多线程同步方法解决哲学家就餐问题
(Dining-Philosophers Problem)

Version : 1.0.1

2013-08-24

DOCUMENT HISTORY

Ed.	Version	Author	Change
1	1.0.0	缪海波	Initial(2013-08-20)
2	1.0.1	缪海波	实验设计局部修改(2013-08-24)

办公地点：孟宁　明德楼 A302 电话:0512-68839302　E-mail:mengning@ustc.edu.cn
缪海波 亲民楼 303 电话:0512-87161305　E-mail:mhb@mail.ustc.edu.cn

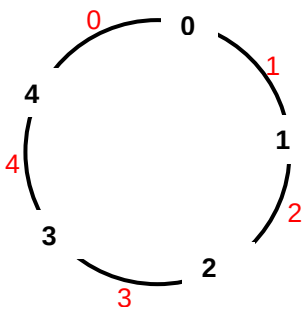
目　录

1 实验场景

哲学家就餐问题：由计算机科学家 Dijkstra 提出的经典死锁场景。

原版的故事里有五个哲学家(我们写的程序可以有 N 个哲学家)，这些哲学家们只做两件事：思考和吃饭，他们思考的时候不需要任何共享资源，但是吃饭的时候就必须使用餐具，而餐桌上的餐具是有限的，原版的故事里，餐具是叉子，吃饭的时候要用两把叉子把面条从碗里捞出来。本实验中，把叉子换成筷子，所以：一个哲学家需要两根筷子才能吃饭。

现在引入问题的关键：这些哲学家很穷，只买得起五根筷子。他们坐成一圈，两个人的中间放一根筷子。哲学家吃饭的时候必须同时得到左手边和右手边的筷子。如果他身边的任何一位正在使用筷子，那他只有等着。假设哲学家的编号是 0、1、2、3、4，筷子编号也是 0、1、2、3、4，哲学家和筷子围成一圈如下图所示：



2 实验内容

使用多线程模拟哲学家进餐问题。每个哲学家都是一个单独的线程，每个线程循环做以下动作：思考 rand()%10 秒，然后先拿左手边的筷子再拿右手边的筷子（筷子这种资源可以用互斥锁 mutex 或信号量表示），有任何一边拿不到就一直等着，全拿到就吃饭 rand()%10 秒，然后放下筷子。

练习 1：编译运行“仿真哲学家就餐场景的示例程序 dining_philosophers.c”

```
gcc dining_philosophers.c –o diding_philosophers –lpthread
./diding_philosophers
```

查看运行结果，分析一下，这个过程有没有可能产生死锁？

调用 usleep( )函数可以实现微秒级的延时，试着用 usleep(10)加快仿真的速度，看能不能观察到死锁现象。

练习 2：修改上述算法避免产生死锁。

哲学家就餐死锁可采取以下几种解决方法：

1）至多只允许 4 个哲学家同时进餐，以保证至少有一个哲学家能够进餐，最终总会释放出他所使用过的两支筷子，从而可使更多的哲学家进餐。

2）仅当哲学家的左、右两支筷子均可使用时，才允许他拿起筷子进餐。

3）规定奇数号哲学家先拿起他左边的筷子，然后再去拿起他右边的筷子；而偶数号哲学家则相反。

3 实验环境

硬件环境：不限

软件环境：VMWare 虚拟机软件、Linux 操作系统、gcc 编译器

4 基础知识

线程，有时被称为轻量级进程(Lightweight Process，LWP），是程序执行流的最小单元。一个标准的线程由线程ID，当前指令指针(PC)，寄存器集合和堆栈组成。另外，线程是进程中的一个实体，是被系统独立调度和分派的基本单位，线程自己不拥有系统资源，只拥有一点在运行中必不可少的资源，但它可**与同属一个进程的其它线程共享进程所拥有的全部资源**。一个线程可以创建和撤消另一个线程，同一进程中的多个线程之间可以并发执行。由于线程之间的相互制约，致使线程在运行中呈现出间断性。线程也有就绪、阻塞和运行三种基本状态。每一个程序都至少有一个线程，若程序只有一个线程，那就是程序本身。线程是程序中一个单一的顺序控制流程。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

