1.进程相关概念

1.1进程的概念

进程：进程是程序的一次执行过程，进程是一个正在执行的任务，

进程是分配资源的最小单位。每一个进程都会分配自己的0-3G的

内存空间，0-3G的内存空间有多份，而3-4G内核空间只有一份。

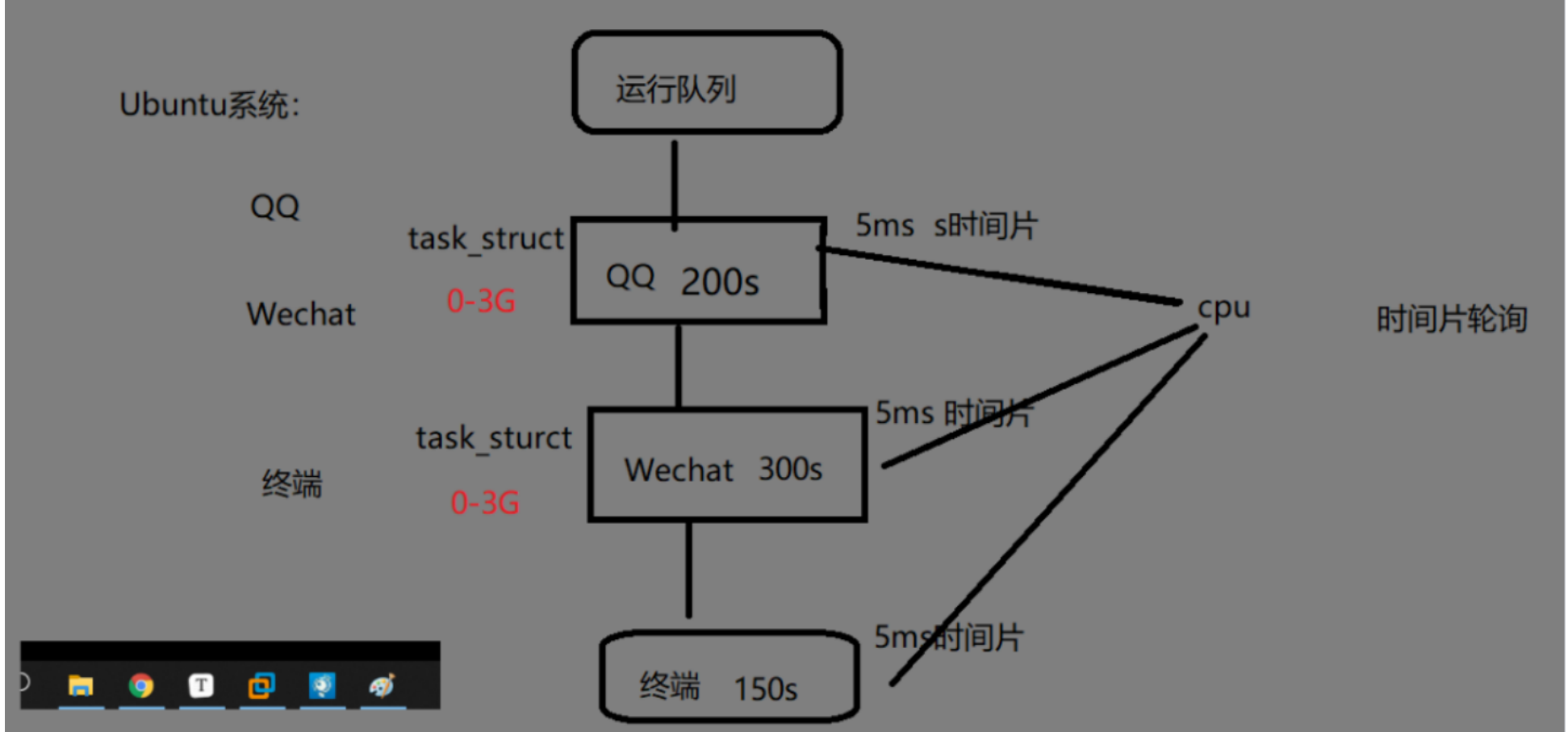
在这0-3G内存空间中堆区、栈区、静态区（缓冲区，文件描述符）。

进程其实是内核创建的，每个进程在内核空间都对应的是一个

task_struct(PCB)的结构体。正在运行的进程会被放到一个运行队

列中，随着时间片轮询依次来执行进程。一个进程的崩溃不会影

响另外一个进程的执行，**进程的安全性高。**



1.2进程和程序的区别

程序：程序是静态的，没有生命周期的概念，它是有序的

指令的集合，在硬盘上存储着

进程：进程是程序的一次执行过程，它是由生命周期的，

随着程序的执行而运行，随着程序的终止而结束，

在内存上存储着。可以分配自己的0-3G的内存空间

1.3进程的组成

进程的组成：进程控制块（PCB task_struct）,文本段，数据段。

进程控制块PCB:进程的标识符PID，进程运行的状态，进程所属的用户uid，内存空间...

