# 进程管理模块

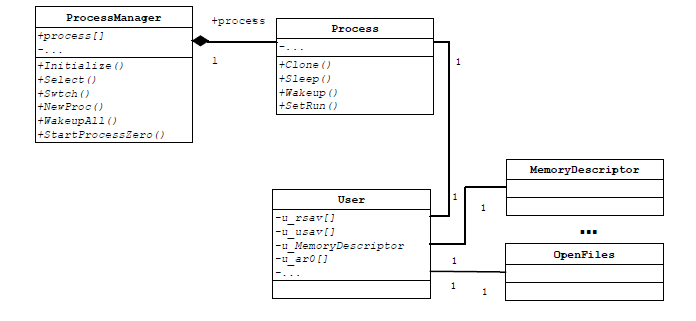
程序在操作系统中运行的实体被称为进程。在操作系统的各个管理模块中，进程管理模块主要将负责管理用户程序进程的创建，运行，终止等操作，统计进程运行时信息，同时提供一套有效公平的进程调度机制。同时，进程管理模块还需要提供一套有效的进程与进程通讯的机制。

用户程序的创建(fork)，终止(exit)，改变图像(exec)等操作都是由进程管理模块提供的，用户通过系统调用接口对操作系统发出请求，由进程管理模块负责具体进程的创建与终止，资源的分配和释放的处理逻辑。

在Unix V6++中，采用了与Unix V6的同样数据结构管理进程信息，只是为进程管理模块添加了ProcessManager类，该类抽象了进程管理的主要工作，例如进程切换调度，进程的唤醒以及进程的创建等。大多数只和进程有关的操作，被转移到Process类中，例如进程的睡眠等。

## 进程管理模块类结构

在进程管理模块中，ProcessManager类封装了所有与进程管理相关的操作，该类管理集中管理了进程的创建，切换调度等。下面的图给出了进程管理模块中几个重要类之间的关系(图7.1)。



#### 图7.1 进程管理模块类结构

Process与User类是进程控制块，保存了进程的运行状态，可以通过该数据结构访问到该进程所占用的所有资源情况。Process类向内核暴露了进程实体所应提供的方法，例如Clone就是在创建子进程时拷贝一份进程控制块，Sleep()函数负责进程的睡眠操作，所有进程相关的方法都被封装在在类中。

User类与进程的核心栈在内存中连续排列，这两部分合并称为进程的ppda区，通常ppda区占据一页4K大小的内存。每个现运行进程的ppda区被映射在操作系统内核空间的最后一页即0xc03ff0000地址。进程的User对象有可能被换出内存而Process对象是常驻内存的。通过User结构，可以找到进程的打开文件信息(被封装在OpenFiles类中)，内存图像占据内存的信息(被封装在MemoryDescriptor类中)等资源使用信息。

在Unix V6中，进程的所有信息都存放在全局数据进程控制块proc[NPROC]数组中，单个进程的信息或存储于进程控制块proc结构中，或由proc结构勾连的扩展进程控制块user数据结构保存。

class ProcessManager

{

public:

Process process[NPROC];

Text text[NTEXT];

};

#### 代码7.1 process和text数组

在Unix V6++中，把进程控制块Process process[NPROC]数组作为ProcessManager类的成员变量。此外，与进程控制块一样，系统中还设置了共享正文段控制块数组Text text[NTEXT]，作为进程管理模块的一部分，也将其作为ProcessManager的成员变量，由该类负责对这类数据结构的操作(代码7.1)。

其中，常量NPROC表示系统能容纳的最大进程数量，一般设置为100；NTEXT为系统最多允许共享正文段的数量，我们将其设置为50。

## 进程控制信息数据结构

### 进程控制块--Process类

Process类包含了进程的基本控制信息，每个进程在内核process[]数组中都占据一项，是进程常驻内存部分，其定义如下(代码7.2)：

class Process

{

public:

/\* 用于标识进程的标识 \*/

short p\_uid; /\* 用户ID \*/

int p\_pid; /\* 进程标识数，进程编号 \*/

int p\_ppid; /\* 父进程标识数 \*/

/\* 进程图像内存中的信息位置 \*/

unsigned long p\_addr; /\* user结构即ppda区在内存中的首地址 \*/

unsigned int p\_size; /\* 除共享正文段的长度，以字节单位 \*/

Text\* p\_textp; /\* 指向该进程所运行的代码段的描述符 \*/

/\* 进程调度状态 \*/

ProcessState p\_stat; /\* 进程当前状态 \*/

int p\_flag; /\* 进程标志位，可以将多个状态组合 \*/

int p\_pri; /\* 进程优先数 \*/

int p\_cpu; /\* 进程占用cpu时间，用于计算p\_pri \*/

int p\_nice; /\* 进程优先数微调参数 \*/

int p\_time; /\* 进程在交换区上(内存中)驻留时间 \*/

/\* 进程睡眠原因，一般为内存地址，等待某个内核数据结构 \*/

unsigned long p\_wchan;

