一、实验基础信息

个人信息

201700130011 — 刘建东 — 17级菁英班

实验信息

日期：2019.11.20

题目：利用信号量实现线程同步

实验目的

1. 理解 $Nachos$ 中如何创建并发线程
2. 理解 $Nachos$ 中信号量与 P、V 操作是如何实现的
3. 如何创建与使用 $Nachos$ 的信号量
4. 理解 $Nachos$ 中是如何利用信号量实现 $producer/consumer\ problem$
5. 理解 $Nachos$ 中如何测试与调试程序
6. 理解 $Nachos$ 中轮转法（RR）线程调度的实现

实验任务

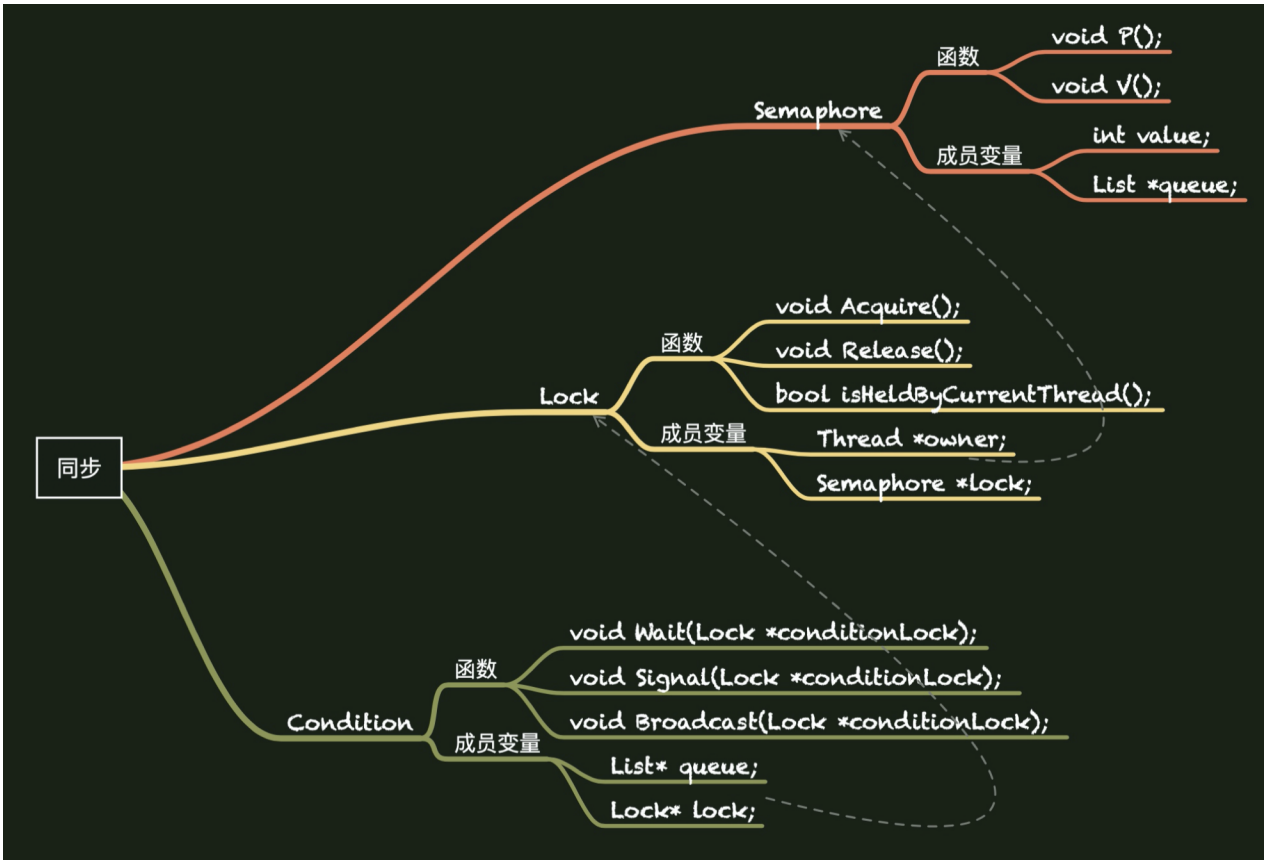
- *阅读代码*：ring.h、ring.cc、main.cc、prodcons++.cc（code/lab3）
- *补充代码*：利用 $Nachos$ 实现的信号量写一个 $producer/consumer\ problem$ 测试程序。（主要在 prodcons++.cc 中补充代码）
- *分析代码*
    1. ../threads/threadtest.cc，理解利用 Thread::Fork() 创建线程的方法
    2. Thread::Fork()，理解内核创建线程的过程
    3. ../threads/synch.cc，理解 $Nachos$ 中信号量是如何实现的
    4. ../monitor/prodcons++.cc，理解信号量的创建与使用方法
    5. Thread::Fork()、Thread::Yiled()、Thread::Sleep()、Thread::Finish()、Scheduler::ReadyToRun()、Scheduler::FindNextToRun()、Scheduler::Run() 等相关函数，理解线程调度及上下文切换的工作过程
    6. $Nachos$ 对参数 -rs 的处理过程，理解时钟中断的实现，以及 RR 调度算法的实现方法

