操作系统八股文 之 进程管理

原创 C 陈同学在搬砖 2021-12-09 10:24

各位好 我是陈同学
一个放弃BAT offer的深圳老师
最近整理了自己的秋招笔记

我当初拿下BAToffer
就靠这份八股文
基本上涵盖了面试中80%的知识点
只要能记下这份八股文+做几个项目+刷一些题
就可以参加校招了
话不多说 正式开始分享

进程管理

Q1 什么是进程?

- 进程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, 它是进行系统资源分配、调度的一个独立单位。

- (我的理解是硬盘上的程序装载到物理内存时运行的一段状态,每个进程占有相应的占有虚拟地址和页表)

Q2 进程的执行过程是什么样的?

一段代码从编译完成到成为进程，进而执行结束的过程是什么样的呢

大致可以分为3个阶段

step1 从源代码编译到可执行程序

代码编译过程

```
预处理
读取源程序，对其中的伪指令（以#开头的指令）
和特殊符号进行处理。
包括宏定义替换、条件编译指令、
头文件包含指令、特殊符号。
 编译程序所完成的
基本上是对源程序的“替代”工作。
经过此种替代，
生成一个没有宏定义、没有条件编译指令、
没有特殊符号的输出文件。
.i预处理后的c文件，.ii预处理后的C++文件。

编译阶段
编译程序所要作得工作
就是通过词法分析和语法分析，
在确认所有的指令都符合语法规则之后，
将其翻译成
等价的中间代码表示或汇编代码。
.s文件


汇编过程
汇编过程实际上指
把汇编语言代码翻译成目标机器指令的过程。
目标文件中所存放的
也就是与源程序等效的
目标的机器语言代码。
.o目标文件

链接阶段
链接程序的主要工作就是
将有关的目标文件彼此相连接，
也即将在一个文件中引用的符号
同该符号在另外一个文件中的定义连接起来，
使得所有的这些目标文件
成为一个能够
操作系统装入执行的统一整体。
```

step2 :程序从硬盘加载到内存

通过 虚拟内存 技术 ， 采用分页或分段的形式将程序从硬盘装载进内存，补充说明在下面的Q3

PS；关于 虚拟内存 技术，会在下面几期的内存管理中写

step3:程序从内存调度到cpu执行

cpu的组成

- CPU 的内部由寄存器、控制器、运算器和时钟四部分组成， 各部分之间通过电信号连通
- 寄存器 是中央处理器内的组成部分。它们可以用来暂存指令、数据和地址。可以将其看作是内存的一种。根据种类的不同，一个 CPU 内部会有 20 - 100个寄存器。
- 不同类型的 CPU ，其内部寄存器的种类，数量以及寄存器存储的数值范围都是不同的。不过，根据功能的不同，可以将寄存器划分为上面这几类
- 控制器 负责把内存上的指令、数据读入寄存器，并根据指令的结果控制计算机
- 运算器 负责运算从内存中读入寄存器的数据
- 时钟 负责发出 CPU 开始计时的时钟信号

cpu的执行过程

- cpu开始调度到某个进程执行 程序装在进内存
- 在代码段存的都是机器码 操作系统告诉cpu第一条指令的地址
- 然后后面开始执行一系列指令 这些指令当中无非有这么几类：

1. 把数据从内存加载我的寄存器里
2. 对寄存器的数据进行运算， 例如把两个寄存器的数加起来
3. 把我寄存器的数据再写到内存里 这些指令可以通过汇编语言解读出来

具体可以分为以下5个阶段： 取指令、指令译码、执行指令、访存取数、结果写回 。

- 取指令阶段是将内存中的指令读取到 CPU 中寄存器的过程，程序寄存器用于存储下一条指令所在的地址
- 指令译码阶段，在取指令完成后，立马进入指令译码阶段，在指令译码阶段，指令译码器按照预定的指令格式，对取回的指令进行拆分和解释，识别区分出不同的指令类别以及各种获取操作数的方法。
- 执行指令阶段，译码完成后，就需要执行这一条指令了，此阶段的任务是完成指令所规定的各种操作，具体实现指令的功能。
- 访问取数阶段，根据指令的需要，有可能需要从内存中提取数据，此阶段的任务是：根据指令地址码，得到操作数在主存中的地址，并从主存中读取该操作数用于运算。
- 结果写回阶段，作为最后一个阶段，结果写回（Write Back，WB）阶段把执行指令阶段的运行结果数据“写回”到某种存储形式：结果数据经常被写到CPU的内部寄存器中，以便被后续的指令快速地存取；

Q3：进程装载到内存中的过程是什么样的?

- 在每个进程创建加载时，内核只是为进程“创建”了虚拟内存的布局 [这个在后面会专门写一期 虚拟内存 来讲]
- 即每个进程都有自己独立的4G(32位系统下)内存空间，每个进程的4G内存空间只是虚拟内存空间，

- 实际上并不立即就把虚拟内存对应位置的程序数据和代码（比如.text .data段）拷贝到物理内存中，
- 只是通过分页式或者分段式里面的页表或者段表建立好虚拟内存和磁盘文件之间的映射 是通过mmap来建立映射的
- 当进程访问某个虚拟地址时，去看页表，如果发现对应的数据不在物理内存中，则缺页异常
- 缺页异常的处理过程，就是把进程需要的数据从磁盘上拷贝到物理内存中，如果内存已经满了，没有空地方了，就根据页面置换算法那就找一个页覆盖，当然如果被覆盖的页曾经被修改过，需要将此页写回磁盘
- 将数据拷贝到内存以后 每次访问内存空间的某个地址，每次都会把地址翻译为实际物理内存地址
- 所有进程共享同一物理内存，每个进程只把自己目前需要的虚拟内存空间映射并存储到物理内存上。
- 虚拟内存还有请求调入功能和置换功能，
- 请求调入功能使得一个大程序可以分页载入内存 防止大作业要求的内存空间超过了内存容量不能全部被装入致使该作业无法运行。
- 置换功能可以将内存中暂时没用到的页置换下来 防止有大量作业要求运行，但只能将少数作业装入内存让它们先运行，而将其它大量的作业留在外存上等待。
- 在linux下面有个交换分区swap就是用来置换用的 我们在安装系统的时候已经建立了
- swap 分区。swap 分区通常被称为交换分区，这是一块特殊的硬盘空间，即当实际内存不够用的时候，操作系统会从内存中取出一部分暂时不用的数据，放在交换分区中，从而为当前运行的程序腾出足够的内存空间。
- 也就是说，当内存不够用时，我们使用 swap 分区来临时顶替。这种“拆东墙，补西墙”的方式应用于几乎所有的操作系统中。

