电子科技大学 学院

**实 验 报 告**

（实验）课程名称 计算机操作系统

**电子科技大学教务处制表**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：唐以恒 学 号：2017110801017 指导教师： 刘杰彦**

**实验地点：主楼A2-412 实验时间：2020年5月**

**一、实验室名称：计算机实验室**

**二、实验项目名称：进程与资源管理实验**

**三、实验学时：6**

**四、实验原理：**

**1.总体设计：**

系统总体架构如图1所示，最右边部分为进程与资源管理器，属于操作系统内核的功能。该管理器具有如下功能：完成进程创建、撤销和进程调度；完成多单元资源的管理；完成资源的申请和释放；完成错误检测和定时器中断功能。



图1 系统总体结构

图1中间绿色部分为驱动程序test shell, 设计与实现test shell，该test shell将调度所设计的进程与资源管理器来完成测试。Test shell的应具有的功能：

从终端或者测试文件读取命令；

将用户需求转换成调度内核函数（即调度进程和资源管理器）；

在终端或输出文件中显示结果：如当前运行的进程、错误信息等。

图1最左端部分为：通过终端（如键盘输入）或者测试文件来给出相应的用户命令，以及模拟硬件引起的中断。

**2.Test Shell设计**

Test shell要求完成的命令（Mandatory Commands）：

-init

-cr <name> <priority>(=1 or 2) // create process

-de <name> // delete process

-req <resource name> <# of units> // request resource

-rel <resource name> <# of units> // release resource

-to // time out

可选实现的命令：

-lp: all processes and their status

-lr: all resources and their status

- provide information about a given process

（注：具体的功能实现此处略去，详见第八部分：实验步骤）

**3.进程管理设计**

进程状态： ready/running/blocked

进程操作：

* 创建(create)： (none) -> ready
* 撤销(destroy)： running/ready/blocked -> (none)
* 请求资源(Request): running -> blocked (当资源没有时，进程阻塞)
* 释放资源(Release): blocked -> ready (因申请资源而阻塞的进程被唤醒)
* 时钟中断(Time\_out): running -> ready
* 调度：ready -> running / running ->ready

**4.进程控制块结构（PCB）**

* PID（name）
* resources //: resource which is occupied
* Status: Type & List// type: ready, block, running…., //List: RL(Ready list) or BL(block list)
* Creation\_tree: Parent/Children
* Priority: 0, 1, 2 (Init, User, System)

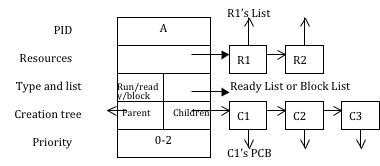


图2 PCB结构示意

就绪进程队列：Ready list (RL)



图3 Ready list 数据结构

3个级别的优先级，且优先级固定无变化

2 =“system”

1 = “user”

0 = “init”

每个PCB要么在RL中，要么在block list中 。当前正在运行的进程，根据优先级，可以将其放在RL中相应优先级队列的首部。

Init进程在启动时创建，可以用来创建第一个系统进程或者用户进程。新创建的进程或者被唤醒的进程被插入到就绪队列（RL）的末尾。

示例：

图4中，虚线表示进程A为运行进程，在进程A运行过程中，创建用户进程B：cr B 1，数据结构间关系图4所示：

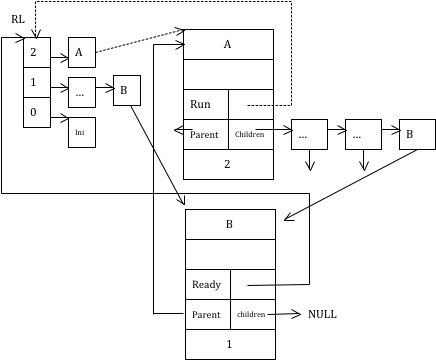


图4 进程数据结构间关系

(为了简单起见，A和B分别指向RL的链接可以不要)

**5.资源管理设计**

资源的表示：设置固定的资源数量，4类资源，R1，R2，R3，R4，每类资源Ri有i个

资源控制块Resource control block (RCB) 如图5所示

* RID: 资源的ID
* Status: 空闲单元的数量
* Waiting\_List: list of blocked process



图5 资源数据结构RCB

**6. 进程调度与时钟中断设计**

调度策略：

* 基于3个优先级别的调度：2，1，0
* 使用基于优先级的抢占式调度策略，在同一优先级内使用时间片轮转（RR）
* 基于函数调用来模拟时间共享
* 初始进程(Init process)具有双重作用：

（1）虚设的进程：具有最低的优先级，永远不会被阻塞

（2）进程树的根

时钟中断（Time out）：模拟时间片到或者外部硬件中断

**7. 系统初始化设计**

启动时初始化管理器：

具有3个优先级的就绪队列RL初始化；

Init进程；

4类资源，R1，R2，R3，R4，每类资源Ri有i个

**五、实验目的：**

设计和实现进程与资源管理，并完成Test Shell 的编写，以建立系统的进程管理、调度、资源管理和分配的知识体系，从而加深对操作系统进程调度和资源管理功能的宏观理解和微观实现技术的掌握。

