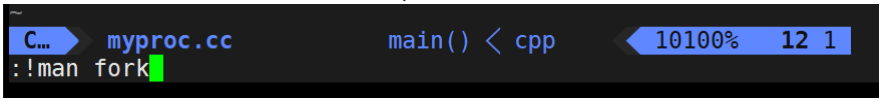
进程概念done

补：

- 进程是具有独立性的
- 写实拷贝：简单点理解就是，当有一个人想要修改父子进程里面的数据的时候，操作系统不会立即就对代码进行修改，而是重新开辟一个进程，把当前进程的执行信息拷贝过去，然后在新开辟的进程中去做数据的修改，这样可以保护数据的独立性
- 我们使用grep去过滤进程时通常使用：**ps axj | grep bush(正在运行的进程名)**，但是当这条命令运行起来时他也是一个进程，会查到他自己，如果不想查到他自己可以:**ps axj | grep bush | grep -v grep**

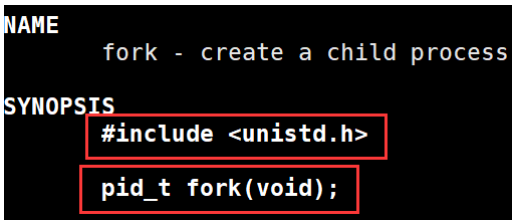
在vim中如果不想退出vim去查man手册可以：

- 在命令行输入!man 后面加要查的(q退出)



- 不退出编译程序：!make
- 不退出执行程序：！./myproc
- vs可以让vim双开进程，ctrl+两次w是两个文件相互切换
- 前台进程(状态后有+)：特点就是可以用ctrl+c 来结束进程，输入一些命令行指令没有反应
- 后台进程(状态后没有+)：特点就是可以执行很多命令行指令，但是ctrl+c已经不能杀掉进程了，用可以用kill杀

通过系统调用创建进程--fork初识



如何理解fork创建子进程

1. **./cmd or run command(跑某些命令),fork**：在操作系统角度，上面的创建进程的方式，没有差别。
2. fork本质是创建进程---->系统里面多了一个进程---->与进程相关的内核数据结构(task_struct)+进程的代码和数据
    - 我们只是fork了创建了子进程，但是子进程对应的代码数据呢？--->**默认情况下会"继承"父进程的代码和数据，内核数据结构task_struct也会以父进程为模板，初始化子进程的task_struct**
    - fork之后父子进程代码是共享的(之前也是相同的只是父进程的程序计数器的属性也会被子进程继承下来，所以子进程不会去执行fork之前的代码)
    - 一般父子代码只有一份，默认情况下数据也是"共享的"，不过需要考虑修改的情况！(通过**写实拷贝**来完成进程数据的独立性)

fork的返回值

- 通过fork的返回值来让子进程和父进程做不同的事情
  - 失败 < 0
  - 成功：给父进程返回子进程的pid，给子进程返回0

```
[xifeng@VM-16-14-centos lesson9]$ ./myporc
hello proc->29777 hello parent->11003 id:29778
hello proc->29778 hello parent->29777 id:0
[xifeng@VM-16-14-centos lesson9]$ cat myproc.cc
#include<iostream>
#include<unistd.h>
int main()
{
 int ret= fork();
  std::cout<<"hello proc->"<<getpid()<<" hello parent->"<<getppid()<<" id:"<<ret<<std::endl;
 sleep(1);
 return 0;
}
[xifeng@VM-16-14-centos lesson9]$
```

  - 父子进程谁先return谁就会发生写实拷贝
- 可以看到一点就是那个父进程的父进程是就是命令行(bash)

```
[xifeng@VM-16-14-centos lesson9]$ ./myporc
hello proc->20430 hello parent->11003
hello proc->20431 hello parent->20430
[xifeng@VM-16-14-centos lesson9]$ ps axj | head -1 && ps axj | grep 11003
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
11002 11003 11003 11003 pts/0    20602 Ss    1001   0:00 -bash
11003 20602 20602 11003 pts/0    20602 R+    1001   0:00 ps axj
11003 20603 20602 11003 pts/0    20602 R+    1001   0:00 grep --color=auto 11003
```

