OS第四次实验文档

181830249 张卓楠

以下为本次实验修改的代码说明（仅为部分主要修改内容）：

系统调用

概述

逻辑：在syscall.asm中添加系统调用入口，陷入中断，传递参数给子例程或内核函数

- **proto.h** 头文件中声明增加的函数
- **global.c** 中添加 `sys_call_table` 作为通过 `kernel.asm->sys_call()` 访问对应系统调用函数的索引
- **syscall.asm** 中添加系统调用入口及部分函数，注意添加相应的导入导出符号以及常量
- **proc.c** 将进程相关的系统调用函数添加在此文件中
- **clock.c** 修改时钟中断处理逻辑

syscall.asm

```
; ========================================================================
;                                  my_milli_seconds xx毫秒内不被分配时间片
; ========================================================================
```

```
4  my_milli_seconds:
5      mov eax, _NR_my_milli_seconds
6      push ebx                        ; 先将ebx内容压栈
7      mov ebx, [esp+8]                ; 此时的栈：函数参数（从右到左）->返回地
   址->ebx[栈顶在这]
8      int INT_VECTOR_SYS_CALL         ; 系统中断门，此时的ebx中保存有第一个参
   数
9      pop ebx                         ; 恢复ebx值
10     ret                             ; 函数调用返回
11
12 sys_my_milli_seconds:
13     push ebx                        ; 将ebx保存的第一个参数压入栈顶
14     call milli_sleep                ; 函数调用，会访问压入的参数
15     pop ebx                         ; ebx首先出栈，注意push和pop一一对应
16     ret
17
18 ; ================================================================
   ==
19 ;                                 my_disp_str 打印字符串，接受char*参数
20 ; ================================================================
   ==
21 my_disp_str:
22     mov eax, _NR_my_disp_str
23     push ebx                        ; 先将ebx内容压栈
24     push ecx
25     mov ebx, [esp+12]               ; 此时的栈：函数参数（从右到左）->返回地
   址->ebx->ecx[栈顶在这]
26     mov ecx, [esp+16]
27     int INT_VECTOR_SYS_CALL         ; 系统中断门，此时的ebx中保存有参数起始
   位置
28     pop ecx
29     pop ebx                         ; 恢复ebx值
30     ret                             ; 函数调用返回
31
32 sys_my_disp_str:
33     pusha                           ; 调用前保存所有寄存器值
34     push ecx                        ; 将ebx保存的第一个参数压入栈顶
35     push ebx                    ; 将ebx保存的第一个参数压入栈顶
36     call disp_color_str             ; 函数调用，会访问压入的参数
37     pop ebx                         ; ebx首先出栈，注意push和pop一一对应
```

```
    pop ecx
    popa                            ; 调用后恢复所有寄存器值
    ret

; ===========================================================================
;                                 my_p_opera 信号量P操作，S--，占据资源
; ===========================================================================
my_p_opera:
    mov eax, _NR_my_p_opera
    push ebx                        ; 先将ebx内容压栈
    mov ebx, [esp+8]                ; 此时的栈：函数参数（从右到左）->返回地址->ebx[栈顶在这]
    int INT_VECTOR_SYS_CALL         ; 系统中断门，此时的ebx中保存有第一个参数
    pop ebx                         ; 恢复ebx值
    ret                             ; 函数调用返回

sys_my_p_opera:
    push ebx                        ; 将ebx保存的第一个参数压入栈顶
    call p_semaphore                ; 函数调用，会访问压入的参数
    pop ebx                         ; ebx首先出栈，注意push和pop一一对应
    ret

; ===========================================================================
;                                 my_v_opera 信号量V操作，S++，释放资源
; ===========================================================================
my_v_opera:
    mov eax, _NR_my_v_opera
    push ebx                        ; 先将ebx内容压栈
    mov ebx, [esp+8]                ; 此时的栈：函数参数（从右到左）->返回地址->ebx[栈顶在这]
    int INT_VECTOR_SYS_CALL         ; 系统中断门，此时的ebx中保存有第一个参数
    pop ebx                         ; 恢复ebx值
    ret                             ; 函数调用返回

