Week11 进程调度

一、 实验概述

在本次实验中，我们将介绍时钟中断，并且实现进程调度中的Round-Robin调度算法。

二、实验目的

1. 了解时钟中断
2. 了解进程调度的管理机制
3. 了解轮询调度算法的实现

三、实验项目整体框架概述

// Week11

```
├── kern
│   ├── debug
│   ├── driver
│   │   ├── clock.c
│   │   ├── clock.h
│   │   ├── console.c
│   │   ├── console.h
│   │   ├── ide.c
│   │   ├── ide.h
│   │   ├── intr.c
│   │   ├── intr.h
│   │   ├── kbdreg.h
│   ├── fs
│   │   ├── fs.h
│   │   ├── swapfs.c
│   │   └── swapfs.h
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   ├── process
│   │   ├── entry.S
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── switch.S
│   ├── schedule
│   │   ├── default_sched.c
│   │   ├── default_sched.h
│   │   ├── sched.c
│   │   └── sched.h
│   ├── sync
│   │   └── sync.h
```

```
│   ├── syscall
│   │   ├── syscall.c
│   │   └── syscall.h
│   └── trap
│       ├── trap.c
│       ├── trapentry.S
│       └── trap.h
├── libs
├── Makefile
├── tools
│   ├── function.mk
│   ├── kernel.ld
│   └── user.ld
└── user
    ├── hello.c
    ├── libs
    └── rr.c
```

四、实验内容

1. 初始化时钟中断
2. 初始化进程调度相关结构
3. 实现RR调度，并进行测试

五、实验过程概述及相关知识点

本次实验中，我们将会在上周代码的基础上实现按照时间片进行轮转调度的RR算法。

之前的调度算法中，我们是执行完一个进程之后，在do_exit()中调度另一个进程，没有实现基于时间片的各种调度算法。在本次实验的代码中，我们将启用时钟中断，时钟中断会按照设定的间隔时间触发中断，中断响应时调用调度算法从而实现基于时间片的进程调度。

第一步. 设置时钟中断

简单来说，时钟中断可以理解为每隔一段时间执行一次的程序。即每隔一段时间，会固定触发一次的中断。在时钟中断的处理时，我们可以完成进程调度，维护相关等操作。

时钟中断需要硬件支持。在ucore中实现时钟中断，我们需要使用：OpenSBI提供的`sbi_set_timer()`接口，我们可以通过这个接口传入一个时刻，在这个时刻会触发一次时钟中断。`rdtime`伪指令，读取一个叫做`time`的CSR的数值，表示CPU启动之后经过的真实时间。

需要注意的一点是，我们需要每隔一定时间就触发一次时钟中断，但是通过`sbi_set_timer()`接口每次只能设置一次时钟中断。所以我们在初始化的时候设置第一次时钟中断，在每一次时钟中断的处理中，设置下一次的时钟中断。

时钟中断的准备工作:

```
//libs/sbi.c

//当time寄存器的值为stime_value的时候触发一个时钟中断
void sbi_set_timer(unsigned long long stime_value) {
    sbi_call(SBI_SET_TIMER, stime_value, 0, 0);
}
```

```
// kern/driver/clock.c

//volatile告诉编译器这个变量可能在其他地方被瞎改一通，所以编译器不要对这个变量瞎优化
volatile size_t ticks;

//对64位和32位架构，读取time的方法是不同的
//32位架构下，需要把64位的time寄存器读到两个32位整数里，然后拼起来形成一个64位整数
//64位架构简单的一句rdtime就可以了
//__riscv_xlen是gcc定义的一个宏，可以用来区分是32位还是64位。
static inline uint64_t get_time(void) {//返回当前时间
#if __riscv_xlen == 64
    uint64_t n;
    __asm__ __volatile__("rdtime %0" : "=r"(n));
    return n;
#else
    uint32_t lo, hi, tmp;
    __asm__ __volatile__(
        "1:\n"
        "rdtimeh %0\n"
        "rdtime %1\n"
        "rdtimeh %2\n"
        "bne %0, %2, 1b"
        : "=&r"(hi), "=&r"(lo), "=&r"(tmp));
    return ((uint64_t)hi << 32) | lo;
#endif
}
```

