第五天 进程控制

1 学习目标

- 了解进程相关的概念
- 掌握 fork/getpid/getppid 函数的使用
- 熟练掌握 ps/kill 命令的使用
- 熟练掌握 execl/execlp 函数的使用
- 说出什么是孤儿进程什么是僵尸进程
- 熟练掌握 wait 函数的使用
- 熟练掌握 waitpid 函数的使用

2 进程相关概念

2.1 程序和进程

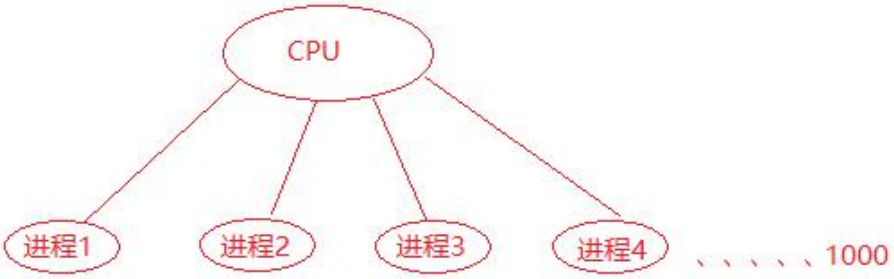
- 程序，是指编译好的二进制文件，在磁盘上，占用磁盘空间，是一个静态的概念.
- 进程，一个启动的程序， 进程占用的是系统资源，如：物理内存，CPU，终端等，是一个动态的概念
- 程序 → 剧本(纸)
- 进程 → 戏(舞台、演员、灯光、道具...)

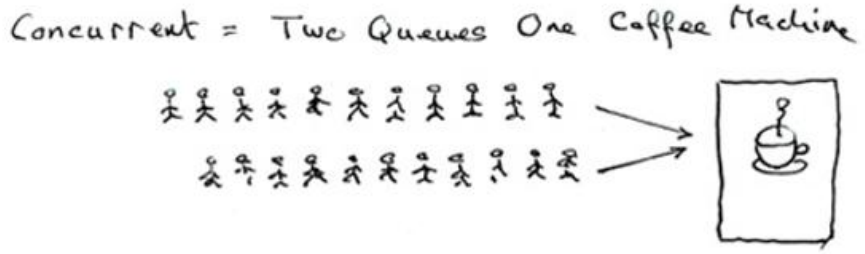
同一个剧本可以在多个舞台同时上演。同样，同一个程序也可以加载为不同的进程(彼此之间互不影响)

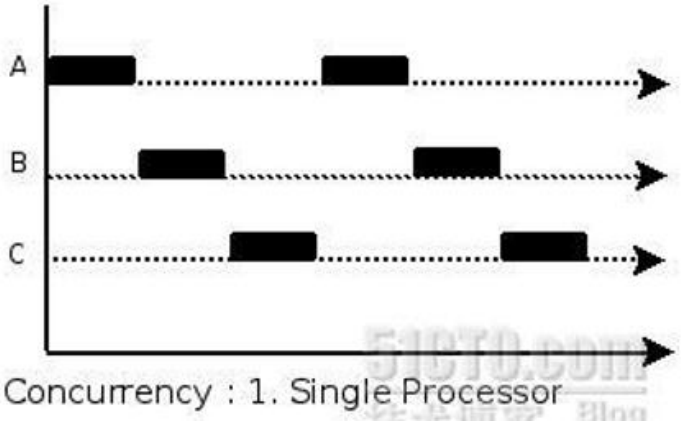
2.2 并行和并发

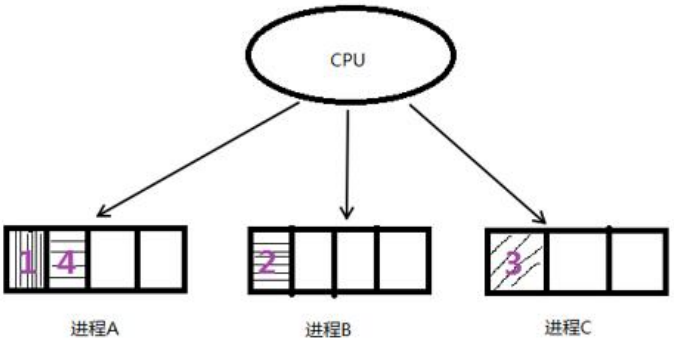
- 并发，在一个时间段内，是在同一个 cpu 上，同时运行多个程序。

  如：若将 CPU 的 1S 的时间分成 1000 个时间片，每个进程执行完一个时间片必须无条件让出 CPU 的使用权，这样 1S 中就可以执行 1000 个进程。

