操作系统　2023总复习

同济大学计算机系

一、进程切换

```
int ProcessManager::Swtch()
{

 1:    User& u = Kernel::Instance().GetUser();
 2:    SaveU(u.u_rsav);     // 现场保护

 3:    Process* procZero = &process[0];
 4:    X86Assembly::CLI();
 5:    SwtchUStruct(procZero);  // 核心页表第1023项指向0#进程PPDA区
 6:    RetU();          // u_rsav中取出ESP和EBP的值，恢复0#进程的Swtch()栈帧
 7:    X86Assembly::STI();

 8:    Process* selected = Select();  // 选中内存中优先级最高的就绪进程（SRUN），selected

 9:    X86Assembly::CLI();
 10:   SwtchUStruct(selected);// 恢复selected进程PPDA区的地址映射关系，让CPU可以访问核心栈和user结构
 11:   RetU();          // 用u_rsav恢复selected进程的Swtch()栈帧
 12:    X86Assembly::STI();
 13:   User& newu = Kernel::Instance().GetUser();
    // 用相对表刷系统用户页表，恢复selected进程用户空间的地址映射关系
 14:   newu.u_MemoryDescriptor.MapToPageTable();
    ……
    return 1;
}
```

MapToPageTable( )，用进程的相对表写系统用户页表

系统用户页表

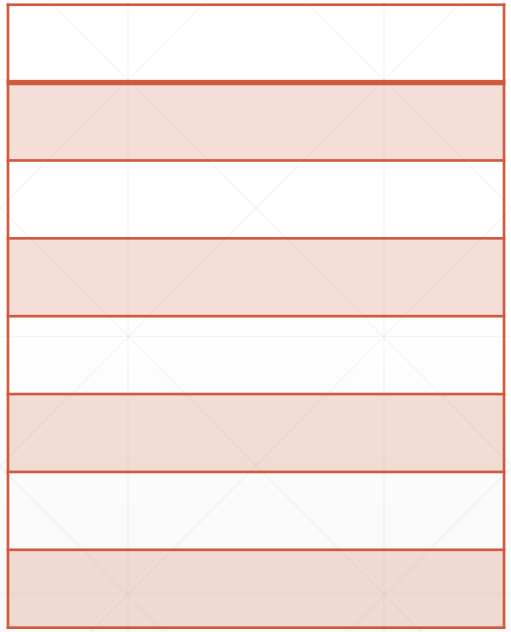
```
pUserPageTable
Machine::Instance().GetUserPageTableArray()
```

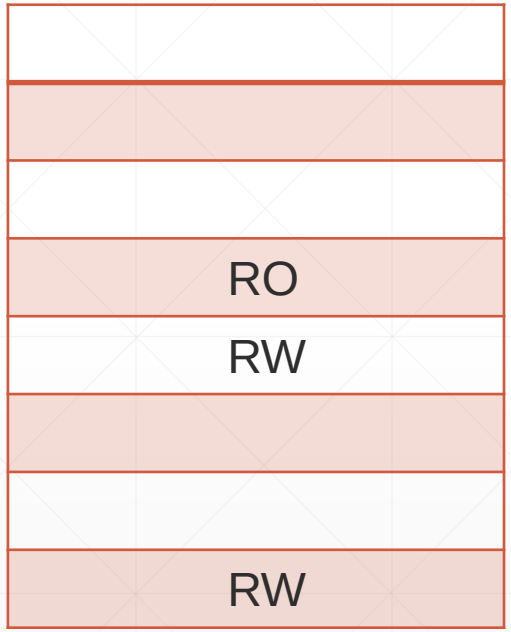
进程的相对表

```
u.u_MemoryDescriptor.m_UserPageTableArray
```

1、清 0 系统用户页表（ P=0 ）
2、遍历相对表，对所有 P==1的表项，写系统页表
- RO 表项，base + 代码段起始页框号
- RW 表项，base + 可交换部分起始页框号

3、写 0# PTE：0 U RW P






```
class MemoryDescriptor
{
        PageTable*      m_UserPageTableArray;

        unsigned long   m_TextStartAddress;
        unsigned long   m_TextSize;
        unsigned long   m_DataStartAddress;
        unsigned long   m_DataSize;
        unsigned long   m_StackSize;
}
```

