操作系统　第四章　进程管理

4.1 Unix V6++进程管理的实现机制(上)

同济大学计算机系

数据结构 1　ProcessManager

```
/*  src/Kernel/Kernel.cpp  */
ProcessManager g_ProcessManager;

class ProcessManager
{
        ……
 public:
        Process process[NPROC];
        Text text[NTEXT];

        int CurPri;
        int RunRun;
        int RunIn;
        int RunOut;
        int ExeCnt;
 private:
        static unsigned int m_NextUniquePid      /* 分配给下一个新建进程的PID号 */

        ……
}
```

Unix V6++有一个 ProcessManager 对象 g_ProcessManager;，负责管理系统中的所有进程。

- process 数组登记所有进程的 Process 结构。process[0]属于 0#进程。
- text 数组管理正在执行的所有应用程序的代码段。
- 字段 CurPri、RunRun、RunIn、RunOut 是进程管理子系统的重要状态变量。
  - CurPri 是现运行进程的优先数，
  - RunRun 是剥夺标识，非 0 值表示现运行进程不再是优先级最高的进程，需要让出 CPU，
  - RunOut，盘交换区无就绪进程，
  - RunIn，内存紧张，无法容纳盘交换区上的就绪进程。

p_stat字段

```
class Process
{
    ......
    ProcessState   p_stat; /* 进程调度状态 */
    int p_flag;        /* 进程标识位图 */

    int p_pri;         /* 进程优先数 */
    int p_cpu;         /*  cpu值，用来衡量进程CPU使用时长 */
    int p_nice;        /* 进程静态优先数的偏置值 */

    int p_time;        /* 用来衡量进程 盘上驻留时间 或 内存驻留时间 */

    unsigned longp_wchan; /* 进程睡眠原因，一般是某个内核变量的地址 */
    ......
}
```

p_stat：进程调度状态。逻辑层面刻画进程的处理器使用资格。p_stat 是 ProcessState 类型的枚举量，有 7 个不同的取值，代码*.*。

（1）SNULL。　用来标识 Process 结构空闲、未分配。创建新进程的时候，需要为它寻找、分配一个 p_stat == SNULL 的空闲 Process 结构。

（2）SSLEEP。　进程因访问磁盘而阻塞，高优先权睡眠状态。这种进程正在执行的系统调用需要执行磁盘 IO 操作，比如打开一个磁盘文件（open 系统调用），或者 读取磁盘文件数据（read 系统调用）。。。在磁盘数据读入内存之前，进程没有新数据可供处理，没资格使用 CPU。这种状态是高优先权睡眠状态。

（3）SWAIT。　低优先权睡眠状态的进程阻塞，但阻塞原因不是因为访问磁盘。导致进程进入低优先权睡眠状态的原因有很多，典型原因包括，读键盘输入，读写网络套接字 socket，读写 pipe 文件，执行 P 操作，执行 sleep(n)系统调用睡 ns。。。和 SSLEEP 进程一样，SWAIT 进程也没资格使用 CPU 的。

（4）SRUN。　非阻塞。包括就绪状态的进程 和 运行状态的进程。在安全的调度点，系统会在 p_stat==SRUN 的进程集合中，挑选优先级最高的进程，分配给它 CPU。

（5）SIDL。　新建。fork 系统调用创建新进程期间，Process 结构写好了，但进程图像尚未写完整。这种进程没资格使用 CPU，给它设置一个特殊的状态：SIDL。

（6）SZOMB。　终止。exit 系统调用释放进程图像，但不回收终止进程的 Process 结构。SZOMB 描述终止进程，Process 结构回收之前的这种状态。SZOMB 状态的进程也没资格使用 CPU。

（7）SSTOP。　被跟踪。被 debug 的进程这种状态。Unix V6++系统，debug 功能尚未完

p_flag字段

```
class Process
{
    ......
    ProcessState    p_stat; /* 进程调度状态 */
    int p_flag;         /* 进程标识位图 */

    int p_pri;              /* 进程优先数 */
    int p_cpu;              /*   cpu值，用来衡量进程CPU使用时长 */
    int p_nice;         /* 进程静态优先数的偏置值 */

    int p_time;         /* 用来衡量进程 盘上驻留时间 或 内存驻留时间 */

    unsigned longp_wchan; /* 进程睡眠原因，一般是某个内核变量的地址 */
    ......
}
```

```
enum ProcessFlag    /* 进程标志位（用于进程图象换进换出）*/
{
    SLOAD   = 0x1,  /* 进程图象在内存中*/
    SSYS    = 0x2,  /* 系统进程图象，不允许被换出*/
    SLOCK   = 0x4,  /* 含有该标志的进程图象暂不允许换出*/
    SSWAP   = 0x8,  /* 该进程被创建时图象就在交换区上*/
    STRC    = 0x10, /* 父子进程跟踪标志，UNIX V6++未使用到*/
    STWED   = 0x20  /* 父子进程跟踪标志，UNIX V6++未使用到*/
};
```

p_flag：进程标识，是一个 32 位的状态位图。状态位图中，所有 bit 彼此独立，每个 bit 刻画进程管理的一个维度。Unix V6++定义 6 个 bit，具体语义如下：