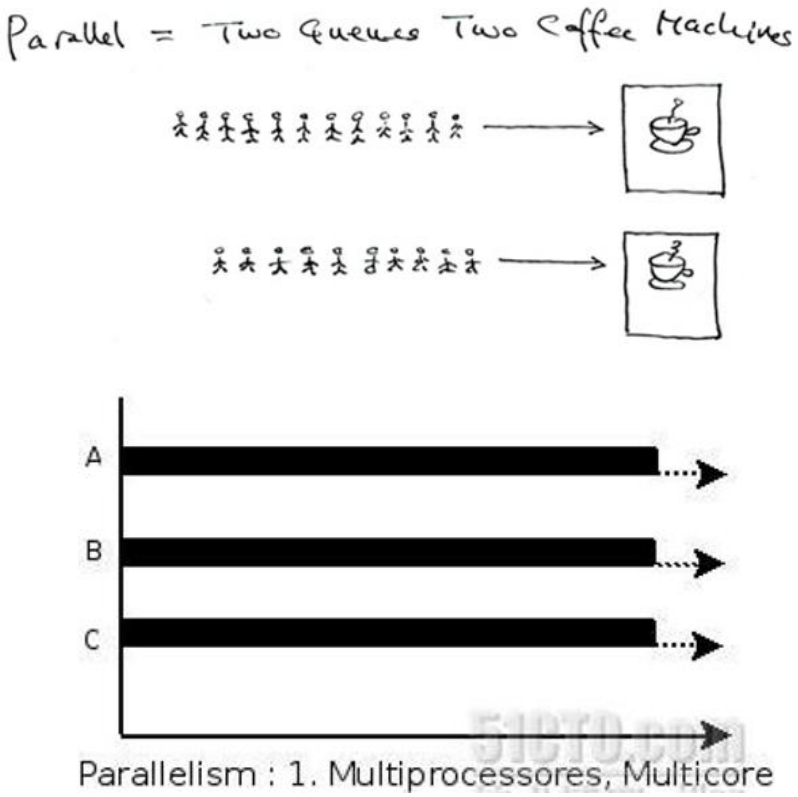








- 并行性指两个或两个以上的程序在同一时刻发生(需要有多颗)。



2.3 PCB-进程控制块

每个进程在内核中都有一个进程控制块（PCB）来维护进程相关的信息，Linux 内核的进程控制块是 task_struct 结构体。

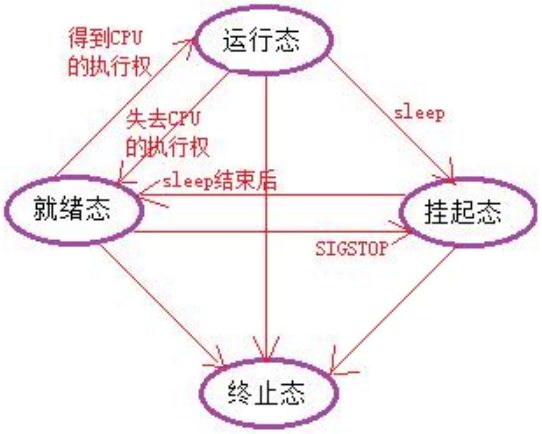
/usr/src/linux-headers-4.4.0-96/include/linux/sched.h 文件的 1390 行处可以查看 struct task_struct 结构体定义。其内部成员有很多，我们重点掌握以下部分即可：

- 进程 id。系统中每个进程有唯一的 id，在 C 语言中用 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）。
  - getcwd --pwd
- 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）。
  - ulimit -a

2.4 进程状态(面试考)

- 进程基本的状态有 5 种。分别为**初始态，就绪态，运行态，挂起态与终止态**。其中初始态为进程准备阶段，常与就绪态结合来看。



3 创建进程

3.1 fork 函数

- 函数作用：创建子进程
- 原型: pid_t fork(void);
  函数参数：无
  返回值：调用成功:父进程返回子进程的 PID，子进程返回 0;
  调用失败:返回-1，设置 errno 值。
- fork 函数代码片段实例

