参考教材

[1] 费翔林，骆斌. 操作系统教程（第5版）[M]. 高等教育出版社，2014.

[2] Tanenbaum A S，Bos H. 现代操作系统（原书第4版）[M]. 陈向群，译，马洪兵 等，译.

[3] 陈海波，夏虞斌. 现代操作系统：原理与实现[M]. 机械工业出版社，2020. 机械工业出版社，2017.

[4] 任炬，张尧学，彭许红. openEuler操作系统[M]. 清华大学出版社，2020.

3.4 管程（Monitor）

- 管程和条件变量
- 基于管程的哲学家就餐问题
- 基于管程的生产-消费问题

- 管程和条件变量
- 基于管程的哲学家就餐问题
- 基于管程的生产–消费问题

3.4.1 管程和条件变量

- 信号量机制的缺点：进程自备同步操作，P(S)和V(S)操作大量分散在各个进程中，不易管理，易发生死锁
- 管程特点：管程（秘书进程，Monitors）封装了同步操作，对进程隐蔽了同步细节，简化了同步功能的调用界面。用户编写并发程序如同编写串行程序

3.4.1：基本思想

- 1974年和1977年，Hoare和Hansen根据抽象数据类型的原理提出了新的同步机制：
  - 把分散在各个进程中的临界区集中起来管理，并把共享资源用数据结构抽象地表示出来
  - 建立一个“秘书”程序管理到来的访问
  - “秘书”每次只让一个进程来访，后“秘书”更名为管程

3.4.1：条件变量

- 管程确保了进程的互斥访问，为了实现同步机制，则引入了 条件变量 的概念
- 条件变量是出现在管程内的全局数据结构，只能通过两个原语操作来控制它：
  - wait()：挂起调用进程并释放管程，直到另一个进程在条件变量上执行signal()
  - signal()：如果有其他进程因对条件变量执行wait()而被挂起，便释放之，如果没有进程在等待，那么信号不被保存

注意和PV操作中的signal, wait 函数的区别！！

3.4.1：条件变量（续）

- 条件变量虽然也是一种信号量，但它并不是P、V操作中纯粹的计数信号量
  - 没有与条件变量关联的值
  - 不能像信号量那样积累供使用，只用于维护等待进程队列
  - 当一个条件变量上不存在等待的进程时，signal操作为空
  - wait操作一般应在signal操作之前发出

管程 vs. 信号量

- 管程在语言层面，对进程的同步过程进行了抽象
  - 属于程序语言中的特殊类型（例如Java中的synchronized 关键字）
  - 由编译器识别，而非程序员来安排互斥操作，确保同一时刻只有一个进程/线程访问，防止有意或无意的违反同步操作
    - 当一个进程调用管程过程时，编译器会在该调用前增加若干指令，来检查在管程中是否存在有其他活跃进程。
    - 如果有，调用进程被阻塞，直到被其他进程唤醒；如果没有，则当前进程可以进入并执行管程过程。
  - 方便程序员来书写同步程序，也便于程序正确性验证

3.4.1：管程实现

```
1.  monitor mo{
2.      condition x, y;

3.      function F1(…){
4.          …
5.          x.wait();
6.      }

7.      function F2(){
8.          …
9.          y.signal();
10.     }

11.     initialization code(…){…}
12. }
```

思考：如果进程P调用了mo.F2()，激活了此前在等待条件变量 y 的进程Q，请问：P，Q 的执行顺序是什么？

3.4.1：管程实现

```
1.  monitor mo{
2.      condition x, y;

3.    function F1(…){
4.        …
5.        x.wait();
6.    }

7.    function F2(){
8.        …
9.        y.signal();
10.   }

11.   initialization code(…){…}
12. }
```

