

操作系统lab4实验报告

191220156 计算机科学与技术系 张桓 2659428453@qq.com

操作系统lab4实验报告

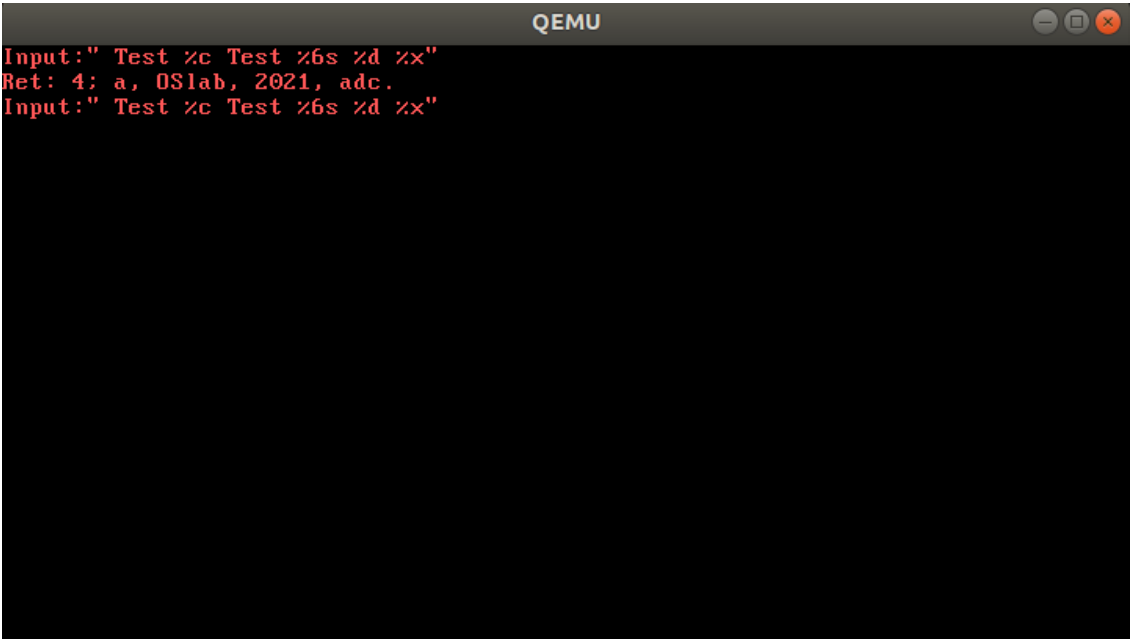
- 实验进度
- 实验结果
- 实验内容
 - 实现格式化输入函数
 - 实现信号量
 - syscallSemInit()
 - syscallSemWait()
 - syscallSemPost()
 - syscallSemDestroy()
 - 解决进程同步问题
- 总结感想

实验进度

实现了格式化输入函数，信号量相关的所有函数，并且用这些函数解决了5个哲学家进餐的问题。

实验结果

- 实现格式化输入函数



- 实现信号量

```
QEMU
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.
```

- 解决进程同步问题，[点击这里有详细的结果解读](#)

```
QEMU
Philosopher 1: think
Philosopher 2: think
Philosopher 0: think
Philosopher 3: think
Philosopher 4: think
Philosopher 0: eat
Philosopher 3: eat
Philosopher 1: eat
Philosopher 4: eat
Philosopher 0: think
Philosopher 2: eat
Philosopher 3: think
Philosopher 0: eat
Philosopher 1: think
Philosopher 3: eat
Philosopher 4: think
Philosopher 1: eat
Philosopher 2: think
Philosopher 4: eat
```

实验内容

实现格式化输入函数

其实也就是完成 `scanf()` 函数，按照之前几个实验的流程，我们需要实现库函数，然后是系统调用，最终完成系统调用服务例程。在框架代码中已经实现好了前两个部分，因此只需要完成 `irqHandle.c` 中的 `syscallReadStdIn()` 函数即可。

