**西北工业大学 操作系统实验 实验报告**

班号： 10012001 姓名： 张俊宁 学号： 2020302766

实验日期： 2022.11.20 实验名称： 线程同步与互斥

**一、实验目的**

掌握GeekOS系统的线程同步与互斥原理，实现线程的同步与互斥。

**二、实验要求**

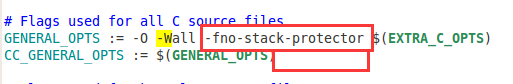
1. 在GeekOS中实现信号量，使用信号量保证用户程序运行时的线程同步，给出关键函数的代码以及实验结果。

2. 设计测试程序，验证线程同步和互斥的结果。

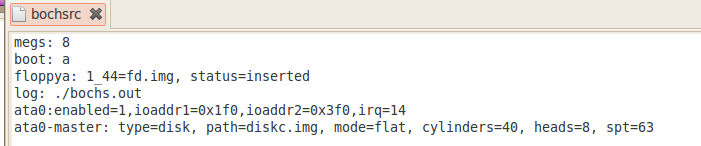
3. 回答课后思考题。第十一章第1题。

**三、实验过程及结果**

修改makefile文件：



拷贝project3中的bochsrc：



1. 本实验是在project2的基础上，也就是创建用户内核线程的基础上，因此需要将project2的修改复制到project3中

首先修改src/geekos/elf.c

1. **int** Parse\_ELF\_Executable(**char** \*exeFileData, ulong\_t exeFileLength,
2. **struct** Exe\_Format \*exeFormat)
3. {
4. //TODO("Parse an ELF executable image");
5. //利用ELF头部结构体指向可执行文件头部，便于获取相关信息
6. elfHeader \*ehdr = (elfHeader\*)exeFileData;
7. //段的个数
8. exeFormat->numSegments = ehdr->phnum;
9. //代码入口地址
10. exeFormat->entryAddr = ehdr->entry;
11. //获取头部表在文件中的位置，便于读取信息
12. programHeader \*phdr = (programHeader\*)(exeFileData + ehdr->phoff);
13. //填充Exe\_Segment
14. unsigned **int** i;
15. **for**(i = 0; i < exeFormat->numSegments; i++, phdr++)
16. {
17. **struct** Exe\_Segment \*segment = &exeFormat->segmentList[i];
18. //获取该段在文件中的偏移量\*
19. segment->offsetInFile = phdr->offset;
20. //获取该段的数据在文件中的长度
21. segment->lengthInFile = phdr->fileSize;
22. //获取该段在用户内存中的起始地址
23. segment->startAddress = phdr->vaddr;
24. //获取该段在内存中的大小
25. segment->sizeInMemory = phdr->memSize;
26. //获取该段的保护标志位
27. segment->protFlags = phdr->flags;
28. }
29. **return** 0;
31. }

之后修改/src/geekos/user.c

1. **int** Spawn(**const** **char** \*program, **const** **char** \*command, **struct** Kernel\_Thread \*\*pThread)
2. {
3. //TODO("Spawn a process by reading an executable from a filesystem");
4. **int** res;
5. //读取 ELF 文件
6. **char** \*exeFileData = NULL;
7. ulong\_t exeFileLength = 0;
8. res = Read\_Fully(program, (**void**\*\*)&exeFileData, &exeFileLength);
9. **if** (res != 0){
10. **if** (exeFileData != NULL) Free(exeFileData);
11. **return** ENOTFOUND;
12. }
14. //分析 ELF 文件
15. **struct** Exe\_Format exeFormat;
16. res = Parse\_ELF\_Executable(exeFileData, exeFileLength, &exeFormat);
17. **if** (res != 0){
18. **if** (exeFileData != NULL) Free(exeFileData);
19. **return** res;
20. }
22. //加载用户程序
23. **struct** User\_Context \*userContext = NULL;
24. res = Load\_User\_Program(exeFileData, exeFileLength, &exeFormat, command, &userContext);
25. **if** (res != 0){
26. **if** (exeFileData != NULL) Free(exeFileData);
27. **if** (userContext != NULL) Destroy\_User\_Context(userContext);
28. **return** res;
29. }
30. **if** (exeFileData != NULL){
31. Free(exeFileData);
32. }
33. exeFileData = NULL;
35. //开始用户进程
36. **struct** Kernel\_Thread \***thread** = NULL;
37. **thread** = Start\_User\_Thread(userContext, **false**);
38. **if** (**thread** == NULL) {//超出内存 创建新进程失败
39. **if** (userContext != NULL){
40. Destroy\_User\_Context(userContext);
41. }
42. **return** ENOMEM;
43. }
45. KASSERT(**thread**->refCount == 2);
46. \*pThread = **thread**;//返回核心进程的指针
47. **return** 0;
48. }
50. **void** Switch\_To\_User\_Context(**struct** Kernel\_Thread\* kthread, **struct** Interrupt\_State\* state)
51. {
52. **static** **struct** User\_Context\* s\_currentUserContext;//之前最近使用过的 userContxt
53. **struct** User\_Context\* userContext = kthread->userContext;//指向User\_Conetxt的指针，并初始化为准备切换的进程
54. KASSERT(!Interrupts\_Enabled());
55. **if** (userContext == 0) **return**;//userContext为0表示此进程为核心态进程就不用切换地址空间
56. **if** (userContext != s\_currentUserContext){
57. Switch\_To\_Address\_Space(userContext);//为用户态进程时则切换地址空间
58. ulong\_t esp0 = ((ulong\_t)kthread->stackPage) + PAGE\_SIZE;//新进程的核心栈指针
59. Set\_Kernel\_Stack\_Pointer(esp0);//设置内核堆栈指针
60. s\_currentUserContext = userContext;//保存新的 userContxt
61. }
62. }

