

lab4 report

进程通信

任课教师：叶保留 助教：尹熙喆，水兵

学院	计算机科学与技术系	专业	计算机科学与技术
学号	201220154	姓名	王紫萁
Email	201220154@smail.nju.edu.cn	开始/完成时间	2022. 4.28/2022.5.1

1. exercises

• exercise 1

exercise1: 请回答一下，哲学家问题方案1什么情况下会出现死锁。

当每个哲学家都执行 `P(fork[i]);` (同时拿起左手的叉子) 后可以继续往下执行，但是这时没有空闲叉子因此会在 `P(fork[(i+1)%N]);` 全部进入等待。从而每个哲学家都没有机会执行 `eat()` 即后续的语句。

• exercise 2

exercise2: 说一下方案2有什么不足？（答出一点即可）

这种方案执行的效率太低，虽然不会出现死锁，但是某个哲学家拿叉子，吃饭，放叉子的流程不能被打断，其他哲学家必须在思考完看着他做完这一套才能执行自己的吃饭流程，多个哲学家吃饭就在准备吃饭时退化成了单线程，不利于多个哲学家一起吃饭。

• exercise 3

exercise3: 正确且高效的解法有很多，请你利用信号量PV操作设计一种正确且相对高效（比方案2高效）的哲学家吃饭算法。（其实网上一堆答案，主要是让大家多看看不同的实现。）

```
1  #define N 5                // 哲学家个数
2  semaphore fork[5];         // 信号量初值为1
3  semaphore mutex;           // 互斥信号量，初值1
4  void philosopher(int i){   // 哲学家编号：0-4
5      while(TRUE){
6          think();           // 哲学家在思考
7          P(mutex);          // 进入临界区
8          P(fork[i]);         // 去拿左边的叉子
9          P(fork[(i+1)%N]);   // 去拿右边的叉子
10         V(mutex);           // 退出临界区
```

```

11     eat();                // 吃面条
12     v(fork[i]);          // 放下左边的叉子
13     v(fork[(i+1)%N]);    // 放下右边的叉子
14 }
15 }

```

产生死锁的本质原因是叉子处于临界区中，因此拿叉子操作不能被打断。但是放叉子操作是可以被打断的。

或者不用mutex信号量，可以让奇数哲学家先拿左边叉子再拿右边叉子，让偶数哲学家先拿右边叉子再拿左边叉子，只须对两种情况安排P叉子操作的顺序即可，V操作顺序不影响。

• exercise 4

☞ exercise4: 为什么要用两个信号量呢? emptyBuffers和fullBuffer分别有什么直观含义?

仅用一个信号量 (比如 semaphore buffer) 在生产者生产前执行 P 操作, 在消费者消费后执行 v 操作, 会导致消费者有可能从一个空的 buffer 中获取资源, 这是不符合逻辑的, 因为只有当缓冲区有资源时生产者才应该能够执行取资源的操作。因此在消费者消费之前必须再P一个条件信号 fullBuffers, 同样生产者在生产之后也需要V出 fullBuffers

emptyBuffers 和 fullBuffers 分别表示了可用的空缓冲区个数以及已有产品的缓冲区数量。只有存在空闲缓冲区时生产者才能继续执行添加产品, 否则说明所有缓冲区都满需要挂起等待。同样只有存在产品时消费者才从其中取产品否则说明所有缓冲区都空就不取。

2. tasks

• 2.1. 格式化输入

起始就是维护好键盘缓冲区, 和数电实验中的键盘缓冲区维护的方法如出一辙。即键盘负责将字符放到缓冲区尾部, 而系统负责从缓冲区头部取字符直到遇到回车。在每读完一个字符后需要将head向后移动 (循环队列判断队列为空的条件为 head==tail)。需要注意的是, 队列为空并不是终止条件, 只有遇到换行符才会终止这一次的中断服务。当将缓冲区读取完毕 (head == tail) 但是没有遇到换行, 需要将当前进程再次阻塞来得到下一个被键盘写好的缓冲队列。