Q4：进程在内存中的结构特征是什么样的

- 在内核给每个进程创建的4G虚拟空间中，总共可以分为以下几个部分

重点聊聊下面几个部分 分别是 `task_struct，正文段(text)，用户数据段(gvar bss 堆 栈)`

task_struct

1.概念

task_struct 称为PCB(Process Control Block) 即进程控制块。是为了描述和控制进程运行而为每个进程定义的一个数据结构，它是进程存在的唯一标志

可以用链接方式或者索引方式对PCB进行组织

链接方式就是把PCB组织成各种队列就绪队列 阻塞队列等等

索引方式就是用索引表的方式对PCB进行组织 就绪索引表 阻塞索引表等等 在 Linux 中,PCB 是一个名为 task_struct 的结构体,称为任务结构体。记录了比如进程的状态和标志 进程的标识。进程的族亲关系等等信息

2.组织形式

task_struct任务结构体的管理

```
1) task[]数组
内核空间设置了一个指针数组 task[],
该数组的每一个元素指向一个任务结构体,
所以 task 数组又称 task 向量。
Task 数组的大小决定了系统中容纳进程最大数量。
它的默认值被定义为 512:
? task[]数组的第一个指针指向一个名字为 init_task 的结构体,
它是系统初始化进程 init 的任务结构体。
```

```
2)nr_tasks
为了记录系统中实际存在的进程数,系统定义了一个全局变量 nr_tasks,
其值随系统中存在的进程数目而变化。
```

```
3)双向循环链表
Linux 中把所有进程的任务结构相互连成一个双向循环链表,
其首结点就是 init 的任务结构体 init_task。
这个双向链表是通过任务结构体中的两个成员项指针相互链接而成:
Struct task_struct *next_task;/*指向后一个任务结构体的指针*/
Struct task_struct *prev_task;/*指向前一个任务结构体的指针*/
```

正文段(text)

- 正文段:存放进程要执行的指令代码。Linux 中正文段具有只读属性。

用户数据段(gvar bss 堆 栈)

- 用户数据段:进程运行过程中处理数据的集合,它们是进程直接进行操作的所有数据,
- 包括 进程运行处理的数据段和进程使用的堆栈。

Q5: 进程的用户态和内核态是指什么?

用户态内核态的概念

- 进程由4g的内存空间，但并不是任何时候都可以访问这4G的内存空间。
- 进程在 `用户态` 只能访问0~3G，只有 `进入内核态` 才能访问3G~4G,
- 进程通过系统调用进入内核态,每个进程虚拟内存空间的3G~4G部分是相同的

- 进程在运行过程中 cpu通过页表找到逻辑地址对应的物理内存地址,执行对应指令,
- 凡是涉及到IO读写、内存分配等硬件资源的操作时， 往往不能直接操作，
- 而是通过得到内核态内存中使用系统调用，然后内核态的CPU执行有关硬件资源操作指令，
- 得到相关的硬件资源后在返回到用户态继续执行 也就是说，进程即可以在用户空间运行，又可以在内核空间中运行。
- 进程在用户空间运行是，被称为进程的用户态， 而陷入内核空间的时候，被称为进程的内核态。

用户态内核态的意义

- 第一点：分开来存放，就让系统的数据和用户的数据互不干扰，保证系统的稳定性，并且管理上很方便；
- 第二点：也是重要的一点，将用户的数据和系统的数据隔离开，就可以对两部分的数据的访问进行控制。这样就可以确保用户程序不能随便操作系统的数据，这样防止用户程序误操作或者是恶意破坏系统。

从用户态到内核态的触发手段

- 1.系统调用：用户进程通过系统调用申请使用内核提供的服务程序来完成工作，调用系统调用后会向CPU发送中断信号。这时CPU会暂停执行下一条指令（用户态）转而执行与该中断信号对应的中断处理程序（内核态）
- 2.异常：当CPU在执行运行在用户态下的程序时，发生了某些事先不可知的异常， 这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常。

上面这两种操作会使cpu发生上下文切换

Q6.进程有几种状态？它们之间是如何转换的?

在Linux系统下进程总共有下面几种状态

- TASK_RUNNING，用字母R缩写，表示可执行状态。
- TASK_INTERRUPTIBLE， 用S缩写，表示可中断的睡眠状态。
- TASK_UNINTERRUPTIBLE，用D缩写，表示不可中断的睡眠状态。
- TASK_STOPPED or TASK_TRACED， 用T缩写，暂停状态或跟踪状态。
- TASK_ZOMBIE 用Z缩写，表示退出状态，进程成为僵尸进程。
- TASK_DEAD 用X缩写 进程在退出的过程中、
- EXIT_DEAD 退出状态，进程即将被完全销毁 他们之间的一个转换过程是这样的

Q7.fork的使用过程和原理?