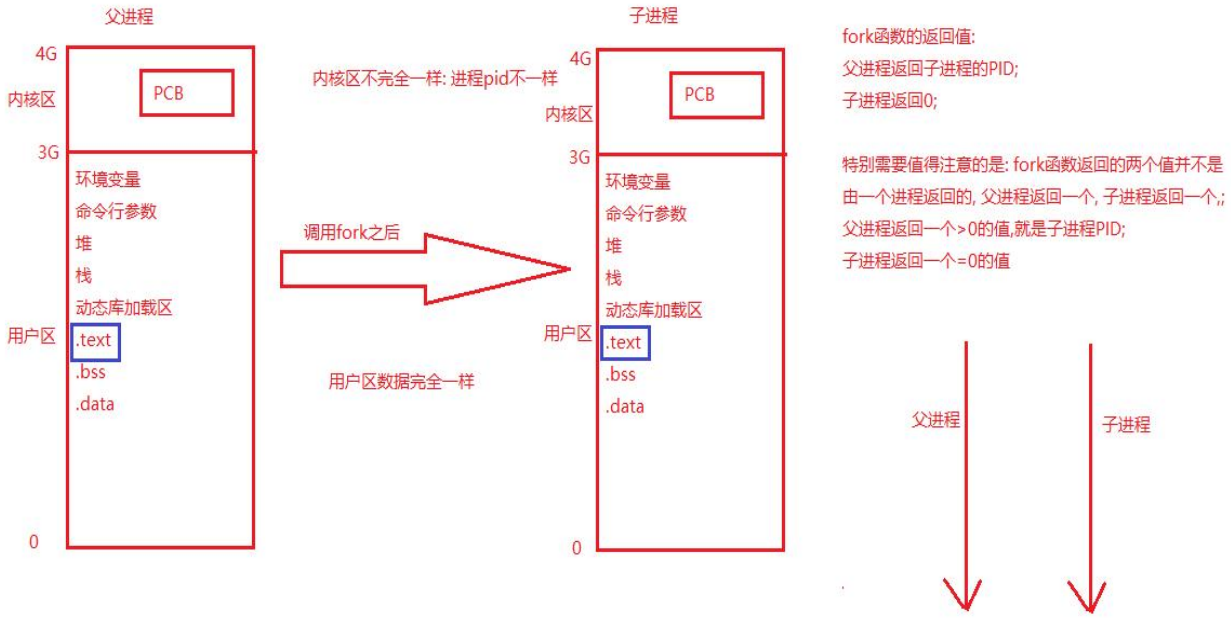
Parent

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

Child

```
int main()
{
    pid_t pid;
    char *message;
    int n;
    pid = fork();
    if (pid < 0) {
        perror("fork failed");
        exit(1);
    }
    if (pid == 0) {
        message = "This is the child\n";
        n = 6;
    } else {
        message = "This is the parent\n";
        n = 3;
    }
    for(; n > 0; n--) {
        printf(message);
        sleep(1);
    }
    return 0;
}
```

- 调用 fork 函数的内核实现原理:



- fork 函数总结

►fork 函数的返回值?

父进程返回子进程的 PID，是一个大于 0 数;

子进程返回 0;

特别需要注意的是：不是 fork 函数在一个进程中返回 2 个值，而是在父子进程各自返回一个值。

►子进程创建成功后，代码的执行位置?

父进程执行到什么位置，子进程就从哪里执行

►如何区分父子进程

通过 fork 函数的返回值

►父子进程的执行顺序

不一定，哪个进程先抢到 CPU，哪个进程就先执行

3.2 ps 命令和 kill 命令

- ps aux | grep "xxx"
- ps ajx | grep "xxx"
  - -a：（all）当前系统所有用户的进程
  - -u：查看进程所有者及其他一些信息
  - -x：显示没有控制终端的进程 -- 不能与用户进行交互的进程【输入、输出】
  - -j: 列出与作业控制相关的信息
- kill -l 查看系统有哪些信号
- kill -9 pid 杀死某个线程

3.3 getpid/getppid

- getpid - 得到当前进程的 PID

  pid_t getpid(void);
- getppid - 得到当前进程的父进程的 PID

  pid_t getppid(void);

