进程管理与Shell

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

用户与操作系统的接口：Shell

用户如何使用操作系统?

- 与Shell执行一系列交互操作:
  - 启动程序
    - 一个或多个程序；前台/后台运行
    - 管道连接多个程序的输入/输出
  - 关闭程序
    - kill $PID
- 与应用程序交互
  - 向应用程序输入指令
  - 观察应用程序的输出

Shell

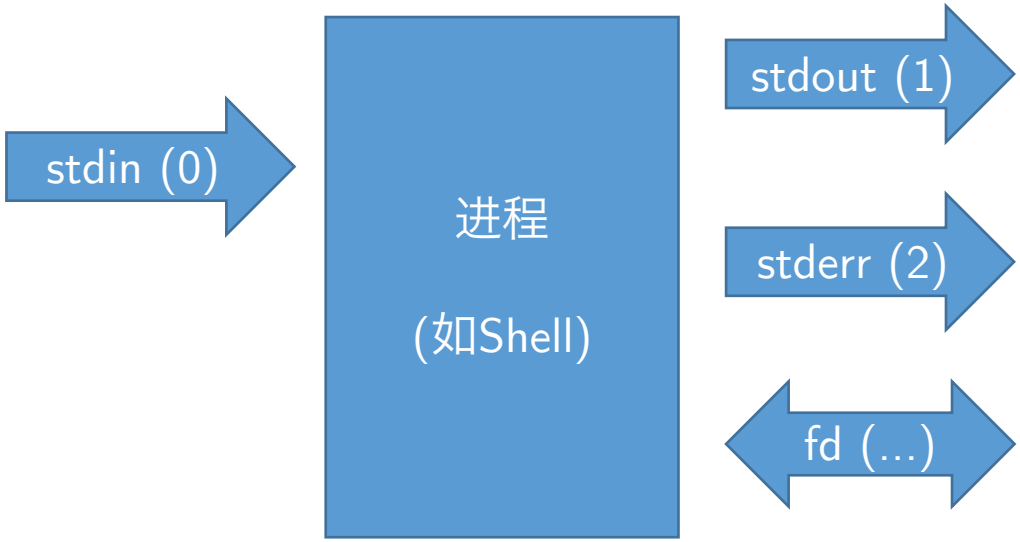
- 和用户交互的“程序管理器”
  - 从标准输入里读取命令并执行
  - 命令实际上是一个小程序(Shell是程序解释器)

- GUI实际也是一个Shell
  - 同样的循环
  - 读取鼠标/键盘事件

```
int main() {
  while (1) {
    write(2, "$ ", 2);
    buf[read(0, buf, nbuf-1)] = '\0';
    execute(buf); // 解释执行buf
  }
  return 0;
}
```

Shell和终端

- Shell的输入和输出通常是连接到终端的
  - Shell启动的进程默认输出输入也是连接到同一个终端的



  - read(0)从终端读取 write(2)向终端输出

从终端读取

- read(fd, buf, size) – fd为终端的文件描述符
  - cooked mode：终端允许行编辑；在终端没有输入时等待
    - 例子：各种命令行工具、OJ程序……
  - raw mode：终端有输入后能立即读到
    - 例子：vi, less, 游戏, ……
  - 操作系统提供设置终端的API (打开时可以non-blocking哦)

向终端输出

- write(fd, buf, size) - fd为终端的文件描述符
  - 终端 $= W \times H$的二维数组：字符、前景色、背景色、(加粗、斜体、下划线……)
  - 特殊字符序列能实现特殊功能
    - \r 回行位；\n 换行; \t 制表符
    - ESCAPE Code: 实现屏幕控制(清屏、移动光标、设置颜色……)

ANSI Escape Codes

- 使用Escape Code可以在终端上实现各类绘制
  - 比如vi的界面 (试着重定向busybox vi的输出到文件)
  - 比如终端版的LiteNES (把图像缩小到$W \times H$；清屏；输出)
  - .....