```

```
sys_my_v_opera:
    push ebx                        ; 将ebx保存的第一个参数压入栈顶
    call v_semaphore                ; 函数调用，会访问压入的参数
    pop ebx                         ; ebx首先出栈，注意push和pop一一对应
    ret
```

proc.c

```
/*======================================================================*
                                schedule  进程调度函数
 *======================================================================*/
PUBLIC void schedule()
{
    //时间片轮转调度，如果下一进程睡眠或者处于阻塞状态就跳过，向后轮询
    while (1){
        int t = get_ticks();    //当前总中断次数
        p_proc_ready++;         //切换至下一个进程
        //如果进程超出则指向进程表的第一个
        if (p_proc_ready >= proc_table + NR_TASKS)p_proc_ready = proc_table;
        //判断进程是否处于阻塞状态或睡眠状态
        if (p_proc_ready->pos == 0 && p_proc_ready->wake_ticks <= t) break;
    }
}
/*======================================================================*
                    sys_get_ticks 通过系统调用访问时钟中断次数
 *======================================================================*/
PUBLIC int sys_get_ticks()
{
    return ticks;
}

/*======================================================================
```

```
=======*
                    milli_sleep xx毫秒内不被分配时间片
 *=================================================================
=======*/
PUBLIC void milli_sleep(int milli_sec){
        //设置当前进程的wake_ticks
        p_proc_ready->wake_ticks=(milli_sec * HZ /1000) + get_tic
ks();
        //当前进程休眠，调度下一进程
        schedule();
}


/*=================================================================
=======*
                            p_semaphore 信号量P操作，S--，占据资源
 *=================================================================
=======*/
PUBLIC void p_semaphore(SEMAPHORE* semaphore){
    //信号量--，占用一份资源
    semaphore->value--;
    //如果信号量小于0，则发生阻塞，将当前进程放入临界区
    if (semaphore->value < 0) {
        p_proc_ready->pos = 1;  //处于阻塞状态
        semaphore->list[semaphore->queue_end] = p_proc_ready;
        semaphore->queue_end = (semaphore->queue_end + 1) % SEMAP
HORE_LIST_SIZE;    //更新临界区尾部
        schedule();     //轮询下一进程
    }

}

/*=================================================================
=======*
                            v_semaphore 信号量V操作，S++，释放资源
 *=================================================================
=======*/
PUBLIC void v_semaphore(SEMAPHORE* semaphore){
    //信号量++，释放一份资源
    semaphore->value++;
```

```
    //如果信号量不大于0，说明临界区不为空，释放一个进程
    if (semaphore->value <= 0) {
        PROCESS* p=semaphore->list[semaphore->queue_start];
        p->pos = 0;
        semaphore->queue_start = (semaphore->queue_start + 1) % SEMAPHORE_LIST_SIZE;    //更新临界区头
    }
}
```

clock.c

```
/*======================================================================*
                           clock_handler    时钟中断处理程序
 *======================================================================*/
PUBLIC void clock_handler(int irq)
{
    //每次时钟中断时监测屏幕是否占满，刷新屏幕
    if(disp_pos>80*50)clear_screen();

    ticks++;        //全局时钟中断次数++
    // p_proc_ready->ticks--;

    if (k_reenter != 0) {   //发生了中断重入，直接返回
        return;
    }

    // if (p_proc_ready->ticks > 0) {       //如果当前进程ticks还没变成0，其他进程就不能进入！
    //  return;
    // }

    schedule();

}
```