二、实验基本方法

该实验主要考察的是对于 $Nachos$ 系统中同步问题的理解深度，因此该实验的基本方法就是一些关于同步的背景知识，信号量以及经典的生产者/消费者问题。

（1）信号量

$Nachos$ 在线程同步问题上实现了一个三级数据结构，分别是信号量、锁、条件变量，其中第二层引用第一层，第三层引用第二层，如下图所示。



而在本实验中主要考察的是用信号量来实现线程同步，即使用 Semaphore 类来完成实验。

（2）生产者/消费者问题

生产者/消费者问题是操作系统同步问题中的一个经典模型，其大致内容是生产者与消费者均可访问共享内存中的一个环形缓冲区，生产者生产出产品后将其放入环形缓冲区，而消费者从环形缓冲区取出产品进行消费。当环形缓冲区满时生产者阻塞直到缓冲区有空时将其唤醒继续执行；同样，当缓冲区为空时消费者阻塞直到缓冲区非空时将其唤醒继续执行。

在这个模型中，比较关键的有两点：

1. 环形缓冲区的大小
2. 用什么同步机制来对缓冲区的取出和放入进行控制

在该实验中，我们采用了信号量来完成该问题，但除信号量之外还有其它的方法也可以实现该模型所需的同步机制，比如管程。

除了上述两个同步问题之外，我们还需要进一步理解 code/threads/main.cc 文件，了解命令行参数、内核初始化、SimpleThread (int which) 函数。

（3）命令行参数

```
int main(int argc, char **argv);
```

在命令行参数中，首要内容就是理解 argc 与 argv 的 含义，其中 argc 表示命令参数个数，argv 大小为 argc，存储每个命令的字符串形式。

```
gene@ubuntu:~/Desktop/OS/code/threads$ ./nachos -t
argc:2
argv:./nachos -t
```

（4）Initialize 函数

在 Initialize 中主要是对于命令行参数的一个初始化，其中主要内容是关于 -d 和 -rs 的使用。

```
for (argc--, argv++; argc > 0; argc -= argCount, argv += argCount) {
  argCount = 1;
  if (!strcmp(*argv, "-d")) {
      if (argc == 1)
    debugArgs = "+";  // 打开所有debug标记
      else {
        debugArgs = *(argv + 1);
        argCount = 2;
      }
  } else if (!strcmp(*argv, "-rs")) {
      ASSERT(argc > 1);     // 设置 -rs 时必须设置随机数种子
      // 传入计数器种子
      RandomInit(atoi(*(argv + 1)));  // 初始化伪随机
      randomYield = TRUE;
      argCount = 2;
  }
```

-d：（1）如果只是-d，则打开所有debug标记；（2）如果是 -d t，则只打开 thread 的 debug 部分，-d之后的具体内容可以参看具体源代码内容。

-rs：从代码中ASSERT部分中可以看到，-rs 一定要设置随机数种子，不然会报错。

```
gene@ubuntu:~/Desktop/OS/code/threads$ ./nachos -rs
argc:2
argv:./nachos -rs
Assertion failed: line 106, file "system.cc"
已放弃 (核心已转储)
```

