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：201705130120 | 姓名：苑宗鹤 | 班级：17 1 班 |
| --- | --- | --- |
| 实验题目：实验一 | | |
| 实验学时：2 | 实验日期：2020/3/13 | |

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；

实验环境：ubuntu18 x64 windows10 clion

实验步骤：

1.阅读 code/lab3 目录下的 ring.h、ring.cc、main.cc 及 prodcons++.cc

2.在理解它们工作机理的基础上，补充目录 lab3 中提供的代码，利用 Nachos 实现的信号量写一个 producer/consumer problem 测试程序。主要是在 prodcons++.cc 中补充代码。

3.分析./threads/threadtest.cc，理解利用 Thread：Fork()创建线程的方法；

4.分析../threads/synch.cc，理解 Nachos 中信号量是如何实现的；

5.分析./monitor/prodcons++.cc，理解信号量的创建与使用方法；

6. 分 析 Thread:Fork(),Thread::Yiled(),Thread::Sleep(),Thread:Finish(),Scheduler: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；

7.分析 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；

8.补全 prodcons++.cc.

实验结果：

仓库地址：

https://github.com/Yuandiaodiaodiao/nachos-cmake-x64

# 理解利用 Thread： Fork()创建线程的方法:

```
void Thread::Fork( VoidFunctionPtr func, _int arg ) {

    DEBUG( flag: 't', format: "Forking thread \"%s\" with func

    StackAllocate( func, arg );

    IntStatus oldLevel = interrupt->SetLevel( level: IntOff );
    scheduler->ReadyToRun( thread: this ); // ReadyToRun assum
                                           // are disabled!
    (void) interrupt->SetLevel( oldLevel );
}
```

Thread::Fork 将一个函数打包为一个线程
StackAllocate 为线程分配 栈空间并在 machineState 中添加线程的初始化函数 线程入口地址 线程参数等等信息
然后将当前进程切换为 *READY 并放入* readyList

# 信号量是如何实现:

Synch.h 中定义了信号量 Class Semaphore
提供了 p 操作 v 操作
P 操作:
Value==0 时将当前线程插入阻塞队列中
当前线程执行 Sleep 从就绪队列中切换出下一个线程
计数器--
V 操作:
计数器++
如果阻塞队列里有线程
则取出执行 ReadyToRun 加入就绪队列中 切换为 ***READY 状态***

```
class Semaphore {
  public:
    Semaphore(char* debugName, int initialValue);
    ~Semaphore();                          // de-allo
    char* getName() { return name;}        // deb

    void P();    // these are the only operations
    void V();    // they are both *atomic*

  private:
    char* name;        // useful for debugging
    int value;         // semaphore value, always
    List *queue;       // threads waiting in P() f
};
```

# 信号量的创建与使用方法

通过对 class Semaphore 实例化来创建信号量
初始化参数可以加入信号量 name 和初值

```
// ....
mutex = new Semaphore( debugName: "mutex", initialValue: 1);
nempty = new Semaphore( debugName: "nempty", initialValue: 0);
nfull = new Semaphore( debugName: "nfull", BUFF_SIZE); //消费
// Put the code to construct a ring buffer object with size
```

通过 Semaphore->P()和 Semaphore->V()执行 PV 操作

# 线程调度及上下文切换

## Thread::Sleep()

将当前线程挂起 等待中断 切换新线程

## Thread:Finish()

准备销毁当前线程并 Sleep()当前线程进行线程切换

## Scheduler:ReadyToRun()

将进程切换为 READY 状态 并放入就绪队列

## Scheduler:FindNextToRun()

从就绪队列中取出就绪队列队首元素

## Scheduler:Run()

把 CPU 分发给下一个运行的线程，调用 switch 代码以保存旧线程状态，同时加载新线程状态。
保存用户线程的寄存器信息
调用 switch，切换新旧线程的状态 （之前运城在当先线程的栈空间，之后运行在 nextThread 的栈空间）；
如果旧线程结束运行 根据 threadToBeDestroyed 删除旧线程
新线程如果是用户程序，则恢复当线程的地址空间。

# -rs 参数和 RR 调度算法:

有-rs 参数时会在初始化时加入一个中断 执行定时为随机时间

```
Timer::Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom)
{
    randomize = doRandom;
    handler = timerHandler;
    arg = callArg;

    // schedule the first interrupt from the timer device
    interrupt->Schedule(TimerHandler,  arg: (_int) this,  when: TimeOfNextInterrupt(),
        type: TimerInt);
}
```

在中断到达时 执行 TimerExpired
将自身的 handle 插入中断队列 并定为新的随机时间

```
void
Timer::TimerExpired()
{
    // schedule the next timer device interrupt
    interrupt->Schedule(TimerHandler,  arg: (_int) this,  when: TimeOfNextInterrupt(),
        type: TimerInt);

    // invoke the Nachos interrupt handler for this device
    (*handler)(arg);
}
```

这样每隔一段时间就会有一个中断被触发 执行线程切换

| |
|---|
| 问题及收获： |
| |

学号姓名实验一.doc；