

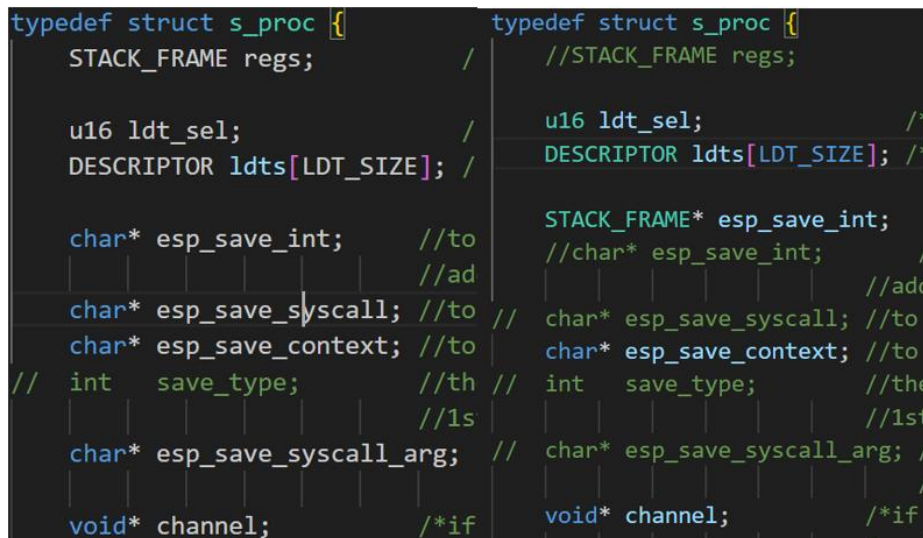
中断过程、进程管理和通讯的修改

路晨宇 2023.11.07

一.本次改动主要内容

1. 对 miniOS 的 PCB 中 REGS 字段的删除, 将 PCB 中的指针 esp_save_int 转换为 STACK_FRAME 型, 并将 REGS 字段的功能移植到指针 esp_save_int 中
2. 对 miniOS 的启动 kernel_main 中 initial_processes 的优化, 创建两个新的函数 init_process 和 init_reg。封装了 PCB 基本信息和内核栈的初始化。
3. 合并进程 PCB 中 char* esp_save_syscall、char* esp_save_syscall_arg 指针的作用, 并统一中断、系统调用和异常处理时, 内核栈的栈帧格式。现在中断、系统调用也需考虑错误码占位符(内核栈栈顶现在共 19 个值, 多了 ERROR)。
4. 优化中断和系统调用过程, 删除 kernel.asm 中 restart_int 和 restart_syscall, 并将其作用分别移入 hwint_maste/slave 和 sys_call 中
5. 删除 kernel.asm 中 switch_pde 函数的作用(对 cr3 的赋值), 其功能移植到 schedule (proc.c) 中, 现在减少了汇编与 C 语言的互相调用
6. 优化 restart_initial 启动过程, 现在将直接启动 initial 进程, 并且通过 pop context 和 pop frame 启动 initial 进程。

二. 具体改动说明



```
typedef struct s_proc {
    STACK_FRAME regs;
    u16 ldt_sel;
    DESCRIPTOR ldts[LDT_SIZE];
    char* esp_save_int;
    char* esp_save_syscall;
    char* esp_save_context;
    int save_type;
    char* esp_save_syscall_arg;
    void* channel;
}

typedef struct s_proc {
    //STACK_FRAME regs;
    u16 ldt_sel;
    DESCRIPTOR ldts[LDT_SIZE];
    STACK_FRAME* esp_save_int;
    //char* esp_save_int;
    char* esp_save_syscall;
    char* esp_save_context;
    int save_type;
    char* esp_save_syscall_arg;
    void* channel;
}
```