之后就是修改/src/geekos/usereg.c

1. #include <geekos/ktypes.h>
2. #include <geekos/kassert.h>
3. #include <geekos/defs.h>
4. #include <geekos/mem.h>
5. #include <geekos/string.h>
6. #include <geekos/malloc.h>
7. #include <geekos/int.h>
8. #include <geekos/gdt.h>
9. #include <geekos/segment.h>
10. #include <geekos/tss.h>
11. #include <geekos/kthread.h>
12. #include <geekos/argblock.h>
13. #include <geekos/user.h>
14. #define DEFAULT\_USER\_STACK\_SIZE 8192
15. **static** **struct** User\_Context\* Create\_User\_Context(ulong\_t size)
16. {
17. **struct** User\_Context \*userContext;
18. size = Round\_Up\_To\_Page(size);
19. userContext = (**struct** User\_Context \*)Malloc(**sizeof**(**struct** User\_Context));//内存分配成功则继续为 userContext 下的 memory 分配内存空间
20. **if** (userContext == NULL){
21. **return** NULL;
22. }
23. userContext->memory = (**char** \*)Malloc(size);
24. **if** (userContext->memory == NULL){
25. Free(userContext);
26. **return** NULL;
27. }
28. memset(userContext->memory, '\0', size);
29. userContext->size = size;
30. userContext->ldtDescriptor = Allocate\_Segment\_Descriptor();//新建一个 LDT 描述符
31. **if** (userContext->ldtDescriptor == NULL){
32. Free(userContext->memory);
33. **return** NULL;
34. }
35. Init\_LDT\_Descriptor(userContext->ldtDescriptor, userContext->ldt, NUM\_USER\_LDT\_ENTRIES);//初始化段描述符
36. userContext->ldtSelector = Selector(KERNEL\_PRIVILEGE, **true**, Get\_Descriptor\_Index(userContext->ldtDescriptor));//新建一个 LDT 选择子
37. Init\_Code\_Segment\_Descriptor(&userContext->ldt[0], (ulong\_t)userContext->memory, size / PAGE\_SIZE, USER\_PRIVILEGE);//新建一个代码段描述符
38. Init\_Data\_Segment\_Descriptor(&userContext->ldt[1], (ulong\_t)userContext->memory, size / PAGE\_SIZE, USER\_PRIVILEGE);//新建一个数据段描述符
39. userContext->csSelector = Selector(USER\_PRIVILEGE, **false**, 0);//新建数据段选择子
40. userContext->dsSelector = Selector(USER\_PRIVILEGE, **false**, 1);//新建代码段选择子
41. userContext->refCount = 0;//将引用数清零
42. **return** userContext;
43. }

46. **static** **bool** Validate\_User\_Memory(**struct** User\_Context\* userContext,
47. ulong\_t userAddr, ulong\_t bufSize)
48. {
49. ulong\_t avail;
51. **if** (userAddr >= userContext->size)
52. **return** **false**;
54. avail = userContext->size - userAddr;
55. **if** (bufSize > avail)
56. **return** **false**;
58. **return** **true**;
59. }
60. **void** Destroy\_User\_Context(**struct** User\_Context\* userContext)
61. {
62. //TODO("Destroy a User\_Context");
63. KASSERT(userContext->refCount == 0);
64. Free\_Segment\_Descriptor(userContext->ldtDescriptor);//释放 LDT descriptor
65. Disable\_Interrupts();//释放内存空间
66. Free(userContext->memory);
67. Free(userContext);
68. Enable\_Interrupts();
69. }
70. **int** Load\_User\_Program(**char** \*exeFileData, ulong\_t exeFileLength,**struct** Exe\_Format \*exeFormat, **const** **char** \*command,**struct** User\_Context \*\*pUserContext)
71. {
72. //TODO("Load a user executable into a user memory space using segmentation");
73. unsigned **int** i;
74. **struct** User\_Context \*userContext = NULL;
75. ulong\_t maxva = 0;//要分配的最大内存空间
76. **for** (i = 0; i < exeFormat->numSegments; i++){//计算用户态进程所需的最大内存空间
77. **struct** Exe\_Segment \*segment = &exeFormat->segmentList[i];
78. ulong\_t topva = segment->startAddress + segment->sizeInMemory;
79. **if** (topva > maxva) maxva = topva;
80. }
81. unsigned **int** numArgs;//程序参数数目
82. ulong\_t argBlockSize;//获取参数块的大小
83. Get\_Argument\_Block\_Size(command, &numArgs, &argBlockSize);
84. ulong\_t size = Round\_Up\_To\_Page(maxva) + DEFAULT\_USER\_STACK\_SIZE;//用户进程大小 = 参数块总大小 + 进程堆栈大小(8192)
85. ulong\_t argBlockAddr = size;//参数块地址
86. size += argBlockSize;
87. userContext = Create\_User\_Context(size);//按相应大小创建一个进程
88. **if** (userContext == NULL){//如果进程创建失败则返回错误信息
89. **return** -1;
90. }
91. **for** (i = 0; i < exeFormat->numSegments; i++)     {//将用户程序中的各段内容复制到分配的用户内存空间
92. **struct** Exe\_Segment \*segment = &exeFormat->segmentList[i];
93. memcpy(userContext->memory + segment->startAddress,exeFileData + segment->offsetInFile,segment->lengthInFile);
94. }
95. Format\_Argument\_Block(userContext->memory + argBlockAddr, numArgs, argBlockAddr, command);//格式化参数块
96. userContext->entryAddr = exeFormat->entryAddr;//初始化数据段
97. userContext->argBlockAddr = argBlockAddr;//初始化堆栈段
98. userContext->stackPointerAddr = argBlockAddr;//初始化代码段信息
99. \*pUserContext = userContext;//将初始化完毕的 User\_Context 赋给\*pUserContext
100. **return** 0;
101. }
102. **bool** Copy\_From\_User(**void**\* destInKernel, ulong\_t srcInUser, ulong\_t bufSize)
103. {
104. //TODO("Copy memory from user buffer to kernel buffer");
105. **struct** User\_Context\* userContext = g\_currentThread->userContext;
106. **if** (!Validate\_User\_Memory(userContext, srcInUser, bufSize)){//如果访问的用户内存空间非法(越界访问)，则直接返回失败
107. **return** **false**;
108. }
109. memcpy(destInKernel, userContext->memory + srcInUser, bufSize);//拷贝当前用户内存空间数据到系统内核空间
110. **return** **true**;//拷贝成功则返回true
111. }
112. **bool** Copy\_To\_User(ulong\_t destInUser, **void**\* srcInKernel, ulong\_t bufSize)
113. {
114. //TODO("Copy memory from kernel buffer to user buffer");
115. **struct** User\_Context\* userContext = g\_currentThread->userContext;//如果需要拷贝的内容超出用户内存空间(越界)，则直接返回失败
116. **if** (!Validate\_User\_Memory(userContext, destInUser, bufSize)){
117. **return** **false**;
118. }
119. memcpy(userContext->memory + destInUser, srcInKernel, bufSize);//拷贝当前系统内核空间数据到用户内存空间
120. **return** **true**;//拷贝成功则返回true
121. }
122. **void** Switch\_To\_Address\_Space(**struct** User\_Context \*userContext)
123. {
124. //TODO("Switch to user address space using segmentation/LDT");
125. ushort\_t ldtSelector = userContext->ldtSelector;//切换到新的局部描述符表(LDT)
126. \_\_asm\_\_ \_\_volatile\_\_ (
127. "lldt %0"
128. :
129. : "a" (ldtSelector)
130. );
131. }

