《操作系统》
实验指导书

南昌大学软件学院

实验 3： 同步与互斥实验

3.1 实验概述

实验目的：

1. 通过编写程序实现进程/线程同步和互斥，掌握同步和互斥的原理、算法，以进一步巩固有关知识；
2. 了解 Linux 中多线程的并发执行机制，学习使用 Linux 中基本同步对象，掌握相应的系统调用函数的使用。

实验语言：c

实验环境：linux、gcc

3.2 实验背景

3.2.1 线程创建

创建一个程序 **t0.c**，其包含两个线程，分别输出“A”和“B”。

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

void *mythread(void *arg) {
    printf("%s\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin\n");
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    // join waits for the threads to finish
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: end\n");
    return 0;
}
```

主程序创建了两个线程，分别执行函数 mythread()，但是传入不同的参数（字符串类型的 A 或者 B）。一旦线程创建，可能会立即运行（取决于调度程序的兴致），或者处于就绪状态，等待执行。创建了两个线程（T1 和 T2）后，主程序调用 pthread_join()，等待特定线程完成。

编译执行示例：（可能的执行结果）

```
[huyong@localhost ch26]$ gcc -o t0 t0.c -pthread
[huyong@localhost ch26]$ ./t0
main: begin
B
A
main: end
```

请注意，输出的 A&B 的排序不是唯一可能的顺序，这取决于调度程序决定在给定时刻运行哪个线程。

3.2.2 线程间数据共享

在 t0.c 基础上开发程序 **t1.c**，线程共享全局变量，每个工作线程循环向共享变量加 1，循环执行 1000 万（10^7）次。因此，预期的最终结果是：20000000。遗憾的是，即使是在单处理器上运行这段代码，也不一定能获得预期结果。

```c
#include <stdio.h>
#include <assert.h>
#include <pthread.h>

static volatile int counter = 0;

// 在循环中重复地将 1 加到 counter 中。
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    printf("main: begin (counter = %d)\n", counter);
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
    rc = pthread_join(p1, NULL); assert(rc == 0);
    rc = pthread_join(p2, NULL); assert(rc == 0);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```

编译执行示例：（可能的执行结果）

```
[huyong@localhost ch26]$ gcc -o t1 t1.c -pthread
[huyong@localhost ch26]$ ./t1
main: begin (counter = 0)
B: begin
A: begin
A: done
B: done
main: done with both (counter = 20000000)
[huyong@localhost ch26]$ ./t1
main: begin (counter = 0)
B: begin
A: begin
A: done
B: done
main: done with both (counter = 18653117)
```

3.2.3 采用互斥锁解决数据共享问题

为了解决 3.2.2 的问题，可以采用互斥锁的方式，在 t1.c 的基础上，修改线程函数成为新的程序 **t2.c**。

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i, rc;
    for (i = 0; i < 1e7; i++) {
        rc = pthread_mutex_lock(&lock); assert(rc == 0);
        counter = counter + 1;
        rc = pthread_mutex_unlock(&lock); assert(rc == 0);
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

编译 t2.c 后执行，会发现每次都能得到正确答案。但是，它运行会有点慢（可以从命令行计时观察到这一点）。

按以下所示调整一下代码成为新的程序 **t3.c**。编译后执行，会发现执行速度变快了。***思考一下，为什么能快一些？这种方式又有什么不利的地方？***

```
void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i, rc;
    rc = pthread_mutex_lock(&lock); assert(rc == 0);
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    rc = pthread_mutex_unlock(&lock); assert(rc == 0);
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

3.2.4 test_and_set()函数（非原子指令）

```
volatile unsigned int mutex = 0; //0未持有, 1持有

void SpinLock(volatile unsigned int *lock) {
    while (*lock == 1) // TEST (lock)
    ; // spin
    *lock = 1;         // SET (lock)
}

void SpinUnlock(volatile unsigned int *lock) {
    *lock = 0;
}

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e8; i++) {
        SpinLock(&mutex);
        counter = counter + 1;
        SpinUnlock(&mutex);
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}
```

t4.c 程序中参考 test_and_set()指令的形式构建自定义函数来实现。注意，为了能测试出执行中的问题，将循环次数放大了 10 倍（1e8）。

编译执行示例：（可能的执行结果）

