到，`open` 的是一个磁盘文件

- `offset`：偏移量，一般进行特殊指定（指定为0即可），如果使用必须指定的是4k 的整数倍，0表示不偏移
- 返回值：返回创建的内存的首地址。失败返回 `MAP_FAILED`(即 `(void *) -1`)
- `int munmap(void *addr, size_t length);`
  - 功能：释放内存映射
  - 参数
    - `addr`：要释放的内存的首地址
    - `length`：要释放的内存的大小，要和 `mmap` 函数中的 `length` 参数的值一样

## 进程间通信种类

- 有关系的进程（父子进程）
  - 还没有子进程的时候，通过唯一的父进程，先创建内存映射区
  - 有了内存映射区以后，创建子进程
  - 父子进程共享创建的内存映射区
- 没有关系的进程间通信
  - 准备一个大小不是0的磁盘文件
  - 进程1 通过磁盘文件创建内存映射区，得到一个操作这块内存的指针
  - 进程2 通过磁盘文件创建内存映射区，得到一个操作这块内存的指针
  - 使用内存映射区通信

## 注意事项

- 要操作映射内存，**必须要有读的权限**，即权限为 `PROT_READ` 或 `PROT_READ | PROT_WRITE`
- 在使用**内存映射**通信时，使用文件的大小不能为0，**`open` 指定的权限不能和 `prot` 参数有冲突**

| <code>prot</code>                   | <code>open</code>                           |
|-------------------------------------|---------------------------------------------|
| <code>PROT_READ</code>              | <code>O_RDONLY</code> 或 <code>O_RDWR</code> |
| <code>PROT_READ   PROT_WRITE</code> | <code>O_RDWR</code>                         |

- 内存映射区通信，是非阻塞
- 一个文件对应一个内存映射区
- 如果对 `mmap` 的返回值 (`ptr`) 做 `++` 操作 (`ptr++`)，`munmap` 是否能够成功？
  - 不能成功，因为回收资源时，需要传递指针，如果变化，将会回收失败
- 如果 `open` 时 `O_RDONLY`，`mmap` 时 `prot` 参数指定 `PROT_READ | PROT_WRITE` 会怎样？
  - 错误，返回 `MAP_FAILED`，`open()` 函数中的权限建议和 `prot` 参数的权限保持一致
- 如果文件偏移量为1000会怎样？
  - 偏移量必须是 4K 的整数倍，返回 `MAP_FAILED`
- `mmap` 什么情况下会调用失败？
  - 第二个参数：`length = 0`
  - 第三个参数：`prot`
    - 只指定写权限



- `prot` 和 `open()` 两者的权限不匹配
- 可以`open`的时候 `O_CREAT` 一个新文件来创建映射区吗?
  - 可以的，但是创建的文件的大小如果为0的话，肯定不行(因为 `mmap` 调用时，长度不允许为0)
- `mmap` 后关闭文件描述符，对 `mmap` 映射有没有影响?
  - 映射区还存在，创建映射区的 `fd` 被关闭，没有任何影响
- 对 `ptr` 越界操作会怎样?
  - 越界操作操作的是非法的内存 -> 段错误

## 实例：父子进程通信

- 思路
  1. 打开指定文件并获取文件长度
  2. 创建内存映射区
  3. 父子进程功能，父进程负责收数据，子进程负责发数据
  4. 回收资源
- code

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <wait.h>
#include <string.h>
#include <stdlib.h>

int main()
{
 // 打开指定文件
 int fd = open("ipc.txt", O_RDWR);
 // 获取给定文件长度
 int size = lseek(fd, 0, SEEK_END);
 // 创建内存映射区
 void* ptr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
 // 判断是否成功
 if (ptr == MAP_FAILED) {
 perror("mmap");
 exit(-1);
 }
 // 创建子进程
 pid_t pid = fork();
 if (pid > 0) {
 // 父进程，用于读取数据
 // 回收子进程
 wait(NULL);
 // 接收数据并打印
 char buf[64];
 // 类型需要强转
 strcpy(buf, (char *)ptr);
 printf("recv : %s\n", buf);
 }
}
```

```

} else if (pid == 0) {
 // 子进程，用于发送数据
 // 类型需要强转
 strcpy((char *)ptr, "hello, i am child process");
} else {
 perror("fork");
 exit(-1);
}

// 关闭内存映射区
munmap(ptr, size);
// 关闭文件
close(fd);
return 0;
}

```

- 注意：程序执行后，文件大小不改变，那么子进程写入的数据会被截断，原因未知

- 执行前

```

u@ubuntu:~/Desktop/Linux$ ll
total 20
drwxrwxr-x 2 u u 4096 Oct 3 20:11 ./
drwxr-xr-x 4 u u 4096 Oct 3 04:52 ../
-rwxrwxr-x 1 u u 70 Oct 3 04:19 delete.sh*
-rw-rw-r-- 1 u u 5 Oct 3 20:11 ipc.txt
-rw-rw-r-- 1 u u 1116 Oct 3 20:07 mmap-parent-child-ipc.c
u@ubuntu:~/Desktop/Linux$ cat ipc.txt
1111
u@ubuntu:~/Desktop/Linux$

```

- 执行后

```

u@ubuntu:~/Desktop/Linux$ gcc mmap-parent-child-ipc.c -o ipc
u@ubuntu:~/Desktop/Linux$./ipc
recv : hello, i am child process
u@ubuntu:~/Desktop/Linux$ ll
total 32
drwxrwxr-x 2 u u 4096 Oct 3 20:12 ./
drwxr-xr-x 4 u u 4096 Oct 3 04:52 ../
-rwxrwxr-x 1 u u 70 Oct 3 04:19 delete.sh*
-rwxrwxr-x 1 u u 8752 Oct 3 20:12 ipc*
-rw-rw-r-- 1 u u 5 Oct 3 20:12 ipc.txt
-rw-rw-r-- 1 u u 1116 Oct 3 20:07 mmap-parent-child-ipc.c
u@ubuntu:~/Desktop/Linux$ cat ipc.txt
hellou@ubuntu:~/Desktop/Linux$

```

## 实例：文件拷贝

- 思路
  1. 需要两个文件，一个是有内容的文件（待拷贝文件），一个是空文件
  2. 由于有两个文件，需要两个内存映射区
  3. 然后将文件A的内存映射区内容拷贝给文件B的内存映射区
  4. 回收资源
- code

```

#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <string.h>
#include <stdlib.h>

int main()
{
 // 打开源文件，获取文件长度并创建对应内存映射区
 int fdSource = open("source.txt", O_RDONLY);
 int len = lseek(fdSource, 0, SEEK_END);
 void *ptrSource = mmap(NULL, len, PROT_READ, MAP_SHARED, fdSource, 0);
 if (ptrSource == MAP_FAILED) {
 perror("mmap");
 exit(-1);
 }

 // 打开目标文件，并创建对应内存映射区
 int fdTarget = open("target.txt", O_RDWR | O_CREAT, 0664);
 // 由于目标文件是通过创建得到，所以需要扩展长度与源文件保持一致
 truncate("target.txt", len);
 // 如果不加，扩展可能失败（保险起见）
 write(fdTarget, " ", 1);
 void *ptrTarget = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED,
fdTarget, 0);
 if (ptrTarget == MAP_FAILED) {
 perror("mmap");
 exit(-1);
 }

 // 内存拷贝
 memcpy(ptrTarget, ptrSource, len);

 // 回收资源
 close(fdTarget);
 close(fdSource);
 munmap(ptrTarget, len);
 munmap(ptrSource, len);

 return 0;
}

```

- output
  - 执行前

```

u@ubuntu:~/Desktop/Linux$ ll
total 20
drwxrwxr-x 2 u u 4096 Oct 3 23:42 ./
drwxr-xr-x 4 u u 4096 Oct 3 04:52 ../
-rw-rw-r-- 1 u u 1209 Oct 3 23:41 copy-file.c
-rwxrwxr-x 1 u u 70 Oct 3 04:19 delete.sh*
-rw-rw-r-- 1 u u 24 Oct 3 23:24 source.txt
u@ubuntu:~/Desktop/Linux$

```

- 执行后

