西北工业大学 操作系统实验 实验报告

班号：10012006　　姓名：夏卓　　学号：2020303245

实验日期：2022/11/6　　实验名称：线程同步与互斥

一、实验目的

掌握 GeekOS 系统的线程同步与互斥原理，实现线程的同步与互斥。

二、实验要求

1. 在 GeekOS 中实现信号量，使用信号量保证用户程序运行时的线程同步，给出关键函数的代码以及实验结果。
2. 设计测试程序，验证线程同步和互斥的结果。
3. 回答课后思考题。第十一章第 1 题。

三、实验过程及结果

实现信号量关键在于实现四个信号量操作：

1. 创建信号量 Semaphore_Create()

该函数首先会检查请求创建的这个信号量的名字是否存在：

```
/* 根据信号量名检查信号量是否存在 */
pSemaphore isSemExistByName(char *nameSem, int nameLen)
{
    pSemaphore sem = g_semList.head;
    while (sem != NULL)
    {
        if (strncmp(sem->semaphoreName, nameSem, nameLen) == 0)
            break;
        sem = Get_Next_In_Semaphore_List(sem);
    }
    return sem;
}
```

如果存在，那么就把这个线程加入到这个信号量所注册的线程链表上；如果不存在，则分配内存给新的信号量，清空它的线程队列，并把当前的这个线程加入到它的线程队列中；设置注册线程数量为 1，初始化信号量的名字，值和信号量的 ID，并把这个信号量添加到信号量链表上，最后返回信号量的 ID：

```
/* 如果不存在则新建一个信号量 */
if (sem == NULL)
{
    sem = (pSemaphore)Malloc(sizeof(struct Semaphore));
    /* 清空线程队列 */
    Clear_Thread_Queue(&sem->waitingThreads);
    /* 清空线程队列，并把当前线程加入到该线程队列中 */
    sem->registeredThreadCount = 0;
    sem->registeredThreads[sem->registeredThreadCount++] = g_currentThread;
    /* 将新创建的信号量加入到信号量列表中 */
    Add_To_Back_Of_Semaphore_List(&g_semList, sem);
    /* 初始化信号量的名字，值和信号量的ID */
    strncpy(sem->semaphoreName, semName, MAX_SEMAPHORE_NAME);
    sem->value = initCount;
    /* 把这个信号量添加到信号量链表上 */
    g_curSID++;
    sem->semaphoreID = g_curSID;
    return sem->semaphoreID;
}
```

2. P 操作 Semaphore_Acquire()

该函数首先检查传入的信号量 ID 是否存在：

```
/* 根据信号量ID检查信号量是否存在 */
pSemaphore isSemExistBySID(int sid)
{
    pSemaphore sem = g_semList.head;
    while (sem != NULL)
    {
        if (sem->semaphoreID == sid)
            break;
        sem = Get_Next_In_Semaphore_List(sem);
    }
    return sem;
}
```

如果存在，接着检查当前线程是否注册使用了这个信号量：

```
int getIndexInRegThreadList(pSemaphore sem)
{
    int i;
    for (i = 0; i < sem->registeredThreadCount; i++)
        if (sem->registeredThreads[i] == g_currentThread)
            break;
    return (i != sem->registeredThreadCount ? i : -1);
}
```

如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就把信号量的值减去 1，如果减去 1 后信号量的值小于 0，那么就把当前线程放入这个信号量的等待队列上：

```
int threadIndex = getIndexInRegThreadList(sem);
/* 如果这两项检查任意一项失败了，那么就返回-1 */
if (threadIndex == -1)
    return -1;
/* 信号量减1 */
sem->value--;
/* 如果信号量的值小于0，则把当前线程放入这个信号量的等待队列上 */
if (sem->value < 0)
    Wait(&sem->waitingThreads);
```

