lab4-1718405l4张福翔

联系方式：499480213@qq.com

1. 实验环境

- 自制OS的CPU：Intel 80386
- 模拟80386环境的虚拟机：QEMU
- 交叉编译的编译器：GCC
- 调试工具：GDB
- 运行平台：Ubuntu 18.04
- 编程语言：C，X86 Assembly

2. 实验要求

本实验通过实现一个简单的生产者消费者程序，介绍基于信号量的进程同步机制。

2.1 实现信号量相关系统调用

实现 `sem_init`，`sem_wait`，`sem_post`，`sem_destroy` 系统调用，并使用用户程序调试。

2.2 多进程进阶

调整框架代码中的 `gdt`、`pcb` 等进程相关的数据和代码，使得你实现的操作系统最多支持到 `8` 个进程(**包括内核idle进程**），简单但考虑细心程度，同时注意**进程分配内存空间大小**，不要越界，最好增量式(**学习git**)修改。

2.3 信号量进程同步进阶

理解**1.1**节中的测试程序，实现两个生产者、四个消费者的生产者消费者问题，不需要考虑**进程调度**，公平调度就行

- 以函数形式实现生产者和消费者，生产者和消费者的第一个参数为int类型，用于区分不同的生产者和消费者
- 每个生产者生产 `8` 个产品，每个消费者消费 `4` 个产品
- 先fork出6个进程，通过getpid获取当前进程的 `pid`，简单粗暴的以 `pid` 小的进程为生产者
- 信号量模拟buffer中的产品数量
- 保证buffer的互斥访问
- 每一行输出需要表达 `pid i, producer/consumer j, operation(, product k)`
- `operation` 包括 `try lock`、`locked`、`unlock`、`produce`、`try consume`、`consumed`
- 除了 `lock`、`unlock` 的其他操作需要包含括号中的内容
- `i` 表示进程号，`j` 表示生产者/消费者编号，`k` 表示生产者/消费者的生产/消费的第几个产品

3. 实验原理

内核维护 `Semaphore` 这一数据结构，并提供 `P`，`V` 这一对原子操作，其中 `W` 用于阻塞进程自身在该信号量上，`R` 用于释放一个阻塞在该信号量上的进程，其伪代码如下：

```
struct Semaphore {
    int value;
    ...
};

typedef struct Semaphore Semaphore;

void P(Semaphore *s) {
    s->value --;
    if (s->value < 0)
        W(s);
}

void V(Semaphore *s) {
    s->value ++;
    if (s->value <= 0)
        R(s);
}
```

3.1 `getpid` 系统调用

为了方便区分当前正在运行的进程，实验4要求先实现一个 `getpid` 系统调用用于返回当前进程标识 `pid`，不允许调用失败。

```
int getpid(void);
```

3.2 `sem_init` 系统调用

`sem_init` 系统调用用于初始化信号量，其中参数 `value` 用于指定信号量的初始值，初始化成功则返回 `0`，指针 `sem` 指向初始化成功的信号量，否则返回 `-1`。

```
int sem_init(sem_t *sem, uint32_t value);
```

3.3 `sem_wait` 系统调用

`sem_post` 系统调用对应信号量的 `V` 操作，其使得 `sem` 指向的信号量的 `value` 增一，若 `value` 取值不大于 `0`，则释放一个阻塞在该信号量上进程（即将该进程设置为就绪态），若操作成功则返回 `0`，否则返回 `-1`。

```c
int sem_post(sem_t *sem);
```

3.4 `sem_post` 系统调用

`sem_wait` 系统调用对应信号量的 `P` 操作，其使得 `sem` 指向的信号量的 `value` 减一，若 `value` 取值小于 `0`，则阻塞自身，否则进程继续执行，若操作成功则返回 `0`，否则返回 `-1`。

```c
int sem_wait(sem_t *sem);
```