之后修改src/geekos/kthread.c

在该文件中需要修改调度算法：

1. **struct** Kernel\_Thread \*Get\_Next\_Runnable(**void**)
2. {
3. //Print\_Queues();
4. KASSERT(g\_curSchedulingPolicy == ROUND\_ROBIN ||g\_curSchedulingPolicy == MULTILEVEL\_FEEDBACK);
5. **struct** Kernel\_Thread \*best = NULL;//查找下一个被调度的线程
6. **if** (g\_curSchedulingPolicy == ROUND\_ROBIN){//轮询调度策略：只需要从 Q0 队列找优先级最高的线程取出
7. best = Find\_Best(&s\_runQueue[0]);//如果找到了符合条件的线程则将其从队列中移出
8. **if** (best != NULL){
9. Remove\_Thread(&s\_runQueue[0], best);
10. }
11. }
12. **else**{
13. **int** i;
14. **for** (i = 0; i < MAX\_QUEUE\_LEVEL; i++){//从最高层队列依次向下查找本层队列中最靠近队首的线程，如果找到则不再向下继续查找
15. best = Get\_Front\_Of\_Thread\_Queue(&s\_runQueue[i]);
16. **if** (best != NULL){
17. Remove\_Thread(&s\_runQueue[i], best);
18. **break**;
19. }
20. }
21. }
22. KASSERT(best != NULL);//如果当前没有可执行进程，则至少应该找到 Idle 线程
23. **return** best;
24. Print("Scheduling %x\n", best);
25. }

除此之外还需要修改Wait函数

1. **void** Wait(**struct** Thread\_Queue\* waitQueue)
2. {
3. **struct** Kernel\_Thread\* current = g\_currentThread;
5. KASSERT(!Interrupts\_Enabled());
7. /\* 如果为 MLF 调度策略则下次运行时线程应进入高一优先级的队列(即队列数减一)
8. RR 调度策略时不受影响，因为已经运行在最高优先级的线程队列 \*/
9. **if**(current->pid != IdleThread->pid && current->currentReadyQueue > 0)
10. --current->currentReadyQueue;
12. /\* Add the thread to the wait queue. \*/
13. current->blocked = **true**;
14. Enqueue\_Thread(waitQueue, current);
16. /\* Find another thread to run. \*/
17. Schedule();
18. }

