# BUAA_OS_Lab3笔记

## 注意事项

进程控制块结构体中有2个链接节点，注意使用时区分！

```
LIST_ENTRY(Env) env_link;     // intrusive entry in 'env_free_list'
TAILQ_ENTRY(Env) env_sched_link;// intrusive entry in 'env_sched_list'
```

`env_status` 有三种状态：`ENV_FREE` `ENV_NOT_RUNNABLE` `ENV_RUNNABLE`，因此题目中说"不处于...“要使用 `!`。

调度队列 `env_sched_list` 和空闲队列 `env_free_list`。

使用现成函数时别把传入的参数 `(*pp)` 看成 `(**p)` ！

指针之间可以比大小，但指针和ulong类型比大小必须把ulong转化为void*

`if (e != NULL && e->env_status == ENV_RUNNABLE)` { 别忘了空指针判断！

## 调试

执行`make test lab=2_1 && make dbg` 进入特定测试点的测试环境；

`layout src` ： 按照**源代码**的文本布局对文件进行调试。

`tb mips_init`;

`c`;

从左向右分别为：

1. **继续**：对应 GDB 中的 `continue` 指令
2. **步过**：对应 GDB 中的 `next` 指令
3. **步入**：对应 GDB 中的 `step` 指令
4. **步出**：对应 GDB 中的 `finish` 指令
5. **重启**：对应 GDB 中的 `start` 指令（在程序运行中使用）
6. **停止**：对应 GDB 中的 `kill` 指令

使用 `layout asm` 显示选择 ASM 的 UI 方式（显示**汇编**代码），可以看到运行所在位置；

使用 `break _start` 在程序入口处（_start）位置打一个断点；

使用 `continue`，可以看到正常进入了程序入口。

`si` ： 按汇编指令进行单步调试。

`layout src` ： 按照**源代码**的文本布局对文件进行调试。但是由于 _start 本来就处于汇编代码中，因此在这里执行用源代码布局显示当然不会出现任何东西。因此，回车之后应该在有源代码（C语言）位置处再打一个断点：

`break mips_init`;

然后再 `continue`，就可以进入C语言源代码的 `mips_init` 实现。

# EDF

无周期和执行时间版本：

```
void schedule(int yield) {
    static int count = 0;
    struct Env *e = curenv;

    count--;

    // ---------------- EDF调度逻辑开始 ----------------
    struct Env *edf_env = NULL;
    u_int min_deadline = ~0;  // 初始化为最大值
    struct Env *iter;

    TAILQ_FOREACH(iter, &env_sched_list, env_sched_link) {
        if (iter->env_status == ENV_RUNNABLE && iter->env_deadline < min_deadline) {
            min_deadline = iter->env_deadline;
            edf_env = iter;
        }
    }

    if (edf_env != NULL && (e == NULL || edf_env->env_deadline < e->env_deadline)) {
        // 如果找到截止时间更早的任务，就直接运行它
        env_run(edf_env);
    }
    // ---------------- EDF调度逻辑结束 ----------------

    // ---------------- 原时间片轮转逻辑 ----------------
    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);
        }

        e = TAILQ_FIRST(&env_sched_list);
        if (e == NULL) {
            panic("schedule: no runnable envs\n");
        }

        count = e->env_pri;
        env_run(e);
    } else {
        env_run(e);
    }
}
```

