黄玟瑜
19335074
huangmy73@mail2.sysu.edu.cn

作业五

操作系统原理, 2021 春

2021-04-21

任务一

完成教材习题 5.2、5.7、5.13、5.14、5.23、5.25。

习题 5.2

a.

编写程序如下：

```
#include <stdio.h>
void READCARD(char *);

FILE *temp1, *temp2;

/* 创建2个临时文件，分别为temp1、temp2 */


void read(){
    char inbuf[80];
    READCARD(inbuf);                          /* 读80列卡片 */
    while(inbuf!=NULL){

        fwrite(inbuf, sizeof(char), 80, temp1); /* 写入临时文件temp1 */

        fputc(' ', temp1);                    /* 在每个卡片后补充空格 */

        READCARD(inbuf);                      /* 读80列卡片 */
    }
}/* 第一个过程 */

void squash(){
    char prev = fgetc(temp1); /* 初始化prev读入第一个字符 */
    if (prev == EOF)
        EXIT();
    char curr = fgetc(temp1);

    while (curr != EOF)
    {
        if (prev == '*' && curr == '*') /* 出现相邻星号（**） */
        {
            fputc('↑', temp2);          /* 替换为字符'↑' */
            prev = fgetc(temp1);        /* 从当前光标处继续循环 */
            if (prev == EOF)
                EXIT();
            curr = fgetc(temp1);
            continue;
        }
        fputc(prev, temp2);     /* 将prev写入第二个临时文件temp2 */
        prev = curr;            /* 更新prev */
```

```
        curr = fgetc(temp1);    /* 更新curr */
    }
    fputc(prev, temp2);         /* 将最后一个字符写入文件 */
}/* 第二个过程 */

void print(){
    char outbuf[125];
    fread(outbuf, sizeof(char), 125, temp2);    /* 以每行125个字符进行打印 */
    OUTPUT(outbuf);
}/* 第三个过程 */

int main(){
    /* 串行执行 */
    read();
    squash();
    print();
    return 0;
}
```

b.

read 过程一次读入 80 个字符，对每一个字符转交给 squash 过程进行处理，处理完 80 个字符后补充空格，再继续循环，squash 过程完成将双星号替换为'↑' 的工作，它先判断当前字符是否为星号，若不是则传递当前字符给 print 过程打印当前字符，否则再调用 read 过程读入下一个字符，根据下一个字符进行条件判断，若下一个字符是星号说明出现了双星号，传递字符'↑' 给 print 过程进行打印，若下一个字符不是星号，则依次传递星号、下一个字符给 print 过程打印，这样就处理了当前字符，调用 read 过程继续读入字符，print 过程将 squash 过程传递过来的字符依次写入缓冲区，并在缓冲区满，即凑够 125 个字符后将字符打印。这三个过程协同合作，任一时刻只有一个过程在执行，共同完成任务，且节约了 I/O 和临时文件的开销。

c.

修改后的代码如下，若 READCARD 把一幅 80 个字符的图像放入 inbuf，read 过程返回 true，否则返回 false：

```
bool read(){
    while(true){
        READCARD(inbuf);
        for (int i = 0; i < 80;i++){
            rs = inbuf[i];
            RESUME squash;
            if (i == 79 && inbuf[i] == '\0')    /* 若正好80个字符，返回true */
                return true;
            if (i != 79 && inbuf[i] == '\0')
                return false;
        }
        rs = " ";
        RESUME squash;
    }
}

void squash(){
```

```
    while(true){
        if (rs == '\0')                 /* 接收到终止符，终止过程 */
        {
            sp = rs;
            RESUME print;
            return;
        }
        if(rs!="*"){
            sp = rs;
            RESUME print;
        }
        else{
            RESUME read;

            if(rs=="*"){
                sp = "↑";
                RESUME print;
            }
            else{
                sp = "*";
                RESUME print;
                if (rs == '\0'){          /* 接收到终止符，终止过程 */
                    sp = rs;
                    RESUME print;
                    return;
                }
            }
        }
        RESUME read;
    }

}
void print(){
    while(true){
        for (int j = 0; j < 125; j++){
            outbuf[j] = sp;
            if(sp=='\0'){               /* 接收到终止符，不满125也要打印，并终止过程 */
                OUTPUT(outbuf);
                return;
            }
            RESUME squash;
        }
    }
}
```

d.

