

该实验仅涉及**进程**，不涉及**线程**

- 在Lab3中，寄存器状态保存在 `KSTACKTOP` 下的一个 `sizeof(TrapFrame)` 大小的区域里：`curenv -> env_tf = *((struct Trapframe *)KSTACKTOP - 1)`，每次被调度和被换下时进行交互
- 进程通过 `yield = 1` 主动让出 CPU 资源时，也要按照整块时间来算作其使用 CPU 的时间，不然如果运行一点就进行主动让出，相当于是“钻了空子”，不断消耗调度次数
- `env_sched_list` 中只保存了 `ENV_RUNNABLE` 的进程块，因此在 `sched.c` 中进行调度算法的时候无需对于已经阻塞的块进行队列的移除（已被其他代码处理）

## 进程

### 进程控制块

使用链表管理进程控制块数组 `envs`

- 调度队列 `env_sched_list`：进程创建时分配进程控制块并加入，在进程被释放时需要将其对应的进程控制块移出
- 空闲队列 `env_free_list`：将空闲的 Env 控制块按链表形式串联起来，一开始所有的进程控制块都是空闲的

调度队列采用双向链表 `TAILQ`，可以实现**正向和反向的遍历**

```
struct Env {
    struct Trapframe env_tf; // 保存当前进程的上下文信息对应的结构体
    Trapframe, 当发生进程调度或陷入内核时，会将当时的进程上下文环境保存在 env_tf
    变量中
    LIST_ENTRY(Env) env_link; // 空闲进程链表 env_free_list
    u_int env_id; // 进程独一无二的标识符
    u_int env_asid; // 记录进程的ASID，进程虚拟地址空间的标识
    u_int env_parent_id; // 父进程的进程 id
    u_int env_status; // 空闲状态；阻塞状态；执行或就绪状态
    Pde * env_pgdir; // 进程页目录的内核虚拟地址
    TAILQ_ENTRY(Env) env_sched_link; // 调度队列 env_sched_list
    u_int env_pri; // 进程的优先级
};
```

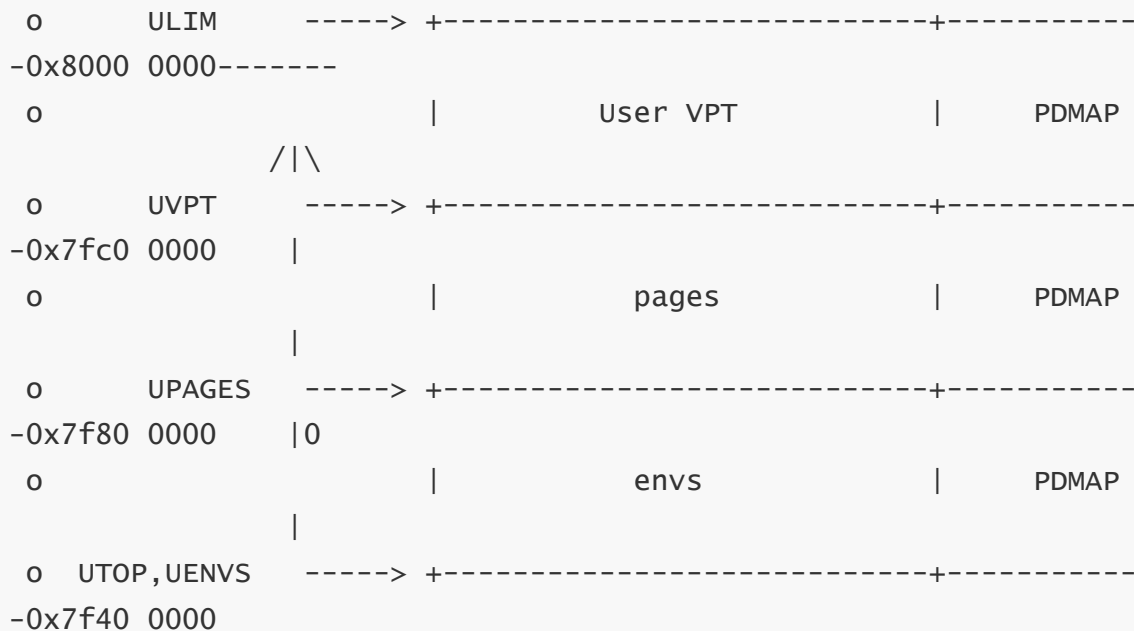
```

struct Trapframe {
    unsigned long regs[32];

    unsigned long cp0_status;
    unsigned long hi;
    unsigned long lo;
    unsigned long cp0_badvaddr;
    unsigned long cp0_cause;
    unsigned long cp0_epc;
}

