### 实验四：进程管理（二）

2．阅读如下程序：

/\* process using time \*/

#include<stdio.h>

#include<stdlib.h>

#include<sys/times.h>

#include<time.h>

#include<unistd.h>

void time\_print(char \*,clock\_t);

int main(void) {

clock\_t start,end;

struct tms t\_start,t\_end;

start = times(&t\_start);

system("grep the /usr/doc/\*/\* > /dev/null 2> /dev/null");

end=times(&t\_end);

time\_print("elapsed",end-start);

puts("parent times");

time\_print("\tuser CPU",t\_end.tms\_utime);

time\_print("\tsys CPU",t\_end.tms\_stime);

puts("child times");

time\_print("\tuser CPU",t\_end.tms\_cutime);

time\_print("\tsys CPU",t\_end.tms\_cstime);

exit(EXIT\_SUCCESS);

}

void time\_print(char \*str,clock\_t time) {

long tps = sysconf(\_SC\_CLK\_TCK);

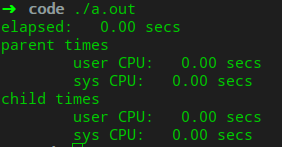
printf("%s: %6.2f secs\n",str,(float)time/tps);

}

编译并运行，分析进程执行过程的时间消耗（总共消耗的时间和CPU消耗的时间），并解释执行结果。再编写一个计算密集型的程序替代grep，比较两次时间的花销。注释程序主要语句。

答：

上述代码执行的结果如下：

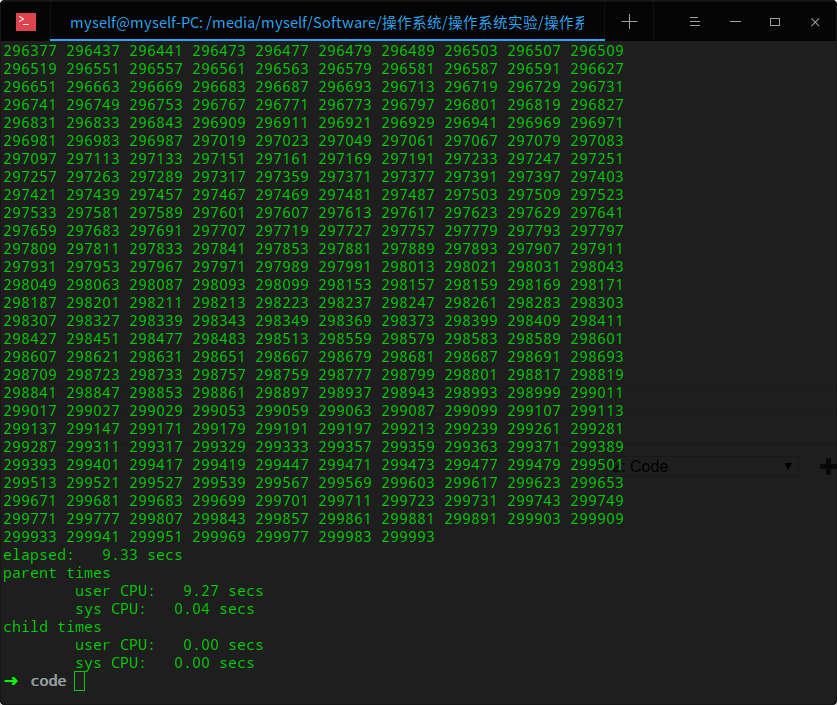


由于该程序的计算量很小，因此消耗的时间比较少，因不足10ms而直接显示为0.00secs。进程的执行时间等于用户CPU时间和系统CPU时间加从硬盘读取数据时间之和。

