Lab3 利用信号量实现线程同步

> Code: Nachos/code/lab3 at main · roomdestroyer/Nachos (github.com)
>
> Report: Nachos/README.md at main · roomdestroyer/Nachos (github.com)

1. **Nachos 创建线程的流程：**

在本实验中，程序进入 `main` 后，调用函数 `ProdCons` 创建生产者线程和消费者线程。在函数 `ProdCons` 中，程序初始化了生产者线程和消费者线程需要使用到的同步信号量和互斥信号量，初始化了生产者线程和消费者线程所使用的共享环形缓冲区，然后使用两个 `for` 循环分别创建生产者线程和消费者线程，并将它们放到线程就绪队列中。

以创建生产者进程为例，首先使用线程类 `thread` 动态分配一个新对象，表示一个新线程，然后对该线程调用 `Fork` 函数，参数为线程分配到的执行内容及参数（生产者函数及参数）。在 `Fork` 函数中，程序先为线程分配函数堆栈（待执行对象的程序和数据堆栈），然后使用 `scheduler` 对象调用该线程，将其放入就绪队列中。在函数 `ReadyToRun` 中，函数首先把线程状态设置为就绪态 `READY`，然后调用 `Append` 方法将其放入就绪队列。

如下图所示，当函数 `ProdCons` 调用完毕后，实际上就完成了所有生产者线程和消费者线程的创建过程。

```
After function ProdCons, invoke Scheduler->Print():
~~~
Now all threads for producers and customers are created.
Ready list contents:
producer_0, producer_1, producer_2, consumer_0, consumer_1,
~~~
```

---

2. **信号量 `P & V` 操作的实现：**

本实验一共涉及到 3 个信号量：互斥信号量 `mutex`，同步信号量 `nfull` 与 `nempty` 。信号量的私有属性有信号量的值，它是一个阀门。线程等待队列中存放所有等待该信号量的线程。信号量有两个操作：`P` 操作和 `V` 操作，这两个操作都是原子操作。

- `P` 操作：① 当 `value == 0` 时，将当前运行线程放入线程等待队列，当前线程进入睡眠状态，并且切换到其它线程运行。② 当 `value > 0` 时，`value--`。
- `V` 操作：① 如果线程等待队列中有等待该信号量的线程，取出其中一个将其设置成就绪态，准备运行。② `value++`。

---

3. **Nachos 如何创建与使用信号量：**

- `Nachos` 需要两个同步信号量和一个互斥信号量，信号量的创建在全局变量中：

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

- 信号量的初始化在线程创建函数 `ProdCons` 中，表示当前信号量只有生产者线程和消费者线程使用：

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

- 对于单个生产者线程，信号量的使用如下：

```
nempty->P();
mutex->P();
ring->Put(message);
mutex->V();
nfull->V();
```

  其中，两个互斥信号量必须在生产操作 `ring->Put(message);` 的紧挨着的两边，保证该操作是原子的，防止了死锁的出现。`nempty->P()` 表示生产消息之前先把缓冲区的空闲量 `-1`，`nfull->V()` 表示生产结束后把缓冲区消息量 `+1`。

- 对于单个消费者线程，信号量的使用如下：

```
nfull->P();
mutex->P();
ring->Get(message);
mutex->V();
nempty->V();
```

  其设置逻辑与生产者线程类似。

---

4. **Nachos 如何通过信号量实现 `producer/consumer problem`：**

当执行命令为 `./nachos` 时，生产者线程和消费者线程按照 `FCFS` 规则调度，即生产者线程先依次填满缓冲区，消费者线程再依次取出缓冲区，然后生产者线程再依次填满缓冲区，如此循环。由于消费者线程中的循环设置为：

```
    for (; ; ) {
        ......
    }
```

因此，**即使有多个消费者线程，也永远只有第一个消费者能够占据处理机运行。**

考虑第一轮，打印中间调试信息，其执行顺序如下：

```
~~~
==========================================
Now in thread "producer_0"
Ready list contents:
```