其他函数细节之前有，不再贴出

1. 将之前的project的修改加入之后，在/src/geekos/sysch.c中定义信号量的结构体，声明相关的信号量操作方法 --- 获取信号量P操作，释放信号量V操作
2. #include <geekos/errno.h>
3. #include <geekos/string.h>
4. #include <geekos/malloc.h>
5. /\* 信号量列表 \*/
6. **struct** Semaphore\_List g\_semList;
7. /\* 当前信号量 ID 最大值 \*/
8. **int** g\_curSID = 0;
10. /\* 信号量入列表 \*/
11. **static** \_\_inline\_\_ **void** Enqueue\_Semaphore(
12. **struct** Semaphore\_List \*list, **struct** Semaphore \*sem)
13. {
14. Add\_To\_Back\_Of\_Semaphore\_List(list, sem);
15. }
16. /\* 信号量出列表 \*/
17. **static** \_\_inline\_\_ **void** Remove\_Semaphore(
18. **struct** Semaphore\_List \*list, **struct** Semaphore \*sem)
19. {
20. Remove\_From\_Semaphore\_List(list, sem);
21. }
23. pSemaphore isSemExistByName(**char** \*nameSem, **int** nameLen)//根据信号量名检查信号量是否存在
24. {
25. **if** (g\_curSID == 0)**return** NULL;
26. pSemaphore sem = g\_semList.head;
27. **while** (sem != NULL){
28. **if** (strncmp(sem->semaphoreName, nameSem, nameLen) == 0)**break**;
29. sem = Get\_Next\_In\_Semaphore\_List(sem);
30. }
31. **return** sem;
32. }
34. pSemaphore isSemExistBySID(**int** sid)//根据信号量名检查信号量是否存在
35. {
36. pSemaphore sem = g\_semList.head;
37. **while** (sem != NULL){
38. **if** (sem->semaphoreID == sid)**break**;
39. sem = Get\_Next\_In\_Semaphore\_List(sem);
40. }
41. **return** sem;
42. }
44. **int** getIndexInRegThreadList(pSemaphore sem)
45. {
46. **int** i;
47. **for** (i = 0; i < sem->registeredThreadCount; i++)
48. **if** (sem->registeredThreads[i] == g\_currentThread) **break**;
49. **return** (i != sem->registeredThreadCount ? i : -1);
50. }
52. **int** Create\_Semaphore(**char** \*semName, **int** nameLen, **int** initCount)
53. {                             //创建一个信号量
54. KASSERT(semName != NULL); //错误中断
55. KASSERT(nameLen > 0 && nameLen <= MAX\_SEMAPHORE\_NAME);
56. KASSERT(initCount >= 0);
57. KASSERT(strnlen(semName, MAX\_SEMAPHORE\_NAME) == nameLen);
58. **if** (g\_curSID == 0){ //如果未初始化信号量列表结构体则先进行初始化
59. g\_semList.head = NULL;
60. g\_semList.tail = NULL;
61. }
62. pSemaphore sem = isSemExistByName(semName, nameLen);//查找是否已经存在同名信号量
63. **if** (sem == NULL){//如果不存在则新建一个信号量
64. sem = (pSemaphore)Malloc(**sizeof**(**struct** Semaphore));//初始化信号量
65. **if** (sem == NULL){
66. Print("Error! Out of Memory Space\n");
67. **return** ENOMEM;
68. }
69. memset(sem, 0, **sizeof**(**struct** Semaphore));
70. g\_curSID++;//设置信号量相关值
71. sem->semaphoreID = g\_curSID;
72. strncpy(sem->semaphoreName, semName, MAX\_SEMAPHORE\_NAME);
73. sem->value = initCount;
74. sem->registeredThreadCount = 0;
75. Clear\_Thread\_Queue(&sem->waitingThreads);
76. Add\_To\_Back\_Of\_Semaphore\_List(&g\_semList, sem);//将新创建的信号量加入到信号量列表中
77. }
78. sem->registeredThreads[sem->registeredThreadCount] = g\_currentThread;
79. sem->registeredThreadCount++;
80. **return** sem->semaphoreID;
81. }
83. **int** P(**int** sid) //信号量 P(获取)操作
84. {
85. KASSERT(sid > 0); //错误中断
87. pSemaphore sem = isSemExistBySID(sid);
88. **if** (sem == NULL)
89. {
90. Print("Error! Connot Find Semaphore with SID=%d\n", sid);
91. **return** -1;
92. }
94. **int** threadIndex = getIndexInRegThreadList(sem);
95. **if** (threadIndex == -1)
96. {
97. Print("Error! Current Thread is not Using the Semaphore with SID=%d\n", sid);
98. **return** -1;
99. }
101. **if** (sem->value == 0)
102. Wait(&sem->waitingThreads);
103. sem->value--;
105. **return** 0;
106. }
108. **int** V(**int** sid) //信号量 V(释放)操作
109. {
110. KASSERT(sid > 0); //错误中断
112. pSemaphore sem = isSemExistBySID(sid);
113. **if** (sem == NULL)
114. {
115. Print("Error! Connot Find Semaphore with SID=%d\n", sid);
116. **return** -1;
117. }
119. **int** threadIndex = getIndexInRegThreadList(sem);
120. **if** (threadIndex == -1)
121. {
122. Print("Error! Current Thread is not Using the Semaphore with SID=%d\n", sid);
123. **return** -1;
124. }
126. sem->value++;
127. **if** (sem->value == 1)
128. Wake\_Up\_One(&sem->waitingThreads);
130. **return** 0;
131. }
133. **int** Destroy\_Semaphore(**int** sid)//销毁一个信号量
134. {
135. KASSERT(sid > 0);//错误中断
136. pSemaphore sem = isSemExistBySID(sid);
137. **if** (sem == NULL){
138. Print("Error! Connot Find Semaphore with SID=%d\n", sid);
139. **return** -1;
140. }
141. **int** threadIndex = getIndexInRegThreadList(sem);
142. **if** (threadIndex == -1){
143. Print("Error! Current Thread is not Using the Semaphore with SID=%d\n", sid);
144. **return** -1;
145. }
146. sem->registeredThreadCount--;
147. **int** i;
148. **for** (i = threadIndex; i < sem->registeredThreadCount; i++){
149. sem->registeredThreads[i] = sem->registeredThreads[i + 1];
150. }
151. sem->registeredThreads[sem->registeredThreadCount] = NULL;
152. **if** (sem->registeredThreadCount == 0){//唤醒该信号量等待队列中所有线程
153. Wake\_Up(&sem->waitingThreads);
154. Free(sem);
155. Remove\_From\_Semaphore\_List(&g\_semList, sem);
156. g\_curSID--;
157. }
158. **return** 0;
159. }

在该函数中就实现了信号量队列的定义、信号量的删除等操作方法，以及对于信号量获取的P方法和释放信号量的V方法

1. 定义了信号量队列之后，需要在/src/geekos/syscall.c方法中实现对于进程信号量创建的操作
2. **static** **int** Sys\_CreateSemaphore(**struct** Interrupt\_State \*state)
3. {
4. **int** res;
5. ulong\_t userAddr = state->ebx;   //信号量名字符串所在用户空间地址
6. ulong\_t nameLen = state->ecx;    //信号量名长度
7. ulong\_t initCount = state->edx; //信号量初始值
8. **if** (nameLen <= 0 || initCount < 0 || nameLen > MAX\_SEMAPHORE\_NAME){//如果传入参数不正确则返回错误
9. Print("Error! Semaphore Params incorrect\n");
10. **return** EINVALID;
11. }
12. **char** \*semName = NULL;
13. res = Copy\_User\_String(userAddr, nameLen, MAX\_SEMAPHORE\_NAME, &semName);//从用户空间拷贝信号量名字符串到内核空间
14. **if** (res != 0){
15. Print("Error! Cannot copy string from user spcce\n");
16. **return** res;
17. }
18. **if** (strnlen(semName, MAX\_SEMAPHORE\_NAME) != nameLen){//判断信号量名的合法性(中间是否含有'\0'字符)
19. Print("Error! Semaphore Name is Invalid\n");
20. **return** EINVALID;
21. }
22. res = Create\_Semaphore(semName, nameLen, initCount);//创建一个信号量
23. **return** res;
24. }