编写程序如下：

```
char rs, sp;
char inbuf[80], outbuf[125];
semaphore read, squash, print;
read = 1;
squash = 0;
print = 0;
bool read(){
    while(true){
        READCARD(inbuf);
```

```
        for (int i = 0; i < 80;i++){
            rs = inbuf[i];
            sem_wait(&read);
            sem_signal(&squash);
        }
        rs = " ";
        sem_wait(&read);
        sem_signal(&squash);
    }
}

void squash(){
    while(true){
        sem_wait(&squash);
        if(rs!="*"){
            sp = rs;
            sem_signal(&print);
        }
        else{
            sem_signal(&read);
            sem_wait(&squash);
            if(rs=="*"){
                sp = "↑";
                sem_signal(&print);
            }
            else{
                sp = "*";
                sem_signal(&print);
                sp = rs;
                sem_signal(&print);
            }
        }
        sem_signal(&read);
    }
}
void print(){
    while(true){
        for (int j = 0; j < 125;j++){
            sem_wait(&print);
            outbuf[j] = sp;
            sem_signal(&squash);
        }
        OUTPUT(outbuf);
    }
}
```

习题 5.7

a.

初始化数组 choosing 和 number 分别为 false 和 0，每个数组的第 i 个元素可由进程 i 读或写，但其他进程只能读。对每个等待执行的进程，记其为第 i 个进程，choosing[i] 被置为 true 说明该进程被选中，它可以获得一个“票号”，这个票号存储在 number[i] 中，且票号的值是数组 number 中的最大票号的值 +1（因为所有待执行的进程都在 number 数组中有一个票号，进程的执行顺序依赖于票号的大小，票号小的进程先执行），拿到票号后 choosing[i] 被置为 false，接下来查看每个进程

的情况，对每一个进程，先查看其是否正在拿号（拿号过程中该进程会被选中），若其正在拿号则等待其取到号，之后判断其是否持有票且票号更小（票号相等时进程号更小），若是，继续等待；若不是，说明所有持有票的票号都更大，该轮到这个进程执行，该进程进入临界区，执行完毕后将票丢弃（将 number[i] 置为 0）。

执行过程如下：

```
//数组choosing和number分别被初始化为false和0
//进程i请求执行
    while(true){
        选定进程i
        分配一个票号存储在number[i]中
        结束选定
        for(int j = 0;j < n; j++){  //对每一个进程
            若进程j被选中，等待
            若进程j持有票且票号更小（或票号相等进程号更小），等待
        }
        执行进程i（临界区）
        将票号丢弃
        其余部分
    }
```

b.

由于对每个进程，别的进程拿号时必须先等待，且必须等待票号比自己更小的进程执行完毕后才能开始执行，保证了任何时刻最多只有一个进程在临界区内，也就是说，任意时刻只有一个进程在执行，从而避免了死锁。

c.

当一个进程在执行时，它的票号仍被持有，因此其他进程会一直等待直到该进程执行完毕后将票号丢弃，从而保证了在任意时刻只有一个进程在执行。

习题 5.13

a.