（1）SLOAD，描述进程图像的位置。1，进程图像在内存；0，进程图像在盘交换区。

（2）SSYS，系统进程标识。1，系统进程；0，其它的普通进程。系统进程和其它进程，最重要的区别在于前者没有用户空间，不会返回用户态执行应用程序；这些进程，图像只包括 Process 结构和 PPDA 区，换言之，只有 PCB 和核心栈。还有一个区别，系统进程不会被换出，它们常驻内存；普通进程，可交换部分，甚至代码段都可能被换出至盘交换区。Unix V6++系统，只有 0#进程这么 1 个系统进程，它是负责进程管理的服务器进程，执行 Sched( )函数，永不终止。

（3）SLOCK。为 1 时段，进程图像锁定物理内存，不可以换出至盘交换区。父进程创建子进程的时候，会用到这个标识。

（4）SSWAP。Unix V6 将进程切换使用的栈帧位置保存在 user 结构中。SSWAP 为 1 时，在 u_ssav 字段。为 0 时，保存在 u_rsav。

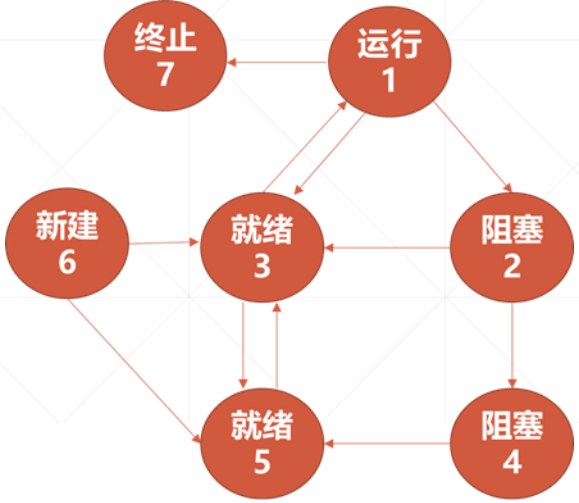
p_flag字段的位操作

换出进程process[i]之后，需要将SLOAD清0：process[i].p_flag &= ~SLOAD;
换入进程process[i]之后，需要将SLOAD置1：process[i] .p_flag |= SLOAD;

```
enum ProcessFlag    /* 进程标志位（用于进程图象换进换出）*/
{
    SLOAD   = 0x1,  /* 进程图象在内存中*/
    SSYS    = 0x2,  /* 系统进程图象，不允许被换出*/
    SLOCK   = 0x4,  /* 含有该标志的进程图象暂不允许换出*/
    SSWAP   = 0x8,  /* 该进程被创建时图象就在交换区上*/
    STRC    = 0x10, /* 父子进程跟踪标志，UNIX V6++未使用到*/
    STWED   = 0x20  /* 父子进程跟踪标志，UNIX V6++未使用到*/
};
```

二、进程状态

- Unix V6++是采用连续存储管理方式的分时多道系统。



1、运行态

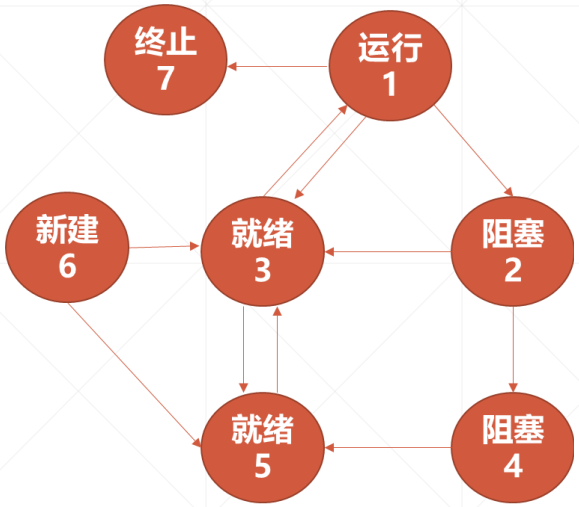
运行态进程正在使用 CPU，是现运行进程。具体标识如下：

p_stat == SRUN，p_flags & SLOAD!=0，p_wchan==0。

现运行进程与其它进程之间的区别在于：

- CPU 正在使用的页表反映该进程的地址映射关系。具体而言，对 Unix V6++系统：CR3 寄存器引用的核心页表（0x201#页框），1023# PTE 的 base 是分配给该进程 PPDA 区的物理页框号。用户页表（0x202#，0x203#页框）中的所有 PTE 分别指向该进程的一页代码、一页数据或一页用户栈，base 等于分配给这个用户页面的物理页框号。
- CPU 内的寄存器是该进程的执行现场。特别地，EIP 是该进程要执行的下条指令的地址，ESP 和 EBP 定位该进程正在使用的栈帧。