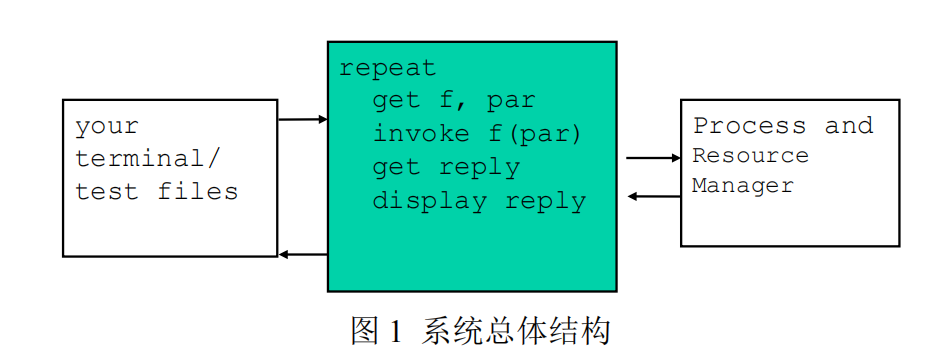
**六、实验内容：**

**七、实验器材（设备、元器件）：**

操作系统Ubuntu 19.02、编程语言C、编译器gcc、makefile

**八、实验步骤：**

实验步骤包括: （以下内容主要针对实验1）

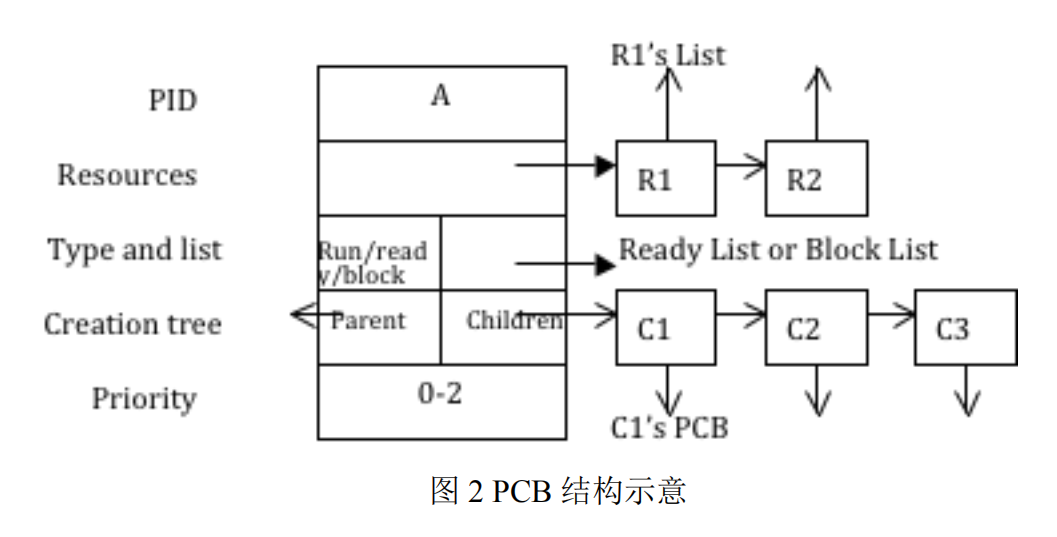
* **a)  系统功能需求分析;**
* 1. 进程控制：能完成进程的控制，如进程的创建与销毁、进程的状态转换
* 2. 进程调度：能基于优先级调度算法完成进程的调度，模拟时钟中断，完成对时钟中断的处理，在同等优先级的进程中按照时间片轮转调度算法对进程进行调度
* 3. 资源管理：能够完成资源的分配与释放
* 4. 进程同步：能够对进程在资源管理过程中的同步
* 5. 测试程序：能够在管理器之外如用户终端发送指令，或者从文件批量读取指令，实际操作进程
* **b)  总体框架设计:说明具体原理，用到的方法或者算法，模块之间的调用关系，总体工作流程**
* 

最右侧为进程与资源管理器部分，属于操作系统内核的功能，它具有的功能有：进程创建与销毁、进程调度多资源管理、进程同步、时钟中断。

中间绿色部分为Shell外壳程序，负责测试我们设计的进程与资源管理器，它具有的功能有：从用户终端或输入文件读取指令、将用户的指令转换为实际模拟进程控制的内核函数、在用户终端或输出文件显示调度结果

* **c)  具体模块的设计 需要理清模块间关系，给出各模块详细的设计思想、设计流程和主要代码，主要代码要求注释;**

1. **进程管理设计**
   1. **进程PCB**

****

- PID：表示进程的唯一标识符

- Resources：表示进程申请到正在使用的资源

- Type and list：表示进程的状态及对应到的队列（就绪队列、阻塞队列...）

- Creation tree：表示进程创建树，一个进程可以创建多个子进程，但一个子进程只能被一个父进程创建，所以使用树型结构来表示这种数据关系

- Priority：表示进程的调度优先级，分为0、1、2（分贝对应初始级、用户级、内核级）

1. /\*
2. \* structure of the process control block (PCB)
3. \*/
4. **struct** pcb {
5. **int** id;// the unique process identifier
6. re\_l\_node\_t  \*\*resource\_list; // the resource list
7. process\_status\_t \*status;// the process status
8. creation\_tree\_t \*creation\_tree;// the pointers to parent process and child process list
9. **int** priority;
10. pcb\_t \*rl\_next;// the next pointer for process in the ready list
11. pcb\_t \*bl\_next;// the next pointer for process in the blocked list
12. pcb\_t \*cl\_next;// the next pointer for process in the child list
13. };
14. /\*
15. \* the structure of the node for resoure list in pcb.
16. \*/
17. **struct** re\_l\_node {
18. rcb\_t \*r;// the rcb
19. **int** n;// the numebr of the resoure
20. re\_l\_node\_t \*next;
21. };
22. /\*
23. \* structure of the status for pcb
24. \*/
25. **struct** process\_status {
26. **int** type; // the process status: ready, run and block
27. pcb\_t \*\*list;// the list process belong to: Blocked List or Ready List
28. };
29. /\*
30. \* the structure of the creation\_tree for pcb.
31. \*/
32. **struct** creation\_tree {
33. pcb\_t \*parent; // the parent process
34. pcb\_t \*\*child;// the child process list
35. };
    1. **进程状态**

- 就绪态：Ready

- 运行态：Runnning

- 阻塞态：Blocked

1. /\* constant for process status \*/
2. #define STATUS\_READY 0
3. #define STATUS\_RUNNING 1
4. #define STATUS\_BLOCKED 2
   1. **进程操作**

- 创建进程（create）：进程创建完成之后进程状态转换为 Ready

- 销毁进程（destroy）：进程销毁完成之后进程PCB被回收，进程状态由Ready/Running/Blocked变为None

- 请求资源（Request）：进程申请资源失败，则进程状态从Running变为Blocked，否则进程状态不发生变化

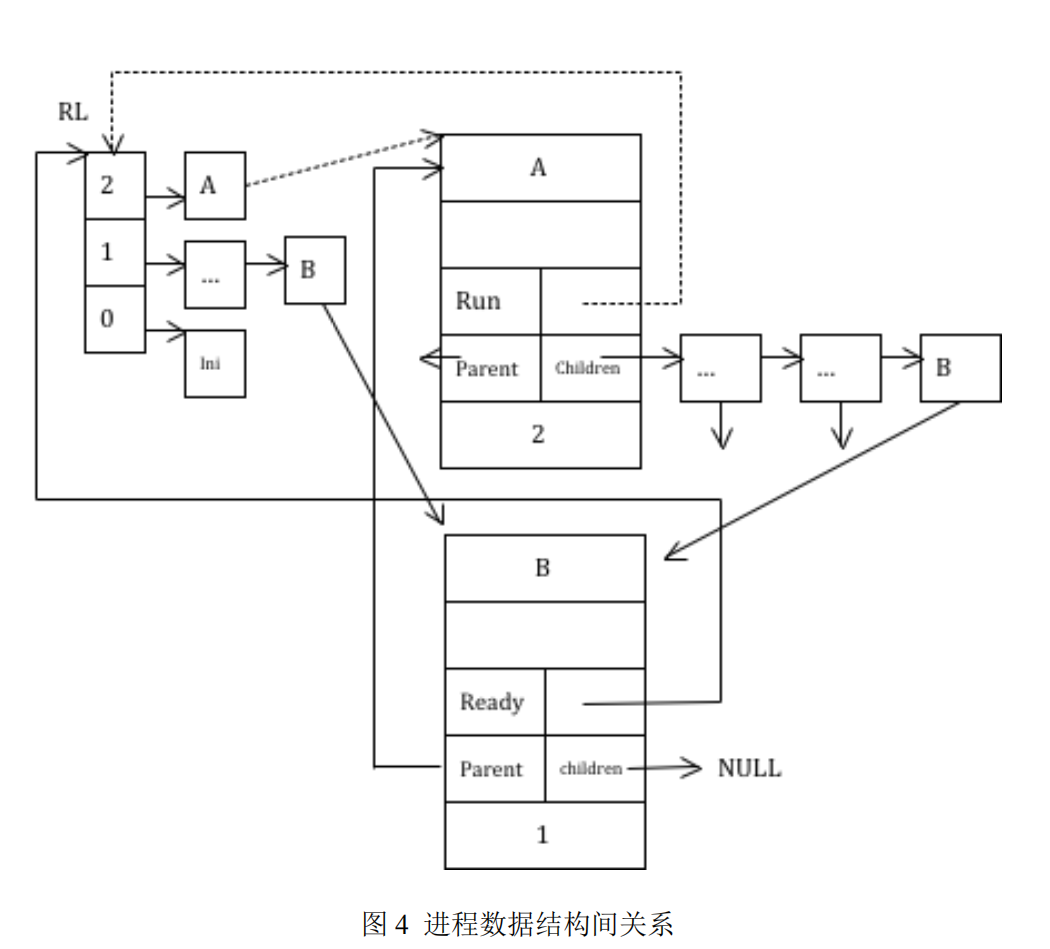
- 释放资源（Release）：进程释放资源之后，如果正好有因为申请此资源而阻塞的进程，并且数目满足需求的话，进程状态从Blocked变为Ready