同一进程的多个线程共享同一地址空间，因此Text Segment、Data Segment都是共享的，如果定义一个**函数**，在各线程中都**可以调用**，如果定义一个**全局变量**，在各线程中都**可以访问**到，除此之外，各线程还共享以下进程资源和环境：

> 文件描述符表
> 每种信号的处理方式（SIG_IGN、SIG_DFL或者自定义的信号处理函数）
> 当前工作目录
> 用户id和组id

但有些资源是每个线程各有一份的：

> 线程id
> 上下文，包括各种寄存器的值、程序计数器和栈指针
> 栈空间
> errno变量
> 信号屏蔽字
> 调度优先级

我们使用的线程库函数是由 POSIX 标准定义的，称为 POSIX thread 或者 pthread。在 Linux 上线程函数位于 libpthread 共享库中，**因此在编译时要加上-lpthread 选项**。

1. 线程创建

pthread_creat()

函数功能: 该函数用来创建新的线程.

函数原型:int pthread_create(

pthread_t *restrict thread,

const pthread_attr_t *restrict attr,

void*(*start_routine),

void*restrict arg);

函数参数:thread 用于保存线程的线程变量;

attr 为要设置的线程属性.

start_routine 用于指向线程执行时调用的函数

arg 为线程要执行函数的调用参数.

返回值:如果成功,则返回 0;否则返回-1.失败时将不会创建新的线程.

在一个线程中调用pthread_create()创建新的线程后，当前线程从pthread_create()返回继续往下执行，而新的线程所执行的代码由我们传给pthread_create的函数指针start_routine决定。start_routine函数接收一个参数，是通过pthread_create的arg参数传递给它的，该参数的类型为void *，这个指针按什么类型解释由调用者自己定义。start_routine的返回值类型也是void *，这个指针的含义同样由调用者自己定义。start_routine返回时，这个线程就退出了，其它线程可以调用pthread_join得到start_routine的返回值，类似于父进程调用wait(2)得到子进程的退出状态。

pthread_create成功返回后，新创建的线程的id被填写到thread参数所指向的内存单元。我们知道进程id的类型是pid_t，每个进程的id在整个系统中是唯一的，调用getpid(2)可以获得当前进程的id，是一个正整数值。线程id的类型是thread_t，它只在当前进程中保证是唯一的，在不同的系统中thread_t这个类型有不同的实现，它可能是一个整数值，也可能是一个结构体，也可能是一个地址，所以不能简单地当成整数用printf打印，调用pthread_self(3)可以获得当前线程的id。

attr参数表示线程属性，这里我们不深入讨论线程属性，所有代码例子都传NULL给attr参数，表示线程属性取缺省值。

线程创建示例程序：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
pthread_t ntid;
```

```
void printids(const char *s)
{       pid_t pid;
        pthread_t tid;

        pid = getpid();
        tid = pthread_self();
        printf("%s pid %u tid %u (0x%x)\n", s, (unsigned int)pid, (unsigned int)tid, (unsigned int)tid);
}

void *thr_fn(void *arg)
{
        printids(arg);
        return NULL;
}

int main(void)
{
         int err;
         err = pthread_create(&ntid, NULL, thr_fn, "new thread: ");
         if (err != 0) {
                 fprintf(stderr, "can't create thread: %s\n", strerror(err)); exit(1);
         }
         printids("main thread:");
         sleep(1);
         return 0;
}
```

编译运行结果如下：