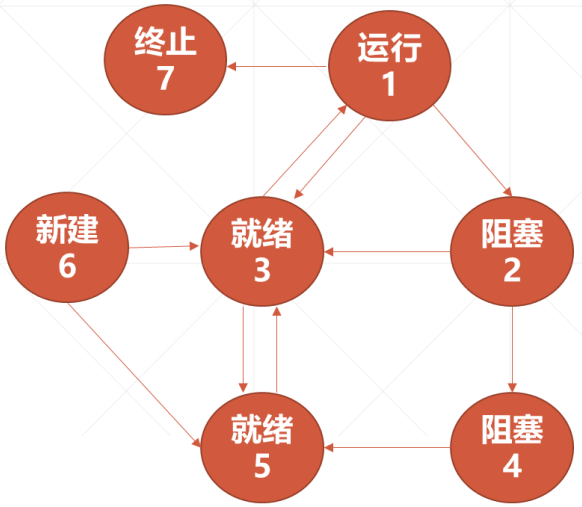
2、就绪态



内存就绪进程可以使用 CPU，具体标识如下：↵

p_stat == SRUN，p_flags & SLOAD!=0，p_wchan==0。↵

这种进程有使用 CPU 的权利。↵



状态 3：阻塞态（内存）↵

阻塞态进程，Unix 喜欢称它们为睡眠进程，没有使用 CPU 的权利。导致进程入睡的可能有 3 个原因（1）它要处理的外部数据内存里没有[注]（2）它要使用的外设或共享数据，锁住了，别人在用（3）它的前驱任务没执行完。具体标识如下：↵

p_stat == SSLEEP/SWAIT，p_flags & SLOAD!=0，p_wchan == &某内核变量。↵

睡眠原因不同，进程会分别进入高优先级睡眠状态或低优先级睡眠状态。状态转换图体现不出它们的区别，故暂时不予以细分。↵

[注] 外部数据不在进程图像中，没有地址映射关系，它们来自文件系统，键盘，网络链接...↵

Unix V6++使用连续内存管理方式，现运行进程全部在内存，不存在部分图像不在内存的情况，陈述 1 成立。若是使用虚拟存储器，此处还应该包括，进程逻辑页面，即代码、数据未驻留主存。↵

Unix V6++是对换系统，使用主硬盘上的交换区扩充物理内存。内存紧张时，进程图像会被换出（swap out），存放在盘交换区上。内存紧张情况得到缓解之后，系统会换入（swap in）盘交换区上的就绪进程，让它重新获得运行机会。为了区分图像在盘交换区上的进程，设置状态 4 和状态 5，用 SLOAD 标识记录进程图像的位置。磁盘上的进程不能运行，因为内存没有它们执行需要的代码或数据。↵