（5）SimpleThread

在 ThreadTest 函数中，调用的最主要的函数是 Thread::Fork()，也正是由于这个Fork()函数，才使得SimpleThread输出时出现了下述的状况。

```
gene@ubuntu:~/Desktop/OS/code/threads$ ./nachos -d t
Entering SimpleTestForking thread "forked thread" with func = 0x804a495, arg = 1
Putting thread forked thread on ready list.
*** thread 0 looped 0 times
Yielding thread "main"
Putting thread main on ready list.
Switching from thread "main" to thread "forked thread"
*** thread 1 looped 0 times
Yielding thread "forked thread"
Putting thread forked thread on ready list.
Switching from thread "forked thread" to thread "main"
Now in thread "main"
```

其中Fork函数调用了StackAllocate函数，并将SimpleThread函数指针传了进去，并用该函数设置了InitialPCState变量，因此Fork的函数在SimpleThread处开始执行，出现了上述的情况。

```
machineState[PCState] = (_int) ThreadRoot;
machineState[StartupPCState] = (_int) InterruptEnable;
machineState[InitialPCState] = (_int) func;
machineState[InitialArgState] = arg;
machineState[WhenDonePCState] = (_int) ThreadFinish;
```

此处需要将 $Nachos$ 的 Fork 函数与 Linux 中的进程 Fork 函数做好区分，Linux 中的Fork会将当前整个进程地址空间拷贝到子进程中，但是如果父进程和子进程都没有对内存内容进行修改，那么该内存页就可以在父进程与子进程之间进行共享。

而 $Nachos$ 中的Fork并不存在这种将当前线程地址空间拷贝到子线程中的操作，而更像是一种分支，并且设定了子线程运行的函数地址与函数参数。

（6）monitor/prodcons++.cc

该部分代码与本次实验所要完成的 lab3/prodcons++.cc 大部分内容相似，最主要的区别在于 lab3 中要求我们用信号量来实现同步与互斥，而 monitor 中则用 Ring 类封装了信号量与条件变量来实现了同步与互斥。

在 Ring 类中的 Put 与 Get 函数中，用 mutex 信号量来保证临界资源只会被一个线程访问，用 notfull 和 notempty 两个条件变量来保证生产信息时缓冲区没有满，消费信息时缓冲区也没有空。

（7）Thread::Yield()

```
void Thread::Yield () {
    /* 关中断 */
    nextThread = scheduler->FindNextToRun();
    if (nextThread != NULL) {
      scheduler->ReadyToRun(this);
      scheduler->Run(nextThread);
    }
    /* 开中断 */
}
```

该函数将当前线程的状态设置为可运行态，将下一个可运行态线程调度出来运行。要注意该函数与 Sleep 函数的区别，Sleep 函数将线程设置为了阻塞态，需要唤醒。而 Yield 函数只是将下一个线程调度出来运行，之前运行的线程仍然属于可运行态。

（8）Thread::Sleep()

```
void Thread::Sleep (){
    Thread *nextThread;
    /* 保证当前为关中断状态 */
    status = BLOCKED; // 设置为阻塞态
    while ((nextThread = scheduler->FindNextToRun()) == NULL)
      interrupt->Idle();  // 没有线程运行，等待中断
    scheduler->Run(nextThread); // 运行下一个线程
}
```

该函数即将当前线程设置为阻塞态，并找到下一个可运行态线程进行调度。

（9）Thread::Finish()

```
void Thread::Finish () {
    /* 关中断 */
    threadToBeDestroyed = currentThread;
    Sleep();          // 触发线程切换
}
```

该函数用于中止当前线程，首先需要将该线程阻塞并进行线程切换，之后再由其它线程将该线程删除。

（10）Scheduler::ReadyToRun()

```
void Scheduler::ReadyToRun (Thread *thread) {
    thread->setStatus(READY); // 线程状态设为可运行态
    readyList->Append((void *)thread); // 将该线程加入可运行态线程队列
}
```

该函数用于将一个线程设置为可运行状态，一共分为两步。第一步设置线程状态，第二步将该线程加入到可运行态线程队列中。

（11）Scheduler::FindNextToRun()

```
Thread * Scheduler::FindNextToRun () {
    return (Thread *)readyList->Remove();
}
```

