

OSLAB4

徐琪 191250165

OSLAB4

一、运行与结果

1. 运行

2. 结果

2.1 读者优先

2.1.1 并发量为1

2.1.2 并发量为2

2.1.3 并发量为3

2.2 写者优先

2.2.1 并发量为1

2.2.2 并发量为2

2.2.3 并发量为3

2.3 读写公平

2.3.1 并发量为1

2.3.2 并发量为2

2.3.3 并发量为3

二、保姆级实现步骤

1. 添加新任务

2. 添加系统调用

2.1 sleep

2.2 print

2.3 P 和 V

3. 进程调度

3.1 读者优先

3.2 写者优先

3.3 读写公平（解决进程饿死问题）

一、运行与结果

1. 运行

1. 在 const.h 中

1. 修改 MAX_READERS 的值，设定可同时读的用户数量（1-3）

2. 修改 STRATEGY 的值，可改变进程调度的策略（0: 读优先 1: 写优先 3: 读写公平，解决饥饿问题）

2. 根目录输入以下命令即可运行

```
make run  
bochs
```

2. 结果

为了让结果区别更显著，可以动态调整 main.c 中各个进程的延迟时间（while 循环中的 milli_delay 方法的参数），设定不同进程到达的频率。也可以在各个进程的 while 循环前添加 milli_delay() 方法，设定进程一开始到达的时间。

2.1 读者优先

2.1.1 并发量为1

此例子中，B 正在读时，A - E 进程均到达，根据读者优先且并发量为1的规则，B 读完后，依次按照 Reader C、Reader A、Writer D、Writer E 的顺序执行。

2.1.2 并发量为2

例子同上，不同的是 B 进程正在读时，Reader C 到达可以直接读，无需等待。而 Reader A 到达时，已有两个进程在读，故在 B 读完后 A 方可读。

全部读完后，才轮到 D 和 E 两个写进程执行。

2.1.3 并发量为3

例子同上，不同的是 B 进程正在读时，Reader C 和 Reader A 到达均可直接读，无需等待。

全部读完后，才轮到 D 和 E 两个写进程执行。

2.2 写者优先

2.2.1 并发量为1

此例子中，B 正在读时，A - E 进程均到达，根据写者优先且并发量为1的规则，B 读完后，先依次执行 Writer D 和 E，再依次执行 Reader C 和 A。

2.2.2 并发量为2

例子同上，不同的是，B 进程正在读时，Reader C 到达可以直接读，无需等待。两个读进程执行完后，依次执行 Writer D 和 E，Reader A 在写进程都执行完后方可执行。

2.2.3 并发量为3

例子同上，结果也与上面类似，因为本例子中，Reader A 到达时，Reader B 和 C 刚好执行结束，所以接下来先执行写进程，最后再执行 Reader A。

2.3 读写公平

2.3.1 并发量为1

2.3.2 并发量为2

2.3.3 并发量为3

二、保姆级实现步骤

基于 Orange'S 6r

1. 添加新任务

1. 在 main.c 中添加进程体

```
void ReaderA() {}  
  
void ReaderB() {}  
  
void ReaderC() {}  
  
void WriterD() {}  
  
void WriterE() {}  
  
void NormalF() {}
```

2. 在 global.c 的 task_table 中添加相应的进程

```
PUBLIC TASK task_table[NR_TASKS] = {{ReaderA, STACK_SIZE_READER_A,  
"ReaderA"},  
                                     {ReaderB, STACK_SIZE_READER_B,  
"ReaderB"},  
                                     {ReaderC, STACK_SIZE_READER_C,  
"ReaderC"},  
                                     {WriterD, STACK_SIZE_WRITER_D,  
"WriterD"},  
                                     {WriterE, STACK_SIZE_WRITER_E,  
"WriterE"},  
                                     {NormalF, STACK_SIZE_NORMAL_F,  
"NormalF"}};
```

