1. **操作系统概念**

操作系统是裸机的第一层软件，是硬件的第一层封装与抽象

**二、操作系统基本功能**

**任务管理：**操作系统是多任务运行的，所以必须存在任务管理，不然任务之间会混乱的

**文件管理：**实现对文件的统一管理，是对文件的存储器的存储空间进行组织，分配，和回收，负责件的存储、检索、共享和保护。

**CPU管理：**分配对CPU的所有权。

**内存管理：**会根据硬件的内存给任务分配合理空间，不会让用户使用内存发生越界象。

**IO设备管理：**操作硬件不是直接操作，而是要通过IO管理模块提供的API间接性的对硬件操作。

三、**进程的几种状态**

**睡眠态：**程序已经装在内存，只是没有准备好运行。

**就绪态：**已经准备好运行，意思就是一个任务已经被操作系统配置好运行时候的所需的环境啦。

**运行态：**就是CPU正在运行的任务就是运行态。

**阻塞态：**等待一定的时间或者等待某一事件就会进入堵塞状态，如果一定时间到了或者某一事件触发了就会进入就绪态。如果一个进程50ms的中断到了还没有进入就绪态，就执行另个进程，就算这个进程优先级很高。如果这个进程50ms中断到之前就进入了就绪态，那么50ms到了应该执行优先级最高的的进程

下面从一个具体的例子来说明一个进程从运行态，阻塞态，就绪态之间的转换，这里把事件（阻塞事件）设置为一具体事件（等待键盘输入）。例程的功能是在屏幕的第三行输出 80 个“a”，当输出到 40 个时，接受一个键盘输入，键盘输入完毕后，继续在第三行输出剩下的 40 个“a”。 代码如下：

#include <toyix.h>

main()

{

int i;

for(i = 0; i < 40; i++)

{

put\_str(3,i,2,"a");

delay(80);

}

get\_char();

for(i = 40; i < 80; i++)

{

put\_str(3,i,2,"a");

delay(80);

}

}

这个主程序就是一个进程，运行代码到get\_char();然后等待按键的事件，将这个进程堵塞起来，直到这个事件响应了才回到就绪态，然后快速回到运行态。自己试验的过程中会发现好像直接从堵塞态到运行态，因为切换时间比较短，所以我们看到进程马上开始执行了，而没有看到进程在就绪态的那个状态。

#include <toyix.h>

void f1()

{

printf("aaaaa\n");

delay(90);

printf("bbbbb\n");

}

void f2()

{

printf("ccccc\n");

}

main()

{

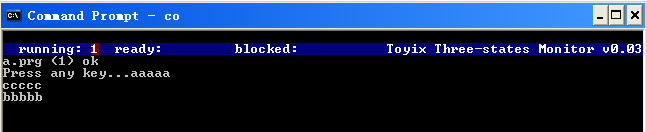
cobegin(f1,f2,0);

getch();

}

下面我来讲解下这个程序，首先main函数相当于一个主进程，这个主进程创建两个轻权进程（进程与轻权进程后面会有解释），创建之后这两个轻权进程就处于就绪状态，f1的优先级高于f2。主进程结束后，两个轻权进程也会强制结束掉，所以主进程创建轻权进程之后用等待按键按下的事件将自己堵塞，又因为toyix系统是设置时间中断，50ms中断一次，中断服务程序将会对优先级最高的就绪的进程进行调度，所以接下来就执行f1函数，打印aaaaa之后，遇到delay函数，这个轻权进程就因为等待时间而进入到堵塞态，因为延时90ms,所以50ms的中断到来后等待的时间也不会结束，所以这个时候就会进行调度就绪的f2轻权进程，打印ccccc，f2结束，这个时候f1的等待的时间基本上也到了，然后将f1从堵塞态转换为就绪态，等到50ms中断到来就会进行调度，将f1从就绪态转换为运行态，继续运行未完成的程序，打印bbbbb，这样f2就结束了，这时我们就可以触发按键，主进程就会响应事件进入就绪态，等到50ms中断到了进行任务调度，将主进程从就绪态转换为运行态，从而主进程结束，释放资源。

