南京航空航天大学

操作系统实践

Proj3

班级： 1618001
学号： 161840230
姓名： 王可

一、 题目简述

1. 信号量(50%)

- 借助spinlock，为xv6添加信号量的支持。

  定义一个信号量结构体`struct semaphore`，成员自定，然后在内核里开辟一个包含100个信号量的空间，假如是s[100]，提供以下**系统调用**给用户程序：

  ```
  int alloc_sem (int v);
  ```

  创建一个初始值为v的信号量，返回信号量的下标。若返回值为-1，表明分配失败。分配失败的原因可能是初始值为负数或者系统中信号量资源不足。

  ```
  int wait_sem(int i);
  ```

  对信号量s[i]执行wait操作；成功返回1，出错返回-1。出错的原因可能是该信号量不存在，如i不在0~99之间，或者信号量尚未分配。

  ```
  int signal_sem(int i);
  ```

  对信号量s[i]执行signal操作；成功返回1，出错返回-1。出错的原因同上。

  ```
  int dealloc_sem(int i);
  ```

  删除信号量s[i]，将s[i]标记为未分配。同时将所有等待s[i]的进程终止（终止进程时参考kill的实现）。成功返回1，失败返回-1。失败的情况包括下标不在合法范围内以及信号量未被分配。

---

2. 消息传递(50%)

xv6没有实现消息传递机制，你需要添加这个支持，实现以下系统调用。

```
int msg_send (int pid, int a, int b, int c);
```

将a,b,c三个整数发送给编号为pid的进程，直到接收者进程执行完receive才返回。若接收者尚未执行receive，则进入阻塞状态。成功返回1，失败返回-1。失败的情况包括系统资源不足或者不存在编号为pid的进程。

```
int msg_receive(int *a, int *b, int *c);
```

接收一个消息(三个整数)，返回发送者的进程编号。如果未接收到任何消息，则陷入阻塞状态。

你可以用前面实现的信号量，但请注意:

- 信号量应当用时分配，不可以提前分配
- 当进程消亡时，它所占用的信号量资源应当释放
- 先执行receive，则receive会阻塞，直到有进程send
- 先执行send，则send会阻塞，直到有进程receive

二、 实验过程

1. 信号量

1.1 全局考虑

1. 信号量的实现较为简单，观察测试程序可以得出，信号量的测试主要是考察如下两点
    - 对于错误情况的处理
    - 阻塞与唤醒的控制

    因此，事实上信号量实验的核心就是实现题给四个函数调用，完成介绍中描述的功能即可。
2. 在本次实验中，需要设计一个数据结构用于表示信号量并且完成一定功能。包括分配，使用和删除。
3. 为了使得删除时杀死所有相关进程，那么需要存储进程相关内容，考虑的方法有数组和链表两种，出于习惯，选择了链表的方式（虽然这给后面的工作带来了很多很多麻烦）。

1.2 数据结构

建立的数据结构主要考虑：需要记录信号量值，需要对信号量加锁，需要记录信号量当前状态，同时使用链表结构记录相关进程，并用一个整型数记录相关进程数量（等待阻塞状态的进程数量）。对于链表中的进程，使用另一个数据结构proc_list,存储是否被使用，进程PCB，下一个结点。

```
//semaphore
struct semaphore{
  struct spinlock lock;
  int value;
  int used;
  struct proc_list *procList;
  int length;
};

struct proc_list{
  int id;
  int used;
  struct proc *proc;
  struct proc_list *next;
};
```

1.3 分配信号量

1.3.1 主要思路

对于信号量的分配主要关注两点

1. 逻辑的实现：

    首先，规定了SEM_SIZE,信号量的最大数量，控制最多能访问的信号量数量，同时检测信号量初始值不能为负。当找到可用的信号量时，将其标记为不可用并进行初始化。
2. 资源的保护

    由于信号量表是所有进程共享的资源，容易存在并发问题，因此需要对其加以保护。使用spinlock对其进行加锁，在更改完表项后解锁。

1.3.2 实现代码

```
int
alloc_sem(int v)
{
  if(v < 0) return -1;
  int index;
  acquire(&semaphores_lock);
  for(index = 0; index < SEM_SIZE; index++){
    if(semaphores[index].used == 0){
      semaphores[index].used = 1;
      semaphores[index].value = v;
      semaphores[index].procList = NULL;
      semaphores[index].length = 0;
      initlock(&semaphores[index].lock,"sem");
      release(&semaphores_lock);
      return index;
    }
  }
  release(&semaphores_lock);
  return -1;
}
```