- 时钟中断（Interrupt）：模拟时钟中断，当前进程状态由Running变为Ready

- 进程调度（Scheduler）：模拟进程调度

1. /\* create a new process with id and priority \*/
2. pcb\_t \*create(**const** **int** id, **int** priority);
4. /\* destroy a process \*/
5. **void** destroy(**int** id);
7. /\* current running process request a certain number of resoures \*/
8. **void** request(**int** id, **int** num);
10. /\* current running process release a certain number of resoures \*/
11. **void** release(**int** id, **int** num);
13. /\* schedule process \*/
14. **void** scheduler();
16. /\* clock interrupt \*/
17. **void** interrupt();
    1. **主要函数**
       1. **创建进程**

根据指定的PID和优先级创建一个新的进程，首先要初始化它的PCB，然后将其插入到当前进程的Creation\_tree中，接着将其插入到就绪队列中，调用Scheduler()进行调度

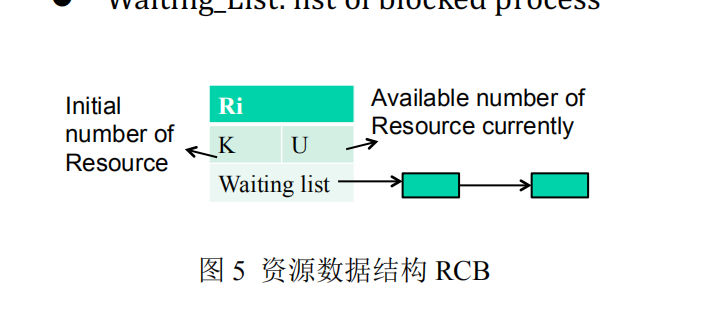


1. pcb\_t \*get\_new\_pcb() {**return** (pcb\_t \*) malloc(**sizeof**(pcb\_t)); }
2. /\*
3. \* create - create a new process
4. \*/
5. pcb\_t \*create(**int** id, **int** priority) {
6. // create a new pcb and initialize the pcb
7. pcb\_t \*p = get\_new\_pcb();
8. p->id = id;
9. p->resource\_list = calloc(1, **sizeof**(re\_l\_node\_t \*));
10. p->status = (process\_status\_t \*) malloc(**sizeof**(process\_status\_t));
11. p->status->type = STATUS\_READY;
12. p->status->list = &rl;
13. p->creation\_tree = (creation\_tree\_t \*) malloc(**sizeof**(creation\_tree\_t));
14. p->creation\_tree->parent = self;
15. p->creation\_tree->child = calloc(1, **sizeof**(pcb\_t \*));
16. p->priority = priority;
17. p->rl\_next = NULL;
18. p->bl\_next = NULL;
19. p->cl\_next = NULL;
21. // insert the process into the child list
22. **if** (self != NULL) insert\_into\_child\_list(self->creation\_tree->child, p); // self means current running process
24. // insert the process into the ready list
25. insert\_into\_ready\_list(&rl, p);
27. // schedule if needed
28. scheduler();
30. **return** p;
31. }

**1.4.2.销毁进程**

销毁一个进程，会递归的销毁它创建出来的所有的子进程，以及子进程创建出来的进程，销毁过程中会将进程从就绪队列、阻塞队列，或者资源排队队列中移除，释放被销毁进程占有的全部资源

1. /\*
2. \* destroy - destroy the target process with its all descendants.
3. \*/
4. **void** destroy(**int** id) {
5. // get the target process
6. pcb\_t \*p = get\_pcb(id);
7. **if** (p == NULL) **return**;
9. // remove itself from the child list of its parent process
10. remove\_from\_child\_list(p->creation\_tree->parent->creation\_tree->child, p);
12. // kill itself and all its descendants
13. kill\_tree(p);
15. // schedule if needed
16. scheduler();
17. }
19. **static** **void** kill\_tree(pcb\_t \*p) {
20. **if** (p == NULL) **return**;
21. // kill its descendants recursivly
22. pcb\_t \*tmp = \*(p->creation\_tree->child);
23. **while** (tmp != NULL) {
24. kill\_tree(tmp);
25. tmp = tmp->cl\_next;
26. }
27. // remove itself from all possible list.
28. **if** (p->status->type == STATUS\_RUNNING) {
29. interrupt();
30. } **else** **if** (p->status->type == STATUS\_READY) {
31. remove\_from\_ready\_list(&rl, p);
32. } **else** **if** (p->status->type == STATUS\_BLOCKED) {
33. remove\_from\_blocked\_list(&bl, p);
34. }
35. // release all its resource
36. release\_all(p);
37. // delete the pcb in memory
38. delete\_pcb(p);
39. }
40. **static** **void** release\_all(pcb\_t \*p) {
41. re\_l\_node\_t \*tmp = \*(p->resource\_list);
42. **while** (tmp != NULL) {
43. tmp->r->status->u += tmp->n;
44. wa\_l\_node\_t \*tmp2 = \*(tmp->r->waiting\_list);
45. **while** (tmp2 != NULL) {
46. **int** req\_num = tmp2->n;
47. **if** (tmp->r->status->u >= req\_num) {
48. remove\_from\_blocked\_list(&bl, tmp2->p);
49. tmp2->p->bl\_next = NULL;
50. tmp2->p->status->type = STATUS\_READY;
51. tmp2->p->status->list = &rl;
52. insert\_into\_ready\_list(&rl, tmp2->p);
53. tmp->r->status->u -= req\_num;
54. insert\_into\_resource\_list(tmp2->p->resource\_list, tmp->r, req\_num);
55. remove\_from\_waiting\_list(tmp->r->waiting\_list, tmp2->p);
56. }
57. tmp2 = tmp2->next;
58. }
59. scheduler();
60. tmp = tmp->next;
61. }
62. }
63. **资源管理设计**
    1. **资源RCB**

****

- RID：表示资源的唯一标识符

- Status：表示资源的状态

- K：表示资源的初始数量

- V：表示资源当前的可用数量

- Waiting List：因申请该资源而被阻塞的进程队列

1. /\*
2. \* structure of the resoure control block (RCB)
3. \*/
4. **struct** rcb {
5. **int** id;// the unique identifier of resoure
6. resource\_status\_t \*status;// the status of the resource
7. wa\_l\_node\_t \*\*waiting\_list;// the list of blocked process
8. };
9. /\*
10. \* structure of the status for rcb
11. \*/
12. **struct** resource\_status {
13. **int** k;// the initial number of resoure
14. **int** u;// the available number of resoure
15. };
    1. **请求资源**

申请资源按照FIFO顺序，一类资源有多个数量的情况，如果资源当前可用数量多于申请数量，则进程申请资源成功，否则进程由运行状态进入阻塞状态，最后调用调度函数Scheduler()

