Lab4实验报告

练习1：分配并初始化一个进程控制块

1.`proc_struct` 中的成员变量说明

`struct context context`

- **含义**:
  - `struct context` 通常用于保存进程的 CPU 寄存器状态和执行上下文。它包含了在上下文切换时需要保存和恢复的信息，比如通用寄存器、程序计数器（PC）、堆栈指针等。
- **在本实验中的作用**:
  - 当操作系统进行上下文切换（即从一个进程切换到另一个进程）时，需要保存当前进程的执行状态，并加载下一个进程的执行状态。`context` 成员变量负责保存这些信息，以便在进程切换后能够恢复到正确的执行状态。
  - 具体来说，当一个进程被挂起或阻塞时，`context` 中的信息会被保存，以便在下次调度时能够恢复这些状态，确保进程可以从它被挂起的地方继续执行。

`struct trapframe *tf`

- **含义**:
  - `struct trapframe` 是一个数据结构，用于保存中断或异常发生时的上下文信息。它包含了中断发生时寄存器的状态、程序计数器和其他重要的信息，这些信息可以帮助操作系统处理异常或中断，并恢复进程状态。
- **在本实验中的作用**:
  - 当进程受到中断（例如时钟中断、I/O 中断）时，CPU 会将当前执行状态（即寄存器的值）保存在 `trapframe` 中，以便在处理中断后恢复执行。
  - 在本实验中，`tf` 成员变量指向一个 `trapframe` 结构体的实例。这个实例用于保存当前进程在中断发生时的状态，以便在中断处理完成后，能够准确地恢复进程的执行。
  - 例如，如果进程在运行期间被调度中断，操作系统需要在中断处理完成后，使用 `tf` 中保存的信息来恢复进程的状态，确保进程可以继续正常运行。

2.`alloc_proc` 函数设计实现过程

分配内存

- 使用 `kmalloc` 函数分配一个 `proc_struct` 结构体的内存空间。如果分配失败（即返回 `NULL`），则可以选择返回 `NULL`，表示无法创建新的进程。

初始化成员变量

- 对 `proc_struct` 的各个成员变量进行初始化。这些变量包括：
  - `state`：设置进程的初始状态为 `PROC_UNINIT`，表示进程尚未初始化。

- `pid`：初始化为 `-1`，表示进程 ID 尚未分配。
- `runs`：初始化为 `0`，表示进程的运行次数。
- `kstack`：设置为 `0`，表示内核栈尚未分配。
- `need_resched`：初始化为 `0`，表示该进程当前不需要重新调度。
- `parent`：设置为 `NULL`，表示没有父进程。
- `mm`：初始化为 `NULL`，表示内存管理结构尚未分配。
- `context`：根据需要初始化上下文信息，用于后续的上下文切换。
- `tf`：初始化为 `NULL`，表示当前没有相关的中断或异常上下文。
- `cr3`：初始化为 `boot_cr3`，表示页目录基址尚未设置。
- `flags`：初始化为 `0`，表示没有特殊标志。
- `name`：将进程名初始化为空字符串或默认值。

返回指针

- 返回初始化后的 `proc_struct` 结构体的指针，以便其他部分的代码可以使用这个新的进程结构体。

错误处理

- 处理内存分配失败的情况，并确保在所有分配和初始化步骤中进行适当的错误处理，以提高代码的健壮性。

```
static struct proc_struct *
alloc_proc(void) {
    struct proc_struct *proc = kmalloc(sizeof(struct proc_struct));
    if (proc != NULL) {
        // 初始化进程结构体的各个字段
        proc->state = PROC_UNINIT;              // 初始状态为未初始化
        proc->pid = -1;                         // PID 未分配
        proc->runs = 0;                         // 运行次数为 0
        proc->kstack = 0;                       // 内核栈地址初始化为 0
        proc->need_resched = 0;                 // 初始不需要重新调度
        proc->parent = NULL;                    // 父进程指针初始化为 NULL
        proc->mm = NULL;                        // 内存管理结构体指针初始化为
NULL

        // 初始化 context 字段
        memset(&proc->context, 0, sizeof(struct context));

        proc->tf = NULL;                        // 陷阱帧初始化为 NULL
        proc->cr3 = boot_cr3;                   // CR3 寄存器值初始化
        proc->flags = 0;                        // 进程标志初始化为 0

        // 初始化进程名称为一个空字符串
        memset(proc->name, 0, PROC_NAME_LEN + 1);
    }
    return proc;
}
```