数据段：存放程序运行时候产生的数据，比如int a；a就在数据段存放

文本段：存放可执行程序本身

1.4进程的种类

进程的种类：交互进程，批处理进程，守护进程

交互进程 ： 交互进程是由shell维护的，通过shell和用户进行交互，例如文本编辑器就是一个交互进程。

批处理进程：批处理进程的优先级比较低，通常情况下批处理进程都会被放到队列中执行，例如gcc编译程序的过程就是批处理进程。

守护进程 ： 守护进程是一个后台运行的进程，随着系统的启动而启动，随着系统的终止而终止，它会脱离终端执行，例如windows上的各种服务。

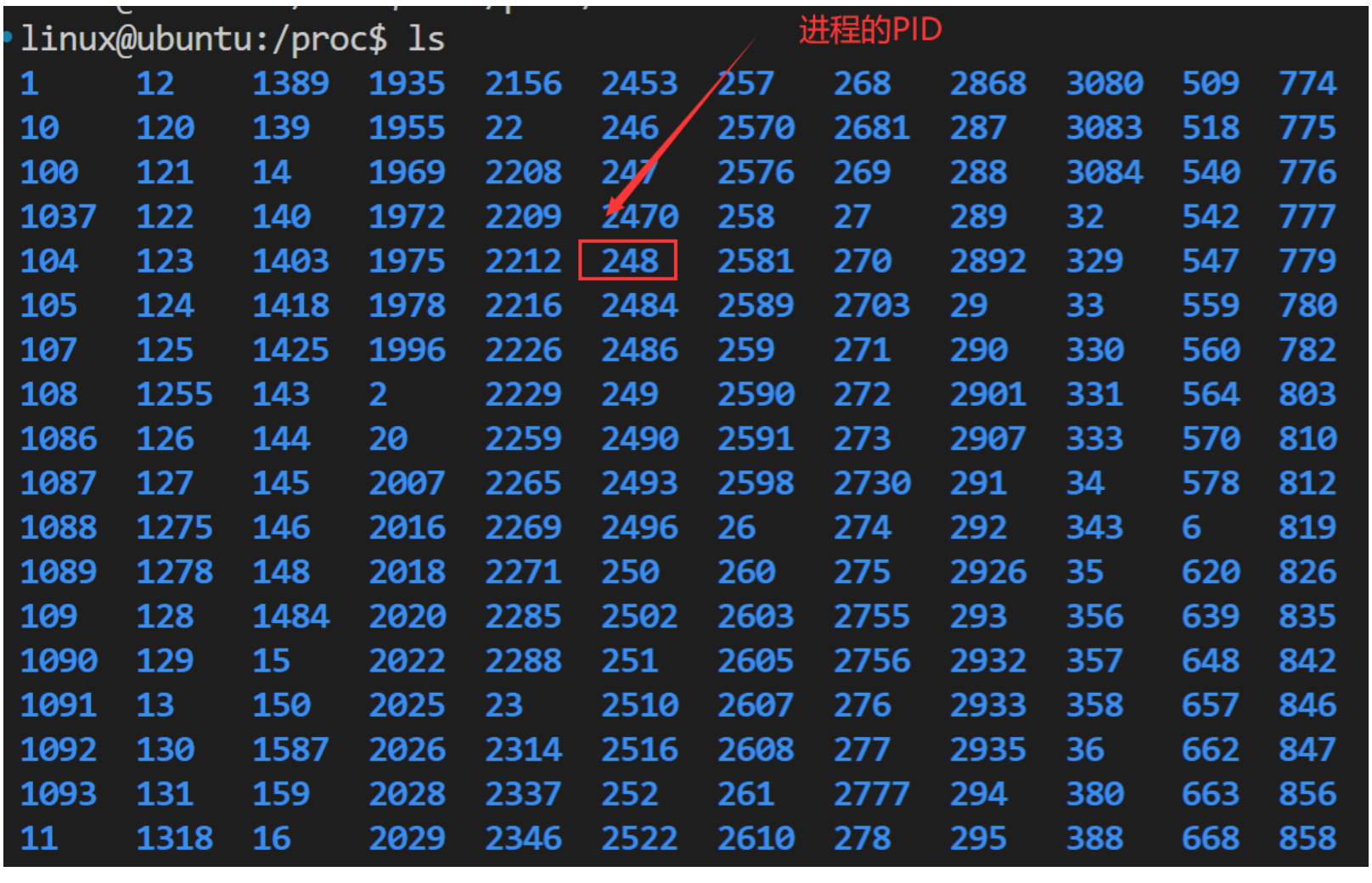
1.5什么是PID

PID(process id):进程号，在linux系统上进程都会被分配一个ID，这个ID就是进程的标号。

在linux系统上所有的进程都可以在/proc目录下查看。PPID是进程的父进程号。在一个系统上

可以通过如下命令查看能创建的最大进程的个数：

cat /proc/sys/kernel/pid_max 131072



1.6特殊PID的进程

0号进程（idle）:在linux系统启动的时候最先运行的进程就是0号进程，0号进程又叫空闲进程。

如果系统上没有其他进程执行那么0号进程就执行。0号进程是1号进程和2号进程的父进程

1号进程（init）:init进程是由0号进程创建得到的，它的主要工作是系统的初始化。当初始化工作执行完之后，它主要负责回收孤儿进程的资源。

2号进程（kthreadd）:kthreadd是有0号进程创建出来的，它主要负责调度工作（调度器进程）

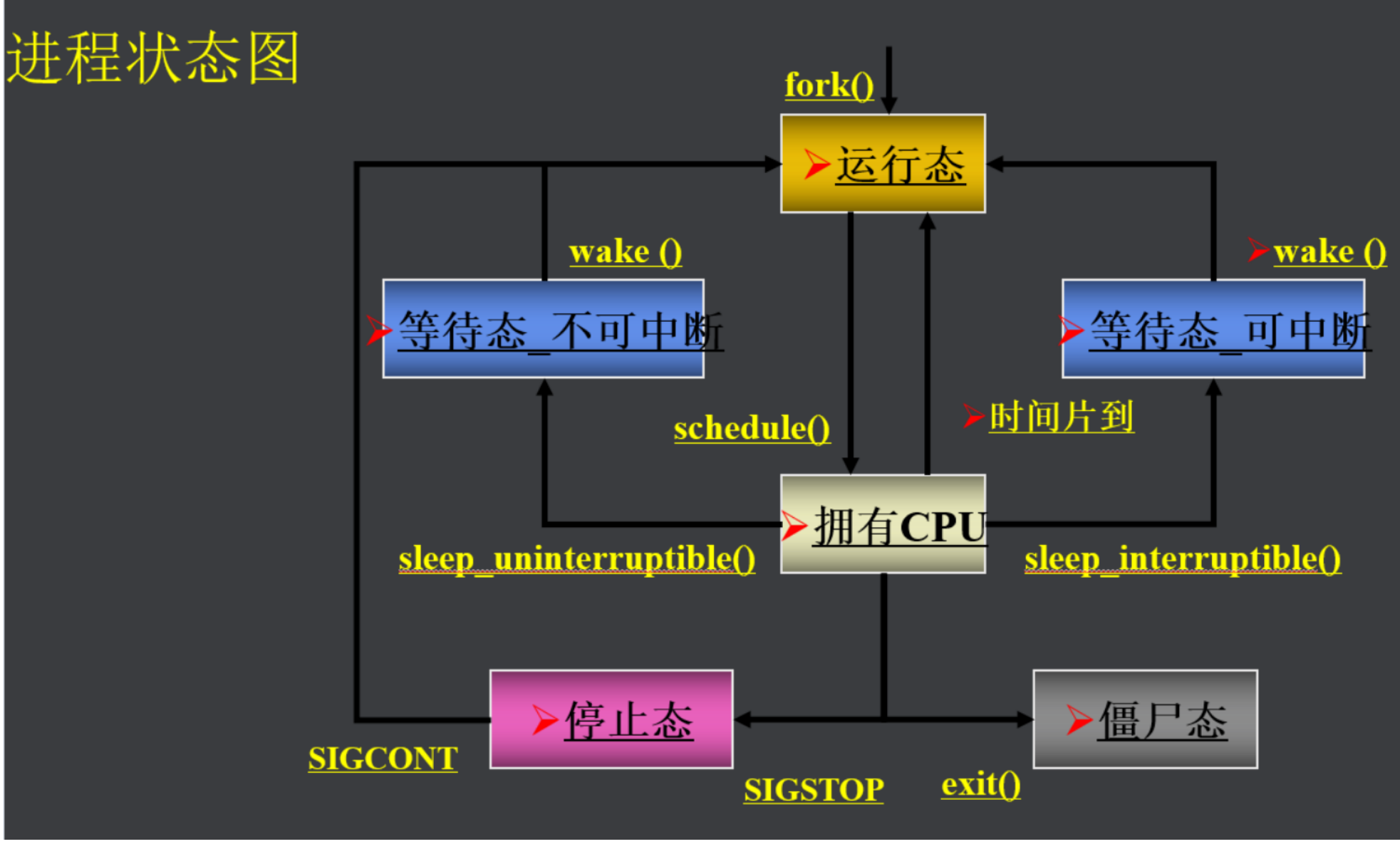
1.7进程相关命令

```
1.ps命令
    linux@ubuntu:~$ ps -ef  //查看进程的父子关系
    UID          PID    PPID  C STIME TTY          TIME CMD
    root           1       0  0 13:29 ?        00:00:03 /sbin/init auto noprompt
    root           2       0  0 13:29 ?        00:00:00 [kthreadd]
    root           3       2  0 13:29 ?        00:00:00 [rcu_gp]
    root           4       2  0 13:29 ?        00:00:00 [rcu_par_gp]
    root           6       2  0 13:29 ?        00:00:00 [kworker/0:0H-kb]
    root           9       2  0 13:29 ?        00:00:00 [mm_percpu_wq]
    root          10       2  0 13:29 ?        00:00:00 [ksoftirqd/0]
    //PID:进程号
    //PPID:父进程号
    //TTY:如果是问号，说明没有终端与之对应
    //CMD:进程名
    linux@ubuntu:~$ ps -ajx   //一般使用这条命令查看，查看到的进程的信息更完全
      PPID    PID   PGID    SID TTY       TPGID STAT   UID   TIME COMMAND
         0      1      1      1 ?            -1 Ss       0   0:03 /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   0:06 [rcu_sched]
          2     12      0      0 ?           -1 S        0   0:00 [migration/0]
