

**数学与信息学院学生实验报告**

**实验课程名称：计算机操作系统 教师：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **实验项目名称** | **进程控制和线程控制** | | | **实验成绩** |  |
| **学生姓名** |  | **学 号** |  | **年级专业班级** |  |
| **小组成员** |  | | | **实验日期** |  |

# 一、实验目的

1.加深对进程概念的理解，明确进程与程序的区别

2.掌握Linux进程、线程的创建、运行和撤销的方法，进一步认识并发现执行的实质

# 二、实验软硬件环境

虚拟机VMware\_workstation\_pro\_12.5.2

CentOS-7-x86\_64-DVD-1708

win 10专业版

# 三、实验内容及要求

1.了解进程的定义以及相关知识点，尽可能全面掌握进程的实质

2.在Linux系统下进程的创建、运行和销毁

3.分析进程的生命周期和系统调用

4.了解线程的定义以及相关知识点，尽可能全面掌握线程的实质

5.在Linux系统下线程的创建、运行和销毁

6.分析线程的生命周期和系统调用

# 四、实验记录

## 1.进程

### 1.1进程的定义

进程（Process）是计算机中的程序关于某数据集合上的一次运行活动，是系统进行资源分配和调度的基本单位，**是操作系统结构的基础**。在早期面向进程设计的计算机结构中，进程是程序的基本执行实体；在当代面向线程设计的计算机结构中，进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

**操作系统引入进程的概念的原因**：从理论角度看，是对正在运行的程序过程的抽象；从实现角度看，是一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450)，目的在于清晰地刻画[动态系统](https://baike.baidu.com/item/%E5%8A%A8%E6%80%81%E7%B3%BB%E7%BB%9F)的内在规律，有效管理和调度进入计算机系统[主存储器](https://baike.baidu.com/item/%E4%B8%BB%E5%AD%98%E5%82%A8%E5%99%A8)运行的程序。

### 1.2进程的特征

动态性：进程的实质是程序在[多道程序系统](https://baike.baidu.com/item/%E5%A4%9A%E9%81%93%E7%A8%8B%E5%BA%8F%E7%B3%BB%E7%BB%9F)中的一次执行过程，进程是动态产生，动态消亡的。

并发性：任何进程都可以同其他进程一起并发执行

独立性：进程是一个能独立运行的基本单位，同时也是系统分配资源和调度的独立单位；

[异步性](https://baike.baidu.com/item/%E5%BC%82%E6%AD%A5%E6%80%A7)：由于进程间的相互制约，使进程具有执行的间断性，即进程按各自独立的、不可预知的速度向前推进

结构特征：进程由程序、数据和[进程控制块](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E6%8E%A7%E5%88%B6%E5%9D%97" \t "_blank)(PCB)三部分组成。

多个不同的进程可以包含相同的程序：一个程序在不同的数据集里就构成不同的进程，能得到不同的结果；但是执行过程中，程序不能发生改变。

### 1.3Linux系统的进程类型

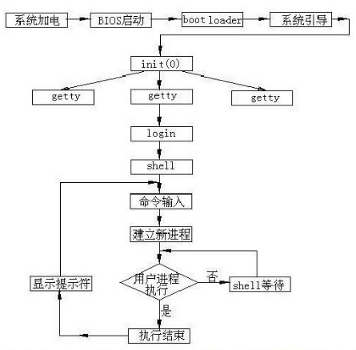
交互进程：shell启动的进程，可以在前台或后台运行

批处理进程：进程序列，不与特定终端关联

守护进程：后台服务进程，独立于控制终端，常在系统引导装入时启动，在系统关闭时终止

### 1.4Linux进程的派生关系

Linux用进程树进行进程管理。0号进程（也称交换进程）完成进程调度和进程在主存与磁盘上进行交换。1号进程（内核态）负责执行内核的部分初始化工作及系统配置，并创建若干个用于高速缓存和虚拟主存管理的内核线程。1号进程调用execve()运行可执行程序init，演变成用户态1号进程，即init进程，完成系统启动工作。



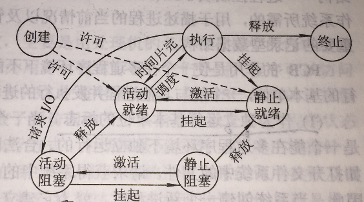
Linux进程派生关系

### 1.5进程的状态

多进程即在状态转换中有多个进程，CPU中同一时间恒仅只有一个进程。

进程的状态有：新建态、结束态、执行态、活动就绪态、活动阻塞态、静止阻塞态和静止就绪态。

多进程状态转换实现：①就绪队列\*1 + 阻塞队列\*1 ②调度策略：FIFO



进程的状态

### 1.6Linux进程管理命令

#### 1.6.1进程终止

终止一个进程或终止一个正在运行的程序：

①kill命令：根据PID向进程发送信号，缺省操作时停止进程。

②如果进程启动了子进程，只终止父进程，子进程运行中任消耗资源成为“僵尸”进程，可用kill -9强制终止退出。

③pkill命令：终止同一进程组内的所有进程。允许指定要终止的进程名称，而非PID。

④killall命令：与pkill应用方法类似，直接杀死运行中的程序。

**PS.**数据库服务器的父进程不能用这些命令杀死（容易产生个鞥多的文件碎片呆滞数据库崩溃）

#### 1.6.2进程优先级设置

进程优先级nice值，进程在计划执行时被赋予，默认值为0，范围从-20（最有利的调度优先级）到19（最不利的调度优先级）

进程优先级设置：

①nice命令：使用比默认nice值更高或者更低的优先级运行进程。

②renice命令：调整已经运行的进程的优先级。

**PS.**root用户可以提高或降低任意进程的nice值，普通用户智能增加进程优先级的值（降低优先级）

### 1.7进程的创建、运行和销毁

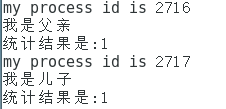
#### 1.7.1创建进程

fork（）产生当前进程的拷贝。系统先给新的进程分配资源，例如存储数据和代码的空间。然后把原来的进程的所有值都复制到新的新进程中，只有少数值与原来的进程的值不同。相当于克隆了一个自己。