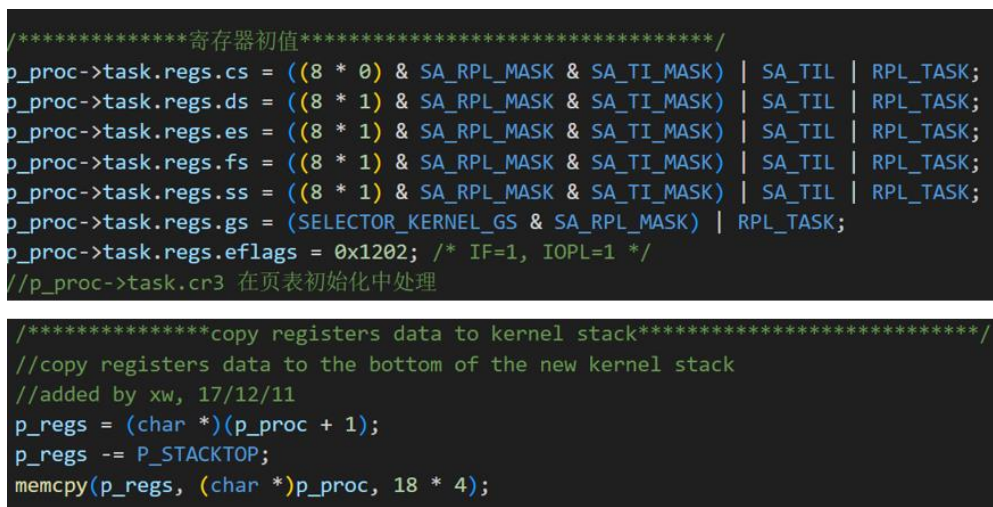
图 1 左图：原 PCB 右图：修改后 PCB

1. 对 miniOS 的 PCB 中 REGS 字段的删除

修改前：

REGS 字段主要用于对进程内核栈栈帧的赋值或修改。如 PCB 初始化、exec 系统调用、子进程和子线程创建时内核栈初始化、进程通讯 signal 等，具体举例如下：

在 main.c 的 initialize_processes() 函数中，在初始化所有进程的 PCB 时，会先对 REGS 字段赋值，再调用 memcpy 函数复制到进程的内核栈，如图 2。



```
/******寄存器初值******/
p_proc->task.regs.cs = ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->task.regs.ds = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->task.regs.es = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->task.regs.fs = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->task.regs.ss = ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK;
p_proc->task.regs.gs = (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK;
p_proc->task.regs.eflags = 0x1202; /* IF=1, IOPL=1 */
//p_proc->task.cr3 在页表初始化中处理

/******copy registers data to kernel stack******/
//copy registers data to the bottom of the new kernel stack
//added by xw, 17/12/11
p_regs = (char *) (p_proc + 1);
p_regs -= P_STACKTOP;
memcpy(p_regs, (char *) p_proc, 18 * 4);
```

图 2 修改前 initialize_processes() 函数中初始化内核栈截图

在其他函数中功能类似, 均是先修改 REGS 字段的值, 再 memcpy 到内核栈。

修改后:

更改指针 char *esp_save_int 为 STACK_FRAME *esp_save_int, 在 PCB 初始化时, 使得 STACK_FRAME *esp_save_int 指针指向内核栈, 直接用该结构体指针初始化内核栈栈帧。如图 3

```
p_regs = (char *) (p_proc + 1);
p_regs -= P_STACKTOP;
//memcpy(p_regs, (char *)p_proc, 18 * 4); //del

/*****some field about process switch***/
//初始化内核栈          added by lcy 2023.10.25
p_proc->task.esp_save_int = (STACK_FRAME*)p_regs;

PRIVATE void init_reg(PROCESS *proc,u32 cs,u32 ds,u32 es,u32 fs,u32 ss,u32 gs,u32 eflags,u32 esp,u32 eip){
    proc->task.esp_save_int->cs = cs;
    proc->task.esp_save_int->ds = ds;
    proc->task.esp_save_int->es = es;
    proc->task.esp_save_int->fs = fs;
    proc->task.esp_save_int->ss = ss;
    proc->task.esp_save_int->gs = gs;
    proc->task.esp_save_int->eflags =eflags;    /* IF=1, IOPL=1 */
    proc->task.esp_save_int->esp = esp;
    proc->task.esp_save_int->eip = eip;
}
```