↵

状态 4：就绪态（磁盘）↵

具体标识：p_stat == SRUN，p_flags & SLOAD==0，p_wchan==0。↵

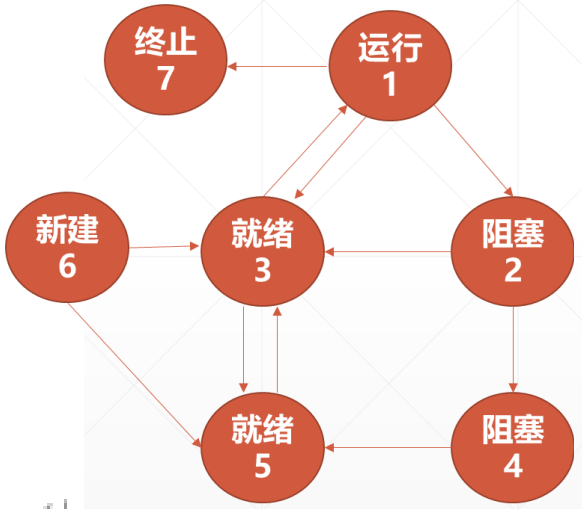
这种进程没有使用 CPU 的权利。↵

↵

状态 5：阻塞态（磁盘）↵

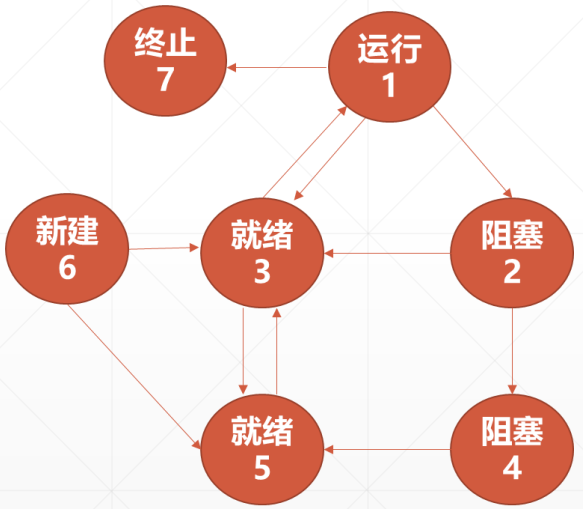
具体标识：p_stat == SSLEEP/SWAIT，p_flags & SLOAD==0，p_wchan == &某内核变量。↵

这种进程没有使用 CPU 的权利。↵



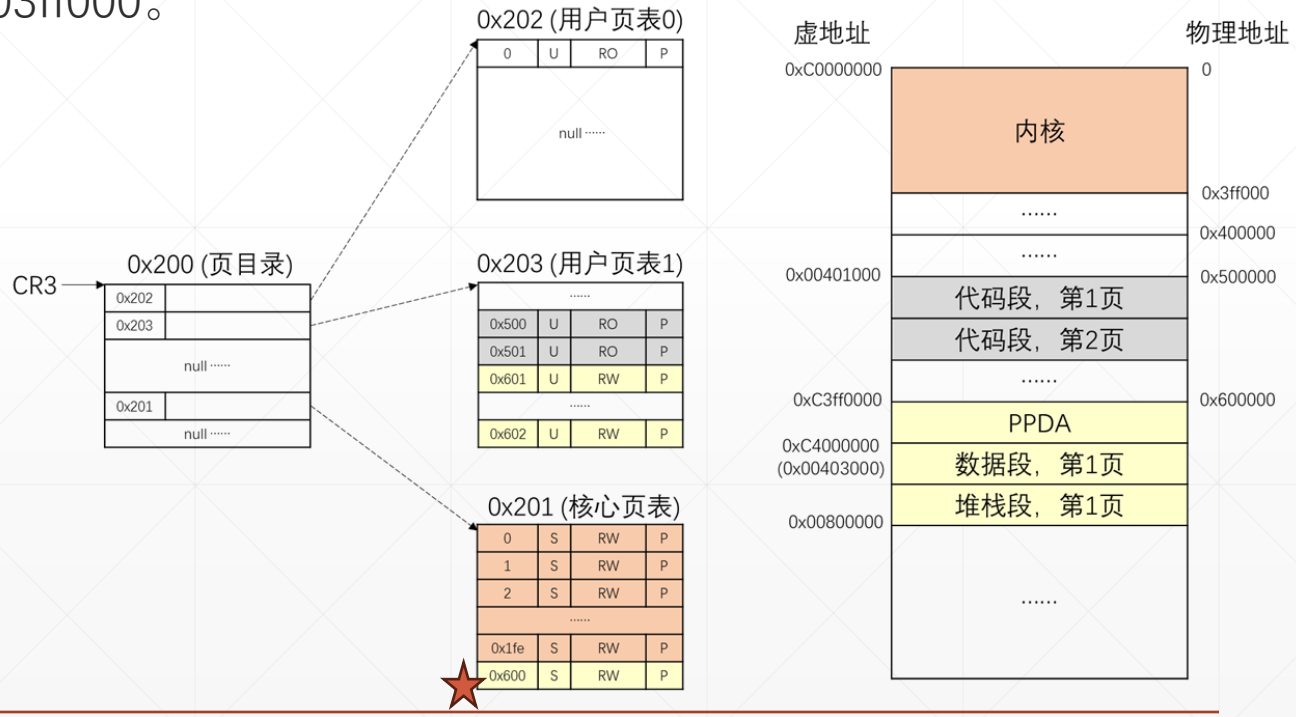
状态6：新建
p_stat == SIDL 和 p_flag字段中的SLOCK标识是父进程创建子进程过程中Process结构使用的临时标识。

状态7：终止
p_stat == SZOMB。标识进程执行exit( )之后，释放所有资源，只剩PCB（Process结构和user结构）的状态。



三、跟踪现运行进程

- 核心态页表（0x300#页表）管理的线性地址空间是[0xC0000000，0xC03fffff]。最后一页（页号是0x3ff）映射现运行进程的PPDA区，首地址是0xC03ff000。PPDA区的首部是user结构。所以，现运行进程user结构的起始地址是0xC03ff000。



Unix V6++用地址0xC03ff000跟踪现运行进程

第一步，找到现运行进程的 user 结构。

```
// u 变量是现进程的 user 结构，首地址 0xC03ff000
User& u = Kernel::Instance().GetUser();

User& Kernel::GetUser()
{
    return *(User*)USER_ADDRESS;     // 0xC03ff000
}
```

