6.1

> 竞争条件为银行系统不能被同时访问，属于互斥竞争。
>
> 解决方法：使用初始量为1的信号量，伪代码如下

```
int sepo = 1
{
    P(sepo);
    deposit(amount);
    withdraw(amout);
    V(sepo);
}
```

6.5

> 为什么通过禁止中断来实现同步原语不适合于多处理器系统

如果禁止一个处理器的中断，则在其他处理器上交叉运行的语句仍然会产生同步问题，如果禁止所有处理器的中断，则会增加上下文切换的次数，增加CPU资源浪费。

6.8

> 请描述如何采用指令 `compare_and_swap()` 来实现互斥并满足有限等待要求。

```
int compare_and_swap(int *value, int expected, int new_value){
    int temp = *value;

    if(*value == expected)
        *value = new_value;
    return temp;
}
```

原理：声明一个全局布尔变量`lock`，并初始化为0。调用`compare_and_swap()`的第一个进程将`lock`设置为1,然后会进入临界区，因为lock原始值等于期待值，随后调用`compare_and_swap()`不会成功，因为`lock`不是预期值0。当一个进程退出临界区时，将`lock`设置为0才可以允许另一个进程进入临界区。

6.14

```
#define MAX_PROCESSES 255
int number_of_processes = 0;

/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;

   if (number_of_processes == MAX_PROCESSES)
      return -1;
   else {
      /* allocate necessary process resources */
      ++number_of_processes;

      return new_pid;
   }
}

/* the implementation of exit() calls this function */
void release_process() {
    /* release process resources */
    --number_of_processes;
}
```

图 6-23　分配和释放进程

> a. 指出竞争条件

`allocate_process()` 和 `release_process()` 都会对 `number_of_processes` 进行修改，存在互斥竞争问题

> b. 假设一个名为 `mutex` 的互斥锁，它有操作 `acquire()` 和 `release()` 。指出应该在哪里加锁以防止竞争条件

如下所示：

```
int mutex = 1;
int allocate_process(){
    int new_pid;
    if(number_of_process == MAX_PROCESSES)
        return -1;
    else{
        acquire(mutex);
        ++number_of_process;
        release(mutex);
        return new_pid;
    }
}
void release_process(){
    acquire(mutex);
    --number_of_process;
    release(mutex);
}
```

> c. 能否使用原子整数 `atomic_t number_of_process = 0` 来取代整数 `int number_of_process = 0` 来防止竞争条件

可以。因为`atomic_t`属于原子操作，不可以被中断，相当于关中断，可以防止互斥竞争。

6.17

> 说明如何在多处理器环境中采用指令`test_and_set()`来实现信号量操作`wait()`和`signal()`。该解决方案应具有最小的忙等待

对于所有进程都有一个全局布尔变量来记录当前状态，在所有处理器中都可以看到。和一个锁变量来指示是否上锁

公用数据结构：`boolean waiting[n]; boolean lock;`

都初始化为`false`。只有在`waiting[i] == false`或者`lock == false`的时候，`Pi`才可以进入临界区。只有当执行`test_and_set()`的时候，`key`的值才能变为`false`；所有其他进程变为等待。只有当正在执行的一个进程离开临界区的时候，`waiting[i]`才能变为`false`允许下一个进程进行访问，且每次只能有一个`waiting[i]`被设置为`false`，来满足互斥条件。

6.26

> 有一个文件被多个进程所共享，每个进程有一个不同的数。这个文件可被多个进程同时访问，且应满足如下的限制条件：所有访问文件的进程数之和应小于n。写一个管程，以协调这个文件的访问进程。

```
{
    int mutex = n;
    condition self[n];
    void init (int i){
        state[i] = false;
    }
    void running(int i){
        P(mutex);
        execute(process[i]);//访问临界区
        V(mutex);
    }
    void
}
```

6.33

6.33 习题 4.17 要求设计一个多线程程序，通过 Monte Carlo 技术估算 π。在那个习题中，要求创建一个线程，以便生成随机点，并将结果存入一个全局变量。一旦该线程退出，父线程执行计算，以估计 π 值。修改这个程序，以便创建多个线程，这里每个线程都生成随机点并确定点是否落入圆内。每个线程应更新所有落在圆内的点的全局计数。通过采用互斥锁，保护对共享全局变量的更新，以防止竞争条件。

> 利用多线程程序实现Monte Carlo估算PI

$$\pi = 4\times(\text{圆内点总数})/(\text{点的总数})$$

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<time.h>
#define TOTAL_NUM 10000000                    //设置点的总数
#define THREAD_NUM 2                          //线程总数
```

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;    //定义互斥锁
int circle_num = 0;                                  //统计圆内点数
int num = 0;                                         //计数用，防止程序出错

void* monte_carlo(void* tid)                         //统计函数，每个子线程算200遍
{
    srand((unsigned)time(NULL));
    int i = 0;
    while(i<TOTAL_NUM/THREAD_NUM)
    {
        pthread_mutex_lock(&lock);                   //申请互斥锁

        /*产生随机的x、y坐标值*/
        double pointX = (double)rand()/(double)RAND_MAX;
        double pointY = (double)rand()/(double)RAND_MAX;
        double l = pointX*pointX+pointY*pointY;

        /*判断是否位于圆内*/
        if(l<=1.0)
            ++circle_num;
        ++num;
        ++i;

        pthread_mutex_unlock(&lock);                 //释放互斥锁

    }
    return 0;
}

int main()
{
    pthread_t thread[THREAD_NUM];                    //线程定义
    int i, state;

    for(i = 0;i<THREAD_NUM;i++)                      //线程创建
    {
        printf("create thread%d\n", i);
        state = pthread_create(&thread[i], NULL, monte_carlo, NULL);
        if(state)
        {
            printf("error!\n");
            exit(-1);
        }
    }

    for(i = 0;i<THREAD_NUM;i++)                      //等待子线程完成
    {
        pthread_join(thread[i],NULL);
    }

    pthread_mutex_destroy(&lock);                    //销毁互斥锁

    /*将pi的结果打印*/
    printf("num = %d, circle_num = %d, pi = %.5lf\n", num, circle_num,
4.0*circle_num/TOTAL_NUM);

    return 0;
}
```

```
PS F:\Codefile of visual studio code> gcc get_pi.c -o get_pi -pthread
PS F:\Codefile of visual studio code> ./get_pi
create thread0
create thread1
create thread2
create thread3
create thread4
num = 10000000, circle_num = 7853635, pi = 3.14145
PS F:\Codefile of visual studio code>
```

得到了pi值，但精度不够高