操作系统课程设计 实验报告

姓名：刘天祺

班级：07121502

学号：1320151097

学院：计算机学院

专业：物联网工程

日期：2018 年 4 月 7 日

实验三　生产者消费者问题

一、实验要求

通过编写多进程程序实现典型的生产者和消费者问题。实验要求如下：

- 完成 Windows 版本和 Linux 版本
- 一个大小为 3 的缓冲区，初始为空
- 2 个生产者
  - 随机等待一段时间，往缓冲区添加数据，
  - 若缓冲区已满，等待消费者取走数据后再添加
  - 重复 6 次
- 3 个消费者
  - 随机等待一段时间，从缓冲区读取数据
  - 若缓冲区为空，等待生产者添加数据后再读取
  - 重复 4 次
- 显示每次添加和读取数据的时间及缓冲区的状态

二、实验环境

2.1 Linux 环境

操作系统：Ubuntu 14.04.5 LTS 64bit

Shell：zsh 5.0.5 (x86_64-pc-linux-gnu)

编译器：gcc 4.8.5 (Ubuntu 4.8.5-2ubuntu1~14.04.1)

2.2 Windows 环境

操作系统：Windows 10 64bit

Shell：cmder 160710

编译器：gcc 3.4.5 (mingw-vista special r3)

三、实验步骤

3.1 在 Linux 环境实现

- 申请并初始化共享主存区

```
// 申请共享主存区
shmid = shmget(SHMKEY, BUFFER_SIZE, 0666|IPC_CREAT);
// 初始化共享主存区(数据缓冲区)
char *addr = shmat(shmid, 0, 0);
for (i=0; i<BUFFER_SIZE; i++){
    addr[i] = '-';
}
shmdt(addr);
```

- 申请并初始化信号量

```
// 申请信号量
full_id  = semget((key_t)FULL_KEY,  1, 0666|IPC_CREAT);
empty_id = semget((key_t)EMPTY_KEY, 1, 0666|IPC_CREAT);
mutex_id = semget((key_t)MUTEX_KEY, 1, 0666|IPC_CREAT);
// 初始化信号量
set_semval(full_id,  0);
set_semval(empty_id, 3);
set_semval(mutex_id, 1);
```

其中 full 和 empty 为同步信号，分别代表已生产项目数量和缓冲区的剩余空间 mutex 为互斥信号，是一个二值信号灯。

- 创建 5 个进程，前 2 个作为生产者，剩余的作为消费者

```
// 创建进程
pid_t pid;
for(i=0; i<5; i++){
    if( (pid = fork()) == -1){
        printf("fork failed\n");
        exit(0);
    } else if(pid == 0){
        // do nothing let i++ and do next fork
        continue;
    } else {
        if(i<2){
            producer();
            // only need one process to be create
            break;
        } else {
            consumer();
            break;
        }
    }
}
```

生产者在生产数据之前，对信号量 empty 执行 P 操作，使 empty 的值减 1，生产数据后，对信号量 full 执行 V 操作，使 full 的值加 1。若生产者在生产数据前，缓冲区已满(empty=0)，则 P(empty)操作将阻塞进程，直至 empty>0 后再执行生产数据的操作。通过同步信号量机制实现了“若缓冲区已满，则等待消费者取走数据后再添加”的实验要求。生产者部分代码如下：

```
void producer(){
    int i; char item;
    for(i=0; i<6; i++){
        usleep(randInt(SLEEP_TIME));
        printf("<%d> ", i+1); // 第几次生产
        item = produceItem();
        P(empty_id);
            P(mutex_id);
                showTime();
                putItemIntoBuffer(item);
                showShm();
            V(mutex_id);
        V(full_id);
        newLine();
    }
}
```

消费者在消费数据之前，对信号量 full 执行 P 操作，使 full 的值减 1，消费数据后，对信号量 empty 执行 V 操作，使 empty 的值加 1。若消费者在要消费数据前，缓冲区中没有生产者生产的数据(full=0)，则 P(full)操作将阻塞进程，直到 full>0 后再执行消费数据的操作。通过同步信号量机制实现了“若缓冲区为空，等待生产者添加数据后再读取”的实验要求。消费者部分代码如下：

```
void consumer(){
    int i; char item;
    for(i=0; i<4; i++){
        usleep(randInt(SLEEP_TIME));
        printf("[%d] ", i+1); // 第几次消费
        P(full_id);
            P(mutex_id);
                showTime();
                item = removeItemFromBuffer();
                showShm();
            V(mutex_id);
        V(empty_id);
        consumeItem(item);
        newLine();
    }
}
```

