

# 计算机操作系统第四次实验说明文档

张洪胤

## 一、通过系统调用完成 delay\_sys 方法

```
delay_sys:
; 准备好现场, 准备发起中断
mov eax, _NR_sys_delay
push ebx
mov ebx, [esp+8]
int INT_VECTOR_SYS_CALL
pop ebx
ret

1 reference
sys_delay:
push ebx
call sys_delay_c
pop ebx
ret
```

图 1 syscall.asm 中的相关代码

该方法在 syscall.asm 中编写完成(如图 1 所示)相关代码, 在 global.c 的系统调用表中添加函数(如图 2 所示), 在 const.h 中修改 NR\_SYS\_CALL 的值, 在 proto.h 中添加相关的系统声明(如图 3 所示), 最终通过调用在 proc.c 文件中编写的 sys\_delay\_c 方法来完成系统调用(如图 4 所示)。

```
PUBLIC system_call    sys_call_table[NR_SYS_CALL] = {
    sys_get_ticks,
    sys_disp_str,
    sys_p,
    sys_v,
    sys_delay
};
```

图 2 global.c 中的系统调用表

```
/* proc.c */
PUBLIC int    sys_get_ticks();           /* sys_call */
PUBLIC void    sys_delay_c(int i);
PUBLIC void    do_sys_p(SEMAPHORE *);
PUBLIC void    do_sys_v(SEMAPHORE *);

/* syscall.asm 系统调用 */
PUBLIC void    sys_call();               /* int_handler */
PUBLIC int    get_ticks();
PUBLIC void    sys_p(SEMAPHORE *);
PUBLIC void    sys_v(SEMAPHORE *);
PUBLIC void    delay_sys(int milli_seconds);
PUBLIC void    disp_str_sys(char *);

PUBLIC void    p_sys(SEMAPHORE *);
PUBLIC void    v_sys(SEMAPHORE *);
PUBLIC void    sys_delay(int i);
PUBLIC void    sys_disp_str(char *);
```

图 3 proto.h 中的函数声明

```
// 不分配时间片来切换，这部分被汇编调用
PUBLIC void sys_delay_c(int i){
    p_proc_ready->ready_tick = get_ticks() + i / (1000 / HZ);
    // delay则进行进程切换
    schedule();
}
```

图 4 sys\_delay\_c 方法实现

为了完成上面的方法中的推迟效果，我需要对进程表结构体进行修改(如图 5 所示)，在其中添加 ready\_tick 属性来表示进程再次就绪的时间，在进程被 delay 后我立即调起 schedule 函数进行调度。

```
/* 进程表结构 */
typedef struct s_proc {
    STACK_FRAME regs; /* process' registers saved in stack frame */

    u16    ldt_sel; /* selector in gdt giving ldt base and limit*/
    DESCRIPTOR ldts[LDT_SIZE]; /* local descriptors for code and data */
    /* 2 is LDT_SIZE - avoid include protect.h */

    int    ready_tick; /* 进程可以再次就绪的时间 */
    SEMAPHORE* waiting_semaphore; /* 进程等待的信号量 */

    u32    pid; /* process id passed in from MM */
    char    p_name[16]; /* name of the process */
}PROCESS;
```

图 5 修改后的 PROCESS 结构体

最后我来描述一下 schedule 函数(如图 6 所示)，其调度算法则是通过遍历来找到一个超过就绪时刻并且没有被信号量阻塞的进程来占用 CPU，这样子我就避免了被 delay 的进程在这段时间内占用时间片，从而解决了 mills\_delay 方法占用时间片的问题。

```
// 进程调度
PUBLIC void schedule(){
    int t = 0;

    while(1){
        t = get_ticks();
        p_proc_ready ++;
        if(p_proc_ready >= proc_table + NR_TASKS){
            p_proc_ready = proc_table;
        }
        if(p_proc_ready->waiting_semaphore == 0 && p_proc_ready->ready_tick <= t){
            break;
        }
    }
}
```

图 6 schedule 调度函数

## 二、通过系统调用完成打印字符串

我实现了系统调用 `disp_str_sys`，接受一个 `char *`类型的参数，将字符串打印到屏幕上，同样是在 `syscall.asm` 中编写相应代码（如图 7 所示），在 `global.c` 中的系统调用表中添加（如图 2 所示），在 `proto.h` 中进行函数声明（如图 3 所示），最后通过调用 `disp_str` 来完成打印。