CPU空转（idle状态）

现运行进程放弃CPU后，如果内存中不存在就绪进程，系统进入idle状态。idle的时候，系统执行hlt指令，现运行进程是高优先权入睡的0#进程，Curpri是下台进程的优先数（>=100）。系统在等中断。

- 来了一个外设中断，中断处理程序会唤醒一个等待IO操作完成的睡眠进程。
  - 如果，被唤醒的是内存中的睡眠进程PA，0#进程把CPU让给PA。
  - 如果，被唤醒的是盘交换区上的睡眠进程PB。会把0#进程也叫起来。0#进程优先级高，拿回CPU，执行sched函数把PB搬回内存（假设内存有空，可以容纳PB进程的图像）。之后入睡放弃CPU，系统选中PB，恢复运行。
- 来了一个时钟中断（非tout时刻）。0#进程就执行时钟中断处理程序，非整数秒累加lbolt，整数秒调整time变量。保持时钟在走。。。之后，继续idle。因为，仍然没有就绪进程，对吧。就这样，等啊等，直到出现第一个可以唤醒个睡眠进程的中断请求。。。系统脱离idle状态，终于开始干活啦。

咱们的PC机，CPU利用率好低，对吧。绝大多数时间，所有CPU都idle，系统里根本就没有就绪进程，office进程全在睡觉。。。

二、进程调度

- 现运行进程放弃CPU，系统选中、执行新运行进程

- Unix V6++核心态不调度，有3个调度点：
  - 入睡
  - 终止

  主动放弃CPU
  - 从核心态返回用户态前夕
    - 系统调用返回
    - 整数秒时钟中断

  被剥夺（被抢占）

主动放弃CPU

```
Sleep( )
{
    ……
    Swtch( );
    ……
}
```

```
Exit( )
{
    ……
    Swtch( );
}
```

被剥夺（被抢占）

例行调度

```
void Time::TimeInterruptEntrance()
{
        SaveContext();
        SwitchToKernel();

        CallHandler(Time, Clock);

        RestoreContext();
        Leave();
        InterruptReturn();
}
```

```
struct pt_context *context;
__asm__ __volatile__ ("        movl %%ebp, %0; addl $0x4, %0 " : "+m" (context) );

if( context->xcs & USER_MODE ) /*先前为用户态*/
{
    while(true)
    {
        X86Assembly::CLI();  // 关中断
        if(Kernel::Instance().GetProcessManager().RunRun > 0)
        {
            X86Assembly::STI();        // 开中断
            Kernel::Instance().GetProcessManager().Swtch();
        }
        else
            break;  // runrun为0，退栈回用户态继续执行应用程序
    }
}
```

void SystemCall::SystemCallEntrance()

void DiskInterrupt::DiskInterruptEntrance()

动态优先权调度算法

- 进程优先数 p_pri，选最小的

- 进程优先数的确定
  - 入睡进程，设置系统调用的入睡优先数
    - 唤醒后，维持不变
  - 被剥夺进程，计算应用程序优先数　SetPri( )
    - p-pri = min { 127 , 进程的静态优先数 + （p_cpu/16）}　// 255
    - 进程的静态优先数 = PUSER(100) + p_nice。

序号	名称	优先数值
1	PSWP	-100
2	PINOD	-90
3	PRIBIO	-50
4	EXPRI	-1
5	PPIPE	1
6	TTIPRI	10
7	TTOPRI	20
8	PWAIT	40
9	PSLEP	90
10	PUSER	100

三、睡眠、唤醒操作

进程入睡，基本工作过程

chan 睡眠原因， pri 是优先数

```
void Process::Sleep(unsigned long chan, int pri)
{
    ……
    if ( pri > 0 )
    {     ……
            X86Assembly::CLI();
            this->p_wchan = chan;
            this->p_stat = Process::SWAIT;
            this->p_pri = pri;
            X86Assembly::STI();
      ……Kernel::Instance().GetProcessManager().Swtch();
    }
```

1、根据优先数判断进程需要进入高优先权睡眠状态还是低优先权睡眠状态
2、设置睡眠原因
3、修改调度状态
4、设置优先数
5、放弃CPU

备注：进入低优先权睡眠状态之前要做额外处理

```
else
{
        X86Assembly::CLI();
        this->p_wchan = chan;
        this->p_stat = Process::SSLEEP;
        this->p_pri = pri;
        X86Assembly::STI();
        Kernel::Instance().GetProcessManager().Swtch();
}
```