```

## 进程创建过程



- **段地址映射**: 创建**模板页表**，插入内核中的 `pages` 和 `envs` 的映射（转化为物理地址），分别映射到用户空间的 `UPAGES` 和 `UENVS`（kuseg虚拟地址）
- **获取/设置进程控制块**: `env_alloc()`
- 获取空闲进程控制块 & 初始化进程控制块内容 & 设置 `env_tf` 寄存器
  - `env_setup_vm()`: 为进程申请自己的页目录 & 复制模板页表到申请的新页表 & 完成页目录自映射
- **加载二进制镜像**: 加载文件的ELF镜像 `load_icode()`
- **创建进程**: `env_create()`
- **进程运行与切换**: `env_run()`
  - 保存当前进程上下文（如果当前没有运行的进程跳过这一步）
  - 恢复要启动的进程的上下文

- 设置全局变量 `cur_pgdir`
- 调用 `env_pop_tf` 汇编函数，恢复现场、异常返回

## 异常

### CPU协处理器

中断/异常部分用到了CPU协处理器中的CP0

- **Status寄存器**：状态寄存器，包括中断引脚使能，其他 CPU 模式等 位域

31	28	27	26	25	24	23	22	21	20	19	18	17	16	15		8	7	6	5	4	3	2	1	0
CU3..CU0	RP	FR	RE	MX	PX	BEV	TS	SR	NMI	0	Impl				IM7..IM0		KX	SX	UX	UM	R0	ERL	EXL	IE
																					KSU			

- **Cause寄存器**：记录导致异常的原因

31	30		29		28		27			16	15		8	7	6			2	1	0
BD	0		CE				0			IP			0		ExcCode			0		

- **EPC寄存器**：异常结束后程序恢复执行的位置

### 异常分发 (`kern/entry.S`)

检测发生了哪种异常（通过异常向量组查找），并调用相应的异常处理程序

如CPU发生异常时（除TLB Miss异常），自动跳转到 `0x80000180`

一般来说，异常分发程序会被要求放在**固定的某个物理地址上**，以保证处理器能在检测到异常时正确跳转

### 时钟中断

硬件定时器产生**时钟中断**告知MOS某一个进程的时间片结束

- 通过异常分发，判断当前异常为中断异常，随后进入相应的中断处理程序，在MOS中为 `handle_int` 函数
- 在中断处理程序中进一步判断 Cause 寄存器中由几号中断位引发的中断，然后进入不同中断对应的中断服务程序
- 中断处理完成，通过 `ret_from_exception` 函数恢复现场，继续执行

### 进程调度 (`kern/sched.c`)

中断处理程序 `handle_int` 的最后，判断为**时钟中断**，调用 `schedule` 函数处理中断)

实验采用**时间片轮转算法**

MOS中的时间片长度是用**时钟中断**衡量的，即时间片被量化为  $N \times \text{TIMER\_INTERVAL}$ ，env 中的**优先级**即为这里的  $N$

当MOS记录到该进程发生了  $N$  个时钟中断时，MOS就知晓该进程的时间片结束了（每次时钟时间到了都要发出一个时钟中断）

### 进程切换情况

- 尚未调度过任何进程（`curenv = NULL`）
- 当前进程已经用完了时间片
- 当前进程不再就绪
- `yield` 参数指定必须发生切换

## 往年题

### Attention

- 对于调度队列，熟练使用 `TAILQ_FOREACH()`

```
struct Env * env_i;  
TAILQ_FOREACH(env_i, &env_sched_list, env_sched_link) {}
```