将grep替换为计算密集型程序，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  14  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | /\*  process using time \*/  #include<stdio.h>  #include<stdlib.h>  #include<sys/times.h>  #include<time.h>  #include<unistd.h>  void time\_print(char \*,clock\_t);  void calc();  int main(void) {  clock\_t start,end;  struct tms t\_start,t\_end;  start = times(&t\_start); //计算起始时间    calc();  // system("grep the /usr/doc/\*/\* > /dev/null 2> /dev/null");  /\* command > /dev/null是将标准输出重定向到空设备中（丢弃）  command 2> /dev/null是将标准错误重定向到空设备中（丢弃） \*/  end=times(&t\_end); //计算结束时间  time\_print("elapsed",end-start);  puts("parent times");  time\_print("\tuser CPU",t\_end.tms\_utime);  time\_print("\tsys CPU",t\_end.tms\_stime);  puts("child times");  time\_print("\tuser CPU",t\_end.tms\_cutime);  time\_print("\tsys CPU",t\_end.tms\_cstime);  exit(EXIT\_SUCCESS);  }  void time\_print(char \*str,clock\_t time) { //输出时间  long tps = sysconf(\_SC\_CLK\_TCK); //获得CPU内部时钟每秒中断个数  printf("%s: %6.2f secs\n",str,(float)time/tps);  }  void calc() //计算密集型程序，计算0-300000之间的素数并输出  {  int i = 0, j = 0, sum = 0;  for(i = 0; i < 300000; i++)  {  for(j = 2; j < i; j++)  {  if(i % j == 0) break;  }  if(j == i)  {  printf("%d ", i);  sum++;  if(sum % 10 == 0) putchar('\n');  }  }  putchar('\n');  } |

编译并执行代码输出结果如下：



可以看到，更改为计算密集型程序后更容易看出消耗时间的差异。

4．阅读下列程序：

/\* usage of kill,signal,wait \*/

#include<unistd.h>

#include<stdio.h>

#include<stdlib.h>

#include<sys/types.h>

#include<wait.h>

#include<signal.h>

int flag;

void stop();

int main(void)

{

int pid1,pid2;

signal(3,stop);

while((pid1=fork()) ==-1);

if(pid1>0){

while((pid2=fork()) ==-1);

if(pid2>0){

flag=1;

sleep(5);

kill(pid1,16);

kill(pid2,17);

wait(0);

wait(0);

printf("\n parent is killed\n");

exit(EXIT\_SUCCESS);

}else{

flag=1;

signal(17,stop);

printf("\n child2 is killed by parent\n");

exit(EXIT\_SUCCESS);

}

}else{

flag=1;

signal(16,stop);

printf("\n child1 is killed by parent\n");

exit(EXIT\_SUCCESS);

}

}

void stop(){

flag = 0;

}

编译并运行，等待或者按^C，分别观察执行结果并分析，注释程序主要语句。

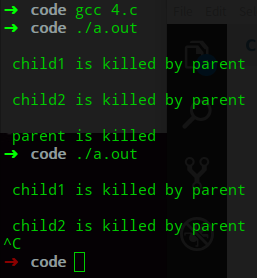
flag有什么作用？通过实验说明。

答：

程序代码注释如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  14  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | /\* usage of kill,signal,wait \*/  #include <unistd.h>  #include <stdio.h>  #include <stdlib.h>  #include <sys/types.h>  #include <wait.h>  #include <signal.h>  int flag;  void stop();  int main(void)  {  int pid1, pid2;  signal(3, stop); //设置信号3的处理函数为stop  while ((pid1 = fork()) == -1) ; //创建一个子进程  if (pid1 > 0) //当前进程为父进程  {  while ((pid2 = fork()) == -1) ; //创建另一个子进程  if (pid2 > 0) //当前进程为父进程  {  flag = 1;  sleep(5);  kill(pid1, 16); //向子进程1发送信号16  kill(pid2, 17); //向子进程2发送信号17  wait(0); //暂停当前进程，直到信号来到或子进程结束  wait(0);  printf("\n parent is killed\n");  exit(EXIT\_SUCCESS);  }  else //当前进程为子进程2  {  flag = 1;  signal(17, stop); //设置信号17的处理函数为stop  printf("\n child2 is killed by parent\n");  exit(EXIT\_SUCCESS);  }  }  else //当前进程为子进程1  {  flag = 1;  signal(16, stop);  printf("\n child1 is killed by parent\n");  exit(EXIT\_SUCCESS);  }  }  void stop()  {  flag = 0;  } |