之后，用 u.u_procp->引用现运行进程 Process 结构的其它字段，比如时钟中断处理程序里的这一段：

```
Process* current = u.u_procp;
current->p_cpu = Utility::Min(++current->p_cpu, 1024);
```

三、Swtch( )

- 现运行进程执行Swtch( )函数放弃CPU，随后系统选中、恢复另一个进程的CPU现场。后者从Swtch( )函数返回，从上次执行的断点开始恢复执行。这是CPU的进程切换操作。

```
int ProcessManager::Swtch()
{

    User& u = Kernel::Instance().GetUser();
    SaveU(u.u_rsav);

    Process* procZero = &process[0];
    X86Assembly::CLI();
    SwtchUStruct(procZero);
    RetU();
    X86Assembly::STI();

    Process* selected = Select();

    X86Assembly::CLI();
    SwtchUStruct(selected);
L:  RetU();
    X86Assembly::STI();
    User& newu = Kernel::Instance().GetUser();
    newu.u_MemoryDescriptor.MapToPageTable();
    ……
    return 1;
}
```

```
int ProcessManager::Swtch()
{
    User& u = Kernel::Instance().GetUser();
    SaveU(u.u_rsav);
```

执行这段代码时，寄存器 ESP 和 EBP 指向 Swtch栈帧

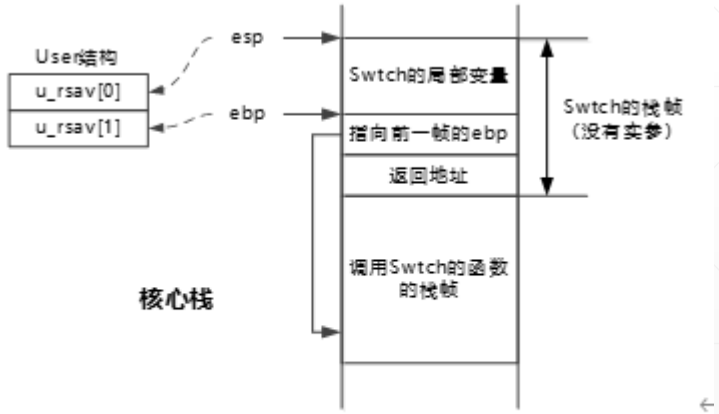


图 3 非现运行进程的核心态现场，栈顶是 Swtch 帧
（所有进程皆如此，包括 0#进程）

第1行，`User& u = Kernel::Instance().GetUser();`

获得下台进程的 user 结构。

第 2 行，`SaveU(u.u_rsav);`

SaveU 宏展开得到两条汇编指令。下台进程执行这两条指令，将核心栈顶指针 ESP 和 Swtch( )栈基地址 EBP 保存进 user 结构中的 u_rsav 数组。

```
movl %esp, u.u_rsav[0]
movl %ebp, u.u_rsav[1]
```

SaveU 宏定义：

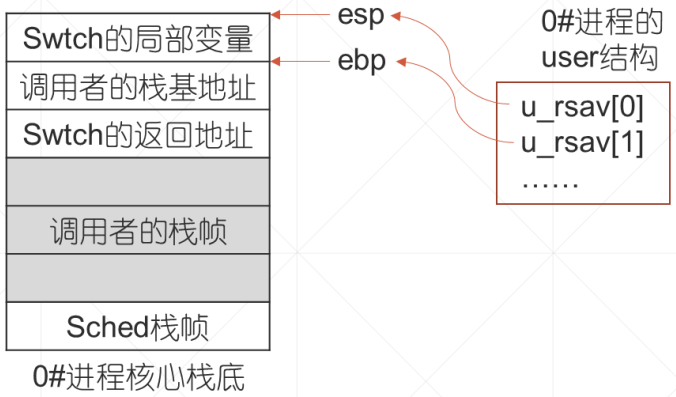
```
#define SaveU(u_sav) \
    __asm__ __volatile__ ( \
        "movl %%esp, %0\n\t \
         movl %%ebp, %1\n\t" \
        :"+m"(u_sav[0]),"+m"(u_sav[1]) \
    );
```

用字符串 u.u_rsav 替换形参 u_sav，得到与上面的 2 条汇编指令相对应的内联汇编语句：

第 1 行和第 2 行语句是下台进程执行的。执行期间，核心栈状态如图 3 所示。这是下台进程放弃 CPU 时的现场，更是所有非现运行进程的核心栈现场。Swtch( )是栈顶帧，下次恢复运行后，执行返回地址指向的指令。称该指令是下台进程的断点。

```
int ProcessManager::Swtch()
{
    ……
    Process* procZero = &process[0];
    X86Assembly::CLI();
    SwtchUStruct(procZero);
    RetU();
    X86Assembly::STI();
```