……

};

#### 代码7.2 进程控制块Process类成员变量

Process中的成员变量基本上是沿用了Unix V6中的proc进程控制块结构，各个成员的命名方式仍然是按p\_作为前缀，Process类中成员大致可分为以下四类：

* 进程标识数

**用户标识数p\_uid：**用户标识数是用户在登录以后内核区别不同用户所使用的用户编号，系统将用户外部使用的用户名字符串转换为系统内部使用的用户编号。

**进程标识数p\_pid：**进程标识数是进程创建时由系统赋予的进程编号，各个进程使用标识数不同。由于p\_pid使用int类型，因此即使采用最简单的进程号分配方式，每次分配的进程标识+1，也可以保证在相当长的时间内各活动进程标识数不重复。

**父进程标识数p\_ppid：**除0#进程以外，其它进程都是应其父进程要求而生成的。p-ppid记录了父、子进程之间的关系。

* 进程图像内存中位置信息

**进程图像地址p\_addr：**该进程User对象在内存中的首地址，也即该进程ppda区的内存首地址，以字节为单位(不同于原Unix V6中以内存字符块64字节或外存字符块512字节为单位)。

**进程图像长度p\_size：**该进程图像(除共享正文段以外部分)的长度，单位是字节。

**共享正文段Text指针p\_ textp：**这个字段包含指向该进程共享正文段控制块Text对象的指针，Text对象记录着该进程图像的代码段在内存中的位置和长度信息。

* 进程调度相关信息

**当前进程状态p\_stat：**标识进程当前是否正在占用CPU，或正等待使用CPU，或是处于睡眠中。Unix V6++中基本沿用原先定义的进程调度状态(代码7.3)。

enum ProcessState

{

SNULL = 0, /\* 未初始化空状态 \*/

SSLEEP = 1, /\* 高优先权睡眠 \*/

SWAIT = 2, /\* 低优先权睡眠 \*/

SRUN = 3, /\* 运行、就绪状态 \*/

SIDEL = 4, /\* 进程创建时的中间状态 \*/

SZOMB = 5, /\* 进程终止时的中间状态 \*/

SSTOP = 6 /\* 进程正被跟踪 \*/

};

#### 代码7.3 进程调度状态

**进程图像状态标志p\_flag：**表示进程图像是否在内存中，或是在磁盘交换区上；以及是否可以被换出至交换区等信息。其定义如代码7.4所示，一个进程图像的状态可以是一个或者多个标志位的组合。

enum ProcessFlag

{

SLOAD = 0x1, /\* 进程图像在内存中 \*/

SSYS = 0x2, /\* 系统进程图像，不允许被换出 \*/

SLOCK = 0x4, /\* 含有该标志的进程图像暂不允许换出 \*/

SSWAP = 0x8, /\* 该进程被创建时图像就在交换区上 \*/

STRC = 0x10, /\* 用于父子进程跟踪机构中 \*/

STWED = 0x20

};

#### 代码7.4 进程图像状态标志定义

**进程优先数p\_pri：**其值越小，优先权越高。它是进程调度的主要依据。数值变化范围是非曲直-100~+127。

**进程占用cpu时间p\_cpu：**p-cpu 反映了进程使用处理机的程度。p-cpu值愈大表示进程使用处理机的程度愈高。它是计算p-pri的一个主要参数。

**优先数微调参数p\_nice：**计算进程优先数时所用的偏置值，经可以由用户设置。

**进程驻留时间p\_time：**进程图象自最近一次调入(或调出后)，在内存(或在盘交换区上)驻留的时间。这是进行进程图象在内、外在之间传送的确良一个重要依据。

* 其它

**进程睡眠原因p\_wchan：**记录进程因等待获取内核中某个资源释放而进入睡眠的原因，一般为该数据结构的内存地址，当该原因消失时，唤醒所有因等待该资源的睡眠进程。

### 进程扩充控制块—User类

User类包含了进程的扩充控制信息，每个进程都有自己的User对象，它与该进程的Process对象一一对应。

每个进程的User对象与核心栈共同构成该进程的ppda区，占据一页4K大小的内存，User对象的数据在该页中从低地址向高地址排布，而核心栈的数据从该页底端往内存低地址方向增长。

