进程的创建、执行与终止

fork

原型: pid_t fork(void);

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

    childPid = fork();

    if (childPid == -1) { /* error occured, handle error */ }
    else if (childPid == 0) { /* child process */ }
    else { /* parent process */ }

    return 0;
}
```

- fork将创建一个与父进程几乎完全相同的子进程
- 父进程将返回子进程pid，子进程返回0，-1表示调用错误
  - 子进程返回0的原因是一个子进程只会有一个父进程，并且可以通过getppid()获得父进程PID
  - 父进程返回子进程PID的原因在于一个进程可有多个子进程，父进程需要对其进行记录

父、子进程间的文件共享

- 子进程会获得父进程所有文件描述符的副本，文件描述符均指向同一个已打开的文件句柄
- 当子进程修改文件的偏移量时，父进程中该文件偏移量同样会改变

文件描述符不同，但文件相同
(子进程关闭文件描述符不会影响父进程)

fork()的内存语义

- 在早期的UNIX实现中，创建子进程时立即拷贝父进程的程序段、堆段、栈段以及数据段 — 过于浪费资源
- 现代UNIX实现
  - 标记代码段(执行程序)为只读，从而使得父、子进程共享代码段，减少数据拷贝
  - 初始化时父、子进程共享进程内存空间(堆、栈以及数据段等)，只有在必须时才进行部分数据的拷贝
    - 虚拟内存采用分页+分段管理机制
    - 父、子进程只有将要修改某页数据时才会进行数据页的拷贝，而非立即拷贝，同时也不是大段大段内存数据拷贝

Copy-On-Write

vfork

原型: pid_t vfork(void);

- vfork将创建一个与父进程完全相同的子进程，调用时:
  父进程将返回子进程pid，子进程返回0，-1表示调用错误
- vfork的设计初衷为子进程立即执行exec()函数群

vfork()调用产生的子进程将完全共享父进程内存空间，直至子进程成功执行exit或_exit退出

- 父进程将会暂时挂起，直到子进程执行完毕

在子进程调用exit()或_exit()退出之前，不会调度父进程

- 系统将保证子进程的优先于父进程调度以使用CPU，而fork调用则无此保证

效率高于fork，但是其语义与fork完全不同

clone

原型: int clone(int (*func) (void *), void *child_stack, int flags, void *func_arg)

- clone主要用于线程库的实现
- **与fork不同的是，子进程继续运行时不以调用处为起点，而是调用参数func所指向的函数**

在Linux内核下，fork、vfork以及clone最终均由do_fork实现，也就是说，Linux中的线程与进程可以认为是同一个东西

flags: clone()调用最为重要，且最为关键的参数

- 低字节: 存放子进程的终止信号(SIGCHLD)
- 其余字节: 用于控制clone()的操作

flags由一系列的位掩码进行"或"操作得到

flags常用位掩码

- CLONE_FILES — 父、子进程将共享同一个打开的文件描述符
  - fork: 无需该标识位
  - POSIX线程则要求进程中所有线程共享文件描述符
- CLONE_FS — 父、子进程将共享与文件系统相关的信息，即无论在哪个进程调用umask()、chdir()或chroot()，都将影响另一个进程
  - fork: 无需该标识位
  - POSIX线程要求进程中所有线程文件系统的属性信息
- CLONE_VM — 父、子进程将共享同一份进程空间(虚拟内存页)
  - fork: 无需该标识位; vfork: 相当于设置该掩码
  - POSIX线程要求进程中所有线程共享进程内存空间
- CLONE_SIGHAND — 父、子进程将共享对信号的处置设置
- CLONE_VFORK — 挂起父进程直至子进程调用exit()或_exit()退出 — vfork: 相当于设置该掩码
- CLONE_NEWNET — 子进程将获得新的网络命名空间
- CLONE_NEWNS — 子进程将获得父进程mount命名空间的副本
- CLONE_NEWPID — 子进程将获取新的进程ID命名空间
- CLONE_NEWUSER — 子进程获得新的用户ID命名空间

namespace系列位掩码
可用于容器技术(例如Docker)

clone标识的使用

- fork — flags: SIGCHLD
- vfork — flags: CLONE_VM | CLONE_FORK | SIGCHLD
- Thread (NPTL) — flags: CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM

exit & _exit

库函数exit(int status) — 终止一进程，并将进程的所有资源(申请的内存、文件描述符等)归还内核，status为进程退出状态

1. 调用退出处理程序(由atexit()和on_exit()注册的函数)
2. **刷新stdio流缓冲区** — 因为此特性的存在，故父、子进程通常一个调用exit()，一个调用_exit()退出
3. 使用由status提供的值执行_exit()系统调用

系统调用_exit(int status) — 为库函数exit()的底层实现，作用同样为正常终止一进程

虽然status类型为int，但是只有低8位可为父进程所用(0~255)。而shell脚本使用的退出状态为128+退出状态值，故通常使用0~127作为退出状态值

wait & waitpid

系统调用wait(&status)

- 若子进程状态尚未终止，则wait()会挂起父进程直至子进程终止
- 子进程的终止状态通过wait()的status指针返回
- 若在调用wait()之前子进程已终止，则立即返回

wait()不能指定等待某一子进程的终止

系统调用waitpid() — 允许等待某一具体的子进程、同一进程组的所有子进程、任意子进程的终止

孤儿进程与僵尸进程

孤儿进程: 子进程结束之前父进程首先结束，此时init进程会接管孤儿进程

僵尸进程: 父进程在调用wait()之前，子进程就已经终止

- 僵尸进程无法通过SIGKILL杀死，只能由父进程通过wait()调用由内核回收相关资源
- 若父进程中无wait()调用，则僵尸进程由init进程接管并自动调用wait()移除僵尸进程
- 若父进程创建某一子进程并且无wait()调用，那么内核的进程表将为子进程永久保留一条记录。如果存在大量僵尸进程，势必会填满内核进程表，从而阻碍新进程的创建。
- 进程表已满时init进程会接管僵尸进程，并调用wait进行进程清理

在设计长生命周期的父进程(如web server或Shell)时，wait()与waitpid具有非常重要的语义: 清理执行完毕的子进程，避免其成为长寿僵尸进程

execve

原型: int execve(const char *pathname, char *const argv[], char *const envp[])

- pathname: 准备载入当前进程空间的新程序的路径名
- argv: 指定传给新进程的命令行参数，atgv[0]通常为载入程序名
- envp: 新程序的环境列表
- 返回值: 永远为-1

系统调用execve()将一新程序载入到某一进程空间，丢弃原有程序，替换栈、堆以及静态数据等。常用于子进程

shell就是使用execve()调用的最佳例子

system

原型: int system(const char *command) — 用于执行shell命令，如 system("ls -lah");

可使用fork、waitpid、execve以及exit等函数实现system库函数调用