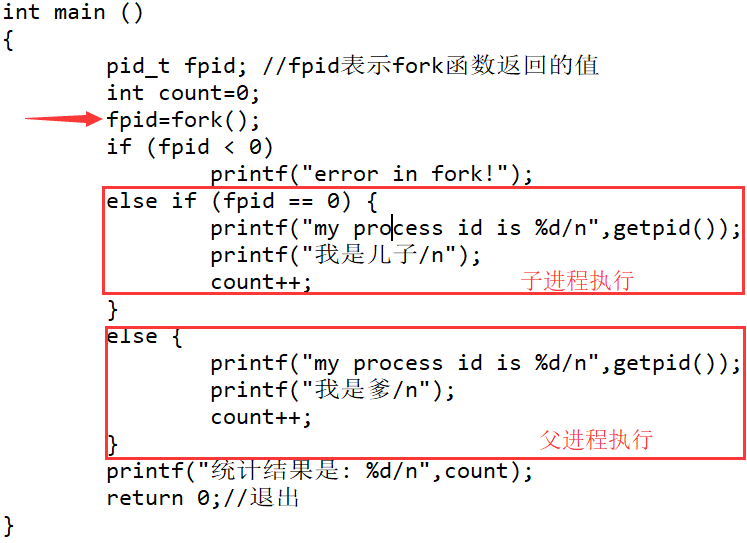
代码：

|  |
| --- |
| #include<unistd.h>  #include<stdio.h>  #include<stdlib.h>  int main()  {  pid\_t fpid;  int count=0;  fpid=fork();  if(fpid<0)  {  printf("error in fork!");  }  else if (fpid==0)  {  printf("my process id is %d\n",getpid());  printf("我是儿子\n");  count++;  }  else  {  printf("my process id is %d\n",getpid());  printf("我是父亲\n");  count++;  }  printf("统计结果是:%d\n",count);  return 0;  } |

运行结果：



#### 1.7.2进程的运行



进程的运行

#### 1.7.3进程的销毁/退出

正常退出：在main（）函数中执行return、调用exit（）函数或者\_exit()函数

异常退出:调用abort（）函数、进程收到信号而终止

区别：exit是一个函数，有参数，把控制权交给系统，return是函数执行完后的返回，将控制权交给调用函数。exit是正常终止进程，abort是异常终止。exit（）在头文件stdlib.h中声明，先执行清楚操作，在将控制权返回给内核，而\_exit（）在头文件unistd.h中生命，执行后立即返回给内核。

### 1.8进程的系统分析

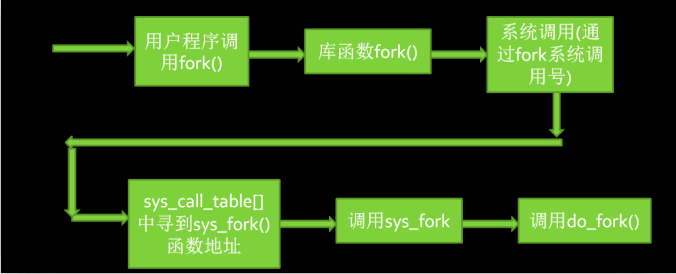
#### 1.8.1创建进程分析

当在用户态调用fork()函数的时候，系统的内核是如何执行这个函数的，子进程在内核是从哪里执行的？他的堆栈有哪些变化呢？当然说道进程，那么进程控制块PCB我们肯定是要了解的。

为了描述和控制进程的运行，系统为每一个进程定义了一个数据结构——**进程控制块**。它是进程实体的一部分，是操作系统中最重要的记录型数据结构。或者说，OS是根据PCB来对并发程序的进程进行控制和管理的。总而言之，PCB是**进程存在的唯一标志**。进程控制块中的信息包括进程标识符、处理机状态、进程调度信息、进程控制信息。然而PCB在linux中具体实现是 task\_struct数据结构，下文有给出。

Linux下用于创建进程的API有三个fork,vfork和clone，这三个函数分别是通过系统调用sys\_fork,sys\_vfork以及sys\_clone实现的

（这里目前讨论的都是基于x86架构的）。而且这三个系统调用，都是通过do\_fork来实现的，只是传入了不同的参数。所以我们可以得出结论：所有的子进程是在do\_fork实现创建和调用的。下面我们就来整理一下整个进程的在用户态到内核态的过程是怎么样的。fork系统调用如下：



fork系统调用图解

下面我们来重点看看do\_fork的代码。[http://codelab.shiyanlou.com/xref/linux-3.18.6/kernel/fork.c#do\_fork](http://codelab.shiyanlou.com/xref/linux-3.18.6/kernel/fork.c%23do_fork)

