一、进程概述

程序是包含一系列信息的文件，这些信息描述了如何在运行时创建一个进程：

- 二进制格式标识：每个程序文件都包含用于描述可执行文件格式的元信息。内核利用此信息来解释文件中的其他信息。（ELF可执行连接格式）
- 机器语言指令：对程序算法进行编码。
- 程序入口地址：标识程序开始执行时的起始指令位置。
- 数据：程序文件包含的变量初始值和程序使用的字面量值（比如字符串）。
- 符号表及重定位表：描述程序中函数和变量的位置及名称。这些表格有多重用途，其中包括调试和运行时的符号解析（动态链接）。
- 共享库和动态链接信息：程序文件所包含的一些字段，列出了程序运行时需要使用的共享库，以及加载共享库的动态连接器的路径名。
- 其他信息：程序文件还包含许多其他信息，用以描述如何创建进程。

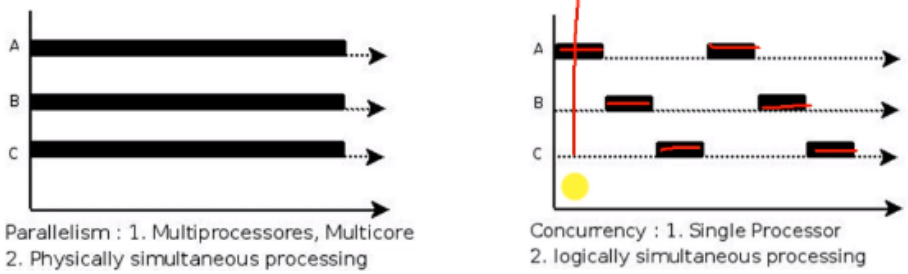
- 进程是正在运行的程序的实例。是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。它是操作系统动态执行的基本单元，在传统的操作系统中，进程既是基本的分配单元，也是基本的执行单元。
- 可以用一个程序来创建多个进程，进程是由内核定义的抽象实体，并为该实体分配用以执行程序的各项系统资源。从内核的角度看，进程由用户内存空间和一系列内核数据结构组成，其中用户内存空间包含了程序代码及代码所使用的变量，而内核数据结构则用于维护进程状态信息。记录在内核数据结构中的信息包括许多与进程相关的标识号（IDs）、虚拟内存表、打开文件的描述符表、信号传递及处理的有关信息、进程资源使用及限制、当前工作目录和大量的其他信息。

时间片（timeslice）又称为“量子（quantum）”或“处理器片（processor slice）是操作系统分配给每个正在运行的进程微观上的一段 CPU 时间。事实上，虽然一台计算机通常可能有多个 CPU，但是同一个 CPU 永远不可能真正地同时运行多个任务。在只考虑一个 CPU 的情况下，这些进程“看起来像”同时运行的，实则是轮番穿插地运行，由于时间片通常很短（在 Linux 上为 5ms－800ms），用户不会感觉到。

时间片由操作系统内核的调度程序分配给每个进程。首先，内核会给每个进程分配相等的初始时间片，然后每个进程轮番地执行相应的时间，当所有进程都处于时间片耗尽的状态时，内核会重新为每个进程计算并分配时间片，如此往复。

并行(parallel)：指在同一时刻，有多条指令在多个处理器上同时执行。

并发(concurrency)：指在同一时刻只能有一条指令执行，但多个进程指令被快速的轮换执行，使得在宏观上具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，使多个进程快速交替的执行。



并行：不同实体（多个 CPU 上）；并发：同一实体（单个 CPU 上）

进程控制块（PCB）

为了管理进程，内核必须对每个进程所做的事情进行清楚的描述。内核为每个进程分配一个 PCB(Processing Control Block)进程控制块，维护进程相关的信息，Linux 内核的进程控制块是 task_struct 结构体。

在 /usr/src/linux-headers-xxx/include/linux/sched.h 文件中可以查看 struct task_struct 结构体定义。其内部成员有很多，我们只需要掌握以下部分即可：

- 进程id：系统中每个进程有唯一的 id，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）

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

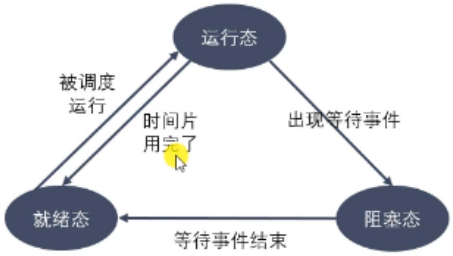
进程可以使用的资源上限（Resource Limit）

Table 1：ulimit 指令 ——> ulimit –a：资源上限

二、进程状态转换

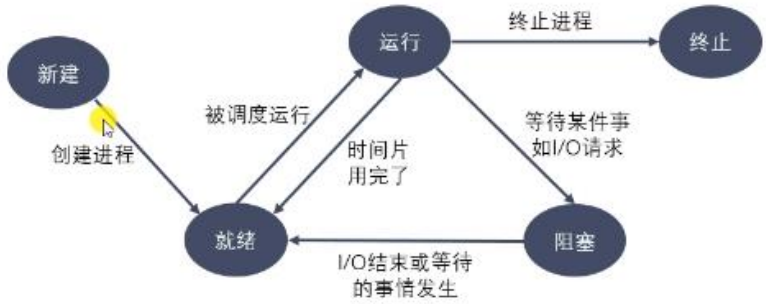
01 / 进程的状态

进程状态反映进程执行过程的变化。这些状态随着进程的执行和外界条件的变化而转换。在三态模型中，进程状态分为三个基本状态，即就绪态，运行态，阻塞态。在五态模型中，进程分为新建态、就绪态，运行态，阻塞态，终止态。



- 运行态：进程占有处理器正在运行
- 就绪态：进程具备运行条件，等待系统分配处理器以便运行。当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列
- 阻塞态：又称为等待(wait)态或睡眠(sleep)态，指进程不具备运行条件，正在等待某个事件的完成

01 / 进程的状态



- 新建态：进程刚被创建时的状态，尚未进入就绪队列
- 终止态：进程完成任务到达正常结束点，或出现无法克服的错误而异常终止，或被操作系统及有终止权的进程所终止时所处的状态。进入终止态的进程以后不再执行，但依然保留在操作系统中等待善后。一旦其他进程完成了对终止态进程的信息抽取之后，操作系统将删除该进程。

