

## 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);

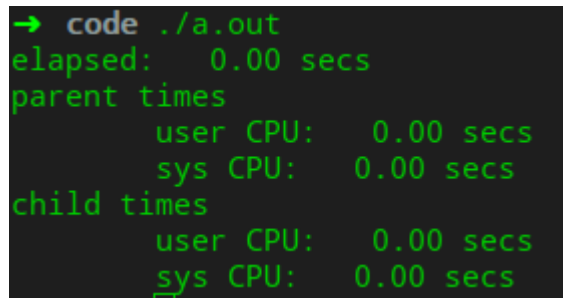
}

```

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：



```

→ code ./a.out
elapsed: 0.00 secs
parent times
    user CPU: 0.00 secs
    sys CPU: 0.00 secs
child times
    user CPU: 0.00 secs
    sys CPU: 0.00 secs

```

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  /* process using time */
2  #include<stdio.h>
3  #include<stdlib.h>
4  #include<sys/times.h>
5  #include<time.h>
6  #include<unistd.h>
7
8  void time_print(char *,clock_t);
9  void calc();
10
11 int main(void) {
12     clock_t start,end;
13     struct tms t_start,t_end;

```

```

14     start = times(&t_start);    //计算起始时间
15
16     calc();
17     // system("grep the /usr/doc/*/* > /dev/null 2>
/dev/null");
18     /* command > /dev/null 是将标准输出重定向到空设备中（丢
弃）
19     command 2> /dev/null 是将标准错误重定向到空设备中（丢
弃） */
20     end=times(&t_end);    //计算结束时间
21
22     time_print("elapsed",end-start);
23     puts("parent times");
24     time_print("\tuser CPU",t_end.tms_utime);
25     time_print("\tsys CPU",t_end.tms_stime);
26
27     puts("child times");
28     time_print("\tuser CPU",t_end.tms_cutime);
29     time_print("\tsys CPU",t_end.tms_cstime);
30
31     exit(EXIT_SUCCESS);
32 }
33
34 void time_print(char *str,clock_t time) {    //输出时间
35     long tps = sysconf(_SC_CLK_TCK);    //获得 CPU 内部时
钟每秒中断个数
36     printf("%s: %6.2f secs\n",str,(float)time/tps);
37 }
38
39 void calc() //计算密集型程序，计算 0-300000 之间的素数并输出
40 {
41     int i = 0, j = 0, sum = 0;
42     for(i = 0; i < 300000; i++)
43     {
44         for(j = 2; j < i; j++)
45         {
46             if(i % j == 0) break;
47         }

```

```

48     if(j == i)
49     {
50         printf("%d ", i);
51         sum++;
52         if(sum % 10 == 0) putchar('\n');
53     }
54 }
55 putchar('\n');
56 }