练习二：为新创建的内核线程分配资源

1.do_fork 函数设计实现过程

1. **分配并初始化进程控制块（alloc_proc函数）**：
   - 使用 alloc_proc 分配一个新的 proc_struct 结构体并初始化其中的字段。如果分配失败，直接跳转到错误处理。

```
proc = alloc_proc();
if (proc == NULL) {
    goto bad_fork_cleanup_proc; // 分配进程结构失败
}
```

2. **分配并初始化内核栈（setup_stack函数）**：
   - 调用 setup_kstack 为新进程分配内核栈。若分配失败，释放进程结构体并跳转到错误处理。

```
if (setup_kstack(proc) != 0) {
    goto bad_fork_cleanup_proc;
}
```

3. **根据clone_flags决定是复制还是共享内存管理系统（copy_mm函数）**：
   - 根据 clone_flags 标志来决定是复制内存（独立地址空间）还是共享内存（相同地址空间）。这取决于 CLONE_VM 标志，如果设置则共享，否则复制。

```
if (copy_mm(clone_flags, proc) != 0) {
    goto bad_fork_cleanup_proc;
}
```

4. **设置进程的中断帧和上下文（copy_thread函数）**：
   - 调用 copy_thread 函数来初始化进程的陷阱帧 trapframe 和上下文 context，确保新进程从正确的入口点和堆栈开始执行。

```
copy_thread(proc, stack, tf);
```

5. **把设置好的进程加入链表**：

- 分配唯一的 `PID`，然后将新进程插入到哈希链表 `hash_list` 和进程链表 `proc_list` 中。使用 `local_intr_save` 和 `local_intr_restore` 禁用和恢复中断，确保对列表的操作原子性。

```
bool intr_flag;
local_intr_save(intr_flag); // 禁用中断，确保原子操作
{
    proc->pid = get_pid();       // 分配唯一的 PID
    hash_proc(proc);              // 将进程添加到哈希列表
    list_add(&proc_list, &(proc->list_link)); // 将进程添加到进程链表
    nr_process++;                 // 增加进程计数
}
local_intr_restore(intr_flag);    // 恢复中断
```

6. **将新建的进程设为就绪态**：

- 调用 `wakeup_proc` 将进程状态设置为 `RUNNABLE`，表示可以被调度器选择运行。

```
wakeup_proc(proc); // 使新进程进入可运行状态
```

7. **将返回值设为线程id**：

- 最后，函数最后将新进程的 `PID` 作为返回值 `ret` 返回。

```
ret = proc->pid;   // 返回新进程的 PID
```

2.说明ucore是否做到给每个新fork的线程一个唯一的id

找到get_pid()函数：

```
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;
}
```

该函数确实在每次调用时分配一个唯一 PID，因为每次都会遍历整个进程链表 `proc_list` 来检测 `last_pid` 是否重复。这确保了即便 `fork` 多个线程或进程，也不会为它们分配相同的 PID。

PID 递增：

- `last_pid` 自增的方式用于尽量找到下一个可用的 PID，从而减少在 `MAX_PID` 范围内的空隙。
- 当 `last_pid` 达到 `MAX_PID` 或遇到冲突时，会重置为 1，再从头开始检查。

冲突检测：

- 通过 `inside` 标签的检查逻辑，函数遍历所有现有进程的 PID。若存在冲突（即某个进程已占用当前 `last_pid`），则 `last_pid` 再自增，直到找到一个不与现有进程冲突的 PID。
- `next_safe` 辅助跟踪安全的 PID 范围，避免重复检测。