该函数用于从可运行线程队列中调度出第一个线程运行。

（12）Scheduler::Run()

```
void Scheduler::Run (Thread *nextThread) {
    Thread *oldThread = currentThread;

    oldThread->CheckOverflow();          // 检查是否越界
    currentThread = nextThread;          // 线程交换
    currentThread->setStatus(RUNNING);  // 设置 nextThread 状态
    SWITCH(oldThread, nextThread);       // 汇编代码实现的线程交换

    if (threadToBeDestroyed != NULL) {
      delete threadToBeDestroyed;   // 销毁线程
      threadToBeDestroyed = NULL;
    }
}
```

该函数用于运行一个可运行态线程，需要检查越界等问题并将当前进程和下一个进程进行进程切换。

（13）RR 调度算法

Nachos 是一个模拟系统，因此时间片的定义不同于实际中的操作系统。在 Nachos 中将时间片消耗分为了两个部分，即用户程序与系统程序。

```
#define UserTick 1
#define SystemTick 10
```

如上所示，对于用户程序来说，每执行一条用户指令，相当于时间片消耗了1。而对于系统程序来说，每当系统从关中断状态转化为开中断状态，相当于时间片消耗了10。因此在系统程序中，只要不进行开关中断的转换，无论执行多长的程序，都不会导致时间片的消耗。

```
int Timer::TimeOfNextInterrupt() {
    if (randomize) return 1 + (Random() % (TimerTicks * 2));
    else return TimerTicks;
}
```

在 Nachos 的 RR 调度算法中，每次需要传入一个随机种子，之后时间片的长度均根据该种子随机产生一个数值用于当前线程的时间片。从上述代码中可以看出，线程的时间片范围在 [1, 2*TimerTicks] 之间。

上述的内容即为本次实验中的一些基本内容与研究方法。

三、源代码及注释

```
#define BUFF_SIZE 3  // the size of the round buffer
#define N_PROD    2  // the number of producers
#define N_CONS    2  // the number of consumers
#define N_MESSG   4  // the number of messages produced by each producer
#define MAX_NAME  16 // the maximum lengh of a name

#define MAXLEN  48
#define LINELEN 24

Thread *producers[N_PROD]; //array of pointers to the producer
Thread *consumers[N_CONS];  // and consumer threads;

char prod_names[N_PROD][MAX_NAME];  //array of charater string for prod names
char cons_names[N_CONS][MAX_NAME];  //array of charater string for cons names

Semaphore *nempty, *nfull; //two semaphores for empty and full slots
Semaphore *mutex;          //semaphore for the mutual exclusion

Ring *ring;

void Producer(_int which) {
    int num;
    slot *message = new slot(0,0);

    // 循环 N_MESSG 次生产信息
    for (num = 0; num < N_MESSG ; num++) {
      // 处理 message 信息
      message->value = num;
      message->thread_id = which;

      // 先同步再互斥
      nempty->P();
      mutex->P();

      // 放入信息
      ring->Put(message);

      // 与 P() 对应的操作
      mutex->V();
      nfull->V();
    }
}

void Consumer(_int which) {
    char str[MAXLEN];
    char fname[LINELEN]; // 文件名
```

```
    int fd;

    slot *message = new slot(0,0);
    sprintf(fname, "tmp_%d", which); // 产生文件名

    if ( (fd = creat(fname, 0600) ) == -1) // 创建文件
    {
      perror("creat: file create failed");
      exit(1);
    }

    for (; ; ) {

      // 先同步后互斥
      nfull->P();
      mutex->P();

      // 获取 message
      ring->Get(message);

      // 与 P() 对应的操作
      mutex->V();
      nempty->V();

      // debug 信息
      sprintf(str,"producer id --> %d; Message number --> %d;\n",
        message->thread_id, message->value);
      if ( write(fd, str, strlen(str)) == -1 ) { // 将上述信息写入文件
        perror("write: write failed");
        exit(1);
      }
    }
}

void ProdCons()
{
    int i;
    DEBUG('t', "Entering ProdCons");

    // 定义三个信号量
    mutex = new Semaphore("mutex",1);              // 互斥信号量
    nempty = new Semaphore("nempty",BUFF_SIZE);    // 判断缓冲区是否为空
    nfull = new Semaphore("nfull",0);              // 判断缓冲区是否为满

    // 定义缓冲区
    ring = new Ring(BUFF_SIZE);

    // 创建 N_PROD 个生产者线程
    for (i=0; i < N_PROD; i++)
```

```
    {
      // 命名生产者线程
      sprintf(prod_names[i], "producer_%d", i);

      // 产生新线程并fork到具体函数位置
      producers[i] = new Thread(prod_names[i]);
      producers[i]->Fork(Producer,i);
    };

    // 创建 N_CONS 个消费者线程
    for (i=0; i < N_CONS; i++)
    {
      // 命名消费者线程
      sprintf(cons_names[i], "consumer_%d", i);

      // 产生消费者线程并fork到具体函数位置
      consumers[i] = new Thread(cons_names[i]);
      consumers[i]->Fork(Consumer,i);
    };
}
```