3. 在 proc.h 中修改 NR_TASKS的值，并定义任务堆栈，修改 STACK_SIZE_TOTAL

```

/* Number of tasks */
#define NR_TASKS 6

/* stacks of tasks */
#define STACK_SIZE_READER_A 0x8000
#define STACK_SIZE_READER_B 0x8000
#define STACK_SIZE_READER_C 0x8000
#define STACK_SIZE_WRITER_D 0x8000
#define STACK_SIZE_WRITER_E 0x8000
#define STACK_SIZE_NORMAL_F 0x8000

#define STACK_SIZE_TOTAL (STACK_SIZE_READER_A + \
                           STACK_SIZE_READER_B + \
                           STACK_SIZE_READER_C + \
                           STACK_SIZE_WRITER_D + \
                           STACK_SIZE_WRITER_E + \
                           STACK_SIZE_NORMAL_F)

```

4. 在 proto.h 中添加任务执行体的函数声明

```

/* main.c */
void ReaderA();
void ReaderB();
void ReaderC();
void WriterD();
void WriterE();
void NormalF();

```

2. 添加系统调用

包括: sleep, print, P, V

1. 在 proto.h 中声明系统级和用户级系统调用函数

```

/* 系统调用 - 系统级 */
/* proc.c */
PUBLIC int sys_get_ticks(); /* sys_call */
// addition
PUBLIC int sys_sleep(int milli_seconds);
PUBLIC int sys_print(char *str);
PUBLIC int sys_P(void *s);
PUBLIC int sys_V(void *s);

/* syscall.asm */
PUBLIC void sys_call(); /* int_handler */

/* 系统调用 - 用户级 */
PUBLIC int get_ticks();
// addition
PUBLIC int sleep(int milli_seconds);
PUBLIC int print(char *str);

```

```
PUBLIC int P(void *s);
PUBLIC int V(void *s);
```

2. 修改 const.h 中 NR_SYS_CALL 的值

```
/* system call */
#define NR_SYS_CALL      5
```

3. 在 global.c 的 sys_call_table 中添加相应的系统调用

```
PUBLIC system_call sys_call_table[NR_SYS_CALL] = {sys_get_ticks,
                                                    sys_sleep,
                                                    sys_print,
                                                    sys_P,
                                                    sys_V};
```

4. 在 syscall.asm 中添加声明

```
_NR_get_ticks      equ 0 ; 要跟 global.c 中 sys_call_table 的定义相对应！
INT_VECTOR_SYS_CALL equ 0x90
; addition
_NR_sleep          equ 1
_NR_print          equ 2
_NR_P              equ 3
_NR_V              equ 4

; 导出符号
global get_ticks
; addition
global sleep
global print
global P
global V
```

并添加相应的用户调用函数体

```
sleep:
    mov eax, _NR_sleep
    mov ebx, [esp + 4]
    int INT_VECTOR_SYS_CALL
    ret

print:
    mov eax, _NR_print
    mov ebx, [esp + 4]
    int INT_VECTOR_SYS_CALL
    ret

P:
```

```

    mov eax, _NR_P
    mov ebx, [esp + 4]
    int INT_VECTOR_SYS_CALL
    ret

V:
    mov eax, _NR_V
    mov ebx, [esp + 4]
    int INT_VECTOR_SYS_CALL
    ret

```

5. 在 proc.c 中添加系统调用处理函数体（详见下文说明）

6. 如果进行了参数传递，在 kernel.asm 中修改 sys_call 中的寄存器进栈出栈

添加 push 和 add 操作

```

sys_call:
    call    save

    sti

    push    ebx
    call    [sys_call_table + eax * 4]
    add     esp,4
    mov     [esi + EAXREG - P_STACKBASE], eax

    cli

    ret

```

2.1 sleep

接受 int 型参数 milli_seconds，调用此方法进程会在 milli_seconds 毫秒内不被分配时间片。

在 proc.h 中为每个 PROCESS 添加以下两个属性

```

int sleep_ticks; /* 睡眠的时间 */
int isBlocked;   /* 是否被阻塞 */

```

在 proc.c 中添加 sys_sleep() 系统调用处理函数体

```

PUBLIC int sys_sleep(int milli_seconds)
{
    p_proc_ready->sleep_ticks = milli_seconds / (1000 / HZ * 500); // 最后的乘500是为了修正系统的时钟中断错误（maybe？）
    schedule();
}