创建信号量函数中就会定义信号量的名称、长度、初始值等

1. 需要创建测试函数，在/user/ping.c中修改
2. #include <conio.h>
3. #include <process.h>
4. #include <sched.h>
5. #include <sema.h>
6. #include <string.h>
8. **int** main(**int** argc, **char** \*\*argv)
9. {
10. **int** i, j;       /\* loop index \*/
11. **int** scr\_sem;    /\* id of screen semaphore \*/
12. **int** time;       /\* current and start time \*/
13. **int** ping, pong; /\* id of semaphores to sync processes b & c \*/
15. time = Get\_Time\_Of\_Day();
16. scr\_sem = Create\_Semaphore("screen", 1); /\* register for screen use \*/
17. ping = Create\_Semaphore("ping", 1);
18. pong = Create\_Semaphore("pong", 0);
20. **for** (i = 0; i < 5; i++)
21. {
22. P(pong);
23. P(scr\_sem);
24. Print("Ping\n");
25. V(scr\_sem);
26. **for** (j = 0; j < 35; j++)
27. ;
28. V(ping);
29. }
31. time = Get\_Time\_Of\_Day() - time;
32. P(scr\_sem);
33. Print("Process Ping is done at time: %d\n", time);
34. V(scr\_sem);
36. Destroy\_Semaphore(pong);
37. Destroy\_Semaphore(ping);
38. Destroy\_Semaphore(scr\_sem);
40. **return** (0);
41. }
42. 修改/user/pong.c, 定义pong进程获取信号量
43. #include <conio.h>
44. #include <process.h>
45. #include <sched.h>
46. #include <sema.h>
47. #include <string.h>
49. **int** main(**int** argc, **char** \*\*argv)
50. {
51. **int** i, j;       /\* loop index \*/
52. **int** scr\_sem;    /\* id of screen semaphore \*/
53. **int** time;       /\* current and start time \*/
54. **int** ping, pong; /\* id of semaphores to sync processes b & c \*/
56. time = Get\_Time\_Of\_Day();
57. scr\_sem = Create\_Semaphore("screen", 1); /\* register for screen use \*/
58. ping = Create\_Semaphore("ping", 1);
59. pong = Create\_Semaphore("pong", 0);
61. **for** (i = 0; i < 5; i++)
62. {
63. P(ping);
64. P(scr\_sem);
65. Print("Pong\n");
66. V(scr\_sem);
67. **for** (j = 0; j < 35; j++)
68. ;
69. V(pong);
70. }
72. time = Get\_Time\_Of\_Day() - time;
73. P(scr\_sem);
74. Print("Process Pong is done at time: %d\n", time);
75. V(scr\_sem);
77. **return** (0);
78. }
79. 修改/user/workload.c，在其中创建ping、pong进程
80. **int** main(**int** argc , **char** \*\* argv)
81. {
82. **int** policy = -1;
83. **int** start;
84. **int** elapsed;
85. **int** quantum;
86. **int** scr\_sem;          /\* sid of screen semaphore \*/
87. **int** id1, id2, id3;        /\* ID of child process \*/
89. **if** (argc == 3) {
90. **if** (!strcmp(argv[1], "rr")) {
91. policy = 0;
92. } **else** **if** (!strcmp(argv[1], "mlf")) {
93. policy = 1;
94. } **else** {
95. Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);
96. Exit(1);
97. }
98. quantum = atoi(argv[2]);
99. Set\_Scheduling\_Policy(policy, quantum);
100. } **else** {
101. Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);
102. Exit(1);
103. }
105. start = Get\_Time\_Of\_Day();
106. scr\_sem = Create\_Semaphore ("screen" , 1)  ;
108. P (scr\_sem) ;
109. Print ("\*\*\*\*\*\*\*\*\*\*\*\*\* Start Workload Generator \*\*\*\*\*\*\*\*\*\n");
110. V (scr\_sem) ;
112. id1 = Spawn\_Program ("/c/long.exe", "/c/long.exe"
113. ) ;
114. P (scr\_sem) ;
115. Print ("Process Long has been created with ID = %d\n",id1);
116. V (scr\_sem) ;

119. id2 = Spawn\_Program ("/c/ping.exe", "/c/ping.exe"
120. ) ;
122. P (scr\_sem) ;
123. Print ("Process Ping has been created with ID = %d\n",id2);
124. V (scr\_sem) ;
126. id3 = Spawn\_Program ("/c/pong.exe", "/c/pong.exe"
127. ) ;
128. P (scr\_sem) ;
129. Print ("Process Pong has been created with ID = %d\n",id3);
130. V (scr\_sem) ;
132. Wait(id1);
133. Wait(id2);
134. Wait(id3);
136. elapsed = Get\_Time\_Of\_Day() - start;
137. Print ("\nTests Completed at %d\n", elapsed) ;
138. **return** 0;
139. }

执行上面这个脚本就可以实现ping、pong的交替输出，除此之外，还可以编写其他的测试脚本，比如semtest.c

1. // A test program for semaphores
3. #include "libuser.h"
4. #include "libio.h"
6. **int** main( **int** argc, **char** \*\* argv)
7. {
8. **int** semkey, result;
10. Print("Create\_Semaphore()...\n");
11. semkey = Create\_Semaphore("semtest", 3);
12. Print("Create\_Semaphore() returned %d\n", semkey);
14. **if** (semkey < 0)
15. **return** 0;
17. Print("P()...\n");
18. result = P(semkey);
19. Print("P() returned %d\n", result);
21. Print("P()...\n");
22. result = P(semkey);
23. Print("P() returned %d\n", result);
25. Print("V()...\n");
26. result = V(semkey);
27. Print("V() returned %d\n", result);

30. Print("Destroy\_Semaphore()...\n");
31. result = Destroy\_Semaphore(semkey);
32. Print("Destroy\_Semaphore() returned %d\n", result);
34. **return** 0;
35. }

或者schedtest.c

1. #include <conio.h>
2. #include <process.h>
3. #include <sched.h>
4. #include <sema.h>
5. #include <string.h>
7. #if !defined (NULL)
8. #define NULL 0
9. #endif
11. **int** main(**int** argc , **char** \*\* argv)
12. {
13. **int** policy = -1;
14. **int** quantum;
16. **int** id1, id2, id3;        /\* ID of child process \*/
18. **if** (argc == 3) {
19. **if** (!strcmp(argv[1], "rr")) {
20. policy = 0;
21. } **else** **if** (!strcmp(argv[1], "mlf")) {
22. policy = 1;
23. } **else** {
24. Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);
25. Exit(1);
26. }
27. //    quantum = atoi(argv[2]);
28. //    Set\_Scheduling\_Policy(policy, quantum);
29. } **else** {
30. Print("usage: %s [rr|mlf] <quantum>\n", argv[0]);
31. Exit(1);
32. }
34. quantum = atoi(argv[2]);
35. Set\_Scheduling\_Policy(policy, quantum);