User对象属于进程图像的非常驻内存部分，当内存不足情况下，会将进程的User对象、核心栈以及进程用户图像部分一起换出内存。从这一点可以看出，Process类和User类都存储进程控制信息，其区别在于Process类中的进程控制信息当进程没有占用CPU时仍需要被内核查询和修改，所以必须驻留在内存中；而那些进程下台时就不需要被用到的进程控制信息则放入User类中。

这也是原Unix V6中为了将进程控制块占据的内存资源最小化而分离出Process和User两部分进程控制块的原因，在Unix V6++中虽然没有了如此苛刻的内存限制条件，但我们仍旧遵照原先的设计。

进程扩充控制块User类的具体定义如下 (代码7.5)：

class User

{

unsigned long u\_rsav[2]; /\* 用于保存esp与ebp指针 \*/

unsigned long u\_ssav[2]; /\* 用于对esp和ebp指针的二次保护 \*/

Process\* u\_procp; /\* 指向该u结构对应的Process结构 \*/

MemoryDescriptor u\_MemoryDescriptor; /\*替代原有内存管理相关字段 \*/

/\* 系统调用相关成员 \*/

unsigned int \*u\_ar0; /\* 指向核心栈现场保护区中EAX寄存器存放单元 \*/

int u\_arg[5]; /\* 存放当前系统调用参数 \*/

char\* u\_dirp; /\* 系统调用参数(一般用于Pathname)的指针 \*/

/\* 时间相关成员 \*/

int u\_utime; /\* 进程用户态时间 \*/

int u\_stime; /\* 进程核心态时间 \*/

int u\_cutime; /\* 子进程用户态时间总和 \*/

int u\_cstime; /\* 子进程核心态时间总和 \*/

/\* 信号处理相关成员 \*/

unsigned long u\_signal[NSIG]; /\* 信号处理表 \*/

unsigned long u\_qsav[2]; /\*接收到信号时直接从Sleep()函数跳回至Trap() \*/

bool u\_intflg; /\* 系统调用期间是否受到信号打断 \*/

/\* 文件系统相关成员 \*/

Inode\* u\_cdir; /\* 指向当前目录的Inode指针 \*/

Inode\* u\_pdir; /\* 指向父目录的Inode指针 \*/

DirectoryEntry u\_dent; /\* 当前目录的目录项 \*/

char u\_dbuf[DirectoryEntry::DIRSIZ]; /\* 当前路径分量 \*/

char u\_curdir[128]; /\* 当前工作目录完整路径 \*/

ErrorCode u\_error; /\* 存放错误码 \*/

int u\_segflg; /\* 表明I/O针对用户或系统空间 \*/

/\* 进程的用户标识 \*/

short u\_uid; /\* 有效用户ID \*/

short u\_gid; /\* 有效组ID \*/

short u\_ruid; /\* 真实用户ID \*/

short u\_rgid; /\* 真实组ID \*/

/\* 文件系统相关成员 \*/

OpenFiles u\_ofiles; /\* 进程打开文件描述符表对象 \*/

/\* 文件I/O操作 \*/

IOParameter u\_IOParam; /\* 读、写文件的偏移量，目标区地址和剩余字节 \*/

};

#### 代码7.5 进程扩充控制块User类的成员变量

User类中包含进程的各类信息，如进程用户表示数、进程运行的时间统计项、系统调用相关项、文件读/写相关的项，以及文件系统相关的项等等。这里并不打算对User类中的所有成员变量逐个展开，仅仅解释进程管理模块涉及到的一些项。

**Process对象指针u\_procp：**指向该进程的进程控制块Process对象，每个进程的Process对象与User对象是一一对应的绑定关系，User对象通过该指针找到对应的Process对象，而Process对象也可以通过p\_addr找到对应的User对象，从而建立每个进程Process与User两者之间的勾连关系。

**现场保护有关项u\_rsav[2]、u\_ssav[2]：**进程放弃CPU时其运行现场保存在核心栈中，这两个成员变量用于进程切换时保存当前函数调用栈帧在核心栈中的位置，换言之，也就是保存ESP和EBP寄存器的值，它们分别指出当前栈帧的顶部和底部在核心栈中的位置。