- 根据手册得知，当 `dev[STD_IN] == 0` 时表示此时没有输入设备资源，需要**阻塞该进程**，然后**调度其他进程**。

- 本次实验**进程阻塞**的操作**大致分为三步**：设置进程自身 `pcb[current]`；将进程添加到信号量的等待链表 `pcb` 中；用 `int 0x20` 调度其他进程。细节如下：
- 将该进程的状态设置为 `STATE_BLOCKED`，并且 `sleepTime` 设置为 `-1`，分析如下：

由于当前的阻塞不是因为时间片耗尽产生的，根据框架代码可知调度是通过 `int $0X20` 用 `timerHandle()` 函数实现的。因此如果阻塞在输入设备上，**不用再维护该进程的时间片信息**，而是直接调度其他 `STATE_RUNABLE` 的进程，阅读框架代码的实现可知 `sleepTime == -1` 可以实现上述要求。

这部分代码如下：

```
1  pcb[current].state = STATE_BLOCKED;
2  pcb[current].sleepTime = -1;
```

- 将该进程添加到“信号量” `dev[STD_IN]` 的**阻塞列表**上，这是个双向链表，并 `dev[STD_IN].value--`；表示阻塞在输入设备上的进程+1，这部分代码参照手册的示例即可：

```
1  dev[STD_IN].value--;
2  pcb[current].blocked.next = dev[STD_IN].pcb.next;
3  pcb[current].blocked.prev = &(dev[STD_IN].pcb);
4  dev[STD_IN].pcb.next = &(pcb[current].blocked);
5  (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

- 然后应该调度其他进程，用 `asm volatile(int $0X20)` 即可。
- 当**按下下一个按键，键盘设备就绪**，调用 `keyBoardHandle()` 函数，将按键 `keyCode` 放到 `keyBuffer` 缓冲区中。然后再看有无进程需要输入，注意到框架代码有 `dev[STD_IN].value < 0` 的判断，当这个值小于0说明**有进程被阻塞在输入设备上**，我们需要**释放一个进程**，工作如下：
 - 从 `dev[STD_IN]` 的阻塞链表中**取一个进程**，按照手册中给出的示例即可
 - 设置该进程的 `state = STATE_RUNNABLE`，`sleepTime = 0`，表示**已经就绪**

```
1  dev[STD_IN].value ++;
2  pt = (ProcessTable*)((uint32_t)(dev[STD_IN].pcb.prev) -
3      (uint32_t)&(((ProcessTable*)0)->blocked));
4  dev[STD_IN].pcb.prev = (dev[STD_IN].pcb.prev)->prev;
5  (dev[STD_IN].pcb.prev)->next = &(dev[STD_IN].pcb);
6  pt->state = STATE_RUNNABLE;
7  pt->sleepTime = 0;
```

- 进程被唤醒后，读取先前存在 `keyBuffer` 中的所有数据，这时该进程就开始正式进行输入操作了，工作主要有几步：
 - 需要切换到用户进程，然后读取缓冲区每个字符 `character` 到 `str` 指针中，根据手册的示例即可。

- 读取结束有两种情况，一种是整个 `bufferHead == bufferTail` 读完整个缓冲区，一种是读到结束符 `\0`，这时跳出循环后需要显示在 `str` 指针处设置结束符 `\0`。
- 最后需要设置返回值，返回值放在当前进程的 `EAX` 中，应该为读取的字符的个数，即 `i`。关键代码如下：

```

1  int sel = sf->ds;
2  char *str = (char*)sf->edx;
3  int max = sf->ebx; // MAX_BUFFER_SIZE, reverse last byte
4  int i = 0;
5  char character = 0;
6  asm volatile("movw %0, %%es::"m"(sel));
7  for(i = 0; i < max - 1; i++)
8  {
9      if(bufferHead!=bufferTail)
10     {
11         character=getChar(keyBuffer[bufferHead]);
12         bufferHead=(bufferHead+1)%MAX_KEYBUFFER_SIZE;
13         putChar(character);
14         if(character != 0)
15             asm volatile("movb %0, %%es:(%1)":"r"(character),"r"(str+i));
16         else break;
17     }
18     else break;
19 }
20 asm volatile("movb $0x00, %%es:(%0)":"r"(str+i));
21 pcb[current].regs.eax = i;

```

- 按照手册中的说明，阻塞在 `dev[STD_IN]` 上的进程**最多只能有一个**，即这个值最小为 `dev[STD_IN]==-1`，如果当前已经有阻塞的进程了，再来的进程**直接返回 -1**，代码如下：