PV 操作

概述

逻辑：添加信号量结构，通过进程中相应的PV操作实现读者写者问题

- **proc.h** 修改进程结构及数量，添加wake_ticks和pos用于唤醒/PV操作，定义信号量结构
- **main.c** 添加全局信号量，添加读者/写者进程及相关操作
- **global.c** 添加任务表 `task_table`

proc.h

```
1  //作为进程表中存放寄存器值的结构
2  typedef struct s_stackframe {/* proc_ptr points here    ↑ Low
   */
3      u32 gs;         /* ┓                                 │
   */
4      u32 fs;         /* ┃                                 │
   */
5      u32 es;         /* ┃                                 │
   */
6      u32 ds;         /* ┃                                 │
   */
7      u32 edi;        /* ┃                                 │
   */
8      u32 esi;        /* ┣ pushed by save() 中断处理程序     │
   */
9      u32 ebp;        /* ┃                                 │
   */
10     u32 kernel_esp; /* <- 'popad' will ignore it        │
   */
11     u32 ebx;        /* ┃                                 ↑栈从高地址
   往低地址增长*/
12     u32 edx;        /* ┃                                 │
   */
13     u32 ecx;        /* ┃                                 │
   */
14     u32 eax;        /* ┛                                 │
   */
15     u32 retaddr;    /* return address for assembly code save()  │
   */
```

```
    u32 eip;        /*  ┐                                   │
*/
    u32 cs;         /*  │                                   │
*/
    u32 eflags;     /*  ├ 由CPU压栈，在ring0->ring1时       │
 */
    u32 esp;        /*  │                                   │
*/
    u32 ss;         /*  ┘                                   ┴High
*/
}STACK_FRAME;

//进程表结构，开头的regs在栈结构中存放相关寄存器值
typedef struct s_proc {
    STACK_FRAME regs;          /* 进程寄存器保存在stack frame中 */

    u16 ldt_sel;               /* GDT的段选择子：LDT的基址和段界限，gd
t selector giving ldt base and limit */
    DESCRIPTOR ldts[LDT_SIZE]; /* LDT局部描述符表，存放进程私有的段描述
符们 */

        int ticks;                 /* 可用的中断次数，每次发生中断就递减
*/
        //int priority;            /* 优先级，固定的值，当所有ticks都
变为0后，在把各自的优先数赋值给各自的ticks */
        int wake_ticks;            /* 用于定时唤醒，表示预期的醒来时间
 */

        int pos;                /* 进程状态，0-非阻塞，1-阻塞 */

    u32 pid;                   /* MM内存中传递的进程id */
    char p_name[16];           /* 进程名字 */
}PROCESS;

//任务结构
typedef struct s_task {
    task_f  initial_eip;        //进程起始地址
    int stacksize;              //堆栈大小
    char    name[32];
}TASK;
```

```
46
47 #define SEMAPHORE_LIST_SIZE 32
48 //信号量定义，包含可用资源值、一个临界区数组和头尾索引
49 typedef struct s_semaphore
50 {
51     int value;
52     PROCESS *list[SEMAPHORE_LIST_SIZE];
53     int queue_start;
54     int queue_end;
55 }SEMAPHORE;
56
57
58 /* Number of tasks  任务数量 */
59 #define NR_TASKS    6
60
61 /* stacks of tasks */
62 // #define STACK_SIZE_TESTA 0x8000
63 // #define STACK_SIZE_TESTB 0x8000
64 // #define STACK_SIZE_TESTC 0x8000
65 //六个进程定义
66 #define STACK_SIZE_A    0x8000
67 #define STACK_SIZE_B    0x8000
68 #define STACK_SIZE_C    0x8000
69 #define STACK_SIZE_D    0x8000
70 #define STACK_SIZE_E    0x8000
71 #define STACK_SIZE_F    0x8000
72
73 #define STACK_SIZE_TOTAL    (   STACK_SIZE_A + \
74                                 STACK_SIZE_B + \
75                                 STACK_SIZE_C + \
76                                 STACK_SIZE_D + \
77                                 STACK_SIZE_E + \
78                                 STACK_SIZE_F)
```