|  |
| --- |
| long do\_fork(unsigned long clone\_flags,  unsigned long stack\_start,  unsigned long stack\_size,  int \_\_user \*parent\_tidptr,  int \_\_user \*child\_tidptr)  {  struct task\_struct \*p;  int trace = 0;  long nr;  /\*  \* Determine whether and which event to report to ptracer. When  \* called from kernel\_thread or CLONE\_UNTRACED is explicitly  \* requested, no event is reported; otherwise, report if the event  \* for the type of forking is enabled.  \*/  if (!(clone\_flags & CLONE\_UNTRACED)) {  if (clone\_flags & CLONE\_VFORK)  trace = PTRACE\_EVENT\_VFORK;  else if ((clone\_flags & CSIGNAL) != SIGCHLD)  trace = PTRACE\_EVENT\_CLONE;  else  trace = PTRACE\_EVENT\_FORK;  if (likely(!ptrace\_event\_enabled(current, trace)))  trace = 0;  }  p = copy\_process(clone\_flags, stack\_start, stack\_size,  child\_tidptr, NULL, trace);  /\*  \* Do this prior waking up the new thread - the thread pointer  \* might get invalid after that point, if the thread exits quickly.  \*/  if (!IS\_ERR(p)) {  struct completion vfork;  struct pid \*pid;  trace\_sched\_process\_fork(current, p);  pid = get\_task\_pid(p, PIDTYPE\_PID);  nr = pid\_vnr(pid);  if (clone\_flags & CLONE\_PARENT\_SETTID)  put\_user(nr, parent\_tidptr);  if (clone\_flags & CLONE\_VFORK) {  p->vfork\_done = &vfork;  init\_completion(&vfork);  get\_task\_struct(p);  }  wake\_up\_new\_task(p);  /\* forking complete and child started to run, tell ptracer \*/  if (unlikely(trace))  ptrace\_event\_pid(trace, pid);  if (clone\_flags & CLONE\_VFORK) {  if (!wait\_for\_vfork\_done(p, &vfork))  ptrace\_event\_pid(PTRACE\_EVENT\_VFORK\_DONE, pid);  }  put\_pid(pid);  } else {  nr = PTR\_ERR(p);  }  return nr;  } |

整段代码挺长，涉及到很多工作的处理，但是整个创建新进程是在上述代码的第29行copy\_process()这个函数实现的。我们前面已经说过，子进程是通过复制实现的。为了探个究竟，我们进入到copy\_process()这个函数体里可以看到几个很重要的函数，列举如下：

复制一个PCB——task\_struct：

|  |
| --- |
| p = dup\_task\_struct(current); |

复制当前进程的PCB描述符task\_struct。我们在进入到该函数dup\_task\_struct体内就可以看到这个pcb是如何复制的。主要的赋值函数是

|  |
| --- |
| err = arch\_dup\_task\_struct(tsk, orig);//这一句是赋值操作 |

当然在dup\_task\_struct函数体内还有其他的一次辅助操作例如：

|  |
| --- |
| tsk = alloc\_task\_struct\_node(node);  ti = alloc\_thread\_info\_node(tsk, node);  tsk->stack = ti;  setup\_thread\_stack(tsk, orig);//这里只是复制thread\_info，而非复制内核//堆栈 |

然而我们再 往dup\_task\_struct(current)函数往下看，后面是大量的修改进程的内容，也就是对复制过来的东西修改为子进程所拥有的数据。也就是初始化一个子进程。我们再往下看，在copy\_process()函数[http://codelab.shiyanlou.com/xref/linux-3.18.6/kernel/fork.c#copy\_process](http://codelab.shiyanlou.com/xref/linux-3.18.6/kernel/fork.c" \l "copy_process)的第1396行有一个非常重要的函数copy\_thread，我们可以来看一看它有什么作用，下面是其一部分重要代码：

|  |
| --- |
| struct pt\_regs \*childregs = task\_pt\_regs(p);  struct task\_struct \*tsk;  int err;  p->thread.sp = (unsigned long) childregs;  p->thread.sp0 = (unsigned long) (childregs+1);  memset(p->thread.ptrace\_bps, 0, sizeof(p->thread.ptrace\_bps));  if (unlikely(p->flags & PF\_KTHREAD)) {  /\* kernel thread \*/  memset(childregs, 0, sizeof(struct pt\_regs));  p->thread.ip = (unsigned long) ret\_from\_kernel\_thread;  task\_user\_gs(p) = \_\_KERNEL\_STACK\_CANARY;  childregs->ds = \_\_USER\_DS;  childregs->es = \_\_USER\_DS;  childregs->fs = \_\_KERNEL\_PERCPU;  childregs->bx = sp; /\* function \*/  childregs->bp = arg;  childregs->orig\_ax = -1;  childregs->cs = \_\_KERNEL\_CS | get\_kernel\_rpl();  childregs->flags = X86\_EFLAGS\_IF | X86\_EFLAGS\_FIXED;  p->thread.io\_bitmap\_ptr = NULL;  return 0;  }  \*childregs = \*current\_pt\_regs();//拷贝父进程的内核堆栈栈底，也就是已有//的内核堆栈数据的拷贝  childregs->ax = 0;//给eax赋值为0，因为子进程返回的是0，系统调用是通//过eax返回的  if (sp)  childregs->sp = sp;//修改栈顶  p->thread.ip = (unsigned long) ret\_from\_fork;//给ip赋值，这就是子进程//执行的起点 |

从用户态的代码看fork();函数返回了两次，即在父子进程中各返回一次，父进程从系统调用中返回比较容易理解，子进程从系统调用中返回，那它在系统调用处理过程中的哪里开始执行的呢？这就涉及子进程的内核堆栈数据状态和task\_struct中thread记录的sp和ip的一致性问题，这是在哪里设定的？copy\_thread in copy\_process

|  |
| --- |
| struct pt\_regs \*childregs = task\_pt\_regs(p);  \*childregs = \*current\_pt\_regs(); //复制内核堆栈栈底  childregs->ax = 0; //为什么子进程的fork返回0，这里就是原因！  p->thread.sp = (unsigned long) childregs; //调度到子进程时的内核栈顶  p->thread.ip = (unsigned long) ret\_from\_fork; //调度到子进程时的第一条//指令地址 |