父子关系

- 如何理解返回值的设置
  - 父：子=1：n因为是一对多，所以需要让父进程来控制子进程的话就需要子进程的pid
  - 可以发现有两个进程在跑

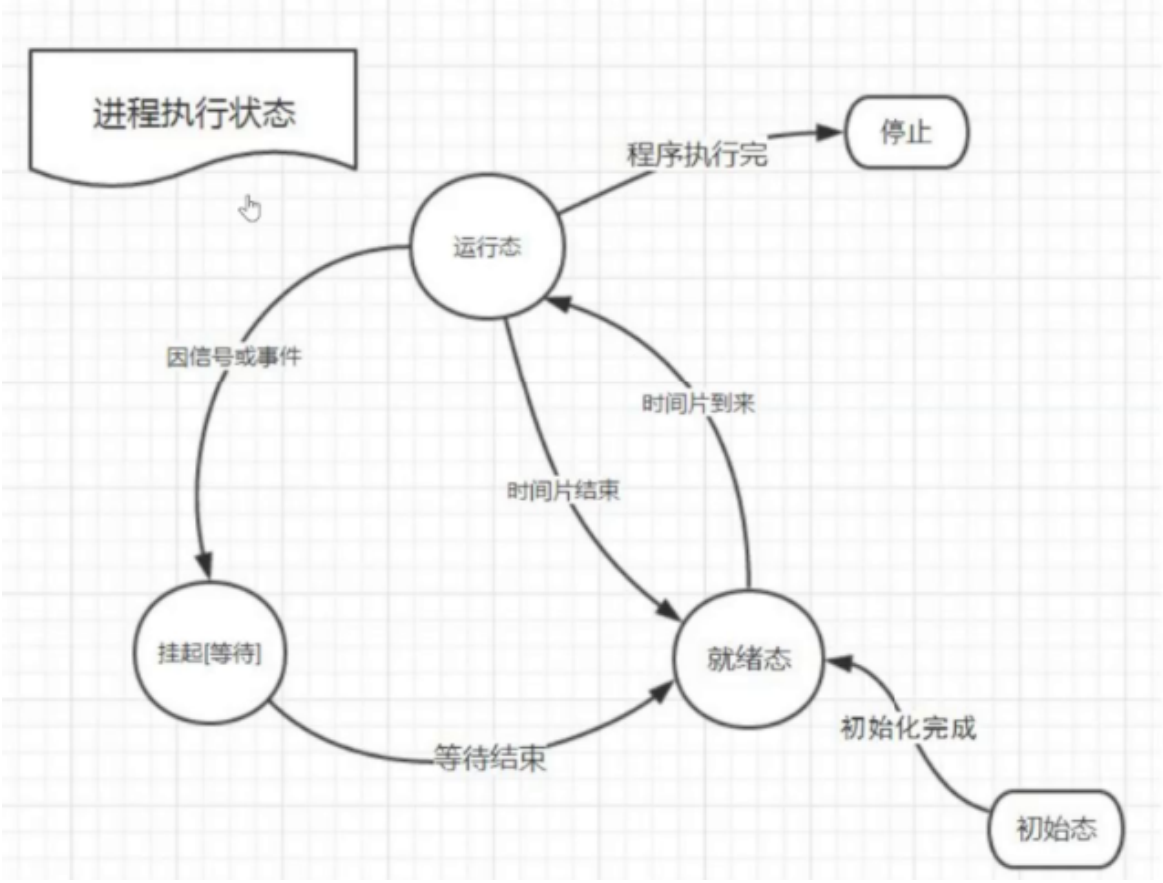
```
#include<iostream>
#include<unistd.h>
int main()
{
  gid_t id =fork();
  if(id==0)
  {
    while(true){
    std::cout<<"I am child,pid:"<<getpid()<<" ppid:"<<getppid()<<std::endl;
    sleep(1);
    }
  }
  else if(id >0)
  {
    while(true){
     std::cout<<"I am father,pid:"<<getpid()<<" ppid:"<<getppid()<<std::endl;
     sleep(1);  }
  }
  else
    sleep(1);
  return 0;
}
```

```
I am father,pid:11015 ppid:11003
I am child,pid:11016 ppid:11015
I am father,pid:11015 ppid:11003
I am child,pid:11016 ppid:11015
I am father,pid:11015 ppid:11003
I am child,pid:11016 ppid:11015
```

```
[xifeng@VM-16-14-centos lesson9]$ ps axj | grep test
11003 11015 11015 11003 pts/0    11015 S+    1001   0:00 ./test
11015 11016 11015 11003 pts/0    11015 S+    1001   0:00 ./test
```