- 查看进程

```
ps aux / ajx
```

a：显示终端上的所有进程，包括其他用户的进程

u：显示进程的详细信息

x：显示没有控制终端的进程

j：列出与作业控制相关的信息

■ STAT参数意义：

D	不可中断 Uninterruptible (usually IO)
R	正在运行，或在队列中的进程
S(大写)	处于休眠状态
T	停止或被追踪
Z	僵尸进程
W	进入内存交换（从内核2.6开始无效）
X	死掉的进程
<	高优先级
N	低优先级
s	包含子进程
+	位于前台的进程组

指令：tty：查看当前终端。

■ 实时显示进程动态

top

可以在使用 top 命令时加上 -d 来指定显示信息更新的时间间隔，在 top 命令执行后，可以按以下按键对显示的结果进行排序：

- M 根据内存使用量排序
- P 根据 CPU 占有率排序
- T 根据进程运行时间长短排序
- U 根据用户名来筛选进程
- K 输入指定的 PID 杀死进程

■ 杀死进程

kill [-signal] pid

kill -l 列出所有信号

kill -SIGKILL 进程ID

kill -9 进程ID

killall name 根据进程名杀死进程

后台运行：./test &。

./test 进程的父进程是当前终端。

03 / 进程号和相关函数

- 每个进程都由进程号来标识，其类型为 pid_t（整型），进程号的范围：0～32767。进程号总是唯一的，但可以重用。当一个进程终止后，其进程号就可以再次使用。
- 任何进程（除 init 进程）都是由另一个进程创建，该进程称为被创建进程的父进程，对应的进程号称为父进程号（PPID）。
- 进程组是一个或多个进程的集合。他们之间相互关联，进程组可以接收同一终端的各种信号，关联的进程有一个进程组号（PGID）。默认情况下，当前的进程号会当做当前的进程组号。
- 进程号和进程组相关函数：
  - pid_t getpid(void);
  - pid_t getppid(void);
  - pid_t getpgid(pid_t pid);

三、创建进程

01 / 进程创建

系统允许一个进程创建新进程，新进程即为子进程，子进程还可以创建新的子进程，形成进程树结构模型。

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

返回值：

- 成功：子进程中返回 0，父进程中返回子进程 ID
- 失败：返回 -1

失败的两个主要原因：

1. 当前系统的进程数已经达到了系统规定的上限，这时 errno 的值被设置为 EAGAIN
2. 系统内存不足，这时 errno 的值被设置为 ENOMEM

Q：为什么父子进程都要执行一次 main()函数里的代码？

在一个典型的多进程程序中，当父进程调用 fork()系统调用创建子进程时，子进程将会继承父进程的代码段、数据段和堆栈等信息。这意味着子进程在创建后会具有与父进程相同的代码和执行流程。（内核的 PID 不被改变；父子进程拥有相同的内存；栈空间里的变量

一开始是共享的，一旦修改，就不共享。即读时共享，写时拷贝）

因此，无论是父进程还是子进程，它们都会从 main 函数开始执行，并按照代码的顺序逐行执行。这是因为操作系统将创建的子进程看作是一个全新的进程，并没有意识到该进程是通过复制父进程而来的，因此它会从 main 函数开始执行。需要注意的是，虽然父进程和子进程在执行相同的代码，但它们是独立的执行体，各自有自己的进程上下文和资源。子进程的执行不会影响父进程的执行，它们是并发执行的。这样的设计使得多进程编程更加灵活，可以实现并行处理和任务分配等功能。

四、父子进程虚拟地址空间

Linux 的 fork 函数使用写时拷贝。内核此时并不复制整个进程的地址空间，而是父子进程共享一个地址空间。在需要写入的时候复制地址空间。资源的复制只有在写入的时候进行，其他时间通过只读方式共享地址空间。Fork 之后父子进程共享文件。文件描述符指向相同的文件列表，引用计数增加。

Q：堆空间和栈空间的区别？

堆和栈是计算机内存中两种不同的存储方式，它们在内存管理和数据存储上有着一些重要的区别。

1. 分配方式：

- 栈（Stack）：栈的内存分配是自动的，由编译器进行管理。每当函数被调用时，函数的局部变量和参数会被分配到栈上，并在函数执

行完毕后自动释放。栈采用先进后出（FILO）的方式进行内存分配和释放。

- 堆（Heap）：堆的内存分配是手动控制的，程序员需要显式地申请和释放内存。堆上的内存分配不会自动释放，需要程序员负责管理。如果不适当地使用或忘记释放堆上的内存，可能会导致内存泄漏。

2. 空间大小：

- 栈：栈的大小是固定的，通常较小。栈的大小在程序编译时就确定，并且在运行时不可改变。栈的大小受限于操作系统和编译器的限制，一般几个 MB 到几十 MB 不等。

- 堆：堆的大小通常比栈大得多。堆的大小受限于系统内存的总量，可以动态地分配和释放内存。

3. 存储内容：

- 栈：栈用于存储局部变量、函数参数、函数调用和返回值等短期存储的数据。它的生命周期与函数的执行周期相关，随着函数的调用和返回而动态变化。

- 堆：堆用于存储动态分配的数据，例如通过 malloc()或 new 操作符在堆上申请的内存。堆上的数据可以在程序的不同部分共享和访问。

4. 内存管理：

- 栈：栈的内存管理由编译器自动完成，无需程序员干预。栈上的内存分配和释放是快速且高效的，但是其生命周期受到限制。

- 堆：堆的内存管理需要手动进行，程序员需要负责显式地申请

和释放内存。堆上的内存管理相对复杂，容易出现内存泄漏和内存访问错误的问题。

栈适合存储短期的局部变量和函数调用相关的数据，内存管理由编译器自动完成；而堆适合存储动态分配的数据，需要程序员手动管理内存。

五、GDB 多进程调试

六、exec 函数族：通过调用 exec，在一个进程中去执行另一个程序。

exec 函数族的作用是根据指定的文件名找到可执行文件，并用它来取代调用进程的内容，换句话说，就是在调用进程内部执行一个可执行文件。

```
nana@nana-virtual-machine:~/linux/linux_duojincheng$ ./ex1
parent process, pid : 25053
child process, pid : 25054
nana@nana-virtual-machine:~/linux/linux_duojincheng$ hello world
```

上图是孤儿进程。

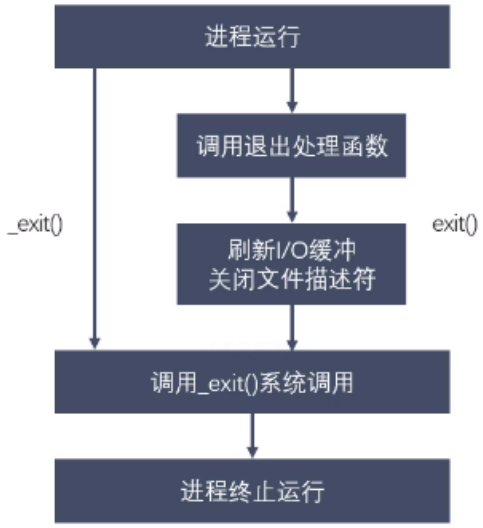
。

七、进程退出、孤儿进程、僵尸进程？

01 / 进程退出

```
#include <stdlib.h>
void exit(int status);

#include <unistd.h>
void _exit(int status);
```