```

编译并执行代码输出结果如下：

```

myself@myself-PC: /media/myself/Software/操作系统/操作系统实验/操作系
296377 296437 296441 296473 296477 296479 296489 296503 296507 296509
296519 296551 296557 296561 296563 296579 296581 296587 296591 296627
296651 296663 296669 296683 296687 296693 296713 296719 296729 296731
296741 296749 296753 296767 296771 296773 296797 296801 296819 296827
296831 296833 296843 296909 296911 296921 296929 296941 296969 296971
296981 296983 296987 297019 297023 297049 297061 297067 297079 297083
297097 297113 297133 297151 297161 297169 297191 297233 297247 297251
297257 297263 297289 297317 297359 297371 297377 297391 297397 297403
297421 297439 297457 297467 297469 297481 297487 297503 297509 297523
297533 297581 297589 297601 297607 297613 297617 297623 297629 297641
297659 297683 297691 297707 297719 297727 297757 297779 297793 297797
297809 297811 297833 297841 297853 297881 297889 297893 297907 297911
297931 297953 297967 297971 297989 297991 298013 298021 298031 298043
298049 298063 298087 298093 298099 298153 298157 298159 298169 298171
298187 298201 298211 298213 298223 298237 298247 298261 298283 298303
298307 298327 298339 298343 298349 298369 298373 298399 298409 298411
298427 298451 298477 298483 298513 298559 298579 298583 298589 298601
298607 298621 298631 298651 298667 298679 298681 298687 298691 298693
298709 298723 298733 298757 298759 298777 298799 298801 298817 298819
298841 298847 298853 298861 298897 298937 298943 298993 298999 299011
299017 299027 299029 299053 299059 299063 299087 299099 299107 299113
299137 299147 299171 299179 299191 299197 299213 299239 299261 299281
299287 299311 299317 299329 299333 299357 299359 299363 299371 299389
299393 299401 299417 299419 299447 299471 299473 299477 299479 299501
299513 299521 299527 299539 299567 299569 299603 299617 299623 299653
299671 299681 299683 299699 299701 299711 299723 299731 299743 299749
299771 299777 299807 299843 299857 299861 299881 299891 299903 299909
299933 299941 299951 299969 299977 299983 299993
elapsed: 9.33 secs
parent times
  user CPU: 9.27 secs
  sys CPU: 0.04 secs
child times
  user CPU: 0.00 secs
  sys CPU: 0.00 secs
→ code

```

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  /*  usage of kill,signal,wait  */
2  #include <unistd.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include <sys/types.h>
6  #include <wait.h>
7  #include <signal.h>
8
9  int flag;
10 void stop();
11 int main(void)
12 {
13     int pid1, pid2;
14     signal(3, stop);    //设置信号 3 的处理函数为 stop
14     while ((pid1 = fork()) == -1) ; //创建一个子进程
16     if (pid1 > 0)    //当前进程为父进程
17     {
18         while ((pid2 = fork()) == -1) ; //创建另一个子进
程
19         if (pid2 > 0)    //当前进程为父进程
20         {

```

```

21         flag = 1;
22         sleep(5);
23         kill(pid1, 16); //向子进程 1 发送信号 16
24         kill(pid2, 17); //向子进程 2 发送信号 17
25         wait(0);      //暂停当前进程，直到信号来到或子进程
    结束
26         wait(0);
27         printf("\n parent is killed\n");
28         exit(EXIT_SUCCESS);
29     }
30     else    //当前进程为子进程 2
31     {
32         flag = 1;
33         signal(17, stop);    //设置信号 17 的处理函数为
    stop
34         printf("\n child2 is killed by parent\n");
35         exit(EXIT_SUCCESS);
36     }
37 }
38 else    //当前进程为子进程 1
39 {
40     flag = 1;
41     signal(16, stop);
42     printf("\n child1 is killed by parent\n");
43     exit(EXIT_SUCCESS);
44 }
45 }
46
47 void stop()
48 {
49     flag = 0;
50 }

```

执行上述代码的结果如下：

```
→ code gcc 4.c
→ code ./a.out

child1 is killed by parent
child2 is killed by parent
parent is killed
→ code ./a.out

child1 is killed by parent
child2 is killed by parent
^C
→ code □
```

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：

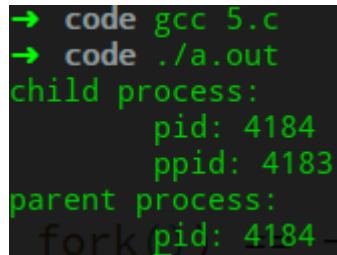
程序代码如下：

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



```
21     return 0;
22 }
```

该程序执行的结果如下：



```
→ code gcc 5.c
→ code ./a.out
child process:
    pid: 4184
    ppid: 4183
parent process:
fork pid: 4184
```

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

程序的执行结果如下：