```

1  else if (dev[STD_IN].value == -1)
2      pcb[current].regs.eax = -1;
3  return ;

```

实现信号量

这一部分实现的是与信号量相关的函数，同样库函数部分和系统调用部分框架代码已经给出，只需要实现4个系统服务例程：`syscallSemInit()`、`syscallSemWait()`、`syscallSemPost()`和`syscallSemDestroy()`即可。

syscallSemInit()

- `sem_init` 系统调用是用于初始化信号量的，其库函数原型为：

```
1 int sem_init(sem_t *sem, uint32_t value) {
2     *sem = syscall(SYS_SEM, SEM_INIT, value, 0, 0, 0);
3     if (*sem != -1) return 0;
4     else return -1;
5 }
```

其中参数 `value` 作为参数传给 `syscall()` 函数，对应寄存器为 `EDX`，用于指定信号量的初始值，初始化成功则返回0，不成功返回-1。显然返回值需要放在 `pcb[current].regs.eax` 中。因此我们的工作是：

- 遍历信号量数组 `sem` 找到一个空闲的信号量，即 `sem[i].state == 0`，找不到设置**返回值**为 -1
- 如果能够找到就**初始化信号量**的一系列值，并且设置**返回值为sem数组内的下标**，根据手册中的示例代码 `initSem()`，代码很容易写出：

```
1 void syscallSemInit(struct StackFrame *sf) {
2     // TODO: complete `SemInit`
3     int i = 0;
4     for(i = 0; i < MAX_SEM_NUM; i++)
5         if(sem[i].state == 0)
6             break;
7     if(i != MAX_SEM_NUM) // success
8     {
9         sem[i].state = 1;
10        sem[i].value = (int32_t)(sf->edx);
11        sem[i].pcb.next = &(sem[i].pcb);
12        sem[i].pcb.prev = &(sem[i].pcb);
13        pcb[current].regs.eax = i;
14    }
15    else //unsuccess
16        pcb[current].regs.eax = -1;
17    return;
18 }
```

syscallSemWait()

`sem_wait` 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。

对应库函数原型为：

```

1  int sem_wait(sem_t *sem) {
2      return syscall(SYS_SEM, SEM_WAIT, *sem, 0, 0, 0);
3  }

```

其中参数 `*sem` 就是需要进行 P 操作的信号量，它实际上是 `sem` 数组的下标，通过查看 `syscall()` 函数可知它放在寄存器 `EDX` 上，而且该函数的返回值放在 `pcb[current].regs.eax` 上。

我们的工作如下：

- 首先取出该信号量对应的下标 `i = (int)sf->edx`，先判断 `i` 范围是否合法，不合法直接返回 `-1`。若合法再判断 `sem[i]` 是否可用，不可用也直接返回 `-1`。
- 再根据 `sem[i].value` 的取值，判断阻塞或不阻塞，两种情况返回值都为 0：
 - 若值 ≥ 1 ，表示有足够资源，进程**不阻塞**，直接 `sem[i].value--;`
 - 若值 ≤ 0 ，表示当前没有资源，进程**阻塞**，同样也需要将 `sem[i].value--;`，并且加入到该信号的阻塞进程链表中，涉及到阻塞的内容之前已经分析过，[在这里](#)，整体代码如下：