02 / 孤儿进程

- 父进程运行结束，但子进程还在运行（未运行结束），这样的子进程就称为孤儿进程（Orphan Process）。
- 每当出现一个孤儿进程的时候，内核就把孤儿进程的父进程设置为 init ，而 init 进程会循环地 wait() 它的已经退出的子进程。这样，当一个孤儿进程凄凉地结束了其生命周期的时候，init 进程就会代表党和政府出面处理它的一切善后工作。
- 因此孤儿进程并不会有什么危害。

```
i : 4, pid : 26318
nana@nana-virtual-machine:~/linux/linux_duojincheng$ child process, pid : 26319, ppid : 1
num: 10
num*=10 : 100
i : 0, pid : 26319
```

孤儿进程的父进程被设置为 init，init 的 pid 是 1。资源回收。

父进程结束后，会切换到当前终端（父进程的父进程）。因为子进程和父进程共享内核区的文件描述符等一些资源，所以子进程的输出也是在当前终端输出。

03 / 僵尸进程 =

- 每个进程结束之后，都会释放自己地址空间中的用户区数据，内核区的 PCB 没有办法自己释放掉，需要父进程去释放。
- 进程终止时，父进程尚未回收，子进程残留资源（PCB）存放于内核中，变成僵尸（Zombie）进程。
- 僵尸进程不能被 kill -9 杀死。
- 这样就会导致一个问题，如果父进程不调用 wait() 或 waitpid() 的话，那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程，此即为僵尸进程的危害，应当避免。

僵尸进程不可以通过 kill -9 命令杀死；可以通过 ctrl+C 杀死进程（信号）。但是可以用 kill -9 命令通过杀死父进程来杀死子进程。僵尸进程的清理工作需要由其父进程负责。父进程可以通过调用 wait() 或 waitpid() 等函数来获取僵尸进程的退出状态，并释放僵尸进程所占用的资源。

八、进程回收：wait()、waitpid()。

04 / 进程回收

- 在每个进程退出的时候，内核释放该进程所有的资源、包括打开的文件、占用的内存等。但是仍然为其保留一定的信息，这些信息主要主要指进程控制块PCB的信息（包括进程号、退出状态、运行时间等）。
- 父进程可以通过调用wait或waitpid得到它的退出状态同时彻底清除掉这个进程。
- wait() 和 waitpid() 函数的功能一样，区别在于，wait() 函数会阻塞，waitpid() 可以设置不阻塞，waitpid() 还可以指定等待哪个子进程结束。
- 注意：一次wait或waitpid调用只能清理一个子进程，清理多个子进程应使用循环。

04 / 退出信息相关宏函数

- WIFEXITED(status) 非0，进程正常退出
- WEXITSTATUS(status) 如果上宏为真，获取进程退出的状态（exit的参数）

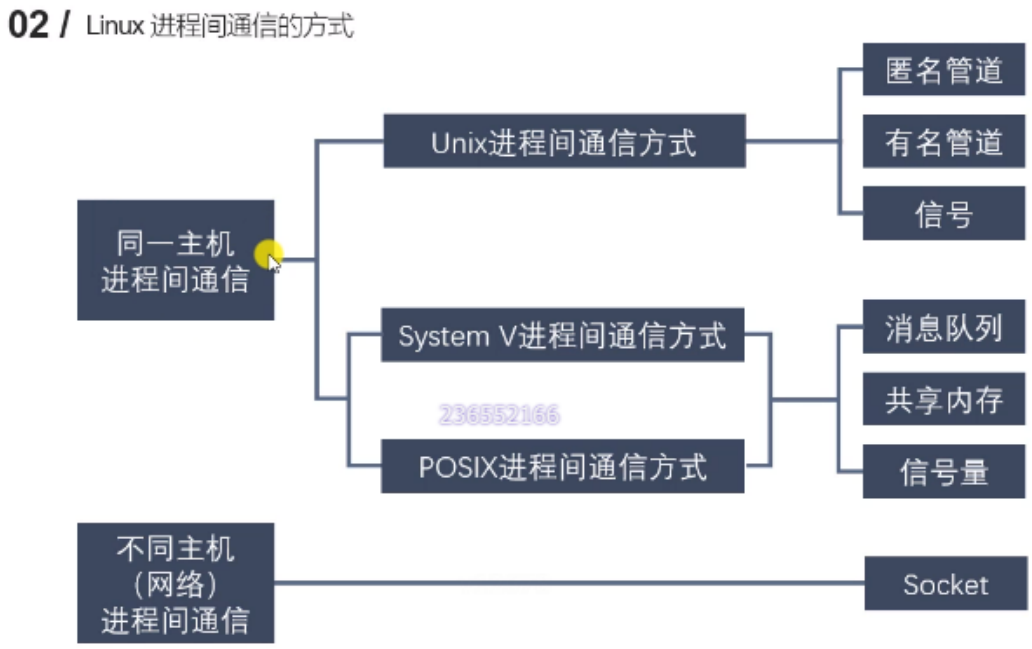
- WIFSIGNALED(status) 非0，进程异常终止
- WTERMSIG(status) 如果上宏为真，获取使进程终止的信号编号

- WIFSTOPPED(status) 非0，进程处于暂停状态
- WSTOPSIG(status) 如果上宏为真，获取使进程暂停的信号的编号
- WIFCONTINUED(status) 非0，进程暂停后已经继续运行