```
u@ubuntu:~/Desktop/Linux$ gcc copy-file.c -o copy
u@ubuntu:~/Desktop/Linux$./copy
u@ubuntu:~/Desktop/Linux$ ll
total 36
drwxrwxr-x 2 u u 4096 Oct 3 23:43 ./
drwxr-xr-x 4 u u 4096 Oct 3 04:52 ../
-rwxrwxr-x 1 u u 8696 Oct 3 23:43 copy*
-rw-rw-r-- 1 u u 1209 Oct 3 23:41 copy-file.c
-rwxrwxr-x 1 u u 70 Oct 3 04:19 delete.sh*
-rw-rw-r-- 1 u u 24 Oct 3 23:24 source.txt
-rw-rw-r-- 1 u u 24 Oct 3 23:43 target.txt
u@ubuntu:~/Desktop/Linux$
```

## 实例：匿名内存映射

- 思路
  1. 匿名内存映射不存在文件实体，那么只能通过父子进程实现
  2. 父子进程操作同一块区域，重点在于内存映射区在创建时新增flags参数 `MAP_ANONYMOUS`
  3. 父进程读，子进程写
- code

```
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <sys/wait.h>

int main()
{
 void *ptr = mmap(NULL, 128, PROT_READ | PROT_WRITE, MAP_SHARED |
MAP_ANONYMOUS, -1, 0);
 if (ptr == MAP_FAILED) {
 perror("mmap");
 exit(-1);
 }
 pid_t pid = fork();
 if (pid > 0) {
 // 父进程
 wait(NULL);
 char buf[128];
 strcpy(buf, (char*)ptr);
 printf("recv : %s\n", buf);
 } else if (pid == 0) {
 // 子进程
 strcpy((char*)ptr, "i am a message");
 } else {
 perror("fork");
 exit(-1);
 }
}
```

```
// 释放资源
munmap(ptr, 128);
return 0;
}
```

- output

```
u@ubuntu:~/Desktop/Linux$ gcc anon-ipc.c -o anon
u@ubuntu:~/Desktop/Linux$./anon
recv : i am a message
u@ubuntu:~/Desktop/Linux$ ls
anon anon-ipc.c delete.sh
u@ubuntu:~/Desktop/Linux$
```

# 进程间通信之信号

## 说明

本部分笔记及源码出自 `slide/02Linux多进程开发/07 进程间通信之信号`

## 基本概念

- 信号是 Linux 进程间通信的最古老的方式之一，是事件发生时对进程的通知机制，有时也称之为软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件
- 发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下
  - 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入 `Ctrl+C` 通常会给进程发送一个中断信号
  - 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令，诸如被 0 除，或者引用了无法访问的内存区域
  - 系统状态变化，比如 alarm 定时器到期将引起 `SIGALRM` 信号，进程执行的 CPU 时间超限，或者该进程的某个子进程退出
  - 运行 kill 命令或调用 kill 函数
- 使用信号的两个主要目的是
  - 让进程知道已经发生了一个特定的事情
  - 强迫进程执行它自己代码中的信号处理程序
- 信号的特点
  - 简单
  - 不能携带大量信息
  - 满足某个特定条件才发送
  - 优先级比较高
- 查看系统定义的信号列表：`kill -l`，前 31 个信号为常规信号，其余为实时信号

```
u@ubuntu:~/Desktop/Linux$ 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

u@ubuntu:~/Desktop/Linux$
```

## 信号一览表及特点

- 可通过 `man 7 signal` 查看帮助
- 信号的 5 中默认处理动作
  - `Term`：终止进程
  - `Ign`：当前进程忽略掉这个信号
  - `Core`：终止进程，并生成一个Core文件
  - `Stop`：暂停当前进程
  - `Cont`：继续执行当前被暂停的进程
- 信号的几种状态：`产生`、`未决`、`递达`
- `SIGKILL` 和 `SIGSTOP` 信号不能被捕捉、阻塞或者忽略，只能执行默认动作
- 红色标记代表需要熟练掌握

| 编号 | 信号名称           | 对应事件                                            | 默认动作          |
|----|----------------|-------------------------------------------------|---------------|
| 1  | SIGHUP         | 用户退出shell时，由该shell启动的所有进程将收到这个信号                | 终止进程          |
| 2  | <b>SIGINT</b>  | 当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号    | 终止进程          |
| 3  | <b>SIGQUIT</b> | 用户按下<Ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号 | 终止进程          |
| 4  | SIGILL         | CPU检测到某进程执行了非法指令                                | 终止进程并产生core文件 |
| 5  | SIGTRAP        | 该信号由断点指令或其他 <code>trap</code> 指令产生              | 终止进程并产生core文件 |
| 6  | SIGABRT        | 调用abort函数时产生该信号                                 | 终止进程并产生core文件 |
| 7  | SIGBUS         | 非法访问内存地址，包括内存对齐出错                               | 终止进程并产生core文件 |
| 8  | SIGFPE         | 在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误     | 终止进程并产生core文件 |

| 编号 | 信号名称           | 对应事件                                                                 | 默认动作          |
|----|----------------|----------------------------------------------------------------------|---------------|
| 9  | <b>SIGKILL</b> | 无条件终止进程。该信号不能被忽略，处理和阻塞                                               | 终止进程，可以杀死任何进程 |
| 10 | SIGUSE1        | 用户定义的信号。即程序员可以在程序中定义并使用该信号                                           | 终止进程          |
| 11 | <b>SIGSEGV</b> | 指示进程进行了无效内存访问（段错误）                                                   | 终止进程并产生core文件 |
| 12 | SIGUSR2        | 另外一个用户自定义信号，程序员可以在程序中定义并使用该信号                                        | 终止进程          |
| 13 | <b>SIGPIPE</b> | Broken pipe向一个没有读端的管道写数据                                             | 终止进程          |
| 14 | SIGALRM        | 定时器超时，超时的时间 由系统调用alarm设置                                             | 终止进程          |
| 15 | SIGTERM        | 程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号 | 终止进程          |
| 16 | SIGSTKFLT      | Linux早期版本出现的信号，现仍保留向后兼容                                              | 终止进程          |

| 编号 | 信号名称           | 对应事件                                           | 默认动作   |
|----|----------------|------------------------------------------------|--------|
| 17 | <b>SIGCHLD</b> | 子进程结束时，父进程会收到这个信号                              | 忽略这个信号 |
| 18 | <b>SIGCONT</b> | 如果进程已停止，则使其继续运行                                | 继续/忽略  |
| 19 | <b>SIGSTOP</b> | 停止进程的执行。信号不能被忽略，处理和阻塞                          | 为终止进程  |
| 20 | SIGTSTP        | 停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号               | 暂停进程   |
| 21 | SIGTTIN        | 后台进程读终端控制台                                     | 暂停进程   |
| 22 | SIGTTOU        | 该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生                 | 暂停进程   |
| 23 | SIGURG         | 套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达 | 忽略该信号  |
| 24 | SIGXCPU        | 进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并发送给该进程           | 终止进程   |

| 编号            | 信号名称                      | 对应事件                                          | 默认动作          |
|---------------|---------------------------|-----------------------------------------------|---------------|
| 25            | SIGXFSZ                   | 超过文件的最大长度设置                                   | 终止进程          |
| 26            | SIGVTALRM                 | 虚拟时钟超时时产生该信号。类似于SIGALRM，但是该信号只计算该进程占用CPU的使用时间 | 终止进程          |
| 27            | SGIPROF                   | 类似于SIGVTALRM，它不公包括该进程占用CPU时间还包括执行系统调用时间       | 终止进程          |
| 28            | SIGWINCH                  | 窗口变化大小时发出                                     | 忽略该信号         |
| 29            | SIGIO                     | 此信号向进程指示发出了一个异步IO事件                           | 忽略该信号         |
| 30            | SIGPWR                    | 关机                                            | 终止进程          |
| 31            | SIGSYS                    | 无效的系统调用                                       | 终止进程并产生core文件 |
| 34<br>~<br>64 | SIGRTMIN<br>~<br>SIGRTMAX | LINUX的实时信号，它们没有固定的含义（可以由用户自定义）                | 终止进程          |

# 信号相关的函数

## core文件生成及调试

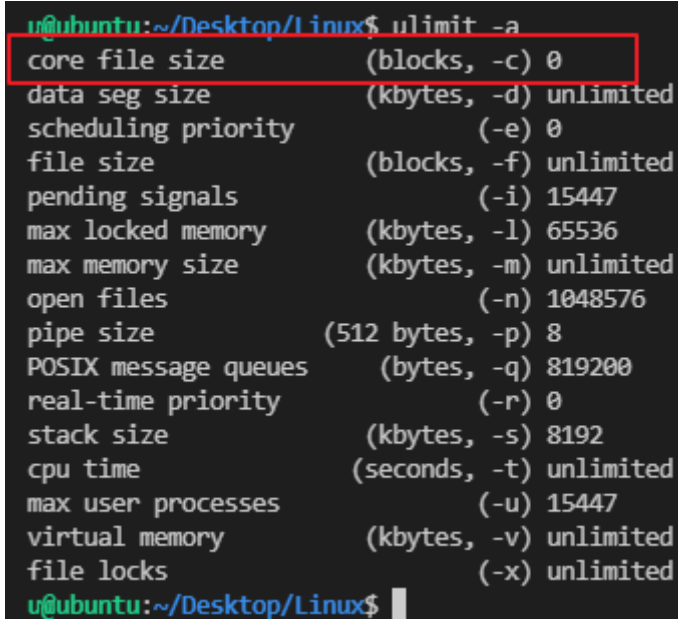
- 当进程异常终止时，会生成 `core` 文件（需要进行相应设置），可以通过 `gdb` 调试查看错误，调试以下程序
- `code`