- 增加异常处理位

kern/genex.S 中建立异常处理宏 `BUILD_HANDLER ... do_...`

kern/traps.c 中添加外部引入函数以及添加异常处理位 `[10] = handler_...`

kern/traps.c 中实现异常处理函数

## 2024-exam

多了两个字段，`env_sched` 代表被选中上CPU的次数；`env_clocks` 代表进程实际消耗的时钟数

```
void schedule(int yield) {  
    static int count = 0; // remaining time slices of current env  
    struct Env *e = curenv;  
    if (curenv) {  
        curenv->env_clocks += ((struct Trapframe *)KSTACKTOP -  
1)->cp0_count;  
    }  
    count--;  
    if (yield || !count || e == NULL || e->env_status !=  
ENV_RUNNABLE) {  
        if (e != NULL && e->env_status == ENV_RUNNABLE) {
```

```

        TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
        TAILQ_INSERT_TAIL(&env_sched_list, e,
env_sched_link);
        e->env_scheds++;
    }
    if (TAILQ_EMPTY(&env_sched_list)) {
        panic("schedule: no runnable envs");
    }
    e = TAILQ_FIRST(&env_sched_list);
    count = e->env_pri;
}
env_run(e);
}

```

## 2024-extra

处理一些操作系统内部无法识别的指令

```

void do_ri(struct Trapframe *tf) {
    u_long va = tf->cp0_epc;
    Pte *pte;
    page_lookup(curenv->env_pgdir, va, &pte);
    u_long pa = PTE_ADDR(*pte) | (va & 0xfff);
    u_long kva = KADDR(pa);
    u_int *instr = (int *)kva;
    u_int code = (*instr) & (0x3ff);
    u_int *rs = &tf->regs[(*instr) >> 21 & (0x1f)];
    u_int *rt = &tf->regs[(*instr) >> 16 & (0x1f)];
    u_int *rd = &tf->regs[(*instr) >> 11 & (0x1f)];
    if (((*instr) >> 26) == 0) {
        if (code == 0x3f) {
            *rd = 0;
            for (int i = 0; i < 32; i += 8) {
                u_int rs_i = *rs & (0xff << i);
                u_int rt_i = *rt & (0xff << i);
                if (rs_i < rt_i) {
                    *rd = *rd | rt_i;
                } else {
                    *rd = *rd | rs_i;
                }
            }
        } else if (code == 0x3e) {
            u_int tmp = *((u_int *)*rs);
            if ((*((u_int *)*rs) == *rt) {
                *((u_int *)*rs) = *rd;
            }
        }
    }
}

```

```

    }
    *rd = tmp;
}
}

tf->cp0_epc += 4;
}

```

## 2023-exam

多了一个字段代表进程的使用用户

```

void schedule(int yield) {
    static int count = 0; // remaining time slices of current env
    struct Env *e = curenv;
    static int user_time[5] = {0};

    int has_ready_env[5] = {0};
    struct Env * env_i;
    TAILQ_FOREACH(env_i, &env_sched_list, env_sched_link) {
        has_ready_env[env_i->env_user] = 1;
    }

    if(yield || count <= 0 || e == NULL || e->env_status !=
ENV_RUNNABLE) {
        if(e != NULL && e->env_status == ENV_RUNNABLE) {
            TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
            TAILQ_INSERT_TAIL(&env_sched_list, e,
env_sched_link);
            user_time[e->env_user] += e->env_pri;
        }
        if(TAILQ_EMPTY(&env_sched_list)) {
            panic("schedule: no runnable envs\n");
        }

        int user_next = -1, min_time = 100000000;
        for(int i = 0; i < 5; i++) {
            if (has_ready_env[i] == 1 && user_time[i] < min_time)
            {
                user_next = i;
                min_time = user_time[i];
            }
        }
        assert(user_next != -1);
        TAILQ_FOREACH(env_i, &env_sched_list, env_sched_link) {

```

```

        if(env_i->env_user == user_next) {
            e = env_i;
            break;
        }
    }
    assert(e != NULL);
    count = e->env_pri;
}
count--;
env_run(e);
}