有周期和执行时间版本：

```
struct Env {
    ...
    u_int env_exec_time;       // 当前周期下的剩余执行时间（初始化为 C_i）
    u_int env_period;          // 周期 T_i
    u_int env_relative_deadline; // 相对截止时间 D_i
    u_int env_deadline;        // 当前周期的绝对截止时间（通常等于 next_arrival + D_i）
    u_int env_next_arrival;    // 下次释放时间（arrival）
};
```

```
void schedule(int yield) {
    static int count = 0;
    struct Env *e = curenv;
    count--;

    struct Env *edf_env = NULL;
    u_int min_deadline = ~0;
    u_int now = get_current_time(); // 假设你有这个时间戳获取函数

    struct Env *iter;

    TAILQ_FOREACH(iter, &env_sched_list, env_sched_link) {
        // 任务释放时间到了，才可运行
        if (iter->env_status == ENV_RUNNABLE && now >= iter->env_next_arrival) {
            if (iter->env_deadline < min_deadline) {
                min_deadline = iter->env_deadline;
                edf_env = iter;
            }
        }
    }

    if (edf_env != NULL && (e == NULL || edf_env->env_deadline < e->env_deadline)) {
        edf_env->env_exec_time--;
        if (edf_env->env_exec_time == 0) {
            // 完成当前周期任务，准备下一周期
            edf_env->env_next_arrival += edf_env->env_period;
            edf_env->env_deadline = edf_env->env_next_arrival + edf_env->env_relative_deadline;
            edf_env->env_exec_time = edf_env->env_pri;  // 这里用 env_pri 做初始 C_i
        }
        env_run(edf_env);
    }

    // 原时间片轮转逻辑作为 fallback
    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);
        }

        e = TAILQ_FIRST(&env_sched_list);
```

```
        if (e == NULL) {
            panic("schedule: no runnable envs\n");
        }

        count = e->env_pri;
        env_run(e);
    } else {
        env_run(e);
    }
}

```

# 异常处理

中断处理的流程：`genex.S`中：

1. 通过异常分发，判断出当前异常为中断异常，随后进入相应的中断处理程序。在 MOS 中即对应 handle_int 函数。
2. 在中断处理程序中进一步判断 Cause 寄存器中是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函数。
3. 中断处理完成，通过 ret_from_exception 函数恢复现场，继续执行。

新增不对齐异常：

| 步骤 | 操作 |
| --- | --- |
| 1 | 获取不对齐的地址（BadVAddr） |
| 2 | 获取异常返回地址（EPC）并读出异常指令 |
| 3 | 分析该指令（假设是 `lw rt, offset(rs)`） |
| 4 | 判断是否为“立即数偏移未对齐”引发异常 |
| 5 | 对齐处理：将立即数调整为4字节对齐并重写指令 |
| 6 | 写回修正后的指令到 `epc` 地址处 |
| 7 | 异常返回，让它重新执行指令 |

当前通过 `BUILD_HANDLER` 宏注册了多个异常类型，我们只需再加一个用于 `adel`（`ExcCode = 4`）：

在你已有的 `.S` 文件中加上这一行：

```
BUILD_HANDLER reserved do_reserved
BUILD_HANDLER adel do_adel    # 👈 新增
```

然后注册处理函数到异常向量表，需要将 `ExcCode=4` 绑定为 `handle_adel`：

```
void (*exception_handlers[32])(void) = {
    [0 ... 31] = handle_reserved,
    [0] = handle_int,
    [2 ... 3] = handle_tlb,
    [4] = handle_adel,      // 👈 新增
#if !defined(LAB) || LAB >= 4
    [1] = handle_mod,
    [8] = handle_sys,
#endif
};
```

添加以下函数到 `trap.c`：

```
void do_adel(struct Trapframe *tf) {
    uint32_t badvaddr, epc, inst;

    // 1. 获取 BadVAddr 和 EPC
    asm volatile("mfc0 %0, $8" : "=r"(badvaddr));   // BadVAddr
    asm volatile("mfc0 %0, $14" : "=r"(epc));        // EPC

    // 2. 获取导致异常的原始指令（32位）
    inst = *(uint32_t *)epc;

    // 3. 分析 lw/sw 指令格式 (opcode: 0x23 = lw, 0x2B = sw)
    uint8_t opcode = inst >> 26;
    uint8_t rs = (inst >> 21) & 0x1f;
    uint8_t rt = (inst >> 16) & 0x1f;
    int16_t offset = inst & 0xffff;

    // 4. 获取 rs 寄存器的值
    uint32_t base = tf->regs[rs];

    uint32_t real_addr = base + (int32_t)offset;
    if (real_addr % 4 != 0) {
        // 5. 对齐 offset，使其变为 4 字节对齐（向下对齐）
        int16_t new_offset = (int16_t)((real_addr & ~0x3) - base);

        // 6. 构造新指令
        uint32_t new_inst = (opcode << 26) | (rs << 21) | (rt << 16) |
((uint16_t)new_offset);

        // 7. 写回修正指令（注意清除高位）
        *(uint32_t *)epc = new_inst;

        printk("Fixed unaligned %s at EPC=0x%08x (original addr=0x%08x →
aligned offset=%d)\n",
            (opcode == 0x23) ? "lw" : ((opcode == 0x2B) ? "sw" : "unknown"),
            epc, real_addr, new_offset);

        // 8. 恢复原执行，回到 EPC 继续执行修正后的指令
        return;
    }

    // 如果不是 lw/sw 或无法修正，还是 panic
```

```
    print_tf(tf);
    panic("AdEL: cannot recover instruction at 0x%08x", epc);
}
```