1.fork的介绍

- 1系统中同时运行着很多进程,这些进程都是从最初只有一个进程开始一个一个 复制出来的。

- 2进程是通过fork系列的系统调用（fork、clone、vfork）来创建的，内核（或内核模块）也可以通过kernel_thread函数创建内核进程。这些创建子进程的函数本质上都完成了相同的功能——将调用进程复制一份，得到子进程。（可以通过选项参数来决定各种资源是共享、还是私有。）

- 3.fork的作用是根据一个现有的进程复制出一个新进程,原来的进程称为父进程(Parent Process),新进程称为子进程(ChildProcess)。一个进程调用fork（）函数后，系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

2. fork的用法

```
#include <unistd.h>
#include <stdio.h>
void main()
{

  pid_t pid=fork();
  int count =0;

  if(pid==0)//说明当前是子进程
  {
        count-=1;
 printf("我是子进程,cout的值为%d,我的进程id是%d,我的父进程id是%d/n",count,getpid(),getppid());
  }
 else if(pid<0)
 {
 printf("fork出错了");
  }
  else//说明当前是父进程
   {
 count+=1;
      printf("我是父进程,cout的值为%d,我的进程id是%d,我的父进程id是%d/n",count,getpid(),getppid());

    }

}
```

- 终结子进程(注意是终结不是销毁)的两种方法

- fork出错可能有两种原因：

1）当前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。

创建新进程成功后，系统中出现两个基本完全相同的进程，这两个进程执行没有固定的先后顺序，哪个进程先执行要看系统的进程调度策略。

3. fork的返回值

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

子进程复制父进程的0到3g空间和父进程内核中的PCB,但id号不同。fork调用一次返回两次

- 父进程中返回子进程ID
- 子进程中返回0
- 读时共享,写时复制

4.fork的原理

- 原始版的fork:

父进程调用fork()后 在虚拟地址空间 子进程会复制父进程的代码段 数据段

内核中的pcb 文件描述附表也会复制

但是其中的进程id不同 在物理存储空间 子进程会复制父进程的数据段 不会复制正文段

子进程虚拟地址空间的正文段指向父进程物理地址地址空间的正文段 这样的问题是一般子进程会执行exec调用别的程序

这样的前面的物理地址空间的复制过程就白费力气了 还是会被exec的程序覆盖掉

所以引入了写时复制

- 升级版的fork(写时复制):

父进程调用fork()后

在虚拟地址空间 子进程会复制父进程的代码段 数据段 内核中的pcb文件描述度表也会复制但是其中的进程id不同

在物理存储空间 子进程和父进程会先共享父进程的物理空间

等到父子进程中出现对物理空间的修改时,再为子进程相应的段分配物理空间。

比如前面的子进程执行exec就是要对物理空间进行修改

这个时候再去为子进程创建物理空间 注意一个问题 fork以后严格意义上说父子进程调度顺序是不确定的

但是一般是先执行子进程 因为假设父进程对物理空间做了变动

然后按照上面所说要为子进程分配物理空间 然后子进程执行exec

还是要覆盖掉刚刚赋值的内容 这样的话复制的过程还是白费力气

- vfork

在写时复制的基础上 在进行改进

子进程连虚拟地址空间也不复制了 也共享父进程的

5.fork()和vfork()的区别

总结有以下三点区别：

- 1．fork （）：子进程拷贝父进程的数据段，代码段 vfork （）：子进程与父进程共享数据段
- 2．fork （）父子进程的执行次序不确定 vfork 保证子进程先运行，在调用exec 或exit 之前与父进程数据是共享的,在它调用exec 或exit 之后父进程才可能被调度运行。
- 3．vfork （）保证子进程先运行，在她调用exec 或exit 之后父进程才可能被调度运行。如果在调用这两个函数之前子进程依赖于父进程的进一步动作，则会导致死锁。

Q8 进程的TASK_RUNNING状态介绍一下?

介绍

- 运行态的进程可以分为3种情况：内核运行态、用户运行态、就绪态。
- 只有在该状态的进程才可能在CPU上运行。而同一时刻可能有多个进程处于可执行状态，这些进程的task_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。
- 进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。
- 很多操作系统教科书将正在CPU上执行的进程定义为RUNNING状态、而将可执行但是尚未被调度执行的进程定义为READY状态，这两种状态在linux下统一为 TASK_RUNNING状态。
- Linux 中把所有处于运行、就绪状态的进程链接成一个双向链表,称为可运行队列(run_queue)。
- 使用任务结构体中的两个指针:
- Struct task_struct *next_run;/*指向后一个任务结构体的指针*/
- Struct task_struct *prev_run;/*指向前一个任务结构体的指针*/
- 该链表的首结点为 init_task。系统设置全局变量 nr_running 记录处于运行、就绪态的进程数。

工作过程

- 1.运行态的进程可以分为3种情况：内核运行态、用户运行态、就绪态。

- 2.Linux 中一个CPU上把所有处于运行态的进程链接成一个双向链表,称为可运行队列(run_queue)。

该链表的首结点为 init_task。系统设置全局变量 nr_running 记录处于运行、就绪态的进程数。

- 3.进程调度器的任务就是从各个CPU的可执行队列中分别选择一个进程在该CPU上运行。具体的选择方式看下方的调度算法

作用

用fork创建子进程后执行的是和父进程相同的程序(但有可能执行不同的代码分支),

子进程往往要调用一种exec函数以执行另一个程序。当进程调用一种exec函数时,该进程的 用户空间代码和数据完全被新程序替换,从新程序的启动例程开始执行。

调用exec并不创建新进程,所以调用exec前后该进程的id并未改变。

执行程序

exec函数族

```
字母p:
不 带 字 母 p(表 示path)的第一个参数
必须是程序的相对路径或绝对路径,
例如“/bin/ls”或“./a.out”,
而不能是“ls”或“a.out”。

对于带字母p的函数:
如果参数中包含/,则将其视为路径名。
否则视为不带路径的程序名,
在PATH环境变量的目录列表中搜索这个程序。


字母l:
带有字母l(表示list)的要求
将新程序的每个命令行参数
都当作一个参数传给它,
命令行参数的个数是可变的,
因此函数原型中有...,...中的
最后一个可变参数应该是NULL


字母v:
对于带有字母v(表示vector)的函数,
则应该先构造一个指向各参数的指针数组,
然后将该数组的首地址当作参数传给它,
数组中的最后一个指针也应该是NULL,
就像main函数的argv参数或者环境变量表一样。


字母e:
对于以e(表示environment)结尾的exec函数,
可以把一份新的环境变量表传给它,
其他exec函数仍使用当前的环境变量表执行新程序。
```

Q9 进程的TASK_INTERRUPTIBLE状态介绍一下?