1. /\*
2. \* request - currently running process request a certain numebr of resources
3. \*/
4. oid request(**int** id, **int** n) {
5. rcb\_t \*r = get\_rcb(id);
6. **if** (r->status->u >= n) {     // allocate
7. r->status->u -= n;
8. insert\_into\_resource\_list(self->resource\_list, r, n);
9. } **else** {                    // block
10. **if** (n > r->status->k) {
11. fprintf(stderr, "request resource error\n");
12. exit(EXIT\_FAILURE);
13. }
14. self->status->type = STATUS\_BLOCKED;
15. self->status->list = &bl;
16. remove\_from\_ready\_list(&rl, self);
17. self->rl\_next = NULL;
18. insert\_into\_blocked\_list(&bl, self);
19. insert\_into\_waiting\_list(r->waiting\_list, self, n);
21. scheduler();           // schedule
22. }
    1. **释放资源**

释放资源，同样按照一类资源有多个数量的情况，在释放资源后，会按照先后顺序检查等待队列中的进程是否能被唤醒，如果进程被唤醒，将其状态由阻塞变为就绪态，从等待队列中移除，并插入到就绪队列中，最后调用调度程序Scheduler()

1. /\*
2. \* release - currently running process release a certain numebr of resources
3. \*/
4. oid release(**int** id, **int** n) {
5. rcb\_t \*r = get\_rcb(id);
7. r->status->u += n;          // deallocate
8. remove\_from\_resource\_list(self->resource\_list, r, n);
10. // try to awake blocked processes in its waiting list.
11. wa\_l\_node\_t \*tmp = \*r->waiting\_list;
12. **while** (tmp != NULL) {
13. **int** req\_num = tmp->n;
15. **if** (r->status->u >= req\_num) {      // unblock
16. pcb\_t \*p = tmp->p;
18. // turn the process from blocked to ready.
19. remove\_from\_blocked\_list(&bl, p);
20. p->bl\_next = NULL;
21. p->status->type = STATUS\_READY;
22. p->status->list = &rl;
23. insert\_into\_ready\_list(&rl, p);
25. // allocate the numebr of resource to the unblocked process
26. r->status->u -= req\_num;
27. insert\_into\_resource\_list(p->resource\_list, r, req\_num);
28. remove\_from\_waiting\_list(r->waiting\_list, p);
29. }
30. tmp = tmp->next;
31. }
33. scheduler();
34. **进程调度**

**调度策略：**

1. 设置了3个调度优先级：0、1、2（优先级高的先调度）

2. 基于优先级的抢占式调度策略，同等优先级采用时间片轮转调度策略

3. 基于函数调用来模拟时钟调度

4. 初始进程具有的功能：

1. 虚拟进程，其具有最低的优先级，永远不会被阻塞，可以保证系统中至少有一个进程能够运行

2. 作为进程树的根，方便我们遍历操作进程创建树

**调度场景：**

1. 新创建的进程的优先级或资源释放后唤醒的进程的优先级高于当前进程的优先级

2. 当前进程因为请求资源而进入阻塞状态，或因为时钟中断使得当前进程变为就绪状态

3. 当前进程被销毁，或者系统初始化时(此时系统没有运行进程)

1. /\*
2. \* scheduler
3. \*/
4. **void** scheduler() {
5. // get highest priority process
6. pcb\_t \*p;
7. **if** (self != NULL && rl->id == self->id) {   // self is in ready list
8. **if** (rl->rl\_next == NULL) **return**; // only one running process
9. **else** p = rl->rl\_next;
10. } **else** {    // self is not in ready list
11. p = rl;
12. }
13. // be preempted
14. **if** ( self == NULL ||    // called from init or destroy
15. self->status->type != STATUS\_RUNNING ||    // called from request or interrupt
16. self->priority < p->priority    // called from create or release
17. ) {
19. **if** (p != NULL) self = preempt(p, self);        // self is preempted by p
20. }
21. }
23. /\*
24. \* preempt
25. \*
26. \*      old\_pcb: running -> ready
27. \*
28. \*      new\_pcb: ready -> running
29. \*/
30. **static** pcb\_t \*preempt(pcb\_t \*new\_pcb, pcb\_t \*old\_pcb) {
32. **if** (old\_pcb != NULL && old\_pcb->status->type != STATUS\_BLOCKED) { // old\_pcb exist and not block
33. old\_pcb->status->type = STATUS\_READY;
34. remove\_from\_ready\_list(&rl, old\_pcb);           // remove old\_pcb from rl
35. old\_pcb->rl\_next = NULL;
36. insert\_into\_ready\_list(&rl, old\_pcb);           // insert old\_pcb into the end of corrsponding priority level in rl
37. }
39. new\_pcb->status->type = STATUS\_RUNNING;
40. remove\_from\_ready\_list(&rl, new\_pcb);           // remove new\_pcb from rl
41. new\_pcb->rl\_next = NULL;
42. insert\_into\_head\_of\_ready\_list(&rl, new\_pcb);   // insert new\_pcb at the head of rl, indicate that the new process is running
44. **return** new\_pcb;
45. }

**时钟中断：**

模拟时间片到，或者外部硬件中断，将当前进程由运行态变为就绪态，转而调用调度程序scheduler()

1. /\*
2. \* interrupt - emulate the time-out interrupt
3. \*       currently running process:   running -> ready
4. \*/
5. **void** interrupt() {
6. self->status->type = STATUS\_READY;
7. scheduler();
8. }
9. **Shell设计**

可以对进程与资源管理器使用的命令有：

- `init`: 初始化进程与资源管理器

- `cr <process name> <priority>`：按照指定名称和优先级创建进程

- `de <process name>`: 删除指定名称的进程

- `req <resource> <# of units>`: 当前进程请求对应数量(如果不填默认为1)的对应资源

- `rel <resource name> <# of units>`: 当前进程释放对应数量(如果不填默认为1)的对应资源

- `to`: 模拟时钟中断

- `list ready`: 列出所有就绪队列中的进程

- `list block`: 列出所有阻塞队列中的进程

- `list res`: 列出所有可用资源信息

- `pr <process name>`: 列出指定名称的进程的PCB信息

ps: `<# of units>` is optional and the default is `1`.

测试程序的主函数：

1. **int** main(**int** argc, **char** \*argv[]) {
2. // handle input stream
3. **if** (argc == 1) {
4. // from stdin 从用户中断读取命令
5. in = stdin;
6. } **else** **if** (argc == 2) {
7. // from file 从输入文件读取命令
8. **if** ((in = fopen(argv[1], "r")) == NULL) {
9. fprintf(stderr, "open file error\n");
10. exit(EXIT\_FAILURE);
11. }
12. } **else** {
13. fprintf(stderr, "Usage: ./shell <input file>(optional)\n");
14. exit(EXIT\_FAILURE);
15. }
16. // run command loop 循环不停的读取命令
17. run\_loop();
19. exit(EXIT\_SUCCESS);
20. }