1.4 wait与signal

1.4.1 主要思路

wait和signal是信号量处理的主要机制，在实现上也有对应的关系：

wait时，首先将信号量值减一，若不足0，则进入阻塞，将自己加入到信号量的等待列表中，并增加链表长度，过程中需要对信号量加锁。

而signal时，先将信号量加一，若非正，则说明有任务在等待，所以将等待列表中的第一项释放，唤醒，减少链表长度后退出。

1.4.2 实现代码

```
int
wait_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  semaphores[i].value--;
  if(semaphores[i].value < 0){
    struct proc_list *p = semaphores[i].procList;
    if(semaphores[i].length == 0) {
      semaphores[i].procList= PLalloc();
      p = semaphores[i].procList;
    }else{
      while(p->next != NULL) p = p->next;
      p->next = PLalloc();
      p = p->next;
    }
    if(p == NULL) return -1;
    semaphores[i].length += 1;
    p->proc = myproc();
```

```
    p->next = NULL;
    sleep(myproc(), &semaphores[i].lock);
  }

  release(&semaphores[i].lock);
  return 1;
}

int
signal_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  semaphores[i].value++;
  if(semaphores[i].value <= 0){
    wakeup(semaphores[i].procList->proc);
    struct proc_list *p = semaphores[i].procList;
    semaphores[i].procList = p->next;
    PLfree(p);
    semaphores[i].length -- ;
  }
  release(&semaphores[i].lock);
  return 1;
}
```

1.5 回收信号量

1.5.1 主要思路

回收信号量，实际上就是将信号量标为未使用状态，但因为还需要同时杀死所有相关进程，所以要遍历信号量的进程链表，找到所有相关进程，用其进程号将其杀死，并回收链表节点资源。

1.5.2 实现代码

```
int
dealloc_sem(int i)
{
  if(i < 0 || i >= SEM_SIZE) return -1;
  if(semaphores[i].used == 0) return -1;
  acquire(&semaphores[i].lock);
  if(semaphores[i].used == 1){
    semaphores[i].used = 0;
    while(semaphores[i].procList != NULL)
    {
      kill(semaphores[i].procList->proc->pid);
      struct proc_list *p = semaphores[i].procList;
      semaphores[i].procList = p->next;
      PLfree(p);
    }
  }
  release(&semaphores[i].lock);
  return 1;
}
```

1.6 问题简述

关于你实现的wait_sem，如果多个进程同时执行wait_sem，但wait的具体信号量不同，它们能并

发执行吗？（你应该用一个spinlock去保护一个信号量，另外用一个spinlock保护资源分配）

- 是可以并发执行的。在分配信号量时，由于需要遍历整个表，我使用了semaphores_lock来进行加锁保护，而分配结束后就进行了释放。对于单个信号量的锁，只保护自身。所以多个信号量之间并不涉及到竞争，是可以并发执行的。

2. 消息传递

2.1 全局考虑

消息的传递分为发送和接受，由于是在不同进程内，实现消息传递的同步需要信号量协同完成。当发送方发送但对方未接收时，或者接收放意图接收但无进程发送消息时，都将进入阻塞状态等待同步。这里面同样关键的一点在于，设计数据结构使得每一次交互得以被记录。我采用双端记录的方式，每个进程只考虑与自己相关的内容，不需要对所有信息进行遍历，一定程度上提高了效率。

2.2 数据结构

设计如下数据结构：

在每一个记录的MSGstate结构中包含许多字段， 其中pid表示当前处理的进程号，记录下进程号是因为未来表中是以PCB为单位进行记录而不是进程号，因此若进程出现切换，可能导致同一个PCB下其实是不同的进程，如果保留原来的接收块则容易出错。used字段标识资源是否被使用，positive为主动标识，表示该块记录的是pid字段中进程主动发出的行为还是其他进程发出的行为。target为目标，记录发送方或者接收方的PCB id或者pid根据使用需求填写。sid为当前通讯所使用的信号量id。最后还记录了数据和下一条指针。