    //PID:进程号
    //PPID:父进程号
    //PGID:进程组ID
    //SID:会话ID
    //在linux系统上新开一个终端就会默认创建一个会话，一个会话包含多个进程组
    //其中进程组有分为前台进程组和后台进程组，前台进程组只有一个，后台进程组有多个。
    //一个进程组内包含很多个进程，进程具备父子关系。
    //TTY:如果是问号，说明没有终端与之对应
    //TPGID:如果是-1就是守护进程
2.top/htop命令
htop动态查看进程信息比top查看的更可视化一些（sudo apt-get install htop）
   PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
     1 root       20   0  220M  9008  6612 S  0.0  0.2  0:03.11 /sbin/init auto noprompt
   485 root       19  -1 95088 16152 15108 S  0.0  0.4  0:00.65 /lib/systemd/systemd-journald
   502 root       20   0 23924   180     4 S  0.0  0.0  0:00.00 /usr/sbin/blkmapd
   540 root       20   0 46716  4836  3124 S  0.0  0.1  0:00.37 /lib/systemd/systemd-udevd
   564 root       20   0 23752   200     0 S  0.0  0.0  0:00.00 /usr/sbin/rpc.idmapd
   657 root       20   0 47604  3560  3164 S  0.0  0.1  0:00.05 /sbin/rpcbind -f -w
   662 systemd-t  20   0  142M  3188  2720 S  0.0  0.1  0:00.13 /lib/systemd/systemd-timesyncd
   663 systemd-r  20   0 70620  4688  4224 S  0.0  0.1  0:00.14 /lib/systemd/systemd-resolved

3.给进程发信号的命令
    linux@ubuntu:~$ 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

    kill  -信号号  PID   //给PID进程发送信号

    SIGINT:打断正在执行的程序（ctrl+c）
    SIGKILL:杀死进程
    SIGSTOP:停止
    SIGCONT:继续

    pidof 可执行程序的名字：查看进程号          //pidof a.out
    killall 可执行程序的名字：杀死有同名的进程  //killall a.out

4.查看最大进程号的命令
    cat /proc/sys/kernel/pid_max
```

1.8进程的状态

```
1.进程的状态
   D    //不可中断的等待态（sleep，不可被信号打断）
   R    //运行状态
   S    //可中断的等待态（sleep,可被信号打断）
   T    //停止状态
   X    //死亡状态
   Z    //僵尸态
   I    //空闲态

2.进程的附加态
   <    //高优先级进程
   N    //低优先级进程
   L    //锁在内存上
   s    //会话组组长
   l    //包含多线程
   +    //前台进程
```



> 孤儿进程：一个进程的父进程死亡，此时当前的进程就是孤儿进程，孤儿进程被init收养
>
> 僵尸态进程：如果一个进程结束，父进程没有为它收尸，此时当前的进程就是僵尸进程。

