作业6

1. “忙等待”这个词的含义是什么？操作系统中还有什么类型的等待？可以避忙等待吗?如何解决?

- “忙等待”（Busy Waiting）也称为“自旋等待”，是一种进程或线程在等待某个条件变为真时所采取的一种策略。在忙等待中，进程或线程会持续检查条件是否满足，这个过程通常是循环执行，每次检查之间几乎没有停顿，因此称为“忙等待”。这种方式会占用处理器时间，导致效率低下，因为它在条件不满足时也会占用CPU资源。
- 操作系统中除了忙等待，还有其他类型的等待策略，例如：

1. **阻塞等待**：当一个进程或线程等待某个事件或资源时，它会被置于阻塞状态，此时CPU可以执行其他任务。一旦等待的条件满足，进程或线程会被唤醒，继续执行。
2. **睡眠等待**：类似于阻塞等待，但进程或线程在被唤醒前会在一定时间内保持睡眠状态，不会立即占用CPU资源。
3. **信号量**：使用信号量（Semaphore）或互斥量（Mutex）等同步机制来实现等待。这些机制允许进程或线程在条件不满足时阻塞，条件满足时被唤醒。
4. **条件变量**：在某些高级的同步机制中，如条件变量（Condition Variable），进程或线程可以等待特定的条件成立，而不是忙等待。

- 避免忙等待的方法主要是通过使用阻塞、睡眠等待或同步机制，这些方法可以让出CPU给其他任务，从而提高系统效率。解决忙等待问题通常需要根据具体的应用场景选择合适的同步和等待策略。例如，在多线程编程中，可以使用锁（Locks）、信号量（Semaphores）、事件（Events）和条件变量（Condition Variables）等机制来避免忙等待。在操作系统的进程调度中，可以通过时间片轮转、优先级调度等策略来减少忙等待的发生。

2. 竞争条件在许多计算机系统中存在。考虑一个银行系统，它通过两个功能来维持账余额... 持账户余额：存款（金额）和取款（金额）。这两个函数传递的是要从银行账户余 额中存入或取出的金额。假设一对夫妻共用一个银行账户。同时，丈夫调用withdraw（）函数，妻子调用deposit（）函数。描述竞争状况是如何发生的，以及如何防止竞争状况的发生

竞争状况：

1. 答出两个函数不是原子操作，函数的执行需要多阶段，比如读取余额，修改余额，更新余额
2. 答出两个函数的执行阶段可能会交叉，导致余额出现不一致

如何防止（言之有理即可）：

1. **互斥锁（Mutexes）**：使用互斥锁可以确保同时只有一个线程可以执行关键代码段。在存款和取款操作前，获取互斥锁，操作完成后释放互斥锁。
2. **原子操作**：确保存款和取款操作是原子性的，即不可中断。这可以通过使用特定的硬件指令或语言提供的原子操作库来实现。
3. **事务**：将存款和取款操作作为数据库事务来处理，利用数据库的事务管理来确保操作的原子性、一致性、隔离性和持久性。
4. **软件事务内存（STM）**：在某些编程语言中，可以使用软件事务内存来管理并发操作，它会自动处理并发中的冲突。
5. **消息队列**：使用消息队列来序列化对共享资源的访问。例如，所有存款和取款请求可以放入一个队列中，然后由一个服务进程依次处理这些请求。
6. **乐观锁**：在某些情况下，可以使用乐观锁，它允许多个线程同时进行操作，但在更新数据时检查数据是否被其他线程修改过。如果数据在操作过程中被修改，则撤销操作并重试。

通过这些方法，可以有效地防止竞争条件的出现，并确保银行账户操作的准确性。

3. 考虑下图所示的分配和释放进程的代码示例。

a. 变量number_of_processes的修改会出现竞争条件

b.

```python
int allocate process(){
    int new pid;
    mutex.acquire();//锁的获取
    if (number_of_processes == MAX PROCESSES)
        mutex.release();//锁的释放
        return -1;
    else{
        ++number_of_processes;
    }
    mutex.release();//锁的释放
}
void release process() {
    mutex.acquire();//锁的获取
    --number_of_process;
    mutex.release();//锁的释放
}
```

c.

单纯替换为atomic_t变量仍然无法防止竞争，还需要保证number_of_processes变量的读取、判断和修改在一个原子操作内完成。例如多个进程同时调用 allocate_process() 函数,它们可能会在同一时间检查number_of_processes 是否等于 MAXPROCESSES ， 若全部通过了检查, 会有多个进程同时增加 number_of_processes , 这可能导致实际进程数超过限制。

4. 考虑一个版本的面包师算法，这个版本未使用变量choosing。该版本是否违反了互斥原则？解释原因。

违法了互斥原则。

由于去掉了 choosing 变量，导致number[i]的更新无法互斥，即多个进程同时更新 number[i] 时，无法正确地决定先后顺序，这导致它们可能会同时进入临界区。

正确的互斥要求是确保每个时刻最多只有一个进程在临界区内，而上述代码在去掉 choosing 后，无法完全避免多个进程同时进入临界区的情况。

5.