```

1  void syscallSemWait(struct StackFrame *sf) {
2      // TODO: complete `SemWait` and note that you need to consider some special
      situations
3      int i = (int)(sf->edx);
4      if(i < 0 || i > MAX_SEM_NUM)//illegal
5      {
6          pcb[current].regs.eax = -1;
7          return;
8      }
9      else if(sem[i].state == 0)
10     {
11         pcb[current].regs.eax = -1;
12         return ;
13     }
14     if(sem[i].value <= 0)//need be blocked
15     {
16         sem[i].value--;
17         pcb[current].state = STATE_BLOCKED;
18         pcb[current].sleepTime = -1;
19         pcb[current].blocked.next = sem[i].pcb.next;
20         pcb[current].blocked.prev = &(sem[i].pcb);
21         sem[i].pcb.next = &(pcb[current].blocked);
22         (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
23         asm volatile("int $0x20");
24         pcb[current].regs.eax = 0; //return value
25     }
26     else //normally
27     {
28         sem[i].value--;
29         pcb[current].regs.eax = 0; //return value
30     }
31     return ;

```

syscallSemPost()

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 STATE_RUNNABLE），若操作成功则返回 0，否则返回 -1。

库函数原型如下：

```
1 int sem_post(sem_t *sem) {
2     return syscall(SYS_SEM, SEM_POST, *sem, 0, 0, 0);
3 }
```

工作和 sem_wait() 很相似，sem_wait() 是申请资源阻塞进程，而 sem_post() 是产生资源释放进程，框架代码中已经给出了对于 i 的合法性判断，接下来就是对 sem[i].value 值的判断了：

- 若值 >= 0，表示当前**没有**进程阻塞在该信号量上，直接 sem[i].value++；即可，返回值为 0
- 若值 < 0，则当前**有**进程阻塞在该信号量上，同样 sem[i].value++；，然后释放一个进程，返回值也为 0。释放进程的工作我们在之前**释放阻塞在键盘设备上的进程**完全一致，详情在[这里](#)，具体工作如下：
 - 从该信号量的阻塞链表中**取下一个进程**，按照手册中给出的示例即可
 - 设置该进程的 state = STATE_RUNNABLE，sleepTime = 0，表示**已经就绪**

```
1 void syscallSemPost(struct StackFrame *sf) {
2     int i = (int)sf->edx;
3     ProcessTable *pt = NULL;
4     if (i < 0 || i >= MAX_SEM_NUM) {
5         pcb[current].regs.eax = -1;
6         return;
7     }
8     // TODO: complete other situations
9     else if(sem[i].state == 0)
10    {
11        pcb[current].regs.eax = -1;
12        return;
13    }
14    if(sem[i].value < 0)//release a process
15    {
16        sem[i].value++;
17        pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
18                               (uint32_t)&((ProcessTable*)0)->blocked));
19        sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
20        (sem[i].pcb.prev)->next = &(sem[i].pcb);
21        pt->state = STATE_RUNNABLE;
22        pt->sleepTime = 0;
23        pcb[current].regs.eax = 0;
```

```

24     }
25     else//no process blocked
26     {
27         sem[i].value++;
28         pcb[current].regs.eax = 0;
29     }
30     return ;
31 }

```

syscallSemDestroy()

`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，可带来未知错误。

- 先判断 `i` 的值是否合理，不合理则返回 `-1`，返回值放在 `pcb[current].regs.eax`，如果当前信号量未被使用，那么就不存在销毁一说，也返回 `-1`。
- 销毁的操作其实就是**初始化该信号量**，按照手册中的参考代码即可，返回值为`0`，整体代码如下：