由于只使用了一个信号量，使用资源的进程在释放资源时唤醒等待的进程时未对计数器进行修改，对新的申请仍按照原来的计数器进行条件判断，从而导致过多的进程获得资源，发生冲突。

另一种想法是，所有对共享资源的访问都使用了同一个信号量 mutex 来实现临界区的保护，这样做看似没有问题，但由于各个进程拿到令牌的顺序无法预测，而共享数据之间隐式的相互依赖，会导致出错。

假设有 7 个进程申请资源，记为 $P1 \sim P7$，P1、P2、P3 先申请资源，它们可以立即获得资源，在最后一个释放资源之前（此时 must_wait 的值 true），P4、P5、P6 依次申请资源，它们在 block 保护的临界区前（Line 9）等待，此时 P7 开始申请资源，在 mutex 保护的临界区前（Line 5）等待，当 P1、P2、P3 中的最后一个进程释放资源时，它唤醒了 3 个被 block 的进程 P4、P5、P6，此时 P4、P5、P6 进入 mutex 保护的临界区前等待（Line 10），接着将 must_wait 置为 false，最后才将

mutex 释放，而这时 P4、P5、P6、P7 都在 mutex 保护的临界区前等待，若 P7 先进入临界区，由于 must_wait 被置为 false，它不会被阻塞，因此最多会有 4 个进程同时获得资源，这会使程序出错。

b.

若将第 6 行的 if 语句更换为 while 语句仍不能解决上述问题，阻塞的进程被唤醒后仍不会被认为是 active 状态，导致 must_wait 的值仍为 false，若有新的进程申请资源不会被阻塞，仍会导致超过 3 个进程同时获得资源。

习题 5.14

a.

当一个进程申请资源时，若 must_wait 的值为 false 则被认为是 active 状态，若 must_wait 的值为 true 则被阻塞，当使用进程的三个资源的的最后一个进程释放资源时他会唤醒被阻塞的最多 3 个进程，同时将它们认为是 active 状态，must_wait 的值依据 active 状态的进程是否等于 3 来确定，从而使新加入的进程被阻塞，从而避免了习题 5.13.a 所述情况。（只有 active 状态的进程才能获得使用资源）

b.

若已有 3 个进程正在使用资源，而后有数个进程依次申请资源且被阻塞，若最后一个离开的使用者离开时间在第 4 个后来申请资源的进程到来之前（即在第 1、2 或 3 个后来申请资源的进程到来后），因为已有进程数 ≤3，可以保证已有等待进程能得到资源，若第 4 个后来申请资源的进程到来后（或第 n>4 个后来申请资源的进程到来后）正在使用资源的进程未完全离开，当最后一个使用者离开时会随机唤醒处于等待状态的进程，所以仍有可能发生新到达的进程插入到已有等待进程前面获得资源的情况。

c.

在这种模式下，释放者为申请者修改计数器，使正在等待的进程被唤醒后立刻被认为是占用资源的状态，在新的申请者加入时可以依据条件将其阻塞。若不然，在只使用一个信号量的情况下，释放者释放资源后资源将会被认为是空闲的，在被唤醒的进程修改计数器之前若有新的进程加入会导致冲突的发生。

习题 5.23

编写程序，如下。（santa：圣诞老人 reindeer：驯鹿 elf：小精灵 sleigh：雪橇 problem：问题）定义信号量，常量和共享变量：

```
1 /* Semaphore */
2 #include <semaphore.h>
3 #define REINDEER 9  /* 驯鹿数量 */
4 #define ELVES 3     /* 小精灵数量*/
```

```
semaphore the3_elves, elf_mutex, rein_mutex, rein_semWait, sleigh, rein_done, elf_done, elf_semWait,
    santa, problem;
int rein_cnt, elf_cnt;
rein_cnt = 0;       /* 麻烦的小精灵数量 */
elf_cnt = 0;        /* 度假结束的驯鹿数量*/
the3_elves = 1;     /* 每次只能有3只小精灵问问题 */
elf_mutex = 1;      /* 更新遇到麻烦的小精灵数量 */
rein_mutex = 1;     /* 更新度假结束的驯鹿数量 */
rein_semWait = 0;   /* 驯鹿不想离开热带，尽可能待到最后可能的时刻 */
sleigh = 0;         /* 圣诞老人准备的雪橇 */
rein_done = 0;      /* 驯鹿派发礼物完毕 */
elf_semWait = 0;    /* 遇到麻烦的小精灵 */
santa = 0;          /* 睡在北极商店的圣诞老人 */
problem = 0;        /* 圣诞老人回答问题 */
elf_done = 0;       /* 小精灵问完问题 */
```