链表表头以PCB为单位，因为进程号是向上无穷增长的，而xv6中进程PCB是有限的，因此，只需要创建NPROC数量的表块即可，分为发送和接收分别进行记录。事实上，经过分析可以发现，每一个进程至多发送一个消息，即只需要一个发送块，因此事实上发送表并不需要链表的结构，其实数组就可以实现了，并且这些发送都是主动行为，其positive字段均应该为1。

```
// Massage trans
struct MSGstate{
  int id;
  int pid;
  int used;
  int positive;
  int target;
  int sid; //semaphore id
  int data[3];
  struct MSGstate *next;
};


struct MSGstate * Msend[NPROC];
struct MSGstate * Mrecv[NPROC];
```

2.3 发送消息

2.3.1 主要思路

在进程发送消息时，首先要考虑对方是否已经在等待接收消息，如果已经在等待，则直接将数据填入结构中并通过信号量通知对方接收即可，若对方无请求，则需要在己方和对方处均进行记录（将自己的发送块设置为使用并将对方信息填入块中，同时为对方建立一个被动接收块），代码如下：

2.3.2 实现代码

```
int
msg_send(int pid, int a, int b, int c){
  int found = getProcNum(pid);
  int semaphore;
  if(found < 0)
    return -1;

  acquire(&Msg_lock);
  int now_pid = myproc()->pid;
  int myNum = getProcNum(now_pid);
  struct MSGstate *p = Mrecv[found];

  for(; p != NULL ; p = p->next){  // if the target is waiting for receiving

    if(p->positive == 1 && p->pid == pid){

      p->data[0] = a;
      p->data[1] = b;
      p->data[2] = c;
      p->target = now_pid;   //set the target to be the process
      signal_sem(p->sid);
      release(&Msg_lock);
      return 1;
    }
  }
  struct MSGstate *myp = Msend[myNum];
  p = Mrecv[found];
  semaphore = alloc_sem(0);
  if(semaphore == -1) return -1;

  if(Mrecv[found] == NULL){   // set a nagetive receive for the target
    Mrecv[found] = MSalloc();
    p = Mrecv[found];
  }else{

    for(; p->next !=NULL; p = p->next);
    p->next = MSalloc();
    p = p->next;
  }
  if(p == NULL) return -1;
  p->pid = pid;
  p->positive = 0;
  p->data[0] = a;
  p->data[1] = b;
  p->data[2] = c;
  p->target = myNum;
  p->sid = semaphore;
```

```
  p->next = NULL;

  Msend[myNum] =  MSalloc();
  myp = Msend[myNum];
  if(myp == NULL) return -1;
  myp->pid = now_pid;  // set a positive send for the process itself
  myp->positive = 1;
  myp->data[0] = a;
  myp->data[1] = b;
  myp->data[2] = c;
  myp->target = found;
  myp->sid = semaphore;
  release(&Msg_lock);

  wait_sem(semaphore);

  acquire(&Msg_lock);
  MSfree(myp);
  Msend[myNum] = NULL;
  dealloc_sem(semaphore);
  release(&Msg_lock);

  return 1;
}
```

2.3 接收消息

2.3.1 主要思路

接收消息时的思路与发送较为类似，首先考虑是否有进程已经向自己发送消息，即检查自己的MRecv表中是否有被动消息，若检查有，则将其中数据和target取出，获得数据并返回对方pid。若没有，则给自己的MRecv表中添加一个被动接收块，以供其他进程查找，自身进入阻塞等待消息。