```
producer_1, producer_2, consumer_0, consumer_1,
producer_0 put msg 0
producer_0 put msg 1
producer_0 put msg 2
producer_0 put msg 3
producer_0 put msg 4

==========================================
Now in thread "producer_1"
Ready list contents:
producer_2, consumer_0, consumer_1,

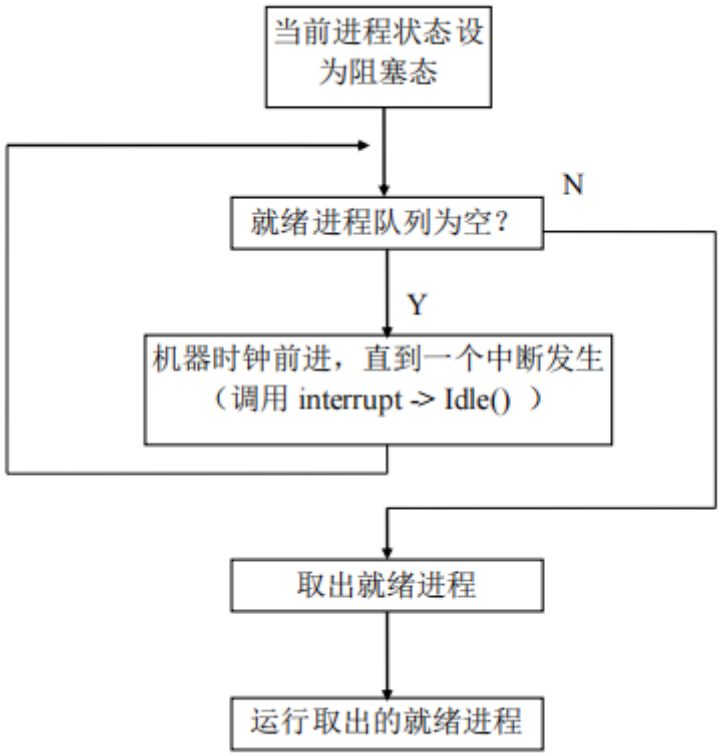
==========================================
Now in thread "producer_2"
Ready list contents:
consumer_0, consumer_1,

==========================================
Now in thread "consumer_0"
Ready list contents:
consumer_1,
consumer_0 get msg 0
consumer_0 get msg 1
consumer_0 get msg 2
consumer_0 get msg 3
consumer_0 get msg 4

==========================================
Now in thread "consumer_1"
Ready list contents:
producer_1, producer_2,

~~~
```

可以发现，当缓冲区大小为 `5` 时，第一轮执行顺序为 `producer_0 -> producer_1 -> producer_2 -> consumer_0 -> consumer_1`，其中，`producer_0` 在第一轮就填满了缓冲区，因此其工作完毕后，`producer_1` 和 `producer_2` 都不会执行，这两个线程会先从就绪队列中取出，查询缓冲区是否还有余量，发现并没有，然后会调用信号量 `P` 操作的 `currentThread->Sleep();` 函数，进入睡眠状态。



在 `Sleep` 方法中，线程的状态设置为 `BLOCKED`，**此时线程并没有放回到就绪队列中，而是在 `while` 循环中不断等待就绪队列中的下一个线程运行**，直到下一个线程运行完毕，就绪队列中没有线程后，阻塞的线程才会继续运行。

观察第二轮的输出结果：

```
~~~
================================================
Now in thread "producer_1"
Ready list contents:
producer_2,
producer_1 put msg 0
producer_1 put msg 1
producer_1 put msg 2
producer_1 put msg 3
producer_1 put msg 4

================================================
Now in thread "producer_2"
Ready list contents:
consumer_0, consumer_1,

================================================
Now in thread "consumer_0"
Ready list contents:
consumer_1,
consumer_0 get msg 0
consumer_0 get msg 1
consumer_0 get msg 2
consumer_0 get msg 3
consumer_0 get msg 4