Shell的主体结构：

1. /\*
2. \* run\_loop - the structure of the working shell.
3. \*/
4. **static** **void** run\_loop() {
5. **char** \*line;
6. **char** \*\*args;
7. **while** (1) {
8. **if** (in == stdin)
9. printf("shell> ");
10. // read line 读取一行字符串
11. line = read\_line();
12. **if** (line == NULL) **continue**;
14. // split line 将字符串分解出tokens数组
15. args = split\_line(line);
16. **if** (args == NULL) **continue**;
18. // execute command 传入命令参数 执行指定命令
19. execute\_cmd(args);
20. free(line);
21. free(args);
22. }
23. }
25. /\*
26. \* read\_line - read a line at once from input stream
27. \*/
28. **static** **char** \*read\_line() {
29. **char** \*line = NULL;
30. **size\_t** bufsize = 0;
31. ssize\_t len = 0;
32. **if** ( (len = getline(&line, &bufsize, in)) == -1) {
33. **if** (feof(in)) {
34. exit(EXIT\_SUCCESS);
35. } **else** {
36. fprintf(stderr, "read line error\n");
37. exit(EXIT\_FAILURE);
38. }
39. }
40. **if** (line == NULL || len == 1) **return** NULL;
41. **else** **if** (line[len - 1] == '\n') line[len - 1] = '\0';
42. **return** line;
43. }
44. /\*
45. \* split\_line - split line to tokens which contains arguments of command.
46. \*/
47. **static** **char** \*\*split\_line(**char** \*line) {
48. **char** \*stringp = line;
49. **int** position = 0;
50. **char** \*\*tokens = malloc(BUFSIZE \* **sizeof**(**char** \*));
51. **char** \*token;
52. **while** ( (token = strsep(&stringp, " ")) != NULL ) {
53. tokens[position++] = token;
54. }
55. tokens[position] = NULL;
56. **return** tokens;
57. }

选择命令的函数：

1. /\*
2. \* execute\_cmd - used for invoke manager routines and print some information.
3. \*/
4. tatic **void** execute\_cmd(**char** \*\*args) {
5. **if** (args[0] == NULL) **return**;
6. **if** (strcmp(args[0], "init") == 0) {
7. **char** \*\*tmp = malloc(BUFSIZE \* **sizeof**(**char** \*));
8. it(tmp);
9. } **else** **if** (strcmp(args[0], "cr") == 0) {
10. cr(args);
11. } **else** **if** (strcmp(args[0], "de") == 0) {
12. de(args);
13. } **else** **if** (strcmp(args[0], "req") == 0) {
14. req(args);
15. } **else** **if** (strcmp(args[0], "rel") == 0) {
16. rel(args);
17. } **else** **if** (strcmp(args[0], "to") == 0) {
18. to(args);
19. } **else** **if** (strcmp(args[0], "list") == 0) {
20. list(args);
21. } **else** **if** (strcmp(args[0], "pr") == 0) {
22. pr(args);
23. } **else** **if** (strcmp(args[0], "exit") == 0) {
24. exit(EXIT\_SUCCESS);
25. } **else** **if** (strcmp(args[0], "help") == 0) {
26. help();
27. } **else** {
28. printf("can't find %s in commands library of the shell, try type in \"help\"\n", args[0]);
29. }

实际处理命令的函数：

1. // functions for executing command
2. **static** **void** it(**char** \*\*);
3. **static** **void** cr(**char** \*\*);
4. **static** **void** de(**char** \*\*);
5. **static** **void** req(**char** \*\*);
6. **static** **void** rel(**char** \*\*);
7. **static** **void** to(**char** \*\*);
8. **static** **void** list(**char** \*\*);
9. **static** **void** pr(**char** \*\*);
10. **static** **void** help();

具体实现在附件中

1. **系统初始化设计**

1. 拥有一个虚拟进程：Init 进程

2. 对应三个优先级的就绪队列的初始化

3. 四类资源R1、R2、R3、R4，分别对应有数量i个

**九、实验数据及结果分析：**

实验数据及结果分析要求:

经过对比与验证，该输出结果与实验指导书中给出的预期输出结果是一致的，说明实验成功。

1. **实验结论：**

通过本次实验，进一步巩固了课堂所学的进程管理和资源管理的相关知识，并使用C语言简单模拟了计算机操作系统对进程和资源的管理和调度，成功实现了时间片轮转（RR）调度算法，并且系统能顺利通过测试用例，取得与预期一致的结果，说明实现取得成功。

1. **总结及心得体会：**

通过本次实验，加深了对课堂所学的进程管理和资源管理相关知识的理解，对进一步理解计算机系统与底层实现原理是一次非常好的指引

**十二、对本实验过程及方法、手段的改进建议：**

在这个最基础的进程和资源管理器之上，添加其他调度算法的实现，如SJF、HRRN等调度算法

**报告评分：**

**指导教师签字：**

**电 子 科 技 大 学**

**实 验 报 告**

**学生姓名：唐以恒 学 号：2017110801017 指导教师：刘杰彦**

**实验地点：主楼A2-412 实验时间：2020年5月**

**一、实验室名称：计算机实验室**

**二、实验项目名称：进程与资源管理实验**

**三、实验学时：6**

**四、实验原理：**

### **1.逻辑地址到线性地址的转换**

逻辑地址：Intel中，一个逻辑地址由一个段标识符，加上一个指定段内相对地址的偏移量。

段标识符: 也称为段选择符，属于逻辑地址的构成部分，段标识符是由一个 16 位长的字段组成，其中前 13 位是一个索引号。后面 3 位包含一些硬件细节

索引号：可以看作是段的编号，也可以看做是相关段描述符在段表中的索引位置。

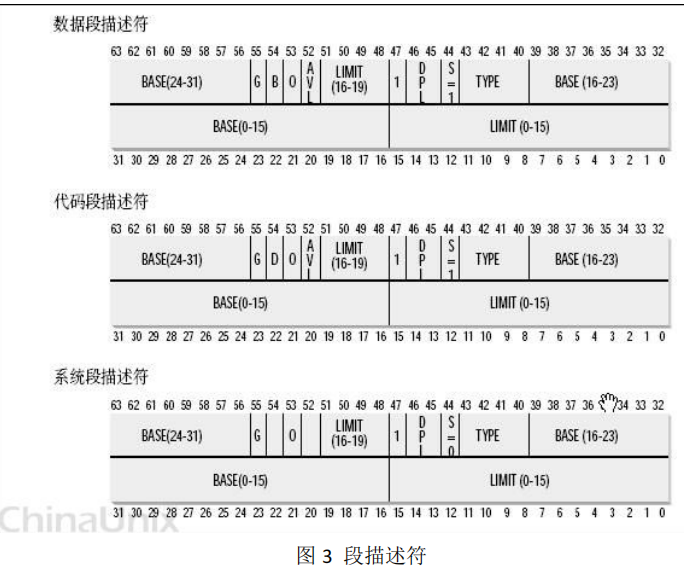
系统中的段表有两类：GDT 和 LDT

GDT：全局段描述符表，整个系统一个，GDT 表中存放了共享段的描述符，以及 LDT 的描述符（每个 LDT 本身被看作一个段）

LDT：局部段描述符表，每个进程一个，进程内部的各个段的描述符，就放在 LDT 中。

TI 字段：TI=0，表示相应的段描述符在 GDT 中，TI=1 表示表示相应的段描述符在 LDT 中。

段描述符(即段表项)：具体描述了一个段。在段表中，存放了很多段描述符。我们可以通过段标识符的前 13 位，直接在段描述符表中找到一个具体的段描述符，也就是说，段标识符的前 13 位是相关段描述符在段表中的索引位置。

****

Base 字段：它描述了一个段的开始位置：段基址。Base(24-31)：基地址的高 8 位，Base(16-23)：基地址的中间 8 位，Base(0-15)：基地址的低 16 位。

相关寄存器：

GDTR：存放 GDT 在内存中的起始地址和大小

LDTR：分两种情况：

(1) 当段选择符中的 TI=1 时，表示段描述符存放在 LDT 中，如何找到 LDT 呢，LDT本身也被看作一个段，LDT 的起始地址存放在 GDT 中，此时 LDTR 存放的就是LDT 在 GDT 中的索引。这也是本实验关注的情况。