1.9进程状态切换实例

```c
#include <head.h>

int main(int argc,const char * argv[])
{
    while(1){
        sleep(1);
    }
    return 0;
}
```

1. 运行上述程序程序状态（前台休眠）

```
linux@ubuntu:~/work$ ps -ajx | grep a.out
 16021  16151  16151  16021 pts/1     16151 S+    1000   0:00 ./a.out
```

2. 将上述的进程变为停止态

`ctrl + Z`

```
linux@ubuntu:~/work$ ps -ajx | grep a.out
 16021  16151  16151  16021 pts/1     16021 T    1000   0:00 ./a.out
```

3. 将停止状态的进程变为后台休眠

```
linux@ubuntu:~/work/day5$ jobs -l
[1]- 16151 停止                 ./a.out
[2]+ 16904 停止                 ./a.out
```


`bg 1`

```
linux@ubuntu:~/work$ ps -ajx | grep a.out
 16021  16151  16151  16021 pts/1     16021 S    1000   0:00 ./a.out
```

4. 让后台休眠进程变为前台休眠

`fg 1`

```
linux@ubuntu:~/work$ ps -ajx | grep a.out
 16021  16151  16151  16021 pts/1     16151 S+    1000   0:00 ./a.out
```

2.进程的创建及特点

2.1如何创建进行

进程的创建是拷贝父进程得到的，通过拷贝过程更容易得到子进程，并且能够标识

进程的父子状态。

2.2创建进程的API

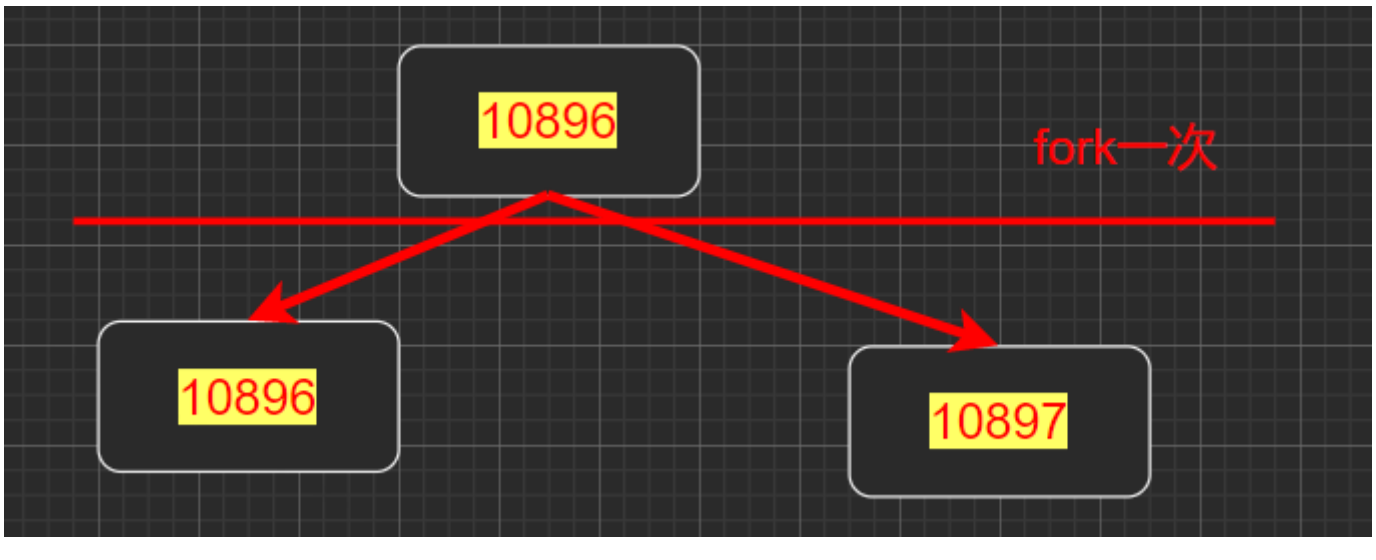
```
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void);
功能：创建一个子进程
参数：
    @无
返回值：成功父进程收到子进程的PID，子进程收到0
        失败父进程收到-1，并置位错误码
```

2.3创建进程的实例

创建一个子进程，不关注返回值

```
#include <head.h>

int main(int argc, const char* argv[])
{
    fork();    // 创建一个子进程

    while (1); // 父子进程都在执行while(1);

    return 0;
}
```