上述读取的过程是在键盘释放进程后实现的 (即 dev[STDIN].value > 0)。当有进程等待 (value < 0) 返回读取失败 (-1)。否则 (value == 0) 说明键盘空闲, 需要将当前进程加入到键盘的等待队列中, 并将他设置为阻塞状态, 同时键盘的 value 就可以置为-1了, 表示当前进程在键盘上挂起, 当敲击按键的动作发生后键盘就会将该进程释放以便读取刚刚放入缓冲区的字符。

• 2.2.测试信号量

信号量的各个系统调用的实现也都相对简单, 需要注意信号量的操作都是原语操作因此不能被中断。

1. sem_init

目的是在 semaphore 列表的空闲位置注册一个新的信号量, 并进行初始化构造, 构造成功就返回列表索引。构造过程如下, 分为3步: ① 设置初值, 状态置1 ② 将前后指针都指向自己 ③ 返回索引

```

1     sem[i].value = value;
2     sem[i].state = 1;
3     sem[i].pcb.prev = &sem[i].pcb;
4     sem[i].pcb.next = &sem[i].pcb;
5     pcb[current].regs.eax = i;

```

2. sem_destroy

最关键的是将用户指出的索引位置的信号量状态置空。

3. sem_wait

P操作，遵循P操作的实现原理完成即可，当前进程阻塞后需要进入时钟中断调度切换到下一个可运行的进程。

```

1 void syscallSemWait(struct StackFrame* sf) {
2     disableInterrupt();
3     int i = (int)sf->edx;
4     if (i < 0 || i >= MAX_SEM_NUM || sem[i].state == 0) {
5         pcb[current].regs.eax = -1;
6         return;
7     } else {
8         sem[i].value--;
9         if (sem[i].value < 0) {
10            pcb[current].state = STATE_BLOCKED;
11            pcb[current].sleepTime = 0x7fffffff;
12            //insert into list head
13            pcb[current].blocked.next = sem[i].pcb.next;
14            pcb[current].blocked.prev = &(sem[i].pcb);
15            sem[i].pcb.next = &(pcb[current].blocked);
16            (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
17            pcb[current].regs.eax = 0;
18            enableInterrupt();
19            asm("int $0x20");
20        }
21    }
22    pcb[current].regs.eax = 0;
23    enableInterrupt();
24 }

```

4. sem_post

V操作，和P操作实现过程相似，值得一提的是 pcb.prev 指向的是目标pcb的blocked域，因此PCB的首地址应该再减去block的偏移量，即对应代码11行的地方。

```

1 void syscallSemPost(struct StackFrame* sf) {
2     disableInterrupt();
3     int i = (int)sf->edx;
4     if (i < 0 || i >= MAX_SEM_NUM || sem[i].state == 0) {
5         pcb[current].regs.eax = -1;
6         return;
7     } else {
8         sem[i].value++;
9         if (sem[i].value <= 0) {
10            ProcessTable* pt = NULL;

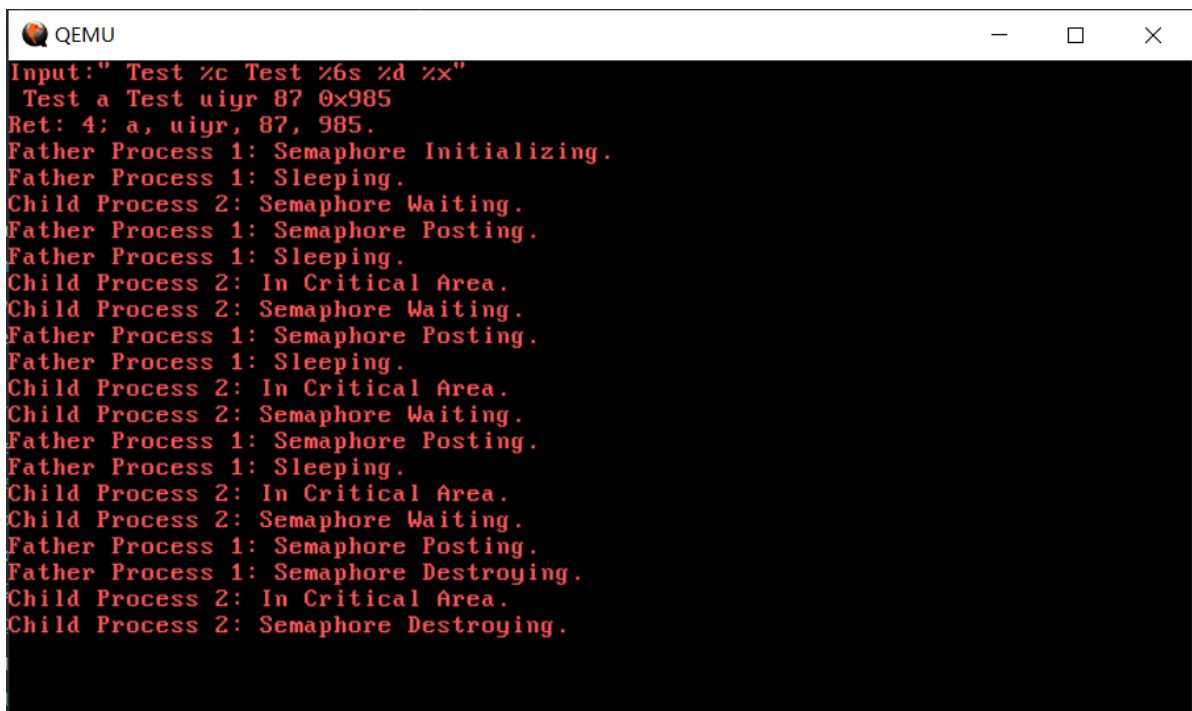
```

```

11         pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) - (uint32_t)
& (((ProcessTable*)0)->blocked));
12         sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
13         (sem[i].pcb.prev)->next = &(sem[i].pcb);
14         pt->state = STATE_RUNNABLE;
15         pcb[current].regs.eax = 0;
16         enableInterrupt();
17         asm volatile("int $0x20");
18     }
19 }
20 pcb[current].regs.eax = 0;
21 enableInterrupt();
22 }