================================================
Now in thread "consumer_1"
```

```
Ready list contents:
producer_2,

~~~
```

由于第一轮中线程 `producer_0` 已经运行完毕，于是第二轮是线程 `producer_1` 上处理机运行（按照被阻塞的顺序），线程 1 生产完毕后，消费者线程 `consumer_0` 取走生产出的消息。

第三轮的输出结果为：

```
===========================================
Now in thread "producer_2"
Ready list contents:

producer_2 put msg 0
producer_2 put msg 1
producer_2 put msg 2
producer_2 put msg 3
producer_2 put msg 4

===========================================
Now in thread "consumer_0"
Ready list contents:
consumer_1,
consumer_0 get msg 0
consumer_0 get msg 1
consumer_0 get msg 2
consumer_0 get msg 3
consumer_0 get msg 4

===========================================
Now in thread "consumer_1"
Ready list contents:
```

规律如同以上分析，第三轮中只有线程 `producer_2` 在生产，消费者 `consumer_0` 紧随其后取走缓冲区的内容，当生产者线程全部结束后，只留下消费者线程 `consumer_0` 在原地踏步等待，而 `consumer_1` 及之后的线程则在就绪队列中等待（但永远没有机会占据处理机，除非更改消费者线程的执行函数，如 `for` 循环的退出条件）。

---

5. **Nachos 中如何测试与调试程序：**

- 首先是可以使用 `gdb` 命令，在相应函数口设置断点，观察程序在整个过程中需要进过哪些步骤，在哪些步骤处变量的值发生了改变，根据变量的值改变情况梳理程序的执行逻辑与出错点。
- 其次，可以在关键函数处打印相应的提示信息，例如，可以在函数 `ProdCons` 的末尾执行 `scheduler->Print()` 观察当前就绪队列中有哪些线程，以此来观察线程的创建顺序。
- 由于直接使用 `print` 在控制台打印不方便其它模块调试，因此可以使用内置的 `DEBUG` 方法，例如，如果要在某个调试点打印如下内容：

  ```
  printf("%s get msg %d\n", currentThread->getName(), message->value);
  ```

则可以改为：

```
DEBUG('m', "%s get msg %d\n", currentThread->getName(), message->value);
```

然后，在控制台使用如下命令：

```
./nachos -d m
```

就可以自定义调试输出了。

---

6. **Nachos 中时间片轮转法（RR）的实现：**

在之前的操作中，`Nachos` 总是按照 `FCFS` 来调度线程的，即按照线程创建的顺序来调度，如果有线程被阻塞，就按照其阻塞的顺序来唤醒。这样做的缺点是降低了系统的并发性和效率，实际的操作系统需要根据各个线程优先级、执行时间来完成调度，放置低优先级、运行时间长的线程占据处理机过长时间。 `Nachos` 中实现了时间片轮转法调度 `RR` ，使用如下命令可以执行时间片轮转的线程调度算法：

```
./nachos -rs <random seed #>
```

其中， `<random seed #>` 是一个随机种子，例如 ，可以输入 `10` 。下面从几个方面来介绍该算法：

- 时钟中断模块（文件 `timer.cc  timer.h` ）

  该模块的作用是模拟时钟中断。Nachos 虚拟机可以如同实际的硬件一样，每隔一定的时间会发生一次时钟中断。这是一个可选项，目前 Nachos 还没有充分发挥时钟中断的作用，只有在 Nachos 指定线程随机切换时启动时钟中断，在每次的时钟中断处理的最后，加入了线程的切换。**时钟中断相当于机器的脉搏，只有每隔一段时间发生了时钟中断，机器才有机会停下当前的事去思考下一步该干什么。**

  时钟中断间隔由 `TimerTicks` 宏决定（100倍 `Tick` 的时间）。在系统模拟时，如果系统就绪进程不止一个的话，每次时钟中断都一定会发生进程的切换，定义在 `system.cc` 中 `TimerInterruptHandler` 函数：

  ```
  TimerInterruptHandler(_int dummy)
  {
      if (interrupt->getStatus() != IdleMode)
      interrupt->YieldOnReturn();
  }
  ```

