Week9 内核进程

一、 实验概述

在本次实验中，我们将实现内核进程管理的相关机制。

二、实验目的

1. 了解内核进程创建、执行的管理过程
2. 了解内核进程的切换和基本调度过程

三、实验项目整体框架概述

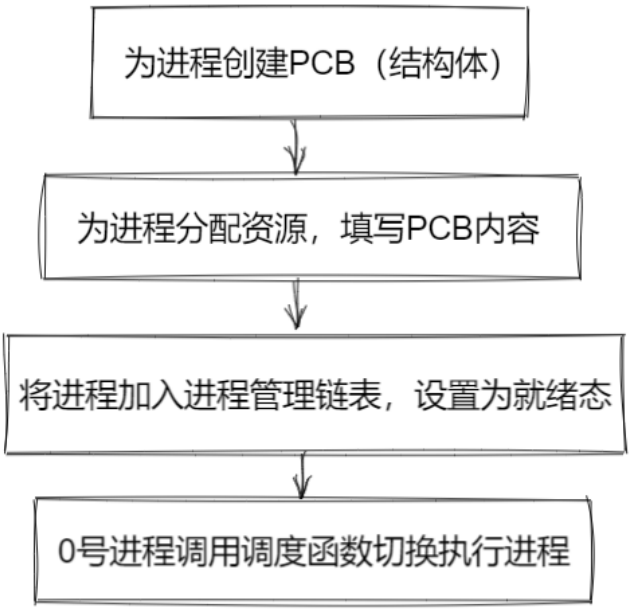
```
// Week9
├── kern
│   ├── debug
│   ├── driver
│   ├── fs
│   ├── init
│   │   ├── entry.S
│   │   └── init.c
│   ├── libs
│   ├── mm
│   ├── process
│   │   ├── entry.S
│   │   ├── proc.c
│   │   ├── proc.h
│   │   └── switch.S
│   ├── schedule
│   │   ├── sched.c
│   │   └── sched.h
│   ├── sync
│   └── trap
│       ├── trap.c
│       ├── trapentry.S
│       └── trap.h
├── libs
├── Makefile
└── tools
```