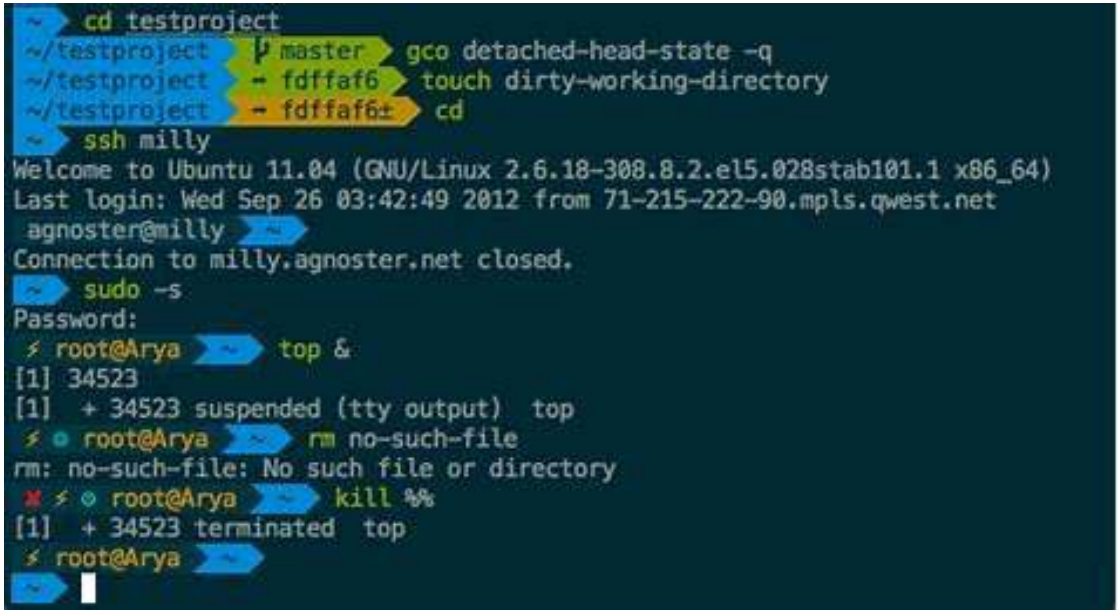
- 演示Escape Codes
  - printf可以做的事比想象要多

VT100

进程API

Shell

- 早期操作系统中最重要的应用程序之一
  - 其他应用基本只需要文件操作(everything is a file)就行了
    - grep, gcc, awk, sed, vi, pstree, ...
  - 但Shell要实现进程管理



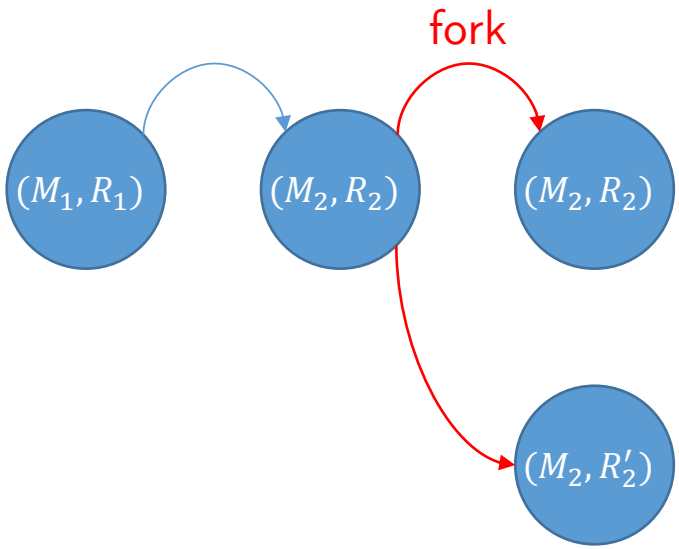
简易的Shell语言

- P ::= RUN [| RUN]... ['&']
- RUN := cmd [arg]... [[number]'>' filename]
  - cmd – 命令
  - arg – 参数
  - > - 重定向
  - & - 后台执行

- 例子
  - ./some-program 1>outfile 2>errfile
  - cat a.txt b.txt | sort -n | uniq > result.txt &

fork

- int fork();