  所以运行 Nachos时，如果以同样的方式提交进程，系统的结果将是一样的。这不符合操作系统的运行不确定性的特性。所以在模拟时钟中断的时候，加入了一个**随机因子**，如果该因子设置的话，时钟中断发生的时机将在一定范围内是随机的。这样有些用户程序在同步方面的错误就比较容易发现。但是这样的时钟中断和真正操作系统中的时钟中断将有不同的含义。不能像真正的操作系统那样通过时钟中断来计算时间等等。是否需要随机时钟中断可以通过设置选项 `-rs` 来实现。

  Timer 类的设计如下：

- ```
  class Timer {
    public:
  ```
  ```

```
    // 初始化方法
    Timer(VoidFunctionPtr timerHandler, _int callArg, bool doRandom);
    ~Timer() {}
    // 发生时钟中断时调用
    void TimerExpired();
    // 计算下一次时钟中断发生的时间
    int TimeOfNextInterrupt();
  private:
    // 是否需要随机时钟中断标志
    bool randomize;
    // 时钟中断处理函数
    VoidFunctionPtr handler;
    // 时钟中断处理函数的参数
    _int arg;
};
```

- 当设置参数 `-rs` 后，实际上时钟中断处理类成员 `randomize` 被设置为 `doRandom`，`TimerHandler` 会调用函数 `TimerExpired` 处理中断，之所以不直接调用 `TimerExpired`， 是因为 `C++` 不允许直接引用一个类内部方法的指针，要想**单独保存函数和参数**，就必须嵌套一层来调用。

  在 `TimerExpired` 中有如下调用：

  ```
  interrupt->Schedule(TimerHandler, (_int) this, TimeOfNextInterrupt(),
  TimerInt);
  (*handler)(arg);
  ```

  `interrupt->Schedule` 函数做了如下操作：

  - 设置时钟中断 `TimerHandler` 的发生时间为 `when = stats->totalTicks + TimeOfNextInterrupt()`
  - 创建一个新中断，其处理函数为 `TimerHandler`，参数为 `(_int) this`，发生时间为 `when`，中断类型为 `TimerInt`
  - 将创建的新中断插入到中断就绪队列中

  因此，当创建了一个时钟中断类后，其构建函数首先调用了 `timerHandler` 方法，然后该方法继续调用了 `TimerExpired` 方法，该方法插入了一个随机时间后会发生的中断，同时又调用了 `timerHandler` 方法执行中断，该方法又调用 `TimerExpired` 方法插入一个随机时间后会发生的中断，不断往复，实现了系统时钟中断发生和处理模块的构造。

- 计算下一个中断将要发生的时间的方法如下：

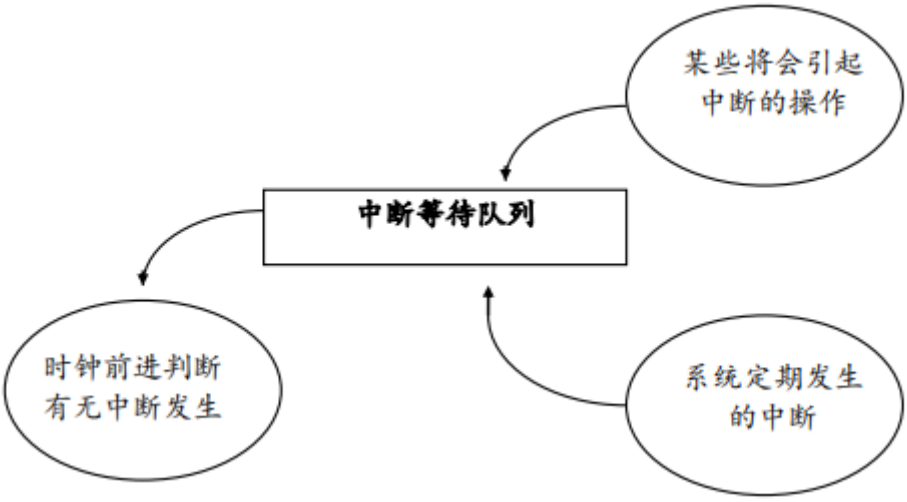
  ```
  ===============================================
  define TimerTicks   100
  ===============================================
  int Timer::TimeOfNextInterrupt()
  {
      if (randomize)
          return 1 + (Random() % (TimerTicks * 2));
      else
          return TimerTicks;
  }
  ===============================================
  int Random()
  {
      return rand();
  }
  ```

