

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：

```
1 | 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。

```
1 | 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。

```
1 | 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。

```
1 | 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。

```
1 | 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。

```
1 | 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。

```
1 //semaphore
2 struct semaphore{
3     struct spinlock lock;
4     int value;
5     int used;
6     struct proc_list *procList;
7     int length;
8 };
9
10 struct proc_list{
11     int id;
12     int used;
13     struct proc *proc;
14     struct proc_list *next;
15 };
```

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

```
1  int
2  alloc_sem(int v)
3  {
4      if(v < 0) return -1;
5      int index;
6      acquire(&semaphores_lock);
7      for(index = 0; index < SEM_SIZE; index++){
8          if(semaphores[index].used == 0){
9              semaphores[index].used = 1;
10             semaphores[index].value = v;
11             semaphores[index].procList = NULL;
12             semaphores[index].length = 0;
13             initlock(&semaphores[index].lock, "sem");
14             release(&semaphores_lock);
15             return index;
16         }
17     }
18     release(&semaphores_lock);
19     return -1;
20 }
```

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

```
1  int
2  wait_sem(int i)
3  {
4      if(i < 0 || i >= SEM_SIZE) return -1;
5      if(semaphores[i].used == 0) return -1;
6      acquire(&semaphores[i].lock);
7      semaphores[i].value--;
8      if(semaphores[i].value < 0){
9          struct proc_list *p = semaphores[i].procList;
10         if(semaphores[i].length == 0) {
11             semaphores[i].procList = PLalloc();
12             p = semaphores[i].procList;
13         }else{
14             while(p->next != NULL) p = p->next;
15             p->next = PLalloc();
16             p = p->next;
17         }
18         if(p == NULL) return -1;
19         semaphores[i].length += 1;
20         p->proc = myproc();
```

```

21     p->next = NULL;
22     sleep(myproc(), &semaphores[i].lock);
23 }
24
25 release(&semaphores[i].lock);
26 return 1;
27 }
28
29 int
30 signal_sem(int i)
31 {
32     if(i < 0 || i >= SEM_SIZE) return -1;
33     if(semaphores[i].used == 0) return -1;
34     acquire(&semaphores[i].lock);
35     semaphores[i].value++;
36     if(semaphores[i].value <= 0){
37         wakeup(semaphores[i].procList->proc);
38         struct proc_list *p = semaphores[i].procList;
39         semaphores[i].procList = p->next;
40         PLfree(p);
41         semaphores[i].length -- ;
42     }
43     release(&semaphores[i].lock);
44     return 1;
45 }

```

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

```

1  int
2  dealloc_sem(int i)
3  {
4      if(i < 0 || i >= SEM_SIZE) return -1;
5      if(semaphores[i].used == 0) return -1;
6      acquire(&semaphores[i].lock);
7      if(semaphores[i].used == 1){
8          semaphores[i].used = 0;
9          while(semaphores[i].procList != NULL)
10             {
11                 kill(semaphores[i].procList->proc->pid);
12                 struct proc_list *p = semaphores[i].procList;
13                 semaphores[i].procList = p->next;
14                 PLfree(p);
15             }
16     }
17     release(&semaphores[i].lock);
18     return 1;
19 }

```

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。

```
1 // Message trans
2 struct MSGstate{
3     int id;
4     int pid;
5     int used;
6     int positive;
7     int target;
8     int sid; //semaphore id
9     int data[3];
10    struct MSGstate *next;
11 };
12
13
14 struct MSGstate * Msend[NPROC];
15 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，则需要为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

```
1  int
2  msg_send(int pid, int a, int b, int c){
3      int found = getProcNum(pid);
4      int semaphore;
5      if(found < 0)
6          return -1;
7
8      acquire(&Msg_lock);
9      int now_pid = myproc()->pid;
10     int myNum = getProcNum(now_pid);
11     struct MSGstate *p = Mrecv[found];
12
13     for(; p != NULL ; p = p->next){ // if the target is waiting for receiving
14
15         if(p->positive == 1 && p->pid == pid){
16
17             p->data[0] = a;
18             p->data[1] = b;
19             p->data[2] = c;
20             p->target = now_pid; //set the target to be the process
21             signal_sem(p->sid);
22             release(&Msg_lock);
23             return 1;
24         }
25     }
26     struct MSGstate *myp = Msend[myNum];
27     p = Mrecv[found];
28     semaphore = alloc_sem(0);
29     if(semaphore == -1) return -1;
30
31     if(Mrecv[found] == NULL){ // set a negative receive for the target
32         Mrecv[found] = Malloc();
33         p = Mrecv[found];
34     }else{
35
36         for(; p->next !=NULL; p = p->next);
37         p->next = Malloc();
38         p = p->next;
39     }
40     if(p == NULL) return -1;
41     p->pid = pid;
42     p->positive = 0;
43     p->data[0] = a;
44     p->data[1] = b;
45     p->data[2] = c;
46     p->target = myNum;
47     p->sid = semaphore;
```