```
$ gcc main.c –lpthread
$ ./a.out
main thread: pid 7398 tid 3084450496 (0xb7d8fac0)
new thread: pid 7398 tid 3084446608 (0xb7d8eb90)
```

因此在Linux上，thread_t类型是一个地址值，属于同一进程的多个线程调用getpid(2)可以得到相同的进程号，而调用pthread_self(3)得到的线程号各不相同。由于pthread_create的错误码不保存在errno中，因此不能直接用perror(3)打印错误信息，可以先用strerror(3)把错误码转换成错误信息再打印。

如果任意一个线程调用了exit或_exit，则整个进程的所有线程都终止，由于从main函数return也相当于调用exit，为了防止新创建的线程还没有得到执行就终止，我们在main

函数return之前延时1秒，这只是一种权宜之计，即使主线程等待1秒，内核也不一定会调度新创建的线程执行

2. 线程终止

如果需要只终止某个线程而不终止整个进程，可以有三种方法：

- 从线程函数return。这种方法对主线程不适用，从main函数return相当于调用exit。
- 一个线程可以调用pthread_cancel终止同一进程中的另一个线程。
- 线程可以调用pthread_exit终止自己，其他线程可以调用pthread_join终止它。

这里只介绍pthread_join，一般情况下，线程终止后，其终止状态一直保留到其它线程调用pthread_join获取它的状态为止，类似于wait与waitpid。

pthread_join():
函数功能:用来等待一个线程的结束.
函数原型:int pthread_join(pthread_t thread,void **value_ptr);
函数参数:pthread 参数为被等待的线程标识符
Value_ptr 为一个用户定义的指针,指向一个保存等待线程的完整退出状态的静态区域.
返回值:如果执行成功,则返回 0;否则返回非 0 值.

3. 线程同步（互斥锁操作）

互斥锁，是一种信号量，常用来防止两个进程或线程在同一时刻访问相同的共享资源。

pthread_mutex_init()
函数功能:实现互斥锁的初始化
函数原型:pthread_mutex_init(
Pthread_mutex_t *restrict mutex.
Const pthread_mutexattr_t *restrict attr);
函数参数:mutex 是指向要初始化的互斥锁的指针;
attr 是指向属性对象的指针,如果指针为 NULL,则使用缺省的属性.
返回值: 成功返回 0,否则返回非 0 值.

pthread_mutex_lock()
函数功能:用于对互斥锁进行加锁操作
函数原型:pthread_mutex_lock(
pthread_mutex_t *mutex);
函数参数:mutex 为指向要锁定的互斥锁的指针.
返回值: 成功返回 0,否否则返回指明错误的错误编号。

pthread_mutex_unlock()

函数功能:对互斥锁进行**解锁**操作

函数原型:pthread_mutex_unlock(pthread_mutex_t *mutex)

函数参数:mutex 指向要解锁的互斥锁的指针

返回值: 成功返回 0,否则返回指明错误的错误编号。

使用形式：

```
pthread_mutex_t mutex;
pthread_mutex_init (&mutex, NULL); /*定义*/
...
pthread_mutex_lock(&mutex); /*获取互斥锁*/
... /*临界资源*/
pthread_mutex_unlock(&mutex); /*释放互斥锁*/
```

示例：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#define NLOOP 5000

int counter; /* incremented by threads */
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;

void *doit(void *);

int main(int argc, char **argv)
{
        pthread_t tidA, tidB;
        pthread_create(&tidA, NULL, doit, NULL);
        pthread_create(&tidB, NULL, doit, NULL);

        /* wait for both threads to terminate */
        pthread_join(tidA, NULL);
        pthread_join(tidB, NULL);
        return 0;
}

void *doit(void *vptr)
{
        int i, val;
        /* * Each thread fetches, prints, and increments the counter NLOOP times. * The value of the counter
        should increase monotonically. */
        for (i = 0; i < NLOOP; i++)
        {
                pthread_mutex_lock(&counter_mutex);
                val = counter;
                printf("%x: %d\n", (unsigned int)pthread_self(), val + 1);
                counter = val + 1;
                pthread_mutex_unlock(&counter_mutex);
        }
    return NULL;
}
```

我们创建两个线程，各自把 counter 增加 5000 次，正常情况下最后 counter 应该等于 10000，但事实上，**若不用互斥锁**，每次运行该程序的结果都不一样，有时候数到 5000 多，有时候数到 6000 多。这里使用互斥锁，运行结果就正常了，每次运行都能数到 10000

4. 线程同步（信号量**Semaphore**）

Mutex变量是非0即1的，可看作一种资源的可用数量，初始化时Mutex是1，表示有一个可用资源，加锁时获得该资源，将Mutex减到0，表示不再有可用资源，解锁时释放该资源，将Mutex重新加到1，表示又有了一个可用资源。信号量（Semaphore）和Mutex类似，表示可用资源的数量，和Mutex不同的是这个数量可以大于1。