<pre>disp_str_sys: ; 准备好现场, 准备发起中断 mov eax, _NR_disp_str_sys push ebx mov ebx, [esp+8] int INT_VECTOR_SYS_CALL pop ebx ret</pre>	<pre>sys_disp_str: ; 具体执行中断内容 pusha push ebx call disp_str pop ebx popa ret</pre>
--	---

图 7 `syscall.asm` 中的相关代码

## 三、通过系统调用执行信号量 PV 操作，并模拟读写者问题

### （一）通过系统调用执行信号量 PV 操作

```
typedef struct semaphore{
    int    number;    // 值数量
    PROCESS* list[SEMAPHORE_SIZE];
    int    start;
    int    end;
} SEMAPHORE;
```

图 8 `proc.h` 中信号量 `SEMAPHORE` 的声明

首先我在 `proc.h` 中完成了对信号量 `SEMAPHORE` 的声明（如图 8 所示），包含了当前信号量剩余的资源数量，等待队列，队列头和队列尾。

然后我实现了系统调用 `p_sys` 和 `v_sys`，完成对信号量的 PV 操作，同样是在 `syscall.asm` 中编写相应代码（如图 9 所示），在

global.c 中的系统调用表中添加(如图 2 所示), 在 proto.h 中进行函数声明(如图 3 所示), 然后分别通过调用在 proc.c 中编写的 do\_sys\_p 和 do\_sys\_v 函数(如图 10 所示)来完成操作。

<pre> p_sys: ; 准备好现场, 准备发起中断 mov eax, _NR_p push ebx mov ebx, [esp+8] int INT_VECTOR_SYS_CALL pop ebx ret  1 reference sys_p: ; 信号量执行P操作 pusha push ebx call do_sys_p pop ebx popa ret </pre>	<pre> v_sys: ; 准备好现场, 准备发起中断 mov eax, _NR_v push ebx mov ebx, [esp+8] int INT_VECTOR_SYS_CALL pop ebx ret  1 reference sys_v: ; 信号量执行V操作 pusha push ebx call do_sys_v pop ebx popa ret </pre>
---	---

图 9 syscall.asm 中和 p\_sys 以及 v\_sys 相关的代码

```

// 执行信号量P操作
PUBLIC void do_sys_p(SEMAPHORE* semaphore){
    semaphore -> number--;
    if(semaphore -> number < 0){
        // 等待一个信号量
        p_proc_ready->waiting_semaphore = semaphore;

        semaphore -> list[semaphore -> end] = p_proc_ready;
        semaphore -> end = (semaphore -> end + 1) % SEMAPHORE_SIZE;
        // 进行进程调度
        schedule();
    }
}

// 执行信号量V操作
PUBLIC void do_sys_v(SEMAPHORE* semaphore){
    semaphore -> number++;
    if(semaphore -> number <= 0){
        // 等待队列中的第一个进程取出来
        PROCESS* p = semaphore -> list[semaphore -> start];
        p -> waiting_semaphore = 0;
        semaphore -> start = (semaphore -> start + 1) % SEMAPHORE_SIZE;
    }
}

```

图 10 proc.c 中 do\_sys\_p 和 do\_sys\_v 的函数实现

如图 10 所示，对于 P 操作，如果我们当前的信号量有足够的资源则占用一个资源，如果当前资源不足，则将当前进程添加到当前的队列中，然后立即执行调度算法（上文已有描述，不再赘述）；对于 V 操作，我们将当前的资源释放，如果当前等待资源队列不为空，则释放对应的进程的信号量，使其可以参加调度。

## （二）模拟读写者问题

### 1. 添加 A-F 的进程

```
/* Number of tasks */
#define NR_TASKS 6

/* stacks of tasks 定义任务栈的大小 */
#define STACK_SIZE_ReaderA 0x8000
#define STACK_SIZE_ReaderB 0x8000
#define STACK_SIZE_ReaderC 0x8000
#define STACK_SIZE_WriterD 0x8000
#define STACK_SIZE_WriterE 0x8000
#define STACK_SIZE_F 0x8000

#define STACK_SIZE_TOTAL (STACK_SIZE_ReaderA + \
    STACK_SIZE_ReaderB + \
    STACK_SIZE_ReaderC + \
    STACK_SIZE_WriterD + \
    STACK_SIZE_WriterE + \
    STACK_SIZE_F)
```