```
#include <stdio.h>
#include <string.h>

int main()
{
 char* buf;
 strcpy(buf, "core test");
 return 0;
}
```

- 生成调试 `core` 文件需要做以下几步

1. 使用 `ulimit -a` 查看资源上限



```
u@ubuntu:~/Desktop/Linux$ ulimit -a
core file size (blocks, -c) 0
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 15447
max locked memory (kbytes, -l) 65536
max memory size (kbytes, -m) unlimited
open files (-n) 1048576
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 8192
cpu time (seconds, -t) unlimited
max user processes (-u) 15447
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
u@ubuntu:~/Desktop/Linux$
```

2. 修改 `core size`: `ulimit -c core-size`



```

u@ubuntu:~/Desktop/Linux$ ulimit -c 1024
u@ubuntu:~/Desktop/Linux$ ulimit -a
core file size (blocks, -c) 1024
data seg size (kbytes, -d) unlimited
scheduling priority (-e) 0
file size (blocks, -f) unlimited
pending signals (-i) 15447
max locked memory (kbytes, -l) 65536
max memory size (kbytes, -m) unlimited
open files (-n) 1048576
pipe size (512 bytes, -p) 8
POSIX message queues (bytes, -q) 819200
real-time priority (-r) 0
stack size (kbytes, -s) 8192
cpu time (seconds, -t) unlimited
max user processes (-u) 15447
virtual memory (kbytes, -v) unlimited
file locks (-x) unlimited
u@ubuntu:~/Desktop/Linux$

```

3. 在编译运行程序时加上 `-g` 选项使得能够被 `gdb` 调试，运行后生成 `core` 文件

```

u@ubuntu:~/Desktop/Linux$ ls
core_test.c delete.sh
u@ubuntu:~/Desktop/Linux$ gcc core_test.c -o test -g
u@ubuntu:~/Desktop/Linux$ ls
core_test.c delete.sh test
u@ubuntu:~/Desktop/Linux$./test
Segmentation fault (core dumped)
u@ubuntu:~/Desktop/Linux$ ls
core core_test.c delete.sh test
u@ubuntu:~/Desktop/Linux$

```

4. 调试 `core` 程序： `gdb test` 进入 `gdb` 终端，使用 `core-file core` 可以查看 `core` 定位错误

```

(gdb) core-file core
[New LWP 8197]
warning: Unexpected size of section `.reg-xstate/8197' in core file.
Core was generated by `./test'.
Program terminated with signal SIGSEGV, Segmentation fault.
warning: Unexpected size of section `.reg-xstate/8197' in core file.
#0 0x0000555c9761d60c in main () at core_test.c:7
7 strcpy(buf, "core test");
(gdb)

```

## kill & raise & abort

- `int kill(pid_t pid, int sig);`
  - 使用 `man 2 kill` 查看帮助
  - 功能：给任何的进程或者进程组 `pid`，发送任何的信号 `sig`
  - 参数
    - `pid`
      - `> 0`：将信号发送给指定的进程
      - `= 0`：将信号发送给当前的进程组
      - `= -1`：将信号发送给每一个有权接收这个信号的进程

- `< -1` : 这个 `pid`=某个进程组的ID取反
- `sig` : 需要发送的信号编号或者是宏值, 0表示不发送任何信号
  - 返回值: 0成功, -1失败
- `int raise(int sig);`
  - 使用 `man 3 raise` 查看帮助
  - 功能: 给**当前进程**发送信号
  - 参数: `sig` : 要发送的信号
  - 返回值: 0成功, 非0失败
- `void abort(void);`
  - 使用 `man 3 abort` 查看帮助
  - 功能: 发送 `SIGABRT` 信号给当前的进程, **杀死当前进程**

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <unistd.h>

int main()
{
 pid_t pid = fork();

 if(pid == 0) {
 // 子进程
 int i = 0;
 for(i = 0; i < 5; i++) {
 printf("child process\n");
 sleep(1);
 }

 } else if(pid > 0) {
 // 父进程
 printf("parent process\n");
 sleep(2);
 printf("kill child process now\n");
 kill(pid, SIGINT);
 }

 return 0;
}
```

```
u@ubuntu:~/Desktop/Linux$ gcc kill.c -o kill
u@ubuntu:~/Desktop/Linux$./kill
parent process
child process
child process
kill child process now
u@ubuntu:~/Desktop/Linux$
```

## alarm & setitimer

- 区别: `alarm` 只能定一次时, `setitimer` 可以周期性定时
- `unsigned int alarm(unsigned int seconds);`
  - 使用 `man 2 alarm` 查看帮助
  - 功能: 设置定时器 (闹钟)。函数调用, 开始倒计时, 当倒计时为0的时候, 函数会给当前的进程发送一个信号: `SIGALARM`
  - 参数: `seconds`, 倒计时的时长, 单位: 秒。如果参数为0, 定时器无效 (不进行倒计时, 不发信号)
  - 取消一个定时器, 通过 `alarm(0)`
  - 返回值
    - 之前没有定时器, 返回0
    - 之前有定时器, 返回之前的定时器剩余的时间
- `SIGALARM`: 默认终止**当前的进程**, 每一个进程都有且只有唯一的一个定时器
- 定时器, 与进程的状态无关 (自然定时法)。无论进程处于什么状态, `alarm`都会计时, 即**函数不阻塞**

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

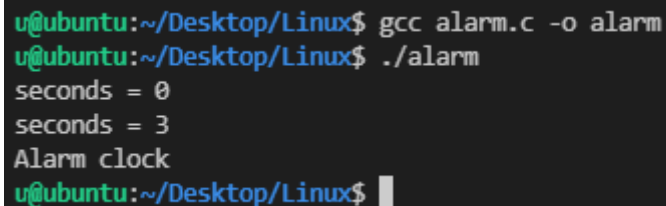
int main() {

 int seconds = alarm(5);
 printf("seconds = %d\n", seconds); // 0

 sleep(2);
 seconds = alarm(2); // 不阻塞
 printf("seconds = %d\n", seconds); // 3

 while(1) {
 }

 return 0;
}
```



```
u@ubuntu:~/Desktop/Linux$ gcc alarm.c -o alarm
u@ubuntu:~/Desktop/Linux$./alarm
seconds = 0
seconds = 3
Alarm clock
u@ubuntu:~/Desktop/Linux$
```

- `int setitimer(int which, const struct itimerval *new_val, struct itimerval *old_value);`
  - 使用 `man 2 setitimer` 查看帮助
  - 功能: 设置定时器 (闹钟)。可以替代`alarm`函数。精度微妙`us`, 可以实现周期性定时
  - 参数
    - `which`: 定时器以什么时间计时

- `ITIMER_REAL`: 真实时间, 时间到达, 发送 `SIGALRM` (常用)
- `ITIMER_VIRTUAL`: 用户时间, 时间到达, 发送 `SIGVTALRM`
- `ITIMER_PROF`: 以该进程在用户态和内核态下所消耗的时间来计算, 时间到达, 发送 `SIGPROF`
- `new_value`: 设置定时器的属性
- `old_value`: 记录上一次的定时的时间参数, 一般不使用, 指定 `NULL`
- 返回值: 成功 0, 失败 -1 并设置错误号
- `struct itimerval`