上述程序执行后进程的关系图：

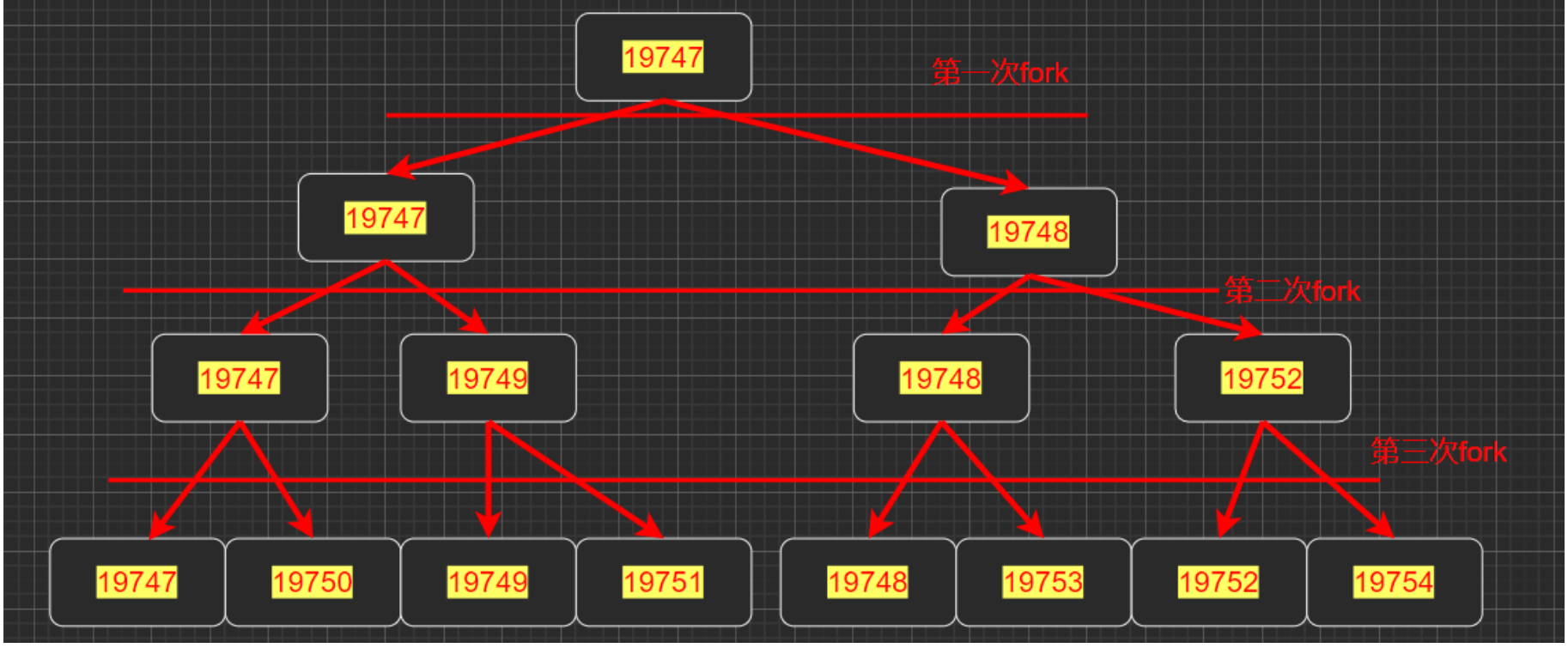


2.4创建进程的实例

创建多个子进程，不关注返回值

```
#include <head.h>

int main(int argc, const char* argv[])
{
    for (int i = 0; i < 3; i++) {
        fork(); // 创建一个子进程
    }
    while (1);

    return 0;
}
```

```
linux@ubuntu:~/work$ ps -ajx| grep a.out
 19659  19747  19747  19659 pts/0     19747 R+    1000   0:05 ./a.out
 19747  19748  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
 19747  19749  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
 19747  19750  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
 19749  19751  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
 19748  19752  19747  19659 pts/0     19747 R+    1000   0:05 ./a.out
 19748  19753  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
 19752  19754  19747  19659 pts/0     19747 R+    1000   0:04 ./a.out
```



> 按照上述的写法：fork n次就产生了2^n个进程

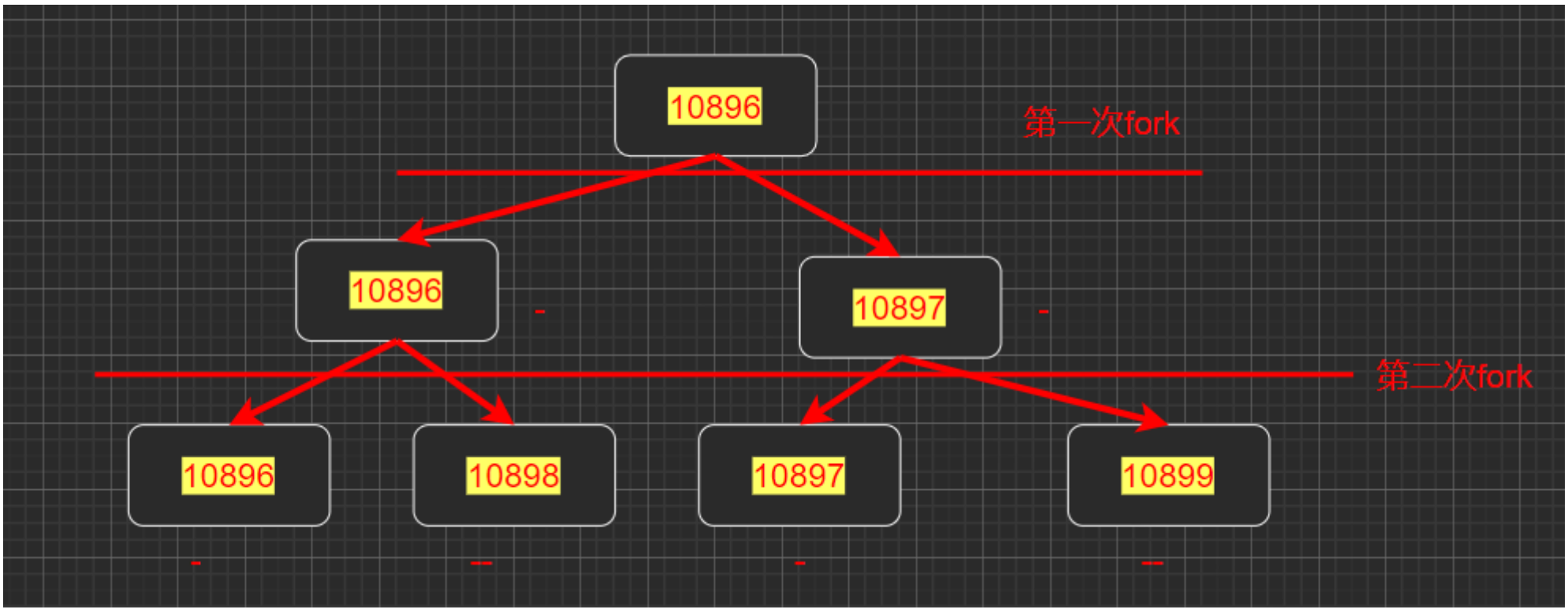
2.5fork和缓冲区结合问题

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    for (int i = 0; i < 2; i++) {
        fork();
        printf("-");
    }