十、进程间通信

01 / 进程间通讯概念

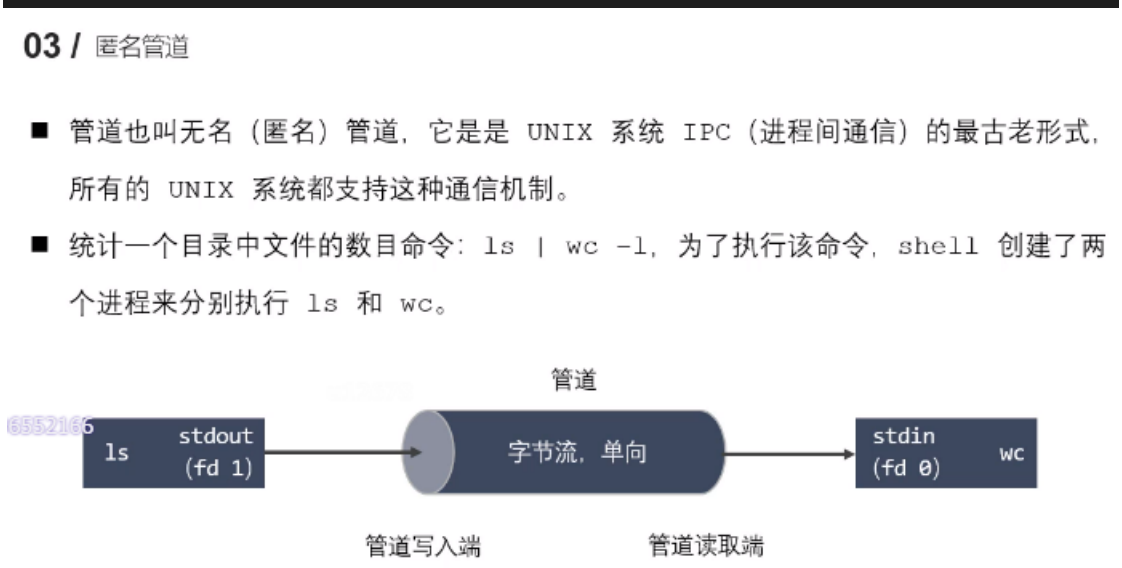
- 进程是一个独立的资源分配单元，不同进程（这里所说的进程通常指的是用户进程）之间的资源是独立的，没有关联，不能在一个进程中直接访问另一个进程的资源。
- 但是，进程不是孤立的，不同的进程需要进行信息的交互和状态的传递等，因此需要进程间通信（ IPC: Inter Processes Communication ）。
- 进程间通信的目的：
  - 数据传输：一个进程需要将它的数据发送给另一个进程。
  - 通知事件：一个进程需要向另一个或一组进程发送消息，通知它（它们）发生了某种事件（如进程终止时要通知父进程）。
  - 资源共享：多个进程之间共享同样的资源。为了做到这一点，需要内核提供互斥和同步机制。
  - 进程控制：有些进程希望完全控制另一个进程的执行（如 Debug 进程），此时控制进程希望能够拦截另一个进程的所有陷入和异常，并能够及时知道它的状态改变。



除了共享内存，内存映射也可以完成进程间通信。

十一、匿名管道

在 fork 函数之前创建管道：对应同一个文件描述符。

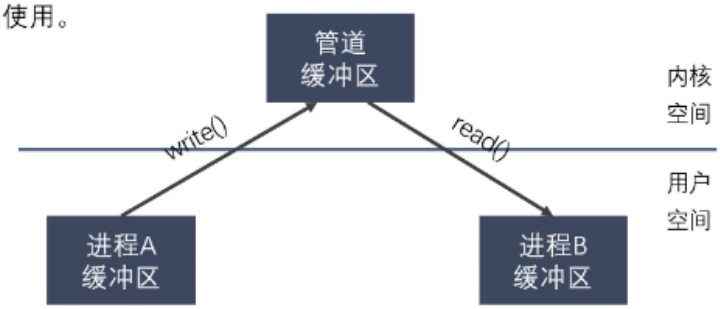


04 / 管道的特点

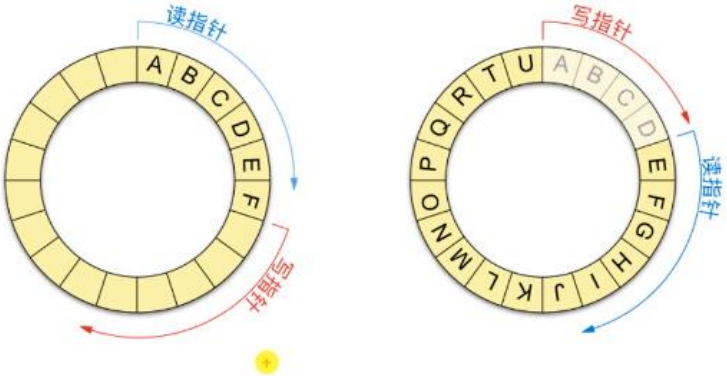
- 管道其实是一个在内核内存中维护的缓冲器，这个缓冲器的存储能力是有限的，不同的操作系统大小不一定相同。
- 管道拥有文件的特质：读操作、写操作。匿名管道没有文件实体，有名管道有文件实体，但不存储数据。可以按照操作文件的方式对管道进行操作。
- 一个管道是一个字节流，使用管道时不存在消息或者消息边界的概念，从管道读取数据的进程可以读取任意大小的数据块，而不管写入进程写入管道的数据块的大小是多少。
- 通过管道传递的数据是顺序的，从管道中读取出来的字节的顺序和它们被写入管道的顺序是完全一样的。

04 / 管道的特点

- 在管道中的数据的传递方向是单向的，一端用于写入，一端用于读取，管道是半双工的。
- 从管道读数据是一次性操作，数据一旦被读走，它就从管道中被抛弃，释放空间以便写更多的数据，在管道中无法使用 `lseek()` 来随机的访问数据。
- 匿名管道只能在具有公共祖先的进程（父进程与子进程，或者两个兄弟进程，具有亲缘关系）之间使用。



06 / 管道的数据结构



管道的数据结构：逻辑上环形的队列。（循环队列）。

十二、父子进程通过匿名管道通信

07 / 匿名管道的使用

- 创建匿名管道

```
#include <unistd.h>
int pipe(int pipefd[2]);
```

- 查看管道缓冲大小命令

```
ulimit -a
```

- 查看管道缓冲大小函数

```
#include <unistd.h>
long fpathconf(int fd, int name);
```

十四、管道的读写特点

管道的读写特点：

使用管道时，需要注意以下几种特殊的情况（假设都是阻塞 I/O 操作）

1.所有的指向管道写端的文件描述符都关闭了（管道写端引用计数为 0），有进程从管道的读端读数据，那么管道中剩余的数据被读取以后，再次 read 会返回 0，就像读到文件末尾一样。

2.如果有指向管道写端的文件描述符没有关闭（管道的写端引用计数大于 0），而持有管道写端的进程也没有往管道中写数据，这个时候有进程从管道中读取数据，那么管道中剩余的数据被读取后，再次 read 会阻塞，直到管道中有数据可以读了才读取数据并返回。

3.如果所有指向管道读端的文件描述符都关闭了（管道的读端引用计数为 0），这个时候有进程向管道中写数据，那么该进程会收到一个信号 SIGPIPE，通常会导致进程异常终止。