3.5 `sem_destroy` 系统调用

`sem_destroy` 系统调用用于销毁 `sem` 指向的信号量，销毁成功则返回 `0`，否则返回 `-1`，若尚有进程阻塞在该信号量上，会带来未知错误。

```c
int sem_destroy(sem_t *sem);
```

4. 框架代码

```
.
├── app
│   ├── main.c
│   └── Makefile
├── bootloader
│   ├── boot.c
│   ├── boot.h
│   ├── Makefile
│   └── start.S
├── kernel
│   ├── include
│   │   ├── common
│   │   │   ├── assert.h
│   │   │   ├── const.h
│   │   │   └── types.h
│   │   ├── common.h
│   │   ├── device
│   │   │   ├── disk.h
│   │   │   ├── keyboard.h
│   │   │   ├── serial.h
│   │   │   ├── timer.h
│   │   │   └── vga.h
│   │   ├── device.h
│   │   ├── x86
│   │   │   ├── cpu.h
│   │   │   ├── io.h
```

```
|   |   |   ├── irq.h
|   |   |   └── memory.h
|   |   └── x86.h
|   ├── kernel
|   |   ├── disk.c
|   |   ├── doIrq.S
|   |   ├── i8259.c
|   |   ├── idt.c
|   |   ├── irqHandle.c
|   |   ├── keyboard.c
|   |   ├── kvm.c
|   |   ├── serial.c
|   |   ├── timer.c
|   |   └── vga.c
|   ├── lib
|   |   └── abort.c
|   ├── main.c
|   └── Makefile
├── lib
|   ├── lib.h
|   ├── syscall.c
|   └── types.h
├── Makefile
└── utils
    ├── genBoot.pl
    └── genKernel.pl
```

本次实验的框架与 `lab3` 几乎完全一致，我们所需要做的是在文件 `irqHandle.c` 中实现关于信号量的几个系统调用，在头文件中修改宏定义以提供更多的并发进程数，并在用户程序 `app/main.c` 中运行自身编写的测试用例即可。

5. 解决思路

5.1 信号量机制实现

在头文件 `memory.h` 中，我们给出了关于内核中存储的信号量数据结构：

```c
struct Semaphore {
    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this semaphore
};
```

其中 `state` 表示信号量是否已投入使用，`0` 表示该信号量未被创建，而 `1` 表示信号量正在被某些进程使用。而 `value` 字段表示信号量的当前取值。`ListHead` 结构指示双向循环队列中的一个表头节点，其用于连接所有因为等待该信号量而被阻塞的进程，因此，进程控制块中也同样包括一个这样的 `ListHead` 结构，尽管我们可以根据 `ListHead` 结构的地址位置，由进程控制块数组 `pcb` 的偏移量找出每个 `ListHead` 节点对应的进程号，但这种实现方法显得不太优雅，因此我们为 `ListHead` 数据结构额外地添加了一个字段 `pid`，用于显式地指出该节点对应的进程号。显然对表头结点而言，这个字段是无效的。

```
struct ListHead {
    struct ListHead *next;
    struct ListHead *prev;
    uint32_t pid;
};
```

考虑了信号量机制所用的数据结构后，我们就可以来实现信号量相关的系统调用。

5.1.1 `sem_init`系统调用实现

```
int sem_init(sem_t *sem, uint32_t value);
```

`sem_init`系统调用用于初始化一个信号量，并将其索引放在指针`sem`所指向的位置，信号量的初值为参数`value`。我们需要根据该系统调用提供的参数，将数据结构`Semaphore`设置好：

```
int i;
for (i = 0; i < MAX_SEM_NUM ; i++) {
    if (sem[i].state == 0) // not in use
        break;
}
if (i != MAX_SEM_NUM) {
    sem[i].state = 1;
    sem[i].value = (int32_t)sf->edx;
    sem[i].pcb.next = &(sem[i].pcb);
    sem[i].pcb.prev = &(sem[i].pcb);
    // sem[i].pcb.pid = current;
    pcb[current].regs.eax = i;
}
else
    pcb[current].regs.eax = -1;
return;
```