上述代码即为本实验中修改部分的核心代码，并在重点位置加上了具体注释。

四、实验测试方法及结果

1. 信号量、缓冲区创建

```
mutex = new Semaphore("mutex",1);
nempty = new Semaphore("nempty",BUFF_SIZE);
nfull = new Semaphore("nfull",0);
ring = new Ring(BUFF_SIZE);
```

mutex 为互斥锁，保证临界资源同时只能有一个线程操作，因此初始值为 1。

nempty 为临界资源是否为空的信号量，当 nempty = 0 时，临界资源为空。每生产一个资源，nempty--；每消耗一个资源，nempty++，因此 nempty 的初值为 BUFF_SIZE，即缓冲区大小。

nfull 为临界资源是否为满的信号量，当 nfull = 0 时，临界资源为空。每生产一个资源，nfull++；每消耗一个资源，nfull--，因此 nfull 的初值为 0，表明缓冲区中无资源。

ring 为缓冲区，BUFF_SIZE 为缓冲区大小。

2. 生产者、消费者线程创建

```
for (i=0; i < N_PROD; i++) {
  sprintf(prod_names[i], "producer_%d", i);
  producers[i] = new Thread(prod_names[i]);
  producers[i]->Fork(Producer,i);
};
for (i=0; i < N_CONS; i++) {
  sprintf(cons_names[i], "consumer_%d", i);
  consumers[i] = new Thread(cons_names[i]);
  consumers[i]->Fork(Consumer,i);
};
```

N_PROD 为生产者个数，prod_names[i] 为第 i 个生产者的名称。N_CONS 为消费者个数，cons_names[i] 为第 i 个消费者的名称。

producers[i]、consumers[i] 分别为第 i 个生产者和第 i 个消费者，调用 Fork 函数即设置该线程的运行函数以及将该线程的状态设置为可运行态。

3. 生产者、消费者函数

```
void Producer(_int which)
{
    int num;
    slot *message = new slot(0,0);

    for (num = 0; num < N_MESSG ; num++) {
      // 修改message信息
      message->value = num;
      message->thread_id = which;

      // 先同步后互斥
      nempty->P();
      mutex->P();

      ring->Put(message);

      // P()操作的对应操作
      mutex->V();
      nfull->V();
    }
}
```

在生产者的函数中，先修改生产的message的信息，然后先判断缓冲区是否满了，即 nempty 信号量的 P 操作，之后再判断互斥信号量是否可用，保证只有一个线程在操作缓冲区。

通过同步和互斥这两个信号量的判断之后，将生产的信息放入缓冲区中，然后释放互斥信号量以及缓冲区是否为空信号量+1。

```
void Consumer(_int which)
{
    char str[MAXLEN], fname[LINELEN];
    int fd;
    slot *message = new slot(0,0);

    // 设置message信息
    sprintf(fname, "tmp_%d", which);

    // 创建一个文件
    if ( (fd = creat(fname, 0600) ) == -1) {
      perror("creat: file create failed");
      exit(1);
    }

    for (; ; ) {
      // 先同步后互斥
      nfull->P();
      mutex->P();
      ring->Get(message);

      // P()操作的对应操作
      mutex->V();
      nempty->V();

      // debug 信息
      sprintf(str,"producer id --> %d; Message number --> %d;\n",
        message->thread_id, message->value);

      // 将信息输出到文件
      if ( write(fd, str, strlen(str)) == -1 ) {
        perror("write: write failed");
        exit(1);
      }
    }
}
```