4.如果有指向管道读端的文件描述符没有关闭（管道的读端引用计数大于 0），而持有管道读端的进程也没有从管道中读数据，这时有进程向管道中写数据，那么在管道被写满的时候再次 write 会阻塞，直到管道中有空位置才能再次写入数据并返回。

总结：

读管道：

管道中有数据，read 返回实际读到的字节数。

管道中无数据：

写端被全部关闭，read 返回 0（相当于读到文件的末尾）

写端没有完全关闭，read 阻塞等待

写管道：

管道读端全部被关闭，进程异常终止（进程收到 SIGPIPE 信号）

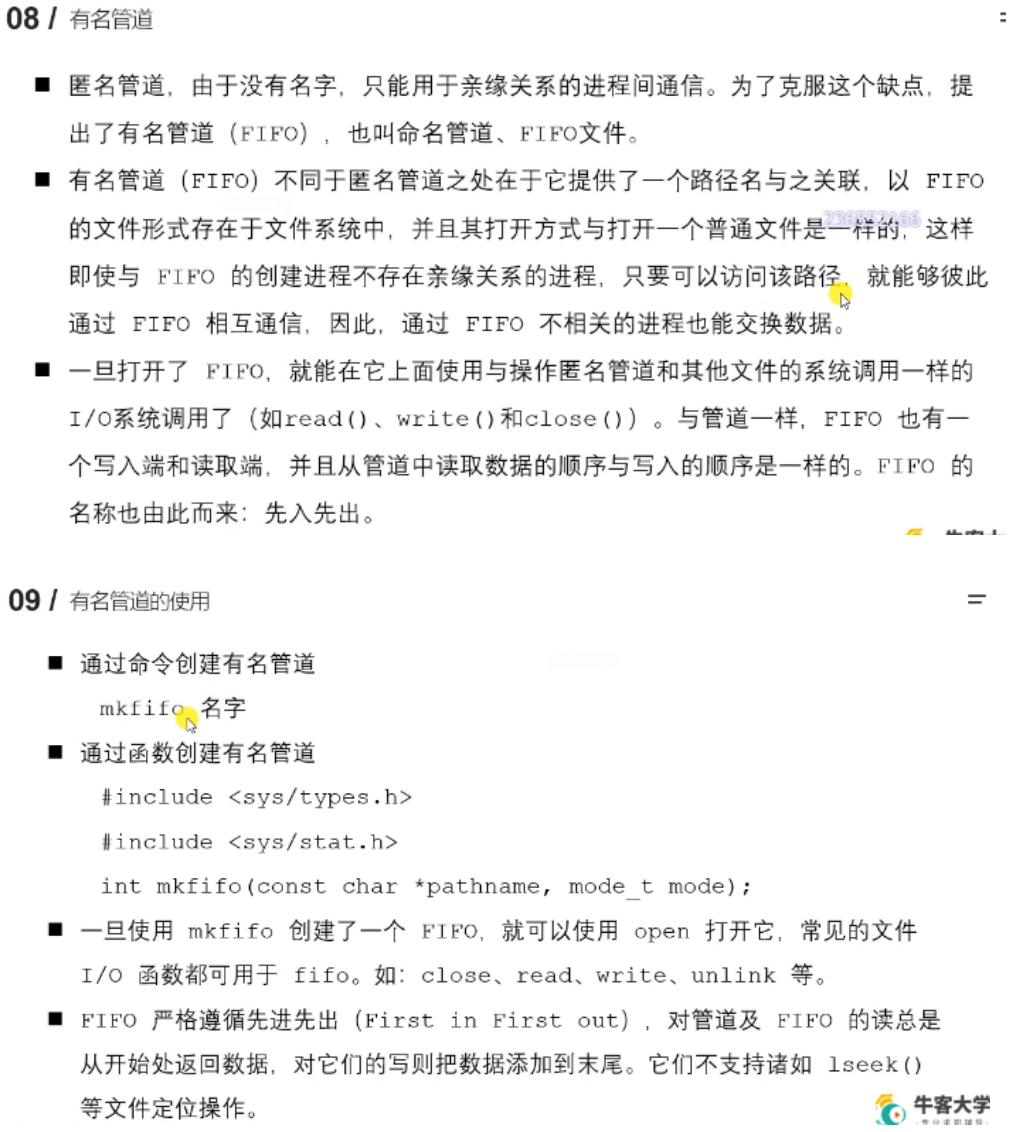
管道读端没有全部关闭：

管道已满，write 阻塞

管道没有满，write 将数据写入，并返回实际写入的字节数。

十五、有名管道

- 有名管道（FIFO）和匿名管道（pipe）有一些特点是相同的，不一样的地方在于：
  1. FIFO 在文件系统中作为一个特殊文件存在，但 FIFO 中的内容却存放在内存中。
  2. 当使用 FIFO 的进程退出后，FIFO 文件将继续保存在文件系统中以便以后使用。
  3. FIFO 有名字，不相关的进程可以通过打开有名管道进行通信。



十七、内存映射：效率较高

内存映射通过使用 mmap()系统调用来创建映射，并将文件或其他设备的内容映射到一块内存区域。（虚拟地址空间的共享库）

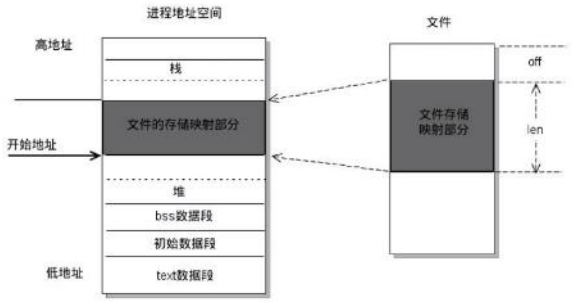
多个进程可以独立的访问同一个文件的映射，但每个进程都有自己的虚拟地址空间和缓存副本。

内存映射适用于频繁访问大文件或需要在多个进程之间共享数据的场景。

内存映射需要进行文件的读写操作，可以持久化的保存数据。

10 / 内存映射

- 内存映射（Memory-mapped I/O）是将磁盘文件的数据映射到内存，用户通过修改内存就能修改磁盘文件。



11 / 内存映射相关系统调用

- `#include <sys/mman.h>`
- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
- `int munmap(void *addr, size_t length);`