进程唤醒，基本工作过程

- **ProcessManager::WakeUpAll(chan) 函数唤醒所有因 p_wchan == chan 而入睡的进程。**

```
void ProcessManager::WakeUpAll(unsigned long chan)
{
    for(int i = 0; i < ProcessManager::NPROC; i++)
    {
        if( this->process[i].IsSleepOn(chan) )  // process[i]. p_wchan == chan ?
        {
            this->process[i].SetRun();
        }
    }
}
```

- Process::SetRun( )，恢复进程的就绪状态。

```
void Process::SetRun()
{
    ProcessManager& procMgr = Kernel::Instance().GetProcessManager();

    /* this是被唤醒进程的Process对象（结构） */
    this->p_wchan = 0;          // 清除睡眠原因
    this->p_stat = Process::SRUN;    // 变就绪
    if ( this->p_pri < procMgr.CurPri )
    {
        procMgr.RunRun++;
    }
        ……
}
```

盘交换区的使用

- RunIn==1。盘交换区上有就绪进程，内存没有足够空间容纳其进程图像。
- RunOut==1。盘交换区上无就绪进程。

- 负责进程图像搬迁任务的是0#进程。平时，0#进程SSLEEP。
  - Sleep(&RunOut，-100)，等待盘交换区出现就绪进程，执行换入操作

  或，Sleep(&RunIn，-100)，等待内存出现可供利用的空间，执行换出操作

- 需要执行换入、换出操作时，系统唤醒0#进程
  - 换出操作：procMgr.WakeUpAll((unsigned long)&procMgr.RunIn)
  - 换入操作：procMgr.WakeUpAll((unsigned long)&procMgr.RunOut);

睡眠、唤醒，完整的过程

```
void Process::Sleep(unsigned long chan, int pri)
{
    ……
    if ( pri > 0 )
    {     ……  // 省略了信号处理
        X86Assembly::CLI();
        this->p_wchan = chan;
        this->p_stat = Process::SWAIT;
        this->p_pri = pri;
        X86Assembly::STI();
        if ( procMgr.RunIn != 0 )
        {
            procMgr.RunIn = 0;
            procMgr.WakeUpAll((unsigned long)&procMgr.RunIn);
        }
        Kernel::Instance().GetProcessManager().Swtch();
    }
```

```
22 void Process::SetRun()
23 {
24     ProcessManager& procMgr = Kernel::Instance().GetProcessManager();
25
26     /* 清除睡眠原因，转为就绪状态 */
27     this->p_wchan = 0;
28     this->p_stat = Process::SRUN;
29     if ( this->p_pri < procMgr.CurPri )
30     {
31         procMgr.RunRun++;
32     }
33     if ( 0 != procMgr.RunOut && (this->p_flag & Process::SLOAD) == 0 )
34     {
35         procMgr.RunOut = 0;
36         procMgr.WakeUpAll((unsigned long)&procMgr.RunOut);
37     }
38 }
```

```
    else
    {
        X86Assembly::CLI();
        this->p_wchan = chan;
        this->p_stat = Process::SSLEEP;
        this->p_pri = pri;
        X86Assembly::STI();
        Kernel::Instance().GetProcessManager().Swtch();
    }
```

情景分析题

- 现运行进程PA执行系统调用sleep(100)。系统状态：RunIn非0，盘交换区上有一个SRUN进程PB。
  - 现运行进程→SWAIT，RunIn非0。内存出现可供利用的空间，RunIn清0，WakeUpAll(&RunIn) 唤醒 0#进程。
  - Swtch放弃CPU后，0#进程成为新选中进程，执行Sched函数，换出PA（分配盘交换区空间，写磁盘），换入PB（分配内存，读磁盘）。
  - 完成后，0#进程，RunOut++， Sleep(&RunOut)，放弃CPU后，PB进程成为新选中进程，恢复运行。

