OSlab4

组员：

2110729 张弛
2111451 朱景博
2111566 米奕霖

练习1：分配并初始化一个进程控制块（需要编码）

> `alloc_proc`函数（位于`kern/process/proc.c`中）负责分配并返回一个新的`struct proc_struct`结构，用于存储新建立的内核线程的管理信息。ucore需要对这个结构进行最基本的初始化，你需要完成这个初始化过程并在实验报告中简要说明你的设计实现过程

在`kern/process/proc.c`的 alloc_proc 函数中，给要创建的进程控制块(process control block，PCB) 指针（`truct proc_struct *proc`）分配了内存空间，并在分配的内存块上对该结构体中的如下字段作了初始化：

```
 below fields in proc_struct need to be initialized
    *       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
```

填充的具体初始化代码与各字段初值解析如下：

```
proc->state = PROC_UNINIT;
proc->pid = -1;
proc->runs = 0;
proc->kstack = 0;
proc->need_resched = 0;
proc->parent = NULL;
proc->mm = NULL;
memset(&(proc->context), 0, sizeof(struct context));
proc->tf = NULL;
proc->cr3 = boot_cr3;
proc->flags = 0;
memset(proc->name, 0, PROC_NAME_LEN);
```

- ***state:*** 进程状态，初始化为`PROC_UNINIT`，表示进程未初始化状态。
- ***pid:*** 进程ID，初始化为-1，表示尚未分配PID。
- ***runs:*** 进程的运行次数，初始化为0，表示进程尚未开始运行。
- ***kstack:*** 进程的内核栈地址，初始化为0。
- ***need_resched:*** 用于表示进程是否需要重新调度，初始化为0，表示不需要重新调度。
- ***parent:*** 指向父进程的指针，初始化为NULL。
- ***mm:*** 指向进程内存管理结构（struct mm_struct）的指针，初始化为NULL，表示进程尚未拥有地址空间。
- ***context:*** 进程上下文，使用memset将其所有字段初始化为0。
- ***tf***: 指向中断帧（struct trapframe）的指针，初始化为NULL。
- ***cr3:*** 存储进程的页目录表基址，初始化为ucore启动时建立好的内核虚拟空间的页目录表首地址`boot_cr3`（在`kern/mm/pmm.c`的`pmm_init`函数中初始化）
- ***flags:*** 进程标志位，初始化为0。
- ***name:*** 进程名称，使用memset将其所有字符初始化为0。

> 请说明`proc_struct`中`struct context context`和`struct trapframe *tf`成员变量含义和在本实验中的作用是啥?

- `struct context context`:

  用于在进程切换的过程中保存这一进程的上下文，具体而言就是保存创建进程时父进程的部分寄存器值：eip, esp, ebx, ecx, edx, esi, edi, ebp，由于其他寄存器在切换进程时值不变，故无需保存

- `struct trapframe *tf`:

  中断帧的指针，总是指向内核栈的某个位置：当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄存器值。

二者之于本实验的作用可以从整个进程创建、启动的过程进行观察。首先，在`proc_init`中先手动初始化了`idle`这一内核线程，之后再通过`kernel_thread`创建了`init`这一内核线程并且输出`Hello world!!`。

在`kernel_thread`中，首先设置了一个`trapframe`。其中`s0`寄存器的位置存入函数地址，`s1`的地址存入函数参数的地址，并且设置`sstatus`。其中`SPP`位为 1 表示陷入这个 trap 前并非处于用户态。`SPIE`表示陷入前开启了中断，`~SSTATUS_SIE`表示关闭当前中断。之后`epc`被设置为了`kernel_thread_entry`地址。

之后调用 `do_fork`，并且指定拷贝内存，将 `tf` 的地址传入。`do_fork` 会先使用 `alloc_proc` 分配一个进程管理块。由于 `fork` 产生的进程一定是由父进程产生的，所以再将 `parent` 设置为 `current`，并且设置进程对应的内核栈空间（分配连续两个页的空间）。之后在 `copy_thread` 中会首先在新创建的进程对应的内核栈的顶部为 `trapframe` 设置空间，将传入的 `tf` 位置对应的内容（此处即为之前所设置的 `s0` `s1` `sepc` 以及 `sstatus`）拷贝到对应的位置；并且将返回值（`a0`）设置为 0 表示这是一个 `fork` 得到的子进程，并且由于传入的栈顶地址为 0，所以 `sp` 会指向 `proc->tf` 对应 `trapframe` 的底部。之后 `copy_thread` 还设置了进程对应的 `context` 的内容，将 `ra` 返回地址设置为了 `forkret`。