2.3.2 实现代码

```
int
msg_receive(int* a, int* b, int* c){
  int i;
  int target;
  int now_pid= myproc()->pid;
  int flag = 0;
  int semaphore;
  int myNum = getProcNum(now_pid);
  struct MSGstate * pre_p = Mrecv[myNum];
  struct MSGstate * p = Mrecv[myNum];

  acquire(&Msg_lock);

  for(i = 0; p != NULL; p = p->next ,flag = 1, i++){  // if other process has sent some msg
      if(flag) pre_p = pre_p->next;
      if(p->pid == now_pid && p->positive == 0){
        *a = p->data[0];
        *b = p->data[1];
        *c = p->data[2];
        // cprintf("now recieve %d \n",*a);
        target = p->target;
```

```
        semaphore = p->sid;
        if(i == 0) Mrecv[myNum] = p->next;
        else{
          pre_p->next = p->next;
        }
        MSfree(p);
        signal_sem(semaphore);
        release(&Msg_lock);
        return target;
      }
  }

  for(i = 0; p != NULL; p = p->next ,flag = 1, i++){  // if other process has sent some msg
      if(flag) pre_p = pre_p->next;
      if(p->pid != now_pid){
        if(i == 0) Mrecv[myNum] = p->next;
        else{
          pre_p->next = p->next;
        }
        MSfree(p);
      }
  }
  semaphore = alloc_sem(0);
  if(semaphore == -1){
    release(&Msg_lock);
    return -1;
  }
  if(Mrecv[myNum]==NULL){
    Mrecv[myNum] = MSalloc();
    // cprintf("\n----------------want things pid: %d myNUM: %d\n\n",now_pid, myNum);
    p = Mrecv[myNum];
  }else{
    pre_p->next = MSalloc();
    p = pre_p->next;
  }
  if(p == NULL) {
    cprintf("allocate wrong\n");
    return -1;
  }

  p->pid = now_pid;
  p->positive = 1;
  p->sid = semaphore;
  p->target = myNum;
  p->next = NULL;

  // cprintf("in %d recv is %d   \n",now_pid,Mrecv[myNum]->pid);
  release(&Msg_lock);
  // cprintf("%d  in waiting\n",p->pid);
  wait_sem(semaphore);
  acquire(&Msg_lock);
  // cprintf("%d  finish waiting\n",p->pid);
  *a = p->data[0];
  *b = p->data[1];
  *c = p->data[2];
  target = p->target;
```

```
  if(i == 0) Mrecv[myNum] = p->next;
  else{
       pre_p->next = p->next;
      }
  MSfree(p);
  dealloc_sem(semaphore);
  release(&Msg_lock);
  return target;
}
```

2.4 辅助函数

在如下函数中，进行了锁和表的初始化，并且实现一个函数，通过pid值获得该进程存储于PCB中的偏移量，以此确定存储位置。

```
// initialize the spinlocks and the tables
void initMsg(){
  initlock(&Msg_lock,"msg_lock");
  initlock(&semaphores_lock,"sem");
  initlock(&PLlock,"proclist_lock");
  for(int i = 0 ; i < 1000; i++){
    PL[i].used = 0;
    PL[i].id = i;
    MS[i].id = i;
    MS[i].used = 0;
  }
  initlock(&MSlock,"massage_lock");
  for(int i = 0; i < NPROC; i++){
      Msend[i] = NULL;
      Mrecv[i] = NULL;
  }
  return;
}

// get the location in ptable by pid
int
getProcNum(int pidd)
{
  struct proc *p;

  acquire(&ptable.lock);
  int num = 0;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++,num++){
    if(p->pid == pidd){
      release(&ptable.lock);
      return num;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

2.4 问题简述

如果receive先执行，但send未执行，你是如何让receive阻塞的（或者说阻塞到哪个信号量上了）？如果send先执行，但receive未执行，你是如何让send阻塞的?

- receive先执行的情况中，接收方进程在自己的MRecv表中建立一个positive字段值为1的接收块，并将自己分配到的信号量id（初始值为0）存入块中，接收方便对此信号量进行wait操作。发送方检查到接收方有主动接收块，则将数据存入其中，并取出semaphore id字段，对其执行signal来唤醒接收方进程。
- 反之类似，send先执行的情况中，发送方将给接收方在MRecv字段中创建一个被动接收块，并将分配得到的Sid存入，进行wait，接收方在receive时先检测到自身MRecv中有被动接受块，则从其中取出数据与Sid, 执行signal操作唤醒发送方。

3. 其他设计

在整个流程中，因为初期想要使用的是链表，因此第一时间想到动态内存分配，可以节省大量空间。

代码完成后，编译意识到在proc中不能使用malloc，我进一步尝试重新编写malloc，换引用关系，甚至添加系统调用，都无法实现动态的内存分配，出于时间原因，我模仿alloc_proc的做法，先将所有资源分配完成，建立1000个可供分配的资源块，在程序申请时进行分配。以此实现了伪分配的效果，代码如下:

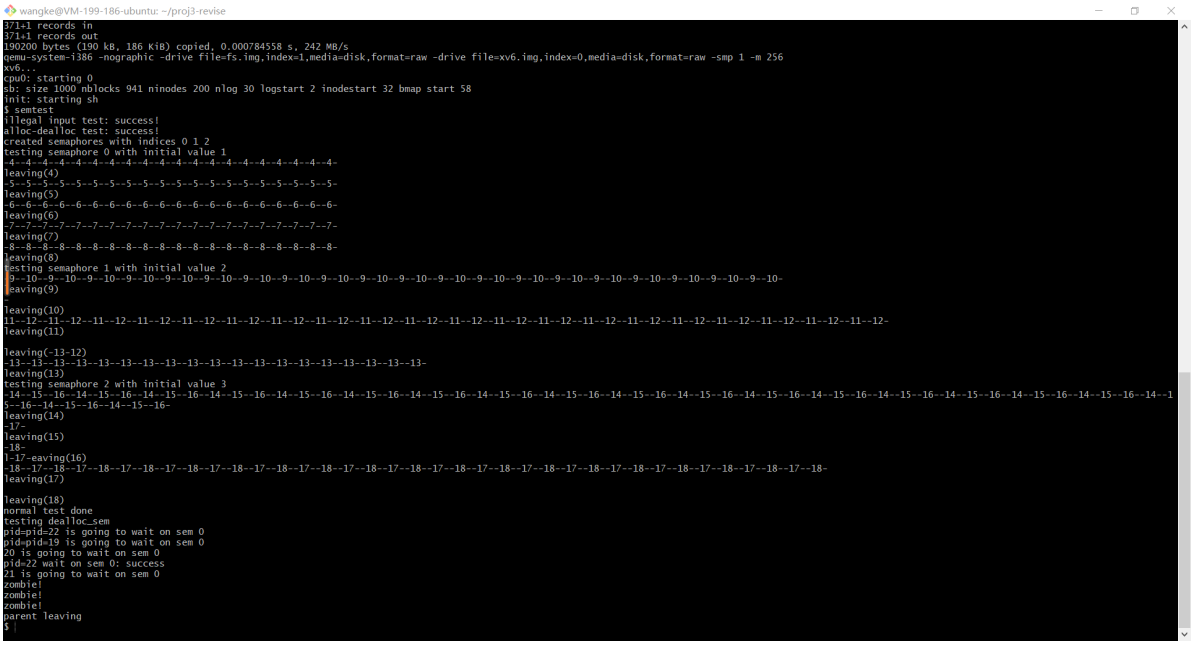
```
struct proc_list PL[1000];
struct MSGstate MS[1000];