可中断的睡眠状态。

- 当进程调用一个阻塞的系统函数时,该进程被 置于睡眠(Sleep)状态,这时内核调度其它进程运行,
- 直到该进程等待的事件发生了(比 如网络上接收到数据包,或者调用sleep指定的睡眠时间到了)它才有可能继续运行。
- 处于这个状态的进程因为等待某某事件的发生（比如等待socket连接、等待信号量），而被挂起。
- 这些进程的task_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。
- 通过ps命令我们会看到，一般情况下，进程列表中的绝大多数进程都处于TASK_INTERRUPTIBLE状态（除非机器的负载很高）。
- 毕竟CPU就这么一两个，进程动辄几十上百个，如果不是绝大多数进程都在睡眠，CPU又怎么响应得过来。

Q10 进程的TASK_UNINTERRUPTIBLE状态介绍一下?

- 与TASK_INTERRUPTIBLE状态类似，进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号。

Q11 进程的TASK_STOPPED or TASK_TRACED状态介绍一下?

- 向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK_STOPPED状态（除非该进程本身处于TASK_UNINTERRUPTIBLE状态而不响应信号）。
- （SIGSTOP与SIGKILL信号一样，是非常强制的。不允许用户进程通过signal系列的系统调用重新设置对应的信号处理函数。）
- 向进程发送一个SIGCONT信号，可以让其从TASK_STOPPED状态恢复到TASK_RUNNING状态。
- 当进程正在被跟踪时，它处于TASK_TRACED这个特殊的状态。“正在被跟踪”指的是进程暂停下来，等待跟踪它的进程对它进行操作。
- 比如在gdb中对被跟踪的进程下一个断点，进程在断点处停下来的时候就处于TASK_TRACED状态。而在其他时候，被跟踪的进程还是处于前面提到的那些状态。
- 对于进程本身来说，TASK_STOPPED和TASK_TRACED状态很类似，都是表示进程暂停下来。
- 而TASK_TRACED状态相当于在TASK_STOPPED之上多了一层保护，处于TASK_TRACED状态的进程不能响应SIGCONT信号而被唤醒。
- 只能等到调试进程通过ptrace系统调用执行PTRACE_CONT、PTRACE_DETACH等操作（通过ptrace系统调用的参

数指定操作），或调试进程退出，被调试的进程才能恢复TASK_RUNNING状态。

Q12 进程的TASK_ZOMBIE状态介绍一下?

介绍

- 1. 一个进程在执行系统调用exit函数结束自己的生命的时候，其实它并没有真正的被销毁， 而是留下一个称为僵尸进程（Zombie）的数据结构（系统调用exit，它的作用是 使进程退出，但也仅仅限于将一个正常的进程变成一个僵尸进程，并不能将其完全销毁）
- 2.在这个退出过程中，进程占有的所有资源将被回收，除了task_struct结构（以及少数资源）以外。于是进程就只剩下task_struct这么个空壳，故称为僵尸。
- 3.之所以保留task_struct，是因为task_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。比如在shell中，$?变量就保存了最后一个退出的前台进程的退出码，而这个退出码往往被作为if语句的判断条件。
- 4.当然，内核也可以将这些信息保存在别的地方，而将task_struct结构释放掉，以节省一些空间。但是使用task_struct结构更为方便，因为在内核中已经建立了从pid到task_struct查找关系，还有进程间的父子关系。释放掉task_struct，则需要建立一些新的数据结构，以便让父进程找到它的子进程的退出信息。

僵尸进程的产生原因

- 解释1:

总之就是,在子进程中调用exit/return 可以终结子进程,但是这种终结不是销毁 ,故子进程此时变成僵尸态,

如果父进程一直没有去主动获取子进程的结束状态值,那么子进程就一直保持僵尸状态. 通过wait/waitpid函数就可以获取退出状态值从而可以回收僵尸进程

- 解释2:

一个进程在终止时会关闭所有文件描述符,释放在用户空间分配的内存,但它的PCB还 保留着,内核在其中保存了一些信息:如果是正常终止则保存着退出状态,如果是异常终止 则保存着导致该进程终止的信号是哪个。

这个进程的父进程可以调用wait或waitpid获取这 些信息,然后彻底清除掉这个进程。

我们知道一个进程的退出状态可以在Shell中用特殊变 量$?查看,因为Shell是它的父进程,当它终止时Shell调用wait或waitpid得到它的退出状 态同时彻底清除掉这个进程。

如果一个进程已经终止,但是它的父进程尚未调用wait或waitpid对它进行清理,这时 的进程状态称为僵尸(Zombie)进程。

任何进程在刚终止时都是僵尸进程,正常情况下,僵 尸进程都立刻被父进程清理了,为了观察到僵尸进程,我们自己写一个不正常的程序,

父进 程fork出子进程,子进程终止,而父进程既不终止也不调用wait清理子进程:

僵尸进程的危害

- 假设这样一个情景，我们有一个父进程在不断的创建子进程，每个子进程的存活时间都很短，父进程对子进程的终止状态都不管不顾，

- 任由发展下去，子子孙孙，系统中就会存在许多的僵尸进程，更重要的是每一个僵尸进程都还没占着对应的进程列表，进程列表可是临界资源是有限的，时间一长内存中就没有多余的地方再让我们创建进程了。

僵尸进程的销毁方法

- 1.父进程可以通过wait系列的系统调用（如wait、waitpid）来等待某个或某些子进程的退出，并获取它的退出信息。然后wait系列的系统调用会顺便将子进程的尸体（task_struct）也释放掉。父进程通过wait和waitpid等函数等待子进程结束，这会导致父进程挂起，相关用法可以通过man命令查看。

- 2. 子进程在退出的过程中，内核会给其父进程发送一个信号，通知父进程来“收尸”。这个信号默认是SIGCHLD，但是在通过clone系统调用创建子进程时，可以设置这个信号。可以用signal函数为SIGCHLD安装handler灰调函数，因为子进程结束后， 父进程会收到该信号，可以在handler中调用wait回收。