- 50s后，PB进程终止

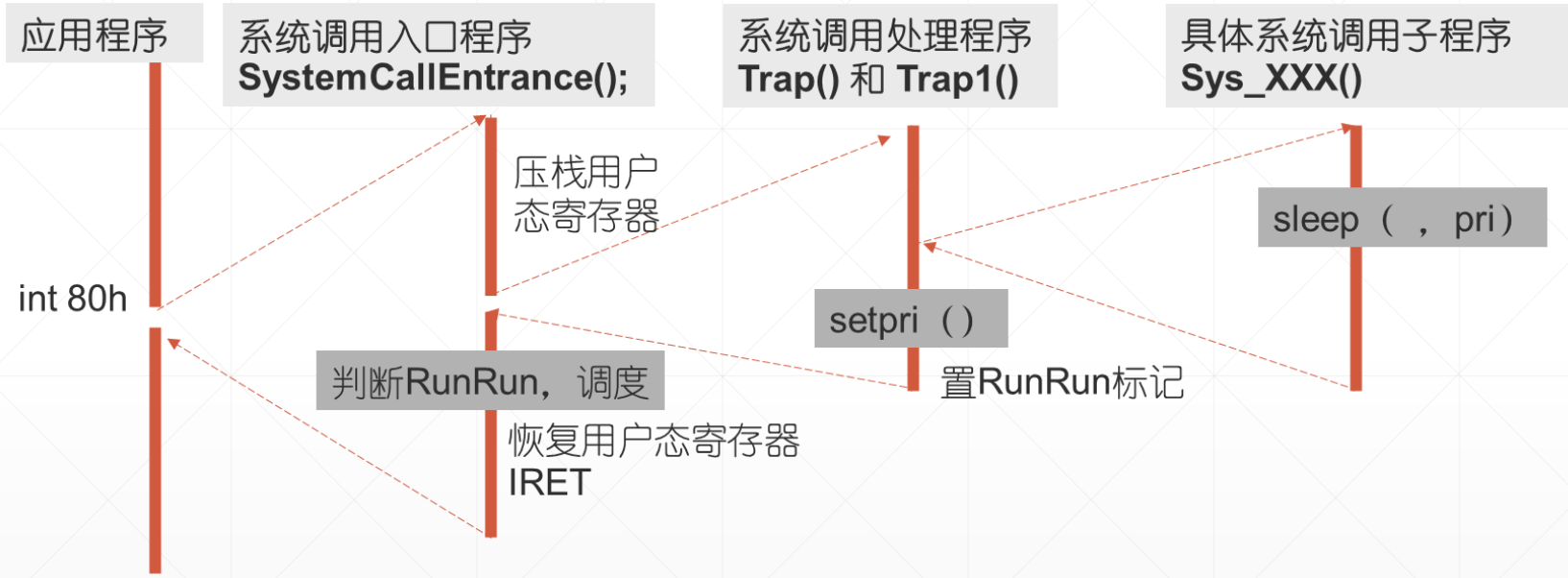
情景分析题，又过了50s

- 定时器到，时钟中断处理程序唤醒盘交换区上的 PA进程。SLOAD是0，RunOut是1，所以，SetRun( )会清0 RunOut，WakeUpAll(&Runout) 唤醒 0#进程。
- 时钟中断返回后，0#进程上台执行，将PA进程的图像搬入内存。
- RunOut++，Sleep(&RunOut)，放弃CPU后，PA成为新选中进程，恢复运行。

四、系统调用

- 快系统调用访问磁盘，会导致进程入睡。SSLEEP，高优先权睡眠。
- 慢系统调用，会导致进程入睡，但不会访问磁盘。SWAIT，低优先权睡眠。执行慢系统调用的进程有可能会睡好久好久，比如sleep（36500s）要睡10个小时。所以，
  - 入睡前要做特殊处理（1）与盘交换区上的就绪进程对换（2）有等待处理的信号，不睡。
  - 睡眠期间，收到信号，提前唤醒，尽快进行信号处理。
- 不会入睡的系统调用 getpid( )