  - 通过if语句来分流；fork之后父子**不确定**谁先运行(**调度器**会自己决定)

进程状态



进程的状态信息在task_struct(PCB)中

- 进程状态的意义：方便OS快速判断进程，完成特定的功能，比如调度(本质是一种分类)
- 具体状态：

1. **R(running运行状态)**：不一定正在占用CPU，处于运行状态的含义是**处于运行队列当中，可以随时被CPU调度**
2. 当我们完成某种任务的时候，任务条件不具备，需要进程进行某种等待可以用**S**或者**D**表示（进程不会只等待CPU资源也会等待其他的资源，这时被称为等待队列）
3. **所谓的进程，在运行的时候，可能因为运行需要，而在不同的队列里！！在不同的队列里所处的状态是不一样的！！**

我们把，从运行状态的task_struct放到等待队列中，就叫做挂起等待(阻塞)；从等待队列，放到运行队列，被CPU调度就叫做唤醒进程

4. **S(sleeping浅度睡眠状态)**：可中断睡眠状态
5. **D(disk sleep深度睡眠状态)**：不可中断状态(进程如果处于D状态，不可被杀掉！)
6. **T(stopped暂停状态)**：数据就不会在进行更新

kill -l 可以查看一些对进程的操作，给进程发信号可以使进程暂停或者运行

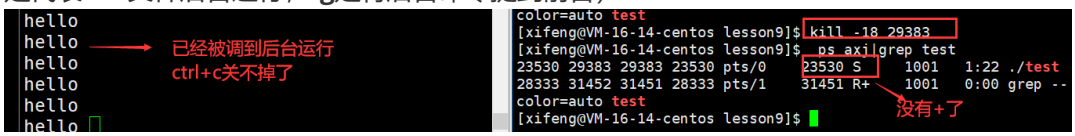
```
[xifeng@VM-16-14-centos lesson9]$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
GTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL
GUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM
GTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP
GTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU
GXFSZ                                                  暂停进程