驯鹿进程：

```
/* Reindeer */
while(true){

    /* 在南太平洋度假 */

    /* 度假结束 */
    sem_wait(&rein_mutex);
    rein_cnt++;

    /* 9头驯鹿都度假回来 */
    if(rein_cnt==REINDEER){
        sem_signal(&rein_mutex);
        sem_signal(&santa);         /* 唤醒圣诞老人 */
    }
    else{
        sem_signal(&rein_mutex);
        sem_wait(&rein_semWait);    /* 不想离开热带，待到最后可能的时刻 */
    }

    /* 在套上雪橇之前，驯鹿会在温暖的棚子里待着 */
    sem_wait(&sleigh);

    /* 派发礼物 */

    /* 等待派发完毕 */
    sem_wait(&rein_done);

    /* 回南太平洋度假 */
}
```

小精灵进程：

```
/* Elf */
while(true){
    /* 每次只能有3只小精灵问问题 */
    sem_wait(&the3_elves);

    /* 遇到麻烦 */
    sem_wait(&elf_mutex);
    elf_cnt++;

    /* 3只小精灵遇到了麻烦 */
```

```
    if(elf_cnt ==ELVES){
        sem_signal(&elf_mutex);
        sem_signal(&santa);          /* 唤醒圣诞老人 */
    }
    else{
        sem_signal(&elf_mutex);
        sem_wait(&elf_semWait);      /* 凑不到3只小精灵时需要等待 */
    }

    /* 问问题 */
    sem_wait(&problem);

    /* 问完问题 */
    sem_wait(&elf_done);

    /* 问完问题的3只小精灵返回 */
    sem_signal(&the3_elves);
}
```

圣诞老人进程：

```
/* Santa */
int i;
while(true){
    sem_wait(&santa);   /* 等待唤醒

    /* 圣诞老人只有2种情形能被唤醒，并且在两种情况都发生时让小精灵等到圣诞节后，因为准备雪橇更重要 */
    /* (1) 所有9头驯鹿都从南太平洋度假归来 */
    if(rein_cnt==REINDEER){

        /* 将前8只待到最后可能的时刻的驯鹿唤回 */
        for (i = 0; i < REINDEER - 1; i++)
            sem_signal(&rein_semWait);

        /* 给驯鹿套上雪橇 */
        for (i = 0; i < REINDEER; i++)
            sem_signal(&sleigh);

        /* 派发礼物 */
        for (i = 0; i < REINDEER; i++)
            sem_signal(&rein_done);

        /* 让9头驯鹿回南太平洋度假 */
        sem_wait(&rein_mutex);
        rein_cnt = 0;
        sem_signal(&rein_mutex);
    }
    /* (2) 3个小精灵遇到了麻烦 */
    else{
        /* 将前2只在等待的小精灵唤醒 */
        for (i = 0; i < ELVES - 1; i++)
            sem_signal(&elf_semWait);


        for (i = 0; i < ELVES; i++){
            sem_signal(&problem);

            /*回答问题*/

            sem_signal(&elf_done);
```

```
        }

        /* 3只小精灵问完问题后返回*/
        sem_wait(&elf_mutex);
        elf_cnt = 0;
        sem_signal(&elf_mutex);
    }
}
```

习题 5.25

在读者程序中，当读者数量为 0，将要释放信号量 wrt 时使用了 Up 来释放，Up 用于释放内核空间里的信号量，sem_wait、sem_signal 是用户空间的获取、释放信号量操作，而信号量无法介于内核态和用户态使用，因此该解法出错。