```

测试结果如下:



```

QEMU
Input: "Test %c Test %6s %d %x"
Test a Test uigr 87 0x985
Ret: 4: a, uigr, 87, 985.
Father Process 1: Semaphore Initializing.
Father Process 1: Sleeping.
Child Process 2: Semaphore Waiting.
Father Process 1: Semaphore Posting.
Father Process 1: Sleeping.
Child Process 2: In Critical Area.
Child Process 2: Semaphore Waiting.
Father Process 1: Semaphore Posting.
Father Process 1: Sleeping.
Child Process 2: In Critical Area.
Child Process 2: Semaphore Waiting.
Father Process 1: Semaphore Posting.
Father Process 1: Sleeping.
Child Process 2: In Critical Area.
Child Process 2: Semaphore Waiting.
Father Process 1: Semaphore Posting.
Father Process 1: Semaphore Destroying.
Child Process 2: In Critical Area.
Child Process 2: Semaphore Destroying.

```

简单来说就是父进程负责释放信号量，以便子进程能够使用信号量进入临界区。打印的数字是通过 `get_pid` 系统调用返回的进程号。

• 2.3.3. 同步测试

- 1. 哲学家问题

```
QEMU
God 1: initiate successfully
Philosopher 0: thinking
Philosopher 1: thinking
Philosopher 2: thinking
Philosopher 3: thinking
Philosopher 4: thinking
God 1: leaving
Philosopher 0: pick up left
Philosopher 0: pick up right
Philosopher 1: pick up left
Philosopher 0: eating
Philosopher 0: put down left
Philosopher 0: put down right
Philosopher 1: pick up right
Philosopher 2: pick up left
Philosopher 1: eating
```

```
QEMU
Philosopher 0: pick up right
Philosopher 1: pick up left
Philosopher 0: eating
Philosopher 0: put down left
Philosopher 0: put down right
Philosopher 1: pick up right
Philosopher 2: pick up left
Philosopher 1: eating
Philosopher 1: put down left
Philosopher 1: put down right
Philosopher 2: pick up right
Philosopher 3: pick up left
Philosopher 2: eating
Philosopher 2: put down left
Philosopher 2: put down right
Philosopher 3: pick up right
Philosopher 4: pick up left
Philosopher 3: eating
Philosopher 3: put down left
Philosopher 3: put down right
Philosopher 4: pick up right
Philosopher 4: eating
Philosopher 4: put down left
Philosopher 4: put down right
```