3.3 练习题

- 编写程序，循环创建多个子进程，要求如下：
  1. 多个子进程是兄弟关系。
  2. 判断子进程是第几个子进程

     画图讲解创建多个子进程遇到的问题

     注意：若让多个子进程都是兄弟进程，必须不能让子进程再去创建新的子进程。
- 编写程序，测试父子进程是否能够共享全局变量

  重点通过这个案例讲解读时共享，写时复制

4 exec 函数族

4.1 函数作用和函数介绍

有的时候需要在一个进程里面执行其他的命令或者是用户自定义的应用程序，此时就用到了 exec 函数族当中的函数。

使用方法一般都是在父进程里面调用 fork 创建处子进程，然后在子进程里面调用 exec 函数。

- execl 函数

函数原型: int execl(const char *path, const char *arg, ... /* (char *) NULL */);

参数介绍:

- path: 要执行的程序的绝对路径
- 变参 arg: 要执行的程序的需要的参数
- arg:占位，通常写应用程序的名字
- arg 后面的: 命令的参数
- 参数写完之后: NULL

返回值: 若是成功，则不返回，不会再执行 exec 函数后面的代码；若是失败，会执行 execl 后面的代码，可以用 perror 打印错误原因。

execl 函数一般执行自己写的程序。

- execlp 函数

函数原型: int execlp(const char *file, const char *arg, .../* (char *) NULL */);

参数介绍:

- file: 执行命令的名字，根据 PATH 环境变量来搜索该命令
- arg:占位
- arg 后面的: 命令的参数
- 参数写完之后: NULL

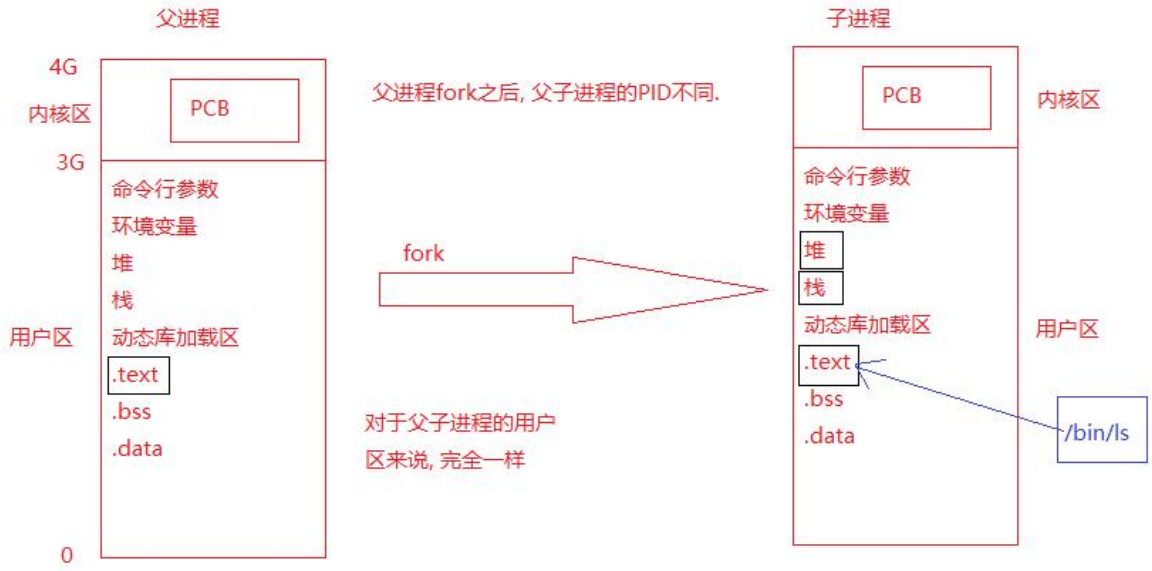
返回值: 若是成功，则不返回，不会再执行 exec 函数后面的代码；若是失败，会执行 exec 后面的代码，可以用 perror 打印错误原因。

execlp 函数一般是执行系统自带的程序或者是命令.