```

1  void syscallSemDestroy(struct StackFrame *sf) {
2      // TODO: complete `SemDestroy`
3      int i = (int)(sf->edx);
4      if(i < 0 || i > MAX_SEM_NUM)
5      {
6          pcb[current].regs.eax = -1;
7          return ;
8      }
9      else if(sem[i].state == 0)
10     {
11         pcb[current].regs.eax = -1;
12         return ;
13     }
14     sem[i].state = 0; // 0: not in use; 1: in use;
15     sem[i].value = 0; // >=0: no process blocked; -1: 1 process blocked;
16     sem[i].pcb.next = &(sem[i].pcb);
17     sem[i].pcb.prev = &(sem[i].pcb);
18     pcb[current].regs.eax = 0;
19     return;
20 }

```


解决进程同步问题

要解决五个哲学家进餐问题，我的方法是手册推荐的方法，即规定**奇数号**哲学家先拿起其左边的叉子，再拿右边的，**偶数号**哲学家则相反，先拿起其右边的叉子，再拿起左边的，这样总会有某一入进餐，不会产生死锁。

工作如下：

- 显然**叉子就是资源**，对于5个叉子都需要设置其对应的信号量，且因为叉子只有一个，所以**初始值都是1**。
- 接下来为每个哲学家创建进程，用 `fork()` 函数实现，当父进程 `fork()` 出一个子进程后，若为子进程 `ret==0` 则得到一个编号并跳出循环执行 `philosopher` 函数，进行 **4 次这样的循环**，创建了4个哲学家进程，**加上父进程本身**一共5个哲学家，编号依次为 `0~5`，初始化信号量和创建5个哲学家进程代码如下：

```
1  int ret;
2  for(int i = 0; i < 5; i++)
3  {
4      ret = sem_init(&sem[i], 1);
5      if(ret == -1) exit();
6  }
7  int no = 0;
8  for(int i = 1; i < 5; i++)
9  {
10     ret = fork();
11     if(ret == -1) exit();
12     if(ret == 0)//the son process
13     {
14         no = i;//the num of the philisopher
15         break;
16     }
17 }
18 philosopher(no);
19 return 0;
```

- 每个哲学家的工作都是思考后就餐，在就餐之前根据编号的奇偶性，先**P**其左(右)侧的叉子，再**P**其右(左)侧的叉子。如果成功**P**到了两个叉子，那么就开始就餐。就餐完成后**V**释放掉这两个叉子。整体代码如下：

```
1  #define N 5                // 哲学家个数
2  sem_t sem[5];              // 5个叉子的信号量
3  void philosopher(int i){    // 哲学家编号: 0-4
4      while(1){
5          printf("Philosopher %d: think\n", i);          // 哲学家在思考
6          sleep(128);
7          if(i%2==0){
8              sem_wait(&sem[i]);          // 去拿左边的叉子
9              sem_wait(&sem[(i+1)%N]);    // 去拿右边的叉子
10         } else {
```