3. V 操作 Semaphore_Release()

该函数与 P 操作大致相同。首先也是检查传入的信号量 ID 是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，那就把信号量的值加上 1，如果加上 1 后信号量的值小于或等于 0，则要把该信号量里等待队列上的一个线程唤醒：

```
int threadIndex = getIndexInRegThreadList(sem);
/* 如果这两项检查任意一项失败了，那么就返回-1 */
if (threadIndex == -1)
    return -1;
/* 信号量加1 */
sem->value++;
/* 如果信号量的值小于等于0，则把该信号量里等待队列上的一个线程唤醒 */
if (sem->value <= 0)
    Wake_Up_One(&sem->waitingThreads);
```

4. 销毁信号量 Semaphore_Destroy()

该函数首先也是检查传入的信号量 ID 是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就把该线程从这个信号量的注册线程数组中删除，并把注册的线程数量减去 1：

```
int threadIndex = getIndexInRegThreadList(sem);
/* 如果这两项检查任意一项失败了，那么就返回-1 */
if (threadIndex == -1)
    return -1;
/* 否则把该线程从这个信号量的注册线程数组中删除 */
int i;
for (i = threadIndex; i < sem->registeredThreadCount; i++)
{
    sem->registeredThreads[i] = sem->registeredThreads[i + 1];
}
sem->registeredThreads[--sem->registeredThreadCount] = NULL;
```

如果这个信号量的注册线程为 0 了，则把这个信号量从信号量链表中删除，并释放它的内存：

```
/* 如果这个信号量的注册线程为0 */
if (sem->registeredThreadCount == 0)
{
    /* 把这个信号量从信号量链表中删除 */
    Remove_From_Semaphore_List(&g_semList, sem);
    g_curSID--;
    /* 并释放它的内存 */
    Free(sem);
}
```

最终实验结果如下：

使用 workload.c 程序分别创建 ping 和 pong 两个线程：

```
Welcome to GeekOS!
$ workload rr 1
g_schedulingPolicy = 0
g_Quantum = 1
************* Start Workload Generator *********
Process Ping has been created with ID = 8
Process Pong has been created with ID = 9
Pong
Ping
Pong
Ping
Pong
Ping
Pong
Ping
Pong
Process Pong is done at time: 0
Ping
Process Ping is done at time: 1

Tests Completed at 2
$
```

Semtest:

```
$ semtest
Create_Semaphore()...
Create_Semaphore() returned 4
P()...
P() returned 0
P()...
P() returned 0
V()...
V() returned 0
Destroy_Semaphore()...
Destroy_Semaphore() returned 0
```

Semtest1:

```
$ semtest1
Semtest1 begins
p3 created
Produced 0
Consumed 0
Produced 1
Consumed 1
Produced 2
Consumed 2
Produced 3
Consumed 3
Produced 4
Consumed 4
p3 executed
```

Semtest2:

```
$ semtest2
Error! Semaphore ID is Invalid
+ Identified unauthorized call
Error! Semaphore ID is Invalid
+ Identified invalid SID
Create_Semaphore() called
Create_Semaphore() returned 4
P() called
P() returned 0
V() called
V() returned 0
Destroy_Semaphore() called
Destroy_Semaphore() returned 0
Error! Connot Find Semaphore with SID=4
+ Removed authority after finish
```

四、实验分析

实验结果 1 中，可以看到 ping、pong 两个线程在信号量的作用下实现了交替输出的效果，这是由于 ping、pong 两个线程中使用了两个同步信号量”ping”和”pong”：

```
ping = Create_Semaphore("ping", 1);
pong = Create_Semaphore("pong", 0);

for (i = 0; i < 5; i++)
{
  P(pong);
  P(scr_sem);
  Print("ping\n");
  V(scr_sem);
  for (j = 0; j < 35; j++);
  V(ping);
}
```

```
ping = Create_Semaphore("ping", 1);
pong = Create_Semaphore("pong", 0);

for (i = 0; i < 5; i++)
{
  P(ping);
  P(scr_sem);
  Print("pong\n");
  V(scr_sem);
  for (j = 0; j < 35; j++);
  V(pong);
}
```