0#进程成为现运行进程，
从user结构中恢复0#进程的栈顶。



第 3 行：Process* procZero = &process[0];↵
procZero 指向 0#进程的 Process 结构。↵
↵
第 5 行：SwtchUStruct(procZero)↵
建立 0#进程的地址映射关系，我们要用它的核心栈跑 select 程序。具体而言，就是将 0#进程 PPDA 区所在的物理页框号填入核心页表（0x201#页框）1023# PTE。0#进程没有用户态地址空间，这里不必更新用户页表。↵
该语句执行完毕后，CPU 可以访问 0#进程的 PPDA 区，访问其 user 结构（用虚地址 0xC03ff000）和 核心栈。这是在为恢复核心栈现场打伏笔。↵
↵
SwtchUStruct 宏定义：↵

```
#define SwtchUStruct(p) \↵
  Machine::Instance().GetKernelPageTable().m_Entrys[Kernel::USER_PA
GE_INDEX].m_PageBaseAddress \↵
  = (p)->p_addr / PageManager::PAGE_SIZE; \↵
  FlushPageDirectory();↵
```

↵
宏展开后是2个操作：↵
1、核心态页表[1023].m_PageBaseAddress = ProcZero->p_addr/4096; ↵
2、FlushPageDirectory(); // PTE 更新后要刷新 TLB↵

第 6 行：RetU(); ↵
恢复0#进程的核心栈现场。具体而言，用虚地址0xC03ff000从0#进程的User结构（u_rsav数组）中取出之前下台时保存的esp和ebp、赋值ESP和EBP，令这两个寄存器指向0#进程核心栈中的Swtch栈桢。上述操作对应宏 RetU，定义如下：↵
↵

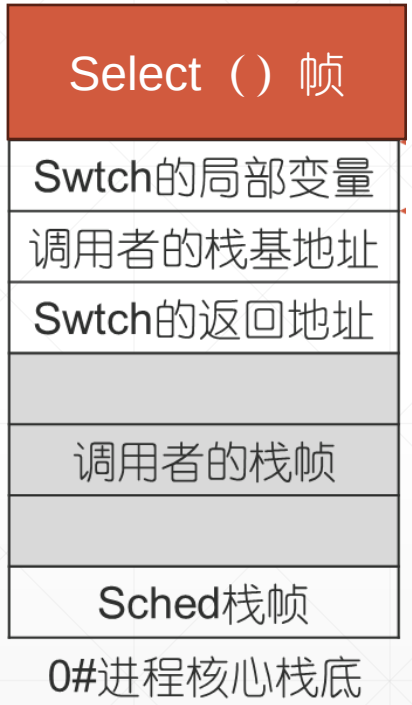
```
#define RetU()  \↵
  __asm__ __volatile__("  movl %0, %%eax;              \↵
                       movl (%%eax), %%esp;     \↵
                       movl 0x4(%%eax), %%ebp;" \↵
                       :                        \↵
                       : "i" (0xc03ff000) );↵
```

↵
至此，0#进程的地址映射关系 和 执行 Swtch()函数的核心栈现场得以恢复，系统完成了 CPU 控制权向 0#进程的转移。EIP 和核心空间地址映射关系未变，CPU 继续执行 Swtch 函数，但第 7 行代码是 0#进程执行的，运行在该进程的核心栈。↵

```
int ProcessManager::Swtch()
{
    ……

    Process* selected = Select();
```

0#进程执行select( )函数，挑选内存中优先级最高的就绪进程。
如果没有，等。。。直到选中新运行进程！！！

selected 指向新选中进程的 Process 结构



```
713  Process* ProcessManager::Select ()
714  {
715      /* 前一次选中上台进程 */
716      static int lastSelect = 0;
717
718      while (true)
719      {
720          int priority = 256;
721          int best = -1;  /* 本轮搜索找到的最合适上台进程 */
722
723          this->RunRun = 0;
724
725          /* 搜索优先级最高的可运行进程 */
726          for ( int count = 0; count < NPROC ; count++ )
727          {
728              /* 从上一次被选中进程的下一个开始回环扫描，而不是每次从0#进程开始，保证各进程机会均等 */
729              int i = (lastSelect + 1 + count) % NPROC;
730              if ( Process::SRUN == process[i].p_stat && (process[i].p_flag & Process::SLOAD) != 0 )
731              {
732                  if ( process[i].p_pri < priority )
733                  {
734                      best = i;
735                      priority = process[i].p_pri;
736                  }
737              }
738          }
739          if ( -1 == best )
740          {
741              __asm__ __volatile__("hlt");
742              continue;
743          }
750          /* 如果选出优先级最高的可运行进程 */
751          this->CurPri = priority;
752          lastSelect = best;
753          //Diagnose::Write("Process %d is runnin
754          return &process[best];
755
756      }
757  }
```