- 3. 如果父进程不关心子进程什么时候结束，那么可以用signal（SIGCHLD,SIG_IGN） 通知内核，自己对子进程的结束不感兴趣，那么子进程结束后，内核会回收， 并不再给父进程发送信号。

- 4. 还有一些技巧，就是fork两次，父进程fork一个子进程，然后继续工作，子进程fork一 个孙进程后退出，那么孙进程被init接管，孙进程结束后，init会回收。不过子进程的回收 还要自己做。

僵尸进程和孤儿进程的区别

僵尸进程和孤儿进程有什么区别，如何处理

僵尸进程: 子进程退出,父进程没有回收子进程资源(PCB),则子进程变成僵尸进程

孤儿进程: 父进程先于子进程结束,则子进程成为孤儿进程,子进程的父进程成为1号进程init进程, 称为init进程领养孤儿进程

- 区别:

僵尸进程占用一个进程ID号，占用资源，危害系统。

孤儿进程与僵尸进程不同的是，由于父进程已经死亡，子系统会帮助父进程回收处理孤儿进程。

所以孤儿进程实际上是不占用资源的，因为它最终是被系统回收了，不会像僵尸进程那样占用ID

wait和waitpid的使用方法

```
#include <sys/types.h>
 #include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

```
函数若成功，返回进程ID，若出错则返回-1；  返回0,说明没有要回收的进程
pid:
< -1 回收指定进程组内的任意子进程
-1 回收任意子进程
0 回收和当前调用waitpid一个组的所有子进程
> 0 回收指定ID的子进程



status:
传入传出参数 获取子进程的退出状态
options
options参数中
指定WNOHANG可以使父进程不阻塞而立即返回0。
```

- 若 调 用成 功则 返回 清理掉的子进程id,若调用出 错 则 返 回-1。

1.父 进 程 调 用 wait 或waitpid时可能会:

- 阻塞(如果它的所有子进程都还在运行)。
- 带子进程的终止信息立即返回(如果一个子进程已终止,正等待父进程读取其终止信 息)。
- 出错立即返回(如果它没有任何子进程)。这两个函数的区别是:
- 如果父进程的所有子进程都还在运行,调用wait将使父进程阻塞,而调用waitpid时如 果在options参数中指定WNOHANG可以使父进程不阻塞而立即返回0。
- wait等待第一个终止的子进程,而waitpid可以通过pid参数指定等待哪一个子进程。

2.可见,调用wait和waitpid不仅可以获得子进程的终止信息,还可以使父进程阻塞等待子进 程终止,起到进程间同步的作用。如果参数status不是空指针,则子进程的终止信息通过 这个参数传出,如果只是为了同步而不关心子进程的终止信息,可以将status参数指定为 NULL。

Q13讲一讲进程的调度?

1.进程调度算法

- （1）先来先服务调度算法（FCFS）。

顾名思义。就是先来的先进入内存或占用处理机。对于作业调度，就是从后备作业队列中选择一个或多个最先进入队列的作业，将其调入内存。对于进程调度就是从就绪队列选择最新进入的进程，为之分配处理机。

- （2）短作业（进程）优先调度算法（SJ(P)F）

顾名思义。就是在选择作业或进程的时候，先估算每个作业、进程的服务时间，选择其中最短的优先获得处理机。

- （3）高优先权优先调度算法。

这种算法给进程加了一个属性，那就是优先权。这个算法的本质就是，高优先权的优先调用。优先权有两种类型，一种是静态的，即每个进程、作业的优先权在它创建的时候就已经确定，此后都不能改变。另一种是动态的，即进程、作业的优先权是可以改变的。最常见的做法就是进程、作业在等待中，优先权以一定速率随时间增长，这样等待时间越长，被调用的可能性就越大。

- （4）基于时间片的轮转调度法。

这就是分时系统中采用的调度算法。原理就是把所有的就绪队列进程按先来先服务的原则排成队列。每次都把CPU分配给队首，让其执行一个时间片，执行完毕，调度器中断进程，并把该进程移至就绪队列的队尾，然后再取一个队首进程，继续执行下一个时间片。时间片是什么，就是一段很短的CPU时间，几毫秒到几百毫秒不等。

- （5）多级反馈队列调度算法。

这是当下公认的比较好的，使用最广泛的调度算法。其原理也不难。例如，某计算机采用多级反馈队列调度算法，设置了5个就绪队列。第一个就绪队列优先级最高，时间片为2ms。第二个就绪队列优先级第二，时间片为4ms，其余队列也一样，优先级依次递减，时间片依次增加。如果某个进程进入就绪队列，首先把它放在第一个就绪队列的末尾，轮到它执行就执行2ms，时间片结束，若该进程还没有执行完毕，就把该进程移入第二个就绪队列的末尾。只有当第一个队列的进程都执行完时间片，才会执行第二个队列。如此依次执行，若该进程服务时间很长，将被移到最后一个就绪队列。在最后一个就绪队列，进程将按照时间片轮转调度法执行。处理机执行过程中，只有当优先级高的队列中的线程都执行完毕，才会执行优先级低的队列。

2.进程调度死锁

3.进程的上下文切换

- cpu上下文

```
电脑执行程序的过程就是cpu不断执行指令的过程。
cpu执行指令的过程，
第一步就是取指令，并将其放入指令寄存器，
然后对指令译码，
进行一些操作，最后计算下条指令的地址，
并送入程序计数器。
cpu根据程序计数器里的地址取指令，
将取到的指令送指令寄存器。
总之，一个用来存当前指令，
一个用来存下条指令的地址。

在每个任务运行前，
需要系统事先帮他
设置好CPU寄存器和程序计数器
他们都是CPU在运行任何任务前，
必须依赖的环境，因此也被叫做CPU上下文。
```

- cpu上下文转换