由于初始时信号量”ping”为 1，”pong”为 0，因此只能先执行 P(ping)操作，从而打印”pong”字符串，然后执行 V(pong)操作，释放一个”pong”信号量，接着就可以执行 P(pong)操作打印”ping”字符串，然后释放一个”ping”信号量，重复上述步骤，便实现了交替输出的效果。

实验结果 2 执行 Semtest 线程中，主要演示了一个信号量初始值大于 1 的 P、V 操作，由于信号量的值初始化为 3，故支持两次 P 操作：

```
Print("P()...\n");
result = P(semkey);
Print("P() returned %d\n", result);

Print("P()...\n");
result = P(semkey);
Print("P() returned %d\n", result);

Print("V()...\n");
result = V(semkey);
Print("V() returned %d\n", result);
```

实验结果 3 执行 Semtest1 线程中，主要演示了一个生产者消费者模型，该线程分别启动了三个线程 p1、p2、p3，其中 p1 是生产者，p2 是消费者，p3 只有等到 p1、p2 执行完毕后方可执行：

```
prod_sem = Create_Semaphore ( "prod_sem" , 0 ) ;
cons_sem = Create_Semaphore ( "cons_sem" , 1 ) ;

for (i=0; i < 5; i++) {
  P(cons_sem);
  Print ("Produced %d\n",i) ;
  V(prod_sem);
}
```

```
cons_sem = Create_Semaphore ( "cons_sem" , 1 ) ;
holdp3_sem = Create_Semaphore ( "holdp3_sem", 0 ) ;

for (i=0; i < 5; i++) {
  P(prod_sem);
  Print ("Consumed %d\n",i) ;
  V(cons_sem);
}

V(holdp3_sem);
```

```
holdp3_sem = Create_Semaphore ("holdp3_sem", 0);

P(holdp3_sem);
P ( scr_sem ) ;
Print("p3 executed\n");
V( scr_sem );
V(holdp3_sem);
```

实验结果 4 执行 Semtest2 线程中，主要演示了操作系统能够对于不正确的信号量操作进行相应的处理和反馈：

错误 1：线程对未授权信号量进行操作

```
/* Unauthorized call */
result = P(0);
if (result<0)
  Print("+ Identified unauthorized call\n");
else
  Print("- Not checking for authority\n");
```

错误 2：线程对无效信号量 ID 进行操作

```
/* Invalid SID*/
result = P(-1);
if (result<0)
  Print("+ Identified invalid SID\n");
else
  Print("- Not checking for invalid SID\n");
```

错误 3：销毁信号量后再对其进行获取操作

```
Print("Destroy_Semaphore() called\n");
result = Destroy_Semaphore(semkey);
Print("Destroy_Semaphore() returned %d\n", result);

/* Unauthorized call */
result = V(semkey);
if (result<0)
  Print("+ Removed authority after finish\n");
else
  Print("- Not removed authority after finish\n");
```

五、所遇问题及解决方法

本次实验主要考察对信号量同步的理解，主要难点在于构造四个信号量操作函数，但由于书上已给出示例代码，因此只需多加理解、进行自主改写，难度并不大。实验关键在于分析并体会信号量同步的作用，设计测试程序，验证线程同步和互斥的结果，并给出合理的解释，由于示例测试程序已经给出，分析可知其分别验证了信号量同步的交替输出、多线程协作访问、生产者消费者模型，以及对于非法操作的处理和反馈制度。

六、思考与练习

1. 简要说明同步和互斥的关系。

互斥是指某一资源同时只允许一个访问者对其访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。互斥量的加锁和解锁必须由同一线程分别对应使用。

同步是指在互斥的基础上，通过其他机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。信号量可以由一个线程释放，另一个线程得到。

综上所述，同步是为了让多个进程协作一起完成某个任务，属于直接制约关系，而互斥是为了让进程间在共享临界区资源时互不影响，属于间接制约关系。