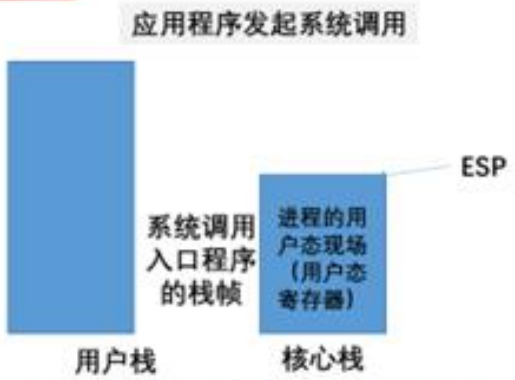
- 系统调用传参，准备返回值
- 系统调用的执行过程
- 添加系统调用



应用程序发起系统调用和系统调用返回继续执行应用程序。

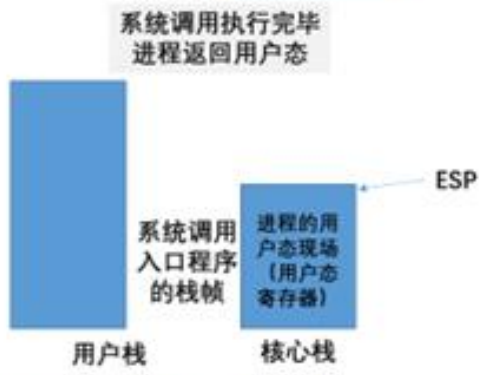
1000

1、第1次现场保护：PA执行系统调用入口函数。会把用户态寄存器保存在自己的核心栈



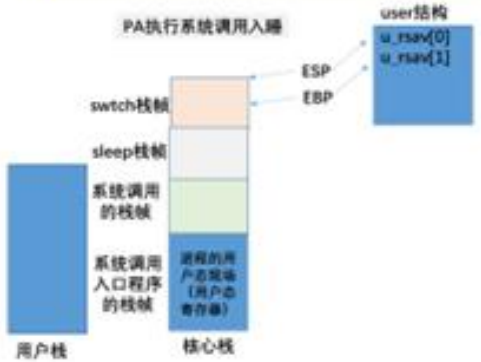
1365

4、恢复现场，第2步、恢复用户态现场：系统调用执行完毕后，从系统调用入口函数的栈帧恢复用户态现场。PA从系统调用的钩子函数返回，继续执行应用程序。（若系统调用返回前重算优先级，PA不再是优先级最高的进程，它会放弃CPU，下次调度上台时，恢复用户态现场）

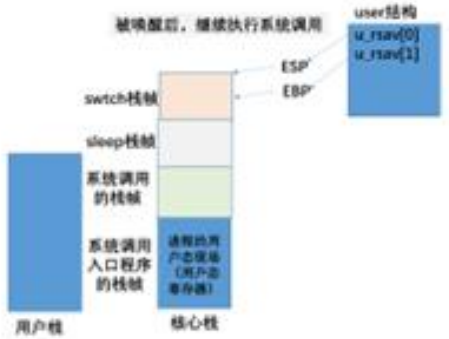


1000s PA进程执行系统调用sleep（365）

2、第2次现场保护：PA入睡时执行swtch（）放弃CPU & 将swtch栈帧定位指针ESP、EBP存入自己的user结构。PA成为睡眠态进程



3、1365s恢复现场，第1步恢复核心态现场：被中断的现运行进程唤醒PA、放弃CPU后，swtch函数选中PA，从它的user结构中取出1000s时保护的swtch栈顶定位指针，并赋值ESP和EBP。EIP会引导PA继续执行系统调用。



五、时钟中断 和 时间片轮转调度

- 应用程序如何时间片轮转轮流使用CPU。系统如何保存恢复应用程序的执行状态。
- 每次时钟中断处理程序，现运行进程p_cpu++。连续运行很久的现运行进程，p_cpu变大很多。
- 每秒结束的时候
  - 所有进程 p_cpu-=SCHMAG（20）。好久没运行的进程， p_cpu会减小。
  - 重算所有应用程序的优先数。运行很久的现运行进程，p_pri变大，优先权下降。系统调度状态发生改变，RunRun++，现运行进程把CPU让给排队时间最长的那个应用程序。

四、Unix V6++系统中存在 3 个 CPU bound 用户态进程 PA、PB 和 PC。3 个进程静态优先数相等：100，p_cpu 是 0。Process[8]、[5]、[9]分别是 PA 、PB、PC 进程的 PCB。T 时刻是整数秒，PA 先运行。