十九、信号

01 / 信号的概念

- 信号是 Linux 进程间通信的最古老的方式之一，是事件发生时对进程的通知机制，有时也称之为软件中断，它是在软件层次上对中断机制的一种模拟，是一种异步通信的方式。信号可以导致一个正在运行的进程被另一个正在运行的异步进程中断，转而处理某一个突发事件。
- 发往进程的诸多信号，通常都是源于内核。引发内核为进程产生信号的各类事件如下：
  - 对于前台进程，用户可以通过输入特殊的终端字符来给它发送信号。比如输入Ctrl+C通常会给进程发送一个中断信号。
  - 硬件发生异常，即硬件检测到一个错误条件并通知内核，随即再由内核发送相应信号给相关进程。比如执行一条异常的机器语言指令，诸如被 0 除，或者引用了无法访问的内存区域。
  - 系统状态变化，比如 alarm 定时器到期将引起 SIGALRM 信号，进程执行的 CPU 时间超限，或者该进程的某个子进程退出。
  - 运行 kill 命令或调用 kill 函数。

- 使用信号的两个主要目的是：
  - 让进程知道已经发生了一个特定的事情。
  - 强迫进程执行它自己代码中的信号处理程序。
- 信号的特点：
  - 简单
  - 不能携带大量信息
  - 满足某个特定条件才发送
  - 优先级比较高
- 查看系统定义的信号列表：kill -l
- 前 31 个信号为常规信号，其余为实时信号。

编号	信号名称	对应事件	默认动作
1	SIGHUP	用户退出shell时，由该shell启动的所有进程将收到这个信号	终止进程
2	SIGINT	当用户按下了<Ctrl+C>组合键时，用户终端向正在运行中的由该终端启动的程序发出此信号	终止进程
3	SIGQUIT	用户按下<Ctrl+\>组合键时产生该信号，用户终端向正在运行中的由该终端启动的程序发出些信号	终止进程
4	SIGILL	CPU检测到某进程执行了非法指令	终止进程并产生core文件
5	SIGTRAP	该信号由断点指令或其他 trap指令产生	终止进程并产生core文件
6	SIGABRT	调用abort函数时产生该信号	终止进程并产生core文件
7	SIGBUS	非法访问内存地址，包括内存对齐出错	终止进程并产生core文件
8	SIGFPE	在发生致命的运算错误时发出。不仅包括浮点运算错误，还包括溢出及除数为0等所有的算法错误	终止进程并产生core文件

编号	信号名称	对应事件	默认动作
9	SIGKILL	无条件终止进程。该信号不能被忽略，处理和阻塞	终止进程，可以杀死任何进程
10	SIGUSE1	用户定义的信号。即程序员可以在程序中定义并使用该信号	终止进程
11	SIGSEGV	指示进程进行了无效内存访问(段错误)	终止进程并产生core文件
12	SIGUSR2	另外一个用户自定义信号，程序员可以在程序中定义并使用该信号	终止进程
13	SIGPIPE	Broken pipe向一个没有读端的管道写数据	终止进程
14	SIGALRM	定时器超时，超时的时间 由系统调用alarm设置	终止进程
15	SIGTERM	程序结束信号，与SIGKILL不同的是，该信号可以被阻塞和终止。通常用来要示程序正常退出。执行shell命令Kill时，缺省产生这个信号	终止进程
16	SIGSTKFLT	Linux早期版本出现的信号，现仍保留向后兼容	终止进程

编号	信号名称	对应事件	默认动作
17	SIGCHLD	子进程结束时，父进程会收到这个信号	忽略这个信号
18	SIGCONT	如果进程已停止，则使其继续运行	继续/忽略
19	SIGSTOP	停止进程的执行。信号不能被忽略，处理和阻塞	为终止进程
20	SIGTSTP	停止终端交互进程的运行。按下<ctrl+z>组合键时发出这个信号	暂停进程
21	SIGTTIN	后台进程读终端控制台	暂停进程
22	SIGTTOU	该信号类似于SIGTTIN，在后台进程要向终端输出数据时发生	暂停进程
23	SIGURG	套接字上有紧急数据时，向当前正在运行的进程发出些信号，报告有紧急数据到达。如网络带外数据到达	忽略该信号
24	SIGXCPU	进程执行时间超过了分配给该进程的CPU时间，系统产生该信号并	终止进程

03 / 信号的 5 种默认处理动作

- 查看信号的详细信息：man 7 signal
- 信号的 5 中默认处理动作
  - Term　　终止进程
  - Ign　　当前进程忽略掉这个信号
  - Core　　终止进程，并生成一个Core文件
  - Stop　　暂停当前进程
  - Cont　　继续执行当前被暂停的进程
- 信号的几种状态：产生、未决、递达
- SIGKILL 和 SIGSTOP 信号不能被捕捉、阻塞或者忽略，只能执行默认动作。

二十、kill、raise、abort、alarm、setitimer

04 / 信号相关的函数

- `int kill(pid_t pid, int sig);`
- `int raise(int sig);`
- `void abort(void);`
- `unsigned int alarm(unsigned int seconds);`
- `int setitimer(int which, const struct itimerval *new_val, struct itimerval *old_value);`

二十三、signal 信号捕捉函数

05 / 信号捕捉函数

- `sighandler_t signal(int signum, sighandler_t handler);`
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);`

二十四、信号集

06 / 信号集

- 许多信号相关的系统调用都需要能表示一组不同的信号，多个信号可使用一个称之为信号集的数据结构来表示，其系统数据类型为 sigset_t。
- 在 PCB 中有两个非常重要的信号集。一个称之为“阻塞信号集”，另一个称之为“未决信号集”。这两个信号集都是内核使用位图机制来实现的。但操作系统不允许我们直接对这两个信号集进行位操作。而需自定义另外一个集合，借助信号集操作函数来对 PCB 中的这两个信号集进行修改。
- 信号的“未决”是一种状态，指的是从信号的产生到信号被处理前的这一段时间。
- 信号的“阻塞”是一个开关动作，指的是阻止信号被处理，但不是阻止信号产生。
- 信号的阻塞就是让系统暂时保留信号留待以后发送。由于另外有办法让系统忽略信号，所以一般情况下信号的阻塞只是暂时的，只是为了防止信号打断敏感的操作。

08 / 信号集相关的函数

- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
- `int sigaddset(sigset_t *set, int signum);`
- `int sigdelset(sigset_t *set, int signum);`
- `int sigismember(const sigset_t *set, int signum);`
- `int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);`
- `int sigpending(sigset_t *set);`

09 / 内核实现信号捕捉的过程

二十七、SIGCHLD 信号

- SIGCHLD信号产生的条件
  - 子进程终止时
  - 子进程接收到 SIGSTOP 信号停止时
  - 子进程处在停止态，接受到SIGCONT后唤醒时
- 以上三种条件都会给父进程发送 SIGCHLD 信号，父进程默认会忽略该信号