经过比对检查，没有哲学家会重复拿起已经被拿起过的叉子，且每个哲学家都吃过饭。

```
1  if (philo < 5) {
2      printf("Philosopher %d: thinking\n", philo);
3      sleep(128);
4      sem_wait(&mutex);
5      printf("Philosopher %d: pick up left\n", philo);
6      sem_wait(&forks[philo]);
7      printf("Philosopher %d: pick up right\n", philo);
8      sem_wait(&forks[(philo + 1) % 5]);
9      sem_post(&mutex);
10     sleep(128);
11     printf("Philosopher %d: eating\n", philo);
12     sleep(128);
13     printf("Philosopher %d: put down left\n", philo);
14     sem_post(&forks[philo]);
```

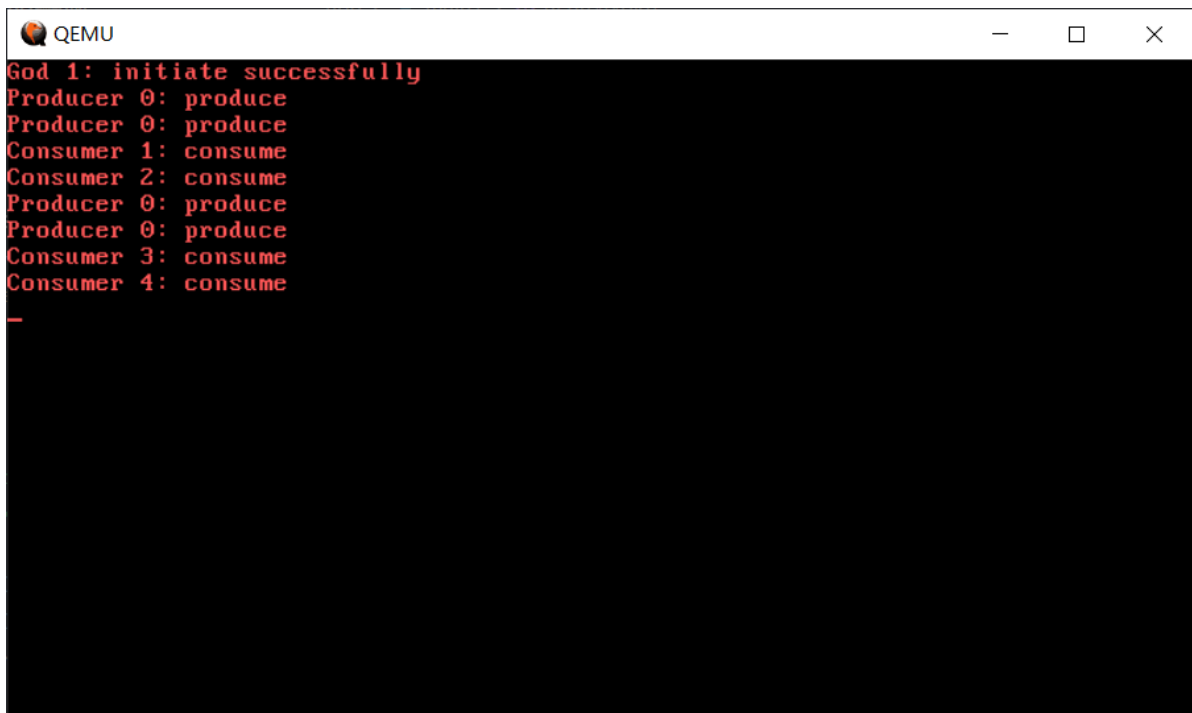
```

15     printf("Philosopher %d: put down right\n", philo);
16     sem_post(&forks[(philo + 1) % 5]);
17
18     printf("Process %d released\n", get_pid());
19     sem_destroy(&forks[philo]);
20     exit();
21
22 } else if (philo == 5) {
23
24     sleep(128);
25     printf("God %d: leaving\n", get_pid());
26 }