四、实验内容

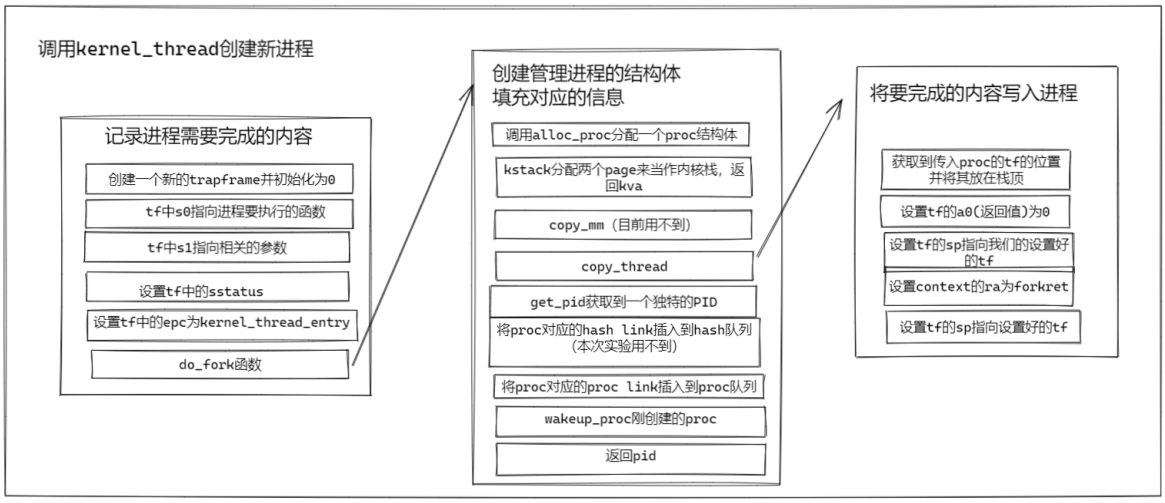
1. 理解进程控制块的定义。
2. 创建内核进程
3. 完成内核进程切换

五、实验过程概述

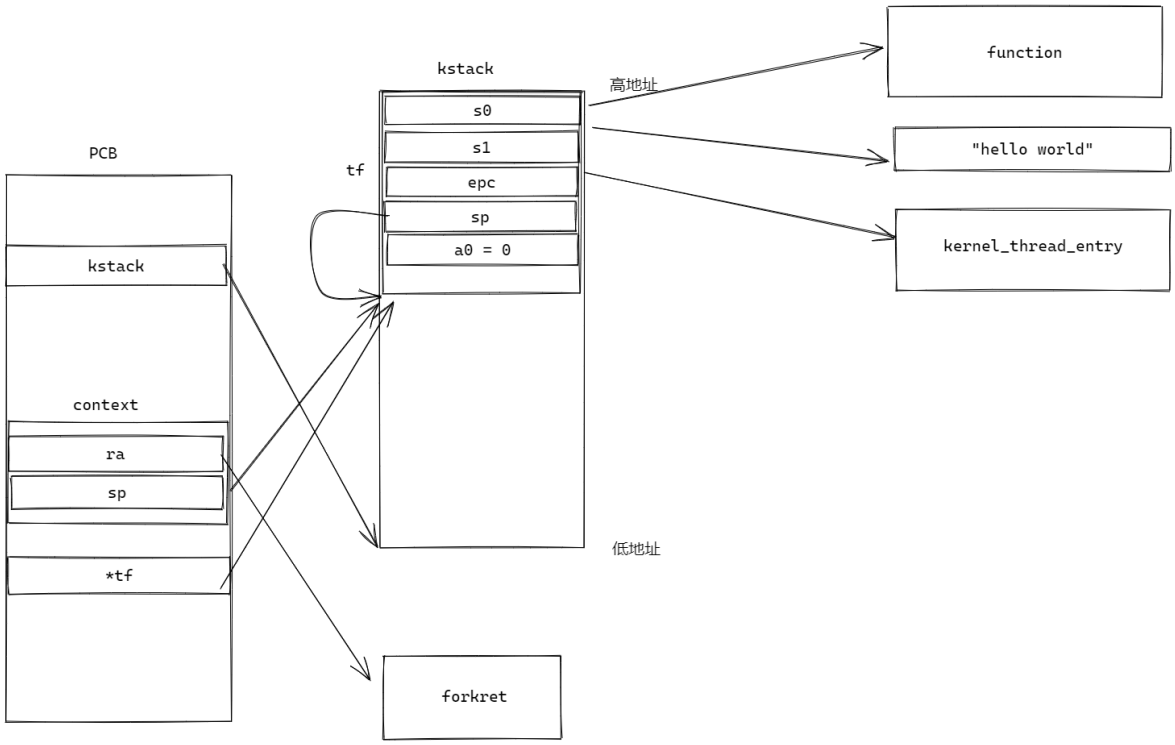
进程执行流程：



创建进程：

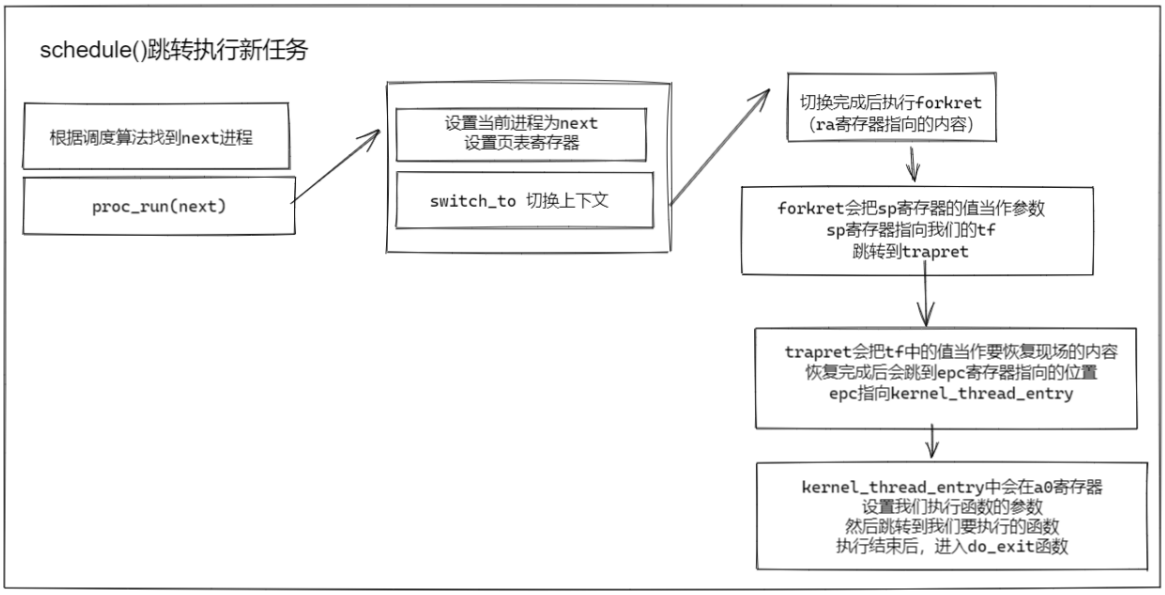


创建完之后的进程状态：



> ra: 当调用ret指令时（switch_to中调用），实际指令为jalr x0, 0(x1)，其中x1即ra寄存器，此指令会使得pc内写入ra的值，从而下一步跳转至ra的存储的指令地址执行。
>
> epc: 调用sret指令时（__trapret中调用），会跳转至epc所指向的函数。

切换流程：



六、实验流程及相关知识点

第一步. 了解进程相关的两个数据结构

在本章实现的进程管理模型中，我们主要维护两个数据结构：进程控制块和进程上下文。进程控制块维护进程的各个信息，包括内存映射，进程名等。进程上下文里面保存了和进程运行状态相关的各个寄存器的值，目的是为了恢复进程运行状态。

进程控制块PCB

我们在ucore中使用结构体`struct proc_struct`来保存和进程相关的控制信息。

`struct proc_struct`内部结构如下：

```
struct proc_struct {
    enum proc_state state;                      // Process state
    int pid;                                    // Process ID
    int runs;                                   // the running times of Proces
    uintptr_t kstack;                           // Process kernel stack
    volatile bool need_resched;                 // bool value: need to be
rescheduled to release CPU?
    struct proc_struct *parent;                 // the parent process
    struct mm_struct *mm;                       // Process's memory management field
    struct context context;                     // Switch here to run process
    struct trapframe *tf;                       // Trap frame for current interrupt
    uintptr_t cr3;                              // CR3 register: the base addr of
Page Directroy Table(PDT)
    uint32_t flags;                             // Process flag
    char name[PROC_NAME_LEN + 1];               // Process name
    list_entry_t list_link;                     // Process link list
    list_entry_t hash_link;                     // Process hash list
};
```