`forkret` 会调用 `forkrets` 这个汇编过程。`forkrets` 首先将 `sp` 设置为 `a0` 并且恢复 `trapframe` 中的所有的上下文，之后使用 `sret` 指令返回 `sepc` 对应的地址。所以在 `forkret` 中将参数设置为当前进程 `current` 的 `tf`。

之后回到 `do_fork` 中，ucore 初始化了 `pid` 并且维护了进程块在哈希表和链表中的位置，再调用 `wakeup_proc`，将进程状态设置为 `PROC_RUNNABLE`。至此完成了初始化工作。之后 ucore 会在 `cpu_idle` 中进行调度，`schedule` 会寻找 `PROC_RUNNABLE` 的进程，并且使用 `proc_run` 执行这一进程。`proc_run` 在设置页表之后会进行 `context` 的切换，并且最后返回到先前设置的 `ra` 的地址处继续执行。而由于设置内核线程时，`epc` 成员设置为了 `kernel_thread_entry` 所以此时 `__trapret` 会将跳转到 `kernel_thread_entry`，将此前设置在 `tf` 中的参数位置 `s1`（此时已写入寄存器）转移到 `a0` 作为实际的参数，并且跳转到 `s0` 对应的地址处，即 `kernel_thread` 调用时传入的 `fn`，此处即为 `init_main` 函数的地址。

至此，一个内核线程就成功被创建并且启动了。

在上述进程创建、调度的过程中，ucore 会首先让 CPU 执行 `context` 中设置的 `ra` 地址，但是此处仍然是处于 Supervisor Mode，此时就需要借助一个“伪造”的 `trapframe` 进行特权级的切换。对于内核线程来说，由于 `sstatus` 中的 `SPP` 位被设置，所以切换之后仍然为 S 态。

练习2：为新创建的内核线程分配资源（需要编码）

> 你需要完成在`kern/process/proc.c`中的`do_fork`函数中的处理过程,在实验报告中简要说明你的设计实现过程

`do_fork`中填充的代码部分如下：

```
int
do_fork(uint32_t clone_flags, uintptr_t stack, struct trapframe *tf)
{
    ...
    if ((proc = alloc_proc()) == NULL) {
        goto fork_out;
    }

    proc->parent = current;

    if (setup_kstack(proc) != 0) {
        goto bad_fork_cleanup_proc;
    }
    if (copy_mm(clone_flags, proc) != 0) {
        goto bad_fork_cleanup_kstack;
    }
    copy_thread(proc, stack, tf);
```

```
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        proc->pid = get_pid();
        hash_proc(proc);
        list_add(&proc_list, &(proc->list_link));
        nr_process ++;
    }
    local_intr_restore(intr_flag);

    wakeup_proc(proc);

    ret = proc->pid;
    ...
}
```

填充部分的详细解析如下：

- `if ((proc = alloc_proc()) == NULL)`：
  - 调用`alloc_proc()`函数分配一个新的进程控制块(`proc_struct`)。如果分配失败，即返回的指针为`NULL`，则跳转到`fork_out`标签，表示分配进程控制块失败，函数返回错误码`-E_NO_FREE_PROC`。
- `proc->parent = current;`：
  - 将新创建的进程的父进程指针指向当前进程，即当前进程是新进程的父进程。
- `if (setup_kstack(proc) != 0)`：
  - 调用`setup_kstack`函数为新进程分配内核栈。如果分配失败，即返回非零值，表示内核栈分配失败，跳转到`bad_fork_cleanup_proc`标签，释放已分配的进程控制块并返回错误码`-E_NO_MEM`。
- `if (copy_mm(clone_flags, proc) != 0)`：
  - 调用`copy_mm`函数根据`clone_flags`参数来复制或共享当前进程的内存管理信息。如果操作失败，即返回非零值，表示内存复制或共享失败，跳转到`bad_fork_cleanup_kstack`标签，释放已分配的内核栈并返回错误码`-E_NO_MEM`。
- `copy_thread(proc, stack, tf);`：
  - 调用`copy_thread`函数设置新进程的中断帧(`tf`)和上下文信息。
- `bool intr_flag; local_intr_save(intr_flag);`：
  - 关中断，确保在修改全局数据结构时不会被中断打断。
- `proc->pid = get_pid(); hash_proc(proc); list_add(&proc_list, &(proc->list_link)); nr_process ++;`：
  - 为新进程分配唯一的进程ID，将进程控制块加入进程哈希表（`hash_proc`函数用于将进程加入哈希表中），并将进程加入进程列表。同时，增加进程计数器`nr_process`。
- `local_intr_restore(intr_flag);`：
  - 恢复中断状态。
- `wakeup_proc(proc);`：
  - 将新进程状态设置为`PROC_RUNNABLE`，使其可以被调度执行。
- `ret = proc->pid;`：
  - 设置函数返回值为新进程的PID，表示进程创建成功。