### 共享正文段控制块—Text类

由于内核允许在多个进程之间共享代码段，所以关于进程代码段的控制信息不能包含在某个特定进程的进程控制块中，因此独立出Text类用于保存共享正文段相关的控制信息。共享正文段控制块Text类的定义如下(代码7.6)：

class Text

{

public:

int x\_daddr; /\* 代码正文段在盘交换区上的地址 \*/

unsigned long x\_caddr; /\* 代码正文段内存起始地址，以字节为单位 \*/

unsigned int x\_size; /\* 代码段长度，以字节为单位 \*/

Inode\* x\_iptr; /\* 内存inode地址 \*/

unsigned short x\_count; /\* 共享正文段的进程数 \*/

unsigned short x\_ccount; /\* 共享该正文段且图像在内存的进程数 \*/

};

#### 代码7.6 共享正文段控制块Text类的成员变量

当共享该正文段的多个进程之一，其图像在内存中时，该共享正文段必定也在内存中；如果共享该正文段且图像在内存的进程数为0，则该共享正文段可以被换出至交换区，是否占据的内存资源。

### 进程控制块的勾连结构

一般而言，对于每个进程，与其对应的进程控制块Process对象、进程扩充控制块User对象以及共享正文段控制块Text对象三者之间存在相互勾连关系，从而可以依据三者之一找到该进程对应的另外两项控制信息，通常是从进程的User对象，沿着u\_procp指针找到对应的Process对象，继而根据Process对象的p\_textp指针得到Text对象。

**p\_addr**

**p\_textp**

**...**

**Text[i]**

**u\_procp**

共享正文段

进程控制块

User对象

**...**

#### 图7.2进程控制块的勾连结构

## 进程的切换--Swtch()

Unix V6++中进程切换由ProcessManager::Swtch()函数完成，基本算法思路与Unix V6相同，主要的不同在于进程切换时，所需要保存的寄存器与底层页表切换代码不同。这里主要讨论的是Unix V6++中进程切换如何实现，至于进程调度策略—即从所有就绪状态进程中选择最适合占用处理机的进程—则不具体展开。在ProcessManager类中使用了两个分离的函数Swtch()和Select()分别负责进程切换和进程的调度(代码7.7)。

class ProcessManager

{

int Swtch(); /\* 负责进程切换时保存、恢复现场的工作 \*/

Process\* Select(); /\* Swtch()中间段0#进程调用此函数选出最适合上台进程 \*/

};

#### 代码7.7 Swtch()和Select()函数

从而将进程的调度策略由进程切换Swtch()函数中独立出来，使得更容易对进程调度策略进行修改和重新实现，提高了代码的可维护性。

### 进程切换Swtch()的基本过程

进程切换Swtch()函数的基本过程如图7.3中所示，该程序的特殊之处在于它的执行会涉及三个进程：原运行进程、0#进程、新选中进程，因而Swtch()函数因此也可划分为对应的三段进行说明我们假设Swtch()的切换是从A进程🡪0#进程🡪B进程。

* 原运行A进程段

原运行进程A在放弃处理机之前，需要保存程序运行的现场，而对于A进程来说，其所有现场信息都包含在核心栈中的函数调用栈帧中，因此只需记录下核心栈中栈帧的位置，也就是将esp和ebp寄存器的值保存到User对象的u\_rsav[2]中。

然后恢复0#进程上台，具体执行的操作分为两步：(1)根据process[0].p\_addr可以得到0#进程ppda区的物理地址，将该地址放入核心态页表的最后一项(第1023项)，建立起对0#进程ppda区的页表映射，使得我们可以获取User对象的u\_rsav[2]；(2)将u\_rsav[2]中恢复出0#进程的esp和ebp寄存器值，从而使0#进程成为现运行进程。

* 0#进程段

0#进程上台以后调用Select()函数，该函数负责具体的进程调度策略，通过线性搜索process表，从系统中所有就绪状态且图像在内存中的进程里面选出优先权最高的进程，并且返回指向被选中进程Process对象的指针。如果没有满足条件的进程，则0#进程在Select()函数中进入停滞状态，等待中断的发生，因为中断请求发生后系统中某些进程就会转入可供调度的就绪状态，0#重新搜索整个process表直到选出最适合调度上台的进程，并将指向该进程的Process对象的指针返回给Swtch()。

选出进程B以后0#进程的工作并没有结束，它还需要使B进程成为现运行进程。同样，将B进程的ppda区首地址放入核心态页表的最后一项(第1023项)，建立对该进程ppda区的映射，然后从它的u\_rsav[2]中恢复出esp和ebp寄存器值，便完成了让B进程成为现运行进程的工作。

* 新选中B进程段

B进程在成为现运行进程之后，还需要恢复自己的用户态空间映射，为将来返回用户态继续执行用户程序做准备。在此步骤之前系统中页目录第0、1项指向的用户态空间页表中仍然是之前A进程留下的用户态空间页表数据，B进程需要根据其2张页表(确切说是相对地址映照表)中的内容以及共享正文段首地址、ppda区首地址重新设置系统中页目录第0、1项指向的页表，建立起B进程的用户态空间映射。

