1.实验要求

本实验要求实现操作系统的信号量及对应的系统调用，然后基于信号量解决哲学家就餐问题

1.1. 实现格式化输入函数

在lab2中就要求大家实现了格式化输出函数，但是真正功能完备的格式化输入函数没有叫大家实现，为什么呢?原因之一是基于中断的`scanf`需要进行进程同步，而这个在前面的实验中没有涉及，本实验首先需要大家实现一个`scanf`格式化输入函数并使用以下代码进行测试，为进程同步内容打下基础:

```
#include "lib.h"
#include "types.h"
int uEntry(void) {
   int dec = 0;
   int hex = 0;
   char str[6];
   char cha = 0;
   int ret = 0;
   while(1){
       printf("Input:\" Test %%c Test %%6s %%d %%x\"\n");
       ret = scanf(" Test %c Test %6s %d %x", &cha, str, &dec, &hex);
       printf("Ret: %d; %c, %s, %d, %x.\n", ret, cha, str, dec, hex);
       if (ret == 4)
         break;
   }
     return 0;
}
```

输入`Test a Test oslab 2023 0xabc`后，屏幕上输出为`Ret: 4; a, oslab, 2023, adc.`

要求非格式化字符原样输入，其它异常(输入和要求格式不符等)可自行斟酌决定如何处理

1.2. 实现信号量相关系统调用

实现`SEM_INIT`、`SEM_POST`、`SEM_WAIT`、`SEM_DESTROY`系统调用，使用以下用户程序测试，并在实验报告中说 明实验结果

```
#include "lib.h"
#include "types.h"
int uEntry(void) {
    int i = 4;
    int ret = 0;
    int value = 2;

        sem_t sem;
        printf("Father Process: Semaphore Initializing.\n");
    ret = sem_init(&sem, value);
    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;
}
```

1.3. 基于信号量解决进程同步问题

基于信号量解决哲学家就餐问题，你可以自行在`uEntry()`中实现验证这个问题，所需的功能性函数也可以直接自己定义在文件`lab4/app/main.c`中，最终提交代码的`lab4/app/main.c`中的代码大致如下:

```
#include "lib.h"
#include "types.h"

int uEntry(void) {
    // For lab4.1
    // Test 'scanf'
        ...
    // For lab4.2
    // Test 'Semaphore'
    ...
        // For lab4.3
    // TODO: You need to design and test the philosopher problem.
        // Producer-Consumer problem and Reader& Writer Problem are optional.
    // Note that you can create your own functions.
    // Requirements are demonstrated in the guide.
    ...

        return 0;
}
```

2.相关资料

2.1. 信号量

相信课上大家已经对信号量有了一定的了解，这里以另一个角度介绍一下信号量

信号是一种抽象数据类型，由一个整型(sem)变量和两个原子操作组成;

P() (Prolaag，荷兰语尝试减少)

- sem减1
- 如sem<0，进入等待，否则继续

V() (Verhoog，荷兰语增加)

- sem加1
- 如sem<=0，唤醒一个等待进程

信号量的实现(伪代码):

```
class Semaphore {
    int sem;
    WaitQueue q;
}

Semaphore::P(){
  sem--;
  if(sem < 0){
    Add this thread t to q;
    block(t)
    }
}

Semaphore::V(){
  sem++;
  if(sem <= 0){
    Remove a thread t from q;
    wakeup(t);
    }
}
```

2.2. 信号量的简单应用

2.2.1. 用信号量实现临界区的互斥访问

每类资源设置一个信号量，其初值为1

```
mutex = new Semaphore(1);
mutex->P();
Critical Section;
mutex->V();
```

这里要注意必须成对使用P()操作和V()操作

- P()操作保证互斥访问临界资源
- V()操作在使用后释放临界资源
- PV操作不能次序错误、重复或遗漏

2.2.2. 用信号量实现条件同步

条件同步设置一个信号量，其初值为0

```
condition = new Semaphore(0);
```

```
thread A                                thread B
... M ...
... N ...                               ... X ...

... Y ...
```

A有M和N模块，B有X和Y模块，这里为了保证B执行到X后，A才能执行N，可以使用信号量实现条件同步

```
 thread A                                thread B
... M ...
condition->P();  -----------+
... N ...                   |           ... X ...
                            +---------> condition->V();
                                        ... Y ...
```