在f1,f2两个轻权进程运行完触发按键，结果如下：



如果你想验证在f1,f2两个轻权进程运行前触发按键后的结果，可以将延时加长一点比如：

#include <toyix.h>

void f1()

{

printf("aaaaa\n");

delay(5000);

printf("bbbbb\n");

}

void f2()

{

printf("ccccc\n");

}

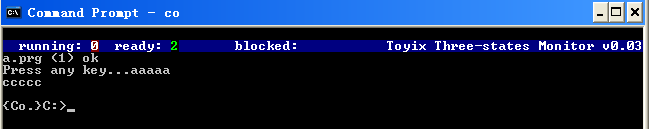
main()

{

cobegin(f1,f2,0);

getch();

}



可以看到bbbbb还没打印，也就说f1函数还没执行完，主进程结束，轻权进程也强制结束。

**挂起态：**TOYIX中50ms中断一次，正在执行的任务突然时间到，或者发生其他中断，从而进入挂起状态。

四、**什么是进程，什么是线程，信号量**

（下面的解释来自阮一峰的网络日志）

1. 计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。
2. 假定工厂的电力有限，一次只能供给一个车间使用。也就是说，一个车间开工的时候，他车间都必须停工。背后的含义就是，单个CPU一次只能运行一个任务。
3. 进程就好比工厂的车间，它代表CPU所能处理的单个任务。任一时刻，CPU总是运行一个进程，其他进程处于非运行状态。
4. 一个车间里，可以有很多工人。他们协同完成一个任务。
5. 线程就好比车间里的工人。一个进程可以包括多个线程。
6. 车间的空间是工人们共享的，比如许多房间是每个工人都可以进出的。这象征一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。
7. 可是，每间房间的大小不同，有些房间最多只能容纳一个人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。
8. 一个防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫["互斥锁"](http://zh.wikipedia.org/wiki/%E4%BA%92%E6%96%A5%E9%94%81" \t "_blank)（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。
9. 还有些房间，可以同时容纳n个人，比如厨房。也就是说，如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。
10. 这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法叫做["信号量"](http://en.wikipedia.org/wiki/Semaphore_(programming)" \t "_blank)（Semaphore），用来保证多个线程不会互相冲突。不难看出，mutex是semaphore的一种特殊情况（n=1时）。也就是说，完全可以用后者替代前者。但是，因为mutex较为简单，且效率高，所以在必须保证资源独占的情况下，还是采用这种设计。

五、**信号量和同步**

**信号量：**好比钥匙，如果设置为1就是1把钥匙，得到钥匙的任务可以访问共享资源，其他的任务只能等待得到钥匙的任务把钥匙归还才能访问其共享资源。如果设置为2就有两把钥匙。下面的程序设计验证猜想。

#include <toyix.h>

void f1()

{

int i;

for(i=1; i < 80; i++)

{

gotoxy(i,7);

delay(60);

printf("a");

}

}

void f2()

{

int i;

for(i=1; i < 80; i++)

{

gotoxy(i,9);

delay(250);

printf("b");

}

}

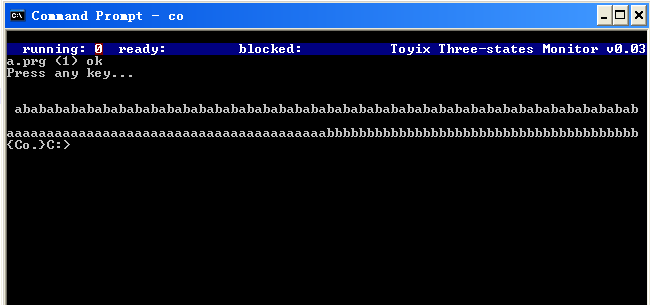
main()

{

cobegin(f1,f2,0);

getch();