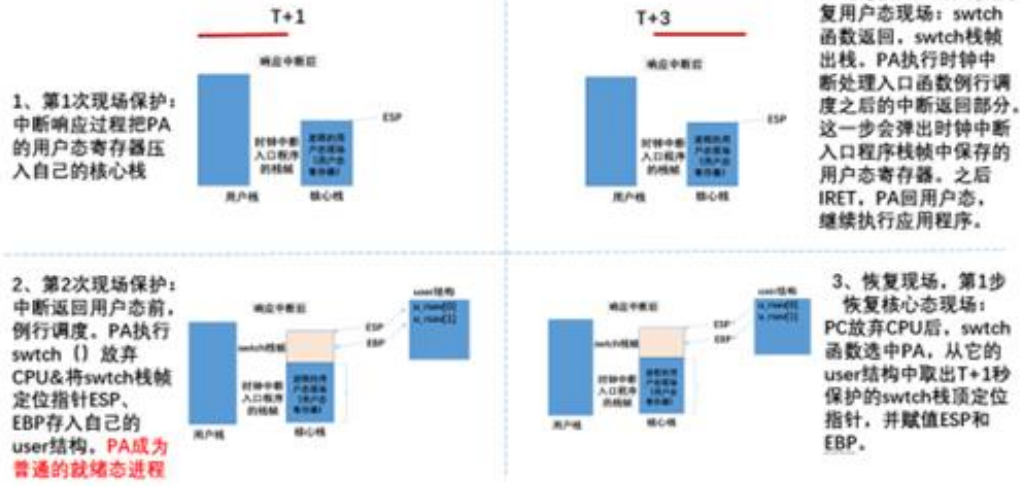
1、 画进程运行时序图。

2、 T+1 时刻，PA 进程用完时间片放弃 CPU。何时，PA 进程会再次得到运行机会？简述 T+1 时刻系统怎样保护 PA 进程的用户态 CPU 执行现场，下次再运行时系统如何恢复 PA 进程的用户态 CPU 执行现场。

答：

1、

2、T+3 时刻 PA 进程再次得到运行机会。T+1 时刻系统保护 PA 进程的用户态 CPU 执行现场，T+3 时刻恢复 PA 进程的用户态 CPU 执行现场的具体过程图示如下：



维护时钟

- lbolt 和 time

六、进程创建 fork 系统调用

- 除0#进程外，所有进程都是fork出来的
- fork系统调用创建子进程
  - 为子进程分配一个Process结构和一个与所有进程均不相同的pid，复制父进程Process结构中的绝大部分字段
  - 为子进程分配相对表，复制父进程的相对表
  - 子进程共享父进程的共享正文段
  - 为子进程的可交换部分分配内存空间，复制父进程的可交换部分
    - 成功，子进程图像在内存
    - 不成功，为子进程可交换部分分配盘交换区空间，启动IO，复制父进程图像
  - 父进程返回，返回值是子进程的pid#

```
int a=0;
main( )
{
    int i ;

    while( ( i=fork( ) ) == -1 );
    if(i)
    {
        a = a+1;
            printf("parent : a = %d\n", &a);
    }
    else
    {
        a=a+4;
        printf("child : a = %d\n", &a);
    }
}
```

- 新建的子进程 SRUN
  - 内存（SLOAD）。立即接受调度。被选中时，Swtch( )从其user结构：u_rsav 中获得栈帧位置（Newproc栈帧），子进程退栈、行fork系统调用返回逻辑，回用户态，返回值是0。
  - 盘交换区（~SLOAD）。p_flag带SSWAP标记。待内存有空，0#进程为可交换部分分配内存空间，将其复制回内存。随后，子进程接受调度，被Swtch选中，SSWAP标记指示Swtch函数从u_ssav 中获得栈帧位置（Newproc栈帧）…
- 子进程首次执行，fork返回

Fork函数 和 Newproc函数，　　Swtch( )

七、exec系统调用

exec的使用方法

```
#include <stdio.h>
#include <stdlib.h>

int main1( )    // tryExec.c
{
   char* argv[4];
   argv[0] = "showCmdParam";
   argv[1] = "arg1";
   argv[2] = "arg2";
   argv[3] = 0;

   if ( fork( ) !=0 )
          ;
   else
      execv(argv[0] , argv) ;

   exit(0);
}
```