```
myself@myself-PC: /media/myself/Software/操作系统/操作系统实验四/code
code sudo ./6
[sudo] myself 的密码:
child process priority: 1.
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_hwdep.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_i915.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_regmap.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_register.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hdaudio_ext.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_chmap.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hda_verbs.h
/usr/src/linux-headers-4.15.0-29deepin/include/sound/hdaaudio.h
/usr/src/linux-headers-4.15.0-29deepin/sound/hda
/usr/src/linux-headers-4.15.0-29deepin/sound/pci/hda
/usr/src/linux-headers-4.15.0-29deepin-generic/include/config/snd/soc/hdac
/usr/src/linux-headers-4.15.0-29deepin-generic/include/config/snd/hda
/usr/src/linux-headers-4.15.0-29deepin-generic/include/config/snd/hda.h
/usr/src/linux-headers-4.15.0-29deepin-generic/include/config/sensors/hdaps.h
/usr/share/alsa/init/hda
find: '/proc/5762/task/5762/net': 无效的参数
find: '/proc/5762/net': 无效的参数 /resource.h
find: '/proc/5768/task/5768/net': 无效的参数
find: '/proc/5768/net': 无效的参数
find: '/proc/5771/task/5771/net': 无效的参数
find: '/proc/5771/net': 无效的参数
find: '/tmp/.mount_jetbraKv5hno': 权限不够
/sys/kernel/debug/tracing/events/hda_intel
/sys/kernel/debug/tracing/events/hda_controller
/sys/kernel/debug/tracing/events/hda
/sys/kernel/debug/tracing/events/hda/hda_send_cmd(id,15);
/sys/kernel/debug/tracing/events/hda/hda_get_response
/sys/kernel/debug/tracing/events/hda/hda_unsol_event
/sys/kernel/debug/regmap/hdaudioCOD0-hdaudio
/sys/devices/pci0000:00:0000:00:1f.3/hdaudioCOD0 / -name hda* 的功能
/sys/bus/hdaudio
/sys/bus/hdaudio/devices/hdaudioCOD0
/sys/bus/hdaudio/drivers/snd_hda_codec_realtek/hdaudioCOD0
/sys/module/snd_hda_codec_generic/drivers/hdaudio:snd_hda_codec_generic
/sys/module/snd_hda_codec_realtek/drivers/hdaudio:snd_hda_codec_realtek
/lib/modules/4.15.0-29deepin-generic/kernel/sound/hda
/lib/modules/4.15.0-29deepin-generic/kernel/sound/pci/hda
/lib/modules/4.15.0-29deepin-generic/kernel/drivers/platform/x86/hdaps.ko
/media/myself/Windows 10/Windows/INF/hdaudbus.inf
/media/myself/Windows 10/Windows/INF/hdaudbus.PNF
/media/myself/Windows 10/Windows/INF/hdaudio.inf
/media/myself/Windows 10/Windows/INF/hdaudio.PNF
/media/myself/Windows 10/Windows/INF/hdaudss.inf
/media/myself/Windows 10/Windows/INF/hdaudss.PNF
/media/myself/Windows 10/Windows/System32/drivers/hdaudbus.sys
/media/myself/Windows 10/Windows/System32/DriverStore/en-US/hdaudbus.inf_loc
/media/myself/Windows 10/Windows/System32/DriverStore/en-US/hdaudio.inf_loc
/media/myself/Windows 10/Windows/System32/DriverStore/en-US/hdaudss.inf_loc
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudbus.inf_amd64_44e1eba3f18ff604
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudbus.inf_amd64_44e1eba3f18ff604/hdaudbus.i
nf
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudbus.inf_amd64_44e1eba3f18ff604/hdaudbus.P
NF
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudbus.inf_amd64_44e1eba3f18ff604/hdaudbus.s
ys
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudio.inf_amd64_a66e777ae88134f8
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudio.inf_amd64_a66e777ae88134f8/hdaudio.inf
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudio.inf_amd64_a66e777ae88134f8/hdaudio.PNF