}



从结果可以看到出现了混乱，两个轻权进程在争夺一个资源，没有有序同步的使用这个资源，要保证原子性，就要用到 P、V 原语。把光标看作一个资源，则设置信号量为 1，就代表有 1 个资源，使用资源时调用 P，释放资源时调用 V，这样让得不到资源的进程睡眠等待。

（没有使用PV原语就相当于这个资源压根没有上锁，信号量是根据这个锁来配相应的钥匙。）下来我们使用PV原语和信号量来协调进程之间的争夺。

#include <toyix.h>

semaphore s;

void f1()

{

int i;

for(i=1; i < 80; i++)

{

p(&s);

gotoxy(i,7);

delay(60);

printf("a");

v(&s);

}

}

void f2()

{

int i;

for(i=1; i < 80; i++)

{

p(&s);

gotoxy(i,9);

delay(250);

printf("b");

v(&s);

}

}

main()

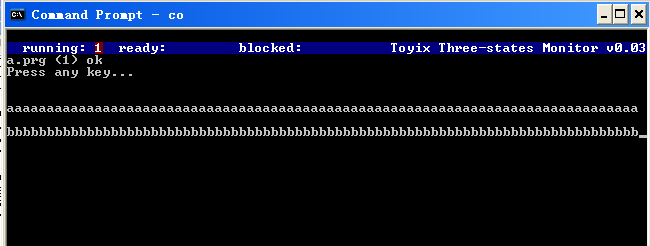
{

set(&s,1);

cobegin(f1,f2,0);

getch();

}



设置了PV原语，并且是信号量为1，就相当于给共享资源上了一把锁，配了一把钥匙，得到钥匙（通过P来得到钥匙）的就可以访问，没有得到的就在那等待，等待得到钥匙的交出钥匙（通过V来交出钥匙），之后才能访问。

那我如果配两把钥匙（设置信号量为2），是不是又会混乱呢？

在这里只用修改主函数，修改如下：

main()

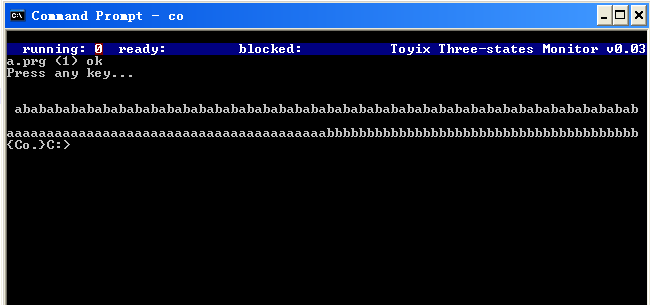
{

set(&s,2);

cobegin(f1,f2,0);

getch();

}



果然发生混乱了，因为有两把钥匙，即信号量为2，所以这两个轻权进程都可以访问共享资源，从而出现两个轻权进程争夺资源发生混乱的现象。

**同步：**任务是独立的，但是任务之间又有各种各样的关系，已成为一个整体，来完成某一项的工作。有时候，一个任务的完成前提是需要另一个任务给出一个结果，任务之间的这种机制的合作运行机制被称为任务间的同步。

下面来程序解释同步：

1.不是用PV原语和信号量,我们的想法是f1运行完的a的结果应该为80给f2使用，然后让f2也打印一行。

#include <toyix.h>

semaphore s;

int a=0;

void f1()

{

int i;

for(i=1; i <=80; i++)

{

gotoxy(i,7);

printf("a");

delay(500);

a++;

}

}

void f2()

{

int i;

for(i=1; i <=a; i++)

{

gotoxy(i,9);

printf("b");

delay(500);

}

}

main()

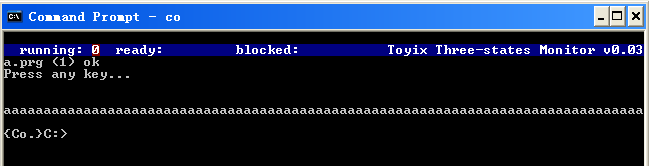
{

cobegin(f1,f2,0);

getch();