2.3. 经典进程同步问题

友情提示:这三个问题是进程同步中非常经典的问题，考察率也非常高，相信课上的时候老师也着重讲过了，
我们的实验中哲学家问题必做，其他两个问题选做。

2.3.1. 哲学家就餐问题

问题描述:

- 5个哲学家围绕一张圆桌而坐
    - 桌子上放着5支叉子
    - 每两个哲学家之间放一支
- 哲学家的动作包括思考和进餐
    - 进餐时需要同时拿到左右两边的叉子
    - 思考时将两支叉子返回原处
- 如何保证哲学家们的动作有序进行?即:不出现有人永远拿不到叉子

方案1:

```
#define N 5                                    // 哲学家个数
semaphore fork[5];                             // 信号量初值为1
void philosopher(int i){                       // 哲学家编号:0-4
  while(TRUE){
    think();                                   // 哲学家在思考
    P(fork[i]);                                // 去拿左边的叉子
        P(fork[(i+1)%N]);                      // 去拿右边的叉子
        eat();                                 // 吃面条
        V(fork[i]);                            // 放下左边的叉子
        V(fork[(i+1)%N]);                      // 放下右边的叉子
    }
}
```

极端情况下不正确，可能导致死锁

方案2:

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

互斥访问正确，但是每次只允许一个人就餐

方案3:

```
#define N 5                                    // 哲学家个数
semaphore fork[5];                             // 信号量初值为1
void philosopher(int i){                       // 哲学家编号:0-4
  while(TRUE){
    think();                                   // 哲学家在思考
    if(i%2==0){
      P(fork[i]);                              // 去拿左边的叉子
      P(fork[(i+1)%N]);                        // 去拿右边的叉子
    } else {
      P(fork[(i+1)%N]);                        // 去拿右边的叉子
      P(fork[i]);                              // 去拿左边的叉子
    }
    eat();                                     // 吃面条
    V(fork[i]);                                // 放下左边的叉子
    V(fork[(i+1)%N]);                          // 放下右边的叉子
    }
}
```

没有死锁，可以实现多人同时就餐

> 有没有更好的方式处理这个就餐问题?

2.3.2. 生产者-消费者问题

生产者---->缓冲区---->消费者

有界缓冲区的生产者-消费者问题描述:

- 一个或多个生产者在生产数据后放在一个缓冲区里
- 单个消费者从缓冲区取出数据处理
- 任何时刻只能有一个生产者或消费者可访问缓冲区

问题分析:

- 任何时刻只能有一个线程操作缓冲区(互斥访问)
- 缓冲区空时，消费者必须等待生产者(条件同步)
- 缓冲区满时，生产者必须等待消费者(条件同步)

用信号量描述每个约束:

- 二进制信号量mutex
- 资源信号量fullBuffers
- 资源信号量emptyBuffers

伪代码描述一下:

```
class BoundedBuffer {
    mutex = new Semaphore(1);
    fullBuffers = new Semaphore(0);
    emptyBuffers = new Semaphore(n);
}
```

```
BoundedBuffer::Deposit(c){
  emptyBuffers->P();
  mutex->P();
  Add c to the buffer;
  mutex->V();
  fullBuffers->V();
}

BoundedBuffer::Remove(c){
  fullBuffers->P();
  mutex->P();
  Remove c from buffer;
  mutex->V();
  emptyBuffers->V();
}
```

> P、V的操作顺序有影响吗?

2.3.3. 读者-写者问题

读者-写者问题主要出现在数据库等共享资源的访问当中，问题描述:

- 共享数据的两类使用者
    - 读者:只读取数据，不修改
    - 写者:读取和修改数据
- 对共享数据的读写

- “读-读”允许，同一时刻，允许有多个读者同时读
- “读-写”互斥，没有写者时读者才能读，没有读者时写者才能写
- “写-写”互斥，没有其他写者，写者才能写

用信号量描述每个约束:

- 信号量WriteMutex，控制读写操作的互斥，初始化为1
- 读者计数Rcount，正在进行读操作的读者数目，初始化为0
- 信号量CountMutex，控制对读者计数的互斥修改，初始化为1，只允许一个线程修改Rcount计数