global.c

```
1 PUBLIC  TASK    task_table[NR_TASKS] = {
2                     {A_Reader, STACK_SIZE_A, "A_Reader"},     //任
```

```
务表数组
                    {B_Reader, STACK_SIZE_B, "B_Reader"},
                    {C_Reader, STACK_SIZE_C, "C_Reader"},
                    {D_Writer, STACK_SIZE_D, "D_Writer"},
                    {E_Writer, STACK_SIZE_E, "E_Writer"},
                    {F_Process, STACK_SIZE_F, "F_Process"},
                    };
```

main.c

```
/*============================================================================*
                                PV操作相关信号量全局变量、初始化函数
 *============================================================================*/
SEMAPHORE x,y,z;
SEMAPHORE rmutex,wmutex;
SEMAPHORE max_reader;
int readcount,writercount;
int priority;  //优先级：0-读者优先，1-写者优先
init_semophore(SEMAPHORE* semaphore,int value){
    semaphore->value=value;
    semaphore->queue_start = semaphore->queue_end =0;
}

int time_piece=1000;
int reader_num=0;       //记录读者数量

/*============================================================================*
                                kernel_main
 *============================================================================*/
PUBLIC int kernel_main()
{
    //作为最后一部分被执行代码
    disp_str("-----\"kernel_main\" begins-----\n");

