山东大学 计算机科学与技术 学院

操作系统 课程实验报告

学号：202200101007	姓名：张祎乾	班级：22.3 班
实验题目：实验：		
实验学时：2	实验日期：2025/2/22	
实验目的： （1）进一步理解 Nachos 中如何创建线程； （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；		
实验环境：WSL、Ubuntu		
源程序清单： prodcons++_v1.cc prodcons++_v2.cc		
编译及运行结果： 对于有两个生产者与两个消费者的情况下，如果每个生产者分别生产 4 个 消息，其运行结果如下：		

```
zhang@zhang:~/OS/nachos-3.4/code/lab3$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 730, idle 0, system 730, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

OS > nachos-3.4 > code > lab3 > 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;

```

OS > nachos-3.4 > code > lab3 > tmp_1

```
1
```

假设有 3 个生产者，2 个消费者。每个生产者向缓冲池中写入 5 个消息。

```
#define BUFF_SIZE 3  // the size of the round buffer 缓冲池的大
#define N_PROD    3  // the number of producers 生产者的数量
#define N_CONS    2  // the number of consumers 消费者的数量
#define N_MESSG   5  // the number of messages produced by ea
#define MAX_NAME  16 // the maximum lengh of a name 名字的最大
```

```
zhang@zhang:~/OS/nachos-3.4/code/lab3$ ./nachos
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1310, idle 0, system 1310, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

OS > nachos-3.4 > code > lab3 > 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 --> 0; Message number --> 4;
producer id --> 1; Message number --> 0;
producer id --> 1; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 1; Message number --> 3;
producer id --> 1; Message number --> 4;
producer id --> 2; Message number --> 0;
producer id --> 2; Message number --> 1;
producer id --> 2; Message number --> 2;
producer id --> 2; Message number --> 3;
producer id --> 2; Message number --> 4;

```

OS > nachos-3.4 > code > lab3 > tmp_1

```
1
```

运行 Nachos 时利用参数-rs 创建一个定时器设备（Timer），如 nachos

-rs 5，结果如下：

```
zhang@zhang:~/OS/nachos-3.4/code/lab3$ ./nachos -rs 5
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1515, idle 5, system 1510, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
```

OS > nachos-3.4 > code > lab3 > tmp_0

```
1    producer id --> 1; Message number --> 0;
2    producer id --> 1; Message number --> 1;
3    producer id --> 0; Message number --> 0;
4    producer id --> 2; Message number --> 1;
5
```

OS > nachos-3.4 > code > lab3 > tmp_1

```
1    producer id --> 1; Message number --> 2;
2    producer id --> 1; Message number --> 3;
3    producer id --> 1; Message number --> 4;
4    producer id --> 2; Message number --> 0;
5    producer id --> 2; Message number --> 2;
6    producer id --> 0; Message number --> 1;
7    producer id --> 2; Message number --> 3;
8    producer id --> 0; Message number --> 2;
9    producer id --> 0; Message number --> 3;
10   producer id --> 0; Message number --> 4;
11   producer id --> 2; Message number --> 4;
12
```

思考：如何将 FCFS 修改成 RR？

答：指南最后有一句“思考：如何将 FCFS 修改成 RR？”，我觉得它的

意思应该让我们在代码层面实现RR,比如producers[0]生产2个message后就换producers[1],

我利用枚举，并写了#define SA RR 来选择调度策略，并定义了 RR_(这三行在 prodcons++.cc 代码中开头紧邻，很容易找到)

RR_：生产者/消费者 每 生产/消费 RR_个 message，就切换线程

我在 Producer()和 Consumer()处进行了少量修改，得到了 RR 调度算法的代码实现，作为 prodcons++_v2.cc

验证如下：

```
#define BUFF_SIZE 20  // the size of the round buffer 缓冲池的大小
#define N_PROD    2  // the number of producers 生产者的数量
#define N_CONS    2  // the number of consumers 消费者的数量
#define N_MESSG   5  // the number of messages produced by each producer
#define MAX_NAME  16 // the maximum lengh of a name 名字的最大长度
```

理论上，会按照如下执行顺序：

p0-p1-c0-c1-p0-p1-c0-c1-p0-p1-c0

c0 会消耗 p0 的 12345，p1 的 0

c1 会消耗 p1 的 1234

结果如下，证明代码有效：

```
OS > nachos-3.4 > code > lab3 >  tmp_0
1    producer id --> 0; Message number --> 0;
2    producer id --> 0; Message number --> 1;
3    producer id --> 0; Message number --> 2;
4    producer id --> 0; Message number --> 3;
5    producer id --> 0; Message number --> 4;
6    producer id --> 1; Message number --> 4;
```

OS > nachos-3.4 > code > lab3 > ≡ tmp_1

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

改动如下：

```
25    enum Scheduling_algorithm{FCFS,RR};
26    #define RR_ 2 // 每RR_行为替换另一个线程
27    #define SA RR // 选择调度算法
```

Producer 中改动：

```
if(SA == RR && (num+1)%RR_==0){currentThread->Yield();}
```

Consumer 中改动：

原本 consumer 是不需要 num 的

```
for (int num=0;;num++) {
```

```
if(SA == RR && (num+1)%RR_==0){currentThread->Yield();}
```

问题及收获：

问题 1

问题描述：指南表示:ring.cc 和 ring.h 不需要修改，但是我通过阅读 ring.cc 发现两个"to be implemented(待实现)"的函数……，算是一个矛盾点，我觉得需要改

ring.cc 中的 Ring 的 Full()和 Empty()均未实现，我决定先把它实现了:

```
int Ring::Empty(){return in == out;}
int Ring::Full(){return out == (in + 1) % size;}
```

补充：阅读代码发现，这两个函数压根没用到，不实现也行。

收获：

经过本次实验,我重温了生产者-消费者问题,对同步互斥有了新的认识。

我学到的 nachos 的操作如下：

线程操作：

```
Thread *t = new Thread("forked thread");//创造一个线程
t->Fork(func, arg);//让线程 t 去执行函数，后边为参数
t->setStatus(state)// 设置线程状态： JUST_CREATED, RUNNING, READY, BLOCKED
Sleep();// 将线程置于睡眠状态并放弃处理器
```

currentThread->Yield();// 当前线程进行让步，换一个线程继续执行

t->Finish();// 线程 t 完成执行

Ring 缓冲池

slot* message = new slot(Tid,num);

Ring* ring = new Ring(size);// 创造一个大小为 size 的缓冲池

ring->Put(message);// 将 message 放入缓冲池

ring->Get(message);// 取出一个 message

ring->Empty();// 判空

ring->Full();// 判满

信号量：

Semaphore* sem;

sem = new Semaphore(debugName,initialValue);// 参数为名字和初始值

sem->getName();

sem->P();

sem->V();

互斥锁(本实验用不到)：

```
Lock * lock= new Lock("lock");// 互斥锁的创建
lock->Acquire();// 获得锁
lock->Release();// 释放锁
lock->isHeldByCurrentThread();// 如果当前线程持有此锁，则为True。
```

条件变量(本实验用不到)：

```
Condition *con = new Condition("con");// 条件变量创建
con->Wait(lock);// 发现锁被占用，进入等待队列
con->Signal(lock);// 通知等待该锁的一个线程继续执行
con->Broadcast(lock);// 通知等待该锁的所有线程继续执行
```