上面赋值复制的内核堆栈并不是父进程的所有内核堆栈的内容，那复制的是哪些部分呢？我们可以看上面代码的第一句，其中复制的内容就是pt\_regs里面的内容。里面的代码如下：

|  |
| --- |
| struct pt\_regs {  long ebx;  long ecx;  long edx;  long esi;  long edi;  long ebp;  long eax;  int xds;  int xes;  int xfs;  int xgs;  long orig\_eax;  long eip;  int xcs;  long eflags;  long esp;  int xss;  }; |

父进程堆栈复制给子进程的就是上面那些参数。从copy\_thread中我们就已经得出堆栈复制和子进程开始执行的起始地方。综上所述，我们对整个do\_fork的分析到此就可以告一段落了。我们在回过头来总结一下，do\_fork()的实现，主要是靠copy\_process()完成的，这就是一环套一环。

**整个过程实现如下：**

p = dup\_task\_struct(current);　为新进程创建一个内核栈、thread\_iofo和task\_struct,这里完全复制父进程的内容，所以到目前为止，父进程和子进程是没有任何区别的。为新进程在其内存上建立内核堆栈。对子进程task\_struct任务结构体中部分变量进行初始化设置，检查所有的进程数目是否已经超出了系统规定的最大进程数，如果没有的话，那么就开始设置进程描诉符中的初始值，从这开始，父进程和子进程就开始区别开了。把父进程的有关信息复制给子进程，建立共享关系。设置子进程的状态为不可被TASK\_UNINTERRUPTIBLE，从而保证这个进程现在不能被投入运行，因为还有很多的标志位、数据等没有被设置。复制标志位（falgs成员）以及权限位(PE\_SUPERPRIV)和其他的一些标志。调用get\_pid()给子进程获取一个有效的并且是唯一的进程标识符PID。return ret\_from\_fork;返回一个指向子进程的指针，开始执行。

#### 1.8.2进程的退出以及销毁

**进程的退出：**

当一个进程运行完毕或者因为触发系统异常而退出时，最终会调用到内核中的函数do\_exit(),在do\_exit()函数中会清理一些进程使用的文件描述符，会释放掉进程用户态使用的相关的物理内存，清理页表，同时进程会调整其子进程的父子关系，会根据实际的情况向父进程发送SIG\_CHLD信号。

下面是经过简化的内核代码：

|  |
| --- |
| fastcall NORET\_TYPE void do\_exit(long code)  {  struct task\_struct \*tsk = current;  int group\_dead;  //设置进程的状态为pf\_exiting  tsk->flags |= PF\_EXITING;  //从定时器队列中删除该进程  del\_timer\_sync(&tsk->real\_timer);  //当前进程需要被trace相应的exit，将该进程的exit事件通过信号  //发送给父进程，也就是trace 进程  if (unlikely(current->ptrace & PT\_TRACE\_EXIT)) {  current->ptrace\_message = code;  ptrace\_notify((PTRACE\_EVENT\_EXIT << 8) | SIGTRAP);  } |

下面的几个exit是将进程同各个描述符分离，主要有内存描述符，信号量，文件描述符，文件系统，命名空间，若相关描述符不再有任何进程使用，会释放掉，后面会分析一下\_\_exit\_mm()和\_\_exit\_files()函数