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudss.inf_amd64_fdec96c9e23f112
/media/myself/Windows 10/Windows/System32/DriverStore/FileRepository/hdaudss.inf_amd64_fdec96c9e23f112/hdaudss.inf
/media/myself/Windows 10/Windows/System32/DriverStore/zh-CN/hdaudbus.inf_loc
/media/myself/Windows 10/Windows/System32/DriverStore/zh-CN/hdaudss.inf_loc
/media/myself/Windows 10/Windows/System32/DriverStore/zh-CN/hdaudio.inf_loc
/media/myself/Windows 10/Windows/WinSxS/amd64_dual_hdaudio.inf_31bf3856ad364e35_10.0.17134.319_none_899e425a384e0e9
0/hdaudio.inf
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudbus.inf.resources_31bf3856ad364e35_10.0.17134.1_en-us_c4a50bcafa
823809/hdaudbus.inf_loc
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudio.inf.resources_31bf3856ad364e35_10.0.17134.1_en-us_c71f9477954
2c0b7/hdaudio.inf_loc
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudss.inf.resources_31bf3856ad364e35_10.0.17134.1_en-us_83339942ebc
60b2d/hdaudss.inf_loc
/media/myself/Windows 10/Windows/WinSxS/amd64_dual_hdaudbus.inf_31bf3856ad364e35_10.0.17134.1_none_24d62d6acbf87ec4
/hdaudbus.inf
/media/myself/Windows 10/Windows/WinSxS/amd64_dual_hdaudbus.inf_31bf3856ad364e35_10.0.17134.1_none_24d62d6acbf87ec4
/hdaudbus.sys
/usr/src/linux-headers-3.2.0-24-generic-pae/include/config/sensors/hdaps.h
/media/myself/Windows 10/Windows/WinSxS/amd64_dual_hdaudio.inf_31bf3856ad364e35_10.0.17134.1_none_8d500b264b81a7d8/
hdaudio.inf
/usr/src/linux-headers-3.2.0-23-generic-pae/include/config/sensors/hdaps.h
/media/myself/Windows 10/Windows/WinSxS/amd64_dual_hdaudss.inf_31bf3856ad364e35_10.0.17134.1_none_55a0a583db595c92/
hdaudss.inf
/usr/src/linux-headers-3.2.0-24-generic-pae/include/sound/hda_hwdep.h
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudss.inf.resources_31bf3856ad364e35_10.0.17134.1_zh-cn_242b7638e1
a4121b/hdaudbus.inf_loc
/usr/src/linux-headers-3.2.0-54-generic-pae/include/config/sensors/hdaps.h
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudio.inf.resources_31bf3856ad364e35_10.0.17134.1_zh-cn_26a5fee57c6
49ac9/hdaudio.inf_loc
/usr/src/linux-headers-3.2.0-23-sound/pci/hda
/media/myself/Windows 10/Windows/WinSxS/amd64_hdaudss.inf.resources_31bf3856ad364e35_10.0.17134.1_zh-cn_e2ba03b0d2e
7e53f/hdaudss.inf_loc
/usr/src/linux-headers-3.2.0-54-generic-pae/include/sound/hda_hwdep.h
/media/myself/Software/Program Files/DriverGenius/data/hdads.dat
/media/myself/Software/Program Files/MATLAB/R2018a/help/opc/ug/hda-intersect.png
/media/myself/Software/Program Files/MATLAB/R2018a/help/opc/ug/hda-resample.png
/media/myself/Software/Program Files/MATLAB/R2018a/help/opc/ug/hda-tree.png
/media/myself/Software/Program Files/MATLAB/R2018a/help/opc/ug/hda-union.png
/media/myself/Software/Program Files/MATLAB/R2018a/resources/opc/en/hda
/media/myself/Software/Program Files/MATLAB/R2018a/toolbox/opc/opc+opc/hdaQualityString.m
/media/myself/Software/Program Files/MATLAB/R2018a/toolbox/opc/opc+opc/hdaSupport.m
find: '/run/user/1000/doc': 权限不够
find: '/run/user/1000/gvfs': 权限不够
child process terminated.
```

## 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_setpriority` 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。
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 prev\_task 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\_cptra, \*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_incr;`

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。