图 3 修改后初始化 PCB 方式

在 exec 系统调用、子进程和子线程创建时内核栈初始化、进程通讯 signal 等改动类似, 现在可以直接用结构体指针初始化或修改相应的值。

2. 对 miniOS 的启动 kernel_main 中 initial_processes 的优化

在 main.c 中创建两个新的函数 init_process 和 init_reg。init_process 封装了 PCB 基本信息的初始化, 如 name、stat 等。init_reg 函数封装了 PCB 内核栈的初始化, 即用 STACK_FRAME *esp_save_int 指针初始化内核栈。函数原型如图 4

```

PRIVATE void init_process(PROCESS *proc, char name[32], enum proc_stat stat, int pid, int priority){
    strcpy(proc->task.p_name, name); //名称
    proc->task.pid = pid; //pid
    proc->task.stat = stat; //初始化状态 -1表示未初始化
    proc->task.ticks = proc->task.priority = priority; //时间片和优先级
    //proc->task.regs.eip = eip;
}

PRIVATE void init_reg(PROCESS *proc, u32 cs, u32 ds, u32 es, u32 fs, u32 ss, u32 gs, u32 eflags, u32 esp, u32 eip){
    proc->task.esp_save_int->cs = cs;
    proc->task.esp_save_int->ds = ds;
    proc->task.esp_save_int->es = es;
    proc->task.esp_save_int->fs = fs;
    proc->task.esp_save_int->ss = ss;
    proc->task.esp_save_int->gs = gs;
    proc->task.esp_save_int->eflags = eflags; /* IF=1, IOPL=1 */
    proc->task.esp_save_int->esp = esp;
    proc->task.esp_save_int->eip = eip;
}

```

图 4 init_process 和 init_reg 函数原型

修改后:

在 main.c 中添加宏定义:

```

#define k_cs ((8 * 0) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_ds ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_es ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_fs ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_ss ((8 * 1) & SA_RPL_MASK & SA_TI_MASK) | SA_TIL | RPL_TASK
#define k_gs (SELECTOR_KERNEL_GS & SA_RPL_MASK) | RPL_TASK

```

图 5 main.c 中用于 PCB 初始化的宏定义

在 PCB 初始化时, 调用 init_process 和 init_reg。其中宏定义用于 init_reg

```

//1>对前NR_TASKS个PCB初始化,且状态为READY(生成的进
/*****基本信息*****/
/*strcpy(p_proc->task.p_name, p_task->name); //
p_proc->task.pid = pid; //pid
p_proc->task.stat = READY; //状
p_proc->task.ticks = p_proc->task.priority = 1;
init_process(p_proc, p_task->name, READY, pid, 1);

//初始化内核栈 added by lcy 2023.10.25
p_proc->task.esp_save_int = (STACK_FRAME*)p_regs; //initialize esp_save_int, added by xw, 17/12/11
init_reg(p_proc, k_cs, k_ds, k_es, k_fs, k_ss, k_gs, 0x1202, (u32)StackLinBase, (u32)p_task->initial_eip);

```

图 6 修改后初始化 PCB 的两个函数

3. 合并进程 PCB 中 char* esp_save_syscall、char* esp_save_syscall_arg 指针。

PCB 中 esp_save_int、esp_save_syscall、esp_save_syscall_arg 三个指针原来的作用:

esp_save_int 在中断时的作用：在未发生中断嵌套时，记录每个进程从其内核栈进入中断栈时，内核栈的栈顶 esp 的值，用于从中断栈返回内核栈；在发生中断嵌套时，只记录第一个进入中断栈的进程内核栈 esp 的值，用于其返回内核栈。并且，在上述两种情况下 esp_save_int 指向内核栈栈顶的位置均是中断栈帧的位置，也用于进程从内核态返回用户态。