4.2 exec 函数族原理介绍

exec 族函数的实现原理图:

如: execlp("ls", "ls", "-l", NULL);



总结:

exec 函数是用一个新程序替换了当前进程的代码段、数据段、堆和栈；原有的进程空

间没有发生变化，并没有创建新的进程，进程 PID 没有发生变化。

4.3 exec 函数练习

- 使用 execl 函数执行一个用户自定义的应用程序
- 使用 execlp 函数执行一个 linux 系统命令

注意：当 execl 和 execlp 函数执行成功后，不返回，并且不会执行 execl 后面的代码逻辑，原因是调用 execl 函数成功以后，exec 函数指定的代码段已经将原有的代码段替换了。

5 进程回收

5.1 为什么要进行进程资源的回收

当一个进程退出之后，进程能够回收自己的用户区的资源，但是不能回收内核空间的 PCB 资源，必须由它的父进程调用 wait 或者 waitpid 函数完成对子进程的回收，避免造成系统资源的浪费。

5.2 孤儿进程

- 孤儿进程的概念:
  若子进程的父进程已经死掉，而子进程还存活着，这个进程就成了孤儿进程。
- 为了保证每个进程都有一个父进程，孤儿进程会被 init 进程领养，init 进程成为了孤儿进程的养父进程，当孤儿进程退出之后，由 init 进程完成对孤儿进程的回收。
- 模拟孤儿进程的案例
  编写模拟孤儿进程的代码讲解孤儿进程，验证孤儿进程的父进程是否由原来的父进程变成了 init 进程。

5.3 僵尸进程

- 僵尸进程的概念:
  若子进程死了，父进程还活着， 但是父进程没有调用 wait 或 waitpid 函数完成对子进程的回收，则该子进程就成了僵尸进程。
- 如何解决僵尸进程
  - 由于僵尸进程是一个已经死亡的进程，所以不能使用 kill 命令将其杀死
  - 通过杀死其父进程的方法可以消除僵尸进程。
    杀死其父进程后，这个僵尸进程会被 init 进程领养，由 init 进程完成对僵尸进程的回收。
- 模拟僵尸进程的案例
  编写模拟僵尸进程的代码讲解僵尸进程，验证若子进程先于父进程退出，而父进程没有调用 wait 或者 waitpid 函数进行回收，从而使子进程成为了僵尸进程.

5.4 进程回收函数

- wait 函数
  - 函数原型：
    pid_t wait(int *status);
  - 函数作用：
    - ✧ 阻塞并等待子进程退出
    - ✧ 回收子进程残留资源
    - ✧ 获取子进程结束状态(退出原因)。
  - 返回值：
    - ✧ 成功：清理掉的子进程 ID；
    - ✧ 失败：-1 (没有子进程)
  - status 参数：子进程的退出状态 -- 传出参数
    - ✧ WIFEXITED(status)：为非 0 → 进程正常结束
      WEXITSTATUS(status)：获取进程退出状态
    - ✧ WIFSIGNALED(status)：为非 0 → 进程异常终止
      WTERMSIG(status)：取得进程终止的信号编号。
- wait 函数练习
  使用 wait 函数完成父进程对子进程的回收
- waitpid 函数
  - 函数原型：
    pid_t waitpid(pid_t pid, int *status, in options);
  - 函数作用
    同 wait 函数
  - 函数参数
    参数：
    pid：
    pid = -1 等待任一子进程。与 wait 等效。
    pid > 0 等待其进程 ID 与 pid 相等的子进程。
    pid = 0 等待进程组 ID 与目前进程相同的任何子进程，也就是说任何和调用 waitpid()函数的进程在同一个进程组的进程。
    pid < -1 等待其组 ID 等于 pid 的绝对值的任一子进程。(适用于子进程在其他组的情况)
    status：子进程的退出状态，用法同 wait 函数。
    options：设置为 WNOHANG，函数非阻塞，设置为 0，函数阻塞。
  - 函数返回值
    >0：返回回收掉的子进程 ID；
    -1：无子进程
    =0：参 3 为 WNOHANG，且子进程正在运行。
- waitpid 函数练习
  使用 waitpid 函数完成对子进程的回收

6 作业

6.1 作业 1

测试父子进程之间是否共享文件

6.2 作业 2

父进程 fork 三个子进程:

其中一个调用 ps 命令;

一个调用自定义应用程序;

一个调用会出现段错误的程序。

父进程回收三个子进程(waitpid)，并且打印三个子进程的退出状态。

=== 段错误 ===

1>. 访问了非法内存

2>. 访问了不可写的区域进行写操作

3>. 栈空间溢出

char* p = “hello，world”

p【0】=‘a’;