这里面值得我们关注的主要有以下几个成员变量：

- `parent`：里面保存了进程的父进程的指针。在内核中，只有内核创建的idle进程没有父进程，其他进程都有父进程。进程的父子关系组成了一棵进程树，这种父子关系有利于维护父进程对于子进程的一些特殊操作。
- `mm`：这里面保存了内存管理的信息，包括内存映射，虚存管理等内容。具体内在实现可以参考之前的章节。
- `context`：`context`中保存了进程执行的上下文，也就是几个关键的寄存器的值。这些寄存器的值用于在进程切换中还原之前进程的运行状态（进程切换的详细过程在后面会介绍）。切换过程的实现在`kern/process/switch.S`。
- `tf`：`tf`里保存了进程的中断帧。当进程从用户空间跳进内核空间的时候，进程的执行状态被保存在了中断帧中（注意这里需要保存的执行状态数量不同于上下文切换）。系统调用可能会改变用户寄存器的值，我们可以通过调整中断帧来使得系统调用返回特定的值。
- `cr3`：`cr3`寄存器是x86架构的特殊寄存器，用来保存页表所在的基址。出于legacy的原因，我们这里仍然保留了这个名字，但其值仍然是页表基址所在的位置

进程上下文context

进程上下文使用结构体`struct context`保存，其中包含了`ra`，`sp`，`s0~s11`共14个寄存器。

这里为什么不需要保存所有的寄存器呢？这里我们巧妙地利用了编译器对于函数的处理。我们知道寄存器可以分为调用者保存（caller-saved）寄存器和被调用者保存（callee-saved）寄存器。因为我们在一个函数中进行线程切换，所以编译器会自动帮助我们生成保存和恢复调用者保存寄存器的代码。在进程切换过程中我们只需要保存被调用者保存寄存器就好啦！

> **调用者保存寄存器（caller saved registers）**

> 也叫**易失性寄存器**，在程序调用的过程中，这些寄存器中的值不需要被保存（即压入到栈中再从栈中取出），如果某一个程序需要保存这个寄存器的值，需要调用者自己压入栈；
>
> **被调用者保存寄存器（callee saved registers）**
>
> 也叫**非易失性寄存器**，在程序调用过程中，这些寄存器中的值需要被保存，不能被覆盖；当某个程序调用这些寄存器，被调用寄存器会先保存这些值然后再进行调用，且在调用结束后恢复被调用之前的值；
>
> 举个例子，fun1会调用fun2，寄存器A需要在调用A前后保持一致
>
> A是调用者保存寄存器：
>
> ```
> fun1(){
>     save A;
>     fun2();
>     restore A;
> }
>
> fun2(){
>     ...
> }
> ```
>
> A是被调用者保存寄存器
>
> ```
> fun1(){
>     fun2();
> }
>
> fun2(){
>     save A;
>     ...
>     restore A;
> }
> ```

第二步. 内核进程idle

当我们的操作系统开始运行的时候，其实它已经可以被视作一个进程了。但是我们还没有为他设计好进程控制块，也就没法进行管理。这里使用0号进程idle来表示这个进程。

我们在文件`kern/process/proc.c`中的`proc_init`创建了第一个进程idle。

```
// kern/process/proc.c
// proc_init - set up the first kernel process idleproc "idle" by itself and
//           - create the second kernel process init_main
void
proc_init(void) {
    int i;

    list_init(&proc_list);//进程链表
    for (i = 0; i < HASH_LIST_SIZE; i ++) {
        list_init(hash_list + i);
    }
```

```
    if ((idleproc = alloc_proc()) == NULL) { //分配"第0个"进程 idle
        panic("cannot alloc idleproc.\n");
    }

    idleproc->pid = 0;
    idleproc->state = PROC_RUNNABLE;
    idleproc->kstack = (uintptr_t)bootstack;
    idleproc->need_resched = 1;
    set_proc_name(idleproc, "idle");
    nr_process ++;
    //全局变量current保存当前正在执行的进程
    current = idleproc;

    //...
}
```

在初始化时，首先需要初始化进程链表。进程链表就是把所有进程控制块串联起来的数据结构，可以记录和追踪每一个进程。然后，调用`proc_alloc`函数来为第一个进程分配其进程控制块。`proc_alloc`函数会使用`kmalloc`分配一段空间来保存进程控制块，并且设定一些初值告诉我们这个进程目前还在初始化中。

