LAB 4 内核进程的管理

实验实现流程

- 在`kern_init()`中调用`proc_init()`开始实现进程的初始化工作
- 在`proc_init()`中，使用`list_init(&proc_list)`初始化进程列表，后通过`idleproc = alloc_proc()`,分配第0个进程结构体给`idleproc`，在`alloc_proc(void)`中返回一个初始化的进程块。之后开始设置idleproc的各个成员的值。其中`pid`设置为0，`state`设置为`PROC_RUNNABLE`表示可运行状态，将`need_resched`设置为1使其在接下来被立刻调度，并将current指向它。后调用`kernel_thread()`函数创建`init_main`进程，这个进程会在用户空间运行，并打印"Hello world!!"
- 在`kernel_thread()`中，创建一个中断帧，并利用其保存内核线程的参数和函数指针，状态和入口点（入口点处实现函数的跳转和参数的传递），后将tf传给`do_fork()`创建一个新进程（内核线程）
- 在`do_fork()`中，为其分配一个进程控制块。
  - 首先通过`alloc_proc()`为其分配并初始化进程控制块。
  - 后通过`setup_kstack(proc)`为进程分配一个内核栈
  - `copy_mm(clone_flags,proc)`中由clone_flags决定是复制还是共享内存管理系统
  - 后通过`copy_thread(proc,stack,tf)`设置进程的中断帧和上下文
  - 然后通过`get_pid()`,`hash_proc(proc)`,`list_add(&proc_list,&(proc->list_link))`,获取进程号，并将其添加入链表和哈希表中
  - 最后通过`wakeup_proc(proc)`唤醒进程`proc->state = PROC_RUNNABLE`,并返回其进程号。
- 后返回main函数，在运行`cpu_idle()`后，将会运行进程调度函数`schedule(void)`，通过`fifo`策列，在`list_link`中找到第一个符合要求的可运行内存，并调用`proc_run(next)`实现进程的切换与运行。
- 在`proc_run()`中，利用`switch_to(&prev->context,&proc->context)`，将保存当前进程current的执行现场（进程上下文），恢复新进程的执行现场，完成进程切换

练习1：分配并初始化一个进程控制块

- proc_struct初始化过程：

```
    proc->state=PROC_UNINIT;//给进程设置为未初始化状态，此进程为一空壳
    proc->pid=-1;//未初始化的进程，其pid为-1
    proc->runs=0;//初始化时间片,刚刚初始化的进程，运行时间一定为零
    proc->kstack=0;//内核栈地址,该进程分配的地址为0，因为还没有执行，也没有被重定位，因为默认地址都是从0开始的。
    proc->need_resched=0;//不需要调度
    proc->parent=NULL;//父进程为空
    proc->mm=NULL;//虚拟内存为空
    memset(&(proc->context),0,sizeof(struct context));//初始化上下文
    proc->tf=NULL;//中断帧指针为空
    proc->cr3=boot_cr3;//页目录为内核页目录表的基址
    proc->flags=0;//标志位为0
    memset(&(proc->name),0,PROC_NAME_LEN);//进程名为0
```

使用上述方法初始化，是由于在proc_init中将会对这些位置做出如下检验，通过上述初始化模式，便于确认是否初始化成功

```
  int *context_mem = (int*) kmalloc(sizeof(struct context));
    memset(context_mem, 0, sizeof(struct context));
    int context_init_flag = memcmp(&(idleproc->context), context_mem,
sizeof(struct context));

    // 以上述同样的方式，我们创建一个和idleproc->name大小一样的内存空间，
    // 并检查idleproc->name在创建时是否被正确地初始化为了0
    int *proc_name_mem = (int*) kmalloc(PROC_NAME_LEN);
    memset(proc_name_mem, 0, PROC_NAME_LEN);
    int proc_name_flag = memcmp(&(idleproc->name), proc_name_mem,
PROC_NAME_LEN);
    // 如果idleproc的所有成员都被正确地初始化为了0，那么我们就打印一条消息，说明
alloc_proc函数正确地工作了
    if(idleproc->cr3 == boot_cr3 && idleproc->tf == NULL &&
!context_init_flag
        && idleproc->state == PROC_UNINIT && idleproc->pid == -1 && idleproc-
>runs == 0
        && idleproc->kstack == 0 && idleproc->need_resched == 0 && idleproc-
>parent == NULL
        && idleproc->mm == NULL && idleproc->flags == 0 && !proc_name_flag
    )
```

- `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?
  - proc_struct进程控制块：其中包含了一个进程的各个信息，包括它的序号，分配的内存地址，它的状态，父进程，使用次数等。通过他们可以实现进程的分配，管理和释放。
  - `struct context 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。在后续`proc_run()`中，通过`switch_to`运行一个新进程，需要利用`context`进行上下文切换。