我们先检查是否还有可供分配的空余信号量，如果内核中的信号量数组已经用满，我们就直接返回调用失败。否则，我们选中索引为`i`的信号量结构，并填充其内容，将状态`state`设置为`1`，将`value`设置为相应的值，并让双向循环队列初始化，随后返回索引`i`，该索引`i`最终会被设置到指针`sem`所指向的位置里。

5.1.2 `sem_wait`系统调用

```
int sem_wait(sem_t *sem);
```

`sem_wait`系统调用用于进程向内核申请一个信号量资源，我们先从参数`sem`中获取对应的信号量索引，然后在内核信号量表项中找到这一项，如果其状态`state`不为`1`，表示该信号量并没有被使用，系统调用出错并返回`-1`。将其`value`减一，如果此时`value`小于`0`，则表明当前进程会由于得不到其所需的资源而被阻塞，我们将该进程加入到信号量的阻塞等待队列中：

```
int i = sf->edx;
```

```
if (sem[i].state == 0)
{
    pcb[current].regs.eax = -1;
    return;
}
sem[i].value--;
if (sem[i].value < 0)
{
    // add the process to the semaphore block list
    // insert at the tail of list
    struct ListHead *tail = sem[i].pcb.prev;
    tail->next = &pcb[current].blocked;
    sem[i].pcb.prev = &pcb[current].blocked;
    pcb[current].blocked.next = sem[i].pcb.prev;
    pcb[current].blocked.prev = tail;
    pcb[current].blocked.pid = current;

    // block current process
    pcb[current].sleepTime = -1;
    pcb[current].state = STATE_BLOCKED;

    // schedule to another process
    asm volatile("int $0x20");
}
pcb[current].regs.eax = 0;
return;
```

我们需要将该进程的进程控制块中对应节点 `blocked` 加入到等待队列的尾部，因此我们通过队列头的 `prev` 字段获取该队列尾部节点，将尾部节点的 `next` 字段指向 `blocked`，将队列头的 `prev` 字段指向 `blocked`，然后修改 `blocked` 的 `next` 和 `prev`，分别使其指向队列头部和原队列头部，这样该节点就成为了新的队尾节点。随后我们修改 `blocked` 的 `pid` 字段为该进程的 `id`（该步也可以直接在初始化 `pcb` 时进行）。然后，我们将当前进程设置为阻塞状态，`sleepTime` 标记为 `-1` 来表示其阻塞与时钟中断无关。随后我们调用访管指令 `int` 触发时钟中断，使得我们得以切换到下一个进程进行执行。

另外，无论当前进程是否需要阻塞，我们都将返回值置为 `0`，表示调用成功。

5.1.3 `sem_post` 系统调用实现

```
int sem_post(sem_t *sem);
```

`sem_post` 系统调用用于当前进程归还指针 `sem` 指向的信号量对应的资源。我们从 `sem` 中获得对应信号量表项中的索引，如果索引对应的信号量 `state` 字段不为 `1`，表明该信号量实际没有投入使用，调用出错，返回 `-1`。否则，我们即让对应信号量的 `value` 加一，如果增加后 `value` 仍然不是正数，说明当前进程归还的信号量可以被某个之前等待该信号量的进程使用，我们从双向循环队列中取出队首的进程节点，并将其恢复为可执行状态：

```
int i = sf->edx;
if (sem[i].state == 0)
{
    pcb[current].regs.eax = -1;
```

```
    return;
}
sem[i].value++;
if (sem[i].value <= 0)
{
    // wake a process in the block list
    if (sem[i].pcb.next == &sem[i].pcb)
    {
        // if no process waiting
        // something wrong
        pcb[current].regs.eax = -1;
        return;
    }
    // remove the head of list
    struct ListHead *p = sem[i].pcb.next;
    sem[i].pcb.next = p->next;
    p->next->prev = &sem[i].pcb;
    p->next = p;
    p->prev = p;
    uint32_t wake = p->pid;
    pcb[wake].state = STATE_RUNNABLE;
    pcb[wake].sleepTime = 0;
}
pcb[current].regs.eax = 0;
return;
```