往年参考：

```
/*
 * Adel异常处理：lh替换lw，lb替换lh
 * 指令替换过程只修改指令 26-31 位，不修改寄存器编号和 offset 字段。
 * lb   100000, lh  100001, lw  100011
 */
 /*往异常向量组的偏移4位置上塞入handle_adel的地址（Adel的ExcCode = 4），具体操作为在lib/traps.c中加入
set_except_vector(4, handle_adel);
通过BUILD_HANDLER实现handle_adel函数，具体操作为在lib/genex.S中加入
BUILD_HANDLER adel do_adel cli
*/
/*根据前面的分析，我们知道异常处理函数handler_adel会先通过SAVE_ALL保存现场，
而对于非时钟中断的异常，会将sp指向KERNEL_SP，然后向下开出大小为TF_SIZE（sizeof(struct Trapframe)）的栈空间，
把寄存器的值保存进去（具体可以查看SAVE_ALL、get_sp的宏定义）。所以EPC也被保存到这里了！
EPC又刚好是发生异常的指令的地址，因此我们就可以通过EPC找到对应的指令了。找到这条指令后，修改其OpCode就不是难事了。
*/
void do_adel(char* sp) {// handler_adel把sp寄存器的值传给do_adel（BUILD_HANDLER中 move a0,sp j do_adel
    struct Trapframe *tf;
    tf = (struct Trapframe*)sp; // 找到保存所有寄存器的栈空间
    u_int instr;    // 对应指令
    u_int* instr_ptr;   //  对应指令的地址（PC）
    instr_ptr = tf->cp0_epc;    // 获取了epc
    instr = *instr_ptr; // 获取对应指令
    u_int opcode = instr & 0xFC000000;
    opcode = opcode >> 26;  // 获取opcode
    instr = instr & 0x03FFFFF; // 把instr的opcode先清0
    /* 开始处理opcode */
    if (opcode == 0b100011) { // lw 变成 lh
        opcode = 0b100001;
    }
    else { // lh 变成 lb
        opcode = 0b100000;
    }
    opcode = opcode << 26;
    instr = instr | opcode;
    *instr_ptr = instr; // 修改指令机器码
    return;
}
//注：需要增加：如果BD位为1，那么我们对应的lw/lh/lb指令的地址是EPC+4.

/*
处理OV异常：add，sub，addi溢出问题
*/

void do_ov(struct Trapframe *tf) {
    curenv->env_ov_cnt++;
```

```
46
47      u_long va = tf->cp0_epc; //va是异常指令的虚拟地址（用户虚拟地址）
48      Pte *pte;
49      page_lookup(curenv->env_pgdir, va, &pte);//通过查询curenv页表，获得页表项
50      u_long pa = PTE_ADDR(*pte) | (va & 0xfff);//由页表项获得物理地址
51      u_long kva = KADDR(pa);
52      //将物理地址转化至"kseg0 区间中对应的虚拟地址"（内核虚拟地址）
53
54      int *instr = (int *)kva;//内核虚拟地址可以直接访存获得指令
55      int code = (*instr)>>26;
56      int subcode = (*instr)&(0xf);
57
58      if (code == 0) {
59          if (subcode == 0) {
60              printk("add ov handled\n");
61          } else if (subcode == 2) {
62              printk("sub ov handled\n");
63          }
64          (*instr) = (*instr)|(0x1); //把指令换成addu或subu
65      } else {
66          tf->cp0_epc += 4;
67          printk("addi ov handled\n");
68          int reg_s = ((*instr)>>21) & (0x1f);
69          int reg_t = ((*instr)>>16) & (0x1f);
70          u_int imm = (*instr) & (0xffff);
71          tf->regs[reg_t] = tf->regs[reg_s]/(u_int)2 + imm/(u_int)2;
72      }
73      return;
74  }
75
76  /*
77  创建一个不涉及加载的fork函数，这个函数可以创建子进程，然后查找给定进程的父进程，和左右兄弟
    进程（可能没有父进程，也可能没有兄弟进程）
78  */
79  //改写env结构体
80  struct Env{
81      //省略
82      int son_num;            // add on exam
83          u_int son_id_arr[1024]; // add on exam
84  };
85  u_int fork(struct Env *e)
86  {
87          struct Env *e_son;
88          env_alloc(&e_son, e->env_id);
89          e_son->env_status = e->env_status;
90          e_son->env_pgdir = e->env_pgdir;
91          e_son->env_cr3 = e->env_cr3;
92          e_son->env_pri = e->env_pri;
93
94          // ---- father ----
95          int son_num = e->son_num;
96          e->son_id_arr[son_num] = e_son->env_id;
97          e->son_num += 1;
98
99          return e_son->env_id;
100 }
```

```c
//输出父和兄弟进程
void lab3_output(u_int env_id)
{
        struct Env *e_now;
        u_int fa_id = 0;