**Swtch()**

将当前栈帧的esp与ebp保存至u\_rsav[2]

恢复0#进程上台

调用Seclect()搜索process表中优先权最高的就绪进程(找不到就在Select()中等待中断)

切换核心态页表最后指向新选中进程ppda区，并恢复u\_rsav[2]中的esp与ebp

为新选中进程装载2张用户空间页表，建立用户空间映射

返回1

0#进程

原运行进程

新选中进程

#### 图7.3 进程切换Swtch()函数

Swtch()函数的返回值为1，该返回值仅在进程创建过程中起重要作用，将在进程创建部分具体解释。另外，由于Swtch()函数中下台进程保存现场与新选中进程恢复现场互为逆过程，所以Swtch()保证了进程放弃处理机以及将来重新占用处理机时现场完全一样，忽略进程两次调度占用处理机的间隔，对于用户来说，将形成进程宏观上连续运行的效果。

### Swtch()函数的实现

代码7.8给出了Unix V6++中进程切换Swtch()函数的实现。

int ProcessManager::Swtch()

{

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

SaveU(u.u\_rsav);

/\* 0#进程上台\*/

Process\* procZero = &process[0];

X86Assembly::CLI();

SwtchUStruct(procZero);

RetU();

X86Assembly::STI();

/\* 挑选最适合上台的进程 \*/

Process\* selected = Select();

/\* 恢复被保存进程的现场 \*/

X86Assembly::CLI();

SwtchUStruct(selected);

RetU();

X86Assembly::STI();

User& newu = Kernel::Instance().GetUser();

/\* 新选中进程建立用户态映射 \*/

newu.u\_MemoryDescriptor.MapToPageTable();

return 1;

}

#### 代码7.8 Swtch()函数的实现

Swtch()函数使用SaveU(u\_sav)宏保存堆栈指针，该宏中将ESP与EBP寄存器存入当前运行进程User对象的u\_rsav[2]数组中。另外代码中成对使用SwtchUStruct(p)和RetU()这两个宏，用于切换进程上台，在其前后分别加上关闭中断、打开中断以保证恢复进程现场的工作不被打断。这两个宏对应恢复进程上台时的两个步骤：切换至目标进程的ppda区和从u\_rsav[2]中恢复出堆栈指针到esp与ebp寄存器。

由于进程切换Swtch()函数的特殊性，SaveU(u\_sav)、SwtchUStruct(p)和RetU()都使用宏而非函数的方式实现(代码7.9)。

**#define** SaveU**(u\_sav) \**

**\_\_asm\_\_ \_\_volatile\_\_( \**

**"movl %%esp, %0\n\t \**

**movl %%ebp, %1\n\t" \**

**:"+m"(u\_sav[0]),"+m"(u\_sav[1]) \**

**);**

**#define** SwtchUStruct**(p) \**

**Machine::Instance().GetKernelPageTable().m\_Entrys[Kernel::USER\_PAGE\_INDEX].m\_PageBaseAddress \**

**= (p)->p\_addr / PageManager::PHY\_MEM\_PAGE\_SIZE; \**

**FlushPageDirectory();**

**#define** RetU**(u) \**

**\_\_asm\_\_ \_\_volatile\_\_(" movl %0, %%eax; \**

**movl (%%eax), %%esp; \**

**movl 0x4(%%eax), %%ebp;" \**

**: \**

**: "i" (0xc03ff000) );**

#### 代码7.9 SaveU(u\_sav)、SwtchUStruct(p)和RetU()三个宏

SaveU(u\_sav)宏用于进程下台时保存现场，保存堆栈指针到User对象的u\_rsav[2]或u\_ssav[2]数组中，因此该宏使用一个传入参数u\_sav进行区别。u\_ssav[2]是对堆栈指针esp和ebp进行二次保护的单元，具体在进程创建部分进行解释。

SwtchUStruct(p)宏用于将目标进程的ppda区首地址放入核心态页表的最后一项(第1023项)，建立对该进程ppda区的映射，其接收的参数为Process对象，从而由p\_addr得到目标进程ppda区首地址。

RetU(u)宏与SwtchUStruct(p)宏总是成对出现，用于在建立对目标进程ppda区映射之后，从ppda区的低地址0xc03ff000，即User对象中u\_rsav[2]占据的内存位置中恢复出堆栈指针。因此我们将u\_rsav[2]数组作为User类中第一成员对象是为了方便RetU()宏能明确知道它们在User类中的偏移位置。