```
struct itimerval { // 定时器的结构体
 struct timeval it_interval; // 每个阶段的时间, 间隔时间
 struct timeval it_value; // 延迟多长时间执行定时器
};

struct timeval { // 时间的结构体
 time_t tv_sec; // 秒数
 suseconds_t tv_usec; // 微秒
};

// 过it_value秒后, 每隔it_interval秒定时一次
```

- 实现过3秒以后, 每隔2秒钟定时一次=>因为没有信号捕捉, 所以还没有实现这样的效果

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>

// 过3秒以后, 每隔2秒钟定时一次
int main()
{
 struct itimerval new_value;

 // 设置间隔的时间
 new_value.it_interval.tv_sec = 2;
 new_value.it_interval.tv_usec = 0;

 // 设置延迟的时间, 3秒之后开始第一次定时
 new_value.it_value.tv_sec = 3;
 new_value.it_value.tv_usec = 0;

 int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
 printf("定时器开始了...\n");

 if(ret == -1) {
 perror("setitimer");
 exit(0);
 }

 getchar();

 return 0;
```

```
}
```

```
u@ubuntu:~/Desktop/Linux$ gcc setitimer.c -o timer
u@ubuntu:~/Desktop/Linux$./timer
定时器开始了...
Alarm clock
u@ubuntu:~/Desktop/Linux$
```

## 信号捕捉函数

### signal

- `sighandler_t signal(int signum, sighandler_t handler);`
  - 使用 `man 2 signal` 查看帮助
  - 功能：设置某个信号的捕捉行为
  - 参数
    - `signum`：要捕捉的信号
    - `handler`：捕捉到信号要如何处理
      - `SIG_IGN`：忽略信号
      - `SIG_DFL`：使用信号默认的行为
      - 自定义回调函数
    - 返回值
      - 成功，返回上一次注册的信号处理函数的地址。第一次调用返回NULL
      - 失败，返回SIG\_ERR，设置错误号
      - 注意：返回值定义在宏 `__USE_GNU` 中，需要指定或者直接在程序中使用 `typedef __sighandler_t sighandler_t;`
    - `SIGKILL` 和 `SIGSTOP` 不能被捕捉，不能被忽略
- 完善过3秒以后，每隔2秒钟定时一次的定时器功能

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
 printf("捕捉到了信号的编号是: %d\n", num);
 printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main()
{

 // 注册信号捕捉
 // signal(SIGALRM, SIG_IGN);
 // signal(SIGALRM, SIG_DFL);
 // void (*sighandler_t)(int); 函数指针，int类型的参数表示捕捉到的信号的值
 // 捕捉的信号右定时器发出
```

```

signal(SIGALRM, myalarm);

struct itimerval new_value;

// 设置间隔的时间
new_value.it_interval.tv_sec = 2;
new_value.it_interval.tv_usec = 0;

// 设置延迟的时间,3秒之后开始第一次定时
new_value.it_value.tv_sec = 3;
new_value.it_value.tv_usec = 0;

int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
printf("定时器开始了...\n");

if(ret == -1) {
 perror("setitimer");
 exit(0);
}

getchar();

return 0;
}

```

```

u@ubuntu:~/Desktop/Linux$ gcc setitimer.c -o timer
u@ubuntu:~/Desktop/Linux$./timer
定时器开始了...
捕捉到了信号的编号是: 14
xxxxxxx
捕捉到了信号的编号是: 14
xxxxxxx
捕捉到了信号的编号是: 14
xxxxxxx
^C
u@ubuntu:~/Desktop/Linux$

```

## sigaction

- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`
  - 使用 `man 2 sigaction` 查看帮助
  - 功能: 检查或者改变信号的处理, 即信号捕捉
  - 参数
    - `signum`: 需要捕捉的信号编号或者宏值 (信号的名称)
    - `act`: 捕捉到信号之后的处理动作
    - `oldact`: 上一次对信号捕捉相关的设置, 一般不使用, 设置为 NULL
  - 返回值: 成功返回 0, 失败返回 -1
- `struct sigaction`

```

struct sigaction {
 // 函数指针，指向的函数就是信号捕捉到之后的处理函数
 void (*sa_handler)(int);
 // 不常用
 void (*sa_sigaction)(int, siginfo_t *, void *);
 // 临时阻塞信号集，在信号捕捉函数执行过程中，临时阻塞某些信号。
 sigset_t sa_mask;
 // 使用哪一个信号处理对捕捉到的信号进行处理
 // 这个值可以是0，表示使用sa_handler,也可以是SA_SIGINFO表示使用sa_sigaction
 int sa_flags;
 // 被废弃掉了
 void (*sa_restorer)(void);
};

```

```

#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

void myalarm(int num) {
 printf("捕捉到了信号的编号是: %d\n", num);
 printf("xxxxxxx\n");
}

// 过3秒以后，每隔2秒钟定时一次
int main() {

 struct sigaction act;
 act.sa_flags = 0;
 act.sa_handler = myalarm;
 sigemptyset(&act.sa_mask); // 清空临时阻塞信号集

 // 注册信号捕捉
 sigaction(SIGALRM, &act, NULL);

 struct itimerval new_value;

 // 设置间隔的时间
 new_value.it_interval.tv_sec = 2;
 new_value.it_interval.tv_usec = 0;

 // 设置延迟的时间,3秒之后开始第一次定时
 new_value.it_value.tv_sec = 3;
 new_value.it_value.tv_usec = 0;

 int ret = setitimer(ITIMER_REAL, &new_value, NULL); // 非阻塞的
 printf("定时器开始了...\n");

 if(ret == -1) {
 perror("setitimer");
 exit(0);
 }

 // getchar();
}

```

```
while(1);