练习三3：编写proc_run 函数

1.proc_run 函数设计实现过程

1. **检查要切换的进程是否与当前正在运行的进程相同，如果相同则不需要切换**：
    - 如果要切换的进程 `proc` 不是当前进程（`proc != current`），则需要进行进程切换。

    ```
    if (proc != current) {
    ```

2. **禁用中断**：
    - 使用 `local_intr_save(intr_flag)` 保存当前中断状态并禁用中断。禁用中断是为了确保进程切换操作是原子性的，即在切换过程中不会被其他中断打断，避免出现竞争条件。

```
bool intr_flag; // 用于保存中断状态
local_intr_save(intr_flag);
```

3. **切换当前进程为要运行的进程**：
   - 将 `current` 进程指针设置为目标进程 `proc`。这意味着接下来调度的就是目标进程。

```
current = proc;
```

4. **切换页表，以便使用新进程的地址空间**：
   - 使用 `lcr3(proc->cr3)` 加载新进程的页目录表基地址，这样 CPU 就会使用目标进程的内存地址空间进行操作。

```
lcr3(proc->cr3);
```

5. **实现上下文切换**：
   - 调用 `switch_to(current, proc)` 进行上下文切换，将 CPU 的执行从当前进程切换到目标进程。`switch_to` 会保存当前进程的寄存器状态，并加载目标进程的寄存器状态。

```
switch_to(current, proc);
```

6. **允许中断**：
   - 在切换完成后，调用 `local_intr_restore(intr_flag)` 恢复之前保存的中断状态，重新启用中断。这保证了在进程切换后，系统中断能够继续处理。

```
local_intr_restore(intr_flag);
```

2.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，系统创建并运行了两个内核线程：

1. **idleproc**：
   - 这是第一个内核进程，它在系统启动时创建。其主要任务是完成内核中各个子系统的初始化工作，例如内存管理、进程调度、文件系统的挂载等。在完成初始化任务后，`idleproc` 进程会被立即调度执行，确保系统中至少有一个可运行的进程存在，通常它会持续运行，直到系统没有其他待执行的进程。`idleproc` 进程的存在确保了系统始终处于运行状态，避免 CPU 进入空闲状态。

2. **initproc**：

- 这是为了执行本实验的功能而创建的内核进程。它通常在系统初始化完成后，由 `idleproc` 或其他进程调度来执行。`initproc` 负责启动其他用户进程或执行系统的其他初始化任务，确保系统能够执行实验要求的功能。`initproc` 是所有用户进程的祖先进程，系统中大多数用户进程的创建都会依赖于它。

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?

找到当时Lab1的代码：

```
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);

#endif /* !__KERN_SYNC_SYNC_H__ */
```

1. `__intr_save` 函数

功能：检查当前中断状态，并禁用中断。

- 如果中断处于启用状态（`SSTATUS_SIE` 为 1），禁用中断，并返回 `1`，表示原先中断是启用的。
- 如果中断已经禁用，则返回 `0`。

2. `__intr_restore` 函数

功能：根据传入的标志恢复中断状态。

- 如果标志为 `1`，表示原先中断是启用状态，则调用 `intr_enable();` 恢复中断。
- 如果标志为 `0`，则保持中断禁用状态。

3. `local_intr_save` 宏

功能：用于保存当前中断的状态并禁用中断。

- 宏内部调用 `__intr_save()`，将当前中断状态保存到 `intr_flag` 中。
- `intr_flag` 的值为 `1` 表示原先中断是启用的，`0` 表示中断已经禁用。

4. `local_intr_restore` 宏

功能：根据保存的标志值恢复中断状态。

- 如果 `intr_flag` 为 `1`，则启用中断。
- 如果 `intr_flag` 为 `0`，则保持中断禁用。

5. 整体流程

- **禁用中断（`local_intr_save`）**：调用 `local_intr_save(intr_flag);` 后，保存当前中断状态并禁用中断，确保后续的操作不会被中断打断。
- **恢复中断（`local_intr_restore`）**：调用 `local_intr_restore(intr_flag);` 后，恢复中断状态。如果原先中断启用，则恢复中断；如果原先已经禁用，则不做操作。