}

结果如下：



我们发现和我们的想法不一致，因为全局变量a还没有赋值的时候，f1堵塞自己了，然后f2将资源抢去，导致输出发现错误。这个时候我们就需要同步，因为f2和f1的关系就是f2要f1运行完的结果，所以必须等待f1运行完才能运行f2，解决方法就是采用PV原语和信号量对f1和f2就行同步。程序设计如下：

#include <toyix.h>

semaphore s;

int a=0;

void f1()

{

int i;

p(&s);

for(i=1; i <=80; i++)

{

gotoxy(i,7);

printf("a");

a++;

}

v(&s);

}

void f2()

{

int i;

p(&s);

for(i=1; i <=a; i++)

{

gotoxy(i,9);

printf("b");

}

v(&s);

}

main()

{

set(&s,1);

cobegin(f1,f2,0);

getch();

}



六、**进程与轻权进程**

1．论述进程与轻权进程之前先了解两个函数

int fork(void);

功能:创建一个与父进程相同的子进程

返回:

创建成功 在父进程中返回子进程的进程号，在子进程中返回0

创建失败 返回-1

说明:进程不共享数据段，父进程结束不撤销子进程。

int frk(void);

功能:创建一个与父进程相同的子进程

返回:

创建成功 父进程返回子进程的进程号，子进程返回0

创建失败 返回-1

说明:进程共享数据段，父进程结束后撤销子进程。

2. 每个进程都有自己的进程号，进程一般由控制信息和本体信息组成。本体信息一般有代码段、数据段、栈段组成，栈段要保存 cpu 现场等进程特有信息，所以它不可能被父进程和子进程共享

从上可知fork 创建的子进程与父进程不共享数据段，frk创建子进程与父进程共享数据段，frk这种设计是为了进程间交换数据的方便和、高效，同时也减少了数据段复制的开销，为了区别于 fork 创建的进程，frk创建的子进程叫轻权进程。Frk 有个特点就是主进程结束后，它创建的子进程会被撤销。

3. 我们现在验证下fork创建的子与父亲是不是不共享数据段，程序设计如下：

#include <toyix.h>

int a = 0;

main()

{

fork();

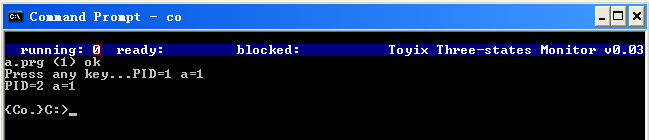
a++;

printf("PID=%d a=%d\n",get\_pid(),a);

delay(1000); /\*延迟 1000 毫秒\*/

}

我们看到的main函数就是主进程，fork()创建一个与主进程一模一样的进程，我们成被创建的为子进程，主进程为父进程。创建之后a++,打印了主进程的进程号和全局变量a的值，然后遇到delay(),主进程等待时间所以进入堵塞态，然后50ms中断到会进行调度就绪的进程，从而让子进程从就绪态转换为运行态。运行子进程的程序，子进程也会执行fork()函数，但是子进程不能再创建子进程了，然后执行a++,再打印子进程的进程号和全局变量a的值，遇到delay()；子进程将自己堵塞。然后父进程和子进程都在堵塞状态，主进程等待的时间是先到的，所以主进程先结束，然后子进程等待时间到，子进程结束。



a 的值都是 1，如果父进程和子进程共用同一数据段则 a 被加两次，应该为 2，结果不是 2，全局变量 a 在不同数据段。

下面判断代码是否在同一代码段，只需把 main 的地址打印出来比较一下就可以了。代码如下：

#include <toyix.h>

main()

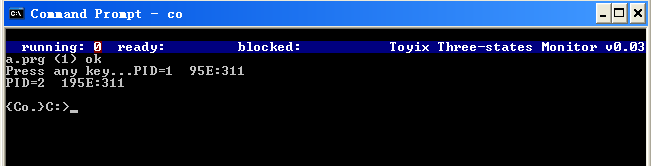
{

fork();

printf("PID=%d %x:%x\n",get\_pid(),\_CS,main);

delay(1000);