图 11 proc.h 中关于进程数量和栈空间的声明

```
/* 任务列表 */
PUBLIC TASK task_table[NR_TASKS] = {
    {ReaderA, STACK_SIZE_ReaderA, "ReaderA"},
    {ReaderB, STACK_SIZE_ReaderB, "ReaderB"},
    {ReaderC, STACK_SIZE_ReaderC, "ReaderC"},
    {WriterD, STACK_SIZE_WriterD, "WriterD"},
    {WriterE, STACK_SIZE_WriterD, "WriterE"},
    {F, STACK_SIZE_F, "F"}};
```

图 12 global.c 中任务列表

我们在 proc.h 中完成 A-F 进程的数量值 NR\_TASK 的修改，并且完成相关栈空间的分配（如图 11 所示），在 global.c 中完成了任务列表的声明（如图 12 所示）。

然后在 main.c 中添加了进程相关的变量，分别是进程名表，

颜色表和正在运行进程表（如图 13 所示），其中 processList 主要是提供给进程 F 使用。

```
// 绿色, 蓝色, 红色, 黄色
int colors[4] = {0x0A, 0x03, 0x0C, 0x0E};
char* names[5] = {"ReaderA", "ReaderB", "ReaderC", "WriterD", "WriterE"};
int processList[5] = {0};
```

图 13 main.c 中进程相关的变量

之后，我们完成了对 A-E 进程内容的实现（如图 14 所示）。

```
void ReaderA()
{
    if(WHO_FIRST){
        reader_write(0, 2);
    }else{
        reader_read(0, 2);
    }
}

void ReaderB()
{
    if(WHO_FIRST){
        reader_write(1, 3);
    }else{
        reader_read(1, 3);
    }
}

void ReaderC()
{
    if(WHO_FIRST){
        reader_write(2, 3);
    }else{
        reader_read(2, 3);
    }
}

void WriterD()
{
    if(WHO_FIRST){
        writer_write(3, 3);
    }else{
        writer_read(3, 3);
    }
}

void WriterE()
{
    if(WHO_FIRST){
        writer_write(4, 4);
    }else{
        writer_read(4, 4);
    }
}
```

图 14 main.c 中 A-E 进程内容的实现

最后我们完成了对普通进程 F 的内容实现（如图 15 所示）

```
void F(){
    while(1){
        if(reader_cnt <= READER_MAX && reader_cnt > 0){
            disp_str_sys("F: ");
            disp_str_sys("The number of ");
            disp_color_str("reader", colors[0]);
            disp_str_sys(" : ");
            char* number = "0";
            number[0] = (char) ('0' + reader_cnt);
            disp_color_str(number, colors[0]);
            disp_str_sys("\n");
            disp_number++;
        }else if(processList[3] != 0 || processList[4] != 0){
            disp_str_sys("F: ");
            disp_str_sys("The name of ");
            disp_color_str("writer", colors[1]);
            disp_str_sys(" : ");

            if(processList[3]){
                disp_str_sys(" ");
                disp_color_str(names[3], colors[1]);
            }else if(processList[4]){
                disp_str_sys(" ");
                disp_color_str(names[4], colors[1]);
            }
            disp_str_sys("\n");
            disp_number++;
        }
        delay_sys(5000);
    }
}
```