|  |
| --- |
| \_\_exit\_mm(tsk);  exit\_sem(tsk);  \_\_exit\_files(tsk);  \_\_exit\_fs(tsk);  exit\_namespace(tsk);  exit\_thread();  exit\_keys(tsk);  if (group\_dead && tsk->signal->leader)  disassociate\_ctty(1);  //exit\_code中存放进程的退出码  tsk->exit\_code = code;  //调整进程子进程的父子关系，向相关进程发出SIG\_CHLD信号  exit\_notify(tsk);  //进程处于zombie或者dead状态，在此调用schedule,该进程就永远回不来了  schedule();  for (;;) ;  }  static inline void \_\_exit\_mm(struct task\_struct \* tsk)  {  struct mm\_struct \*mm = tsk->mm;  //对于vfork来说，父进程会等待，直到子进程退出，在这里唤醒父进程  mm\_release(tsk, mm);  if (!mm)  return;  /\* more a memory barrier than a real lock \*/  task\_lock(tsk);  //将进程的内存描述符设为空  tsk->mm = NULL;  up\_read(&mm->mmap\_sem);  //使当前的cpu进入懒惰tlb模式  enter\_lazy\_tlb(mm, current);  task\_unlock(tsk);  //在这里真正的去释放内存描述符及相关所属资源  mmput(mm);  }  void mmput(struct mm\_struct \*mm)  {  //在多线程的情况下，可能多个线程会共享同一个进程描述符，mm\_users就是//指明了有多少个线程正在使用该描述符  if (atomic\_dec\_and\_test(&mm->mm\_users)) {  exit\_aio(mm);  //没有进程使用该内存描述符了，应该可以释放掉该内存描述符所描述的一些进//程用户态空间内存，并释放掉所有的vm\_area\_struct  exit\_mmap(mm);  if (!list\_empty(&mm->mmlist)) {  spin\_lock(&mmlist\_lock);  list\_del(&mm->mmlist);  spin\_unlock(&mmlist\_lock);  }  //释放掉交换标记  put\_swap\_token(mm);  //释放掉内存描述符和pgd表，释放掉内存描述符  mmdrop(mm);  }  }  static void exit\_notify(struct task\_struct \*tsk)  {  int state;  struct task\_struct \*t;  struct list\_head ptrace\_dead, \*\_p, \*\_n;  write\_lock\_irq(&tasklist\_lock);  INIT\_LIST\_HEAD(&ptrace\_dead);  //改变进程子进程的父子关系  forget\_original\_parent(tsk, &ptrace\_dead);  t = tsk->real\_parent;  if ((process\_group(t) != process\_group(tsk)) &&  (t->signal->session == tsk->signal->session) &&  will\_become\_orphaned\_pgrp(process\_group(tsk), tsk) &&  has\_stopped\_jobs(process\_group(tsk))) {  \_\_kill\_pg\_info(SIGHUP, (void \*)1, process\_group(tsk));  \_\_kill\_pg\_info(SIGCONT, (void \*)1, process\_group(tsk));  }  if (tsk->exit\_signal != SIGCHLD && tsk->exit\_signal != -1 &&  ( tsk->parent\_exec\_id != t->self\_exec\_id ||  tsk->self\_exec\_id != tsk->parent\_exec\_id)  && !capable(CAP\_KILL))  tsk->exit\_signal = SIGCHLD;  //线程为组长线程且线程组已经空了，这个时候可以通知父进程sigchild消息了  //exit\_signal为-1只有在其为非组长线程的线程的情况下才发生  if (tsk->exit\_signal != -1 && thread\_group\_empty(tsk)) {  int signal = tsk->parent == tsk->real\_parent ? tsk->exit\_signal : SIGCHLD;  do\_notify\_parent(tsk, signal);  } else if (tsk->ptrace) {  do\_notify\_parent(tsk, SIGCHLD);  }  state = EXIT\_ZOMBIE;  //对于线程而言，线程若不为trace进程trace的话，可以直接  //将exit\_state置位exit\_dead,对于单线程进程，exit\_state为EXIT\_DEAD  if (tsk->exit\_signal == -1 && tsk->ptrace == 0)  state = EXIT\_DEAD;  tsk->exit\_state = state;  /\*  \* Clear these here so that update\_process\_times() won't try to deliver  \* itimer, profile or rlimit signals to this task while it is in late exit.  \*/  tsk->it\_virt\_value = 0;  tsk->it\_prof\_value = 0;  write\_unlock\_irq(&tasklist\_lock);  list\_for\_each\_safe(\_p, \_n, &ptrace\_dead) {  list\_del\_init(\_p);  t = list\_entry(\_p,struct task\_struct,ptrace\_list);  release\_task(t);  }  /\* If the process is dead, release it - nobody will wait for it \*/  //对于非组长线程的线程，对其进行清理，对于组长  //线程的清理则是在发送了sig\_chld信号后，由其父进程  //进行清理  if (state == EXIT\_DEAD)  release\_task(tsk);  /\* PF\_DEAD causes final put\_task\_struct after we schedule. \*/  preempt\_disable();  tsk->flags |= PF\_DEAD;  } |

从exit\_notify代码中，我们可以看出发送SIG\_CHLD信号的条件：

①对于单线程进程，当进程退出就发送sig\_chld信号。

②对多线程进程，当线程组无其他线程时，才会发送sig\_chld信号，发送sig\_chld信号主要是为了让父进程处理回收组长进程，普通的非组长线程自己会把自己清理掉的。组长线程退出且线程组无其他线程。非组长线程退出且线程组无其他线程。

③进程被trace。 线程组组长进程是最后一个被撤销处理掉的。

**进程的撤销：**

当进程终止运行后，一般会处于僵死状态，需要由父进程来执行wait操作来回收进程的进程描述符及内核栈所占内存，同时把僵死进程从进程相关的各个表上摘除下来。对应的处理函数是release\_task(),该函数的处理过程是：

①递减进程拥有者进程的个数， atomic\_dec(&p->user->processes);

②如果进程被跟踪，把它从调试程序的ptrace\_children链表中删除。\_\_ptrace\_unlink(p);

③调用\_\_exit\_signal()删除所有的挂起信号并释放signal\_struct描述符，若没有其它轻量级进程使用该signal\_struct的话，会删除该数据结构: \_\_exit\_signal(p);

④调用\_\_exit\_sighand()删除信号处理函数： \_\_exit\_sighand(p);

⑤调用\_\_unhash\_process()，变量nr\_threads减1。两次调用detach\_pid(),分别从PIDTYPE\_PID和PIDTYPE\_TGID类型的PID散列表中删除进程描述符。 如果进程是线程组的领头进程，那么调用两次detach\_pid()从PIDTYPE\_PGID, PIDTYPE\_SID类型的散列表中删除该进程描述符。用RMOVE\_LINKS从进程链表中删除该进程。

⑥如果进程不是线程组的领头进程，领头进程处于僵死状态，且该进程时线程组中的最后一个成员，该进程向领头进程的父进程发出一个信号，通知该进程已经死亡。

⑦调用sched\_exit()函数来调整父进程的时间片。

⑧调用put\_task\_struct()递减进程描述符的引用计数，当计数器变为0时，则函数终止所有残留的对进程的引用。递减进程所有者的user\_struct 数据结构的使用计数，如果该引用计数变为0，释放该结构。释放进程描述符以及thread\_info描述符和内核态堆栈。

## 2.线程

### 2.1线程的定义

线程有时候又被称为轻量级进程，是程序执行的最小单元。一个进程可对应多个线程，而一个线程只属于一个进程。进程的执行是以线程为单位进行得，比如说一个简单“hello world”程序只有一个线程，就是main（）函数对应的线程。