下面的问题曾用于一次测验：罗纪公园有一个恐龙博物馆和一个公 园。有m名旅客和n辆车，每辆车只能容纳1名旅客。旅客在博物馆中 一会儿后，排队乘坐旅行车。当一辆车可用时，它载入一名旅客，然 后绕公园行驶任意长时间。若辆车都已被旅客乘坐游玩，则想坐车的 旅客需要等待：若辆车已就绪，但没有旅客等待，那么这辆车等待。 使用信号量同步m名旅客进程和n辆车进程下面的代码框架是在教室 的地板上发现的。忽略语法错误和丢掉的变量声明，请判定它是否正确。P和V分别对应于semWait 和 semSignal 。

passenger_released信号量没有区分不同的车辆，这样可能导致passenger_released信号量的等待和释放无法一一对应，即A车调用V(passenger_released) 释放的乘客，可能是B车实际上搭载的乘客。

6. 下面对一个写者/多个读者问题的解法错在哪里?

这样的实现使得只要当前有读者正在读资源，后续的读者都可以陆续读取（只需要增加readcount）。写者必须等待所有读者完全退出才能开始写入（readcount为0的时候才释放写者的信号量），可能导致写者饥饿问题

7.

a. 安全。

顺序：T2 T0 T1 T3 T4 （答案不唯一）

	Allocation	Max	Need	Available
	ABCD	ABCD	ABCD	ABCD
T0	3141	6473	3332	2224
T1	2102	4232	2130	
T2	2413	2533	0120	

T3	4110	6332	2222	
T4	2221	5675	3454	

顺序	T2	T0	T1	T3	T4
Available	4 6 3 7	7 7 7 8	9 8 7 10	13 9 8 10	15 11 10 11

b. 不能

c. 能, T2 T1 T0 T3 T4

d. 能, T3 T2 T1 T0 T4

8. 选做

变量初始化 Python

```python
elves = 0
reindeer = 0
santaSem = Semaphore(0)
reindeerSem = Semaphore(0)
elfTex = Semaphore(1)
mutex = Semaphore(1)
```

elves（精灵）和reindeer（驯鹿）都是计数器，都受互斥锁mutex保护。 精灵和驯鹿得到互斥锁来修改计数器; 圣诞老人检查它们。

圣诞老人等待santaSem信号量，直到精灵或驯鹿向他发出信号。

驯鹿等待reindeerSem信号量，直到圣诞老人发出信号示意他们进入围场并拉车。

当三个精灵得到帮助时，精灵们使用elfTex防止额外的精灵进入。

圣诞老人 Python

```python
while True:
    santaSem.wait()
    mutex.wait()
    if reindeer >= 9:
        prepareSleigh()
        cnt = 0
        while cnt < 9:
            reindeerSem.signal()
            reindeer -= 1
            cnt += 1

    else if elves == 3:
        helpElves()
    mutex.signal()

```

当圣诞老人醒来时，他会检查这两个条件中的哪一个成立，并与驯鹿或等待的精灵打交道。 如果有九个驯鹿在等待，圣诞老人会调用prepareSleigh，然后向reindeerSem发出九次信号，让驯鹿调用getHitched。 如果有精灵等待，圣诞老人只会调用helpElves。 精灵没有必要等圣诞老人；一旦他们发出santaSem信号，他们就可以立即调用getHelp。

圣诞老人没有必要减少精灵的数量，因为精灵在外出的路上会这样做。

驯鹿 Python

```python
mutex.wait()
reindeer += 1
if reindeer == 9:
    santaSem.singal()
mutex.signal()
reindeerSem.wait()
getHitched()
```

第九只驯鹿向圣诞老人发出信号，然后加入到其他正在等待reindeerSem的驯鹿。 当圣诞老人发出信号时，驯鹿全部执行getHitched。

精灵 Python

```python
elfTex.wait()
mutex.wait()
elves += 1
if elves == 3:
    santaSem.signal()
else:
    elfTex.signal()
mutex.signal()

getHelp()

mutex.wait()
elves -= 1
if elves == 0:
    elfTex.signal()
mutex.signal()
```

前两个精灵在释放互斥锁的同时释放elfTex，但最后一个精灵拥有elfTex，禁止其他精灵进入，直到所有三个精灵都调用了getHelp。

最后一个离开的精灵释放elfTex，允许下一批精灵进入。