```

48     p->next = NULL;
49
50     Msend[myNum] = MAlloc();
51     myp = Msend[myNum];
52     if(myp == NULL) return -1;
53     myp->pid = now_pid; // set a positive send for the process itself
54     myp->positive = 1;
55     myp->data[0] = a;
56     myp->data[1] = b;
57     myp->data[2] = c;
58     myp->target = found;
59     myp->sid = semaphore;
60     release(&Msg_lock);
61
62     wait_sem(semaphore);
63
64     acquire(&Msg_lock);
65     Mfree(myp);
66     Msend[myNum] = NULL;
67     dealloc_sem(semaphore);
68     release(&Msg_lock);
69
70     return 1;
71 }

```

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

```

1  int
2  msg_receive(int* a, int* b, int* c){
3      int i;
4      int target;
5      int now_pid= myproc()->pid;
6      int flag = 0;
7      int semaphore;
8      int myNum = getProcNum(now_pid);
9      struct MSGstate * pre_p = Mrecv[myNum];
10     struct MSGstate * p = Mrecv[myNum];
11
12     acquire(&Msg_lock);
13
14     for(i = 0; p != NULL; p = p->next ,flag = 1, i++){ // if other process
has sent some msg
15         if(flag) pre_p = pre_p->next;
16         if(p->pid == now_pid && p->positive == 0){
17             *a = p->data[0];
18             *b = p->data[1];
19             *c = p->data[2];
20             // cprintf("now recieve %d \n",*a);
21             target = p->target;

```



```

22     semaphore = p->sid;
23     if(i == 0) Mrecv[myNum] = p->next;
24     else{
25         pre_p->next = p->next;
26     }
27     MSfree(p);
28     signal_sem(semaphore);
29     release(&Msg_lock);
30     return target;
31 }
32 }
33
34 for(i = 0; p != NULL; p = p->next ,flag = 1, i++){ // if other process
has sent some msg
35     if(flag) pre_p = pre_p->next;
36     if(p->pid != now_pid){
37         if(i == 0) Mrecv[myNum] = p->next;
38         else{
39             pre_p->next = p->next;
40         }
41         MSfree(p);
42     }
43 }
44 semaphore = alloc_sem(0);
45 if(semaphore == -1){
46     release(&Msg_lock);
47     return -1;
48 }
49 if(Mrecv[myNum]==NULL){
50     Mrecv[myNum] = MSalloc();
51     // cprintf("\n-----want things pid: %d myNUM:
%d\n\n",now_pid, myNum);
52     p = Mrecv[myNum];
53 }else{
54     pre_p->next = MSalloc();
55     p = pre_p->next;
56 }
57 if(p == NULL) {
58     cprintf("allocate wrong\n");
59     return -1;
60 }
61
62 p->pid = now_pid;
63 p->positive = 1;
64 p->sid = semaphore;
65 p->target = myNum;
66 p->next = NULL;
67
68 // cprintf("in %d recv is %d \n",now_pid,Mrecv[myNum]->pid);
69 release(&Msg_lock);
70 // cprintf("%d in waiting\n",p->pid);
71 wait_sem(semaphore);
72 acquire(&Msg_lock);
73 // cprintf("%d finish waiting\n",p->pid);
74 *a = p->data[0];
75 *b = p->data[1];
76 *c = p->data[2];
77 target = p->target;

```

```

78     if(i == 0) Mrecv[myNum] = p->next;
79     else{
80         pre_p->next = p->next;
81     }
82     MSfree(p);
83     dealloc_sem(semaphore);
84     release(&Msg_lock);
85     return target;
86 }

```

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。