```

```
25      TASK*       p_task      = task_table;   //任务表
26      PROCESS*    p_proc      = proc_table;   //进程表
27      char*       p_task_stack    = task_stack + STACK_SIZE_TOTAL;
//任务栈
28      u16     selector_ldt    = SELECTOR_LDT_FIRST;   //LDTR 16位,
对应GDT中LDT描述符的段选择子
29      int i;
30      for (i = 0; i < NR_TASKS; i++) {        //初始化每一个进程
31          strcpy(p_proc->p_name, p_task->name);   // name of the p
rocess
32          p_proc->pid = i;                        // pid
33
34          p_proc->ldt_sel = selector_ldt;         // LDTR
35
36          //LDT包含两个描述符，分别初始化为内核代码段和内核数据段
37          memcpy(&p_proc->ldts[0], &gdt[SELECTOR_KERNEL_CS >> 3],s
izeof(DESCRIPTOR));
38          p_proc->ldts[0].attr1 = DA_C | PRIVILEGE_TASK << 5;
//改变DPL优先级
39          memcpy(&p_proc->ldts[1], &gdt[SELECTOR_KERNEL_DS >> 3],s
izeof(DESCRIPTOR));
40          p_proc->ldts[1].attr1 = DA_DRW | PRIVILEGE_TASK << 5;
//改变DPL优先级
41
42          //寄存器初始化，除了cs指向LDT中第一个描述符，ds、es、fs、ss都指向L
DT中第二个描述符，gs指向显存只是RPL变化
43          p_proc->regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) |
SA_TIL | RPL_TASK;
44          p_proc->regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
SA_TIL | RPL_TASK;
45          p_proc->regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
SA_TIL | RPL_TASK;
46          p_proc->regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
SA_TIL | RPL_TASK;
47          p_proc->regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) |
SA_TIL | RPL_TASK;
48          p_proc->regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | R
PL_TASK;
49          p_proc->regs.eip = (u32)p_task->initial_eip;
50          p_proc->regs.esp = (u32)p_task_stack;
```

```
51          p_proc->regs.eflags = 0x1202; /* IF=1, IOPL=1 */
52
53          //堆栈从高到低生长
54          p_task_stack -= p_task->stacksize;
55          p_proc++;
56          p_task++;
57          selector_ldt += 1 << 3;
58      }
59
60      //初始化各进程的优先级，由于是循环读写模式，区分优先级，所以不使用进程优先级
61      // proc_table[0].ticks = proc_table[0].priority =  15;
62      // proc_table[1].ticks = proc_table[1].priority =  5;
63      // proc_table[2].ticks = proc_table[2].priority =  3;
64
65      k_reenter = 0;  //全局中断嵌套次数，因为restart中自减了该变量，所以初始化为0
66      ticks = 0;      //全局时钟中断次数
67
68      //初始化各信号量
69      init_semophore(&x,1);
70      init_semophore(&y,1);
71      init_semophore(&z,1);
72      init_semophore(&rmutex,1);
73      init_semophore(&wmutex,1);
74      init_semophore(&max_reader,3);   //同时读书的最大读者数量，1、2、3
75      readcount=0;
76      writercount=0;
77      priority=1;             //优先级：0-读者优先，1-写者优先
78
79      //当前进程赋值
80      p_proc_ready    = proc_table;
81
82          /* 初始化 8253 PIT */
83          out_byte(TIMER_MODE, RATE_GENERATOR);
84          out_byte(TIMER0, (u8) (TIMER_FREQ/HZ) );
85          out_byte(TIMER0, (u8) ((TIMER_FREQ/HZ) >> 8));  // 中断 10ms发生一次
86
```

```
87          put_irq_handler(CLOCK_IRQ, clock_handler); /* 设定时钟中断处理程序 */
88          enable_irq(CLOCK_IRQ);                     /* 让8259A可以接收时钟中断 */
89
90      clear_screen(); //清屏函数调用
91      milli_delay(2000);      //防止清除输出文本
92
93      restart();      //进程从ring0-ring1
94
95      while(1){}
96  }
97
98  /*==========================================================================*
99                                      六个进程
100             A、B、C为读者进程，D、E为写者进程，F 为普通进程
101                 * A阅读消耗2个时间片
102                 * B、C阅读消耗3个时间片
103                 * D写消耗3个时间片
104                 * E写消耗4个时间片
105  *==========================================================================*/
106
107 void A_Reader(){Reader("A",0x09,2);}
108
109 void B_Reader(){Reader("B",0x0A,3);}
110
111 void C_Reader(){Reader("C",0x0C,3);}
112
113 void D_Writer(){Writer("D",0x0D,3);}
114
115 void E_Writer(){Writer("E",0x0E,4);}
116
117 void F_Process(){
118     while(1){
119         my_disp_str("Report:",0x0F);
120         if(readcount>0){
121             char ch[2];
122             ch[0]=reader_num+'0';       //readcount是运行进程数+阻
```

```
塞进程数
123             ch[1]='\0';
124             my_disp_str(ch,0x0F);
125             my_disp_str(" Read ;   ",0x0F);
126         }
127         if(readcount==0 && writercount>0){          //只有当readc
ount==0时wmutex才会被打开
128             my_disp_str("Write ;   ",0x0F);
129         }
130         my_milli_seconds(time_piece);
131     }
132 }
133
134 /*================================================================
========*
135                                   读者进程
136  *================================================================
========*/
137 PUBLIC void Reader(char* name,int color,int proc_ticks){
138     if(priority==0){        //读者优先
139         while(1){
140                 //开始读
141                 my_disp_str(name,color);
142                 my_disp_str(" come for reading!   ",color);
143
144
145             my_p_opera(&rmutex);
146             if(readcount==0)my_p_opera(&wmutex);    //如果之前没有
人在读，就将wmutex--