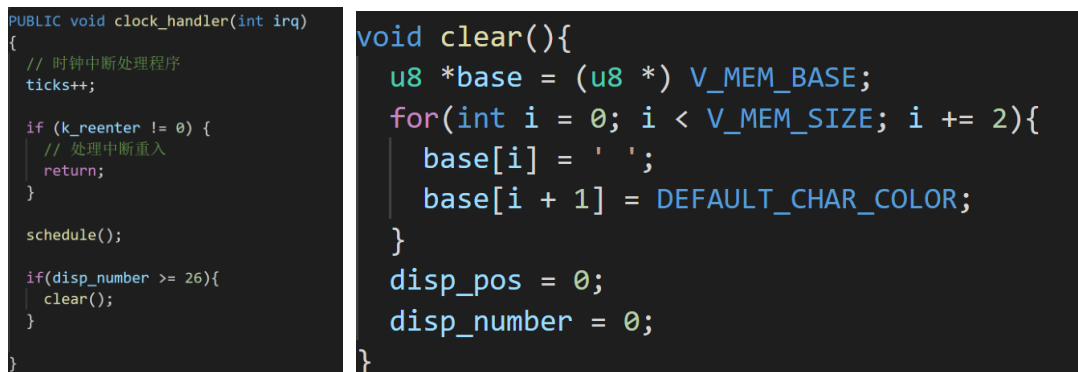
图 15 main.c 中对普通进程 F 的实现



在上面的实现中，我们通过判断当前是在读还是写，如果是读，则输出读进程的个数，如果是写，则输出当前在写的进程是什么。

## 2. 添加清屏函数

由于打印出来的信息相对比较多，为了方便查看，我们需要根据打印的信息的行数满屏后进行清屏，具体实现如下。在 global.h 中添加 disp\_number 变量，表示当前打印的行数，每次打印时都会调用变更。在 clock.c 中时钟中断处理部分，添加代码保证屏幕写满后清空屏幕。在 proto.h 中添加 clear 函数的函数声明，在 main.c 中完成函数 clear 的实现（如图 16 所示）



```
PUBLIC void clock_handler(int irq)
{
    // 时钟中断处理程序
    ticks++;

    if (k_reenter != 0) {
        // 处理中断重入
        return;
    }

    schedule();

    if (disp_number >= 26) {
        clear();
    }
}

void clear(){
    u8 *base = (u8 *) V_MEM_BASE;
    for(int i = 0; i < V_MEM_SIZE; i += 2){
        base[i] = ' ';
        base[i + 1] = DEFAULT_CHAR_COLOR;
    }
    disp_pos = 0;
    disp_number = 0;
}
```

图 16 清屏相关代码部分

## 3. 信号量相关配置和说明

我们在 main.c 中添加了两个变量 reader\_cnt 和 writer\_cnt 分别用来表示当前正在读或写的进程的数量。同时引入了 6 个信号量用来解决读者优先和写者优先问题，以及其中的进程饥饿问题。

信号量信号量 mutexw 用来限制变量 writer\_cnt，信号量 reader 用来控制读进程，信号量 writer 用来控制写进程，信号量

writeBlock 用来控制只有没有读进程时才可以写，信号量 S 可以用来控制读写者问题中的饿死问题，在 main.c 中完成初始化（如图 17 所示）。

```
// 读者
initSemaphore(&reader, READER_MAX);
// 写者
initSemaphore(&writer, 1);
// 对writer_cnt的互斥信号量
initSemaphore(&mutexw, 1);
// 是否允许写的信号量
initSemaphore(&writeBlock, 1);
// 用于解决饿死问题的信号量
initSemaphore(&S, 1);

void initSemaphore(SEMAPHORE* semaphore, int number){
    semaphore->number = number;
    semaphore->start = 0;
    semaphore->end = 0;
}
```

图 17 main.c 中的信号量初始化部分

```
// 阅读者上限
#define READER_MAX 2
// 阅读时间
#define READING_TIME 10 * 50000 / HZ
// 写入时间
#define WRITING_TIME 10 * 50000 / HZ
// 每次任务间隔时间
#define GAP_TIME 50000 / HZ
// 读者优先还是写者优先 0是读者优先 1是写者优先
#define WHO_FIRST 0
```

图 18 main.c 中的常量定义

同样的，为了实现实验要求，我们在 main.c 中完成了常量的定义（如图 18 所示），用来方便地修改阅读者上限、读者优先和写者优先等部分。

#### 4. 读者优先

在读写者问题中，我们通过实现 reader\_read 和 writer\_read 来实现读者优先策略并且通过信号量 S 解决了读者优先问题中的写者进程饥饿问题。



```

void reader_read(int i, int time){
    while(1){
        p_sys(&S);
        p_sys(&reader);
        if(reader_cnt == 0){
            p_sys(&writeBlock);
        }
        reader_cnt++;
        v_sys(&S);

        disp_one_line(i, " is waiting ... ", colors[0]);
        disp_one_line(i, " starts reading ... ", colors[0]);
        processList[i] = 1;
        disp_one_line(i, " is reading ... ", colors[0]);
        milli_delay(time * READING_TIME);
        disp_one_line(i, " stop reading ... ", colors[0]);
        processList[i] = 0;

        v_sys(&reader);
        reader_cnt--;
        if(reader_cnt == 0){
            v_sys(&writeBlock);
        }

        milli_delay(GAP_TIME);
    }
}