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
GPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2
GRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7
GRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12
GRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
GRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8
GRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3
GRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

继续进程

暂停用法就是:**kill -19 PID**

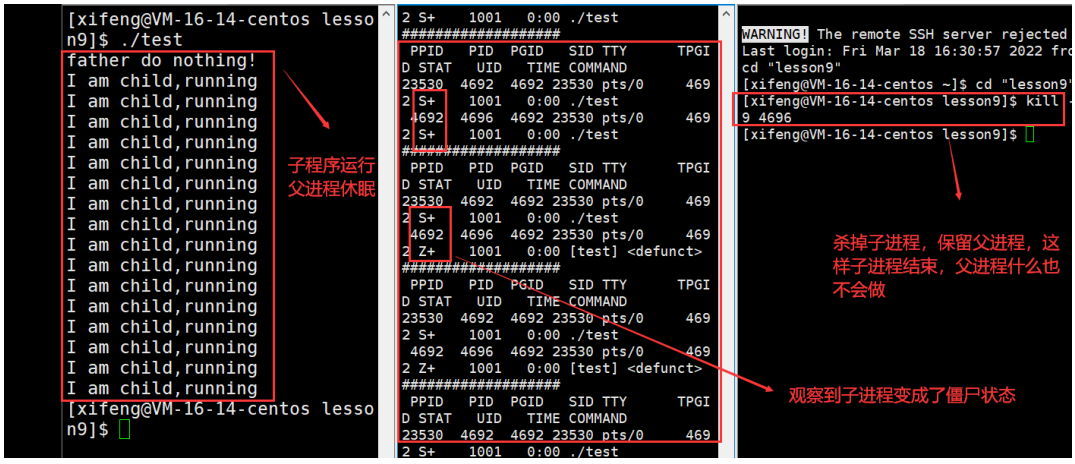
```
8333 31046 31045 28333 pts/1    31045 R+    1001   0:00 grep --
olor=auto test
xifeng@VM-16-14-centos lesson9]$ kill -19 29383
xifeng@VM-16-14-centos lesson9]$  ps axj|grep test
3530 29383 29383 23530 pts/0    23530 T     1001   1:21 ./test
8333 31116 31115 28333 pts/1    31115 R+    1001   0:00 grep --
olor=auto test
xifeng@VM-16-14-centos lesson9]$
```

继续运行用法：**kill -18 PID**（但是会被调到后台运行可以用 kill -9 来结束运行, **./test &**这个是代表test文件后台运行，**fg**是将后台命令提到前台）