```

2.2 print

接受 char* 型参数 str, 打印字符串

在 const.h 中定义相应的颜色相关常量和宏函数

```
/* color */
#define BLACK 0x0          /* 0000 */
#define WHITE 0x7         /* 0111 */
#define RED 0x4            /* 0100 */
#define GREEN 0x2         /* 0010 */
#define BLUE 0x1          /* 0001 */
#define YELLOW 0x6        /* 0110 */
#define PURPLE 0x5        /* 0101 */
#define FLASH 0x80        /* 1000 0000 */
#define BRIGHT 0x08       /* 0000 1000 */
#define MAKE_COLOR(x, y) (x | y) /* MAKE_COLOR(Background,Foreground) */
```

在 proc.c 中添加 sys_print() 系统调用处理函数体

表驱动, 根据进程不同的偏移量, 选择不同颜色

```
PUBLIC int sys_print(char *str)
{
    if (disp_pos >= 80 * 25 * 2)
    {
        memset(0xB8000, 0, 80 * 25 * 2);
        disp_pos = 0;
    }
    int offset = p_proc_ready - proc_table;
    int colors[] = {RED, GREEN, BLUE, YELLOW, PURPLE, WHITE};
    disp_color_str(str, BRIGHT | MAKE_COLOR(BLACK, colors[offset]));
}
```

2.3 P 和 V

信号量 PV 操作

在 proc.h 中定义 s_semaphore 结构体 (信号量)

head 和 tail 充当指针作用, 便于用数组实现队列

```
typedef struct s_semaphore
{
    int value;
    int head;
    int tail;
    PROCESS *pQueue[NR_TASKS]; /* 等待信号量的进程队列 */
} SEMAPHORE;
```

在 proc.c 中添加 sys_P() 和 sys_V() 的系统调用处理函数体

```

PUBLIC int sys_P(void *sem)
{
    disable_irq(CLOCK_IRQ); // 保证原语
    SEMAPHORE *s = (SEMAPHORE *)sem;
    s->value--;
    if (s->value < 0)
    { // 将进程加入队列尾
        p_proc_ready->isBlocked = TRUE;
        s->pQueue[s->tail] = p_proc_ready;
        s->tail = (s->tail + 1) % NR_TASKS;
        schedule();
    }
    enable_irq(CLOCK_IRQ);
}

PUBLIC int sys_V(void *sem)
{
    disable_irq(CLOCK_IRQ); // 保证原语
    SEMAPHORE *s = (SEMAPHORE *)sem;
    s->value++;
    if (s->value <= 0)
    { // 释放队列头的进程
        PROCESS *proc = s->pQueue[s->head];
        proc->isBlocked = FALSE;
        s->head = (s->head + 1) % NR_TASKS;
    }
    enable_irq(CLOCK_IRQ);
}

```

3. 进程调度

1. 在 const.h 中定义 MAX_READERS 和 STRATEGY，分别表示可以同时读的读者数量和调度策略

```

#define MAX_READERS 3 /* 可同时读的数量 */
#define STRATEGY 0 /* 读优先--0 写优先--1 读写公平--2 */

```

定义 TIME_SLICE，表示一个时间片的长度

```

#define TIME_SLICE 10000 /* 一个时间片长度 */

```

2. 在 global.h 中定义各种信号量，以及记录读者写者数量的变量


```

EXTERN int readerNum;
EXTERN int writerNum;
EXTERN int readerNum_rf; // 读优先中，真正在读的人数

EXTERN SEMAPHORE readerLimit; // 同时读同一本书的人数
EXTERN SEMAPHORE writeBlock; // 限制写进程
EXTERN SEMAPHORE readBlock; // 限制读进程，保证写优先
EXTERN SEMAPHORE mutex_readerNum; // 保护 readerNum 的变化
EXTERN SEMAPHORE mutex_writerNum; // 保护 writerNum 的变化
EXTERN SEMAPHORE mutex_fair; // 实现读写公平