在分配完空间后，我们对于`idle`进程的控制块进行一定的初始化：

```
idleproc->pid = 0;
idleproc->state = PROC_RUNNABLE;
idleproc->kstack = (uintptr_t)bootstack;
idleproc->need_resched = 1;
set_proc_name(idleproc, "idle");
nr_process ++;
```

从这里开始，`idle`进程具有了合法的进程编号，`0`。我们把`idle`进程的状态设置为`RUNNABLE`，表示其可以执行。因为这是第一个内核进程，所以我们可以直接将ucore的启动栈分配给他。需要注意的是，后面再分配新进程时我们需要为其分配一个栈，而不能再使用启动栈了。我们再把`idle`进程标志为需要调度，这样一旦`idle`进程开始执行，马上就可以让调度器调度另一个进程进行执行。

第三步. 创建一个内核进程

接下来我们来创建一个的内核进程。在本次实验中，我们先实现一个输出`Hello World`的进程。下面是创建内核进程的代码：

```
int pid = kernel_thread(init_main, "Hello world!!", 0);
if (pid <= 0) {
panic("create init_main failed.\n");
}

initproc = find_proc(pid);
set_proc_name(initproc, "init");
```

我们使用 kernel_thread函数来创建一个进程，`init_main`是我们要执行的函数，"Hello world!!"是我们要传入的参数。

```
// kernel_thread - create a kernel thread using "fn" function
```

```
// NOTE: the contents of temp trapframe tf will be copied to
//       proc->tf in do_fork-->copy_thread function
int
kernel_thread(int (*fn)(void *), void *arg, uint32_t clone_flags) {
    struct trapframe tf;
    memset(&tf, 0, sizeof(struct trapframe));

    tf.gpr.s0 = (uintptr_t)fn;
    tf.gpr.s1 = (uintptr_t)arg;
    tf.status = (read_csr(sstatus) | SSTATUS_SPP | SSTATUS_SPIE) & ~SSTATUS_SIE;
    tf.epc = (uintptr_t)kernel_thread_entry;
    return do_fork(clone_flags | CLONE_VM, 0, &tf);
}
```

我们将寄存器 `s0` 和 `s1` 分别设置为需要进程执行的函数和相关参数列表，之后设置了 `status` 寄存器使得进程切换后处于中断使能的状态。我们还设置了 `epc` 使其指向 `kernel_thread_entry`，这是进程执行的入口函数。`epc` 寄存器记录了中断恢复现场之后执行的指令的位置，在这里，我们进行进程切换时，会执行恢复现场操作，完成进程的切换，在下面会介绍这个过程。

最后，调用 `do_fork` 函数把当前的进程复制一份。

```
// do_fork -  parent process for a new child process
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf) {
    int ret = -E_NO_FREE_PROC;
    struct proc_struct *proc;
    if (nr_process >= MAX_PROCESS) {
        goto fork_out;
    }
    ret = -E_NO_MEM;

    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }
    if ((ret = setup_kstack(proc)) == -E_NO_MEM) {
        goto bad_fork_cleanup_proc;
    }
    copy_mm(clone_flags, proc);//这个函数本次实验中没有做什么。

    copy_thread(proc, stack, tf);

    const int pid = get_pid();
    proc->pid = pid;
    list_add(hash_list + pid_hashfn(pid), &(proc->hash_link));
    list_add(&proc_list, &(proc->list_link));
    nr_process++;

    wakeup_proc(proc);
    ret = pid;
fork_out:
    return ret;

bad_fork_cleanup_kstack:
    put_kstack(proc);
bad_fork_cleanup_proc:
```

```
    kfree(proc);
    goto fork_out;
}
```