```
CPU上下文切换，
就是先把前一个任务的CPU上下文
(也就是CPU寄存器和程序计数器)保存起来，
然后加载新任务的上下文，
到这些寄存器和程序计数器，
最后再跳转到程序计数器所指的新位置，
运行新任务。
而保存下来的上下文，会存储在系统内核中，
并在任务重新调度执行的时候再加载进来。
这样就能保证任务原来的状态不受影响，
让任务看起来还是连续运行。

会带来cpu上下文切换的几种场景主要有以下几种
```

- 进程用户态内核态转换发生的上下文转换

- 进程间的上下文转换
- 线程上下文切换
- 中断上下文切换

Q14 介绍一下进程之间的关系?

从宏观角度来讲，进程间的关系可以分为 `父子进程、进程组、会话`

从微观角度来讲，进程间的关系可以分为 `互斥、同步`

一、宏观角度

父子进程

见问题 `Q7.fork的使用过程和原理?`

进程组

会话

`概念`

linux系统有7个终端tty,当我们登录其中任意一个终端时就形成了一次会话,

然后在该终端中往后运行的进程基于相当于是会话中的进程之一,

每个会话都有一个会话首领（leader）叫控制进程，即创建会话的进程。

会话id(SID)就是控制进程的id,这个控制进程与控制终端相连接,一个会话中的几个进程组可被分为一个前台进程组以及一个或多个后台进程组。

所以一个会话中，应该包括控制进程（会话首进程），一个前台进程组和任意后台进程组。

一个会话只能有一个控制终端，而一个控制终端只能控制一个会话。

用户通过控制终端，可以向该控制终端所控制的会话中的进程发送信号,

比如关闭该终端的时候该会话中的进程也就关闭了。会话可能有也可能没有控制终端

`例子`

建立新会话时,先调用fork, 父进程终止,子进程调用setsid 通过这个函数更改当前进程的会话id

通过这个函数查看进程的会话id

守护进程

- 概念
- 实现
- 例子

```
代码模型
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
void daemonize(void)
{
pid_t pid;
/*
* 成为一个新会话的首进程,失去控制终端
*/
if ((pid = fork()) < 0) {
perror("fork");
exit(1);
} else if (pid != 0) /* parent */
exit(0);
setsid();
/*
* 改变当前工作目录到/目录下.
*/
if (chdir("/") < 0) {
perror("chdir");
exit(1);
}
/* 设置umask为0 */
umask(0);
/*
* 重定向0,1,2文件描述符到 /dev/null,因为已经失去控制终端,再操作0,1,2没有意义.
*/
close(0);
open("/dev/null", O_RDWR);
dup2(0, 1);
dup2(0, 2);
}
int main(void)
{
daemonize();
while(1);
/* 在此循环中可以实现守护进程的核心工作 */
}
```

二、微观角度

互斥

进程互斥是进程之间的间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。

比如进程B需要访问打印机，但此时进程A占有了打印机，进程B会被阻塞，直到进程A释放了打印机资源,进程B才可以继续执行。

同步

进程同步也是进程之间直接的制约关系，是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系。进程间的直接制约关系来源于他们之间的合作。

比如说进程A需要从缓冲区读取进程B产生的信息，当缓冲区为空时，进程B因为读取不到信息而被阻塞。而当进程A产生信息放入缓冲区时，进程B才会被唤醒。

Q15 什么是进程间的通信?

Q16 进程的通信有哪些措施?

整体来说，有下面这些措施

Q17 介绍一下信号机制

1.信号的介绍

2.常见的几种信号

3.信号的诞生

信号的诞生可以分为硬件来源和软件来源

硬件来源:比如我们按下了键盘或者其它硬件故障；

软件来源:一些非法运算等操作,人为的调用信号发送函数指定发送的信号,

4.信号的软件来源

信号常见的软件来源可以通过下面几个API来产生 `kill(),raise()、 sigqueue()、alarm()、setitimer()、abort()

- kill()
- raise()

```
raise ()
#include <signal.h>

int raise(int signo)

向进程本身发送信号，参数为即将发送的信号值。
调用成功返回 0；否则，返回 -1。
```

- sigqueue()

5.2　sigqueue ()

```
#include <sys/types.h>
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval val)
```

返回值:

调用成功返回 0；否则，返回 -1。

参数:

sigqueue的第一个参数是指定接收信号的进程ID，
第二个参数确定即将发送的信号，
第三个参数是一个联合数据结构union sigval，
指定了信号传递的参数，即通常所说的4字节值。

```
typedef union sigval {
               int  sival_int;
               void *sival_ptr;

}sigval_t;
```

使用如下

```
        union sigval mysigval;

        int i;

        int sig;

        pid_t pid;

        char data[10];

        memset(data,0,sizeof(data));

        for(i=0;i < 5;i++)

                data[i]='2';

        mysigval.sival_ptr=data;
```

区别:

sigqueue()是比较新的发送信号系统调用，
主要是针对实时信号提出的（当然也支持前32种），
支持信号带有参数，与函数sigaction()配合使用。

sigqueue()比kill()传递了更多的附加信息，
但sigqueue()只能向一个进程发送信号，
而不能发送信号给一个进程组。
如果signo=0，将会执行错误检查，
但实际上不发送任何信号，
0值信号可用于检查pid的有效性
以及当前进程是否有权限向目标进程发送信号。

在调用sigqueue时，sigval_t指定的信息
会拷贝到对应sig
注册的3参数信号处理函数的siginfo_t结构中，
这样信号处理函数就可以处理这些信息了。
由于sigqueue系统调用支持发送带参数信号，
所以比kill()系统调用的功能要灵活和强大得多。

- alarm()

```
 alarm ()
#include <unistd.h>

unsigned int alarm(unsigned int seconds)

系统调用alarm安排内核为
调用进程在指定的seconds秒后
给调用alarm的进程发出一个SIGALRM的信号。
如果指定的参数seconds为0,
则不再发送 SIGALRM信号。
后一次设定将取消前一次的设定。
该调用返回值为上次定时调用到
发送之间剩余的时间,
或者因为没有前一次定时调用而返回0。

注意，在使用时,
alarm只设定为发送一次信号,
如果要多次发送，就要多次使用alarm调用。
```

- setitimer()