线程是程序中一个单一的顺序控制流程。进程内一个相对独立的、可调度的执行单元，是系统独立调度和分派CPU的基本单位指运行中的程序的调度单位。在单个程序中同时运行多个线程完成不同的工作，称为多线程。

线程中的实体基本上不拥有系统资源，只是有一点必不可少的、能保证独立运行的资源。（比如，线程控制块TCB，是用于指示被执行指令序列的程序计数器、保留局部变量、少数状态参数和返回地址等的一组寄存器和堆栈。）

### 2.2线程的构成

线程ID。用于标实线程。

当前指令指针PC。标明下一指令执行点。

寄存器集合和堆栈。该线程的可用空间。

### 2.3多线程

大多数软件应用中，线程的数量都不止一个。多线程可用并发的执行，并共享进程的全局便来那个和堆的数据。

#### 2.3.1多线程优势：

某个操作可能会陷入长时间的等待。采用多线程，当一个线程等待的时候，可执行其他线程，充分利用cpu。

某操作（如计算）可能会消耗大量时间，而导致和用户之间的交互中断。多线程可让一个线程负责计算，另一线程负责交互。

软件本身就要求并发操作，如多端下载软件

多核cpu，本身就具备同时执行多个线程的能力

### 2.4线程的调度

最好的情况是：当处理器数量大于要处理的线程数目的时候，所有线程都可以同时执行。实现真正意义上的并发。而这种情况在现实中基本不可能。现实中的并发只是一种模拟出来的状态，特别是在单核处理对于多线程的时候。它通过让多个线程交替执行，每个线程执行很短时间，从表面上看，这些线程同时执行，实现并发。每个线程都想被执行，但是每次执行的线程数量是有限的，所以就要有一种方法来从众多的线程中选出要执行的线程，现在讨论下单核的情况，多核的类似。在操作系统中有专门的线程调度算法来实现，下面列几个简单的“调度算法”。

#### 2.4.1线程的调度算法

**“先进先出”策略。**所有的线程组成一个队列，新生的线程加入到队列末尾，每次取队头执行。有一个缺陷就是，如果新生成的线程是紧急操作，需要操作系统尽快相应，这种调度方法就不能满足了。

**按优先级调度。**每个线程都有自己的优先级，并且是可以被操作系统修改的，调度时候每次选取优先级最高的执行。这种方法弥补了上一种方法的缺陷。对于需要及时相应的紧急事件，可以给他一个高优先级，这样就能在下次被调度。然而这种方法也有一个问题，也就是所谓的饥饿。如果某线程“看似”无关紧要，被给予一个低得优先级，以后每次产生的线程优先级都比他高，那么这个线程会一直得不到执行，成为饿死。一个解决的办法是随之事件的推移而提升线程的优先级。这样只要事件足够长，低优先级的线程也会获得高优先级而被执行。

从上面的分析来开，线程似乎有两个状态：执行和不执行（等待）。其实操作系统中的每个线程都对应三个状态。

#### 2.4.2线程的状态

上面说了线程调度的问题，只有准备就绪的线程（这种线程称为就绪的线程），才能被调度。调度以后，线程就在处理器中被执行，这时线程的状态为运行时。如果该线程在等待某种事件的发生（如响应I/O），这种状态成为等待。

总结下线程的五种状态：

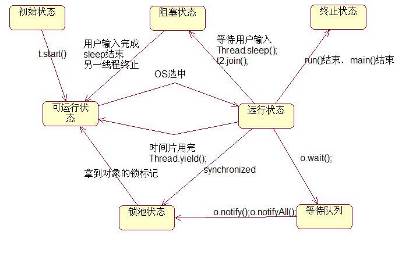
**新建状态(New)**：新创建了一个线程对象。

**就绪状态(Runnable)**：线程对象创建后，其它线程调用了该对象的start()方法。该状态的线程位于可执行线程池中，变得可执行，等待获取CPU的使用权。

**执行状态(Running)**：就绪状态的线程获取了CPU。执行程序代码。

**堵塞状态(Blocked)**：堵塞状态是线程由于某种原因放弃CPU使用权。临时停止执行。直到线程进入就绪状态，才有机会转到执行状态。

**死亡状态(Dead)**：线程运行完了或者因异常退出了run()方法，该线程结束生命周期。



线程状态转换图解

### 2.5线程安全

多线程并发时，在访问数据方面会出现一些问题。特别是当多个线程访问同一个变量的时候。

下面将用一个例子来说明可能出现的问题：

线程A、线程B都对变量X进行操作，操作顺序如下：

线程A对X赋值

线程A对X自加

线程B使用X的值（比如说把它赋值给另一个变量）

代码经过编译之后，在处理器中执行代码时，通常一个很简单的运算都会被分为多个步骤执行。

比如当线程A对X自加运算的时候，编译后的自加运算共分为三步，当没有执行完这三步的时候，可能线程A就会被切出（比如说有需要即时响应的操作发生）。也就是说线程A在对数据还没处理完全的时候被切出了，这样当线程B执行的时候，使用的X值将不是我们期望的值，显然，发生了错误。

**解决策略：**

上面问题的出现本质上是因为一个不应该被打断的操作被强行中断了，那么有一种解决的办法就是设置一种规定一些操作，在执行的时候不能被中断。这样就避免了操作还没完成就被换出的情况。把这些简单的操作称为原子的操作。windows中对于这种操作也有支持。