esp_save_syscall 的作用：1.记录系统调用时栈帧的位置，用于从内核态返回用户态。2.用于每个进程在内核态时对信号的处理，用于复制内核栈帧到其用户栈中，并修改内核栈栈帧中的 EIP 为 PCB 中_Handler 字段，修改 ESP 为用户栈栈顶，此后返回到用户态执行信号处理函数，当在用户态下执行完信号处理函数后，再通过 esp_save_syscall 恢复原内核栈中的数据。

esp_save_syscall_arg 的作用：在系统调用时，用于获取系统调用参数。

将三个指针合并的条件：在系统调用时用到指针，如果此时发生中断，则指针会被重新指向新的中断栈帧，因此系统调用相关功能会报错。故需取消 esp_save_int 在中断时的作用，这时问题便可解决。但中断返回时如何从中断栈回到内核栈，再从内核栈回到用户态呢？在进入中断时，用过 esi 寄存器记录内核栈栈顶的值，因此切换到中断栈后 push esi，这样从中断栈返回时，pop esp 便可回到内核栈，而且同样是中断栈帧的位置，与之前使用指针的作用相同。此时在中断过程不需要指针的作用了，唯一的指针可以只在系统调用时使用，并且不需要担心指针会被重新指向。指针自始至终都会指向每个进程的内核栈栈顶中断栈帧的位置。因此所有功能便可由 STACK_FRAME* esp_save_int 一个指针实现。

将 esp_save_syscall、esp_save_syscall_arg 两个指针合并的条件：只需将 esp_save_syscall_arg 的作用移植到 esp_save_syscall 中便可。

目前修改的，已经将 `esp_save_syscall`、`esp_save_syscall_arg` 两个指针合并，并统一了中断、系统调用和异常处理时，内核栈的栈帧格式。现在中断、系统调用也需考虑错误码占位符。栈帧保持了一致。

4. 优化中断和系统调用过程，删除 `kernel.asm` 中 `restart_int` 和 `restart_syscall`，并将其作用分别移入 `hwint_maste/slave` 和 `sys_call` 中。

修改前系统调用的调用逻辑：

在 `sys_call` 中首先 `call save_syscall`（压栈返回地址 `retaddr`）。在 `save_syscall` 保存了栈帧后，`push restart_syscall`，然后通过宏定义计算出 `retaddr` 在内核栈的偏移量，从而 `jmp` 回 `sys_call`。紧接着在 `sys_call` 中调用完具体的系统调用处理函数后，`ret` 回 `restart_syscall`。`restart_syscall` 里使得 `esp` 指向从内核栈返回的栈帧位置后，调用 `sched`。等到该进程下一次被调度上 CPU 执行后，继续执行 `restart_syscall` 的最后一句 `jmp restart_restore`，从而准备从内核态返回用户态。

修改后系统调用的调用逻辑：

在 `sys_call` 中将错误码占位符入栈后，`call save_syscall`（压栈返回地址 `retaddr`）。在 `save_syscall` 保存了栈帧后直接 `jmp` 回 `sys_call`。在 `sys_call` 中调用完具体的系统调用处理函数后，使得 `esp` 指向从内核栈返回的栈帧位置，然后调用 `sched`。等到该进程下一次被调度上 CPU 执行，继续执行 `sys_syscall` 的最后一句 `jmp restart_restore`，从而准备从内核态返回用户态。

修改前中断的调用逻辑：

在 `hwint_maste/slave` 中，`call save_int`（压栈返回地址 `retaddr`），在 `save_int` 保存了栈帧后：若是第一次进入中断，则需要使 PCB 中 `esp_save_int` 指针指向此时内核栈中断栈帧，然后切换到中断栈，通过指令 `push restart_int` 将 `restart_int`