38. id3 = Spawn\_Program ( "/c/sched3.exe", "/c/sched3.exe") ;
39. id1 = Spawn\_Program ( "/c/sched1.exe", "/c/sched1.exe") ;
40. id2 = Spawn\_Program ( "/c/sched2.exe", "/c/sched2.exe") ;

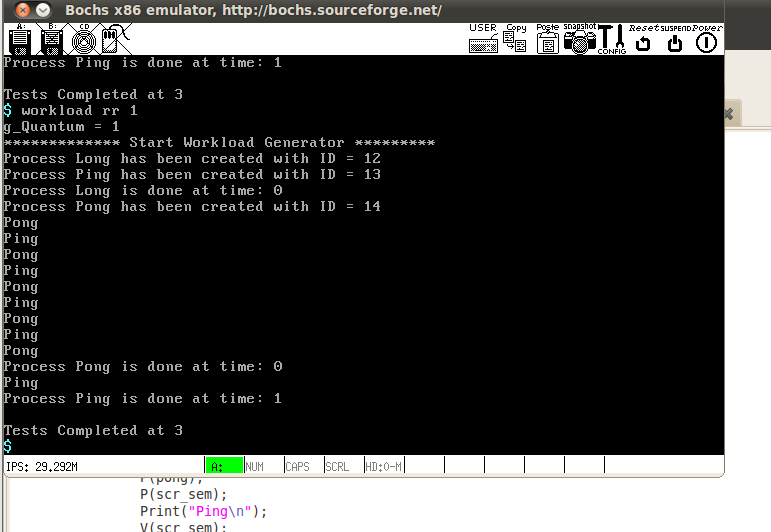
43. Wait(id1);
44. Wait(id2);
45. Wait(id3);
47. Print("\n");
49. **return** 0;
50. }