    return 0;
}
```

问：上述程序执行结束，会打印多少个'-'，为什么?

答：上述程序会打印8个'-'。原因是上述程序printf没有刷新缓冲区，所以

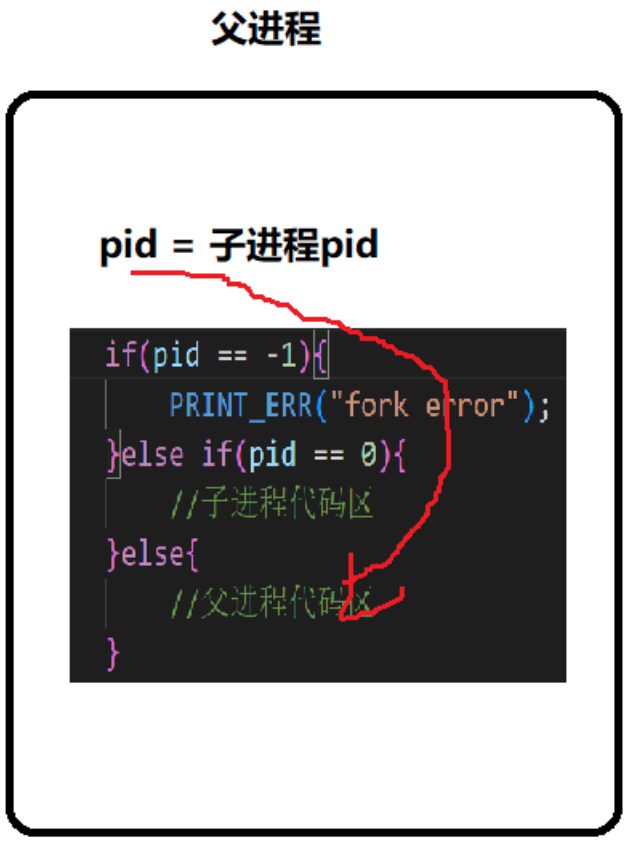
在fork进程的时候，会将父进程缓冲区的内容也fork过来，所以最终打印了8个'-'

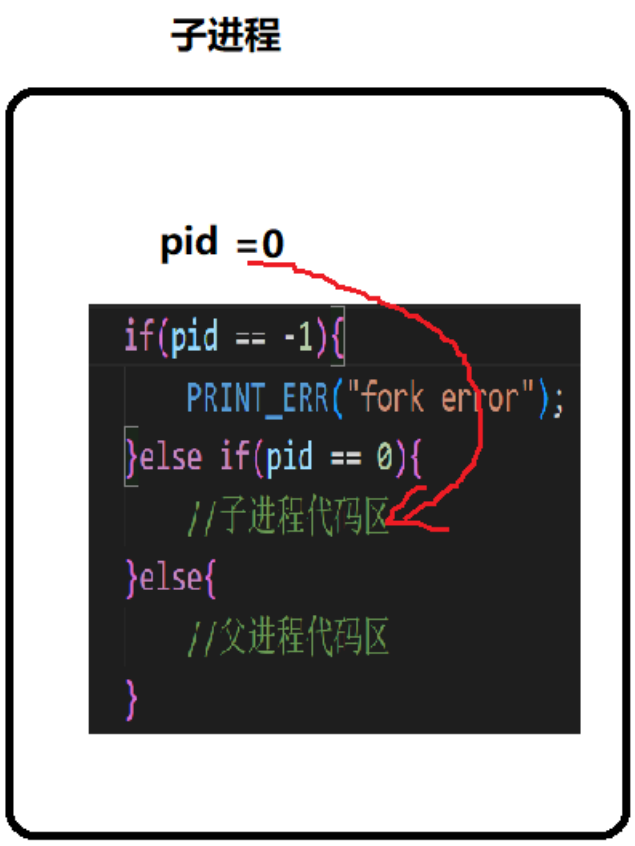


2.6关注fork返回值