```
5.4    setitimer ()

api
现在的系统中很多程序不再使用alarm调用，而是使用setitimer调用来设置定时器，用getitimer来得到定时器的状态，这两个调用的声明格式如下:

int getitimer(int which, struct itimerval *value);

int setitimer(int which, const struct itimerval *value, struct itimerval *ovalue);

在使用这两个调用的进程中加入以下头文件:

#include <sys/time.h>

 参数:

该系统调用给进程提供了三个定时器,
它们各自有其独有的计时域，当其中任何一个到达,
就发送一个相应的信号给进程，并使得计时器重新开始。
三个计时器由参数which指定，如下所示:


TIMER_REAL：按实际时间计时,
计时到达将给进程发送SIGALRM信号。


ITIMER_VIRTUAL：仅当进程执行时才进行计时。
计时到达将发送SIGVTALRM信号给进程。


ITIMER_PROF：当进程执行时和系统为该进程执行动作时都计时。
与ITIMER_VIR-TUAL是一对,
该定时器经常用来统计进程在用户态和内核态花费的时间。
计时到达将发送SIGPROF信号给进程。
```

定时器中的参数value用来指明定时器的时间，其结构如下：

```
struct itimerval {

        struct timeval it_interval; /* 下一次的取值 */

        struct timeval it_value; /* 本次的设定值 */

};
```

该结构中timeval结构定义如下：

```
struct timeval {

        long tv_sec; /* 秒 */

        long tv_usec; /* 微秒，1秒 = 1000000 微秒*/

};
```

返回值 ：
在setitimer 调用中，参数ovalue如果不为空，
则其中保留的是上次调用设定的值。
定时器将it_value递减到0时，产生一个信号，
并将it_value的值设定为it_interval的值，
然后重新开始计时，如此往复。当it_value设定为0时，
计时器停止，或者当它计时到期，
而it_interval 为0时停止。
调用成功时，返回0；错误时，
返回-1，并设置相应的错误代码errno：

EFAULT：参数value或ovalue是无效的指针。
EINVAL：参数which不是ITIMER_REAL、
ITIMER_VIRT或ITIMER_PROF中的一个。

- abort()

```
  abort()
#include <stdlib.h>

void abort(void);
```

向进程发送SIGABORT信号，默认情况下进程会异常退出，
当然可定义自己的信号处理函数。
即使SIGABORT被进程设置为阻塞信号，
调用abort()后，SIGABORT仍然能被进程接收。
该函数无返回值。

3.信号的注册

1.未决信号集

在进程表的task_struct表项中
有一个软中断信号域(未决信号集合),
该域中每一位对应一个信号。

内核给一个进程发送软中断信号的方法,
是在进程所在的进程表项的信号域
设置对应于该信号的位。

如果信号发送给一个正在睡眠的进程,
如果进程睡眠在可被中断的优先级上,
则唤醒进程;

否则仅设置进程表中信号域相应的位,
而不唤醒进程。

如果发送给一个处于可运行状态的进程,
则只置相应的域即可。

2.未决信号集相关数据结构

进程的task_struct结构中
有关于本进程中未决信号的数据成员：
struct sigpending pending;

```
struct sigpending{

        struct sigqueue *head, *tail;

        sigset_t signal;

};
```

第一、第二个成员
分别指向一个sigqueue类型的结构链
（称之为"未决信号信息链"）的首尾，
信息链中的每个sigqueue结构
刻画一个特定信号所携带的信息，
并指向下一个sigqueue结构：

```
struct sigqueue{

        struct sigqueue *next;

        siginfo_t info;

}
```

第三个成员是进程中所有未决信号集，
信号集数据结构

信号集被定义为一种数据类型：

```
typedef struct {
 unsigned long sig[_NSIG_WORDS];

} sigset_t
```

信号集用来描述信号的集合，每个信号占用一位。

信号在进程中注册指的就是信号值
加入到进程的未决信号集
sigset_t signal（每个信号占用一位）中，
并且信号所携带的信息
被保留到未决信号信息链的某个sigqueue结构中。
只要信号在进程的未决信号集中，
表明进程已经知道这些信号的存在，
但还没来得及处理，或者该信号被进程阻塞。

3.未决信号集相关操作函数

就是上面的信号发送函数

4.未决信号可靠非可靠信号注册区别:

当一个实时信号发送给一个进程时,
不管该信号是否已经在进程中注册,
都会被再注册一次,
因此,信号不会丢失,
因此,实时信号又叫做"可靠信号"。
这意味着同一个实时信号
可以在同一个进程的未决信号信息链中
占有多个sigqueue结构
(进程每收到一个实时信号,
都会为它分配一个结构来登记该信号信息,
并把该结构添加在未决信号链尾,
即所有诞生的实时信号都会在目标进程中注册)。

当一个非实时信号发送给一个进程时,
如果该信号已经在进程中注册
(通过sigset_t signal指示),
则该信号将被丢弃,造成信号丢失。
因此,非实时信号又叫做"不可靠信号"。
这意味着同一个非实时信号
在进程的未决信号信息链中,
至多占有一个sigqueue结构。

总之信号注册与否,
与发送信号的函数(如kill()或sigqueue()等)
以及信号安装函数(signal()及sigaction())无关,
只与信号值有关
(信号值小于SIGRTMIN的信号最多只注册一次,
信号值在SIGRTMIN及SIGRTMAX之间的信号,
只要被进程接收到就被注册)

5.信号集操作
信号集操作函数

Linux所支持的所有信号
可以全部或部分的出现在信号集中，
主要与信号阻塞相关函数配合使用。
下面是为信号集操作定义的相关函数：