可用于解决僵尸进程的问题。

二十八、共享内存：需要进程同步

共享内存是一种通过将一块内存区域映射到多个进程的地址空间，使它们可以直接访问共享数据的机制。直接共享一块内存区域。

共享内存通过使用 shmget()、shmat() 等系统调用来创建共享内存区域，并将其附加到进程的地址空间。

多个进程可以同时访问同一块共享内存，它们可以直接读写共享内存

中的数据，而无需进行复制或数据传输。

共享内存适用于需要高性能且频繁交换数据的进程间通信场景。共享内存在断电或重新启动后会丢失数据。

01 / 共享内存

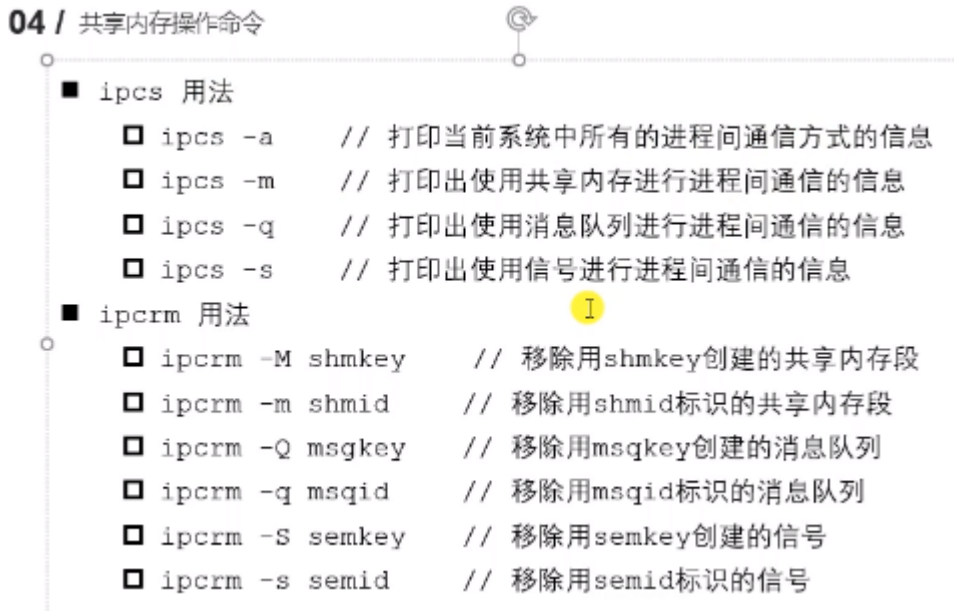
- 共享内存允许两个或者多个进程共享物理内存的同一块区域（通常被称为段）。由于一个共享内存段会称为一个进程用户空间的一部分，因此这种 IPC 机制无需内核介入。所有需要做的就是让一个进程将数据复制进共享内存中，并且这部分数据会对其他所有共享同一个段的进程可用。
- 与管道等要求发送进程将数据从用户空间的缓冲区复制进内核内存和接收进程将数据从内核内存复制进用户空间的缓冲区的做法相比，这种 IPC 技术的速度更快。

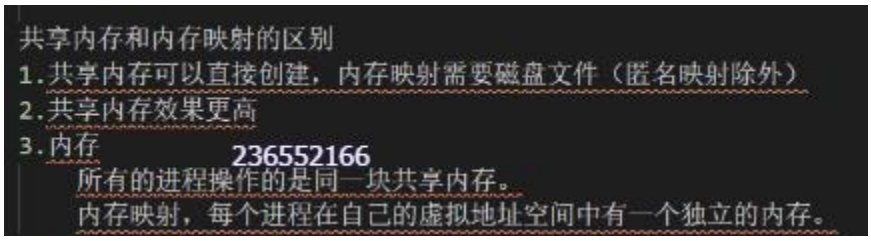
02 / 共享内存使用步骤

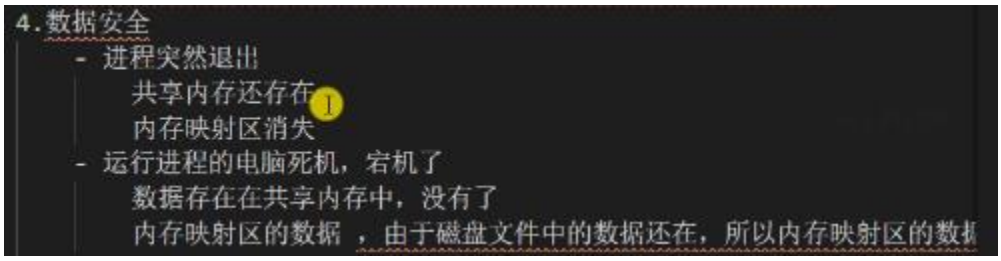
- 调用 shmget() 创建一个新共享内存段或取得一个既有共享内存段的标识符（即由其他进程创建的共享内存段）。这个调用将返回后续调用中需要用到的共享内存标识符。
- 使用 shmat() 来附上共享内存段，即使该段成为调用进程的虚拟内存的一部分。
- 此刻在程序中可以像对待其他可用内存那样对待这个共享内存段。为引用这块共享内存，程序需要使用由 shmat() 调用返回的 addr 值，它是一个指向进程的虚拟地址空间中该共享内存段的起点的指针。
- 调用 shmdt() 来分离共享内存段。在这个调用之后，进程就无法再引用这块共享内存了。这一步是可选的，并且在进程终止时会自动完成这一步。
- 调用 shmctl() 来删除共享内存段。只有当当前所有附加内存段的进程都与之分离之后内存段才会销毁。只有一个进程需要执行这一步。

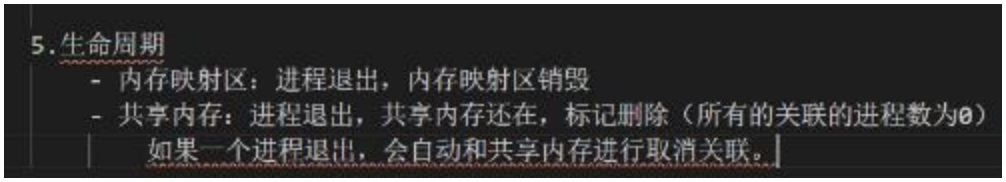
03 / 共享内存操作函数

- `int shmget(key_t key, size_t size, int shmflg);`
- `void *shmat(int shmid, const void *shmaddr, int shmflg);`
- `int shmdt(const void *shmaddr);`
- `int shmctl(int shmid, int cmd, struct shmid_ds *buf);`
- `key_t ftok(const char *pathname, int proj_id);`