```

## 2023-extra

实现对于 `add & sub & addi` 的溢出异常处理

```

void do_ov(struct Trapframe *tf) {
    // printk("Got an ov\n");
    u_int badva = tf->cp0_epc;
    Pde *pgdir = curenv->env_pgdir;
    Pte **ppte = NULL;
    struct Page *p = page_lookup(pgdir, badva, ppte);
    // pgdir_walk(pgdir, badva, 0, &p);
    u_int badpa = page2pa(p) + (badva & 0xffff);
    u_int *badk0va = 0x80000000 badpa;
    u_int code = *badk0va;
    // printk("got code %x\n", code);
    if (code & 0x20000000) { // addi
        u_int s = (code & 0x3e00000) >> 21;
        u_int t = (code & 0x1f0000) >> 16;
        u_int svalue = tf->regs[s];
        u_int imm = code & 0xffff;
        tf->regs[t] = svalue / 2 + imm / 2;
        tf->cp0_epc += 4;
        // printk("inform:%x %d %d %x %x\n", code, s, t, svalue, tf-
>regs[t]);
        printk("addi ov handled\n");
    } else if ((code & 0x1f) == 0) { // add
        printk("add ov handled\n");
        *badk0va = code + 1;
    } else { // sub
        printk("sub ov handled\n");
        *badk0va = code + 1;
    }
}

```

```
curenv->env_ov_cnt++;  
}
```

## 上机准备

### EDF 算法

```
void schedule(int yield) {  
    static int system_tick = 0; // remaining time slices of  
    current env  
    struct Env *e = curenv;  
  
    struct Env *env_i;  
    u_int mn = -1;  
    TAILQ_FOREACH(env_i, &env_sched_list, env_sched_link) {  
        if(system_tick >= env_i -> begin_time &&  
            system_tick < env_i -> begin_time + env_i ->  
period &&  
            env_i -> run_time != 0) {  
                if(env_i -> begin_time + env_i -> period <  
mn) {  
                    e = env_i;  
                    mn = env_i -> begin_time + env_i ->  
period;  
                }  
            }  
        }  
  
    env_run(e);  
    e -> run_time -= 1;  
    if(e -> run_time == 0) e -> begin_time += e -> period;  
    system_tick += 1;  
}
```

对于未对齐的 `sw` 和 `lw` 指令（假设小端存储且 `offset = 0`）

（未进行内核虚拟地址转换）

```
void do_sl(struct Trapframe *tf) {  
    u_int bad_addr = tf->cp0_badvaddr; // 似乎也等于是 regs[base]  
    u_int epc = tf->cp0_epc;  
    u_int instr = *(u_int *)epc;  
    u_int opcode = (instr >> 26) & 0x3F; // 提取操作码  
    u_int base = (instr >> 21) & 0x1F; // 提取基址寄存器  
    u_int rt = (instr >> 16) & 0x1F; // 提取目标寄存器
```

```

u_int offset = instr & 0xFFFF;          // 提取偏移量(有符号扩展)

// 如果偏移量是负数(最高位为1), 则进行符号扩展
if (offset & 0x8000) {
    offset |= 0xFFFF0000;
}

// 检查是否为lw或sw指令
if (opcode == 0x23 || opcode == 0x2B) { // lw = 0x23, sw =
0x2B
    // 计算对齐后的地址(向下对齐到4字节边界)
    u_int aligned_addr = bad_addr & ~0x3;

    if (opcode == 0x23) { // 处理lw指令
        // 读取对齐后地址的值
        u_int data = *(u_int *)aligned_addr;

        // 根据未对齐地址的偏移量, 移位数据
        int shift = (bad_addr & 0x3) * 8;
        data = (data >> shift);

        // 将数据存储到目标寄存器
        tf->regs[rt] = data;
    } else { // 处理sw指令
        // 读取当前对齐地址的值
        u_int current = *(u_int *)aligned_addr;
        u_int data = tf->regs[rt];

        // 根据未对齐地址的偏移量, 合并数据
        int shift = (bad_addr & 0x3) * 8;
        u_int mask = 0xFFFFFFFF << shift;

        // 清除目标位置的原始数据, 然后写入新数据
        current &= ~mask;
        current |= (data << shift);

        // 写回对齐后的地址
        *(u_int *)aligned_addr = current;
    }

    // 跳过当前指令, 继续执行下一条指令
    tf->cp0_epc += 4;
} else {
    // 如果不是lw或sw指令, 则报错
    print_tf(tf);
}

```

```
panic("Address alignment exception for non-lw/sw  
instruction: opcode=%x", opcode);  
}  
}
```