```

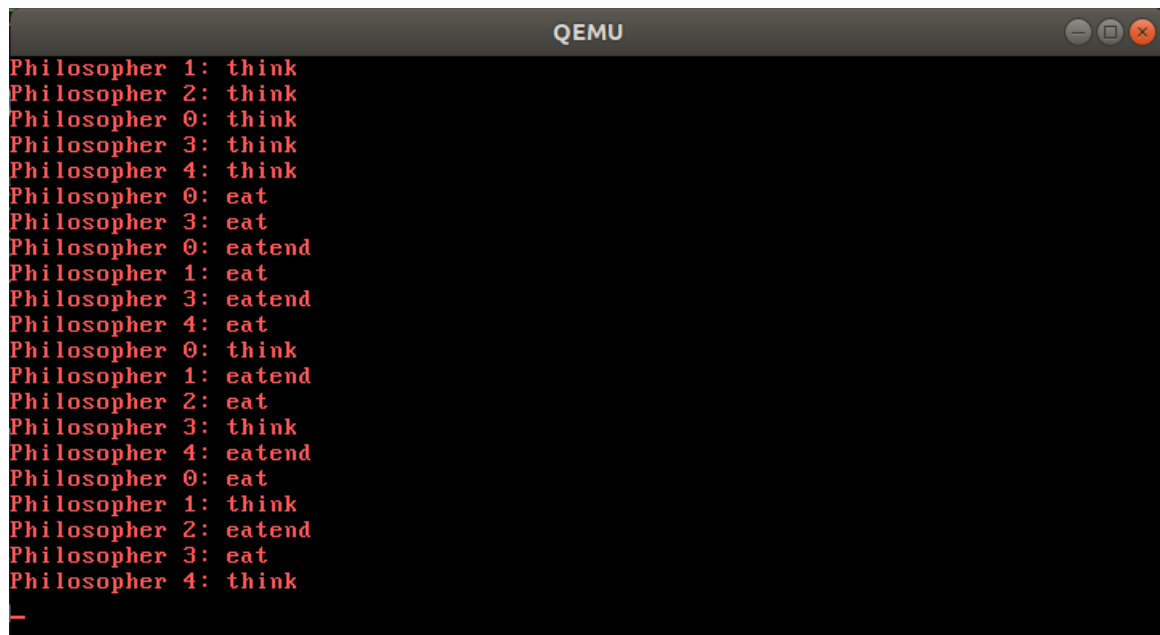
11     sem_wait(&sem[(i+1)%N]);    // 去拿右边的叉子
12     sem_wait(&sem[i]);          // 去拿左边的叉子
13 }
14 printf("Philosopher %d: eat\n", i);          // 吃面条
15 sleep(128);
16 sem_post(&sem[i]);              // 放下左边的叉子
17 sem_post(&sem[(i+1)%N]);        // 放下右边的叉子
18 //printf("Philosopher %d: eatend\n", i);
19 sleep(128);
20 }
21 }

```

在编写代码时发现原本的框架代码中 `MAX_SEM_NUM==4`，导致最多只能分配4个信号量，而且 `MAX_PCB_NUM` 也为4，最多只能创建4个进程，因此需要将这两个值修改大。

为了让演示更清楚，我加了一条输入，当哲学家吃完**放下左右两个叉子之后**输出 `philosopher i eatend` (源码中已注释)，这样可以明显看出每个哲学家的就餐情况：哲学家0拿起了叉子0和1开始吃，哲学家3拿起了叉子3和4开始吃，然后哲学家0放下了叉子0和1，哲学家1拿起了叉子1和2开始吃，哲学家3放下叉子3和4，哲学家4又拿起叉子4和5开始吃，一直持续下去.....

可以看到就餐时叉子**一直没有发生冲突**，函数可以**一直运行下去不死锁**：



```

QEMU
Philosopher 1: think
Philosopher 2: think
Philosopher 0: think
Philosopher 3: think
Philosopher 4: think
Philosopher 0: eat
Philosopher 3: eat
Philosopher 0: eatend
Philosopher 1: eat
Philosopher 3: eatend
Philosopher 4: eat
Philosopher 0: think
Philosopher 1: eatend
Philosopher 2: eat
Philosopher 3: think
Philosopher 4: eatend
Philosopher 0: eat
Philosopher 1: think
Philosopher 2: eatend
Philosopher 3: eat
Philosopher 4: think
_

```

总结感想

这次实验整体来说不难，很多关键的代码手册中均已给出，实现起来还是比较容易的。在实现哲学家问题的时候我一开始没想出来几个哲学家的进程该怎么创建，之前的编程作业里一直都直接调用的 `pthread_create()` 函数，后来查阅资料发现还可以单纯用 `fork()` 函数和 `for` 循环实现，这个技巧挺实用的。

映像比较深的 bug 就是解决哲学家问题时，发现创建信号量时居然会返回 -1，后来顺藤摸瓜才发现原来的框架中 MAX_SEM_NUM 最大为4，然后在创建进程时又发现居然也会创建失败，才发现框架代码中 MAX_PCB_NUM 也是4，于是我把这两个宏都改大了，问题顺利解决。这个 bug 找的还是比较顺利的，因为在编写之初就针对创建失败的情况做出了对应输出，但还是觉得很坑，**建议在教程中增加这部分的改动。**

再有就是几个信号量的相关函数了，信号量的结构体相比进程结构体更简单，比上次实验的代码量更少。此外还**建议是把手册格式改成 PDF 版本**，加上目录跳转翻阅起来更方便一些。

- 参考文章：[信号量解决哲学家问题](#)

最后感谢助教gg/助教jj的耐心批改 😊😊

如有疑问请联系我：2659428453@qq.com