```

并在 global.c 中初始化信号量

```

PUBLIC SEMAPHORE readerLimit = {MAX_READERS, 0, 0};
PUBLIC SEMAPHORE writeBlock = {1, 0, 0};
PUBLIC SEMAPHORE readBlock = {1, 0, 0};
PUBLIC SEMAPHORE mutex_readerNum = {1, 0, 0};
PUBLIC SEMAPHORE mutex_writerNum = {1, 0, 0};
PUBLIC SEMAPHORE mutex_fair = {1, 0, 0};

```

3. 在 main.c 中添加 init() 方法，用于清屏和对变量的初始化

```

PUBLIC void init()
{
    // 清屏
    disp_pos = 0;
    for (int i = 0; i < 80 * 25; i++)
    {
        disp_str(" ");
    }
    disp_pos = 0;

    // 初始化变量
    readerCount = 0;
    writerCount = 0;
    trueReaderCount = 0;
}

```

并在 kernel_main() 方法中的 restart() 方法前调用

kernel_main() 方法中，还要赋值优先级和 isBlocked 属性的初值

```

for (int i = 0; i < NR_TASKS; i++)
{
    proc_table[i].ticks = proc_table[i].priority = 1;
    proc_table[i].isBlocked = proc_table[i].sleep_ticks = 0;
}

```

4. 完善 proc.c 中的 schedule() 函数

让所有正在休眠的进程等待时间 - 1

```
for (p = proc_table; p < proc_table + NR_TASKS; p++)
{
    if (p->sleep_ticks > 0)
        p->sleep_ticks--;
}
```

找等待时间最久的进程时，要跳过正在睡眠or被阻塞的进程

```
if (p->sleep_ticks > 0 || p->isBlocked == TRUE)
    continue;
```

重设 ticks 时，跳过被阻塞的进程

```
if (p->ticks > 0) // 说明被阻塞了
    continue;
```

5. 在 proto.h 中声明总的读写接口函数（WRITER() 和 READER() 方法），和三种策略各自的读写函数

```
/* 读写接口函数 */
PUBLIC void WRITER(char *name, int time_slice);
PUBLIC void READER(char *name, int time_slice);
/* 读优先 */
PUBLIC void WRITER_rf(char *name, int time_slice);
PUBLIC void READER_rf(char *name, int time_slice);
/* 写优先 */
PUBLIC void WRITER_wf(char *name, int time_slice);
PUBLIC void READER_wf(char *name, int time_slice);
/* 读写公平 */
PUBLIC void WRITER_fair(char *name, int time_slice);
PUBLIC void READER_fair(char *name, int time_slice);
```

6. 在 main.c 中定义相应的函数体（详见下文说明）

7. 在 main.c 中填写进程体的定义

进程 A-E 以固定的延迟（milli_delay）不断调用相应的读/写接口函数。可自行调整延迟长短，也可在 while(TRUE) 前添加延迟。

```
void ReaderA()
{
    while (TRUE)
    {
        READER("A\0", 2);
        milli_delay(TIME_SLICE / 2);
    }
}

void ReaderB()
{
    while (TRUE)
```

```

    {
        READER("B\0", 3);
        milli_delay(TIME_SLICE / 2);
    }
}

void ReaderC()
{
    while (TRUE)
    {
        READER("C\0", 3);
        milli_delay(TIME_SLICE / 2);
    }
}

void WriterD()
{
    while (TRUE)
    {
        WRITER("D\0", 3);
        milli_delay(TIME_SLICE / 2);
    }
}

void WriterE()
{
    while (TRUE)
    {
        WRITER("E\0", 4);
        milli_delay(TIME_SLICE / 2);
    }
}

```

进程 F 使用系统调用封装 (sleep)，每隔一个时间片打印当前的读写状态和在读人数
如果无人读写则跳过该时间片，继续轮询

```

void NormalF()
{
    // sleep(TIME_SLICE);
    char isR[23] = " process is reading> ";
    char isW[15] = "<is writing> ";
    while (TRUE)
    {
        if (readerNum > 0)
        {
            if (STRATEGY == 0)
            { // 读优先
                print("<");
                char tmp[2] = {readerNum_rf + '0', '\0'};
                print(tmp);
                print(isR);
            }
        }
    }
}