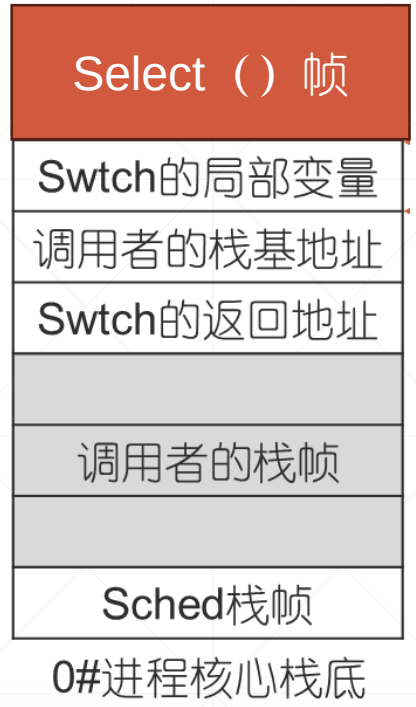
lastSelect 是类变量，系统初始化时赋 0

1、清强迫调度标识

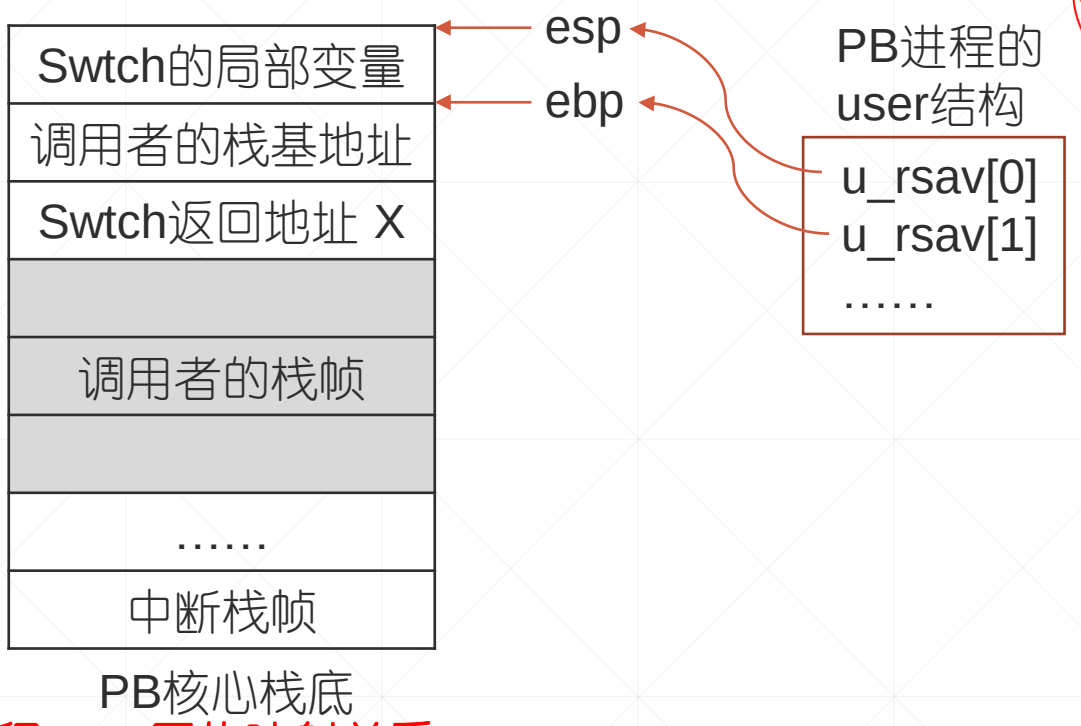
2、遍历process数组，选内存中，优先级最高的就绪进程，best是其Process结构的下标，priority是其优先级。