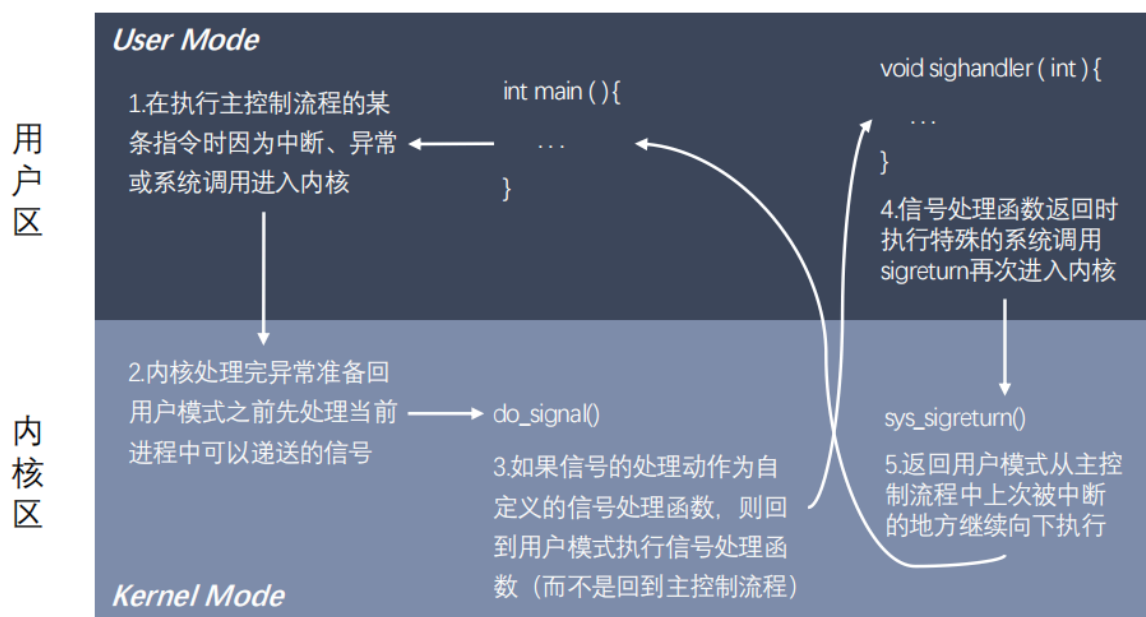
return 0;
}
```

```
u@ubuntu:~/Desktop/Linux$ gcc sigaction.c -o sigaction
u@ubuntu:~/Desktop/Linux$./sigaction
定时器开始了...
捕捉到了信号的编号是: 14
xxxxxxx
捕捉到了信号的编号是: 14
xxxxxxx
捕捉到了信号的编号是: 14
xxxxxxx
^C
u@ubuntu:~/Desktop/Linux$
```

## signal和sigaction区别

- 参数区别
- 版本区别，`signal` 在不同版本Linux中，行为不一致，所以推荐使用 `sigaction`（ubutun 下两者一致）

## 内核实现信号捕捉的过程



## 未解决

- `signal` 中可以使用一个 `getchar()` 阻塞信号，而 `sigaction` 中调用几次回调函数，就要使用多少个 `getchar()`



# 信号集

## 基本概念

- 使用 `man 3 sigset` 查看帮助
- 许多信号相关的系统调用都需要能表示一组不同的信号，多个信号可使用一个称之为信号集的数据结构来表示，其系统数据类型为 `sigset_t`
- 在 PCB 中有两个非常重要的信号集。一个称之为 **阻塞信号集**，另一个称之为 **未决信号集**。这两个信号集都是**内核使用位图机制来实现的**。但操作系统不允许我们直接对这两个信号集进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改
- 信号的 **未决** 是一种状态，指的是**从信号的产生到信号被处理前的这一段时间**
- 信号的 **阻塞** 是一个开关动作，指的是**阻止信号被处理，但不是阻止信号产生**。信号的阻塞就是让系统暂时保留信号留待以后发送。由于另外有办法让系统忽略信号，所以一般情况下信号的阻塞只是暂时的，只是为了防止信号打断敏感的操作

## 阻塞信号集与非阻塞信号集说明

1. 用户通过键盘 `Ctrl + C`，产生2号信号 `SIGINT` (信号被创建)
2. 信号产生但是没有被处理（未决）
  - 在内核中将所有的没有被处理的信号存储在一个集合中（未决信号集）
  - `SIGINT` 信号状态被存储在第二个标志位上
    - 这个标志位的值为0，说明信号不是未决状态
    - 这个标志位的值为1，说明信号处于未决状态
3. 这个未决状态的信号，需要被处理，处理之前需要和另一个信号集（阻塞信号集），进行比较
  - 阻塞信号集默认不阻塞任何的信号
  - 如果想要阻塞某些信号需要用户调用系统的API
4. 在处理的时候和阻塞信号集中的标志位进行查询，看是不是对该信号设置阻塞了
  - 如果没有阻塞，这个信号就被处理
  - 如果阻塞了，这个信号就继续处于未决状态，直到阻塞解除，这个信号就被处理

## 操作自定义信号集函数(sigemptyset等)

- 使用 `man 3 sigemptyset` 查看帮助
- `int sigemptyset(sigset_t *set);`
  - 功能：清空信号集中的数据，将信号集中的所有的标志位置为0
  - 参数：`set`，传出参数，需要操作的信号集
  - 返回值：成功返回0，失败返回-1
- `int sigfillset(sigset_t *set);`
  - 功能：将信号集中的所有的标志位置为1
  - 参数：`set`，传出参数，需要操作的信号集
  - 返回值：成功返回0，失败返回-1
- `int sigaddset(sigset_t *set, int signum);`

- 功能：设置信号集中的某一个信号对应的标志位为1，表示阻塞这个信号
- 参数
  - `set`：传出参数，需要操作的信号集
  - `signum`：需要设置阻塞的那个信号
- 返回值：成功返回0，失败返回-1
- `int sigdelset(sigset_t *set, int signum);`
  - 功能：设置信号集中的某一个信号对应的标志位为0，表示不阻塞这个信号
  - 参数
    - `set`：传出参数，需要操作的信号集
    - `signum`：需要设置不阻塞的那个信号
  - 返回值：成功返回0，失败返回-1
- `int sigismember(const sigset_t *set, int signum);`
  - 功能：判断某个信号是否阻塞
  - 参数
    - `set`：传入参数，需要操作的信号集
    - `signum`：需要判断的那个信号
  - 返回值
    - 1： `signum` 被阻塞
    - 0： `signum` 不阻塞
    - -1： 失败

```
#include <signal.h>
#include <stdio.h>

int main()
{
 // 创建一个信号集
 sigset_t set;

 // 清空信号集的内容
 sigemptyset(&set);

 // 判断 SIGINT 是否在信号集 set 里
 int ret = sigismember(&set, SIGINT);
 if(ret == 0) {
 printf("SIGINT 不阻塞\n");
 } else if(ret == 1) {
 printf("SIGINT 阻塞\n");
 }

 // 添加几个信号到信号集中
 sigaddset(&set, SIGINT);
 sigaddset(&set, SIGQUIT);

 // 判断SIGINT是否在信号集中
 ret = sigismember(&set, SIGINT);
```

```

if(ret == 0) {
 printf("SIGINT 不阻塞\n");
} else if(ret == 1) {
 printf("SIGINT 阻塞\n");
}

// 判断SIGQUIT是否在信号集中
ret = sigismember(&set, SIGQUIT);
if(ret == 0) {
 printf("SIGQUIT 不阻塞\n");
} else if(ret == 1) {
 printf("SIGQUIT 阻塞\n");
}

// 从信号集中删除一个信号
sigdelset(&set, SIGQUIT);

// 判断SIGQUIT是否在信号集中
ret = sigismember(&set, SIGQUIT);
if(ret == 0) {
 printf("SIGQUIT 不阻塞\n");
} else if(ret == 1) {
 printf("SIGQUIT 阻塞\n");
}

return 0;
}

```

```

u@ubuntu:~/Desktop/Linux$ ls
delete.sh sigset.c
u@ubuntu:~/Desktop/Linux$ gcc sigset.c -o sigset
u@ubuntu:~/Desktop/Linux$./sigset
SIGINT 不阻塞
SIGINT 阻塞
SIGQUIT 阻塞
SIGQUIT 不阻塞
u@ubuntu:~/Desktop/Linux$

```

## 操作内核信号集函数(sigprocmask & sigpending)

- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
  - 使用 `man 2 sigprocmask` 查看帮助
  - 功能: 将自定义信号集中的数据设置到内核中 (设置阻塞, 解除阻塞, 替换)
  - 参数
    - `how`: 如何对内核阻塞信号集进行处理
      - `SIG_BLOCK`: 将用户设置的阻塞信号集添加到内核中, 内核中原来的数据不变。假设内核中默认的阻塞信号集是 `mask`, 相当于 `mask | set`
      - `SIG_UNBLOCK`: 根据用户设置的数据, 对内核中的数据进行解除阻塞。相当于 `mask &= ~set`
      - `SIG_SETMASK`: 覆盖内核中原来的值
    - `set`: 已经初始化好的用户自定义的信号集

- `oldset` : 保存设置之前的内核中的阻塞信号集的状态，一般不使用，设置为 `NULL` 即可
  - 返回值：成功返回0，失败返回-1
- `int sigpending(sigset_t *set);`
  - 使用 `man 2 sigpending` 查看帮助
  - 功能：获取内核中的未决信号集
  - 参数：set，传出参数，保存的是内核中的未决信号集中的信息
  - 返回值：成功返回0，失败返回-1

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
 // 设置自定义信号集
 sigset_t set;
 // 清空信号集
 sigemptyset(&set);
 // 设置2 3号信号阻塞
 sigaddset(&set, SIGINT);
 sigaddset(&set, SIGQUIT);
 // 修改内核中的阻塞信号集
 sigprocmask(SIG_BLOCK, &set, NULL);
 int num = 0;
 // 循环获取当前的未决信号集的数据
 while (1) {
 // 计数，用以退出循环
 num++;
 sigset_t pendingset;
 // 清空
 sigemptyset(&pendingset);
 // 获取当前的未决信号集的数据
 sigpending(&pendingset);
 // 遍历前32位
 for(int i = 1; i <= 31; i++) {
 if(sigismember(&pendingset, i) == 1) {
 printf("1");
 }else if(sigismember(&pendingset, i) == 0) {
 printf("0");
 }else {
 perror("sigismember");
 exit(0);
 }
 }
 printf("\n");
 sleep(1);
 if(num == 10) {
 // 解除阻塞
 sigprocmask(SIG_UNBLOCK, &set, NULL);
 }
 }
}
```

```
 return 0;
}
```

```
u@ubuntu:~/Desktop/Linux$ gcc sigprocmask.c -o sigprocmask
u@ubuntu:~/Desktop/Linux$./sigprocmask
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
00000000000000000000000000000000
^C01000000000000000000000000000000
01000000000000000000000000000000
01000000000000000000000000000000
^\\01100000000000000000000000000000
01100000000000000000000000000000
01100000000000000000000000000000
01100000000000000000000000000000
u@ubuntu:~/Desktop/Linux$
```