接下来就可以直接运行查看结果

首先make clean清理一下.o文件， 之后make编译之后，运行bochs -f bochsrc

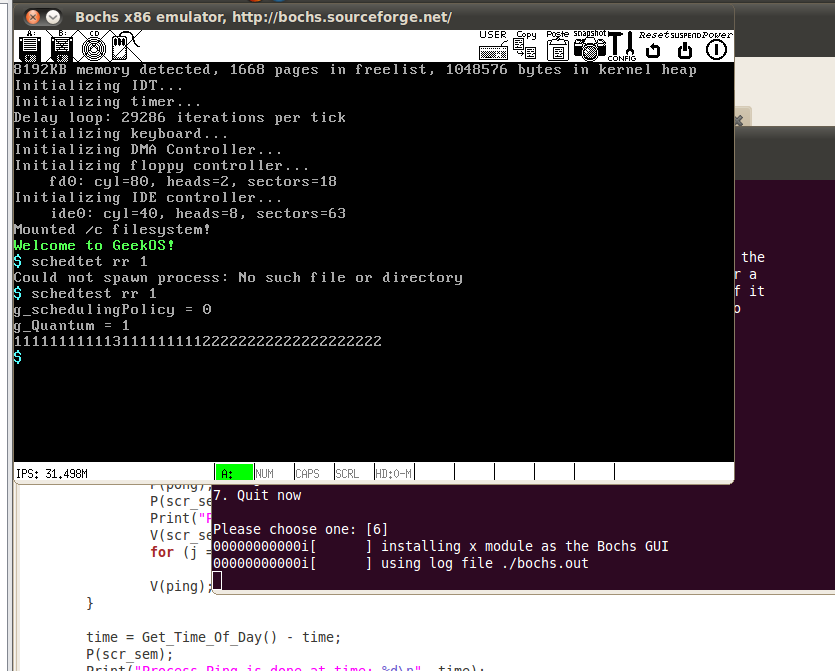
测试的结果如下

Workload rr 1

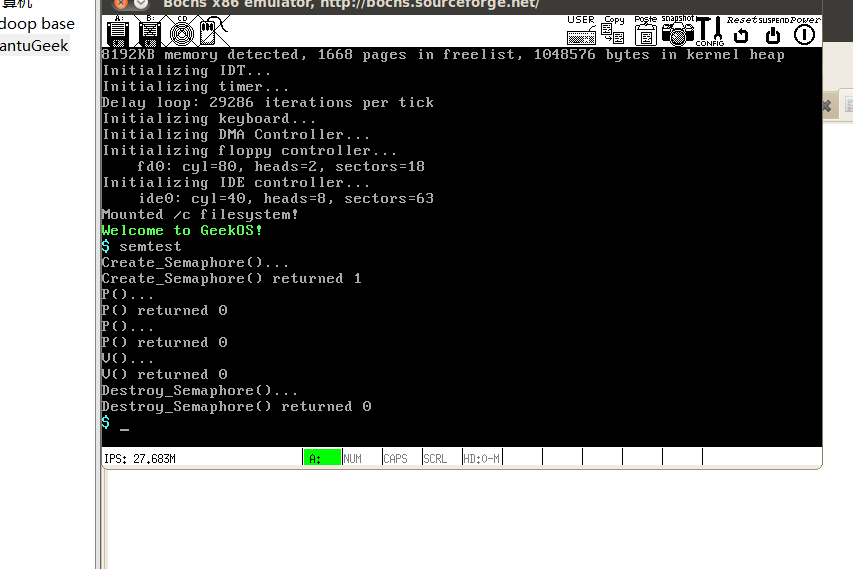


可以看到Ping和Pong交替打印了5次

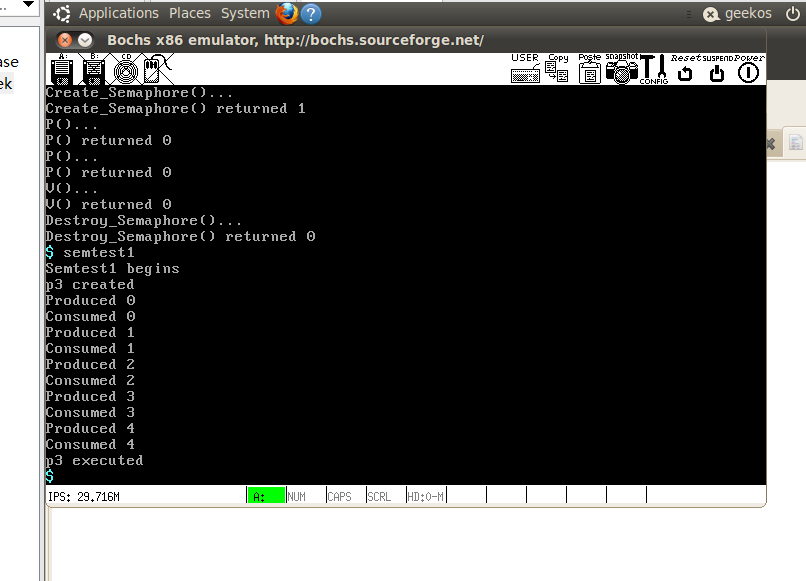
执行schedtest脚本



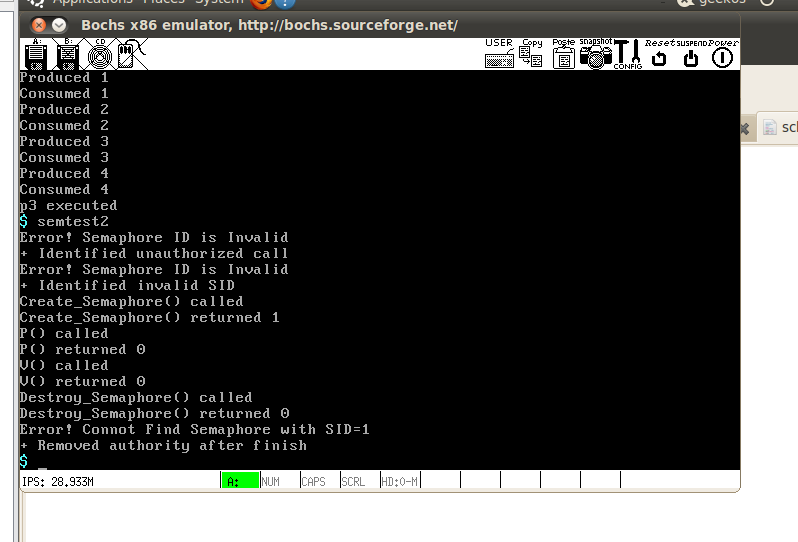
执行semtest脚本



执行semtest1脚本



执行semtest2脚本



实验结果完全符合预期，证明了进程Ping和Pong在互相争夺等待信号量

**四、实验分析**

Create\_Semaphore( )函数首先检查请求创建的这个信号量的名字是否存在，如果存在，那么就把 这个线程加入到这个信号量所注册的线程链表上；如果不存在，则分配内存给新的信号量，清空它的 线程队列，把当前的这个线程加入到它的线程队列中，设置注册线程数量为 1，初始化信号量的名字， 值和信号量的 ID，并把这个信号量添加到信号量链表上，最后返回信号量的 ID。

P 操作 Semaphore\_Acquire( )中，首先检查传入的信号量 ID 是否存在，如果存在，接着检查当前 线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就 把信号量的值减去 1，如果减去 1 后信号量的值小于 0，那么就把当前线程放入这个信号量的等待队 列上。

V 操作 Semaphore\_Release( )中，首先也是检查传入的信号量 ID 是否存在，如果存在，接着检查当前线程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了， 那就把信号量的值加上 1，如果加上 1 后信号量的值小于或等于 0，则要把该信号量里等待队列上的 一个线程唤醒。

Semaphore\_Destroy( )中，首先也是检查传入的信号量 ID 是否存在，如果存在，接着检查当前线 程是否注册使用了这个信号量，如果这两项检查任意一项失败了，那么就返回-1。如果成功了，就把该线程从这个信号量的注册的线程数组中删除，并把注册的线程数量减去 1。如果这个信号量的注册线程为 0 了，则把这个信号量从信号量链表中删除，并释放它的内存。到.\src\geekos\syscal.c 中实现 Sys\_CreateSemaphore(struct Interrupt\_State\* state)，注意，用户程序 传递的参数在 Interrupt\_State 对应的变量中，同时需要将用户程序内存空间的字符串复制到内核空间。

信号量分为整形信号量，记录型信号量，AND信号量以及信号量集。最初的信号量就是整形信号量，定义信号量为一个整型变量，仅能通过两个原子操作P，V来操作，所谓原子操作就是一组相连的操作要么不间断执行，要么不执行，两个操作称为wait和signal操作或者down和up操作，也就是PV操作，P意味测试，V意为增加

P操作信号量S – 1， V操作信号量S + 1。P操作，将S的值减少1，标识该进程需要一个临界资源，如果S < 0，那么说明S <=0 ,没有资源可用，将进程阻塞到与信号量S相关的阻塞队列中，如果S.value < 0， 那么S.value的绝对值实际上标识阻塞队列的长度，也就是等待使用资源的进程数量； V操作就是S.value增加1，表示释放一个资源，如果S.value <=0, 说明原来的S.value < 0，说明阻塞对垒中是有进程的，于是唤醒该队列中的一个进程，大于0就说明该阻塞队列没有进程了。

信号量机制运行的过程：

用原语对信号量进行保护

用信号量解决同步、互斥问题

上面说过，P和V操作是一对原语，这实际上是对信号量的保护。信号量是一种被多个进程共享的资源，也是一种临界资源，因此对信号量的修改(PV操作)必须是互斥的，同一时间只能有一个进程对信号量进行修改，这就是对信号量的保护，也就是对信号量的修改必须是原语。让PV操作变成原语（对信号量的保护）可以采用互斥的软/硬件实现方法。最简单的方法就是开关中断（仅适用于单核CPU）或者TS指令（信号量给临界资源上锁，TS指令再给信号量上锁），使信号量的修改变成原语。

信号量-同步：初值为0

对于一个资源的使用和释放，不同的进程之间有明确先后关系，比如生产者产生数据，消费者才能使用数据。具体如下：

semaphore s = 0;

// 执行顺序： producer -> consumer

producer() {

// ...