执行上述代码的结果如下：



父进程和子进程都有一个flag，为1表示该进程正在运行，为0时表示该进程结束。

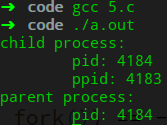
5．编写程序，要求父进程创建一个子进程，使父进程和个子进程各自在屏幕上输出一些信息，但父进程的信息总在子进程的信息之后出现。

答：

程序代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  14  16  17  18  19  20  21  22 | #include <stdio.h>  #include <unistd.h>  #include <wait.h>  int main(void)  {  int p;  while((p = fork()) == -1); //创建一个子进程  if(p > 0) //父进程  {  wait(0);  printf("parent process:\n");  printf("\tpid: %d\n", p);  }  else //子进程  {  printf("child process:\n");  printf("\tpid: %d\n", getpid());  printf("\tppid: %d\n", getppid());  }  return 0;  } |

该程序执行的结果如下：

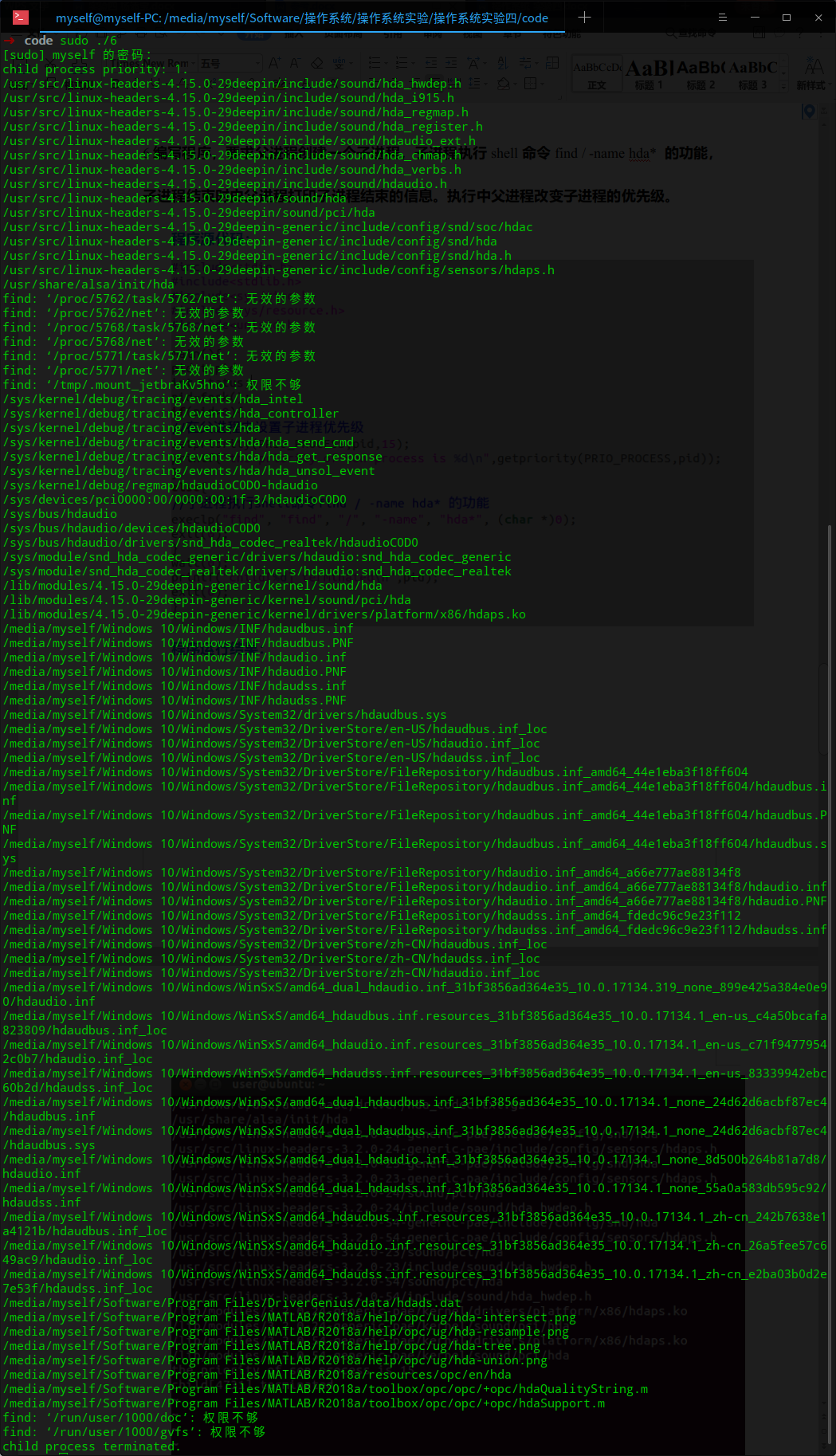


6．编写程序，要求父进程创建一个子进程，子进程执行shell命令find / -name hda\*的功能，子进程结束时由父进程打印子进程结束的信息。执行中父进程改变子进程的优先级。

答：程序代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  14  16  17  18  19  20  21  22 | #include <stdio.h>  #include <unistd.h>  #include <wait.h>  #include <sys/resource.h>  int main(void)  {  int p;  while((p = fork()) == -1); //创建一个子进程  if(p > 0) //父进程  {  setpriority(PRIO\_PROCESS, p, 1);  printf("child process priority: %d.\n", getpriority(PRIO\_PROCESS, p));  wait(0); //等待子进程结束  printf("child process terminated.\n");  }  else //子进程  {  execlp("find", "find", "/", "-name", "hda\*", NULL);  }  return 0;  } |