04 / 共享内存操作命令
ipcs 用法
ipcs -a // 打印当前系统中所有的进程间通信方式的信息
ipcs -m // 打印出使用共享内存进行进程间通信的信息
ipcs -q // 打印出使用消息队列进行进程间通信的信息
ipcs -s // 打印出使用信号进行进程间通信的信息
ipcrm 用法
ipcrm -M shmkey // 移除用shmkey创建的共享内存段
ipcrm -m shmid // 移除用shmid标识的共享内存段
ipcrm -Q msgkey // 移除用msqkey创建的消息队列
ipcrm -q msqid // 移除用msqid标识的消息队列
ipcrm -S semkey // 移除用semkey创建的信号
ipcrm -s semid // 移除用semid标识的信号


共享内存和内存映射的区别
1.共享内存可以直接创建，内存映射需要磁盘文件（匿名映射除外）
2.共享内存效果更高
3.内存
236552166
所有的进程操作的是同一块共享内存。
内存映射，每个进程在自己的虚拟地址空间中有一个独立的内存。


4.数据安全
- 进程突然退出
共享内存还存在
内存映射区消失
- 运行进程的电脑死机，宕机了
数据存在在共享内存中，没有了
内存映射区的数据 ，由于磁盘文件中的数据还在，所以内存映射区的数


5.生命周期
- 内存映射区：进程退出，内存映射区销毁
- 共享内存：进程退出，共享内存还在，标记删除（所有的关联的进程数为0）
如果一个进程退出，会自动和共享内存进行取消关联。

三十、守护进程

01 / 终端

- 在 UNIX 系统中，用户通过终端登录系统后得到一个 shell 进程，这个终端成为 shell 进程的控制终端（Controlling Terminal），进程中，控制终端是保存在 PCB 中的信息，而 fork() 会复制 PCB 中的信息，因此由 shell 进程启动的其它进程的控制终端也是这个终端。
- 默认情况下（没有重定向），每个进程的标准输入、标准输出和标准错误输出都指向控制终端，进程从标准输入读也就是读用户的键盘输入，进程往标准输出或标准错误输出写也就是输出到显示器上。
- 在控制终端输入一些特殊的控制键可以给前台进程发信号，例如 Ctrl + C 会产生 SIGINT 信号，Ctrl + \ 会产生 SIGQUIT 信号。

echo $$：查看当前进程的 id。 tty ：查看当前进程的号

02 / 进程组

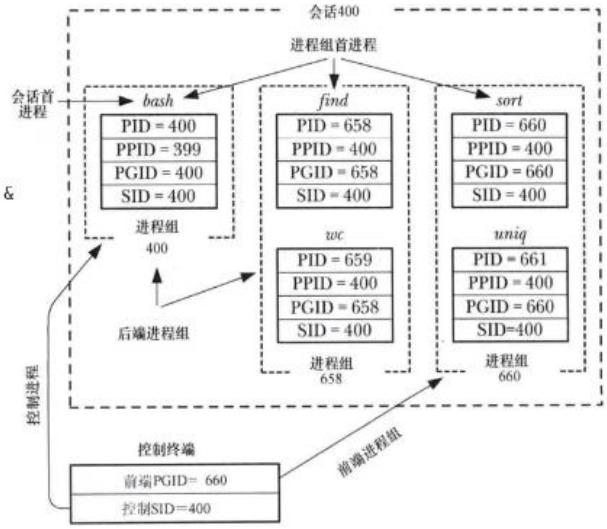
- 进程组和会话在进程之间形成了一种两级层次关系：进程组是一组相关进程的集合，会话是一组相关进程组的集合。进程组合会话是为支持 shell 作业控制而定义的抽象概念，用户通过 shell 能够交互式地在前台或后台运行命令。
- 进行组由一个或多个共享同一进程组标识符（PGID）的进程组成。一个进程组拥有一个进程组首进程，该进程是创建该组的进程，其进程 ID 为该进程组的 ID，新进程会继承其父进程所属的进程组 ID。
- 进程组拥有一个生命周期，其开始时间为首进程创建组的时刻，结束时间为最后一个成员进程退出组的时刻。一个进程可能会因为终止而退出进程组，也可能会因为加入了另外一个进程组而退出进程组。进程组首进程无需是最后一个离开进程组的成员。

03 / 会话

- 会话是一组进程组的集合。会话首进程是创建该新会话的进程，其进程 ID 会成为会话 ID。新进程会继承其父进程的会话 ID。
- 一个会话中的所有进程共享单个控制终端。控制终端会在会话首进程首次打开一个终端设备时被建立。一个终端最多可能会成为一个会话的控制终端。
- 在任一时刻，会话中的其中一个进程组会成为终端的前台进程组，其他进程组会成为后台进程组。只有前台进程组中的进程才能从控制终端中读取输入。当用户在控制终端中输入终端字符生成信号后，该信号会被发送到前台进程组中的所有成员。
- 当控制终端的连接建立起来之后，会话首进程会成为该终端的控制进程。

04 / 进程组、会话、控制终端之间的关系

- `find / 2 > /dev/null | wc -l &`
- `sort < longlist | uniq -c`



05 / 进程组、会话操作函数

- `pid_t getpgrp(void);`
- `pid_t getpgid(pid_t pid);`
- `int setpgid(pid_t pid, pid_t pgid);`
- `pid_t getsid(pid_t pid);`
- `pid_t setsid(void);`

06 / 守护进程

- 守护进程（Daemon Process），也就是通常说的 Daemon 进程（精灵进程），是 Linux 中的后台服务进程。它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。
- 守护进程具备下列特诊：
  - 生命周期很长，守护进程会在系统启动的时候被创建并一直运行直至系统被关闭。
  - 它在后台运行并且不拥有控制终端。没有控制终端确保了内核永远不会为守护进程自动生成任何控制信号以及终端相关的信号（如 SIGINT、SIGQUIT）。
- Linux 的大多数服务器就是用守护进程实现的。比如，Internet 服务器 inetd，Web 服务器 httpd 等。

07 / 守护进程的创建步骤

- 执行一个 fork()，之后父进程退出，子进程继续执行。
- 子进程调用 setsid() 开启一个新会话。
- 清除进程的 umask 以确保当守护进程创建文件和目录时拥有所需的权限。
- 修改进程的当前工作目录，通常会改为根目录（/）。
- 关闭守护进程从其父进程继承而来的所有打开着的文件描述符。
- 在关闭了文件描述符0、1、2之后，守护进程通常会打开/dev/null 并使用dup2()使所有这些描述符指向这个设备。
- 核心业务逻辑

1、 父进程退出是因为不会在运行结束后出现 shell 提示符；子进程会从父进程处继承进程组的 ID。

2、 子进程调用 setsid()不会有控制终端；没有冲突。