// 产生数据的代码

V(s);

}

consumer() {

// ...

// 使用数据的代码

P(s);

P和V组成一对，分别在不同的进程中，先执行的进程为P，后执行的进程为V。

考虑进程同步主要依靠以下步骤：

找出一前一后的同步关系（画前驱图，每条线是一个信号量）

设置同步信号量的初始值为0

释放资源用V，消耗资源用P

信号量-互斥：初值为1

对于一个共享资源的读写，不同进程间是互斥的，这时使用初值为1的互斥信号量（也称为互斥锁），实现进程互斥。

这里可以将临界区看做是数量为1的资源，因此初值为1。

semaphore mutex = 1;

p1() {

P(mutex); // 测一下有没有进程在使用这个信号量

// ... 读写资源

V(mutex); // 释放该信号量

}

p2() {

P(mutex);

// ...

V(mutex);

PV组成一对，在同一个进程中，要读写共享资源前先P一下，看是否有人用。读写完再V一下释放这个资源。P和V来包裹读写资源的操作，读写临界资源的代码放在临界区。

思考步骤：

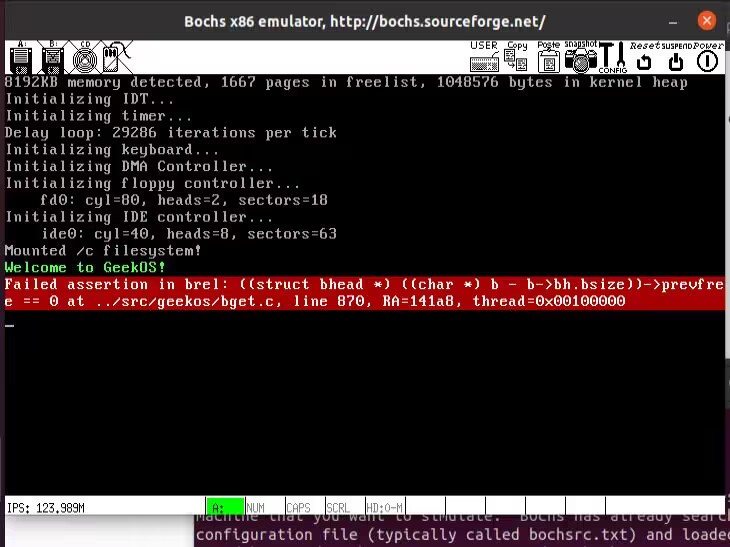
分析问题，确定临界区

设置互斥信号量，初始值为1

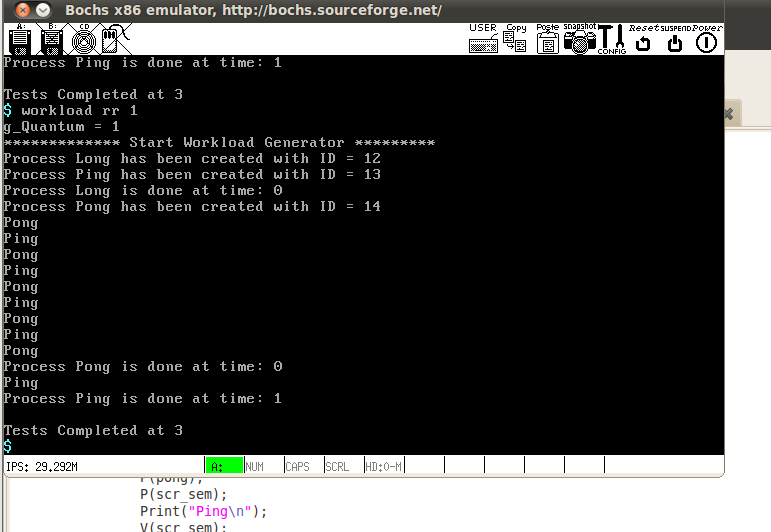
临界区前后P V夹紧

**五、所遇问题及解决方法**

* 最开始运行项目显示错误



解决办法： 分析发现是.o文件出现了问题，但是.o文件时编译之后的文件，所以这里就采用make clean清除了.o文件之后，重新make depend，再make，重新进行bochs -f bochsrc 运行结果正常



* 在完成项目代码填写进行测试的时候，发现 schedtest.exe 这个文件在 /project3/build/user/文件夹中存在，但是却无法在系统中执行，总是会显示找不到文件。

解决方法：分析 GeekOS 系统编译阶段 make程序的执行结果，会发现 schedtest.exe 文件的名称并没有显示完全，也就是说在GeekOS的挂载磁盘中，这个文件的名称为“schedtest.ex” 而非“schedtest.exe”，这也就是为什么直接输入程序名或者加上路径都无法执行该文件的原因。同时还能发现了 Makefile文件中并没有对 semtest.c和 null.c进行编译。exe文件是通过/tools/builtFat.ex这个小程序生成 img格式磁盘映像文件的，找到这个文件的源代码文件位置/project3/src/tools/buildFat.c，会发现是其中使用的 directoryEntry 这个结构中 fileName 字段 的长度不足。因此通过 grep 指令找到 directoryEntry 结构体所在文件/project3/include/geekos/pfat.h。最后修改/build/Makefile文件，在“USER\_C\_SRCS”后面添加“semtest.c”和“null.c”。

**六、思考与练习**

1. 说明同步与互斥的关系。

* **同步**是一种合作关系，为完成某种任务而建立的多个进程或者线程之间的协调调用，次序等待，传递消息告知资源占用情况。
* **互斥**是一种制约关系，当一个进程或者多个进程进入临界区后会进行加锁操作，此时其他进程（线程）无法进入临界区，只有当该进程（线程）使用后进行解锁其他人才可以使用，这种技术往往是通过阻塞完成。

互斥：是指某一资源同时只允许一个访问者对其进行访问，具有唯一性和排它性。但互斥无法限制访问者对资源的访问顺序，即访问是无序的。

同步：是指在互斥的基础上（大多数情况），通过其它机制实现访问者对资源的有序访问。在大多数情况下，同步已经实现了互斥，特别是所有写入资源的情况必定是互斥的。少数情况是指可以允许多个访问者同时访问资源。