如果执行 `value++` 后有 `value <= 0`，而我们却在信号量的等待队列中发现队列为空，表明没有进程在等待该信号量，这说明出现了某些异常错误，我们直接返回调用出错。否则，我们移出队首的节点，即代码中对应指针 `p` 所指向的节点。我们将队列头的 `next` 指向 `p` 的 `next` 节点，将 `p` 的 `next` 节点的 `prev` 指针指向队列头节点，从而从队列中移出了 `p` 所指向的节点。随后让指针 `p` 的两个指针都指向自己以回归初始状态。我们取出 `p` 对应的 `pid` 字段，这即是我们需要唤醒的进程，我们将其状态 `state` 重新设置为可运行的，并清空其 `sleepTime` 字段。随后设置调用成功的返回值 `0`。

5.1.4 `sem_destroy` 系统调用实现

```
int sem_destroy(sem_t *sem);
```

`sem_destroy` 用于摧毁一个信号量，其步骤较为简单，我们从指针 `sem` 中获取对应的信号量索引，并将该信号量表项的状态设置为 `0`，并设置返回值为调用成功 `0` 即可。另外，考虑到调用失败，我们可以先判断该信号量的状态是否为 `1`，如果不是 `1` 就使其调用失败，返回 `-1`。

```
int i = sf->edx;
if (sem[i].state == 1)
{
    pcb[current].regs.eax = -1;
    return;
}
sem[i].state = 0;
pcb[current].regs.eax = 0;
return;
```

5.2 多进程进阶

我们需要修改内核的相关框架代码，使其从支持 `4` 个进程数变为支持 `8` 个进程数，我们在之前的实验中了解到，内核支持的进程数与我们的宏 `MAX_PCB_NUM` 有关，因此，我们查看 `MAX_PCB_NUM` 的定义：

```
#define MAX_PCB_NUM ((NR_SEGMENTS-2)/2)
```

可以看出其定义与另一个宏 `NR_SEGMENTS` 的定义有关，我们查看这个宏，发现其被定义为 `10`，因此经过计算，我们知道当前内核支持 `4` 个进程。所以我们想让内核支持 `8` 个进程，通过简单的计算，我们知道需要让宏 `NR_SEGMENTS` 的值为 `18`，因此我们修改其值为 `18`。实际上，`NR_SEGMENTS` 宏的实际意义是段表的个数，我们不难发现，此时段表的使用情况是：`0` 为空段表，`1-2` 是内核使用的代码段和数据段表，往后每两个段表分别表示一个用户进程的代码段和数据段表，最后一个段表为 `TSS` 使用段表。

另外需要注意，我们需要防止修改后，内核所占空间超出范围（我们在此前规定内核的地址空间范围是 `0x100000-0x200000`），我们可以考虑将内核栈的大小减少为此前的一半，即从原先的 `1024` 栈大小变为 `512`，尽管这样处理后，由于 `pcb` 还有一些其他字段，会导致内核使用空间还是比修改前的更大，但增大的数目不多，不会导致超出内核地址空间的事情发生。

5.3 测试用例 `1` 调试

第一个测试用例由框架代码给出，提供了两个进程的简单同步测试：

```
int i = 4;
int ret = 0;
sem_t sem;
printf("Father Process: Semaphore Initializing.\n");
ret = sem_init(&sem, 2);
if (ret == -1) {
    printf("Father Process: Semaphore Initializing Failed.\n");
    exit();
}
ret = fork();
if (ret == 0) {
    while( i != 0) {
        i --;
        printf("Child Process: Semaphore Waiting.\n");
        sem_wait(&sem);
```

```
        printf("Child Process: In Critical Area.\n");
    }
    printf("Child Process: Semaphore Destroying.\n");
    sem_destroy(&sem);
    exit();
}
else if (ret != -1) {
    while( i != 0) {
        i --;
        printf("Father Process: Sleeping.\n");
        sleep(128);
        printf("Father Process: Semaphore Posting.\n");
        sem_post(&sem);
    }
    printf("Father Process: Semaphore Destroying.\n");
    sem_destroy(&sem);
    exit();
}
return 0;
```

