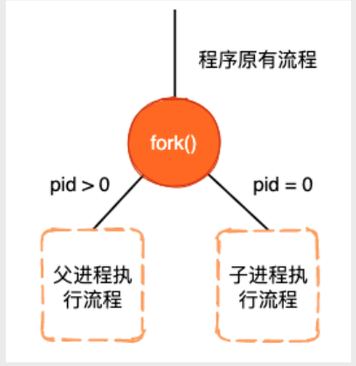
进程的创建: fork()

进程的创建

系统调用 fork() 允许一个进程（父进程）创建出一个新的进程（子进程），子进程获得父进程的堆、栈、数据段和执行文本段的拷贝，程序开始一分为二执行




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

    if (pid == 0) {
        printf("这里是子进程， pid 为 : %d \n", getpid());
    }

    if (pid > 0) {
        printf("这里是父进程, pid 为 : %d \n", getpid());
    }
}
```

- 内存拷贝并不一开始就发生
  - 从理论上来说，fork() 创建出来的子进程中的堆、栈、数据段应该是父进程的完整拷贝。但是，在大部分情况下，fork() 在调用后会直接调用 exec() 函数族来执行其它的程序或者是命令，重新的初始化堆、栈以及数据段等内容
  - 因此，如果 fork() 在一开始就进行内存拷贝，很多情况下会被视为"无用之举"。因此，内核会使用写时复制的方式进行 lazy copy。也就是说当子进程确实需要这一份内存时，内核才会进行拷贝动作
- 创建多个子进程
  - 在一般的业务需求下，通常都是由当前进程创建出多个子进程以供使用

```
int main() {
    pid_t pid;

    for (int i = 0; i < 4; i++) {
        pid = fork();

        if (pid <= 0)    // 子进程不允许进入 for 循环内，保证进程只由父进程创建
            break;
    }

    if (pid == 0)       // 子进程跳出循环以后，我们可以在这里判断当前进程
}
```

监控子进程

- 在绝大多数程序设计中，父进程可能会创建出多个子进程出来，并且父进程需要知道子进程在何时停止或者何时退出
- wait() 与 waitpid()
  - 这两个系统调用均可以使得父进程等待子进程的结束。区别在于 wait() 调用不能确切等待某一个子进程结束，并且也没有办法进行非阻塞的等待
  - 因此，在绝大多数的情况下，我们都是用 waitpid() 这一系统调用来等待子进程的结束
- waitpid()
  - pid_t waitpid(pid_t pid, int *status, int options);
  - pid
    - **pid > 0** — waitpid() 则会等待确切的一个子进程结束，此时 pid 为子进程进程号
    - pid = 0 — 等待与父进程同一个进程组的所有子进程
    - **pid = -1** — 等待任意子进程的结束，使用较多
    - pid < -1 — 等待进程组标识符与 pid 绝对值相等的所有子进程
  - status — 子进程的终止状态将写入至该指针变量中，用来表示子进程是以哪种方式结束的。因为 status 虽然定义成一个整型，但是实际上只用到了其最低的两个字节，所以我们需要使用一些宏来对其进行断言
  - options
    - 位掩码，可以按位或多个选项
    - WUNTRACED — 除了返回终止子进程的信息外，还返回因信号而停止的子进程信息。
    - WCONTINUED — 返回那些因收到 SIGCONT 信号而恢复执行的已停止子进程的状态信息。
    - **WNOHANG** — **非阻塞的等待，若子进程的状态并未发生改变，那么 waitpid() 则返回 0**
- 调用 waitpid() 的必要性
  - **如果父进程创建了某一子进程，但并未执行 waitpid()，那么在内核的进程表中将为该子进程永久保留一条记录。并且，即使是银弹 SIGKILL 也无法"杀死"这个子进程，使其从进程记录表中移除，因此，这样的子进程我们通常称之为僵尸进程**
  - 如果存在大量此类僵尸进程，它们势必将填满内核进程表，从而阻碍新进程的创建
  - 既然无法用信号杀死僵尸进程，那么从系统中将其移除的唯一方法就是杀掉它们的父进程 (或等待其父进程终止)，此时 init 进程将接管和等待这些僵尸进程，从而从系统中将它们清理掉

SIGCHLD

- 前面我们讨论了调用 waitpid() 的必要性: 不能让子进程变成僵尸进程。但是，waitpid() 要么阻塞调用，要么只能使用轮询的方式使用，多少会有些不便，有没有一种方式使得子进程结束时，通知到父进程，然后父进程再调用 waitpid() 呢?
- **答案就是 SIGCHLD。当子进程执行完毕并退出时，系统将会给其父进程发送一个 SIGCHLD 信号。因此，我们就可以为 SIGCHLD 注册一个信号处理函数，在该函数内部，直接使用非阻塞的方式调用 waitpid()**
- 不对信号进行排队处理
  - SIGCHLD 为标准信号，因此 Linux 内核并不会对其进行排队处理
  - 也就是说，假设当前父进程有 4 个子进程，这些子进程先后退出时，父进程可能只能收到 3 个或者其它个数的 SIGCHLD 信号
  - 如此一来，就算我们调用了 waitpid() ，仍然可能会漏掉一些子进程，使其变为僵尸进程
- 上一个问题的处理方式就是在信号处理函数中循环的调用 waitpid() 方法，直到没有子进程退出为止

```
void handler(int signum) {

    // do something

    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;

    // do something
}
```

- 这个循环会一直进行下去，直到再也没有僵尸子进程为止