时钟中断的初始化：

```
// Hardcode timebase
static uint64_t timebase = 100000;

void clock_init(void) {
    // sie这个CSR可以单独使能/禁用某个来源的中断。默认时钟中断是关闭
    的 // 所以我们要在初始化的时候，使能时钟中断
    set_csr(sie, MIP_STIP); // enable timer interrupt in
    sie //设置第一个时钟中断事件
    clock_set_next_event();
    // 初始化一个计数器，每次产生时钟中断tick+1
    ticks = 0;

    cprintf("setup timer interrupts\n");
}
//设置时钟中断：timer的数值变为当前时间 + timebase 后，触发一次时钟中断
void clock_set_next_event(void) { sbi_set_timer(get_time() + timebase); }
```

时钟中断的处理：

```
void interrupt_handler(struct trapframe *tf) {
    //...
    switch (cause) {
        //...
        case IRQ_S_TIMER:
            clock_set_next_event();//发生这次时钟中断的时候，我们要设置下一次时钟中断
            //完成进程调度等工作
            break;
        //...
}
```

第二步. 设置进程调度的相关数据结构

管理进程调度的抽象类

类似前面的实验，进程调度中我们也使用一个抽象类来完成的进程调度的管理。在这个类中我们需要定义以下数据结构和函数：

```
struct sched_class {
    // 调度类的名字
    const char *name;

    // 初始化函数，完成初始化运行队列的任务
    void (*init)(struct run_queue *rq);

    // 把进程放入运行队列
    void (*enqueue)(struct run_queue *rq, struct proc_struct *proc);

    // 把进程从运行队列中移除
    void (*dequeue)(struct run_queue *rq, struct proc_struct *proc);

    // 选择下一个要执行的进程
    struct proc_struct *(*pick_next)(struct run_queue *rq);

    // 每次时钟中断应该完成的操作
    void (*proc_tick)(struct run_queue *rq, struct proc_struct *proc);
};
```

运行队列

在调度类中，我们会有一个链表管理就绪的进程（之前我们的有一个管理所有进程的链表，这里是只管理可以运行的进程的链表），这个链表我们称之为运行队列。一个进程wakeup_proc()时，会被设置成Runnable并且被enqueue。当进程分到时间片开始运行时会被移除出队列。当进程时间片执行结束且进程状态仍为Runnable时会被重新加入队尾。

```
struct run_queue {
    // 保存着链表头指针
    list_entry_t run_list;

    // 运行队列中的线程数
    unsigned int proc_num;

    // 最大的时间片大小
    int max_time_slice;
};
```

进程控制块中的相关改动

我们在进程控制块中增加了指向运行队列的指针，存放在运行队列的表项以及进程剩余的时间片。

```
struct proc_struct {
    // ...
    // 表示这个进程是否需要调度
    volatile bool need_resched;

    // run queue的指针
    struct run_queue *rq;

    // 与这个进程相关的run queue表项
    list_entry_t run_link;

    // 这个进程剩下的时间片
    int time_slice;
};
```

第三步. RR算法的实现

RR调度算法非常简单。它为每一个进程维护了一个最大运行时间片。当一个进程运行够了其最大运行时间片那么长的时间后，调度器会把它标记为需要调度，并且把它的进程控制块放在队尾，重置其时间片。这种调度算法保证了公平性，每个进程都有均等的机会使用CPU。

`enqueue` 操作：RR算法把需要入队的进程放在调度队列的尾端，并且如果这个进程的剩余时间片为0（刚刚用完时间片被收回），则需要将剩余时间片设为最大时间片。

```
static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

`dequeue` 操作：将相应的项从队列中删除

```
static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}
```

`pick_next` 操作：选取队列头的表项，获得对应的进程控制块