```
[huyong@localhost ch26]$ ./t4
main: begin (counter = 0)
B: begin
A: begin
B: done
A: done
main: done with both (counter = 200000000)
[huyong@localhost ch26]$ ./t4
main: begin (counter = 0)
B: begin
A: begin
A: done
B: done
main: done with both (counter = 198753090)
```

可见，如同 3.2.2 一样，这种自定义的锁在没有硬件支持的情况下无法保证正确的执行结果。***思考一下，为什么会出现错误？***

3.2.4 xchg 函数（原子执行，x86 平台）

将 3.2.3 中 8-16 行的代码改为原子执行的 xchg 函数（程序 **t5.c**）：

```
 8 static inline unsigned int xchg(volatile unsigned int *addr, unsigned int newval)
 9 {
10     unsigned int result;
11     asm volatile("lock; xchg %0, %1"
12         : "+m" (*addr), "=a" (result) : "1" (newval) : "cc");
13     return result;
14 }
15
16 void SpinLock(volatile unsigned int *lock) {
17     while (xchg(lock, 1) == 1) ; // spin
18 }
19
20 void SpinUnlock(volatile unsigned int *lock) {
21     xchg(lock, 0);
22 }
```

xchg 函数中采用了 gcc 的内联汇编方式，通过汇编指令 lock，确保原子方式执行类同 CAS 指令的功能:“返回 addr 指向的原内容，同时将 newval 存入 addr”。

为了更好地理解，采用 objdump 得到 xchg 的汇编代码：

```
080484d4 <xchg>:
 80484d4:       55                      push   %ebp
 80484d5:       89 e5                   mov    %esp,%ebp
 80484d7:       83 ec 10                sub    $0x10,%esp
 80484da:       8b 55 08                mov    0x8(%ebp),%edx
 80484dd:       8b 45 0c                mov    0xc(%ebp),%eax
 80484e0:       8b 4d 08                mov    0x8(%ebp),%ecx
 80484e3:       f0 87 02                lock xchg %eax,(%edx)
 80484e6:       89 45 fc                mov    %eax,-0x4(%ebp)
 80484e9:       8b 45 fc                mov    -0x4(%ebp),%eax
 80484ec:       c9                      leave
 80484ed:       c3                      ret
```

编译执行示例：

```
[huyong@localhost ch26]$ gcc -o t5 t5.c -pthread
[huyong@localhost ch26]$ ./t5
main: begin (counter = 0)
B: begin
A: begin
B: done
A: done
main: done with both (counter = 20000000)
```

3.2.5 采用信号量解决数据共享问题

现在采用 linux 下的互斥信号量（二进制信号量）来解决，在 t1.c 的基础上改造为新程序 **t6.c**。

```
#include <pthread.h>
#include <semaphore.h>

static volatile int counter = 0;
sem_t mutex;

void *mythread(void *arg) {
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        sem_wait(&mutex);
        counter = counter + 1;
        sem_post(&mutex);
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p1, p2;
    int rc;
    sem_init(&mutex, 0, 1);
    printf("main: begin (counter = %d)\n", counter);
    rc = pthread_create(&p1, NULL, mythread, "A"); assert(rc == 0);
    rc = pthread_create(&p2, NULL, mythread, "B"); assert(rc == 0);
```

1 引入头文件

2 定义信号量

3 初始化信号量

4 使用信号量

3.3 实验内容

3.3.1 生产者消费者问题

参考教材中生产者消费者算法，创建 5 个线程，其中两个线程为生产者线程，3 个线程为消费者线程。一个生产者线程试图不断地在一个缓冲区中写入大写英文字母，另一个生产者线程试图不断地在缓冲区中写入小写英文字母。3 个消费者不断地从缓冲区中读取一个字符并输出。为了使得程序的输出易于看到结果，可分别在生产者和消费者线区程的合适的位置加入一些随机睡眠时间。缓冲区的大小为 10。

可选的实验：在上面实验的基础上实现部分消费者有选择地消费某些产品。例如一个消费者只消费小写字符，一个消费者只消费大写字母，而另一个消费者则无选择地消费任何产品。消费者要消费的产品没有时，消费者进程被阻塞。注

意缓冲区的管理。

3.3.2 读者写者问题

教材中对读者写者问题算法有描述。对于读优先的算法，在不断地有读者流的情况下，写者会被阻塞。

编写一个写者优先解决读者写者问题的程序，其中读者和写者均是多个线程，用信号量作为同步互斥机制。