第一步：父进程为应用程序准备命令行参数。
注意，第一个参数是新程序的文件名；最后一个参数是0，表示命令行参数结束。

接着，执行fork( )系统调用，创建子进程。

新建的子进程刚开始时和父进程一样执行 tryExec 程序，是 tryExec 进程。
接下来，它执行**exec系统调用刷新用户空间，装入新程序的图像**。exec 系统调用的第1个参数是新程序的文件名，第二个参数是新程序的命令行参数。
exec系统调用返回后，子进程回用户态，从main函数的第一条指令开始执行新程序。

exec的使用方法

```
#include <stdio.h>
#include <stdlib.h>

int main1(int argc, char *argv[])   // showCmdParam.c
{
   int i;

   printf("The command parameter of showCmdParam\n");

   for(i = 0; i < argc; i++)
      printf("argv[%d]:\t%s\n", i, argv[i]);

   exit(0);
}
```

应用程序，main函数有2个入口参数，argc是命令行参数的数量，argv是命令行参数。
本例，新程序输出命令行参数。

命令行参数是exec加载的目标应用程序main函数的参数

Part 1、Unix 系统加载应用程序

创建一个子进程，让它承担执行应用程序的任务

```
main( )
{
  ......
  int i ;
  if( i = fork( ))
       ......
  else
       exec ( "gcc" , arg1, arg2.....);
}
```

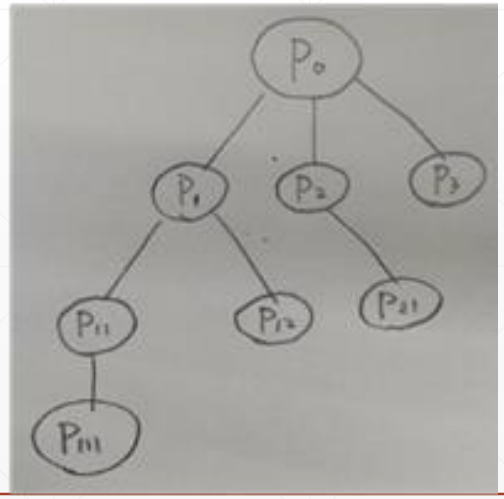
子进程要执行的程序　　程序的命令行参数

shell进程加载应用程序的方法：创建一个子进程，让它执行exec系统调用，从 main( )函数的第一条指令开始执行。

这个进程承担执行目标应用程序的任务，应用程序执行完毕，进程终止。进程的用户空间，是目标应用程序的代码和数据。

- 如果是串行程序，负责执行目标应用程序的，只有这一个进程。
- 如果是并行程序，这个进程会执行fork系统调用创建子进程，或是执行pthread创建线程。负责执行目标应用程序的，是一颗进程树。所有进程，用户空间是同一个程序。

```
L1:   #include <stdio.h>
L2:   void main(void)
L3:   {   int i;
L4:       printf ("%d  %d \n",  getpid ( ),  getppid ( ) );
L5:       for (i = 0; i < 3; ++i)
L6:           if ( fork( ) == 0 )
L7:               printf ("%d  %d \n",  getpid ( ),  getppid ( ) );
L8:   }
```



没有 0#进程和 1#进程

八、exit 和 wait

- exit系统调用的功能和需要执行的主要操作。
- wait系统调用的功能和需要执行的主要操作。
- 回收终止子进程PCB的时刻。
- 孤儿进程，是父进程已经终止的进程。
- 僵尸进程，是自己已经终止的进程。

```
#include   <stdio.h>
#include <sys.h>
main( )
{
    int i=10, j =20;
    if( i=fork( ) )
    {
        printf("It is parent process. PID = %d, First Son: %d\n", getpid( ), i);
        if( i=fork( ) ) {
            printf("It is parent process. PID = %d, Second Son: %d\n", getpid( ), i);
            i = wait( &j );
            printf("Exit Son: %d. Exit Status= %d\n", i, j);
        }
        else {
            printf("It is child pr
            exit( 2 );
        }
    }
    else
    {
        sleep(100);
        printf("It is child process. PID = %d, i = %d\n", getpid( ), i);
        exit(1);
    }
}
```

我们统一一下，这里不移位了，终止码是2