程序的执行结果如下：



8．查阅Linux系统中struct task\_struct的定义，说明每项成员的作用。

注：search in /usr/src/linux-2.6/include/linux/sched.h

答：

广义上，所有的进程信息被放在一个叫做进程控制块的数据结构中，可以理解为进程属性的集合。每个进程在内核中都有一个进程控制块(PCB)来维护进程相关的信息,Linux内核的进程控制块是task\_struct结构体。task\_struct是Linux内核的一种数据结构，它会被装载到RAM里并且包含着进程的信息。每个进程都把它的信息放在task\_struct这个数据结构里，task\_struct包含了这些内容：

（1）标示符：描述本进程的唯一标示符，用来区别其他进程。

（2）状态：任务状态，退出代码，退出信号等。

（3）优先级：相对于其他进程的优先级。

（4）程序计数器：程序中即将被执行的下一条指令的地址。

（5）内存指针：包括程序代码和进程相关数据的指针，还有和其他进程共享的内存块的指针。

（6）上下文数据：进程执行时处理器的寄存器中的数据。

（7）I/O状态信息：包括显示的I/O请求,分配给进程的I/O设备和被进程使用的文件列表。

（8）记账信息：可能包括处理器时间总和，使用的时钟数总和，时间限制，记账号。

… …

保存进程信息的数据结构叫做task\_struct，并且可以在 include/linux/sched.h 里找到它。

所有运行在系统里的进程都以task\_struct链表的形式存在内核里。进程的信息可以通过 /proc 系统文件夹查看。

**task\_struct一些字段的介绍：**

**1. 调度数据成员**

(1) volatile long states;

表示进程的当前状态

(2) unsigned long flags;

进程标志

(3) long priority;

进程优先级。优先级可通过系统调用sys\_setpriorty改变。

(4) unsigned long rt\_priority;

rt\_priority给出实时进程的优先级，rt\_priority+1000给出进程每次获取CPU后可使用的时间(同样按jiffies计)。实时进程的优先级可通过系统调用sys\_sched\_setscheduler()改变(见kernel/sched.c)。

(5) long counter;

在轮转法调度时表示进程当前还可运行多久。在进程开始运行是被赋为priority的值，以后每隔一个tick(时钟中断)递减1，减到0时引起新一轮调度。重新调度将从run\_queue队列选出counter值最大的就绪进程并给予CPU使用权，因此counter起到了进程的动态优先级的作用(priority则是静态优先级)。