struct proc_list *
PLalloc(){
  acquire(&PLlock);
  int i;
  for(i = 0 ; i < 1000; i++){
    if(PL[i].used == 0){
      goto found;
    }
  }
  release(&PLlock);
  return NULL;

found:
  PL[i].used = 1;
  release(&PLlock);
  return &PL[i];
}

struct MSGstate *
MSalloc(){
  acquire(&MSlock);
  int i;
  for(i = 0 ; i < 1000; i++){
    if(MS[i].used == 0){
      goto found;
    }
  }
  release(&MSlock);
  return NULL;

found:
```

```
  MS[i].used = 1;
  release(&MSlock);
  return &MS[i];
}

int
PLfree(struct proc_list *pl){
  acquire(&PLlock);
  pl->used = 0;
  release(&PLlock);
  return 1;
}

int
MSfree(struct MSGstate *ms){
  acquire(&MSlock);
  ms->used = 0;
  release(&MSlock);
  return 1;
}
```

三、 实验结果

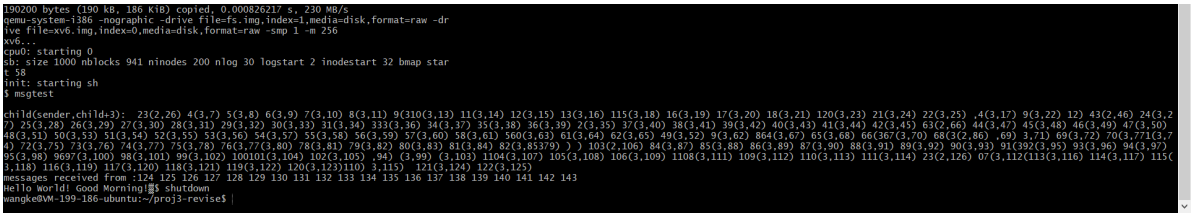
1. 信号量

在xv6中输入semtest，按要求输出了指定内容：



2. 消息传递

在xv6中输入msgtest，输出以下信息：



四、实验心得

本次实验中，我在动态分配内存上花的时间比较多，虽然最后没有能真正实现内存的分配，但是通过模仿alloc_Proc, 也完成了目标需求，在指针操作的编写和调试时，出现了一些问题，比如指针指向方向并没有连接到链表中，出现了trap等等。指针的错也只能回去一行行看代码，好在进行实验前，我已经对于整体框架的设计比较细节，虽然还有一些没有考虑到的点，但在出错后能很快通过输出标识定位，并解决问题。