思考：如果进程P调用了mo.F2()，激活了此前在等待条件变量 y 的进程Q，请问：P，Q 的执行顺序是什么？

- Signal and wait:
- P退出cpu，待Q离开或者等待另一变量

- Signal and continue:
- Q待P离开或等待另一变量

- 管程和条件变量
- 基于管程的哲学家就餐问题
- 基于管程的生产-消费问题

3.* Dining-Philosophers

- 基本原理：不是对每只筷子设置条件变量，而是对每个哲学家设置条件变量。

monitor dp

```
1.  enum {THINKING, HUNGRY, EATING} states[5];
2.  condition self[5];

3.  void test(int i); //试图吃饭
4.  void pickup(int i); //拿起叉子
5.  void putdown(int i); //放下叉子
```

3.* Dining-Philosophers

- 基本原理：不是对每只筷子设置条件变量，而是对每个哲学家设置条件变量。

```
monitor dp
1.  enum {THINKING, HUNGRY, EATING} states[5];
2.  condition self[5];

3.  void test(int i); //试图吃饭
4.  void pickup(int i); //拿起叉子
5.  void putdown(int i); //放下叉子
```

```
1.  dp.pickup(i);
2.  …
3.     eat
4.  …
5.  dp.putdown();
```

3.* Dining-Philosophers

```
void pickup(int i)
1.   {
2.     state[i] = HUNGRY;
3.     test(i);
4.     if(state[i] != EATING){
5.       self[i].wait();
6.     }
7.   }
```

如果通过test()进入吃饭状态不成功，那么当前哲学家就在该信号量阻塞等待，直到其他的哲学家进程通过test()将该哲学家的状态设置为EATING。

3.* Dining-Philosophers

```
void test(int i)
1.  {
2.    if (state[(i+1)%5] != EATING
3.      && state[(i+4)%5] != EATING
4.      && state[i] = HUNGRY){
5.      state[i] = EATING;
6.      self[i].signal();
7.    }
8.  }
```

如果当前处理的哲学家处于饥饿状态且两侧哲学家不在吃饭状态，则当前哲学家通过 test()函数试图进入吃饭状态

3.* Dining-Philosophers

```
void putdown(int i)
1.  {
2.     state[i] = THINKING;
3.     test((i+1)%5);
4.     test((i+4)%5);
5.  }
```

当一个哲学家进程放下筷子的时候，会测试它的邻居。如果邻居处于饥饿状态，且该邻居的邻居不在吃饭状态，则该邻居进入吃饭状态。

- 管程和条件变量
- 基于管程的哲学家就餐问题
- 基于管程的生产–消费问题

3. ** Bounded-Buffer

class PorductBuffer

```
1.   int n; //缓冲区只有一个整数值
2.   boolean valueSet = false;
3.   synchronized int get() {
4.     if(!valueSet) wait();
5.     valueSet = false;
6.     notify();
7.     return n;
8.   }

9.   synchronized void put(int n) {
10.    if(valueSet) wait();
11.    this.n = n;
12.    valueSet = true;
13.    notify();
14.  }
```

3. ** Bounded-Buffer

class Producer implements Runnable

```
1.   ProductBuffer pb;
2.
3.   Producer(ProductBuffer _pb) {
4.     this.pb = _pb;
5.     new Thread(this, "Producer").start();
6.   }

7.   public void run() {
8.     int i = 0;
9.     while(true) {
10.      q.put(i++);
11.    }
12.  }
```

3. ** Bounded-Buffer

class Consumer implements Runnable

```
1.   ProductBuffer pb;
2.
3.   Consumer(ProductBuffer _pb) {
4.     this.pb = _pb;
5.     new Thread(this, "Consumer").start();
6.   }

7.   public void run() {
8.     while(true) {
9.       q.get();
10.    }
11.  }
```

3. ** Bounded-Buffer

class BoundedBufferProblem

```
1.  public static void main(String args[]) {
2.      PorductBuffer pb = new PorductBuffer();
3.      new Producer(pb);
4.      new Consumer(pb);
5.  }
```