## SIGCHLD信号

### 基本介绍

- 作用：解决僵尸进程问题，能够在不阻塞父进程的情况下，回收子进程的资源

### 实例：僵尸问题解决

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void myalarm(int num) {
 printf("捕捉到了信号的编号是: %d\n", num);
 // 回收子进程PCB的资源
 // 因为可能多个子进程同时死了，所以使用while循环
 // 不使用wait是因为会造成阻塞，父进程不能继续
 // 使用waitpid可以设置非阻塞
 while (1) {
 int ret = waitpid(-1, NULL, WNOHANG);
 if(ret > 0) {
 // 回收一个子进程
 printf("child die , pid = %d\n", ret);
 } else if(ret == 0) {
 // 说明还有子进程活着
 break;
 } else if(ret == -1) {
 // 没有子进程
 break;
 }
 }
}

int main()
{
```

```

// 提前设置好阻塞信号集，阻塞SIGCHLD，因为有可能子进程很快结束，父进程还没有注册完信号捕捉
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGCHLD);
sigprocmask(SIG_BLOCK, &set, NULL);

pid_t pid;
// 创建一些子进程
for (int i = 0; i < 20; i++) {
 pid = fork();
 // 如果是子进程，不在作为父进程继续创建子进程
 if (pid == 0) {
 break;
 }
}
// 子进程先结束，父进程循环=>产生僵尸进程
if (pid > 0) {
 // 父进程
 // 使用sigaction捕捉子进程死亡时发送的SIGCHLD信号
 struct sigaction act;
 act.sa_flags = 0;
 act.sa_handler = myalarm;
 sigemptyset(&act.sa_mask);
 sigaction(SIGCHLD, &act, NULL);

 // 注册完信号捕捉以后，解除阻塞
 sigprocmask(SIG_UNBLOCK, &set, NULL);

 while (1) {
 printf("parent process : %d\n", getpid());
 sleep(2);
 }
} else {
 // 子进程
 printf("child process : %d\n", getpid());
}

return 0;
}

```

## 注意

- 可能会出现段错误（不一定能复现）
  - 原因：在捕获信号注册前，子进程已经执行完

如果从开始注册信号到注册成功这段时间里，有n个SIGCHLD信号产生的话，那么第一个产生的SIGCHLD会抢先将未决位置为1，余下的n-1个SIGCHLD被丢弃，然后当阻塞解除之后，信号处理函数发现这时候对应信号的未决位为1，继而执行函数处理该信号，处理函数中的while循环顺带将其他n-1子进程也一网打尽了，在这期间未决位的状态只经历了两次变化，即0->1->0

```
u@ubuntu:~/Desktop/Linux$./sigchld
child process : 51398
child process : 51399
child process : 51400
child process : 51401
child process : 51409
child process : 51403
child process : 51404
child process : 51413
child process : 51407
child process : 51408
child process : 51410
child process : 51405
child process : 51402
child process : 51406
child process : 51412
child process : 51414
child process : 51411
child process : 51415
child process : 51416
child process : 51417
Segmentation fault (core dumped)
u@ubuntu:~/Desktop/Linux$
```

- 捕捉一次可能会回收多个子进程

```
u@ubuntu:~/Desktop/Linux$./sigchld
child process : 51543
child process : 51544
child process : 51545
child process : 51546
捕捉到了信号的编号是：17
child die , pid = 51543
child die , pid = 51544
child die , pid = 51545
child process : 51553
child die , pid = 51546
捕捉到了信号的编号是：17
parent process : 51542
child process : 51552
child process : 51555
```

## 进程间通信之共享内存

### 说明

本部分笔记及源码出自 [slide/02Linux多进程开发/08 进程间通信之共享内存](#)

### 基本概念

- **共享内存允许两个或者多个进程共享物理内存的同一块区域（通常被称为段）。**由于一个共享内存段会称为一个进程用户空间的一部分，因此这种 **IPC** 机制无需内核介入。所有需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用
- 与管道等要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 **IPC** 技术的速度更快

### 共享内存使用步骤

1. 调用 `shmget()` 创建一个新共享内存段或取得一个既有共享内存段的标识符（即由其他进程创建的共享内存段）。这个调用将返回后续调用中需要用到的共享内存标识符

2. 使用 `shmat()` 来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分
3. 此刻在程序中可以像对待其他可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 `shmat()` 调用返回的 `addr` 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针
4. 调用 `shmdt()` 来分离共享内存段。在这个调用之后，进程就无法再引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步
5. 调用 `shmctl()` 来删除共享内存段。只有当当前所有附加内存段的进程都与之分离之后内存段才会销毁。只有一个进程需要执行这一步

## 共享内存操作函数

- `int shmget(key_t key, size_t size, int shmflg);`
  - 使用 `man 2 shmget` 查看帮助
  - 功能：创建一个新的共享内存段（新创建的内存段中的数据都会被初始化为0），或者获取一个既有的共享内存段的标识
  - 参数
    - `key`： `key_t` 类型是一个整形，通过这个找到或者创建一个共享内存。一般使用**16进制**表示，非0值
    - `size`： 共享内存的大小
    - `shmflg`： 属性
      - 访问权限
      - 附加属性：创建/判断共享内存是不是存在
        - 创建： `IPC_CREAT`
        - 判断共享内存是否存在： `IPC_EXCL`，需要和 `IPC_CREAT` 一起使用，即 `IPC_CREAT | IPC_EXCL | 0664`
  - 返回值
    - 失败： -1 并设置错误号
    - 成功： >0 返回共享内存的引用的ID，后面操作共享内存都是通过这个值
- `void *shmat(int shmid, const void *shmaddr, int shmflg);`
  - 使用 `man 2 shmat` 查看帮助
  - 功能： 和当前的进程进行关联
  - 参数
    - `shmid`： 共享内存的标识（ID），由 `shmget` 返回值获取
    - `shmaddr`： 申请的共享内存的起始地址，设置为NULL，表示由内核指定
    - `shmflg`： 对共享内存的操作
      - 读： `SHM_RDONLY`，必须要有读权限
      - 读写： 指定为0即为有读写权限
  - 返回值： 成功： 返回共享内存的首（起始）地址。 失败（void \*） -1
- `int shmdt(const void *shmaddr);`
  - 使用 `man 2 shmdt` 查看帮助



- 功能：解除当前进程和共享内存的关联
- 参数：shmaddr：共享内存的首地址
- 返回值：成功 0，失败 -1
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
  - 使用 `man 2 shmctl` 查看帮助
  - 功能：对共享内存进行操作。删除共享内存，共享内存要删除才会消失，创建共享内存的进程被销毁了对共享内存是没有任何影响
  - 参数
    - shmid：共享内存的ID
    - cmd：要做的操作
      - IPC\_STAT：获取共享内存的当前的状态
      - IPC\_SET：设置共享内存的状态
      - IPC\_RMID：标记共享内存被销毁
    - buf：需要设置或者获取的共享内存的属性信息
      - IPC\_STAT：buf 存储数据
      - IPC\_SET：buf 中需要初始化数据，设置到内核中
      - IPC\_RMID：没有用，设置为NULL
- `key_t ftok(const char *pathname, int proj_id);`
  - 使用 `man 3 ftok` 查看帮助
  - 功能：根据指定的路径名，和int值，生成一个共享内存的key
  - 参数
    - pathname：指定一个**存在的路径**
    - proj\_id：int类型的值，但是系统调用只会使用其中的1个字节，范围：0-255 一般指定一个字符 'a'
  - 返回值：shmget 中用到的 key

## 共享内存操作命令

---

### ipcs

- `ipcs -a`：打印当前系统中**所有的**进程间通信方式的信息
- `ipcs -m`：打印出**使用共享内存**进行进程间通信的信息
- `ipcs -q`：打印出**使用消息队列**进行进程间通信的信息
- `ipcs -s`：打印出**使用信号**进行进程间通信的信息

### ipcrm

- `ipcrm -M shmkey`：移除用 shmkey 创建的**共享内存段**
- `ipcrm -m shmid`：移除用 shmid 标识的**共享内存段**
- `ipcrm -Q msgkey`：移除用 msgkey 创建的**消息队列**
- `ipcrm -q msqid`：移除用 msqid 标识的**消息队列**

- `ipcrm -S semkey`: 移除用 `semkey` 创建的信号
- `ipcrm -s semid`: 移除用 `semid` 标识的信号

## 实例：进程间通信（注意）

### 写端

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main()
{
 // 1. 创新新共享内存
 // key不能随意指定，比如用key=100时会产生段错误
 int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
 // 2. 将进程与共享内存关联
 void* ptr = shmat(shmId, NULL, 0);
 // 3. 往共享内存中写数据
 // 操作内存只能使用memcpy，使用strcpy会产生段错误
 // strcpy((char*)addr, "hello, world");
 char* str = "helloworld";
 printf("send : %s\n", str);
 // 包含结束符'\0'
 memcpy(ptr, str, strlen(str) + 1);
 // 为了程序不被直接停掉，如果停掉，那么共享内存不复存在
 printf("按任意键继续\n");
 getchar();
 // 4. 分离内存段
 shmdt(ptr);
 // 5. 删除共享内存段（标记删除）
 shmctl(shmId, IPC_RMID, NULL);
 return 0;
}
```