```c
#include <head.h>

int main(int argc,const char * argv[])
{
    pid_t pid;
```

```
6
7       //关注返回值可以让父子进程执行不同的代码区
8       pid = fork();
9       if(pid == -1){
10          PRINT_ERR("fork error");
11      }else if(pid == 0){
12          //子进程代码区
13      }else{
14          //父进程代码区
15      }
16
17      return 0;
18  }
```





2.7父子进程执行先后顺序

父子进程执行没有先后顺序，时间片轮询，上下文切换。

2.8父子进程内存空间问题

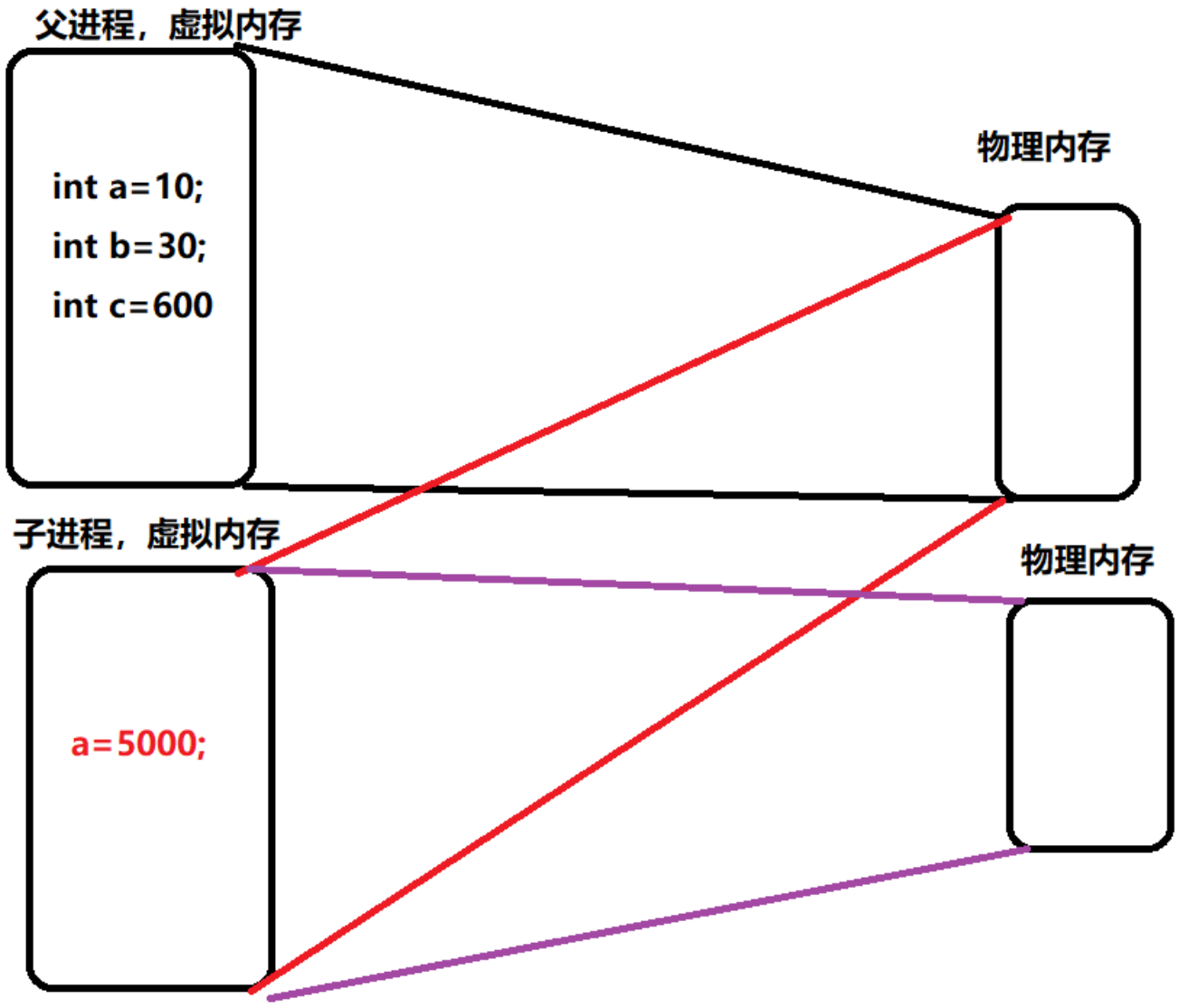
在fork前父进程中所有的变量多被拷贝到了子进程中，在子进程中

打印的变量的虚拟地址和父进程一样，但是两者在物理地址上一定

是不同的，所以在子进程内修改变量的值，父进程中变量不会改变。

父子进程内存空间相互独立。

写时拷贝



> 写时拷贝cow（copy on write）:在使用fork产生子进程的时候，
>
> 此时父子进程共用同一块物理内存，但是在子进程或者父进程
>
> 中尝试修改a变量的时候，此时就会分配一块新的物理内存，
>
> 此时父子进程a变量虚拟内存相同，但是对应的物理内存不同。

2.9多进程练习

要求：使用两个进程拷贝同一个文件，父进程拷贝文件

的前一半，子进程拷贝文件的后一半。

./a.out srcfile destfile

多进程拷贝文件的代码实现：

```c
#include <head.h>
int get_file_len(const char* file)
{
    int fd, len;
    if ((fd = open(file, O_RDONLY)) == -1)
        PRINT_ERR("open error");
    len = lseek(fd, 0, SEEK_END);
    close(fd);
    return len;
}
int init_src_file(const char* file)
{
    int fd;
    if ((fd = open(file, O_WRONLY | O_CREAT | O_TRUNC, 0666)) == -1)
        PRINT_ERR("open error");

    close(fd);
    return 0;
}
int copy_file(const char* src, const char* dest, int start, int len)
{
    char s[20];
    int fd1, fd2;
    int ret, count = 0;
    // 1.以只读方式打开源文件，以只写方式打开目标文件
    if ((fd1 = open(src, O_RDONLY)) == -1)
        PRINT_ERR("open src error");
    if ((fd2 = open(dest, O_WRONLY)) == -1)
        PRINT_ERR("open dest error");
    // 2.定位源和目标文件的光标
    lseek(fd1, start, SEEK_SET);
    lseek(fd2, start, SEEK_SET);
```

```c
    // 3.循环拷贝
    // while (1) {
    //     ret = read(fd1, s, sizeof(s)); // 从源文件中读
    //     count += ret; // 将每次读的数据加到count中
    //     if (count >= len) {
    //         write(fd2, s, (ret - (count - len)));
    //         break;
    //     }
    //     write(fd2, s, ret);
    // }
    while (count < len) {
        ret = read(fd1, s, sizeof(s)); // 从源文件中读
        count += ret; // 将每次读的数据加到count中
        write(fd2, s, ret);
    }
    // 4.关闭文件
    close(fd1);
    close(fd2);
    return 0;
}

int main(int argc, const char* argv[])
{
    int len;
    pid_t pid;
    // 1.检查参数个数
    if (argc != 3) {
        fprintf(stderr, "input error,try again\n");
        fprintf(stderr, "usage:./a.out srcfile destfile\n");
        return -1;
    }
    // 2.获取源文件大小
    len = get_file_len(argv[1]);

    // 3.创建出目标文件，并清空
    init_src_file(argv[2]);

    // 4.fork进程，拷贝文件
    if ((pid = fork()) == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程
        copy_file(argv[1], argv[2], len / 2, (len - len / 2));
    } else {
        // 父进程
        copy_file(argv[1], argv[2], 0, len / 2);
        wait(NULL);
    }

    return 0;
}
```

父子进程光标问题：

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    pid_t pid;
    int fd;

    //如果文件是在fork前打开的，此时父子进程共用同一个光标，
    //如果在父进程修改光标，子进程会受到影响。如果不想让两个
    //进程共用光标，可以在两个进程内分别打开。
    if((fd = open("./hello.txt",O_RDWR))==-1)
        PRINT_ERR("open error");

    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
      // 子进程
      lseek(fd,5,SEEK_SET);
    } else {
      //父进程
      sleep(1);
      char ch;
      read(fd,&ch,1);
      printf("ch = %c\n",ch);
    }
    close(fd);
    return 0;
}
```

3.进程相关的API接口

3.1getpid/getppid

3.1.1getpid/getppid函数的功能

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
功能：获取当前进程进程号
pid_t getppid(void);
功能：获取父进程进程号
```

3.1.2getpid/getppid函数的实例