与生产者函数类似，消费者函数首先定义了输出信息存储的文件路径，然后依次执行了两个信号量。第一个信号量用于判断缓冲区是否为空，第二个信号量则是保证只有一个线程在操作缓冲区。判断完同步和互斥之后，消费者从缓冲区中拿走message，然后释放了互斥信号量以及缓冲区是否为满信号量+1。

4. 结果测试

实现上述三个步骤之后，即可完成本实验要求的同步过程。接下来我们开始测试，先测试非时间片轮转线程调度算法的情况，两个输出文件如下所示。

```
tmp_0:
producer id --> 0; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 0;
producer id --> 1; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 1; Message number --> 3;

tmp_1:
```

可以看到所有生产者产生的message都被0号消费者消耗了，而1号消费者没有消耗任何信息。接下来我们通过输出中间过程信息得到下述的调试信息。

```
gene@ubuntu:~/Desktop/OS/code/lab3$ ./nachos -d -t
Entering ProdConsForking thread "producer_0" with func = 0x804bb6a, arg = 0
Putting thread producer_0 on ready list.
Forking thread "producer_1" with func = 0x804bb6a, arg = 1
Putting thread producer_1 on ready list.
Forking thread "consumer_0" with func = 0x804bc35, arg = 0
Putting thread consumer_0 on ready list.
Forking thread "consumer_1" with func = 0x804bc35, arg = 1
Putting thread consumer_1 on ready list.
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "producer_0"
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Putting thread producer_0 on ready list.
Putting thread producer_1 on ready list.
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_0"
Now in thread "producer_0"
Deleting thread "main"
Putting thread consumer_0 on ready list.
```

由上述调度输出过程以及源代码，我们可以得知该情境下的调度过程。

1. 执行main函数，并依次生成pro0、pro1、con0、con1线程，并依次激活放入可运行线程的等待队列中，main函数执行结束。
2. 执行pro0，生产了3个message之后，缓冲区满，pro0被阻塞放入信号量的等待队列中。
3. 执行pro1，缓冲区满，无法生产message，被阻塞并放入信号量的等待队列中。
4. 执行con0，消费第一个message，pro0被激活，放入可运行线程的等待队列中；消费第二个message，pro1被激活并放入可运行线程的等待队列中；消费第三个message，缓冲区为空，con0被阻塞并放入信号量的等待队列中。
5. 执行con1，缓冲区为空，被阻塞并放入信号量的等待队列中。
6. 执行pro0，生产了1个message，con0被激活并放入可运行线程等待队列中，循环结束，pro0线程运行结束。
7. 执行pro1，生产第1个message，con1被激活并放入可运行线程等待队列中；生产第二个message，缓冲区满，pro1被阻塞并放入信号量的等待队列中。

8. 执行con0，消费第一个message，pro1被激活并放入可运行线程等待队列中；消费第二、三个message后缓冲区为空，con0被阻塞并放入信号量的等待队列中。
9. 执行con1，缓冲区为空，con1被阻塞并放入信号量的等待队列中。
10. 执行pro1，生产第1个message激活con0，生产第2个message激活con1，pro1执行结束。
11. 执行con0，连续消耗两个message，至此生产者生产的message均被消耗完毕。

至此，观察上述线程调度过程，我们可以得知为何在非时间片调度中，生产者所生产的message均被0号消费者消耗。接下来我们再来观察加上时间片调度之后，两个消费者所消耗的内容。

```
tmp_0:
producer id --> 0; Message number --> 0;
producer id --> 1; Message number --> 0;
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 1;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 3;

tmp_1:
producer id --> 0; Message number --> 1;
producer id --> 1; Message number --> 2;
```

用终端输出具体的调试信息如下。

```
Finishing thread "main"
Sleeping thread "main"
Switching from thread "main" to thread "producer_0"
Yielding thread "producer_0"
Putting thread producer_0 on ready list.
Switching from thread "producer_0" to thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Sleeping thread "consumer_0"
Switching from thread "consumer_0" to thread "consumer_1"
Sleeping thread "consumer_1"
Switching from thread "consumer_1" to thread "producer_0"
Now in thread "producer_0"
Deleting thread "main"
Putting thread producer_1 on ready list.
Putting thread consumer_0 on ready list.
Putting thread consumer_1 on ready list.
Sleeping thread "producer_0"
Switching from thread "producer_0" to thread "producer_1"
Now in thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_0"
Now in thread "consumer_0"
```