最后，如果在上述过程中发生了错误，例如进程控制块分配失败或者内核栈分配失败，会跳转到相应的标签，释放已分配的资源，然后返回错误码。整个函数的目标是创建一个新的进程，为其分配必要的资源，设置好进程控制块、内核栈、中断帧等信息，最终将其设置为可运行状态，以便后续被调度执行。

> 请说明ucore是否做到给每个新fork的线程一个唯一的id？请说明你的分析和理由

ucore 分配进程 pid 的功能模块在 `kern/process/proc.c/get_pid` 函数中

```
// get_pid - alloc a unique pid for process
static int
get_pid(void) {
    static_assert(MAX_PID > MAX_PROCESS);
    struct proc_struct *proc;
    list_entry_t *list = &proc_list, *le;
    static int next_safe = MAX_PID, last_pid = MAX_PID;
    if (++ last_pid >= MAX_PID) {
        last_pid = 1;
        goto inside;
    }
    if (last_pid >= next_safe) {
    inside:
        next_safe = MAX_PID;
    repeat:
        le = list;
        while ((le = list_next(le)) != list) {
            proc = le2proc(le, list_link);
            if (proc->pid == last_pid) {
                if (++ last_pid >= next_safe) {
                    if (last_pid >= MAX_PID) {
                        last_pid = 1;
                    }
                    next_safe = MAX_PID;
                    goto repeat;
                }
            }
            else if (proc->pid > last_pid && next_safe > proc->pid) {
                next_safe = proc->pid;
            }
        }
    }
    return last_pid;
}
```

根据上述 `get_pid` 的函数实现可知，ucore 在分配新的进程时确保了为每个新的 `fork` 线程分配唯一的 pid (其实从注释中的 `a unique pid` 也能看出来...)。具体分析如下：

- **唯一性保障**：在 `get_pid` 函数中，`last_pid` 表示上一个分配的进程ID，它会不断递增，当超过最大PID（`MAX_PID`）时，重新从1开始分配。在每次分配新的PID之前，会检查当前的 `last_pid` 是否已经大于等于 `next_safe`，如果是，则会从头开始遍历进程列表，寻找已经被分配的PID。如果找到相同的PID，`last_pid` 会递增并重新检查，直到找到一个没有被分配的PID为止。这样确保了新分配的PID在当前进程列表中是唯一的。
- **遍历进程列表**：在函数内部，通过遍历 `proc_list`，获取进程控制块（`struct proc_struct`），检查其PID和 `last_pid` 的关系。如果找到一个进程的PID等于 `last_pid`，则

`last_pid`会递增，继续寻找，直到找到一个未被分配的PID。这样可以保证在进程列表中不存在相同的PID。

综上所述，`ucore`的`get_pid`函数通过遍历当前的进程列表，确保为每个新的`fork`线程分配一个唯一的PID，保证了新创建的线程有唯一的标识符。

练习3：编写 proc_run 函数（需要编码）

> proc_run用于将指定的进程切换到CPU上运行,请完成编写`proc_run`函数

`proc_run`函数的具体实现如下;

```
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);
    }
}
```

`proc_run`函数用于在内核中切换到指定进程，函数的详细解析如下：

- **判断当前进程是否为目标进程**：首先，函数会检查当前运行的进程是否已经是目标进程，如果是，则无需进行切换，直接返回。这个检查可以避免不必要的上下文切换。
- **保存中断状态并进行上下文切换**：如果当前进程不是目标进程，函数会保存当前中断状态（`local_intr_save(intr_flag);`），然后进行上下文切换。上下文切换的步骤如下：
  - `current = proc;`：将当前进程指针指向目标进程，表示切换到目标进程执行。
  - `lcr3(next->cr3);`：切换页目录表，将CR3寄存器设置为目标进程的页目录表基址，以确保进程的虚拟地址空间正确映射。
  - `switch_to(&(prev->context), &(next->context));`：调用`switch_to`函数实现上下文切换，将前一个进程的上下文保存到`prev->context`，并将目标进程的上下文加载到CPU寄存器中，使得处理器从当前进程切换到目标进程。
- **恢复中断状态**：在上下文切换完成后，函数会恢复之前保存的中断状态（`local_intr_restore(intr_flag);`），确保中断处理正常进行。

> 在本实验的执行过程中，创建且运行了几个内核线程?