`do_fork`函数内部主要进行了如下操作：

1. 分配并初始化进程控制块（`alloc_proc`函数）
2. 分配并初始化内核栈（`setup_stack`函数）
3. 根据`clone_flags`决定是复制还是共享内存管理系统（`copy_mm`函数）
4. 设置进程的中断帧和上下文（`copy_thread`函数）
5. 把设置好的进程加入链表
6. 将新建的进程设为就绪态
7. 将返回值设为线程id

```
// copy_thread - setup the trapframe on the  process's kernel stack top and
//             - setup the kernel entry point and stack of process
static void
copy_thread(struct proc_struct *proc, uintptr_t esp, struct trapframe *tf) {
    proc->tf = (struct trapframe *)(proc->kstack + KSTACKSIZE - sizeof(struct
trapframe));
    *(proc->tf) = *tf;

    // Set a0 to 0 so a child process knows it's just forked
    proc->tf->gpr.a0 = 0;
    proc->tf->gpr.sp = (esp == 0) ? (uintptr_t)proc->tf : esp;

    proc->context.ra = (uintptr_t)forkret;
    proc->context.sp = (uintptr_t)(proc->tf);
}
```

在这里我们首先在上面分配的内核栈上分配出一片空间来保存`trapframe`。然后，我们将`trapframe`中的`a0`寄存器（返回值）设置为0，说明这个进程是一个子进程。之后我们将上下文中的`ra`设置为了`forkret`函数的入口，并且把`trapframe`放在上下文的栈顶。

第四步. 进程切换

我们已经把创建好的内核进程放在了队列中，在内核初始化一些功能之后，会进入`cpu_idle()`函数。

```
void
cpu_idle(void) {
    while (1) {
        if (current->need_resched) {
            schedule();
        }
    }
}
```

当前的current进程为idle进程。这时，它会陷入死循环，不断检查自己是否需要调度：因为我们之前在初始化中把`idle`进程的`need_resched`设为了`1`，所以它总会调用`schedule`函数来检查是否有进程可以调度。由于我们已经创建了另外一个内核进程，所以会调度到这个进程。

我们实现的`schedule`函数非常的简单：当需要调度的时候，把当前的进程放在队尾，从队列中取出第一个可以运行的进程，切换到它运行。这就是FIFO调度算法。`schedule`函数会调用`proc_run`来唤醒选定的进程。`proc_run`函数内部如下：

```
void
proc_run(struct proc_struct *proc) {
    if (proc != current) {
        bool intr_flag;
        struct proc_struct *prev = current, *next = proc;
        local_intr_save(intr_flag);
        {
            current = proc;
            lcr3(next->cr3);
            switch_to(&(prev->context), &(next->context));
        }
        local_intr_restore(intr_flag);
    }
}
```

函数中主要进行了三个操作：

1. 将当前运行的进程设置为要切换过去的进程
2. 将页表换成新进程的页表
3. 使用`switch_to`切换到新进程

在`switch_to`函数中，将需要保存的寄存器进行保存和调换。这里只需要调换被调用者保存寄存器即可。由于我们在初始化时把上下文的`ra`寄存器设定成了`forkret`函数的入口，所以这里会返回到`forkret`函数，进一步进入到`forkrets`。`forkrets`函数很短：

```
  .globl forkrets
forkrets:
    # set stack to this new process's trapframe
    move sp, a0
    j __trapret
```

这里把传进来的参数，也就是进程的中断帧放在了`sp`，这样在`__trapret`中就可以直接从中断帧里面恢复所有的寄存器啦！我们在初始化的时候对于中断帧做了一点手脚，`epc`寄存器指向的是`kernel_thread_entry`，`s0`寄存器里放的是新进程要执行的函数，`s1`寄存器里放的是传给函数的参数。在`kernel_thread_entry`函数中：

```
.text
.globl kernel_thread_entry
kernel_thread_entry:        # void kernel_thread(void)
    move a0, s1
    jalr s0

    jal do_exit
```

我们把参数放在了`a0`寄存器，并跳转到`s0`执行我们指定的函数！这样，我们就完成了调度。

七、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. PCB的构造
2. 进程需要运行需要哪些资源
3. 进程的管理
4. 进程的切换

八、下一实验简单介绍

下一次实验，我们将进行用户进程管理的内容。