```
static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

`proc_tick`函数：这个函数会在时钟中断时被调用。对当前正在运行的进程的剩余时间片减一。如果在减一后，如果剩余时间片为0，那么我们就把这个进程标记为“需要调度”，这样在中断处理结束之后，在内核会判断进程是否需要调度的时候，进行调度：

```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

第四步. 测试RR算法

在上一次的实验中，我们在user_main执行了一个简单的Hello程序，只进行了输出操作。在本次实验中，我们将执行另一个用户程序，命名为RR，在这个程序我们完成测试。在RR用户程序中，我们会执行fork系统调用，创建多个子进程，我们会控制每个进程的运行时间大于多个时间片，保证我们的测试效果。

实现流程：

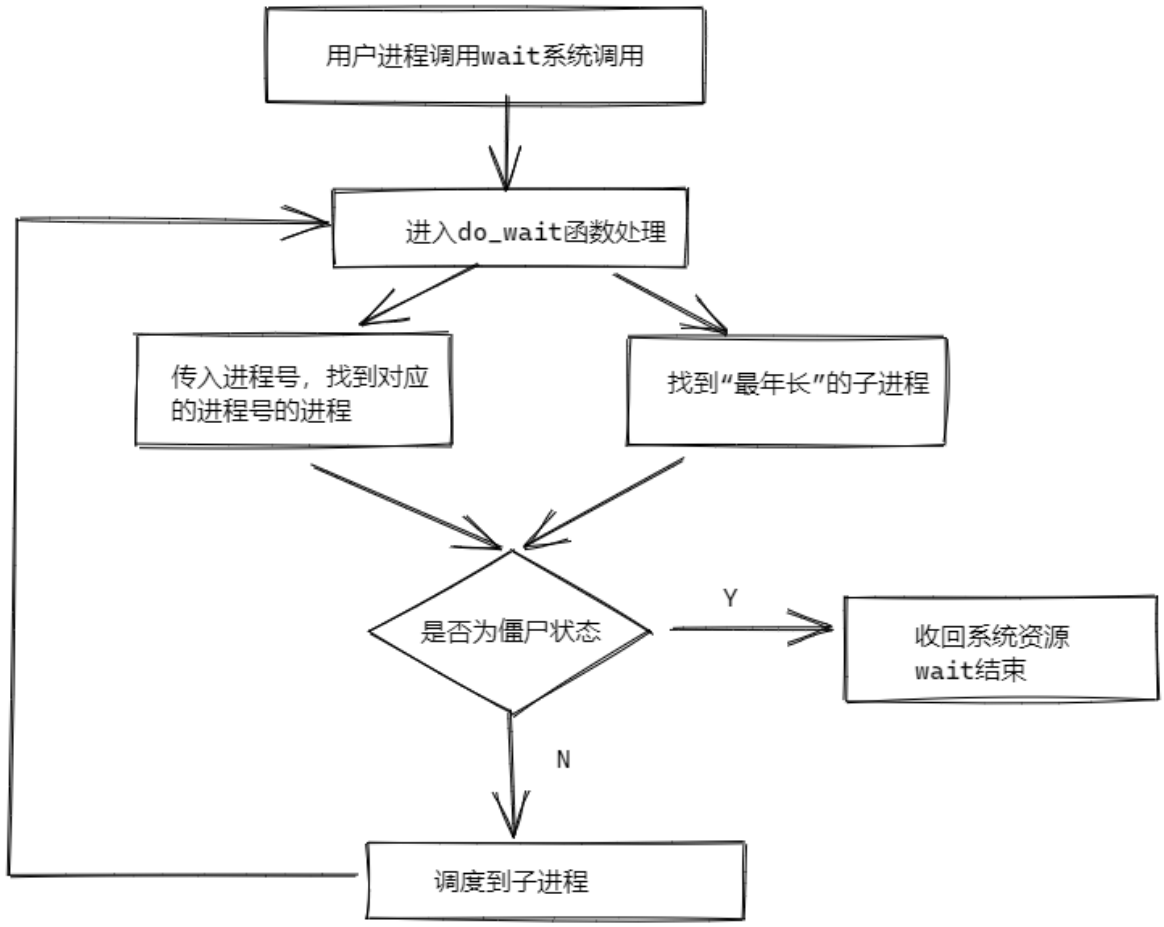
```
//用户程序中的主函数内容
int i,time;
memset(pids, 0, sizeof(pids));