但是这种策略只适用于简单的情况。对于复杂的情况，我们用一种称为同步与锁的机制来实现。

**同步与锁**

简单的说，就是在一个线程对数据访问结束之前，其他线程不能对这个数据进行访问。这样的话，对数据的访问就原子化了。

这种机制的实现也很简单：每个线程对数据访问的时候都会尝试获取锁，当访问结束后释放。在获取锁的时候如果有线程在访问数据，就会获取失败，这时候线程会等待， 直到访问数据的那个线程释放锁。

**二元信号量**

是最简单的一种锁，它只有两个状态：占用与非占用。它用于只能被一个线程访问的资源。只有资源状态为非占用 的时候，才能被线程获取，获取之后修改资源状态为占用，访问结束后修改资源状态为非占用。

**信号量**

稍微复杂一些，他适用于可以被多个线程同时访问的资源。一个初始值为N=n的信号量能被n个线程同时访问。当要访问数据的时候，先查看N值，（N的值代表还有多少个线程能访问资源）如果N值大于0，该线程能访问资源，线程进入后把N值减一。当访问结束后N值+1。如果信号量的值小于0，则进入等待状态。

**互斥量**

和二元信号相似，资源只能被一个线程访问，但是同一个信号量只能被获取该信号量的线程释放，也就是说对于二元信号量，同一个互斥量可以被别的（任意）线程释放。相对二元信号量来说更严格了。

**临界区**

是比互斥量更严格的同步手段。临界区和上面的区别在于，互斥量和信号量在任何进程都是可见的，也就是说，一个进程创建了互斥量和信号量，在其他进程都是可见的，而临界区的作用范围仅限于本进程。

**读写锁**

读写锁用于更加通用的场合。对于同一个数据，多个进程同时读是没问题的，但是如果有线程要对数据进行修改，就要使用同步手段来避免出错。对于同一个读写锁，有两种获取方式：

**共享的。**对数据只进行读操作，可以多个线程同时进行。

**独占的。**会修改数据，在修改完成之前，不能有其他线程操作数据。

当锁处于自由状态时，以任何一种方式获取锁都会成功，并将锁至于相应的状态。如果锁处于共享状态，那么其他与以共享方式获取锁的线程都能成功，共同读数据。对于独占式获取的，则要等到以共享方式获取的所有线程释放后(锁重新回到自由状态)才能获取。并且对于以独占方式获取的锁，其他任何对锁的请求都不会成功。

**条件变量**

条件变量类似于一个发令枪，可以有多个线程等待枪响，枪响的时候，这些等待枪响的线程会同时恢复执行。发令枪何时响也可由线程来决定。也就是说，条件变量可以让多个线程等待某件事的发生，当时间发生时（条件变量被唤醒），所有的线程可以一起恢复执行。

### 2.6 pthread函数库的安装（也可以不安装，下文代码案例即为不安装示例）

pthread不是Linux下的默认的库，所以在linux环境下，有的有些系统没有安装有pthread，所以在编译有关多线程程序时，也就是在程序链接的时候，无法找到pthread库中函数的入口地址，于是链接会失败。所以在此我们必须先安装好pthread库。安装过程很简单使用下面命令即可。

sudo apt-get install glibc-doc

sudo apt-get install manpages-posix-dev

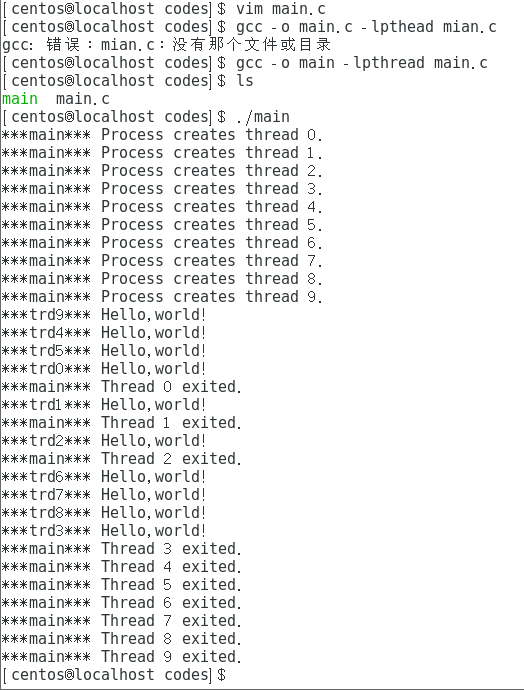
然后再用man -k pthread就可以查找到当前manpages中关于pthread的手册。

### 2.7线程的创建、运行、销毁

**源代码：**

|  |
| --- |
| #include<pthread.h>  #include<stdio.h>  #include<stdlib.h>  #include<assert.h>  #define NUM\_THREADS 10  void \* perform(void \* argu)  {  int value\_of\_argu;  value\_of\_argu = \*((int \*)argu);  printf("\*\*\*trd%d\*\*\* Hello,world!\n",value\_of\_argu);  return NULL;  }  int main(int argc, char \*\* argv)  {  pthread\_t threads[NUM\_THREADS];  int thread\_args[NUM\_THREADS];  int value\_of\_result = 0;  int index = 0;  for(index = 0; index < NUM\_THREADS; index++)  {  thread\_args[index] = index;  printf("\*\*\*main\*\*\* Process creates thread %d.\n",index);  value\_of\_result = pthread\_create(&threads[index],NULL,perform,&thread\_args[index]);  assert(!value\_of\_result);  }  for(index = 0; index < NUM\_THREADS; index++)  {  value\_of\_result = pthread\_join(threads[index],NULL);  assert(!value\_of\_result);  printf("\*\*\*main\*\*\* Thread %d exited.\n",index);  }  return 0;  } |