从代码中我们可以看到，我们初始化了一个初值为 `2` 的信号量，并创建一个进程，子进程使用 `sem_wait` 系统调用申请信号量，而父进程使用 `sem_post` 系统调用，并根据 `sleep` 函数周期性地释放信号量，整个过程重复 `4` 次循环。由于子进程对信号量的申请没有等待时间，因此其会不断地向内核申请信号量，在使用完一开始的 `2` 个信号量资源后，子进程就被迫进入等待队列，直到父进程的睡眠结束，释放出一个信号量，并唤醒子进程，子进程才能重获信号量资源，并进入临界区。而父进程再次陷入睡眠，子进程的下一次申请信号量又陷入阻塞，直至父进程睡眠结束并释放一个信号量资源。子进程获取资源进入临界区后，完成工作，并销毁这个信号量。所以对父进程来说，其最后两次释放信号量的操作与摧毁信号量的调用都是失败的。我们通过在 `WriteStdOut` 系统调用中使用 `putChar` 命令，将 `qemu` 输出结果从串口输出到终端，即可记录程序执行结果：

```
Father Process: Semaphore Initializing.
Father Process: Sleeping.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Waiting.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Child Process: In Critical Area.
Child Process: Semaphore Destroying.
Father Process: Semaphore Posting.
Father Process: Sleeping.
Father Process: Semaphore Posting.
Father Process: Semaphore Destroying.
```

其结果与我们的分析一致，即子进程可以迅速地在前两次进入临界区，但后两次只有在父进程的`sleep`结束，并使用`sem_post`归还资源后才能重新进入临界区。另外，子进程先结束循环并摧毁信号量，这会使得父进程最后的两次`sem_post`和`sem_destroy`操作无效。

5.4 `git`的使用与分支创建

我们使用`git`来保留该测试用例的测试代码，创建`.gitignore`文件来忽视项目中产生的二进制文件。我们创建分支`test1.1`用于在`app/main.c`中保留实验任务1.1的测试用例，`git`分支情况如下：

```
sicer@ubuntu:~/oslab/lab4$ git branch
* master
  test1.1
```

我们使用`master`分支来继续下面的实验操作。

5.5 信号量进程同步进阶

为了实现测试用例，我们需要先实现一个`getpid`的系统调用，用于向用户进程返回自身进程的`pid`，其具体实现非常简单，我们只需根据当前执行的进程号`current`，将其赋值给相应的返回寄存器即可。

```
void syscallGetPid(struct StackFrame *sf) {
    sf->eax = current;
    return;
}
```

另外，我们在用户调用库`lib.h`和系统调用函数文件`syscall.c`中添加对该系统调用的封装函数：

```
int getpid()
{
    return syscall(SYS_GETPID, 0, 0, 0, 0, 0);
}
```

我们需要通过进程同步实现一个**生产者消费者问题**，为此我们需要使用`3`个信号量，分别代表一个**互斥锁**、一个表示**临界区空位数**的信号量和一个表示**临界区产品数**的信号量，我们分别用全局变量`mutex`，`spaces`和`items`表示。

```
sem_t mutex;
sem_t items;
sem_t spaces;
```

我们在用户进程中调用`sem_init`函数对其分别进行初始化，将`mutex`初始化为`1`，`items`初始化为`0`，`spaces`初始化为临界区的总大小，我们定义宏`USER_BUFFER_SIZE`表示临界区大小，理论上这个大小可以取任意合理的正值，我们在本实验中将其定义为`3`。