for (i = 0; i < TOTAL; i ++) {
    acc[i]=0;
    if ((pids[i] = fork()) == 0) {
        acc[i] = 0;
        while (1) {
            spin_delay();//自旋延迟，就是进进行一些运算操作
            ++ acc[i];
            if(acc[i]%4000==0) {
                if((time=gettime_msec())>MAX_TIME) {
                    cprintf("child pid %d, acc %d, time
%d\n",getpid(),acc[i],time);
                    exit(acc[i]);
                }
            }
        }
    }
    if (pids[i] < 0) {
        goto failed;
    }
```

```
}

cprintf("main: fork ok,now need to wait pids.\n");

for (i = 0; i < TOTAL; i ++) {
    status[i]=0;
    waitpid(pids[i],&status[i]);
    cprintf("main: pid %d, acc %d, time %d\n",pids[i],status[i],gettime_msec());
}
cprintf("main: wait pids over\n");
return 0;
```

首先我们在user_main(在这里我们称之为父进程)中调用fork产生五个子进程，这里子进程的编号为3，4，5，6，7。注意，这个时候代码中的含fork语句的if条件为假，主进程会执行到`cprintf("main: fork ok,now need to wait pids.\n");`。

主进程会第一次调用waitpid，这时候主进程会找到pids[0]=3对应的进程，判断是否为僵尸状态，如果为僵尸状态则会收回他的资源，wait结束。如果不是僵尸状态，父进程进入休眠状态，调用schedule函数，调度到子进程。

这时候3号子进程开始执行，3号子进程会在fork语句的位置开始执行，此时fork返回值为0，这时候就会进入到循环了！由于我们的子进程运行的时间大于一个时间片，每次时钟中断剩余的时间片就会-1，如果为零，则会设置为需要调度的状态，在trap.c中的trap函数中会进行调度。这时候就调度到4号进程。重复这个过程。3，4，5，6，7，3，4，5，6，7，……（RR调度算法）。直到某一个进程执行结束，由于我们创建的子进程是相同的，这里是3号进程最先执行结束。进程最后会进入do_exit函数，设置自己的状态为僵尸状态，唤醒父进程，将父进程插入到运行队列，然后进行调度。这里调度到其他进程，要么执行一个时间片大小，要么进程执行结束，然后进行调度。经过几次调度，会调度到父进程，父进程这时候还在wait系统调用中，父进程会重复刚才的判断过程，这时候子进程为僵尸状态了，收回其资源，wait()结束。

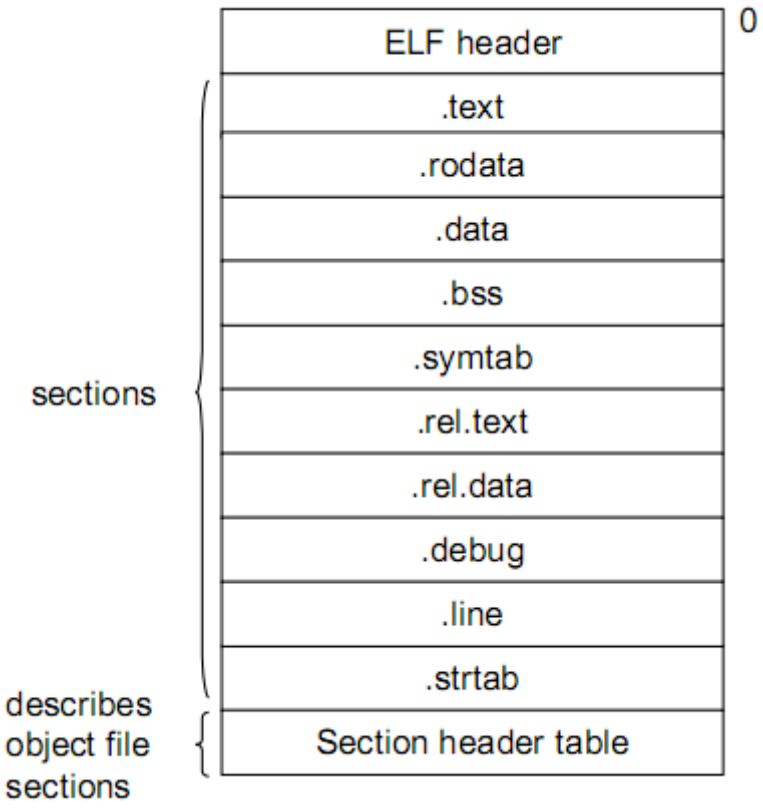


每次wait，进程会收回一个进程的资源。父进程会重复执行5次wait，直到收回所有子进程的资源，然后父进程返回，执行结束。

第五步. 补充内容：load_icode

在创建用户程序时，我们使用了load_icode加载了编译好的elf文件。下面我们来看一下这个函数的执行过程和作用。

elf文件的结构：



在load_icode函数中，我们会传入elf文件的起始位置和大小。elf文件的头是我们获取信息最重要的地方！