(2) 当段选择符中的 TI=0 时，表示段描述符存放在 GDT 中，通过 GDTR 找到 GDT， 当 TI=0 时，不涉及对 LDT 和 LDTR 的使用。

段选择符：如在 DS，SS 等寄存器内存储，取高 13 位作为在相应段表（如上例中的 DS 的高 13 位为对应段在 LDT）中的索引。

线性地址： 段标识符用来标明一个段的编号，具体的，我们需要通过段的编号，查找段表，来获得这个段的起始地址，即段基址。如前所述，这里的段基址，不是相应的段在内存中的起始地址，而是程序编译链接以后，这个段在逻辑

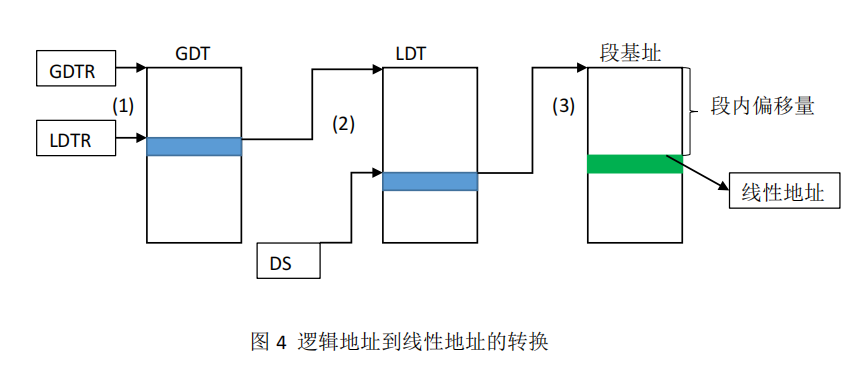
地址空间里的起始位置。进一步的，段基地址＋段内偏移量，就得到线性地址（即要访问的数据在整个程序逻辑(虚拟)地址空间中的位置）。

从逻辑地址到线性地址的转换过程：此时从段选择符 DS 中分离出段索引号（高 13 位）和 TI 字段，TI=1，表明段描述符存放在 LDT 中：

（1）从 GDTR 中获得 GDT 的地址，从 LDTR 中获得 LDT 在 GDT 中的偏移量，查找 GDT，从中获取 LDT 的起始地址；

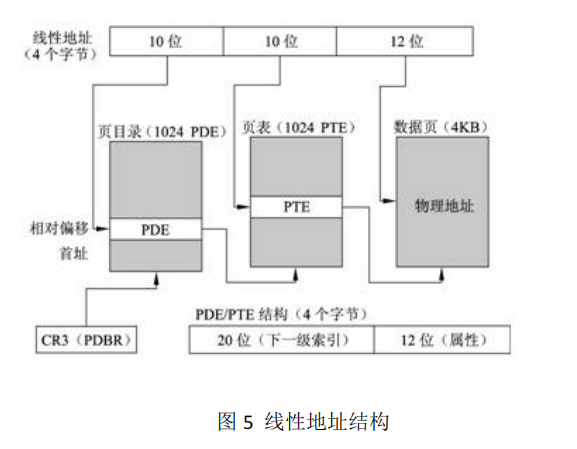
（2）从 DS 中的高 13 位获取 DS 段在 LDT 中索引位置，查找 LDT，获取 DS 段的段描述符，从而获取 DS 段的基地址；

（3）根据 DS 段的基地址＋段内偏移量，获取所需单元的线性地址。



### **2.线性地址到物理地址的转换**

物理地址：分段是面向用户，而分页则是面向系统，以提高内存的利用率，简言之，内存空间是按照分页来管理的。一个 32 位的机器，支持的内存空间是 4G，在页面大小为 4KB 的情况下，如果采用二级分页管理方式，线性地址结构如图所示：

****

每一个 32 位的线性地址被划分为三部份， 页目录索引(10 位)：页表索引(10 位)：偏移(12 位，因为页面大小为 4K)。最终，我们需要根据线性地址，来获得物理地址。

将线性地址转换成物理地址的步骤：

(1)、因为页目录表的地址放在 CPU 的 cr3 寄存器中，因此首先从 cr3 中取出进程的页目录表（第一级页表）的起始地址（操作系统负责在调度进程的时候，已经把这个地址装入对应寄存器）；

(2)、根据线性地址前十位，在页目录表（第一级页表）中，找到对应的索引项，因为引入了二级管理模式，线性地址的前十位，是第一级页表中的索引值，根据该索引，查找页目录表中对应的项，该项即保存了一个第二级页表的起始地址。

(3)、查找第二级页表，根据线性地址的中间十位，在该页表中找到数据页的起始地址；

(4)、将页的起始地址与页内偏移量（即线性地址中最后 12 位）相加，得到最终我们想要的物理地址；

**五、实验目的：**

本实验运行了一个设置了全局变量的的循环程序，通过查看段寄存器，LDT表，GDT表等信息，经过一系列段、页式地址转换，找到程序中该全局变量的物理地址

1. **实验内容：**

通过手工查看系统内存，并修改特定物理内存的值，实现控制程序运行的目的。

**七、实验器材（设备、元器件）：**

操作系统Ubuntu 19.02

**八、实验步骤：**

实验步骤包括:

