```
int i;
for(i=0;i<2;i++){ //创建四个线程
    arg = i;
    stack =(char*)malloc(4096);
    retval = clone(producer,&(stack[4095]),clone_flag,(void*)&arg);
    stack =(char*)malloc(4096);
    retval = clone(consumer,&(stack[4095]),clone_flag,(void*)&arg);
    sleep(30);
}
exit(1);
```

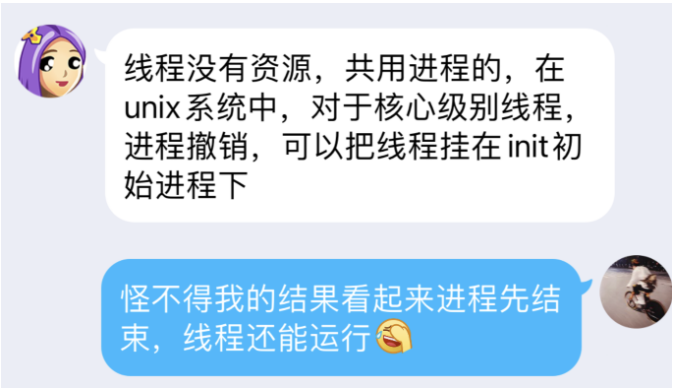
图一

```
Desktop/OS/exp1$ ./exp12.out
Consumer Create
Produer Create
producer0 produce aaa in 0
producer0 produce aaa in 1
producer0 produce aaa in 2
consumer0 get aaa in 2
producer0 produce aaa in 2
producer0 produce aaa in 3
consumer0 get aaa in 3
producer0 produce aaa in 3
consumer0 get aaa in 3
producer0 produce aaa in 3
Consumer Create
Produer Create
producer1 produce bbb in 4
producer1 produce bbb in 5
consumer0 get bbb in 5
producer1 produce bbb in 5
producer0 produce aaa in 6
consumer1 get aaa in 6
consumer0 get bbb in 5
producer1 produce bbb in 5
consumer0 get bbb in 5
producer1 produce bbb in 5
producer0 produce aaa in 6
consumer1 get aaa in 6
consumer0 get bbb in 5
producer1 produce bbb in 5
consumer0 get bbb in 5
consumer0 get bbb in 4
producer0 is over!
consumer0 get aaa in 4
consumer0 is over!
producer1 produce bbb in 3
```

图二

```
}
int producer(void* args){
    printf("Produer Create\n");
    int id = *((int*)args);
    int i;
    for(i=0;i<10;i++){
        sleep(i+1); //表现线程速度差别
        sem_wait(&warehouse);
        pthread_mutex_lock(&mutex);
        if(id==0)
            strcpy(buffer[bp],"aaa\0");
        else
            strcpy(buffer[bp],"bbb\0");
        bp++;
        printf("producer%d produce %s in %d\n"
```

图三




```
int main(int argc,char** argv)
{
    pthread_mutex_init(&mutex,NULL);
    sem_init(&product,0,0);
    sem_init(&warehouse,0,8);
    int clone_flag,arg,retval;
    char *stack;
    clone_flag = CLONE_VM|CLONE_SIGHAND|CLONE_FS|CLONE_FILES;
    int i;
    for(i=0;i<2;i++){ //创建四个线程
        arg = i;
        stack =(char*)malloc(4096);
        retval = clone(producer,&(stack[4095]),clone_flag,(void*)&arg);
        stack =(char*)malloc(4096);
        retval = clone(consumer,&(stack[4095]),clone_flag,(void*)&arg);
        sleep(0.1);
    }
    //printf("OBER!!!!");
    exit(1);
}
```

图一这里的逻辑是，在主进程中先创建两个线程（图二第一个红框中的两个 Create）后进程 sleep 一段时间（比如 30s），然后 cpu 将给到创建出来的线程，在 30s 中两个线程可以运行，由于线程中 sleep 的存在（第三张图）会比较耗时。

在 30s 结束后，CPU 将重新给到主进程，并再次创建两个线程（图二第二个红框），并再次 sleep 30s ，包括上次创建出来还没运行完的两个线程和新创建的两个线程，一共四个线程。
此时 CPU 给到这四个线程上。

在第二次 30s 结束后，主进程结束（exit），所以程序结束，输出停止。发现只有前两个创建出来的线程正常结束了（图二第三个红框），而第二次创建的两个线程并没有结束。

这时我们发现问题的所在是 30s 时间太短了。因此将时间改为 50 或 100 甚至更多时可以看到所有线程正常运行完（即生产和消费各 10 个）并输出 over。

将时间改为 0.1 后的反常行为可以看成一种机制，（或者说一个操作系统的 bug，我自己觉得）这种机制使主进程结束后，线程依旧还在运行。导致所有线程都会运行到正常结束，并各自输出 over。

（我本来想在 exit 前面加一个输出，但是似乎这样做之后会破坏这种机制，结果反而不能得到正常线程的输出了。最后一张图）