### 读端

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <string.h>

int main()
{
 // 1. 判断并获取共享内存
 // 注意IPC_EXCL只能在创建共享内存时使用
 int shmId = shmget(100, 1024, IPC_CREAT);
 // int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
 // 2. 将进程与共享内存关联
 void* addr = shmat(shmId, NULL, 0);
 // 3. 从共享内存中读数据
 // 此时字符串内存即为共享内存内容
```

```

printf("recv : %s\n", (char*)addr);
// 为了程序不被直接停掉，如果停掉，那么共享内存不复存在
printf("按任意键继续\n");
getchar();
// 4. 分离内存段
shmdt(addr);
// 5. 删除共享内存段（标记删除）
shmctl(shmId, IPC_RMID, NULL);
return 0;
}

```

## 注意

### 虚拟机和实体机

1. 虚拟机在启动情况下，有默认共享内存，而实体机（服务器）没有

- 虚拟机

```

u@ubuntu:~/Desktop$ ipcs -m

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00000000 8 u 600 524288 2 dest
0x00000000 9 u 600 33554432 2 dest
0x00000000 12 u 600 524288 2 dest
0x00000000 13 u 600 524288 2 dest
0x00000000 16 u 600 524288 2 dest
0x00000000 17 u 600 524288 2 dest
0x00000000 18 u 600 524288 2 dest

```

- 实体机

```

root@ubuntu:~$ ipcs -m

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
root@ubuntu:~$

```

### 执行顺序与代码（注意）

1. 先执行读端，再执行写端，且关键代码如下时，此时读端读到空数据，写端会先输出内容然后产生段错误

```

// write
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
// read
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);

```

```

u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./write
send : helloworld
Segmentation fault (core dumped)
u@ubuntu:~/Desktop$

u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./read
recv :
按任意键继续
u@ubuntu:~/Desktop$

```

2. 先执行写端，再执行读端，且关键代码如下时，此时写端正常写数据，读端会产生段错误

```
// write
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
// read
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./write
send : helloworld
按任意键继续
u@ubuntu:~/Desktop$
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./read
Segmentation fault (core dumped)
u@ubuntu:~/Desktop$
```

3. 先执行读端，再执行写端，且关键代码如下时，此时读端产生段错误，写端会先输出内容然后产生段错误且当前key=100（十六进制为64）被占用，按先写后读顺序时，需要手动回收内存，否则不能继续该块内存，如下图所示

```
// write
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
// read
int shmId = shmget(100, 1024, IPC_CREAT);
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./write
send : helloworld
Segmentation fault (core dumped)
u@ubuntu:~/Desktop$
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./read
Segmentation fault (core dumped)
u@ubuntu:~/Desktop$
```

```
u@ubuntu:~/Desktop$ ipcs -m
```

| ----- Shared Memory Segments ----- |       |       |       |          |        |        |
|------------------------------------|-------|-------|-------|----------|--------|--------|
| key                                | shmId | owner | perms | bytes    | nattch | status |
| 0x00000000                         | 8     | u     | 600   | 524288   | 2      | dest   |
| 0x00000000                         | 9     | u     | 600   | 33554432 | 2      | dest   |
| 0x00000000                         | 12    | u     | 600   | 524288   | 2      | dest   |
| 0x00000000                         | 13    | u     | 600   | 524288   | 2      | dest   |
| 0x00000000                         | 16    | u     | 600   | 524288   | 2      | dest   |
| 0x00000000                         | 17    | u     | 600   | 524288   | 2      | dest   |
| 0x00000000                         | 18    | u     | 600   | 524288   | 2      | dest   |
| 0x00000064                         | 25    | u     | 0     | 1024     | 0      |        |

```
u@ubuntu:~/Desktop$ ipcs -m

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00000000 8 u 600 524288 2 dest
0x00000000 9 u 600 33554432 2 dest
0x00000000 12 u 600 524288 2 dest
0x00000000 13 u 600 524288 2 dest
0x00000000 16 u 600 524288 2 dest
0x00000000 17 u 600 524288 2 dest
0x00000000 18 u 600 524288 2 dest
0x00000064 25 u 0 1024 0 dest

u@ubuntu:~/Desktop$ ipcrm -m 25
u@ubuntu:~/Desktop$ ipcs -m

----- Shared Memory Segments -----
key shmid owner perms bytes nattch status
0x00000000 8 u 600 524288 2 dest
0x00000000 9 u 600 33554432 2 dest
0x00000000 12 u 600 524288 2 dest
0x00000000 13 u 600 524288 2 dest
0x00000000 16 u 600 524288 2 dest
0x00000000 17 u 600 524288 2 dest
0x00000000 18 u 600 524288 2 dest

u@ubuntu:~/Desktop$
```

4. 先执行写端，再执行读端，且关键代码如下时，**正常执行**

```
// write
int shmId = shmget(100, 1024, IPC_CREAT | IPC_EXCL | 0664);
// read
int shmId = shmget(100, 1024, IPC_CREAT);
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./write
send : helloworld
按任意键继续

u@ubuntu:~/Desktop$
```

```
u@ubuntu:~/Desktop$ ls
delete.sh read read.c write write.c
u@ubuntu:~/Desktop$./read
recv : helloworld
按任意键继续