```c
#include <head.h>

int main(int argc, const char* argv[])
{
    pid_t pid;

    // 关注返回值可以让父子进程执行不同的代码区
    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程代码区
        printf("child:pid = %d,ppid = %d\n",getpid(),getppid());
    } else {
        // 父进程代码区
        printf("parent:pid = %d,ppid = %d,cpid = %d\n",getpid(),getppid(),pid);

    }
    return 0;
}
```

3.2exit/_exit函数

3.2.1exit/_exit函数的功能

```
void exit(int status);
功能：结束进程，它是库函数，当使用exit结束进程时会刷新缓冲区
参数：
    @status:进程退出的状态值 [0-255]
        EXIT_SUCCESS (0)
        EXIT_FAILURE (1)
返回值：无
void _exit(int status);
功能：结束进程，它是系统调用，当使用_exit结束进程时不会刷新缓冲区
参数：
    @status:进程退出的状态值 [0-255]
        EXIT_SUCCESS (0)
        EXIT_FAILURE (1)
返回值：无
```

3.2.2exit/_exit函数的实例

```c
#include <head.h>
void func(void)
{
    printf("111111111\n"); //打印
    printf("2222222222");  //打印
    exit(EXIT_SUCCESS);    //进程退出，子进程变成了僵尸进程
    printf("3333333333\n");//不会执行
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    // 关注返回值可以让父子进程执行不同的代码区
    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程代码区
        func();
        while(1); //不会执行
    } else {
        // 父进程代码区
        while (1);
```

```
    }
    return 0;
}
```

```c
#include <head.h>
void func(void)
{
    printf("111111111\n"); //打印
    printf("2222222222");  //执行了，但是不会打印到终端，因为没刷新缓冲区
    _exit(EXIT_SUCCESS);   //退出进程，进程变成僵尸进程
    printf("3333333333\n");//没有执行
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    // 关注返回值可以让父子进程执行不同的代码区
    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程代码区
        func();
        while(1); //没有执行
    } else {
        // 父进程代码区
        while (1);
    }
    return 0;
}
```

3.3wait/waitpid函数

3.3.1wait/waitpid函数功能

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
功能：在父进程中调用wait回收子进程的资源（阻塞等待子进程结束）
参数：
    @wstatus:接收到子进程_exit/exit退出的状态值
返回值：成功返回回收掉资源的进程号，失败返回-1，置位错误码
wait(NULL); //回收子进程的资源，但不关注子进程退出状态

int wstatus;
wait(&wstatus);
WIFEXITED(wstatus):如果进程遇到exit/_exit/return正常退出，这个宏返回真
WEXITSTATUS(wstatus)：获取wstatus中bit8-bit15这8个bit，代表子进程退出的状态
WIFSIGNALED(wstatus)：如果是信号导致进程退出，这个宏返回真
WTERMSIG(wstatus)：获取信号号，信号号对应的是wstatus中bit0-bit6这7个bit位

pid_t waitpid(pid_t pid, int *wstatus, int options);
功能：指定回收pid号进程的资源
参数：
    @pid:进程号
        < -1    回收pid绝对值同组的任意的子进程的资源
        -1      回收任意子进程的资源
        0       回收和调用进程同组的子进程的资源
        > 0     表示回收pid对应的子进程的资源
    @wstatus:收到调用的子进程退出的状态
    @options:
            0         :阻塞回收
            WNOHANG   :非阻塞回收
返回值：成功返回回收掉的子进程的pid
         如果是非阻塞，没有回收掉子进程返回0
         如果失败返回-1置位错误码

wait(NULL)  ===等价于 == waitpid(-1,NULL,0)
wait(&wstatus) ===等价于 == waitpid(-1,&wstatus,0)
```

3.3.2wait函数使用实例

```c
#include <head.h>
void func(void)
{
    printf("111111111\n");
    printf("2222222222");
    // while(1);
    _exit(34);
```

```
    printf("3333333333\n");
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    // 关注返回值可以让父子进程执行不同的代码区
    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程代码区
        func();
        while(1);
    } else {
        // 父进程代码区
        int wstatus;
        pid_t pid1;
        if((pid1 = wait(&wstatus))==-1)
            PRINT_ERR("wait error");

        printf("pid = %d,pid1 = %d\n",pid,pid1);
        if(WIFEXITED(wstatus)){ //如果为真是正常退出的
            printf("status = %d\n",WEXITSTATUS(wstatus));//获取wstatus中bit8-bit15
        }
        if(WIFSIGNALED(wstatus)){//信号导致子进程退出
            printf("signo = %d\n", WTERMSIG(wstatus)); //获取wstatus中bit0-bit6
        }
        while(1);
    }
    return 0;
}
```

3.3.3waitpid函数实例

```
#include <head.h>
void func(void)
{
    printf("111111111\n");
    printf("2222222222");
    sleep(1);
    printf("我是第一个子进程\n");
    // while (1);
    _exit(34);
    printf("3333333333\n");
}
int main(int argc, const char* argv[])
{
    pid_t pid;

    // 关注返回值可以让父子进程执行不同的代码区
    pid = fork();
    if (pid == -1) {
        PRINT_ERR("fork error");
    } else if (pid == 0) {
        // 子进程代码区
        func();
        while (1)
            ;
    } else {
        pid_t pid1;
        if ((pid1 = fork()) == -1) {
            PRINT_ERR("fork error");
        } else if (pid1 == 0) {
            sleep(1);
            printf("我是第二个子进程\n");
            exit(0);
        }

        // 父进程代码区
        int wstatus;
        pid_t pid2;
        if ((pid2 = waitpid(-1, &wstatus, WNOHANG)) == -1)
            PRINT_ERR("wait error");

        printf("pid = %d,pid2 = %d\n", pid, pid2);
        if (WIFEXITED(wstatus)) { // 如果为真是正常退出的 ,bit8-bit15
            printf("status = %d\n", WEXITSTATUS(wstatus));
        }
        if (WIFSIGNALED(wstatus)) {
            printf("signo = %d\n", WTERMSIG(wstatus)); // bit0-bit6
        }
        while (1);
```

```
    }
    return 0;
}
```