1、点击 bochs.exe 安装 bochs。

2、拷贝 bootimage-0.11-hd 、 diska.img 、 hdc-0.11-new.img 、mybochsrc-hd.bxrc 至安装目录。

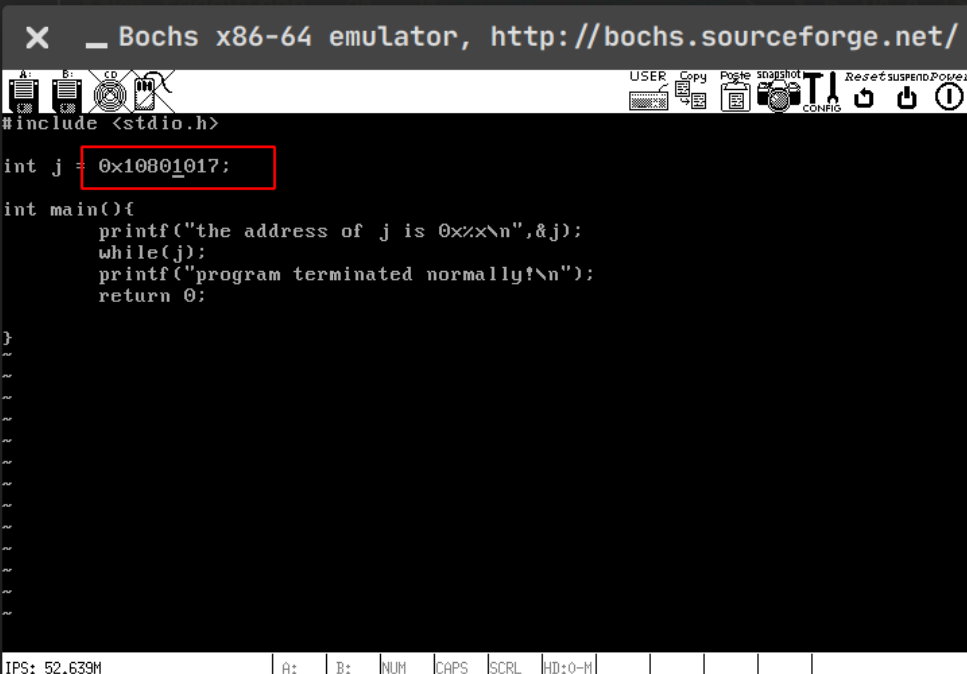
3、在安装目录中找到 bochsdbg.exe 程序，并运行。

4、在弹出的界面中，点击“Load”加载配置文件“mybochsrc-hd.bxrc”。随后，点击“Start”启动 Bochs 虚拟机。

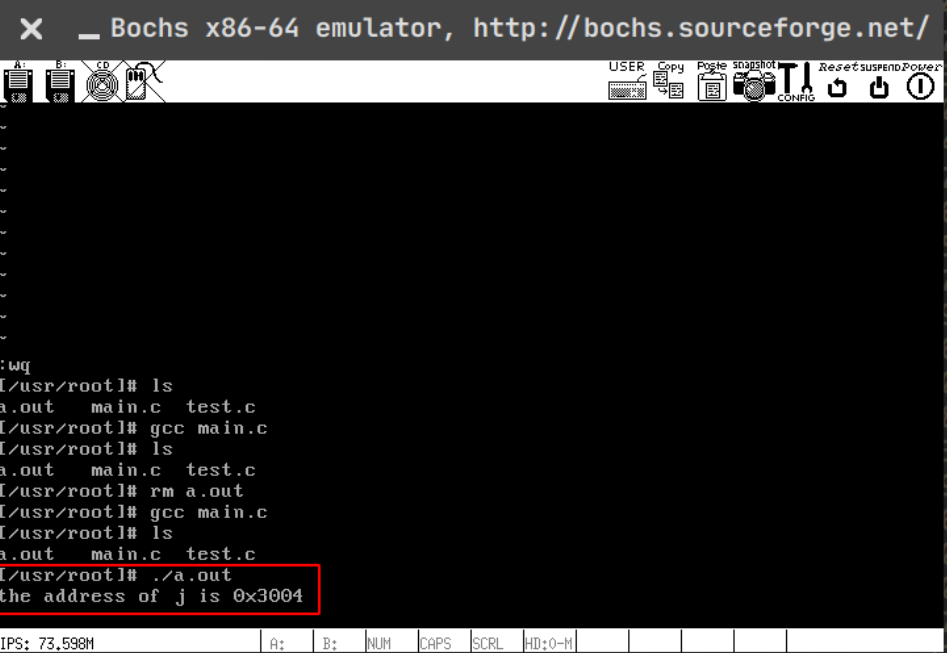
5、虚拟机启动后，出现两个窗口，一个为 Bochs 控制窗口，另一个为Linux 操作系统运行窗口（主显示窗口）。

6、在控制窗口输入“c”后回车，加载 Linux 操作系统。

7、使用 vi 工具编写 mytest.c 源文件。随后执行“gcc -o mytest mytest.c”命令编译并生成“mytest”可执行文件。



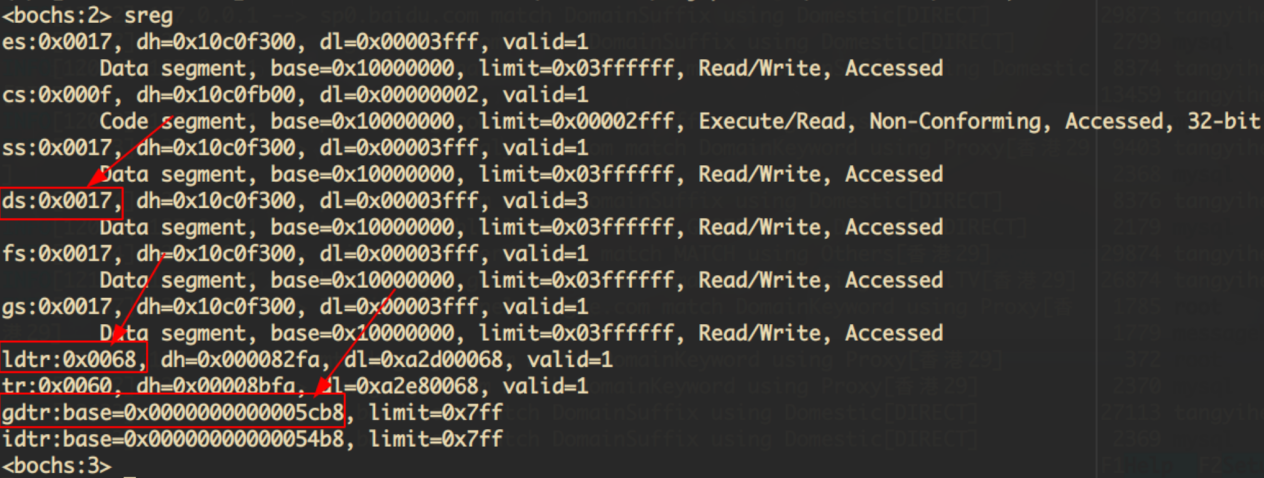
8、在 Linux 操作系统中，运行“./mytest”可执行文件



9、控制窗口按 Ctrl+c，进入调试状态。

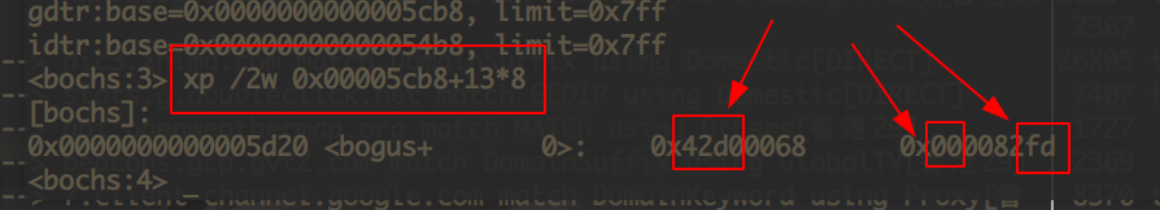
10、在控制窗口中输入 sreg 命令，查看段的具体信息。

可以看到 ds 段的段标识符信息是 0x0017（0000000000010111）,对应 TI=1，表明段描述符在 LDT 中，右移 3 位之后为 0x02，即表示在局部描述符表 LDT 的偏移量为 2。

****

11、查看 LDTR 寄存器，其中存放了 LDT 在 GDT 的位置。0x0068 对应 TI=0，右移 3 位之后为 0x0D，即在 GTD 中的索引为(0x0D) 13。

12、gdtr 存放了 GDT 的起始地址，用 xp /2w 0x00005cb8+13\*8（每个描述符占 8 个字节）查看 GDT 中对应表项，得到的 LDT 段描述符，从而我们可以得到 LDT 的基址为 0x00fd42d0。

****

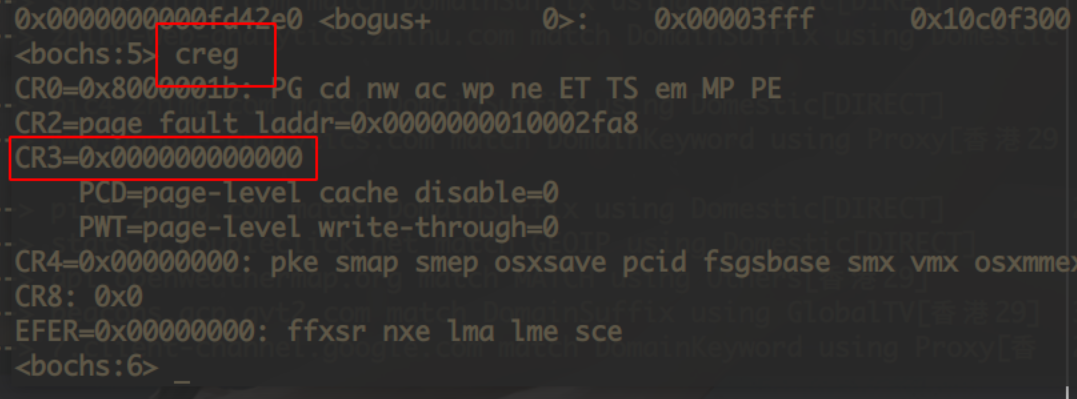
1. 用 xp /2w 0x00fd42d0+2\*8，查看 LDT 中第 2 项段描述符（即 ds 段的描述符信息，应与 ds 寄存器（dl、dh）中的数值完全相同