在信号量初始化完成后，我们对用户进程 `1` 使用 `fork` 系统调用，使其复制出 `6` 个进程，对应 `pid` 的值为 `2-7`，我们让 `pid` 为 `2` 和 `3` 的进程作为生产者， `pid` 为 `4-7` 的进程作为消费者，分别调用对应函数。我们可以采用以下的方法来简单地创建 `6` 个进程：

```
// fork 6 process
int ret = 1;
for (int i=0; i<6; i++)
    if (ret > 0)
        ret = fork();
    else
        break;
```

我们先考虑生产者函数的实现，我们对其进行 `8` 次循环生产，为了体现生产与消费的关系，我们为每次生产前添加了 `10` 时钟单位的延迟。生产时，我们先请求空位信号量 `spaces` ，再请求互斥锁信号量 `mutex` ，随后通过 `printf` 系统调用作为生产过程，生产结束后，归还互斥锁，并释放一个产品信号量 `items` 。

```
void producer_func(uint32_t pid)
{
    int no = pid - 1;
    for (int i = 0; i < 8; i++)
    {
        sleep(10);
        sem_wait(&spaces); // ask for buffer space
        printf("pid%d Producer%d: try lock\n", pid, no);
        sem_wait(&mutex); // ask for mutex
        printf("pid%d Producer%d: locked\n", pid, no);

        printf("pid%d Producer%d: produce product %d\n", pid, no, i);

        printf("pid%d Producer%d: unlock\n", pid, no);
        sem_post(&mutex); // return mutex resource
        sem_post(&items); // signal item resource
    }
}
```

消费者函数的实现与其类似，我们进行 `4` 次循环消费，先请求产品信号量 `items` 保证临界区中有可以消费的产品，再请求互斥锁 `mutex` 以访问临界区，通过 `printf` 作为消费过程，最后归还互斥锁 `mutex` ，并释放一个空位信号资源 `spaces` 。

```
void consumer_func(uint32_t pid)
{
    int no = pid - 1 - NR_PRODUCER;
    for (int i = 0; i < 4; i++)
    {
        printf("pid%d Consumer%d: try consumed %d\n", pid, no, i);
        sem_wait(&items); // ask for a item in buffer
        printf("pid%d Consumer%d: try lock\n", pid, no);
        sem_wait(&mutex); // ask for mutex
        printf("pid%d Consumer%d: locked\n", pid, no);

        printf("pid%d Consumer%d: product %d consumed\n", pid, no, i);
```

```
13
14          printf("pid%d Consumer%d: unlock\n", pid, no);
15          sem_post(&mutex); // return mutex
16          sem_post(&spaces); // signal space resource
17      }
18  }
```

对于父进程，即用户进程 `1`，我们让其主动睡眠一个足够大的值，在睡眠结束后摧毁这三个信号量。

```
1  sleep(10000); // wait a long time till process 2-7 end
2  // destroy all the semaphores
3  sem_destroy(&mutex);
4  sem_destroy(&items);
5  sem_destroy(&spaces);
6  exit();
```