```

```

    }
    else
    {
        print("<");
        char tmp[2] = {readerNum + '0', '\0'};
        print(tmp);
        print(isR);
    }
}
else if (writerNum > 0)
{
    print(isW);
}
else
{
    continue;
}
sleep(TIME_SLICE);
}
}

```

3.1 读者优先

使用的信号量：

信号量	解释
readerLimit	限制同时读一本书的读者数量
mutex_readerNum	保护 readerNum 变量能正确进行加减操作
writeBlock	限制写进程

- 读函数

```

void READER_rf(char *name, int time_slice)
{
    P(&mutex_readerNum);
    if (readerNum == 0)
        P(&writeBlock); // 有读者，则禁止写
    readerNum++;
    V(&mutex_readerNum);

    P(&readerLimit);
    // 进行读，对写操作加锁
    readerNum_rf++;
    .....(print start and is)
    sleep(time_slice * TIME_SLICE);

    // 完成读
    .....(print finish)
}

```

```

    readerNum_rf--;

    P(&mutex_readerNum);
    readerNum--;
    if (readerNum == 0)
        V(&writeBlock); // 无读者，可写
    V(&mutex_readerNum);

    V(&readerLimit);
}

```

- 写函数

```

void WRITER_rf(char *name, int time_slice)
{
    P(&writeBlock);

    .....(print start)
    writerNum++;
    .....(print is)
    sleep(time_slice * TIME_SLICE);
    .....(print finish)
    writerNum--;

    V(&writeBlock);
}

```

3.2 写者优先

添加的信号量：

信号量	解释
readBlock	限制读进程，以实现写优先
mutex_writerNum	保护 writerNum 变量能正确进行加减操作

- 读函数

```

void READER_wf(char *name, int time_slice)
{
    P(&readerLimit);

    P(&readBlock);

    P(&mutex_readerNum);
    if (readerNum == 0)
        P(&writeBlock); // 有读者，则禁止写
    readerNum++;
    V(&mutex_readerNum);
}

```

```

V(&readBlock);

// 进行读，对写操作加锁
.....(print start and is)
sleep(time_slice * TIME_SLICE);

// 完成读
.....(print finish)
P(&mutex_readerNum);
readerNum--;
if (readerNum == 0)
    V(&writeBlock); // 无读者，可写
V(&mutex_readerNum);

V(&readerLimit);
}

```

- 写函数

```

void WRITER_wf(char *name, int time_slice)
{
    P(&mutex_writerNum);
    writerNum++;
    if (writerNum == 1)
        P(&readBlock); // 有写者，则禁止读
    V(&mutex_writerNum);

    // 开始写
    P(&writeBlock);
    .....(print start and is)
    sleep(time_slice * TIME_SLICE);

    // 完成写
    P(&mutex_writerNum);
    writerNum--;
    if (writerNum == 0)
        V(&readBlock); // 无写者，可读
    V(&mutex_writerNum);

    .....(print finish)

    V(&writeBlock);
}

```

3.3 读写公平（解决进程饿死问题）

添加的信号量：

信号量	解释
mutex_fair	实现读写公平

在读者优先的基础上，添加对 mutex_fair 信号量的 PV 操作，以实现读写公平

- 读函数

```
void READER_fair(char *name, int time_slice)
{
    // 开始读
    P(&mutex_fair);

    P(&readerLimit);
    P(&mutex_readerNum);
    if (readerNum == 0)
        P(&writeBlock);
    V(&mutex_fair);

    readerNum++;
    V(&mutex_readerNum);

    // 进行读，对写操作加锁
    .....(print start and is)
    sleep(time_slice * TIME_SLICE);

    // 完成读
    .....(print finish)
    P(&mutex_readerNum);
    readerNum--;
    if (readerNum == 0)
        V(&writeBlock);
    V(&mutex_readerNum);

    V(&readerLimit);
}
```

- 写函数

```
void WRITER_fair(char *name, int time_slice)
{
    P(&mutex_fair);
    P(&writeBlock);
    V(&mutex_fair);

    // 开始写
    .....(print start)
    writerNum++;
    .....(print is)
    sleep(time_slice * TIME_SLICE);
    // 完成写
    .....(print finish)
    writerNum--;

    V(&writeBlock);
}
```