```
/* file header */
struct elfhdr {
    uint32_t e_magic;     // must equal ELF_MAGIC
    uint8_t e_elf[12];
    uint16_t e_type;      // 1=relocatable, 2=executable, 3=shared object,
4=core image
    uint16_t e_machine;   // 3=x86, 4=68K, etc.
    uint32_t e_version;   // file version, always 1
    uint64_t e_entry;     // entry point if executable
    uint64_t e_phoff;     // file position of program header or 0
    uint64_t e_shoff;     // file position of section header or 0
    uint32_t e_flags;     // architecture-specific flags, usually 0
    uint16_t e_ehsize;    // size of this elf header
    uint16_t e_phentsize; // size of an entry in program header
    uint16_t e_phnum;     // number of entries in program header or 0
    uint16_t e_shentsize; // size of an entry in section header
    uint16_t e_shnum;     // number of entries in section header or 0
    uint16_t e_shstrndx;  // section number that contains section name strings
};
```

我们会创建一个mm结构体，设置进程对应的根页表。接下来我们用强制类型转换获取到elf文件的头。利用elf文件头的内容，获取到程序头表。

> 程序头表 (program header table) 是一个结构体数组，数组中的每个结构体元素是一个程序头 (program header)，每个程序头描述一个段 (segment)。

```
/* program section header */
struct proghdr {
    uint32_t p_type;   // loadable code or data, dynamic linking info,etc.
    uint32_t p_flags;  // read/write/execute bits
    uint64_t p_offset; // file offset of segment
    uint64_t p_va;     // virtual address to map segment
    uint64_t p_pa;     // physical address, not used
    uint64_t p_filesz; // size of segment in file
    uint64_t p_memsz;  // size of segment in memory (bigger if contains bss）
    uint64_t p_align;  // required alignment, invariably hardware page size
};
```

第一步：验证elf的magic码。看一看是不是elf文件。

第二步：为每个程序头设置相关的标志位，建立vma并插入到mm中。

第三步：分配新的物理页面并建立相关映射，并将程序头指向的虚拟地址的内容复制到新分配的物理页面中。

第四步：如果存在bss段，则分配bss段的空间，并初始化为0。

第五步：建立用户栈的vma，并分配页面，建立映射

第六步：设置当前进程的mm为我们新分配的mm，设置根页表地址写入寄存器，初始化当前的进程的tf（设置为0），设置tf的sp指向用户栈，epc指向代码部分的起始的虚拟地址，设置tf中的sstatus寄存器。

注意：用户程序中的虚拟地址在生成elf文件的时候已经产生（这里可以看一下user.ld），我们所做的工作就是把分配物理页面，建立虚拟地址和我们新分配的物理页面的地址的映射，把elf文件的内容加载到我们分配的物理页面中。

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 如何设置时钟中断
2. 基于时间片的RR调度算法实现
3. load_icode

七、下一实验简单介绍

下一次实验，我们会了解同步互斥的相关内容。