```
//----------------------------------------------
void RandomInit(unsigned seed)
{
    srand(seed);
}
//----------------------------------------------
```

在 `Nachos` 中，命令 `-rs` 的参数调用了 `RandomInit` 函数，生成了随机种子然后函数 `TimeOfNextInterrupt` 调用 `rand` 函数，基于随机种子生成了随机数 `Random()` ，最终得到的时间是一个 `1 ~ 201` 之间的数，即每隔这个时间就生成一个时钟中断，将当前线程让出处理机，调度下一个线程上处理机运行。

- 如何实现线程的轮转：

  当插入时钟中断到中断队列后，接下来就是执行中断的模块。中断等待队列是 Nachos 虚拟机最重要的数据结构之一，它记录了当前虚拟机可以预测的将在未来发生的所有中断。当系统进行了某种操作可能引起未来发生的中断时，如磁盘的写入、向网络写入数据等都会将中断插入到中断等待队列中；对于一些定期需要发生的中断，如**时钟中断**、终端读取中断等，系统会在中断处理后将下一次要发生的中断插入到中断等待队列中。中断的插入过程是一个优先队列的插入过程，其优先级是中断发生的时间，也就是说，**先发生的中断将优先得到处理**。当时钟前进或者系统处于 Idle 状态时，Nachos 会判断中断等待队列中是否有要发生的中断，如果有中断需要发生，则将该中断从中断等待队列中删除，调用相应的中断处理程序进行处理。



- 系统每执行一个操作，都会将时钟滴答一次：`OneTick`，而在该函数中，又存在如下语句：

```
while (CheckIfDue(FALSE));      // check for pending interrupts
```

该语句检查中断就绪队列中的中断，如果某条中断应该在当前时间点发生，则执行它；否则，什么也不做。实际上，中断处理类 `Timer` 的构造函数中使用了中断处理函数 `TimerInterruptHandler`，其方法定义如下：

```
static void TimerInterruptHandler(_int dummy)
{
    if (interrupt->getStatus() != IdleMode)
    interrupt->YieldOnReturn();
}
```

它表明，只要处理了该时钟中断，就需要当前线程让出处理机，具体的方法在函数 `YieldOnReturn` 中。

```
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# ./nachos -rs 1

Now all threads for producers and customers are created.


No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1453, idle 43, system 1410, 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...
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 0;
producer id --> 0; Message number --> 4;
producer id --> 1; Message number --> 2;
producer id --> 2; Message number --> 4;
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# cat tmp_1
producer id --> 1; Message number --> 1;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 3;
producer id --> 2; Message number --> 0;
producer id --> 2; Message number --> 1;
producer id --> 1; Message number --> 4;
producer id --> 2; Message number --> 2;
producer id --> 2; Message number --> 3;
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3#
```

```
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# ./nachos -rs 12

Now all threads for producers and customers are created.


No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 1555, idle 115, system 1440, 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...
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# cat tmp_0
producer id --> 0; Message number --> 0;
producer id --> 0; Message number --> 1;
producer id --> 0; Message number --> 2;
producer id --> 1; Message number --> 0;
producer id --> 0; Message number --> 4;
producer id --> 2; Message number --> 1;
producer id --> 1; Message number --> 1;
producer id --> 1; Message number --> 2;
producer id --> 2; Message number --> 2;
producer id --> 1; Message number --> 4;
producer id --> 2; Message number --> 3;
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3# cat tmp_1
producer id --> 2; Message number --> 0;
producer id --> 0; Message number --> 3;
producer id --> 1; Message number --> 3;
producer id --> 2; Message number --> 4;
root@ecs-a3e0:~/OS/nachos-3.4-SDU/code/lab3#
```