## 进程的创建—NewProc()

创建新进程的基本任务是为新进程构造一个图像，包括进程控制块Process对象、ppda区(User对象和核心栈组成)、共享正文段、用户态数据段和栈段。新进程创建后即成为系统的一个调度单位，可由Swtch()切换上台占用处理机。并且新进程应能从其User对象和核心栈中获得第一次运行时所需现场。

进程创建的基本方式是复制进程图像，除与进程的状态、标识以及与时间有关的少数控制项以外，子进程复制或共享父进程的图像。

Unix V6++将进程创建时对进程图像的复制分为两部分，第一部分是复制进程控制块Process数据结构，对正文段的共享应该也归于第一部分，因为子进程仅仅复制了父进程Process中的p\_textp成员的值；第二部分是对ppda区、数据段、栈段等父子进程应当各自独立的进程图像部分的复制。

Unix V6中使用newproc()函数完成以上所有工作。在Unix V6++中，将进程控制块Process的复制等工作移动到了Process::Clone()函数中，而ProcessManager::NewProc()函数主要处理空闲进程控制块的选取，分配内存空间并复制出父子进程各自独立的进程图像部分等工作。

void Process::Clone(Process& proc); /\* 子进程拷贝进程控制块中各项 \*/

int ProcessManager::NewProc(); /\* 进程创建，它调用Clone()复制Process对象 \*/

#### 代码7.10 Process::Clone()与ProcessManager::NewProc()函数

### 进程创建的基本过程

除0#进程以外，进程都经由程序ProcessManager::NewProc()创建，在进程创建整个各个阶段父子进程关系如图7.4中(a) ~ (f)所示。

在NewProc()执行的最初阶段，父进程的Process控制块、进程图像ppda区、数据段和栈段之间关系如图7.4(a)中所示。随后，父进程分配一个空闲Process对象，作为子进程的控制块，并将父进程ppda区中User对象的u\_procp指向子进程新分配Process对象，这是为了保证后面复制进程图像时，子进程图像拷贝中的u\_procp是指向自己的Process对象而非指向父进程控制块，否则到最后父子进程独立运行时子进程将无法找到自己的Process对象，因为其拷贝获得的图像中u\_procp是指向父进程Process对象的。此时，子进程除了仅拥有一个进程控制块以外，不占用其它任何内存空间(图7.4(b))。

父进程为子进程新分配的Process控制块对象设置初值，此步骤由父进程调用Process::Clone()函数来完成，例如p-stat设置为SRUN；进程标识数p-pid设置为与现存所有进程不同的一个数值；p-ppid设置为父进程标识数等，子进程Process对象中的大部分初始值都从父进程控制块相应字段中拷贝而来。此外还递增打开文件控制块File结构引用计数，增加对当前工作目录的引用计数，以及增加对共享正文段引用计数等。

父进程为子进程图像分配内存，分配大小根据父进程自身图像大小p\_size决定，也即是ppda区、数据段、栈段占据内存长度值之和。如果内存分配成功，父进程将自己的图像复制一份至新分配区域作为子进程图像，子进程p\_addr指向该内存区域首地址(图7.4(c))。

如果内存分配失败，则子进程p\_addr指向父进程ppda区，借用父进程图像(图7.4(d))，并随后将自己进程图像(实际上是父进程图像)换出到交换区(图7.4(e))。