```
#include <signal.h>
// 以下函数，成功，返回 0；失败，返回 -1.
int sigemptyset(sigset_t *set)                  // 清除信号集中的所以信号
int sigfillset(sigset_t *set)                   // 初始化set信号集中的信号，并把所有该程序的信号加入到信号集中
int sigaddset(sigset_t *set, int signo)         // 添加signo信号到信号集中
int sigdelset(sigset_t *set, int signo)         // 信号集中删除signo信号
int sigismember(const sigset_t *set, int signum); //判断signum位置是否为1
```

sigemptyset(sigset_t *set)初始化由set指定的信号集，信号集里面的所有信号被清空；

sigfillset(sigset_t *set)调用该函数后，set指向的信号集中将包含linux支持的64种信号；

sigaddset(sigset_t *set, int signum)在set指向的信号集中加入signum信号；

sigdelset(sigset_t *set, int signum)在set指向的信号集中删除signum信号；

sigismember(const sigset_t *set, int signum)判定信号signum是否在set指向的信号集中。

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
void print_sigset(sigset_t *set);
int main(void)
{
    sigset_t myset;
    sigemptyset(&myset);
    sigaddset(&myset,SIGINT);
    sigaddset(&myset,SIGQUIT);
    sigaddset(&myset,SIGUSR1);
    sigaddset(&myset,SIGRTMIN);
    print_sigset(&myset);

    return 0;

}
void print_sigset(sigset_t *set)
{
    int i;
    for(i = 1; i < NSIG; ++i){
        if(sigismember(set,i))
            printf("1");
        else
            printf("0");
    }
    putchar('\n');
}
```

4.信号的屏蔽

每个进程都有一个用来描述哪些信号递送到进程时将被阻塞的信号集，该信号集中的所有信号在递送到进程后都将被阻塞。下面是与信号阻塞相关的几个函数：

```c
int sigsuspend(const sigset_t *mask));
```

sigsuspend(const sigset_t *mask))用于在接收到某个信号之前，临时用mask替换进程的信号掩码，并暂停进程执行，直到收到信号为止。sigsu

```c
int sigpending(sigset_t *set));
```

sigpending(sigset_t *set))获得当前已递送到进程，却被阻塞的所有信号，在set指向的信号集中返回结果。

5.信号的处理时机

在其从内核空间返回到用户空间时会检测是否有信号等待处理。

如果存在未决信号等待处理且该信号没有被进程阻塞，

则在运行相应的信号处理函数前，进程会把信号在未决信号链中占有的结构卸掉。

对于非实时信号来说，由于在未决信号信息链中最多只占用一个sigqueue结构，因此该结构被释放后，应该把信号在进程未决信号集中删除（信号注销完毕）；

而对于实时信号来说，可能在未决信号信息链中占用多个sigqueue结构，因此应该针对占用sigqueue结构的数目区别对待：

如果只占用一个sigqueue结构（进程只收到该信号一次），则执行完相应的处理函数后应该把信号在进程的未决信号集中删除（信号注销完毕）。

否则待该信号的所有sigqueue处理完毕后再在进程的未决信号集中删除该信号。

当所有未被屏蔽的信号都处理完毕后，即可返回用户空间。

对于被屏蔽的信号，当取消屏蔽后，在返回到用户空间时会再次执行上述检查处理的一套流程。

6.信号递达以后的执行动作

收到信号的进程对各种信号有不同的处理方法。处理方法可以分为三类：

第一种方法是，对该信号的处理保留系统的默认值，这种缺省操作，对大部分的信号的缺省操作是使得进程终止。

第二种方法是，忽略某个信号，对该信号不做任何处理，就象未发生过一样。

第三种是对于需要处理的信号，进程可以指定指定处理函数，即信号的的安装 见下一点

7.信号的安装

如果进程要处理某一信号，那么就要在进程中安装该信号。

安装信号主要用来确定信号值及进程针对该信号值的动作之间的映射关系，即进程将要处理哪个信号；该信号被传递给进程时，将执行何种操作。

linux主要有两个函数实现信号的安装：signal()、sigaction()。其中signal()只有两个参数，不支持信号传递信息，主要是用于前32种非实时信号的安装；

而sigaction()是较新的函数（由两个系统调用实现：sys_signal以及sys_rt_sigaction），有三个参数，支持信号传递信息，主要用来与 sigqueue() 系统调用配合使用，

当然，sigaction()同样支持非实时信号的安装。sigaction()优于signal()主要体现在支持信号带有参数。

- sigaction
- signal

```
#include <signal.h>
#include <unistd.h>
#include <stdio.h>
void sigroutine(int dunno)
{ /* 信号处理例程，其中dunno将会得到信号的值 */
        switch (dunno) {
        case 1:
        printf("Get a signal -- SIGHUP ");
        break;
        case 2:
        printf("Get a signal -- SIGINT ");
        break;
        case 3:
        printf("Get a signal -- SIGQUIT ");
        break;
        }
        return;
}

int main() {
        printf("process id is %d ",getpid());
        signal(SIGHUP, sigroutine); //* 下面设置三个信号的处理方法
        signal(SIGINT, sigroutine);
        signal(SIGQUIT, sigroutine);
        for (;;) ;
}
```

其中信号SIGINT由按下Ctrl-C发出，信号SIGQUIT由按下Ctrl-发出。该程序执行的结果如下：

```
localhost:~$ ./sig_test
```

```
process id is 463

Get a signal -SIGINT //按下Ctrl-C得到的结果

Get a signal -SIGQUIT //按下Ctrl-得到的结果

//按下Ctrl-z将进程置于后台

 [1]+ Stopped ./sig_test

localhost:~$ bg

 [1]+ ./sig_test &

localhost:~$ kill -HUP 463 //向进程发送SIGHUP信号

localhost:~$ Get a signal — SIGHUP

kill -9 463 //向进程发送SIGKILL信号，终止进程

localhost:~$
```

Q18 介绍一下管道机制

1.有名管道

2.无名管道

Q18 介绍一下消息队列机制

1.概念

2.特点

3.使用

Q6 介绍一下共享内存机制

1.介绍

2.特点

3.原理

4.指令

5.使用

Q19 介绍一下socket套接字机制

一般用于两个不同IP主机之间的通信，在后面 网络编程 那一部分会讲

Q20 介绍一下信号量机制

1.介绍

2.实现机制

3.PV操作

4.应用：生产者消费者问题

5.应用：读者写者问题

6.应用：哲学家进餐问题