3、CurPri，新运行进程优先级

4、修正lastSelect，指向选中进程

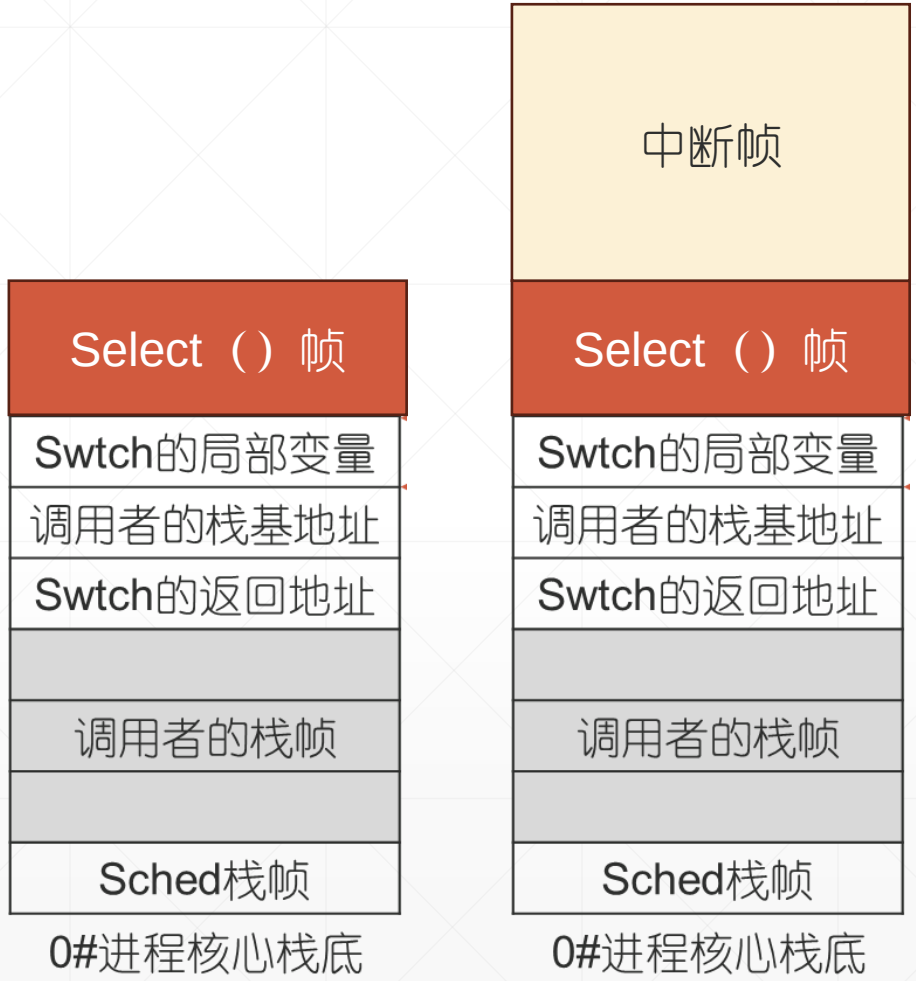


```
int ProcessManager::Swtch()
{
    User& u = Kernel::Instance().GetUser();
    SaveU(u.u_rsav);

    Process* procZero = &process[0];
    X86Assembly::CLI();
    SwtchUStruct(procZero);
    RetU();
    X86Assembly::STI();

    Process* selected = Select();

    X86Assembly::CLI();
    SwtchUStruct(selected);  //3.1 恢复新运行进程PPDA区的映射关系
    RetU();          //3.2 user结构中恢复swtch栈帧
    X86Assembly::STI();
    User& newu = Kernel::Instance().GetUser(); //3.3 用虚地址0xC03ff000取选中进程的user结构
    //3.4 用相对虚实地址映射表写系统页表，建立选中进程用户空间的虚实地址映射关系
    newu.u_MemoryDescriptor.MapToPageTable();
    ……
    return 1;     // 选中进程从Swtch返回：     执行Swtch()返回地址处的指令 X，eax==1 。
}
```



```
713  Process* ProcessManager::Select ()
714  {
715      /* 前一次选中上台进程 */
716      static int lastSelect = 0;
717
718      while (true)
719      {
720          int priority = 256;
721          int best = -1;  /* 本轮搜索找到的最合适上台进程 */
722
723          this->RunRun = 0;
724
725          /* 搜索优先级最高的可运行进程 */
726          for ( int count = 0; count < NPROC ; count++ )
727          {
728              /* 从上一次被选中进程的下一个开始回环扫描，而不是每次从0#进程开始，保证各进程机会均等 */
729              int i = (lastSelect + 1 + count) % NPROC;
730              if ( Process::SRUN == process[i].p_stat && (process[i].p_flag & Process::SLOAD) != 0 )
731              {
732                  if ( process[i].p_pri < priority )
733                  {
734                      best = i;
735                      priority = process[i].p_pri;
736                  }
737              }
738          }
739          if ( -1 == best )
740          {
741              __asm__ __volatile__("hlt");
742              continue;
743          }
```

3、如果没有，best == -1。0#进程执行hlt指令，CPU停机等中断。EIP是continue。



停机状态CPU响应中断，先前态是核心态，中断返回无例行调度，CPU执行continue指令回到while循环的开始，再去找就绪进程。

如果，中断有唤醒睡眠进程，后者会立即被选中执行

进程入睡，基本工作过程

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

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();
    }
    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();
        }
    }
}
```

进程唤醒，基本工作过程

- 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++;
    }
        ……
}
```