任务二

编写程序，利用信号量实现有界缓存的生产者/消费者问题，要求有结果截屏和解释。

实验过程

核心代码如下所示：

```
sem_t remains, mutex, used;
sem_init(&remains, 0, BUFFERSIZE);
sem_init(&mutex, 0, 1);
sem_init(&used, 0, 0);
void* producer(){
    while(1){
        /* Produce */
        int num = rand() % 10;
        printf("PRODUCER produce: %d\n", num);

        sem_wait(&remains);
        sem_wait(&mutex);

        /* Append */
        buffer[in] = num;
        in = (in + 1) % BUFFERSIZE;
        printf("PRODUCER append: %d\n", num);

        sem_post(&mutex);
        sem_post(&used);
    }
}

void* consumer(){
    while(1){
        sem_wait(&used);
        sem_wait(&mutex);

        /* Take */
```

```
        int num = buffer[out];
        out= (out+ 1) % BUFFERSIZE;
        printf("CONSUMER take: %d\n", num);

        sem_post(&mutex);
        sem_post(&remains);

        /* Consume */
        printf("CONSUMER consume: %d\n", num);
    }
}
```

解释：

- remains 缓冲区剩余空间大小
- mutex 控制缓冲区读写
- used 缓冲区已占用空间大小
- 生产者随机产生一个随机数（produce），将随机数放入有界缓冲区（append）
- 消费者读取缓冲区的随机数（take），并将其打印（consume）

使用了 remains、mutex、used 这三个信号量解决有界缓存的生产者/消费者问题。
完整代码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <sys/time.h>
#include <unistd.h>
#define BUFFERSIZE 7

int buffer[BUFFERSIZE];
int in = 0, out = 0;

sem_t remains, mutex, used;

long thread_count;

void Get_args(int argc, char *argv[]);
void* producer();
void* consumer();

int main(int argc, char*argv[]){
    long       thread;  /* Use long in case of a 64-bit system */
    pthread_t* thread_handles;

    Get_args(argc, argv);
    thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
    /* 初始化过程 */
    sem_init(&remains, 0, BUFFERSIZE);
    sem_init(&mutex, 0, 1);
    sem_init(&used, 0, 0);

```

```
31      /* 让运行程序的头一半线程运行生产者程序，后一半运行消费者程序，事实上生产者和消费者数量也可以不一
            致 */
32      for (thread = 0; thread < thread_count / 2; thread++)
33          pthread_create(&thread_handles[thread], NULL,
34                         producer, NULL);
35
36      for (thread = thread_count / 2; thread < thread_count; thread++)
37          pthread_create(&thread_handles[thread], NULL,
38                         consumer, NULL);
39
40      for (thread = 0; thread < thread_count; thread++)
41          pthread_join(thread_handles[thread], NULL);
42
43      /* 使用完毕后释放信号量 */
44      sem_destroy(&remains);
45      sem_destroy(&mutex);
46      sem_destroy(&used);
47      free(thread_handles);
48
49      return 0;
50  }
51
52  void Get_args(int argc, char* argv[]) {
53      if (argc != 2)
54          printf("Miss thread_count.");
55      else
56          thread_count = strtol(argv[1], NULL, 10);
57  }  /* Get_args */
58
59  void* producer(){
60      while(1){
61          int n;
62
63          /* Produce */
64          int num = rand() % 10;
65          printf("PRODUCER produce: %d\n", num);
66
67          sem_wait(&remains);
68          sem_wait(&mutex);
69
70          /* Append */
71          buffer[in] = num;
72          in = (in + 1) % BUFFERSIZE;
73          sem_getvalue(&used,&n);
74          printf("PRODUCER append: %d when used: %d \n", num, n);
75
76          sem_post(&mutex);
77          sem_post(&used);
78      }
79  }
80
81  void* consumer(){
82      while(1){
83          int n;
84
85          sem_wait(&used);
86          sem_wait(&mutex);
87
88          /* Take */
```

```
89          int num = buffer[out];
90          out= (out+ 1) % BUFFERSIZE;
91          sem_getvalue(&remains,&n);
92          printf("CONSUMER take: %d when remains: %d \n",num, n);
93
94          sem_post(&mutex);
95          sem_post(&remains);
96
97          /* Consume */
98          printf("CONSUMER consume: %d\n", num);
99      }
100 }
```

实验结果

设置缓冲区大小为 4。

当有一个生产者和一个消费者的情况下，如下所示：