```

图 19 reader\_read 方法实现

在 reader\_read 方法（如图 19 所示）中，我们使用了 S、writeBlock、reader 这三个信号量，其作用之前已经说明不再赘述。

```

void writer_read(int i, int time){
    while(1){
        p_sys(&S);
        p_sys(&writeBlock);

        disp_one_line(i, " is waiting ... ", colors[1]);
        disp_one_line(i, " starts writing ... ", colors[1]);
        processList[i] = 1;
        disp_one_line(i, " is writing ... ", colors[1]);
        milli_delay(time * WRITING_TIME);
        disp_one_line(i, " finishes writing ... ", colors[1]);
        processList[i] = 0;

        v_sys(&writeBlock);
        v_sys(&S);
        milli_delay(GAP_TIME);
    }
}

```

图 20 writer\_read 方法实现

在 writer\_read 方法（如图 20 所示）中，我们使用了 S 和 writeBlock 两个信号量，其中如果能获取到信号量 writeBlock

则意味着没有读进程在进行，从而保证写进程执行时没有读进程，而信号量 S 可以保证在写进程进入时，之后达到的读进程不会进入到等待队列而解决了读者优先中的写者饿死的问题。

## 5. 写者优先

```
void reader_write(int i, int time){
    while(1){
        // 其实S是用来解决写者优先的饿死问题的
        //p_sys(&S);
        p_sys(&mutexw);
        p_sys(&reader);
        if(reader_cnt == 0){
            p_sys(&writeBlock);
        }
        reader_cnt++;
        v_sys(&mutexw);
        //v_sys(&S);

        disp_one_line(i, " is waiting ... ", colors[0]);
        disp_one_line(i, " starts reading ... ", colors[0]);
        processlist[i] = 1;
        disp_one_line(i, " is reading ... ", colors[0]);
        milli_delay(time * READING_TIME);
        disp_one_line(i, " stop reading ... ", colors[0]);
        processlist[i] = 0;

        v_sys(&reader);
        reader_cnt--;
        if(reader_cnt == 0){
            v_sys(&writeBlock);
        }
        milli_delay(GAP_TIME);
    }
}

void writer_write(int i, int time){
    while(1){
        // 控制写进程
        //p_sys(&S);
        p_sys(&writer);
        writer_cnt++;
        if(writer_cnt == 1){
            p_sys(&mutexw);
        }
        v_sys(&writer);

        p_sys(&writeBlock);
        disp_one_line(i, " is waiting ... ", colors[1]);
        disp_one_line(i, " starts writing ... ", colors[1]);
        processlist[i] = 1;
        disp_one_line(i, " is writing ... ", colors[1]);
        milli_delay(time * WRITING_TIME);
        disp_one_line(i, " finishes writing ... ", colors[1]);
        processlist[i] = 0;
        v_sys(&writeBlock);

        p_sys(&writer);
        writer_cnt--;
        if(writer_cnt == 0){
            v_sys(&mutexw);
        }
        v_sys(&writer);
        //v_sys(&S);
        milli_delay(GAP_TIME);
    }
}
```

图 21 写者优先中的读写方法实现

在 reader\_write 方法（如图 21 所示）中，我们使用了 mutexw、writeBlock、reader 这四个信号量，在写者优先问题中没有解决饿死问题（即引入信号量 S），我们只需要将左侧和右侧中注释掉的和 S 相关的 PV 操作解除注释即可解决写者优先的进程饿死问题。

在 writer\_write 方法（如图 21 所示）中，我们使用了 writer、mutexw 和 writeBlock 三个信号量，其中如果能获取到信号量 writeBlock 则意味着没有读进程在进行，从而保证写进程执行时没有读进程。