写者进程

```
P(WriteMutex);
write;
V(WriteMutex);
```

读者进程

```
P(CountMutex);
if (Rcount == 0)
  P(WriteMutex);
++Rcount;
V(CountMutex);
read;
P(CountMutex);
--Rcount;
if (Rcount == 0)
  V(WriteMutex);
V(CountMutex);
```

2.4. 相关系统调用

sem_init

`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);
```

sem_post

`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`

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

sem_wait

`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`

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

sem_destroy

`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

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

3.实验攻略

在攻略之前，先带大家看一看实验4新增的或修改的数据结构等:

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

struct Device {
    int state;
    int value;
    struct ListHead pcb; // link to all pcb ListHead blocked on this device
};
typedef struct Device Device;

Semaphore sem[MAX_SEM_NUM];
Device dev[MAX_DEV_NUM];
```

信号量`Semaphore`相信看完**相关资料**，大家都能理解，因为我们将信号量的定义成数组，所以添加了一个`state`成员，表示当前信号量是不是正在使用，1表示正在使用，0表示未使用

那么Device是干啥用的，为什么和信号量的定义这么相似，说这个之前我们需要想一下stdin标准输入，在操作 系统中，我们不可能通过一直监听键盘中断来进行输入，这样太浪费系统资源了，所以我们需要一个键盘输入 缓冲区和类似信号量的东西来实现条件同步，在键盘中断将输入存入缓冲区后再让用户程序读取，所以代码中 定义了Device，他其实就是信号量，只不过不能由用户通过系统调用控制，而是直接和硬件绑定

在实验中，我们将stdin，stdout都抽象成了Device，其中

```
#define STD_OUT 0
#define STD_IN 1
```

实际上，stdout是非阻塞式的，stdin上才会有进程阻塞.

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

ListHead是一个双向链表

如下两个函数用于初始化sem和dev

```
void initSem() {
```

```
    int i;
        for (i = 0; i < MAX_SEM_NUM; i++) {
                sem[i].state = 0; // 0: not in use; 1: in use;
        sem[i].value = 0; // >=0: no process blocked; -1: 1 process blocked; -2:
2 process blocked;...
        sem[i].pcb.next = &(sem[i].pcb);
        sem[i].pcb.prev = &(sem[i].pcb);
    }
}

void initDev() {
    int i;
    for (i = 0; i < MAX_DEV_NUM; i++) {
        dev[i].state = 1; // 0: not in use; 1: in use;
        dev[i].value = 0; // >=0: no blocked; -1: 1 process blocked; -2: 2
process blocked;...
        dev[i].pcb.next = &(dev[i].pcb);
        dev[i].pcb.prev = &(dev[i].pcb);
    }
}
```

修改的数据结构:PCB中添加对应的双向链表结构

```
 struct ProcessTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct TrapFrame regs;
    uint32_t stackTop;
    uint32_t prevStackTop;
    int state;
    int timeCount;
    int sleepTime;
    uint32_t pid;
    char name[32];
+   struct ListHead blocked; // sempahore, device, file blocked on
 };
 typedef struct ProcessTable ProcessTable;
```

这样将current线程加到信号量i的阻塞列表可以通过以下代码实现

```
    pcb[current].blocked.next = sem[i].pcb.next;
    pcb[current].blocked.prev = &(sem[i].pcb);
    sem[i].pcb.next = &(pcb[current].blocked);
    (pcb[current].blocked.next)->prev = &(pcb[current].blocked);
```

以下代码可以从信号量i上阻塞的进程列表取出一个进程:

```
    pt = (ProcessTable*)((uint32_t)(sem[i].pcb.prev) -
            (uint32_t)&(((ProcessTable*)0)->blocked));
  sem[i].pcb.prev = (sem[i].pcb.prev)->prev;
    (sem[i].pcb.prev)->next = &(sem[i].pcb);
```

irqHandle.c中的syscallWrite也有一些**变化**:

```
void syscallWrite(struct TrapFrame *sf) {
   switch(sf->ecx) { // file descriptor
-           case 0:
-                   syscallPrint(sf);
+           case STD_OUT:
+                   if (dev[STD_OUT].state == 1) {
+                       syscallWriteStdOut(sf);
+                   }
               break; // for STD_OUT
            default:break;
        }
}
```

另外对syscallPrint进行了重命名

```
-void syscallPrint(struct TrapFrame *sf) {
+void syscallWriteStdOut(struct TrapFrame *sf) {
```