14、计算出 ds 段的基地址为 0x10000000（与用 sreg 得到信息一致）。

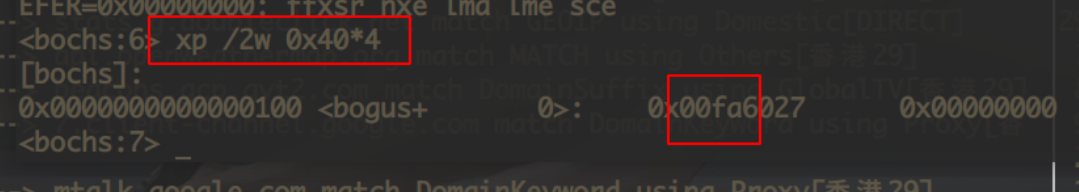
15、计算线性地址 0x10000000+0x3004=0x10003004， 将其用 0 补满 32 位(0001 0000 0000 0000 0011 0000 0000 0100)，然后按照 10-10-12 比特的方式划分，为 0x40-0x03-0x04。即第一级页表内的索引为 0x40，第二级页表内的索引为 0x03，页内偏移为 0x04。

16、使用 creg 查看寄存器 CR3 值为 0，即页目录表（第一级页表）的起始地址为 0

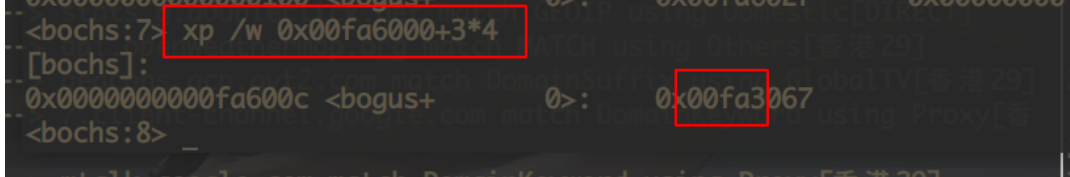


17、使用 xp /w 64\*4 查看 PDE 为 0x00fa6027,下一级索引为 0x00fa6000。

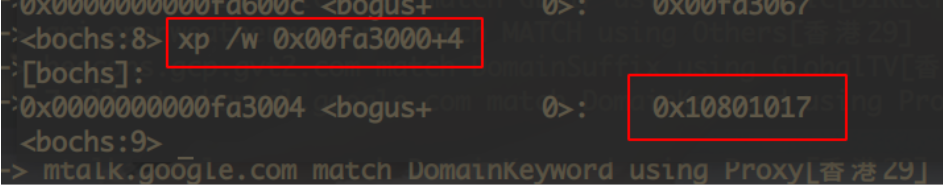
（或执行 xp /2w 0x40\*4）



18、使用 xp /w 0x00fa6000+3\*4 查看 0x00fa3067,下一级索引为 0x00fa3000，得到物理地址为 0x00fa6000+4。

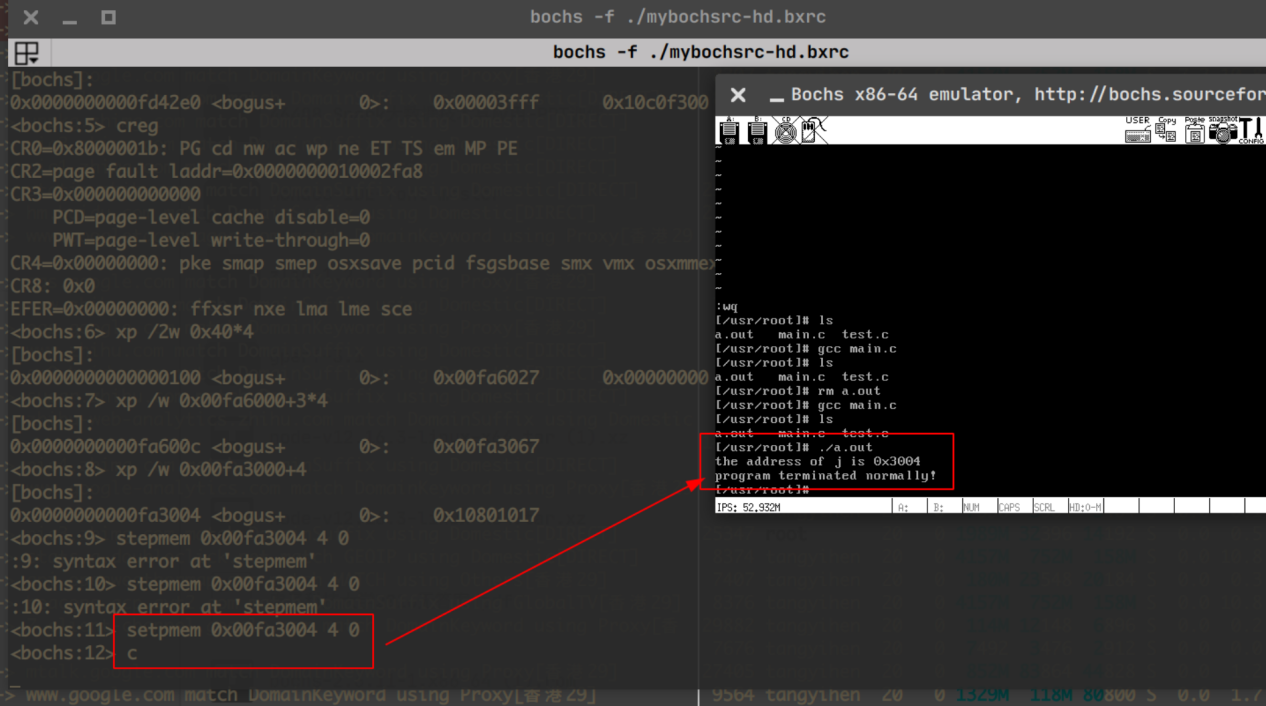


19、使用 xp /w 0x00fa3000+4,内容为 0x10801017 与我们所设的值相同。



20、使用 setpmem 0x00fa6004 4 0,设置 0x00fa6004 开始的四个字节均为 0，并检查是否成功。

21、成功之后输入 c 继续运行，显示程序正常结束。



1. **实验结论：**

通过本次实验，详细学习了计算机的段页式内存管理机制，掌握了地址转换的过程，并且在实际操作中，能成功寻找到变量存储的具体位置，并对变量的数值成功进行修改，程序能正常运行退出，实验取得成功。

1. **总结及心得体会：**

接触到了 Linux 操作系统内核，对计算机的段页式内存管理机制以及其中的地址转换有了更为深入的理解

**十一、对本实验过程及方法、手段的改进建议：**

无

**报告评分：**

**指导教师签字：**