地址压入中断栈栈顶，然后 jmp 回 hwint_maste/slave；若已处于中断栈中（中断嵌套），则跳转到 instack: 通过 push restart_restore 指令将 restart_restore 压入中断栈，然后 jmp 回 hwint_maste/slave。在 hwint_maste/slave 中调用完具体的中断处理函数后，若是第一次中断，则 ret 回 restart_int，使得 esp 指向从内核栈返回的栈帧位置，然后调用 sched。等到该进程下一次被调度上 CPU 执行，继续执行 restart_int 的最后一句 jmp restart_restore，从而准备从内核态返回用户态；若是嵌套中断，则直接 ret 回 restart_restore，直接从中断栈返回。

注：中断嵌套时，进程 PCB 的 esp_save_int 指针只记录第一次中断时，需要返回内核栈的位置。而后续嵌套中断并没有使用 esp_save_int 指针，并且后续嵌套中断处理完时，不会调用 sched，而是直接 resart_restore,在中断栈上返回。

修改后中断的调用逻辑：

在 hwint_maste/slave 中，call save_int（压栈返回地址 retaddr），在 save_int 保存了栈帧后：若是第一次进入中断，则需要使 esi 指向此时内核栈中断栈帧，然后切换到中断栈，通过指令 push esi 将内核栈栈顶的值压入中断栈栈顶，然后 jmp 回 hwint_maste/slave；若已处于中断栈中（中断嵌套），则跳转到 instack:直接 jmp 回 hwint_maste/slave。在 hwint_maste/slave 中调用完具体的中断处理函数后，若是第一次中断，则 pop esp 使得 esp 指向之前 esi 保存的从内核栈返回的栈帧位置，然后调用 sched。等到该进程下一次被调度上 CPU 执行，继续执行 hwint_maste/slave 的最后一句 jmp restart_restore，从而准备从内核态返回用户态；若是嵌套中断，则直接跳到 restart_restore，直接从中断栈返回。

5.删除 kernel.asm 中 switch_pde 函数的作用(对 cr3 的赋值), 其功能移植到 schedule (proc.c) 中, 现在减少了汇编与 C 语言的互相调用

修改前: renew_env 中, 首先会调用 call switch_pde, 从已经选好上 cpu 运行的进程的 PCB 中, 取出 CR3——页目录基址寄存器的值, 赋值给全局变量 cr3_ready, 然后返回到 renew_env, 将 cr3_ready 赋值给 cr3 寄存器。

修改后: 在 schedule 中选好下一个被调度上 CPU 运行的进程后, 便将其 CR3 的值赋值给 cr3_ready。此后仍然是在执行 renew_env 时, 将 cr3_ready 赋值给 cr3 寄存器。因此并没有改变 cr3 寄存器被赋值的时机, 只改变了从 PCB 中取出 CR3 字段的时机。

6.优化 restart_initial 启动过程,现在将直接启动 initial 进程, 并且通过 pop context 和 pop frame 启动 initial 进程。

现在在 kernel_main 中, 执行 restart_initial 之前, p_proc_current 直接指向 initial 进程。并且 restart_initial 将通过 pop context, 执行 ret 返回 restart_restore。而修改之前, 直接 jmp 到 restart_restore。如图 8

```
// p_proc_current = proc_table;  
p_proc_current = &proc_table[16];  
cr3_ready=p_proc_current->task.cr3;  
kernel_initial = 0; //kernel initialization is done. added by xw, 18/5/31
```

图 7 修改后的 kernel_main 中, 执行 restart_initial 之前的工作

<pre>restart_initial: ; mov eax, [p_proc_current] ; lldt [eax + P_LDT_SEL] ; lea ebx, [eax + INIT_STACK_SIZE] ; mov dword [tss + TSS3_S_SP0], ebx call renew_env mov eax, [p_proc_current] mov esp, [eax + ESP_SAVE_INT] jmp restart_restore</pre>	<pre>restart_initial: ; mov eax, [p_proc_current] ; lldt [eax + P_LDT_SEL] ; lea ebx, [eax + INIT_STACK_SIZE] mov dword [tss + TSS3_S_SP0], ebx call renew_env mov ebx, [p_proc_current] mov esp, [ebx + ESP_SAVE_CONTEXT] popad popfd ret</pre>
--	--

图 8 左图为修改前 右图为修改后