当前输出的调试结果可能不能很好的表现出时间片轮转算法的特性，因此我们可以采取输出具体时钟信息的方法来进一步观察。

```
== Tick 690 ==
        interrupts: on -> off
Time: 690, interrupts off
Pending interrupts:
Interrupt handler timer, scheduled at 724
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
Finishing thread "producer_1"
Sleeping thread "producer_1"
Switching from thread "producer_1" to thread "consumer_1"
Now in thread "consumer_1"
Deleting thread "producer_1"
        interrupts: off -> on

== Tick 700 ==
        interrupts: on -> off
Time: 700, interrupts off
Pending interrupts:
Interrupt handler timer, scheduled at 724
End of pending interrupts
        interrupts: off -> on
        interrupts: on -> off
        interrupts: off -> on
```

从上述调试信息中可以更直观的看到各个时间点上系统所执行的线程调度，以及nachos中系统程序时间片的计算方法是根据中断从关->开算10的方法来进行计算的。

我们也可以在程序中直接输出中间过程，获得更为具体的生产者消费者之间的运行关系。

```
gene@ubuntu:~/Desktop/OS/code/lab3$ ./nachos -rs 5
Producer_0 produces message_0
Producer_0 produces message_1
Producer_1 produces message_0
Consumer_0 consumes message_0 from Producer_0
Consumer_1 consumes message_1 from Producer_0
Producer_0 produces message_2
Consumer_0 consumes message_0 from Producer_1
Producer_1 produces message_1
Consumer_0 consumes message_2 from Producer_0
Consumer_0 consumes message_1 from Producer_1
Producer_1 produces message_2
Producer_0 produces message_3
Producer_1 produces message_3
Consumer_1 consumes message_2 from Producer_1
Consumer_0 consumes message_3 from Producer_0
Consumer_0 consumes message_3 from Producer_1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!
```

至此我们即完成了本次实验的要求，既实现了生产者、消费者之间的同步关系，也通过nachos的debug功能、文件输出或中间过程输出三种方式观察了各个线程具体的运行结果。

五、实验体会

1. 此次实验主要是对于生产者、消费者问题的实现与加深理解，不单深入理解了nachos中为同步所设置的信号量、锁、条件变量等数据结构，还掌握了nachos中线程调度、线程运行的各个系统函数以及时间片轮转调度的具体原理和方法。
2. 除了查看nachos的代码之外，本次实验也是第一个要求自己写代码的实验，第一次用自己的代码来完善nachos的功能，也算是为之后的实验打下了基础。
3. 本次实验中主要介绍了生产者、消费者这个经典模型，其中用到了3个信号量，分别表示临界资源的互斥、缓冲区是否为空、缓冲区是否满了三个同步信号，以及一个公用的环形缓冲区，之所以设计成环形缓冲区也是为了更好地利用缓冲区的空间。该实验加深了我对于该经典模型的理解，当然该模型也并非只能用信号量来实现，还可以用条件变量以及管程来实现。
4. 该实验还应用了nachos系统中时间片轮转调度的方法，其中对于系统程序来说，只有中断从关的状态变为开的状态才算消耗了时间片，该系统中定义此中情况消耗了10；而对于用户程序来说，每当执行一条用户指令就算时间片消耗了1。这种定义方法也与正常系统按照CPU脉冲作为一个时间片的方案有很大的区别，提供给了我们一种软件模拟硬件的新思路。
5. 除了上述关于原理、模型的体会之外，本次实验还教会了我三种调试方法。第一种是该实验中所提供的输出到文件的方法，此种方法输出的信息比较简洁也比较容易控制。

   第二种方法是nachos提供的debug参数方法，这种方法的优点是输出的信息非常完整，但是信息量非常大，调试起来会非常麻烦。

   第三种方法就是程序调试中常用的中间过程输出方法，这种方法由自己定义调试位置，信息输出内容，比较灵活、简洁，而且可以直观的看到每一个输出之间的先后关系。
6. 总结一下，此次实验介绍了同步的经典模型，加深了实验者对于nachos同步、线程、调度、时间片轮转等知识的理解并且使得实验者开始自行实现代码，为之后的实验打下了良好的基础。