(6) unsigned long policy;

该进程的进程调度策略，可以通过系统调用sys\_sched\_setscheduler()更改(见kernel/sched.c)。调度策略有:

SCHED\_OTHER   0   非实时进程，基于优先权的轮转法(round robin)。

SCHED\_FIFO     1   实时进程，用先进先出[算法](http://lib.csdn.net/base/datastructure)。

SCHED\_RR       2   实时进程，用基于优先权的轮转法。

**2. 信号处理**

(1) unsigned long signal;

进程接收到的信号。每位表示一种信号，共32种。置位有效。

(2)  unsigned long blocked;

进程所能接受信号的位掩码。置位表示屏蔽，复位表示不屏蔽。

(3) struct signal\_struct \*sig;

因为signal和blocked都是32位的变量，Linux最多只能接受32种信号。对每种信号，各进程可以由PCB的sig属性选择使用自定义的处理函数，或是系统的缺省处理函数。指派各种信息处理函数的结构定义在include/linux/sched.h中。对信号的检查安排在系统调用结束后，以及“慢速型”中断服务程序结束后(IRQ#\_interrupt())。

**3. 进程队列指针**

(1) struct task\_struct \*next\_task, \*prev\_task;

所有进程(以PCB的形式)组成一个双向链表。next\_task和就是链表的前后指针。链表的头和尾都是init\_task(即0号进程)。

(2) struct task\_struct \*next\_run, \*prev\_run;

由正在运行或是可以运行的，其进程状态均为TASK\_RUNNING的进程所组成的一个双向循环链表，即run\_queue就绪队列。该链表的前后向指针用next\_run和prev\_run，链表的头和尾都是init\_task(即0号进程)。

(3) struct task\_struct \*p\_opptr, \*p\_pptr;和struct task\_struct \*p\_cptr, \*p\_ysptr, \*p\_osptr;

以上分别是指向原始父进程(original parent)、父进程(parent)、子进程(youngest child)及新老兄弟进程(younger sibling，older sibling)的指针。

**4. 进程标识**

(1) unsigned short uid，gid;

uid和gid是运行进程的用户标识和用户组标识。

(2) int groups[NGROUPS];

与多数现代UNIX[操作系统](http://lib.csdn.net/base/operatingsystem)一样，Linux允许进程同时拥有一组用户组号。在进程访问文件时，这些组号可用于合法性检查。

(3) unsigned short euid，egid;

euid和egid又称为有效的uid和gid。出于系统安全的权限的考虑，运行程序时要检查euid和egid的合法性。通常，uid等于euid，gid等于egid。有时候，系统会赋予一般用户暂时拥有root的uid和gid(作为用户进程的euid和egid)，以便于进行运作。

(4) unsigned short fsuid, fsgid;

fsuid和fsgid称为文件系统的uid和gid，用于文件系统操作时的合法性检查，是Linux独特的标识类型。它们一般分别和euid和egid一致，但在NFS文件系统中NFS服务器需要作为一个特殊的进程访问文件，这时只修改客户进程的fsuid和fsgid。

(5) unsigned short suid, sgid;

suid和sgid是根据POSIX标准引入的，在系统调用改变uid和gid时，用于保留真正的uid和gid。

(6) int pid, pgrp, session;

进程标识号、进程的组织号及session标识号，相关系统调用(见程序kernel/sys.c)有sys\_setpgid、sys\_getpgid、sys\_setpgrp、sys\_getpgrp、sys\_getsid及sys\_setsid几种。

(7) int leader;

是否是session的主管，布尔量。

**5. 时间数据成员**

(1) unsigned long timeout;

用于软件定时，指出进程间隔多久被重新唤醒。采用tick为单位。

(2) unsigned long it\_real\_value, it\_real\_iner;

用于itimer(interval timer)软件定时。采用jiffies为单位，每个tick使it\_real\_value减到0时向进程发信号SIGALRM，并重新置初值。初值由it\_real\_incr保存。具体代码见kernel/itimer.c中的函数it\_real\_fn()。

(3) struct timer\_list real\_timer;