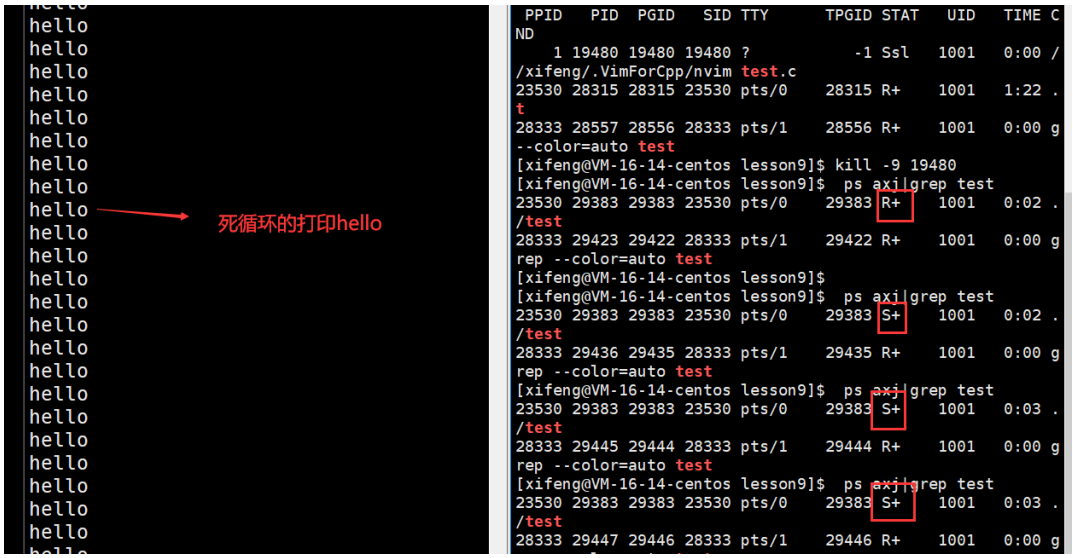


7. **t(tracing stop追踪状态)**：也是一种暂停，断点的应用就是追踪状态
8. **X(dead 死亡状态)**：回收进程资源=进程相关的内核数据结构+代码和数据
9. **Z(zombie 僵尸状态)**： 辨别退出死亡原因(获取进程退出的信息)；如果没有人检测或回收进程(父进程)，该进程退出就进入Z

   自动查看进程的命令行脚本：while : ; do ps axj |head -1 && ps axj |grep test |grep -v grep; sleep 1; echo "####################"; done (大概意思就是每1秒查看一次进程的数据，并且打印完之后会打印################来分隔)

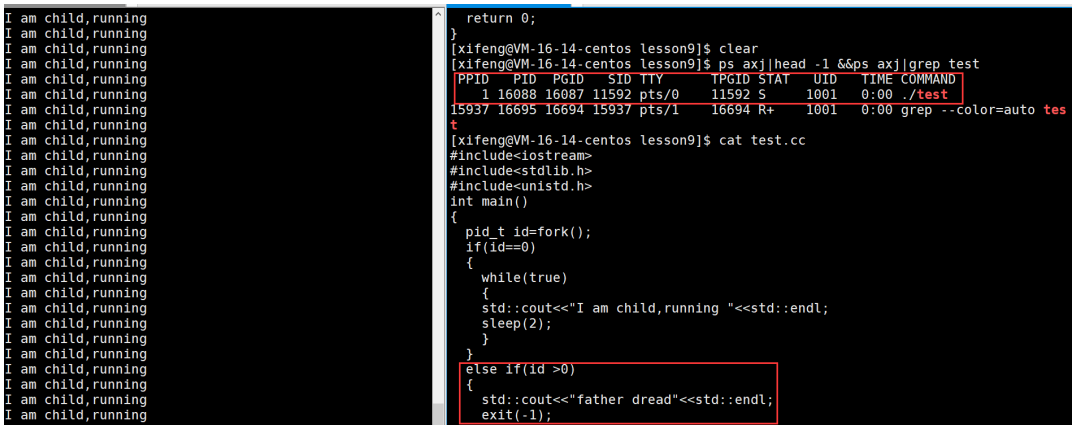


10. 有一种现象



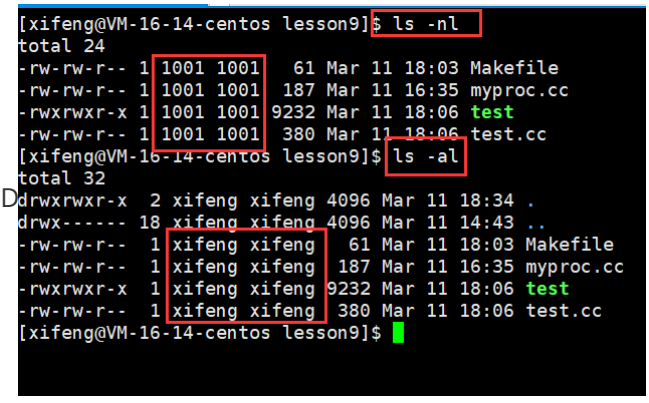
当死循环打印hello的时候可以发现只有很少的时间进程处于R状态，大部分时间处于S状态！！！这是因为数据输出到外设很慢，IO等待外设是需要花时间的，所以大部分时间都在休眠

- **孤儿进程**
  - 父进程先死那么就会由1号进程领养，1号进程也就是操作系统



进程优先级

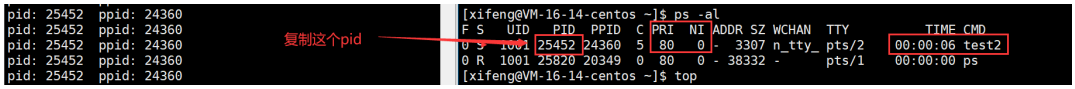
- **基本概念**
  - CPU资源分配的先后顺序，就是指进程的优先权
  - 优先权高的进程有优先执行权利。配置进程优先权对多任务环境的Linux很有用，可以改善系统性能。
  - 还可以把进程运行到指定的CPU上，这样一来，把不重要的进程安排到某个CPU，可以大大改善系统整 体性能。
- **查看系统进程**
  - **ps -l** 查看进程(PRI就是优先级)
  - **ls -nl** 查看UID



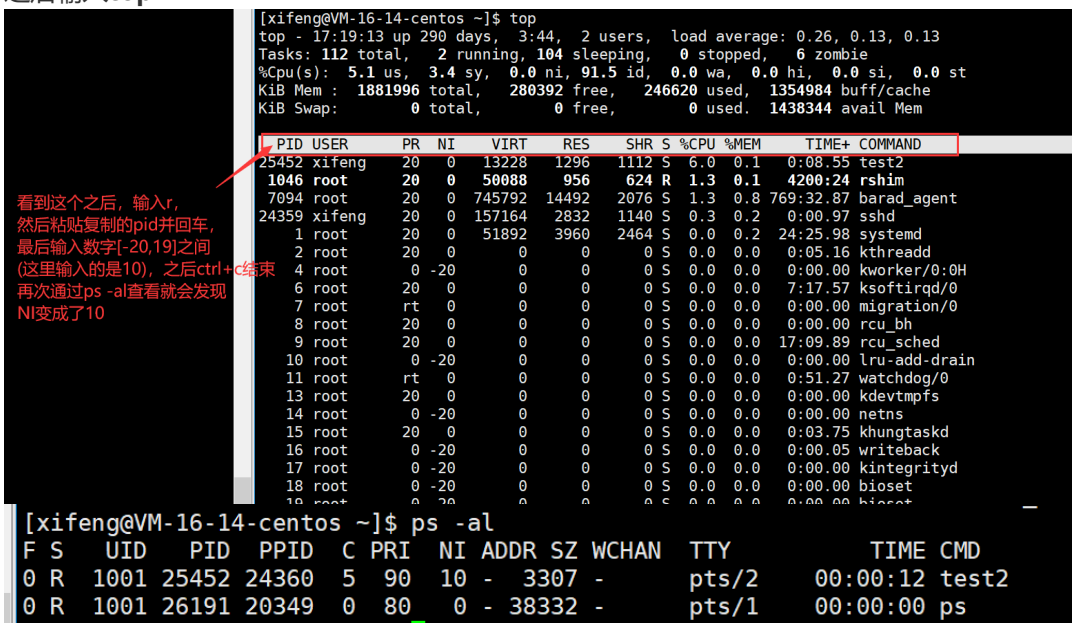