```
ehpc@f811ea9516cc:~/Documents$ ./os 2
PRODUCER produce: 3
PRODUCER append: 3 when used: 0
PRODUCER produce: 6
PRODUCER append: 6 when used: 1
PRODUCER produce: 7
PRODUCER append: 7 when used: 2
PRODUCER produce: 5
PRODUCER append: 5 when used: 3
PRODUCER produce: 3
CONSUMER take: 3 when remains: 0
CONSUMER consume: 3
CONSUMER take: 6 when remains: 1
CONSUMER consume: 6
PRODUCER append: 3 when used: 2
PRODUCER produce: 5
PRODUCER append: 5 when used: 2
CONSUMER take: 7 when remains: 0
CONSUMER consume: 7
CONSUMER take: 5 when remains: 1
CONSUMER consume: 5
CONSUMER take: 3 when remains: 2
CONSUMER consume: 3
CONSUMER take: 5 when remains: 3
CONSUMER consume: 5
PRODUCER produce: 6
PRODUCER append: 6 when used: 0
PRODUCER produce: 2
PRODUCER append: 2 when used: 1
PRODUCER produce: 9
CONSUMER take: 6 when remains: 2
CONSUMER consume: 6
CONSUMER take: 2 when remains: 2
CONSUMER consume: 2
PRODUCER append: 9 when used: 0
PRODUCER produce: 1
```

可以看到，生产者在 append 时缓冲区的使用空间总不是满的（used < 4），消费者在 take 时缓冲区总不是空的 (remains>0)。生产者 produce 的顺序和消费者 consume 的顺序相同。

当有两个生产者和两个消费者的情况下，如下所示：

```
ehpc@f811ea9516cc:~/Documents$ ./os 4
PRODUCER produce: 3
PRODUCER append: 3 when used: 0
PRODUCER produce: 7
PRODUCER append: 7 when used: 1
PRODUCER produce: 5
PRODUCER append: 5 when used: 2
PRODUCER produce: 3
CONSUMER take: 3 when remains: 1
CONSUMER consume: 3
CONSUMER take: 7 when remains: 1
PRODUCER produce: 6
CONSUMER consume: 7
PRODUCER append: 3 when used: 0
PRODUCER produce: 5
CONSUMER take: 5 when remains: 1
CONSUMER consume: 5
PRODUCER append: 6 when used: 0
PRODUCER produce: 6
PRODUCER append: 5 when used: 1
PRODUCER produce: 2
CONSUMER take: 3 when remains: 0
PRODUCER append: 6 when used: 1
CONSUMER consume: 3
CONSUMER take: 6 when remains: 0
CONSUMER consume: 6
PRODUCER produce: 9
PRODUCER append: 2 when used: 1
PRODUCER produce: 1
CONSUMER take: 5 when remains: 0
CONSUMER consume: 5
CONSUMER take: 6 when remains: 0
CONSUMER consume: 6
PRODUCER append: 9 when used: 0
PRODUCER produce: 2
PRODUCER append: 2 when used: 1
```

与前一种情况相似，生产者在 append 时缓冲区的使用空间总不是满的（used < 4），消费者在 take 时缓冲区总不是空的 (remains>0)。生产者 produce 的顺序和消费者 consume 的顺序相同。

但在缓冲区被占用的空间的平均值减小了。

(输出信号量的值仅仅只是一个大致参考，因为获取信号量的值（sem_getvalue）到打印输出该值 (printf) 这段时间内信号量的值仍有可能改变，因此输出的值不是最新的，但仍能分析出信号量的值的范围。)

综上，可知该算法实现了预期功能，生产者 produce 的顺序和消费者 consume 的顺序相同，这是由于该缓冲区用队列方式的实现，队列的读写模式为 FIFO，在多个生产者和多个消费者的情况下缓冲区的平均使用率减小了，分析可能的原因是消费者 consume 的平均速度更快。