最后，恢复父子进程独立运行，其u\_procp指向各自的Process对象(图7.4(f))。

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

ppda

核心态页表第1023项

**u\_procp**

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

ppda

核心态页表第1023项

**u\_procp**

**p\_addr**

子进程控制块

(a)NewProc()之前

(b)父进程u\_procp指向子进程控制块

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

内存

核心态页表第1023项

**u\_procp**

**p\_addr**

子进程控制块

子进程图像

栈段

数据段

内存

**u\_procp**

复制

(c)分配内存，复制父进程图像ppda区、数据段和栈段

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

内存

核心态页表第1023项

**u\_procp**

**p\_addr**

子进程控制块

子进程图像

栈段

数据段

交换区

**u\_procp**

复制

(e)复制父进程ppda区、数据段和栈段至交换区

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

内存

核心态页表第1023项

**u\_procp**

**p\_addr**

子进程控制块

子进程图像

栈段

数据段

内存/交换区

**u\_procp**

(f)父子进程恢复独立运行，现运行进程为父进程

**p\_addr**

父进程控制块

父进程图像

栈段

数据段

ppda

核心态页表第1023项

**p\_addr**

子进程控制块

(d)内存不足，准备复制父进程图像至交换区

**u\_procp**

#### 图7.4 NewProc()过程中父子进程关系

具体NewProc()函数的流程图如下(图7.5)：



#### 图7.5 NewProc()

当一个创建好的子进程被调度占用处理机时，它从父进程调用newproc的返回点处开始执行程序。这各父进程是完全相同的。于是就出现了一个问题，从此以后父、子进程是否就只能以完全相同的方式执行同样的程序段呢？对这一问题的回答是否定的，因为创建子进程的目的一般是使它执行一个和父进程不同的任务，因此子进程应能执行与父进程完全不同或者至少不同的程序段。那么，这一点双是如何实现的呢？比较一般的方法是：创建子进程时，将预定的子进程起始执行地址送入它的进程控制块(或栈)中存放EIP的单元；当进程调度程序选中子进程时，就从该单元中恢复EIP，于是子进程就能从预定地址开始其执行过程。而Unix V6++中NewProc()的实现采用了另一种方法，它规定子进程的起始执行点就是父进程调用NewProc()时的返回地址，但是父进程从NewProc()返回时，返回值为0；子进程从Swtch()中退出时，返回值为1。于是，按照返回值就可以判别是父进程还是子进程，并按需使它们执行不同程序段。所以调用NewProc()的方式应为(代码7.11)：

int ans = Kernel::Instance().GetProcessManager().NewProc();

if ( ans == 0 )

//父进程执行部分

else

//子进程执行部分

#### 代码7.11 NewProc()函数调用方式

## 进程的睡眠和唤醒

进程可因各种同步、互斥需要进入睡眠状态，进程通过调用Sleep(chan,pri)使进程进入睡眠状态。当睡眠原因消失后，调用wakeup(chan)将有关睡眠进程唤醒。一般而言，系统中可能有若干进程同时因等待使用某一资源而入睡。当该资源有一个可用时，即将所有这些进程唤醒。它们再次竞争使用该资源，其中有一个进程得到满足，其它进程则不得不再次进入睡眠状态。

在Unix V6++中，由于已经对下层机器硬件做了较为完整封装，因此进程管理模块中，除了进程切换Swtch()与进程创建NewProc()部分会涉及到与体系结构相关的函数之外， Unix V6++大部分算法都继承了原Unix V6中的方式，尤其是针对进程的睡眠与唤醒、进程的终止与同步等下层无关的进程操作，在实现上遵照其原先的算法思路，并不需要进行改动。只是使用面向对象化的设计方法将部分方法归属与Process类，将部分全局操作的方法归属于ProcessManager类中。

Unix V6++将Sleep(chan)函数移动到了Process类中，这样明确的区分了唤醒与睡眠职责范围，因为进程睡眠从语义上来说应该是单个进程的行为，所以作为Process类的成员函数更为合适。在ProcessManager类中添加了WakeupAll(chan)函数，该函数与原Unix V6中wakeup(chan)函数职责相同，唤醒因为chan睡眠的所有进程，该函数会调用Process类的SetRun()函数将进程转入就绪状态(代码7.12)。

void ProcessManager::WakeUpAll(unsigned long chan)

{

/\* 唤醒系统中所有因chan而进入睡眠的进程 \*/

for(int i = 0; i < ProcessManager::NPROC; i++)

if( this->process[i].IsSleepOn(chan) ) this->process[i].SetRun();

}

#### 代码7.12 ProcessManager::WakeUpAll()函数的实现

### 进程睡眠—Sleep(chan,pri)

进程调用Sleep(chan,pri)进入高、低优先权睡眠状态。参数chan表示睡眠原因，一般这是一个变量、数组或数据结构的内存地址。例如，某进程因竞争使用某一资源不能得到满足而需进入睡眠状态时，睡眠原因就是一个指针，它指向代表该资源的一个数据结构。又如某些变量的值表示了相应事件是否已经发生。如果因等待该事件而进入睡眠状态，则睡眠原因就是相应标志变量的指针。Pri是睡醒后该进程的优先数，若其值为负，则该进程进入高优先权睡眠状态，否则进入低优先睡眠状态。



#### 图7.6 Sleep(chan,pri)

* Sleep(chan,pri)的基本工作过程

Sleep(chan,pri)函数的核心部分如代码7.13所示：

X86Assembly::CLI();

this->p\_wchan = chan;

this->p\_stat = Process::SSLEEP; /\* 高优先权睡眠 \*/

(或this->p\_stat = Process::SWAIT; /\* 低优先权睡眠 \*/)

this->p\_pri = pri;

X86Assembly::STI();

Kernel::Instance().GetProcessManager().Swtch();