在本实验中，一共创建了两个内核线程：

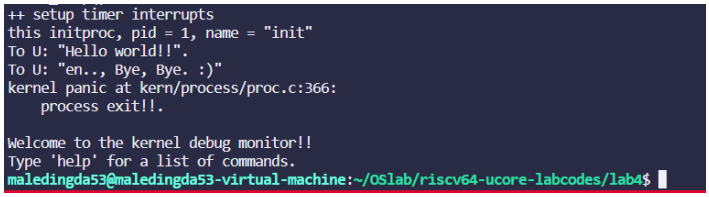
- ***创建第0个内核线程 idleproc***。`init.c/kern_init`函数调用了 `proc_init`函数。`proc_init`` 函数启动了创建内核线程的步骤。首先当前的执行上下文（从`kern_init` 启动至今）就可以看成是 uCore 内核（也可看做是内核进程）中的一个内核线程的上下文。为此，ucore 通过给当前执行的上下文分配一个进程控制块以及对它进行相应初始化，将其打造成第 0 个内核线程 –*idleproc*。

- ***创建第1个内核线程 initproc***。第 0 个内核线程主要工作是完成内核中各个子系统的初始化，然后就通过执行 `cpu_idle` 函数开始过退休生活了。所以 ucore 接下来还需创建其他进程来完成各种工作，但 $idleproc$ 内核子线程自己不想做，于是就通过调用 `kernel_thread` 函数创建了一个内核线程 `init_main`。在Lab4中，这个子内核线程的工作就是输出一些字符串后返回（参看 `init_main` 函数）。但在后续的实验中，`init_main` 会承担创建特定的其他内核线程或用户进程的工作。

```
// init_main - the second kernel thread used to create user_main kernel
threads
static int
init_main(void *arg) {
    cprintf("this initproc, pid = %d, name = \"%s\"\n", current->pid,
get_proc_name(current));
    cprintf("To U: \"%s\".\n", (const char *)arg);
    cprintf("To U: \"en.., Bye, Bye. :)\"\n");
    return 0;
}
```

效果验证

完成上述对 `proc.c` 中各模块的补充后，即可进行 `make qemu` 验证了，验证结果如下：



值得一提的是，由于在执行 `do_exit` 函数时会直接调用 `panic` 并且 `kmonitor` 中调用了 `sbi_shutdown` 所以操作系统在执行完 `init_main` 之后会直接关机。

之后再进行 `make grade` 即可完成验证：

扩展练习 Challenge：

> 说明语句 `local_intr_save(intr_flag);....local_intr_restore(intr_flag);` 是如何实现开关中断的?

相关宏定义可以定位到 `kern/sync/sync.h` 中：

```
static inline bool __intr_save(void) {
    if (read_csr(sstatus) & SSTATUS_SIE) {
        intr_disable();
        return 1;
    }
    return 0;
}

static inline void __intr_restore(bool flag) {
    if (flag) {
```

```
        intr_enable();
    }
}

#define local_intr_save(x) \
    do {                   \
        x = __intr_save(); \
    } while (0)
#define local_intr_restore(x) __intr_restore(x);
```

上述宏通过读取和修改 CSR 中的SIE（Supervisor Interrupt Enable）位来实现开关中断的操作。下面是对这些宏定义的详细解释：

- `__intr_save()`：
  - 该内联函数通过读取CSR寄存器（`sstatus`）中的SIE位来判断中断是否开启。
  - 如果SIE位为1（表示中断开启），则调用`intr_disable()`函数来关闭中断，同时返回1表示中断之前是开启状态。
  - 如果SIE位为0（表示中断关闭），则不执行任何操作，返回0表示中断之前是关闭状态。
- `__intr_restore(bool flag)`：
  - 该内联函数根据传入的`flag`参数，判断之前的中断状态。
  - 如果`flag`为1，表示中断之前是开启状态，则调用`intr_enable()`函数来重新开启中断。
  - 如果`flag`为0，表示中断之前是关闭状态，则不执行任何操作，维持中断关闭状态。
- `local_intr_save(x)`：
  - 宏定义了一个`do-while`循环，实际上是调用了`__intr_save()`函数，将中断状态保存到传入的参数`x`中。
- `local_intr_restore(x)`：
  - 该宏直接调用了`__intr_restore(x)`函数，根据传入的参数`x`来恢复之前的中断状态。

这样，`local_intr_save(intr_flag)`会在执行时保存当前中断状态，并临时关闭中断。然后，在关键代码执行完成后，`local_intr_restore(intr_flag)`会根据保存的中断状态来恢复中断状态。

通过这种方式，宏定义了一种简洁的方式来实现开关中断，确保在关键代码段中，中断不会被打断，以维护关键操作的完整性和可靠性。