生产者(消费者)在生产(消费)数据的前后，分别对互斥信号量 mutex 执行 P 操作和 V 操作，这保证了生产(消费)过程的原子性质，可以避免多个生产者(消费者)之间对共享资源的竞争。

3.2 在 Windows 环境实现

- 建立并初始化共享主存区

```
//建立共享内存
HANDLE CurrentProcess = GetCurrentProcess();

hMap = CreateFileMapping(
    INVALID_HANDLE_VALUE,   //使用页式临时文件
    NULL,                   //默认的安全性
    PAGE_READWRITE,         //可读写权
    0,                      //最大尺寸（高32位）
    sizeof(*shm),           //最小尺寸（低32位）
    "buffer");              //该文件映射的作为共享内存的缓冲区，取名为“buffer”

//在文件映射上创建视图
LPVOID pData = MapViewOfFile(
    hMap,               //保存文件的对象
    FILE_MAP_WRITE,     //映射可读可写
    0,                  //在文件的开头处（高32位）开始
    0,                  //在文件的开头处（低32位）
    sizeof(*shm));      //整个文件要映射4个字节

if (pData != NULL) {
    ZeroMemory(pData, sizeof(shm));  //分配内存空间，并清零
}

// 初始化数据缓冲区
hMap = OpenFileMapping(FILE_MAP_WRITE, FALSE, "buffer");
shm = (struct SHM*)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, sizeof(*shm));
for (i = 0; i<3; i++) {
    strcpy(shm->buf[i], "-");
}
```

- 建立并初始化信号量

```
//建立信号量
SEM_FULL  = CreateSemaphore(NULL, 0, 3, "FULL");
SEM_EMPTY = CreateSemaphore(NULL, 3, 3, "EMPTY");
SEM_MUTEX = CreateSemaphore(NULL, 1, 1, "MUTEX");
```

- 建立 5 个子进程

```
//建立5个子进程
for (i = 0; i<5; i++) {
    nH[i] = StartClone(++nClone);
}
```

其中 StartClone()函数中调用 Win32API CreateProcess()创建子进程。

nClone 为进程的编号，其中 0 号为主进程，1-2 号为生产者进程，3-5 号为消费者进程。

在生产者进程中，首先获取信号量和共享主存区的句柄，用于对其进行操作，使用 WaitForSingleObject()实现 P 操作，ReleaseSemaphore()实现 V 操作。其余部分与 linux 基本相同。生产者部分代码如下：

```
else if (nClone > 0 && nClone < 3) {        // id:1,2 生产者
    //获得句柄
    SEM_EMPTY = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "EMPTY");
    SEM_FULL  = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "FULL");
    SEM_MUTEX = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "MUTEX");
    hMap = OpenFileMapping(FILE_MAP_WRITE, FALSE, "buffer");
    shm = (struct SHM*)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, sizeof(*shm));
    for (i = 1; i<=6; i++) {
        time = (1+randInt(nClone))*randInt(SLEEP_TIME);
        Sleep(time);
        WaitForSingleObject(SEM_EMPTY, INFINITE);   // P(empty)
        WaitForSingleObject(SEM_MUTEX, INFINITE);   // P(mutex)
        //向缓冲区添加产品,把0置为1
        char t_buf[2];
        for (j = 0; j<3; j++){
            if(strcmp(shm->buf[j], "-") == 0){
                sprintf(t_buf, "%c", 'A'+(randInt(26)+nClone*randInt(26))%26);
                strcpy(shm->buf[j], t_buf);
                break;
            }
        }
        SYSTEMTIME curtime;
        GetSystemTime(&curtime);
        //输出时间、操作、缓冲区状态
        printf("[%d] P%d\t%02d:%02d:%02d\t-> buf[%d] = %s\t"
            ,i, nClone, curtime.wHour, curtime.wMinute, curtime.wSecond, j, t_buf);
        for (j = 0; j<3; j++) {
            printf("%s ", shm->buf[j]);
        }
        printf("\n");
        ReleaseSemaphore(SEM_MUTEX, 1, NULL);   // V(mutex)
        ReleaseSemaphore(SEM_FULL,  1, NULL);   // V(full)
    }
    //关闭句柄
    CloseHandle(SEM_MUTEX);
    CloseHandle(SEM_EMPTY);
    CloseHandle(SEM_FULL);
    CloseHandle(hMap);
}
```