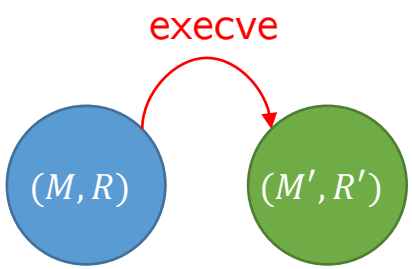
让当前进程分叉，得到除了返回值不同外完全相同的两个进程

- 父进程：fork() = 子进程进程号
- 子进程：fork() = 0

共享打开的所有文件

execve

- int execve(const char *filename, char **argv, char **envp);
  - v = argv, e = envp



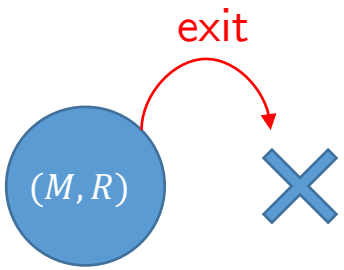
让当前进程“变身”

- 执行filename程序的main(argc, argv, envp)

打开的所有文件不变

_exit

- void _exit(int status);
  - 为什么是_exit?



让当前进程消失

- 释放所有资源

自动关闭所有打开的文件

Shell实现：The UNIX Way

- 用pid = fork()创建子进程
  - 父进程：fork()
  - 子进程：管理重定向/创建管道；执行execve()

- 进程管理
  - 父进程使用wait等待子进程
  - 用信号在进程间通信
    - signal注册事件处理
    - kill发送信号

讨论

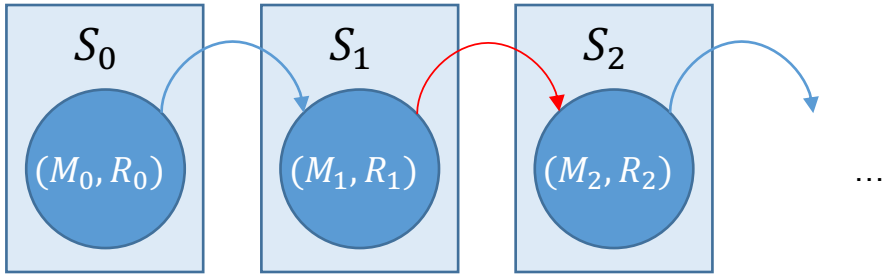
- fork, execve, exit, waitpid的好处与坏处?

- CreateProcess, WaitProcess的好处与坏处?