```

哲学家吃饭的过程如上，采用了[exercise 3](#)中的实现方案。

- 2. 生产者消费者问题



```

QEMU
God 1: initiate successfully
Producer 0: produce
Producer 0: produce
Consumer 1: consume
Consumer 2: consume
Producer 0: produce
Producer 0: produce
Consumer 3: consume
Consumer 4: consume

```

我们让生产者生产四件商品，并且缓冲区大小为2，只有当有缓冲区不空时消费者才取产品，当存在空缓冲区时生产者生产产品。

```

1  int merchanCnt = 0;
2  if (idx == 0) { //producer
3      while (merchanCnt < 4) { //produce 4 merchandises
4          sem_wait(&emptyBuffers);
5          sem_wait(&mutex);
6          printf("Producer 0: produce\n");
7          merchanCnt++;
8          sleep(128);
9          sem_post(&mutex);
10         sem_post(&fullBuffers);
11     }
12     sleep(256); //avoid advance destroy
13     sem_destroy(&emptyBuffers);
14     sem_destroy(&mutex);
15     sem_destroy(&fullBuffers);

```

```

16         exit();
17     } else if (idx < 5) { //1,2,3,4 consumers;
18         sem_wait(&fullBuffers);
19         sem_wait(&mutex);
20         printf("Consumer %d: consume\n", idx);
21         sleep(128);
22         sem_post(&mutex);
23         sem_post(&emptyBuffers);
24         exit();
25     }

```

3. 读者写者问题

```

QEMU
Writer 0: write
Writer 1: write
Writer 2: write
Reader 3: read
reader Rcount: 1
Reader 4: read
reader Rcount: 2
Reader 5: read
reader Rcount: 3
Reader 3: read
reader Rcount: 3
Reader 4: read
reader Rcount: 3
Reader 5: read
reader Rcount: 3
Reader 3: read
reader Rcount: 3
Reader 4: read
reader Rcount: 3
Reader 5: read
reader Rcount: 3
Writer 0: write
Writer 1: write
Writer 2: write

```

读者写者问题涉及到共享内存变量 (Rcount) 的问题，在此我们提供一个简化的解决方案，即通过系统调用访问处于内核态的共享内存空间的拷贝。通过 `store_global`, `fetch_global` 系统调用分别将更改后的全局变量存储到该空间中，以及获取共享空间中的变量值。

同样为了方便测试，我们会让读写者分别执行两次读写操作。

```

1     int writeTime = 0;
2     int readTime = 0;
3     if (idx < 3) {
4         while (writeTime < 2) {
5             sem_wait(&writeMutex);
6             printf("Writer %d: write\n", idx);
7             writeTime++;
8             sleep(128);
9             sem_post(&writeMutex);
10        }
11        exit();
12    } else if (idx < 6) {
13        while (readTime < 3) {
14            sem_wait(&cntMutex);
15            fetch_global(&Rcount);
16            if (Rcount == 0) sem_wait(&writeMutex);

```

```

17         ++Rcount;
18         store_global(Rcount);
19         sem_post(&cntMutex);
20
21         printf("Reader %d: read\n", idx);
22         readTime++;
23         printf("reader Rcount: %d\n", Rcount);
24         sleep(128);
25
26         sem_wait(&cntMutex);
27         fetch_global(&Rcount);
28         --Rcount;
29         store_global(Rcount);
30         if (Rcount == 0) sem_post(&writeMutex);
31         sem_post(&cntMutex);
32     }
33     exit();
34 }

```

在运行过程中会发现，在写操作时会有明显时间上的阻塞，而读操作几乎同时被打印出来，说明写互斥以及并行读的实现是正确的。并且在读取时写着不会写说明读-写互斥也是正确的。

3. 总结

本次实验了解了进程间的通信，通过信号量机制的条件信号和锁信号使并发进程能按序执行。在实现了信号量后简单复现了三种经典的进程通信的实例，并在最后一个实例中体会了共享内存的一个简单实现方案（受限于信号量无法在进程之间传递变量值的瓶颈）。

同时还理解了不同进程对键盘的抢占。在一段时间只能有一个进程申请使用键盘，并在键盘上等待。当有按键中断到来时该进程会被释放并将输入内容返回到用户程序中