}



父子进程代码部分的段地址不同，偏移地址相同，代码不在同一代码段。得出的结论：Fork 创建的子进程和父进程的关系是：代码段、数据段、栈段都是独立的。

那么我大胆的猜想就是 进程之间的关系也是：代码段，数据段，栈段都是独立的。

程序设计如下：

首先我们先了解一个函数

int exec(char far \*p);

功能:执行一个可执行文件，原进程将被覆盖

参数:

p 文件名字符串指针

返回:失败返回-1

说明:主进程进程的本体信息不能被覆盖

在a.c文件中的程序如下：

#include <toyix.h>

int a=0;

main()

{

int i = fork();

if(i > 0)

{

a++;

printf("PID=%d %x:%x a=%d\n",get\_pid(),\_CS,main,a);

}

else

{

exec("b.prg");

printf("abcdef”);

}

}

在b.c文件中的程序如下：

#include <toyix.h>

int a=0;

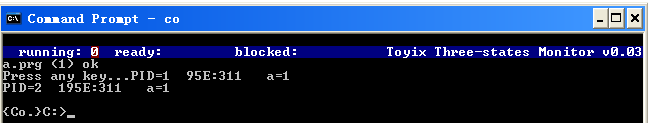
main()

{

a++;

printf("PID=%d %x:%x a=%d\n",get\_pid(),\_CS,main,a);

}



我们可以知道我们主进程创建的子进程执行exec("b.prg");子进程被另个进程覆盖掉，然后我们打印这个两个进程的资源信息，我们发现这两个进程的代码段，数据段，堆栈段都各自独立。

到这里我们就应该知道什么是进程了吧。下来我们来看看轻权进程的面貌。

4.我们来验证下frk创建的轻权进程与父亲是不是不共享数据段，程序设计如下：

#include <toyix.h>

int a = 0;

main()

{

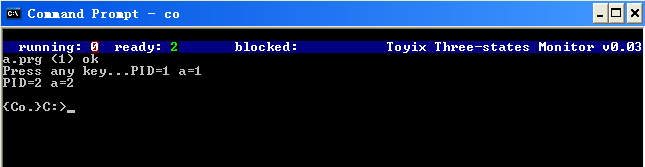
frk();

a++;

printf("PID=%d a=%d\n",get\_pid(),a);

delay(1000); /\*延迟 1000 毫秒\*/

}



a 的值是2，说明轻权进程和父进程共享数据段

从上可以知道轻权进程是使用父进程的资源的，那么父进程结束，轻权进程也就必须结束。前面的一些实验可以看到这点。比如上面的cobegin函数创建的轻权进程。

那么当在主进程里运行exec()函数，就会发生主进程被覆盖掉，得到一个新的进程，但是轻权进程不能使用这个新进程的资源，那么会发生什么呢？

#include <toyix.h>

main()

{

int i = frk();

if(i > 0)

{

exec("b.prg");

printf("a father run!\n");

getch();

}

else

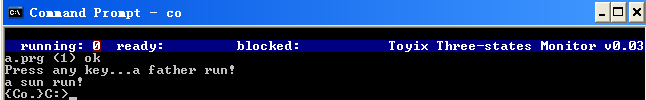
{

printf("a sun run!");

}

}

运行结果 如下图：



因为主进程被覆盖了，轻权进程就没有了数据段，这样的错误不能发生，所以exec就没有执行成功，大家可以发音exec的返回值来知道这个函数确实没有成功。

总结下：轻权进程必须有自己的父进程，一个进程中可以有多个轻权进程。所有的轻权进程共享父进程的数据段，父进程结束，轻权进程也相应的必须结束。轻权进程之间是共享数据段，所以他们需要有序的使用有限的资源就是同步问题，怎么做到呢，就采用PV原语和信号量来把各个轻权进程协调起来。轻权进程不会脱离进程而独立存在。

声明一下：轻权进程就和线程非常的相似，大家也可以当做线程理解。