```

1  // initialize the spinlocks and the tables
2  void initMsg(){
3      initlock(&Msg_lock,"msg_lock");
4      initlock(&semaphores_lock,"sem");
5      initlock(&PLlock,"proclist_lock");
6      for(int i = 0 ; i < 1000; i++){
7          PL[i].used = 0;
8          PL[i].id = i;
9          MS[i].id = i;
10         MS[i].used = 0;
11     }
12     initlock(&MSlock,"message_lock");
13     for(int i = 0; i < NPROC; i++){
14         Msend[i] = NULL;
15         Mrecv[i] = NULL;
16     }
17     return;
18 }
19
20 // get the location in ptable by pid
21 int
22 getProcNum(int pidd)
23 {
24     struct proc *p;
25
26     acquire(&ptable.lock);
27     int num = 0;
28     for(p = ptable.proc; p < &ptable.proc[NPROC]; p++,num++){
29         if(p->pid == pidd){
30             release(&ptable.lock);
31             return num;
32         }
33     }
34     release(&ptable.lock);
35     return -1;
36 }

```

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：

```
1  struct proc_list PL[1000];
2  struct MSGstate MS[1000];
3
4  struct proc_list *
5  PLalloc(){
6      acquire(&PLlock);
7      int i;
8      for(i = 0 ; i < 1000; i++){
9          if(PL[i].used == 0){
10             goto found;
11         }
12     }
13     release(&PLlock);
14     return NULL;
15
16 found:
17     PL[i].used = 1;
18     release(&PLlock);
19     return &PL[i];
20 }
21
22 struct MSGstate *
23 MSalloc(){
24     acquire(&MSlock);
25     int i;
26     for(i = 0 ; i < 1000; i++){
27         if(MS[i].used == 0){
28             goto found;
29         }
30     }
31     release(&MSlock);
32     return NULL;
33
34 found:
```

```

35     MS[i].used = 1;
36     release(&MSlock);
37     return &MS[i];
38 }
39
40 int
41 PLfree(struct proc_list *pl){
42     acquire(&PLlock);
43     pl->used = 0;
44     release(&PLlock);
45     return 1;
46 }
47
48 int
49 MSfree(struct MSGstate *ms){
50     acquire(&MSlock);
51     ms->used = 0;
52     release(&MSlock);
53     return 1;
54 }

```

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：

[illegible]

2. 消息传递

在xv6中输入msgtest，输出以下信息：

```
190200 bytes (190 kB, 188 KiB) copied, 0.000826217 s, 230 MB/s
new-system-1386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -dr
ive file=xv6.img,index=0,media=disk,format=raw -smp 1 -m 256
xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap star
t 58
init: starting sh
$ msgtest
child(sender,child=3): 23(2,26) 4(3,7) 5(3,8) 6(3,9) 7(3,10) 8(3,11) 9(310(3,13) 11(3,14) 12(3,15) 13(3,16) 115(3,18) 16(3,19) 17(3,20) 18(3,21) 120(3,23) 21(3,24) 22(3,25) ,4(3,17) 9(3,22) 12) 43(2,46) 24(3,2
7) 25(3,28) 26(3,29) 27(3,30) 28(3,31) 29(3,32) 30(3,33) 31(3,34) 33(3,36) 34(3,37) 35(3,38) 36(3,39) 2(3,35) 37(3,40) 38(3,41) 39(3,42) 40(3,43) 41(3,44) 42(3,45) 63(2,66) 44(3,47) 45(3,48) 46(3,49) 47(3,50)
48(3,51) 50(3,53) 51(3,54) 52(3,55) 53(3,56) 54(3,57) 55(3,58) 56(3,59) 57(3,60) 58(3,61) 560(3,63) 61(3,64) 62(3,65) 49(3,32) 9(3,62) 864(3,67) 63(3,68) 66(367(3,70) 68(3,72,86) ,69) 3,71) 69(3,72) 70(3,771(3,7
8) 72(3,75) 73(3,76) 74(3,77) 75(3,78) 76(3,77(3,80) 78(3,81) 79(3,82) 80(3,83) 81(3,84) 82(3,85379) ) ) 103(2,106) 84(3,87) 85(3,88) 86(3,89) 87(3,90) 88(3,91) 89(3,92) 90(3,93) 91(392(3,95) 93(3,96) 94(3,97)
95(3,98) 9697(3,100) 98(3,101) 99(3,102) 100101(3,104) 102(3,105) ,94) (3,99) (3,103) 1104(3,107) 105(3,108) 106(3,109) 1108(3,111) 109(3,112) 110(3,113) 111(3,114) 23(2,126) 07(3,112(113(3,116) 114(3,117) 115(
3,118) 116(3,119) 117(3,120) 118(3,121) 119(3,122) 120(3,123)110) ,1,115) ,121(3,124) 127(2,125)
messages received from :124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143
Hello world! Good Morning!$ shutdown
hangsdBW-159-186-ubuntu:~/proj3-revises$
```

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。