消费进程同理,代码如下：

```
else if (nClone>2 && nClone< 6) {       //id:3,4,5 消费者
    //获得句柄
    SEM_EMPTY = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "EMPTY");
    SEM_FULL  = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "FULL");
    SEM_MUTEX = OpenSemaphore(SEMAPHORE_ALL_ACCESS, NULL, "MUTEX");
    hMap = OpenFileMapping(FILE_MAP_WRITE, FALSE, "buffer");
    shm = (struct SHM*)MapViewOfFile(hMap, FILE_MAP_WRITE, 0, 0, sizeof(*shm));
    for (i = 1; i<=4; i++) {
        time = (1+randInt(nClone))*randInt(SLEEP_TIME);
        Sleep(time);
        WaitForSingleObject(SEM_FULL,  INFINITE);   // P(full)
        WaitForSingleObject(SEM_MUTEX, INFINITE);   // P(mutex)
        SYSTEMTIME curtime;
        GetSystemTime(&curtime);
        char t_buf[2];
        for (j = 2; j>=0; j--){
            if(strcmp(shm->buf[j], "-") != 0){
                strcpy(t_buf, shm->buf[j]);
                strcpy(shm->buf[j], "-");
                break;
            }
        }
        printf("<%d> C%d\t%02d:%02d:%02d\t<- buf[%d] = %s\t"
            ,i, nClone - 2, curtime.wHour, curtime.wMinute, curtime.wSecond, j, t_buf);
        for (j = 0; j<3; j++) {
            printf("%s ", shm->buf[j]);
        }
        printf("\n");
        ReleaseSemaphore(SEM_MUTEX, 1, NULL);   // V(mutex)
        ReleaseSemaphore(SEM_EMPTY, 1, NULL);   // V(empty)
    }
    CloseHandle(SEM_MUTEX);
    CloseHandle(SEM_EMPTY);
    CloseHandle(SEM_FULL);
    CloseHandle(hMap);
}
```

四、实验结果

4.1 在 Linux 环境下的实验结果

```
12:40 taqini@q /home/taqini/Desktop/os_ep/ep3/linux/src
$ ./bbp
times/pid       date            time            operation       shm
<1> 5985        2018-04-07      12:40:06        -> buf[0] = M   M - -
<2> 5985        2018-04-07      12:40:06        -> buf[1] = N   M N -
<1> 5984        2018-04-07      12:40:07        -> buf[2] = G   M N G
[1] 5986        2018-04-07      12:40:07        <- buf[2] = G   M N -
[2] 5986        2018-04-07      12:40:07        <- buf[1] = N   M - -
[1] 5987        2018-04-07      12:40:07        <- buf[0] = M   - - -
<3> 5985        2018-04-07      12:40:08        -> buf[0] = M   M - -
[1] 5988        2018-04-07      12:40:08        <- buf[0] = M   - - -
<2> 5984        2018-04-07      12:40:09        -> buf[0] = Q   Q - -
[2] 5987        2018-04-07      12:40:09        <- buf[0] = Q   - - -
<4> 5985        2018-04-07      12:40:09        -> buf[0] = N   N - -
[3] 5986        2018-04-07      12:40:09        <- buf[0] = N   - - -
<5> 5985        2018-04-07      12:40:09        -> buf[0] = U   U - -
[2] 5988        2018-04-07      12:40:09        <- buf[0] = U   - - -
<3> 5984        2018-04-07      12:40:10        -> buf[0] = A   A - -
[3] 5987        2018-04-07      12:40:10        <- buf[0] = A   - - -
<6> 5985        2018-04-07      12:40:11        -> buf[0] = J   J - -
[4] 5986        2018-04-07      12:40:11        <- buf[0] = J   - - -
<4> 5984        2018-04-07      12:40:11        -> buf[0] = H   H - -
[3] 5988        2018-04-07      12:40:11        <- buf[0] = H   - - -
<5> 5984        2018-04-07      12:40:13        -> buf[0] = T   T - -
[4] 5987        2018-04-07      12:40:13        <- buf[0] = T   - - -
<6> 5984        2018-04-07      12:40:13        -> buf[0] = Q   Q - -
[4] 5988        2018-04-07      12:40:13        <- buf[0] = Q   - - -
```