一种定时器结构(Linux共有两种定时器结构，另一种称作old\_timer)。[数据结构](http://lib.csdn.net/base/datastructure)的定义在include/linux/timer.h中，相关操作函数见kernel/sched.c中add\_timer()和del\_timer()等。

(4) unsigned long it\_virt\_value, it\_virt\_incr;

关于进程用户态执行时间的itimer软件定时。采用jiffies为单位。进程在用户态运行时，每个tick使it\_virt\_value减1，减到0时向进程发信号SIGVTALRM，并重新置初值。初值由it\_virt\_incr保存。具体代码见kernel/sched.c中的函数do\_it\_virt()。

(5) unsigned long it\_prof\_value, it\_prof\_incr;

同样是itimer软件定时。采用jiffies为单位。不管进程在用户态或内核态运行，每个tick使it\_prof\_value减1，减到0时向进程发信号SIGPROF，并重新置初值。初值由it\_prof\_incr保存。 具体代码见kernel/sched.c中的函数do\_it\_prof。

(6) long utime，stime，cutime，cstime，start\_time;

以上分别为进程在用户态的运行时间、进程在内核态的运行时间、所有层次子进程在用户态的运行时间总和、所有层次子进程在核心态的运行时间总和，以及创建该进程的时间。

**6. 信号量数据成员**

(1) struct sem\_undo \*semundo;

进程每操作一次信号量，都生成一个对此次操作的undo操作，它由sem\_undo结构描述。这些属于同一进程的undo操作组成的链表就由semundo属性指示。当进程异常终止时，系统会调用undo操作。sem\_undo的成员semadj指向一个数据数组，表示各次undo的量。结构定义在include/linux/sem.h。

(2) struct sem\_queue \*semsleeping;

每一信号量集合对应一个sem\_queue等待队列(见include/linux/sem.h)。进程因操作该信号量集合而阻塞时，它被挂到semsleeping指示的关于该信号量集合的sem\_queue队列。反过来，semsleeping。sleeper指向该进程的PCB。

**7. 进程上下文环境**

(1) struct desc\_struct \*ldt;

进程关于CPU段式存储管理的局部描述符表的指针，用于仿真WINE Windows的程序。其他情况下取值NULL，进程的ldt就是arch/i386/traps.c定义的default\_ldt。

(2) struct thread\_struct tss;

任务状态段，其内容与INTEL CPU的TSS对应，如各种通用寄存器.CPU调度时，当前运行进程的TSS保存到PCB的tss，新选中进程的tss内容复制到CPU的TSS。结构定义在include/linux/tasks.h中。

(3) unsigned long saved\_kernel\_stack;

为MS-DOS的仿真程序(或叫系统调用vm86)保存的堆栈指针。

(4) unsigned long kernel\_stack\_page;

在内核态运行时，每个进程都有一个内核堆栈，其基地址就保存在kernel\_stack\_page中。

**8. 文件系统数据成员**

(1) struct fs\_struct \*fs;

fs保存了进程本身与VFS的关系消息，其中root指向根目录结点，pwd指向当前目录结点，umask给出新建文件的访问模式(可由系统调用umask更改)，count是Linux保留的属性，如下页图所示。结构定义在include/linux/sched.h中。

(2) struct files\_struct \*files;

files包含了进程当前所打开的文件(struct file \*fd[NR\_OPEN])。在Linux中，一个进程最多只能同时打开NR\_OPEN个文件。而且，前三项分别预先设置为标准输入、标准输出和出错消息输出文件。

(3) int link\_count;

文件链(link)的数目。

Array. 内存数据成员

(1) struct mm\_struct \*mm;

在Linux中，采用按需分页的策略解决进程的内存需求。task\_struct的数据成员mm指向关于存储管理的mm\_struct结构。其中包含了一个虚存队列mmap，指向由若干vm\_area\_struct描述的虚存块。同时，为了加快访问速度，mm中的mmap\_avl维护了一个AVL树。在树中，所有的vm\_area\_struct虚存块均由左指针指向相邻的低虚存块，右指针指向相邻的高虚存块。 结构定义在include/linux/sched.h中。