- **Linux中的优先级数据，值越小优先级越高**

PRI and NI

- PRI就是对应的优先级数据
- NI(nice)就是对应的修正数据
- 加入nice值后，PRI(new)=PRI(old)+nice (每一次都会重置,例如：第一次设置的NI是10，PRI是90，第二次NI改5的话，PRI是85，也就是每一次PRI都会以80来计算)
- nice其取值范围是-20至19，一共40个级别
- 修改进程的优先级，系统提供了一些接口，但是不推荐随意去改优先级；也有指令更改如没有运行起来可以用**nice**，运行起来的进程可以用**renice**，具体使用可以去查看man手册
- 这里使用**top**命令来简单的操作一下



之后输入**top**



我们发现NI已经变成10，而且PRI也变成了90，说明更改成功了(超过[-20,19]，去两边的极值来算，不会超过-20或19)

nice值为什么是一个相对比较小的范围呢！

- 会产生严重的进程“饥饿问题”
- 就相当于排队打饭，增加优先级就相当于是插队，如果就你很老实，其他人都往你前面插队，这样你就会一直打不到饭没有资源，从而产生很严重的问题

PRI vs NI

- 要强调一点的是，进程的nice值不是进程的优先级，他们不是一个概念，但是进程nice值会影响到进程的优先级变化。
- 可以理解nice值是进程优先级的修正数据

进程优先级的调整：

- 优先级再怎么设置，也只能是一种相对的优先级，不能出现绝对的优先级，否则会出现很严重的进程**“饥饿问题”**
- **调度器：较为均衡的让每个进程享受CPU的资源**

其他概念：

- **竞争性**: 系统进程数目众多，而CPU资源只有少量，甚至1个，所以进程之间是具有竞争属性的。为了高效完成任务，更合理竞争相关资源，便具有了优先级
- **独立性**: 多进程运行，需要独享各种资源，多进程运行期间互不干扰

- **并行**: 多个进程在多个CPU下分别，同时进行运行，这称之为并行(**任何一个时刻**)
- **并发**: 多个进程在一个CPU下采用进程切换的方式，在一段时间之内，让多个进程都得以推进，称之为并发(**某个时间段内**)
- **并行并发可以同时存在**