u@ubuntu:~/Desktop$
```

5. 出现的原因

- 当先执行读端时，此时共享内存中没有内容或者没有创建

## 总结

### 常见问题

- 操作系统如何知道一块共享内存被多少个进程关联？
  - 共享内存维护了一个结构体 `struct shmid_ds`，这个结构体中有一个成员 `shm_nattch`
  - `shm_nattch` 记录了关联的进程个数
- 可不可以对共享内存进行多次删除 `shmctl`
  - 可以，因为执行 `shmctl` 表示**标记删除共享内存（key变为0）**，**不是直接删除**。当和共享内存关联的进程数为0的时候，就真正被删除
  - 如果一个进程和共享内存取消关联，那么这个进程就不能继续操作这个共享内存

# 共享内存与内存映射区别

- **共享内存**可以直接创建，**内存映射**需要磁盘文件（匿名映射除外）
- 共享内存效率更高
- **共享内存**所有的进程操作的是同一块共享内存，**内存映射**，每个进程在自己的虚拟地址空间中有一个独立的内存
- 数据安全
  - 进程突然退出：**共享内存**还存在，**内存映射区**消失
  - 运行进程的电脑死机(宕机)：**共享内存**中的数据消失，内存映射区的数据也消失，但由于磁盘文件中的数据还在，所以可以说**内存映射区的数据还存在**
- 生命周期
  - 共享内存
    - 进程退出时共享内存还在，只会标记删除
    - 只有当所有的关联的进程数为0或者关机时，才会真正删除
    - 如果一个进程退出，会自动和共享内存进行取消关联
  - 内存映射区：进程退出，内存映射区销毁

## 守护进程

### 说明

本部分笔记及源码出自 `slide/02Linux多进程开发/09 守护进程`

### 前置知识

#### 终端

- 在 `UNIX` 系统中，用户通过终端登录系统后得到一个 `shell` 进程，这个终端成为 `shell` 进程的 **控制终端**（`Controlling Terminal`），进程中，控制终端是保存在 `PCB` 中的信息，而 `fork()` 会复制 `PCB` 中的信息，因此由 `shell` 进程启动的其它进程的控制终端也是这个终端
- 默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端
  - 进程从标准输入读也就是读用户的键盘输入
  - 进程往标准输出或标准错误输出写也就是输出到显示器上
- 在控制终端输入一些特殊的控制键可以给前台进程发信号，例如 `Ctrl + C` 会产生 `SIGINT` 信号，`Ctrl + \` 会产生 `SIGQUIT` 信号

#### 进程组

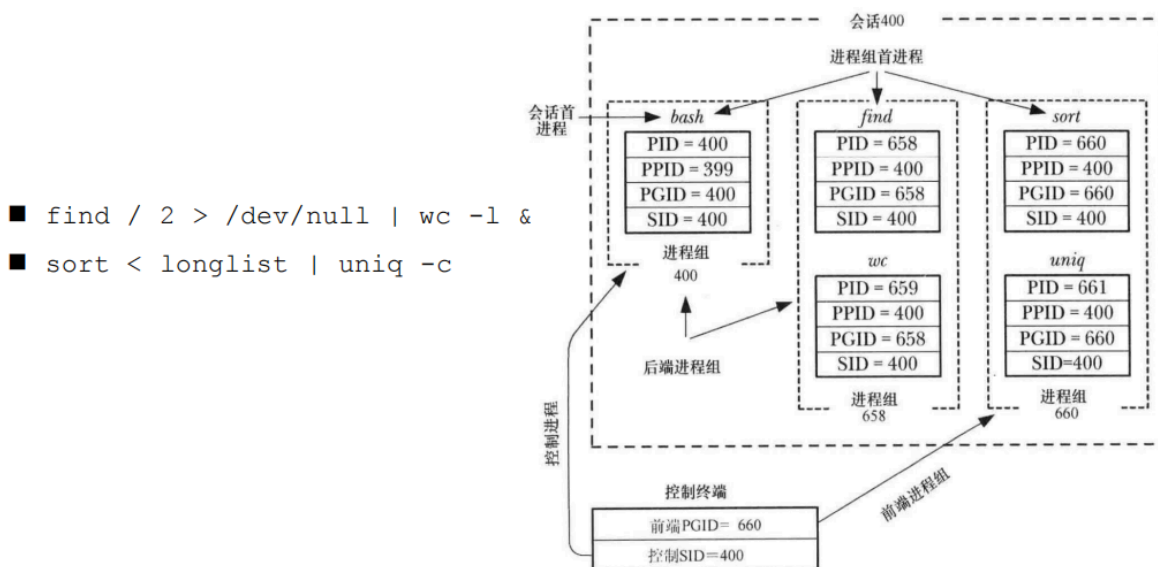
- **进程组**和**会话**在进程之间形成了一种两级层次关系
  - 进程组是一组相关进程的集合，会话是一组相关进程组的集合
  - 进程组和会话是为支持 `shell` 作业控制而定义的抽象概念，用户通过 `shell` 能够交互式地在前台或后台运行命令
- 进程组由一个或多个共享同一进程组标识符（`PGID`）的进程组成
- 一个进程组拥有一个**进程组首进程**，该进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID

- 进程组拥有一个生命周期，其**开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻**
- 一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组
- 进程组首进程无需是最后一个离开进程组的成员

## 会话

- **会话**是一组进程组的集合
- **会话首进程**是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID
- 一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立
- 一个终端最多可能会成为一个会话的控制终端
- 在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入终端字符生成信号后，该信号会被发送到前台进程组中的所有成员
- 当控制终端的连接建立起来之后，会话首进程会成为该终端的控制进程

## 进程组、会话、控制终端之间的关系



## 进程组、会话操作函数

- `pid_t getpgrp(void);`
- `pid_t getpgid(pid_t pid);`
- `int setpgid(pid_t pid, pid_t pgid);`
- `pid_t getsid(pid_t pid);`
- `pid_t setsid(void);`

## 守护进程概念

- **守护进程 (Daemon Process)**，也就是通常说的 Daemon 进程（精灵进程），是Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字

- 守护进程特征
  - 生命周期很长，守护进程会在系统启动的时候被创建并一直运行直至系统被关闭
  - 它在后台运行并且不拥有控制终端。没有控制终端确保了内核永远不会为守护进程自动生成任何控制信号以及终端相关的信号（如 `SIGINT`、`SIGQUIT`）
- Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 `inetd`，Web 服务器 `httpd` 等

## 守护进程的创建步骤

1. 执行一个 `fork()`，之后父进程退出，子进程继续执行
2. 子进程调用 `setsid()` 开启一个新会话
3. 清除进程的 `umask` 以确保当守护进程创建文件和目录时拥有所需的权限
4. 修改进程的当前工作目录，通常会改为根目录（`/`）
5. 关闭守护进程从其父进程继承而来的所有打开着的文件描述符
6. 在关闭了文件描述符0、1、2之后，守护进程通常会打开 `/dev/null` 并使用 `dup2()` 使所有这些描述符指向这个设备
7. 核心业务逻辑

## 实例：守护进程实现每隔两秒获取时间并写入磁盘

```
/*
 写一个守护进程，每隔2s获取一下系统时间，将这个时间写入到磁盘文件中。
*/

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/time.h>
#include <signal.h>
#include <time.h>
#include <string.h>

void mywork(int num) {
 // 捕捉到信号之后，获取系统时间，写入磁盘文件
 time_t tm = time(NULL);
 struct tm * loc = localtime(&tm);

 char* str = asctime(loc);
 int fd = open("time.txt", O_RDWR | O_CREAT | O_APPEND, 0664);
 write(fd, str, strlen(str));
 close(fd);
}

int main()
{
 // 1. fork产生子进程，并退出父进程
 pid_t pid = fork();
```



```

if (pid > 0) {
 exit(0);
}
// 2. 子进程调用 setsid() 开启一个新会话
setsid();
// 3. 设置掩码
umask(022);
// 4. 修改进程的当前工作目录，通常设为/，这里应该是权限不够，所以改为当前目录
chdir("/home/u/Desktop");
// 5. 关闭、重定向文件描述符
int fd = open("/dev/null", O_RDWR);
dup2(fd, STDIN_FILENO);
dup2(fd, STDOUT_FILENO);
dup2(fd, STDERR_FILENO);
// 6. 业务逻辑
// 捕捉定时信号
struct sigaction act;
act.sa_flags = 0;
act.sa_handler = myWork;
sigemptyset(&act.sa_mask);
sigaction(SIGALRM, &act, NULL);

// 设置定时器
struct itimerval val;
val.it_interval.tv_sec = 2;
val.it_interval.tv_usec = 0;
val.it_value.tv_sec = 2;
val.it_value.tv_usec = 0;
setitimer(ITIMER_REAL, &val, NULL);

// 不让进程结束
while(1) {
 sleep(10);
}

return 0;
}

```

```

u@ubuntu:~/Desktop$ gcc daemon.c -o daemon
u@ubuntu:~/Desktop$./daemon
u@ubuntu:~/Desktop$ ls
daemon daemon.c delete.sh
u@ubuntu:~/Desktop$ ls
daemon daemon.c delete.sh time.txt
u@ubuntu:~/Desktop$ cat time.txt
Sat Oct 23 08:12:43 2021
Sat Oct 23 08:12:45 2021
Sat Oct 23 08:12:47 2021
Sat Oct 23 08:12:49 2021
Sat Oct 23 08:12:51 2021
u@ubuntu:~/Desktop$

```

# 实用技巧

## 后台运行进程

- code

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

int main()
{
 while (1) {
 printf("this is a test...\n");
 sleep(1);
 }
 return 0;
}
```

- 进程切换到后台运行: `./test &`, 切换到后台后, 当前终端可以使用其他命令, 此时无法通过 `CTRL C` 终止

```
u@ubuntu:~/Desktop/Linux$./test &
[1] 16364
u@ubuntu:~/Desktop/Linux$ this is a test...
this is a test...
this is a test...
this is a test...
this is a test...
this is a test...
lsthis is a test...
delete.sh sigprocmask.c test test.c
u@ubuntu:~/Desktop/Linux$ this is a test...
this is a test...
```

- 后台进程切换到前台: `fg`, 切换后, 可以通过 `CTRL C` 终止

```
u@ubuntu:~/Desktop/Linux$ this is a test...
this is a test...
this is a test...
fgthis is a test...

./test
this is a test...
this is a test...
^C
u@ubuntu:~/Desktop/Linux$
```