    ```
    //保存一个进程（或线程）的执行上下文
    struct context {
        uintptr_t ra;//表示返回地址寄存器的值。当一个函数调用另一个函数时，会把返回地址
    保存在ra寄存器中
        uintptr_t sp;//表示栈指针寄存器的值。栈指针寄存器指向进程的栈顶，用于支持函数调
    用和局部变量的存储
        //剩下的是各种通用寄存器
        uintptr_t s0;
        uintptr_t s1;
        uintptr_t s2;
        uintptr_t s3;
        uintptr_t s4;
        uintptr_t s5;
        uintptr_t s6;
        uintptr_t s7;
        uintptr_t s8;
        uintptr_t s9;
        uintptr_t s10;
        uintptr_t s11;
    };//上下文只保存了部分寄存器，因为线程切换在一个函数当中
    ```

```
//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
//在实际的进程切换过程中我们只需要保存被调用者保存寄存器
```

- `struct trapframe *tf`:`tf`中保存了中断帧，当进程从用户空间跳入内核空间时，进程的状态保存在中断帧中。在后续`do_fork()`为新创建的内核线程分配资源时，`copy_thread()`将通过`tf`复制父进程的中断帧和上下文信息.

```
struct trapframe {
    struct pushregs gpr;//用于保存通用寄存器的值
    uintptr_t status;//保存处理器的状态寄存器值
    uintptr_t epc;//保存异常程序计数器的值。当发生异常时，处理器会把发生异常的指令的地址保存到异常程序计数器中
    uintptr_t badvaddr;//用于保存引发异常的虚拟地址
    uintptr_t cause;
};
```

练习2：为新创建的内核线程分配资源

- 设计实现过程

  在`do_fork()`函数中将为新进程分配一个进行控制块，其作用是，创建当前内核线程的一个副本，它们的执行上下文、代码、数据都一样，但是存储位置不同。因此在`do_fork()`中，我们需要传入`stack`和`trapframe`其具体过程为：

```
//    1. call alloc_proc to allocate a proc_struct
    //    分配并初始化进程控制块alloc_proc
        if((proc=alloc_proc())==NULL)
        {
            goto fork_out;
        }
        //将子进程的父节点设置为当前进程
        proc->parent = current;
    //    2. call setup_kstack to allocate a kernel stack for child process,此时为分配两个页表
        if(setup_kstack(proc)!=0)//2.调用setup_stack()函数为进程分配一个内核栈
        {
            goto bad_fork_cleanup_kstack;
        }
    //    3. call copy_mm to dup OR share mm according clone_flag
    //    clone_flags决定是复制还是共享内存管理系统（copy_mm函数）
       if( copy_mm(clone_flags,proc)!=0)
       {
            goto bad_fork_cleanup_kstack;
       }
    //    4. call copy_thread to setup tf & context in proc_struct
        //设置进程的中断帧和上下文（copy_thread函数）
        copy_thread(proc,stack,tf);//复制父进程的中断帧和上下文信息
        //do_fork函数会调用copy_thread函数来在新创建的进程内核栈上专门给进程的中断帧分配一块空间
    //    5. insert proc_struct into hash_list && proc_list
        bool intr_flag;
        local_intr_save(intr_flag);//屏蔽中断
        {
            proc->pid=get_pid();//返回一个链表中尚未使用的节点号（最近的）
            hash_proc(proc);
```

```
            list_add(&proc_list,&(proc->list_link));
            nr_process++;//进程数加一
        }
         local_intr_restore(intr_flag);//恢复中断
    //    6. call wakeup_proc to make the new child process RUNNABLE
        wakeup_proc(proc);//唤醒新进程 proc->state = PROC_RUNNABLE;
    //    7. set ret vaule using child proc's pid
        ret=proc->pid;
```

- 请说明ucore是否做到给每个新fork的线程一个唯一的id?

  每个新fork的id通过`get_pid()`获取，`get_pid()`,会遍历当前`proc_list`所有进程的id,并返回一个最靠前的，`proc_list`没有的id。因此在`proc_list`它的id号是唯一的，但是如果我们在分配进程时，最后不通过`list_add(&proc_list,&(proc->list_link))`把它放入进程表中，则可以拿到重复id的fork。

练习3：编写proc_run函数

- 设计实现过程

  在proc_run中，实现当前进程的切换。其中进程的切换需要涉及页表的切换，以便使用新的进程的地址空间，并需要通过context实现上下文切换。

```
struct proc_struct *prev=current,*next = proc;//用一个prev来存储当前的进程信息
      bool intr_flag;//返回一个是否禁用成功的函数
      local_intr_save(intr_flag);//禁用中断
      {
      current=proc;//切换当前进程为要运行的进程
      lcr3(proc->cr3);//切换页表，以便使用新进程的地址空间
      switch_to(&prev->context,&proc->context);//上下文切换
      }
      local_intr_restore(intr_flag);//允许中断
```

- 在本实验的执行过程中，创建且运行了几个内核线程?
  - 本次实验中，创建并运行了两个内核进程，分别是`idleproc`,和`initproc`。其中`idleproc`为一空闲进程，它的主要目的是在系统没有其他任务需要执行时，占用 CPU 时间，同时便于进程调度的统一化,其主要完成内核中各个子系统的初始化，然后就通过执行cpu_idle函数开始摆烂，切换为`initproc`执行`init_main`.