        u_int first_son_id = 0;
        u_int bro_bf_id = 0; // "bf" means "before"
        u_int bro_af_id = 0; // "af" means "after"

        envid2env(env_id, &e_now, 0);
        // son part
        first_son_id = e_now->son_id_arr[0];

        // parent part
        fa_id = e_now->env_parent_id;
        if (fa_id == 0) { // do not have parent
                // three 0 now
                bro_bf_id = 0;
                bro_af_id = 0;
        } else { // have a parent
                struct Env *e_fa;
                envid2env(fa_id, &e_fa, 0);
                int index = 0;
                for (index = 0; index < 1024; index++) {
                        if (e_fa->son_id_arr[index] == env_id) {
                                break;
                        } else {
                                continue;
                        }
                }
                // index is the env of father now
                if (index > 0) { // have bro bf
                        bro_bf_id = e_fa->son_id_arr[index - 1];
                }

                // have a bro_af
                if (e_fa->son_num > index + 1) {
                        bro_af_id = e_fa->son_id_arr[index + 1];
                }
        }
        // fa_id, fist_son_id, bro_bf, bro_af
        printk("%08x %08x %08x %08x\n", fa_id, first_son_id, bro_bf_id,
bro_af_id);
}
//实现函数：lab3_get_sum 函数，函数的功能为：给定一个进程的env_id，返回以该进程为根节点的子进程树中进程的数目（包括它本身）
int lab3_get_sum(u_int env_id)
{
        struct Env *e_now;
        envid2env(env_id, &e_now, 0);
        int son_num = e_now->son_num;
        // if e_now has no son
        if (son_num == 0) {
                return 1;
        } else {
                // have many sons, recuring
```

```
                int ans = 1;
                int i = 0;
                for (i = 0; i < son_num; i++) {
                        struct Env *e_son;
                        u_int son_id = e_now->son_id_arr[i];
                        envid2env(son_id, &e_son, 0); // now got a son
                        ans += lab3_get_sum(son_id);
                }
                return ans;
        }
}

/*
 * 调度问题，选择最佳时间片优先调度
*/
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 runnable_user[5] = {0};
    struct Env * ei;
    TAILQ_FOREACH(ei, &env_sched_list, env_sched_link) {
        runnable_user[ei->env_user] = 1;
    }
    if (yield || count <= 0 || e == NULL || e->env_status != ENV_RUNNABLE)
{
        if (e != NULL) {
            TAILQ_REMOVE(&env_sched_list, e, env_sched_link);
            if (e->env_status == ENV_RUNNABLE) {
                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");
        }

        int select_user = -1;
        for (int i = 0; i < 5; i++) {
            if (runnable_user[i]) {
                if (select_user == -1 || user_time[i] <
user_time[select_user]) {
                    select_user = i;
                }
            }
        }

        TAILQ_FOREACH(ei, &env_sched_list, env_sched_link) {
            if (ei->env_status == ENV_RUNNABLE && ei->env_user ==
select_user) {
                e = ei;
                break;
            }
        }
```

```
208             count = e->env_pri;
209         }
210         count--;
211         env_run(e);
212     }
```