#### 代码7.13 Sleep(chan,pri)的高低优先权睡眠

This指针表示当前调用Sleep(chan,pri)进入睡眠的进程控制块。这三条语句构成了一个临界区。为了防止其它进程进入同类临界区，在该临界区前，屏蔽中断进入，在其后则打开终端。然后调用进程切换Swtch()程序，使本进程放弃CPU并选择其它进程上台。

* 低优先权睡眠处理

在低优先权睡眠处理部分插入两个操作。一个是在其首、尾部分进行信号检测，如果收到信号，则直接跳转至Trap()陷入处理子程序进行信号处理。另一个操作则与标志runin有关。

runin标志与进程图象在内存和盘交换区之间的传送有关。如若runin标志已经设置则表示在盘交换区上有准备就绪进程图象需要调入内存，但是在内存中无足够大的空闲区，而且也没有合适的进程可以调出。当一个进程进入低优先权睡眠状态后，因为它等待的事件一般进行得比较缓慢，因此可以将其图象送到盘交换区上。于是，就可以唤醒因runin而睡眠的0#进程,使它再次试图进行进程图象在内存和盘交换区之间传送操作。

### 进程唤醒—WakeUpAll(chan)

WakeUpAll(chan)唤醒所有因chan而进入睡眠的进程，因为系统中可能有多个进程因等待相同的资源而睡眠，唤醒时需要将所有这些进程全部唤醒，所以唤醒操作并不像Sleep(chan,pri)那样属于单个进程的行为，而是针对与chan有关的所有进程的操作。因而在Unix V6++中唤醒操作是进程管理类ProcessManager的成员，为了突出进程的睡眠与唤醒这两个操作的非对称性，我们将执行唤醒睡眠进程的函数命名为WakeUpAll(chan)。与原Unix V6中wakeup(chan)函数功能相同。

* 基本工作过程

WakeUpAll(chan)的工作过程见图7.7。它线性扫描ProcessManager类中的process表，对于每个被检查的进程，调用Process::IsSleepOn(chan)函数检查进程控制块中p\_wchan成员，查看该进程是否因为chan原因处于睡眠，如果是的话，则调用SetRun()函数实施具体唤醒进程操作，Setrun()的工作过程见图7.8。

#### 图7.7 WakeUpAll(wchan) 图7.8 SetRun()

* 关于runout标志

runout是盘交换区上没有进程图像可以调入内存的标志。当0#进程发交换区上已经没有任何可被调入内存的进程图像时，也就是没有进程处于就绪(SRUN)状态并且图像驻留在交换区上，于是将runout标志置为1，0#进程进入睡眠状态。当某进程转入就绪(SRUN)状态，且其图像在交换区上时，应将0#进程唤醒，由它将交换区上的图像调入内存。

## 进程终止和父、子进程同步

在用户态程序中，如果一个用系统调用fork创建的子进程希望终止自己，那就应该使用系统调用exit。内核中执行系统调用exit程序的主体部分是程序exit()。它使调用它的进程进入SZOMB状态，并等待父进程作善后处理。

在用户态程序中，父进程可以用系统调用wait等待其子进程终止。内核中实施系统调用wait的程序为wait()。它负责对于SZOMB状态的子进程进行善后处理。

### 进程终止—exit()

进程自我终止时，除暂时保留内存中进程控制块Process对象和写到盘交换区上的ppda区副本外，放弃它战胜的一切资源，如内存、进程打开文件，递减对当前工作目录和共享正文段的引用计数，清空信号处理函数。在执行完exit()之后，该进程处于SZOMB状态，等待父进程作善后处理。

****

#### 图7.9 exit()

在用户程序中，通过使用系统调用exit(status)来终止进程，参数status 是终止进程向父进程传递的参数，父进程用wait取得该参数。

### 父进程等待子进程终止

父进程用系统调用wait(status)等待它的一个子进程终止，这是内核中实现父、子进程同步的主要方式。如果在使用wait前已经有一个子进程结束了，则对其作简短善后处理后就立即返回；如果它没有子进程，立即返回-1。系统调用wait的返回值有两个：一个通过返回值返回，一个通过变量status返回，其值是子进程用系统调用exit(status)终止时，希望传给父进程的参数status。

内核中实施系统调用wait的程序为wait()，其流程如下(图7.10)。



#### 图7.10 wait()

从图中可以看出，善后处理主要包括两部分：其一释放子进程占用的Process进程控制块，使其成为自由项；其二是将子进程的u-utime、u-cutime、u-stime、u-cstime分别加到父进程u-cutime和u-cstime上去。