4.2 在 Windows 环境下的实验结果

```
λ .\bbp.exe
[1] P1  08:17:04        -> buf[0] = W   W - -
[2] P1  08:17:06        -> buf[1] = I   W I -
[1] P2  08:17:06        -> buf[2] = Z   W I Z
<1> C1  08:17:06        <- buf[2] = Z   W I -
[3] P1  08:17:07        -> buf[2] = Q   W I Q
<1> C3  08:17:07        <- buf[2] = Q   W I -
[4] P1  08:17:08        -> buf[2] = W   W I W
<1> C2  08:17:08        <- buf[2] = W   W I -
[2] P2  08:17:08        -> buf[2] = U   W I U
<2> C1  08:17:09        <- buf[2] = U   W I -
[5] P1  08:17:09        -> buf[2] = C   W I C
<2> C2  08:17:09        <- buf[2] = C   W I -
[3] P2  08:17:09        -> buf[2] = D   W I D
<2> C3  08:17:10        <- buf[2] = D   W I -
<3> C1  08:17:11        <- buf[1] = I   W - -
[6] P1  08:17:11        -> buf[1] = Q   W Q -
<3> C3  08:17:11        <- buf[1] = Q   W - -
[4] P2  08:17:12        -> buf[1] = H   W H -
<4> C1  08:17:13        <- buf[1] = H   W - -
<3> C2  08:17:14        <- buf[0] = W   - - -
[5] P2  08:17:15        -> buf[0] = L   L - -
[6] P2  08:17:17        -> buf[1] = D   L D -
<4> C3  08:17:17        <- buf[1] = D   L - -
<4> C2  08:17:20        <- buf[0] = L   - - -
```

五、实验总结

生产者与消费者问题的核心是“如何避免多进程对共享资源的竞争”，为了解决这个问题，引入了同步信号量来解决生产者进程和消费者进程之间对共享资源竞争的问题，引入了互斥信号量来解决多个生产者(或消费者)之间对共享资源竞争的问题。

最初我做这个实验的时候，没有使用 sys/sem.h 里面定义的信号量机制，而是将 3 个信号量放入共享主存区，在生产者或消费者生产或消费数据的前后对信号量进行加或减，若信号量的值小于 0 则阻塞。我希望通过这种方式让多个进程之间共享信号量的信息，但是结果还是出现因为各种竞争导致的程序结果错误。分析其原因，是因为信号量本是就是为控制共享主存区的访问而设置的，它不能作为共享主存区的一部分。

我做实验一(编译内核)时的源代码还留着，于是找到了信号量机制的源码/ipc/sem.c，粗略的浏览一下，大概是使用了很多种 lock，才实现了对临界区的保护，于是放弃了自己实现信号量的想法...

```
* Internals:
* - scalability:
*   - all global variables are read-mostly.
*   - semop() calls and semctl(RMID) are synchronized by RCU.
*   - most operations do write operations (actually: spin_lock calls) to
*     the per-semaphore array structure.
*   Thus: Perfect SMP scaling between independent semaphore arrays.
*         If multiple semaphores in one array are used, then cache line
*         trashing on the semaphore array spinlock will limit the scaling.
* - semncnt and semzcnt are calculated on demand in count_semcnt()
* - the task that performs a successful semop() scans the list of all
*   sleeping tasks and completes any pending operations that can be fulfilled.
*   Semaphores are actively given to waiting tasks (necessary for FIFO).
*   (see update_queue())
* - To improve the scalability, the actual wake-up calls are performed after
*   dropping all locks. (see wake_up_sem_queue_prepare(),
*   wake_up_sem_queue_do())
* - All work is done by the waker, the woken up task does not have to do
*   anything - not even acquiring a lock or dropping a refcount.
* - A woken up task may not even touch the semaphore array anymore, it may
*   have been destroyed already by a semctl(RMID).
* - The synchronizations between wake-ups due to a timeout/signal and a
*   wake-up due to a completed semaphore operation is achieved by using an
*   intermediate state (IN_WAKEUP).
* - UNDO values are stored in an array (one per process and per
*   semaphore array, lazily allocated). For backwards compatibility, multiple
*   modes for the UNDO variables are supported (per process, per thread)
*   (see copy_semundo, CLONE_SYSVSEM)
* - There are two lists of the pending operations: a per-array list
*   and per-semaphore list (stored in the array). This allows to achieve FIFO
*   ordering without always scanning all pending operations.
*   The worst-case behavior is nevertheless O(N^2) for N wakeups.
*/
```