**运行截图：**



### 2.8线程的系统调用分析

在代码中我用到了两个在多线程程序中经常用到的两个函数：创建线程函数pthread\_create()和阻塞线程函数pthread\_join()。

**pthread\_create(pthread\_t thread, const pthread\_attr\_t \*attr, void (start\_routine)(void\*), void \*arg);**

start\_routine为新线程的入口函数，arg为传递给start\_routine的参数。

每个线程都有自己的线程ID，函数第一个参数为线程id，以便在进程内区分。线程ID在pthread\_create调用时回返给创建线程的调用者。

**pthread\_join (pthread\_t thread, void\*\* threadreturn);**

如果一个线程要等待另一个线程的终止，可以使用pthread\_join函数，该函数的作用是调用pthread\_join的线程将被挂起直到线程ID为参数thread的线程终止。第一个参数为线程id。

Linux中线程的创建以及运行、销毁其实和进程的系统调用异曲同工。

Linux中的thread 是用sys\_clone系统调用完成的。本质上说，每个线程其实就是一个进程，只不过线程共享部分资源。操作系统对线程的管理，实则就是对进程的管理。在/proc目录下我们可以找到这些进程。Linuxthread 是通过 \_\_pthread\_manager\_thread 管理其他线程的。线程有各自的线程id，由LinuxThreads分配和维护。其实也有各自的真实pid，但由结构隐藏了，对外的pid为主进程的pid。线程对操作系统来说就是进程，因此对信号的处理是以进程为单位分发的。并且线程总数受限于系统进程数，线程之间的调度是由内核调度器来处理的。

# 五、实验小结（心得体会、遇到的问题及解决方式、未解决/需进一步研讨的问题或建议新实验方法等）

**回顾：**

linux创建一个新的进程是从复制开始的，在系统内核里首先是将父进程的进程控制块PCB进行拷贝，然后再根据自己的情况修改相应的参数，获取自己的进程号，再开始执行。我觉得整个过程重点就是理解子进程如何创建，在内核调用的几个重要的内核函数，以及子进程怎么返回开始执行的。

线程的实现有两种方式，一是继承Thread类，二是实现Runnable接口，但无论如何，当我们new了这个对象后。线程就进入了初始状态；当该对象调用了start()方法，就进入可执行状态；进入可执行状态后，当该对象被操作系统选中。获得CPU时间片就会进入执行状态；进入执行状态后情况就比较复杂了。run()方法或main()方法结束后，线程就进入终止状态；当线程调用了自身的sleep()方法或其它线程的join()方法，就会进入堵塞状态（该状态既停止当前线程，但并不释放所占有的资源）。当sleep()结束或join()结束后。该线程进入可执行状态，继续等待OS分配时间片；线程调用了yield()方法，意思是放弃当前获得的CPU时间片，回到可执行状态，这时与其它进程处于同等竞争状态，

OS有可能会接着又让这个进程进入执行状态。当线程刚进入可执行状态（注意，还没执行），发现将要调用的资源被synchroniza（同步），获取不到锁标记。将会马上进入锁池状态，等待获取锁标记（这时的锁池里或许已经有了其它线程在等待获取锁标记，这时它们处于队列状态，既先到先得），一旦线程获得锁标记后，就转入可执行状态。等待OS分配CPU时间片。当线程调用wait()方法后会进入等待队列（进入这个状态会释放所占有的全部资源，与堵塞状态不同）。进入这个状态后。是不能自己主动唤醒的，必须依靠其它线程调用notify()或notifyAll()方法才干被唤醒（因为notify()仅仅是唤醒一个线程，但我们由不能确定详细唤醒的是哪一个线程。或许我们须要唤醒的线程不可以被唤醒，因此在实际使用时，一般都用notifyAll()方法，唤醒有所线程），线程被唤醒后会进入锁池。等待获取锁标记。

**问题：**

进程以及线程的销毁的系统调用实在太难理解，看了这么久还是只是对这个过程有一个模糊的概念，但是具体是怎么完成的还是一脸懵逼。相对于销毁来说，进程以及线程的创建倒是异曲同工，结构和框架以及具体的系统函数的调用也可以在网上查找到源代码，所以看的清楚一些。

线程控制块TCB概念模糊，相较于PCB来说似乎只是一个大致的抽象数据。

想用宏来制作像一些大型程序员交流网站上所看到的那种代码特定的区域，但是很可惜没有实现，便退而求其次用插入表格的方式来区分代码和文字。

# 六、参考文献

1. Always\_TDX pthread/Linux 简单多线程编程.CSDN 2015-03-17

<http://blog.csdn.net/always2015/article/details/44351501>

2.寒魔影 Linux 进程与线程一（创建-关闭线程）.博客园 2016-09-10

<http://www.cnblogs.com/zhanggaofeng/p/5860052.html>

3. phlsheji 线程的状态转换图.博客园 2016-01-04

<http://www.cnblogs.com/bhlsheji/p/5099362.html>

4. Geek\_Ling 我认识的线程.博客园 2012-02-27

<http://www.cnblogs.com/yanlingyin/archive/2012/02/27/2369120.html>

5. Always\_TDX 进程系统调用——fork函数深入理解.CSDN 2015-04-12

<http://blog.csdn.net/Always2015/article/details/45008785>

6. riskyer linux进程解析--进程的退出及销毁.博客园 2013-09-02

<http://www.cnblogs.com/riskyer/archive/2013/09/02/3297107.html>

7. yff1030 Linux 进程创建及多进程.CSDN 2012-08-12

<http://blog.csdn.net/yff1030/article/details/7857844>