信号量其实就是一个计数器，也是一个整数。每一次调用 wait 操作将会使 semaphore 值减一，而如果 semaphore 值已经为 0，则 wait 操作将会阻塞。每一次调用 post 操作将会使 semaphore 值加一。

sem_init()
函数功能：初始化信号量
函数原型：sem_init(sem_t *sem,int pshared,unsigned int value);
函数参数：第一个参数是一个类型为 sem 的指针
返回值：成功返回 0，否则返回其它值

sem_wait()
函数功能：**阻塞**减少**信号量**
函数原型：sem_wait(sem_t *sem);
函数参数：参数是一个类型为 sem 的指针
返回值：成功返回 0，否则返回其它值

sem_post()
函数功能：**增加** sem 所指示的**信号量**
函数原型：sem_post(sem_t *sem);
函数参数：参数是一个类型为 sem 的指针
返回值：成功返回 0，否则返回其它值

sem_destroy()
函数功能：**销毁信号量**状态
函数原型：sem_destroy(sem_t *sem);
函数参数：参数是一个类型为 sem 的指针
返回值：成功返回 0，否则返回其它值

semaphore变量的类型为sem_t，sem_init()初始化一个semaphore变量，value参数表

示可用资源的数量，pshared参数为0表示信号量用于同一进程的线程间同步，本节只介绍这种情况。在用完semaphore变量之后应该调用sem_destroy()释放与semaphore相关的资源。调用sem_wait()可以获得资源，使semaphore的值减1，如果调用sem_wait()时semaphore的值已经是0，则挂起等待。如果不希望挂起等待，可以调用sem_trywait()。调用sem_post()可以释放资源，使semaphore的值加1，同时唤醒挂起等待的线程。

使用形式：

```
sem_t sem;
sem_init(&sem, 0, 1); /*信号量初始化*/
...

sem_wait(&sem);     /*等待信号量*/
... /*临界资源*/
sem_post(&sem);     /*释放信号量*/
```

程序示例：生产者－消费者

```
#include <stdlib.h>
#include <pthread.h>
#include <stdio.h>
#include <semaphore.h>
#define NUM 5
int queue[NUM];
sem_t blank_number, product_number;

void *producer(void *arg)
 {     int p = 0;
       while (1) {
       sem_wait(&blank_number);
       queue[p] = rand() % 1000 + 1;
       printf("Produce %d\n", queue[p]);
       sem_post(&product_number);
       p = (p+1)%NUM;
       sleep(rand()%5);
     }
}
void *consumer(void *arg)
 {         int c = 0;
           while (1) {
           sem_wait(&product_number);
```

```
            printf("Consume %d\n", queue[c]);
            queue[c] = 0;
            sem_post(&blank_number);
            c = (c+1)%NUM;
            sleep(rand()%5);
            }
}
 int main(int argc, char *argv[])
 {
        pthread_t pid, cid;
        sem_init(&blank_number, 0, NUM);
        sem_init(&product_number, 0, 0);

        pthread_create(&pid, NULL, producer, NULL);
        pthread_create(&cid, NULL, consumer, NULL);
        pthread_join(pid, NULL);
        pthread_join(cid, NULL);

        sem_destroy(&blank_number);
        sem_destroy(&product_number);
        return 0;
}
```

信号量(semaphore)和互斥锁（mutex）间的区别:

锁必须是同一个线程获取以及释放，否则会死锁。而信号量则不必。

作用域

信号量: 进程间或线程间(linux 仅线程间)

互斥锁: 线程间

上锁时

信号量: 只要信号量的 value 大于 0，其他线程就可以 sem_wait 成功，成功后信号量的 value 减一。若 value 值不大于 0，则 sem_wait 阻塞，直到 sem_post 释放后 value 值加一

互斥锁: 只要被锁住，其他任何线程都不可以访问被保护的资源成功后否则就阻

5 实验考核

1. 运行示例程序，仿真出死锁现象。
2. 实现一种解决死锁的算法，并解释相关代码。

6 参考资料

[1] 互斥锁 pthread_mutex_t 的使用

http://blog.163.com/coffee_666666/blog/static/184691114201182125470/

[2] [Linux] 线程同步: mutex, semaphore, condition(ZT)

http://blog.chinaunix.net/uid-8735300-id-2017125.html