3.1. 实现格式化输入函数

为了降低实验难度，`syscall.c`中的`scanf`已经完成，同学们只需要完成对应的中断处理例程`irqHandle.c`中添加了`syscallRead`函数处理各种读数据:

```
void syscallRead(struct StackFrame *sf) {
    switch(sf->ecx) {
        case STD_IN:
            if (dev[STD_IN].state == 1)
                syscallReadStdIn(sf);
            break; // for STD_IN
        default:
            break;
    }
}
```

在这一节主要关注的就是`syscallReadStdIn`，同学们需要去完成它，那么如何完成呢，它是和键盘中断有条 件同步的，所以的这一步还要结合`keyboardHandle`一起完成

在实验2中，有很多同学不知道下面的代码是干啥的

```
extern uint32_t keyBuffer[MAX_KEYBUFFER_SIZE];
extern int bufferHead;
extern int bufferTail;
```

其实这就是键盘输入的缓冲区，把所有零碎的知识拼凑在一起，`keyboardHandle`要做的事情就两件:

1. 将读取到的`keyCode`放入到`keyBuffer`中
2. 唤醒阻塞在`dev[STD_IN]`上的一个进程

接下来安排`syscallReadStdIn`，它要做的事情也就两件:

1. 如果`dev[STD_IN].value == 0`，将当前进程阻塞在`dev[STD_IN]`上
2. 进程被唤醒，读`keyBuffer`中的所有数据

值得注意的就是最多只能有一个进程被阻塞在`dev[STD_IN]`上，多个进程想读，那么后来的进程会返回`-1`，其 他情况`scanf`的返回值应该是实际读取的字节数

和实验2中printf的处理例程类似，以下代码可以将读取的字符`character`传到用户进程

```
int sel = sf->ds;
char *str = (char *)sf->edx;
int i = 0;
asm volatile("movw %0, %%es"::"m"(sel));
asm volatile("movb %0, %%es:(%1)"::"r"(character),"r"(str + i));
```

完成这一步后请测试`scanf`，并在实验报告展示结果

3.2. 实现信号量

这一部分也只需要完善处理例程，其它部分已经实现，所有的信号量相关调用有一个总的处理:

```
void syscallSem(struct StackFrame *sf) {
    switch(sf->ecx) {
        case SEM_INIT:
            syscallSemInit(sf);
            break;
        case SEM_WAIT:
            syscallSemWait(sf);
            break;
        case SEM_POST:
            syscallSemPost(sf);
            break;
        case SEM_DESTROY:
            syscallSemDestroy(sf);
            break;
        default:break;
    }
}
```

需要完成的是4个子例程:`syscallSemInit`、`syscallSemWait`、`syscallSemPost`和`syscallSemDestroy`

在实现时，因为信号量以数组形式存在，所以只要一个**下标**就可以定位信号量

完成后请进行测试，并在实验报告中展示结果

3.3. 解决进程同步问题

为了方便区分进程，你可以实现`getpid`系统调用，用来返回当前进程的`pid`

3.3.1. 哲学家就餐问题

同学们需要在`lab4/app/main.c`中实现哲学家就餐问题

要求:

- 5个哲学家同时运行
- 哲学家思考，`printf("Philosopher %d: think\n", id);`
- 哲学家就餐，`printf("Philosopher %d: eat\n", id);`
- 任意P、V及思考、就餐动作之间添加`sleep(128);`

3.3.2. 生产者-消费者问题和读者-写者问题(选做)

如果你有多余的精力和兴趣，你可以选择额外完成其它两个进程同步问题。实现方式不限，问题要求如下:

生产者-消费者问题:

- 4个生产者，1个消费者同时运行
- 生产者生产，`printf("Producer %d: produce\n", id);`
- 者消费，`printf("Consumer : consume\n");`
- 任意P、V及生产、消费动作之间添加`sleep(128);`

读者-写者问题:

- 3个读者，3个写者同时运行
- 读者读数据，`printf("Reader %d: read, total %d reader\n", id, Rcount);`
- 写者写数据，`printf("Writer %d: write\n", id);`
- 任意P、V及读、写动作之间添加`sleep(128);`

4.作业提交

本次作业需提交可通过编译的实验相关源码与报告，提交前请确认 make clean 过。

截止时间:2023-5-29 23:55:00