操作系统的本质

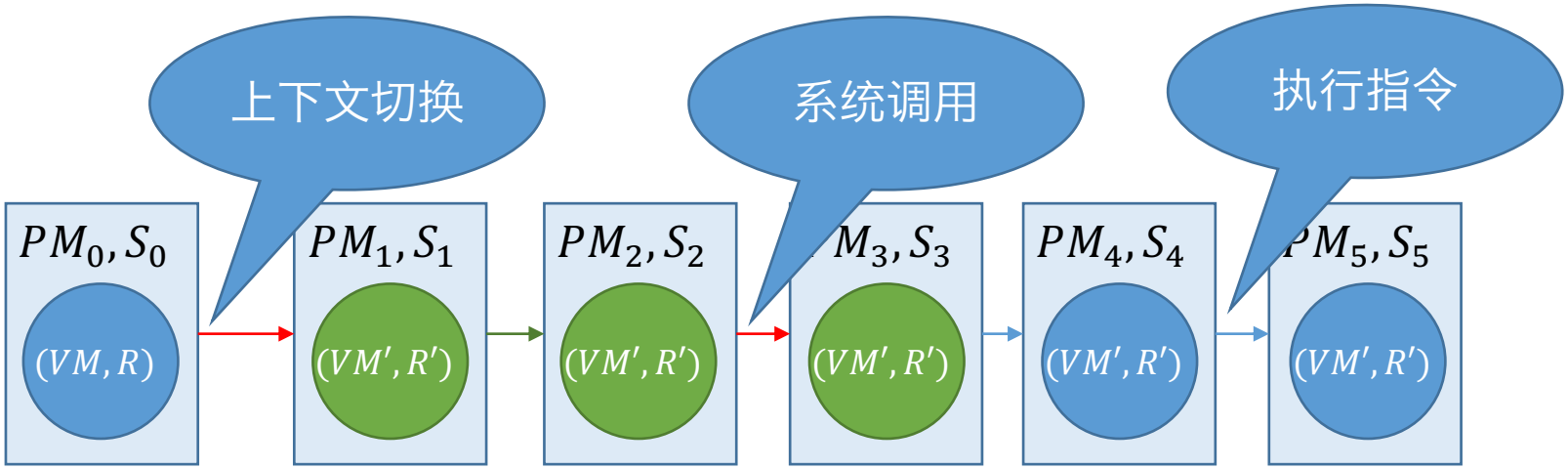
Rethinking of States

- 进程不能直接访问资源
  - 资源可以更好地被操作系统管理
- 操作系统维护了内部状态S
  - 打开的文件(文件描述符)、进程通信(信号、套接字)、……
  - 但进程本身不可见，只能通过系统调用访问
    - 禁止进程直接访问资源，实现了虚拟化



Rethinking of States (cont'd)

- 一个处理器、一份内存、一份寄存器
  - 管理多个并发执行的进程
  - 管理系统中的各类资源：所有打开的文件、设备等等

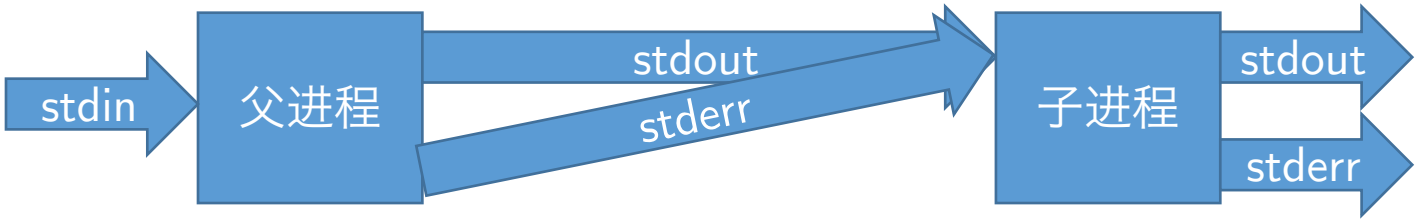


操作系统的规约(Specification)

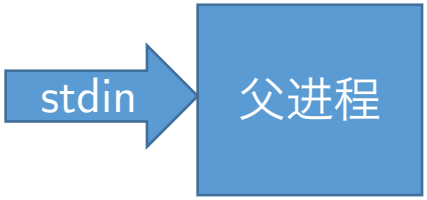
- 定义了系统调用的行为
  - 但描述行为远比想象中要复杂
    - Intel的手册；POSIX标准……无一对所有行为进行了完全精确的定义
    - C标准里有undefined/unspecified behavior; Java依靠虚拟机获得成功
- 例子：描述行为时需要考虑的情况
  - 进程之间存在共享资源
    - 通过文件描述符/handle/进程号访问
  - 共享资源可能并发访问
    - 先后读/写同一个文件；先后杀死同一个进程……
  - 与物理设备交互
    - I/O设备可能非常缓慢，但多个程序突发大量写入

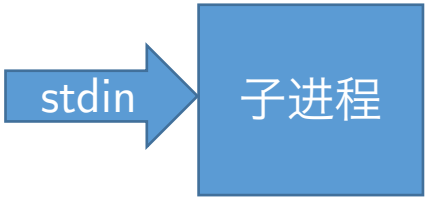
问题：多个进程的文件共享

- 前台/后台执行；终端/管道/磁盘文件行为有别；信号……



共享文件描述符的语义？





后台执行echo &
同时读取stdin

一个人很难理解操作系统中的所有行为
但要理解基本的设计原则
并能通过资料/实验理解系统的实际行为

/proc里存在一些隐藏的目录，ls/readdir看不到，但却可以cd (???)