147             readcount++;                //读者数++
148             my_v_opera(&rmutex);
149             my_p_opera(&max_reader);    //max_reader信号量控制最多
可读人数
150
151                 reader_num++;
152                 //读文件
153                 my_disp_str(name,color);
154                 my_disp_str(" is reading!   ",color);
155                 //int t=ticks;
156                 //while(get_ticks()-t<proc_ticks){}
```

```
157                 milli_delay(proc_ticks * time_piece);       //自
    定义时间片长度
158                 reader_num--;
159
160             my_v_opera(&max_reader);    //当前进程结束读，max_reade
    r++
161             my_p_opera(&rmutex);
162             readcount--;                //读完成，读者数--
163             if(readcount==0) my_v_opera(&wmutex);   //如果没有人在
    读就释放wmutex++
164             my_v_opera(&rmutex);
165
166                 //读完成
167                 my_disp_str(name,color);
168                 my_disp_str(" finish reading!   ",color);
169
170             my_milli_seconds(10000);        //饥饿问题
171         }
172     }
173     else{               //写者优先
174         while(1){
175                 //开始读
176                 my_disp_str(name,color);
177                 my_disp_str(" come for reading!   ",color);
178
179
180             my_p_opera(&z);
181             my_p_opera(&rmutex);
182             my_p_opera(&x);
183             if(readcount==0) my_p_opera(&wmutex);       //读时不能
    写，wmutex--
184             readcount++;
185             my_v_opera(&x);
186             my_v_opera(&rmutex);
187             my_v_opera(&z);
188             my_p_opera(&max_reader);    //max_reader信号量控制最多
    可读人数
189
190                 reader_num++;
191                 //读文件
```

```
192                 my_disp_str(name,color);
193                 my_disp_str(" is reading!   ",color);
194                 //int t=ticks;
195                 //while(get_ticks()-t<proc_ticks){}
196                 milli_delay(proc_ticks * time_piece);       //自
    定义时间片长度
197                 reader_num--;
198
199             my_v_opera(&max_reader);    //当前进程结束读，max_reade
    r++
200             my_p_opera(&x);
201             readcount--;
202             if(readcount==0) my_v_opera(&wmutex);       //没有读
    者，可以写，wmutex++
203             my_v_opera(&x);
204
205                 //读完成
206                 my_disp_str(name,color);
207                 my_disp_str(" finish reading!   ",color);
208         }
209
210     }
211 }
212
213 /*================================================================
    ========*
214                                 写者进程
215  *================================================================
    ========*/
216 PUBLIC void Writer(char* name,int color,int proc_ticks){
217     if(priority==0){        //读者优先
218         while(1){
219                 //开始写
220                 my_disp_str(name,color);
221                 my_disp_str(" come for writing!   ",color);
222
223             my_p_opera(&wmutex);
224             writercount++;
225
226                 //写文件
```

```
227                 my_disp_str(name,color);
228                 my_disp_str(" is writing!   ",color);
229                 //int t=ticks;
230                 //while(get_ticks()-t<proc_ticks){}
231                 milli_delay(proc_ticks * time_piece);       //自
    定义时间片长度
232
233             my_v_opera(&wmutex);
234             writercount--;
235
236                 //写完成
237                 my_disp_str(name,color);
238                 my_disp_str(" finish writing!   ",color);
239         }
240     }
241     else{               //写者优先
242         while(1){
243                 //开始写
244                 my_disp_str(name,color);
245                 my_disp_str(" come for writing!   ",color);
246
247             my_p_opera(&y);
248             writercount++;
249             if(writercount==1) my_p_opera(&rmutex);     //如果没有
    人在写，把读互斥量rmutex--
250             my_v_opera(&y);
251
252             my_p_opera(&wmutex);
253                 //写文件
254                 my_disp_str(name,color);
255                 my_disp_str(" is writing!   ",color);
256                 //int t=ticks;
257                 //while(get_ticks()-t<proc_ticks){}
258                 milli_delay(proc_ticks * time_piece);       //自
    定义时间片长度
259             my_v_opera(&wmutex);
260
261             my_p_opera(&y);
262             writercount--;
263             if(writercount==0) my_v_opera(&rmutex);
```

```
            my_v_opera(&y);

                //写完成
                my_disp_str(name,color);
                my_disp_str(" finish writing!   ",color);

            my_milli_seconds(10000);        //饥饿问题
        }
    }
}

/*================================================================
========*
                                清屏函数
 *================================================================
========*/
PUBLIC void clear_screen(){
    disp_pos=0;
    for(int i=0;i< 80*25;i++){
        disp_color_str(" ",0x07);
    }
    disp_pos=0;
}
```