运行该测试用例，我们即可获取输出结果：

```
1  pid4 Consumer1: try consumed 0
2  pid5 Consumer2: try consumed 0
3  pid6 Consumer3: try consumed 0
4  pid7 Consumer4: try consumed 0
5  pid2 Producer1: try lock
6  pid2 Producer1: locked
7  pid2 Producer1: produce product 0
8  pid2 Producer1: unlock
9  pid3 Producer2: try lock
10 pid3 Producer2: locked
11 pid3 Producer2: produce product 0
12 pid3 Producer2: unlock
13 pid4 Consumer1: try lock
14 pid4 Consumer1: locked
15 pid4 Consumer1: product 0 consumed
16 pid4 Consumer1: unlock
17 pid4 Consumer1: try consumed 1
18 pid5 Consumer2: try lock
19 pid5 Consumer2: locked
20 pid5 Consumer2: product 0 consumed
21 pid5 Consumer2: unlock
22 pid5 Consumer2: try consumed 1
23 pid2 Producer1: try lock
24 pid2 Producer1: locked
25 pid2 Producer1: produce product 1
26 pid2 Producer1: unlock
27 pid3 Producer2: try lock
28 pid3 Producer2: locked
29 pid3 Producer2: produce product 1
30 pid3 Producer2: unlock
31 pid6 Consumer3: try lock
32 pid6 Consumer3: locked
33 pid6 Consumer3: product 0 consumed
34 pid6 Consumer3: unlock
```

```
pid6 Consumer3: try consumed 1
pid7 Consumer4: try lock
pid7 Consumer4: locked
pid7 Consumer4: product 0 consumed
pid7 Consumer4: unlock
pid7 Consumer4: try consumed 1
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 2
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 2
pid3 Producer2: unlock
pid4 Consumer1: try lock
pid4 Consumer1: locked
pid4 Consumer1: product 1 consumed
pid4 Consumer1: unlock
pid4 Consumer1: try consumed 2
pid5 Consumer2: try lock
pid5 Consumer2: locked
pid5 Consumer2: product 1 consumed
pid5 Consumer2: unlock
pid5 Consumer2: try consumed 2
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 3
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 3
pid3 Producer2: unlock
pid6 Consumer3: try lock
pid6 Consumer3: locked
pid6 Consumer3: product 1 consumed
pid6 Consumer3: unlock
pid6 Consumer3: try consumed 2
pid7 Consumer4: try lock
pid7 Consumer4: locked
pid7 Consumer4: product 1 consumed
pid7 Consumer4: unlock
pid7 Consumer4: try consumed 2
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 4
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 4
pid3 Producer2: unlock
pid4 Consumer1: try lock
pid4 Consumer1: locked
pid4 Consumer1: product 2 consumed
```

```
pid4 Consumer1: unlock
pid4 Consumer1: try consumed 3
pid5 Consumer2: try lock
pid5 Consumer2: locked
pid5 Consumer2: product 2 consumed
pid5 Consumer2: unlock
pid5 Consumer2: try consumed 3
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 5
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 5
pid3 Producer2: unlock
pid6 Consumer3: try lock
pid6 Consumer3: locked
pid6 Consumer3: product 2 consumed
pid6 Consumer3: unlock
pid6 Consumer3: try consumed 3
pid7 Consumer4: try lock
pid7 Consumer4: locked
pid7 Consumer4: product 2 consumed
pid7 Consumer4: unlock
pid7 Consumer4: try consumed 3
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 6
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 6
pid3 Producer2: unlock
pid4 Consumer1: try lock
pid4 Consumer1: locked
pid4 Consumer1: product 3 consumed
pid4 Consumer1: unlock
pid5 Consumer2: try lock
pid5 Consumer2: locked
pid5 Consumer2: product 3 consumed
pid5 Consumer2: unlock
pid2 Producer1: try lock
pid2 Producer1: locked
pid2 Producer1: produce product 7
pid2 Producer1: unlock
pid3 Producer2: try lock
pid3 Producer2: locked
pid3 Producer2: produce product 7
pid3 Producer2: unlock
pid6 Consumer3: try lock
pid6 Consumer3: locked
pid6 Consumer3: product 3 consumed
pid6 Consumer3: unlock
```

```
pid7 Consumer4: try lock
pid7 Consumer4: locked
pid7 Consumer4: product 3 consumed
pid7 Consumer4: unlock
```

总计共`144`条输出信息，符合实验要求中的同步关系，由于我们对生产者启用了一个生产延时，我们可以看出该生产着消费者过程是由**生产者驱动**的，即**消费者不断等待生产者生产出新的产品。**当然，在**实际的应用问题中，也会出现生产者等待消费者消费的情况出现**，这也正是并发程序的多样性所在，我们在此只是模拟一种生产着消费者问题的情况。