

Lab5 Process-I

一、实验概述

学习使用fork(), getpid(), exec(), wait(), signal(), kill()等调用, 了解zombie进程。

阅读ucore代码, 理解ucore如何创建并运行内核进程。

二、实验目的

1. 学习创建进程
2. 学习父子进程
3. 理解ucore如何创建并运行内核进程

三、实验内容

1. fork()
2. getpid()
3. wait()
4. Zombie
5. exec()
6. kill(), signal()
7. ucore process

四、实验流程及相关知识点

第一步. 通过fork()创建进程

调用fork()函数, 可以为当前进程创建一个“副本”子进程。"副本"子进程几乎完全复制父进程的所有信息。

```
#include<stdio.h>
#include<unistd.h>
int main(){
    int i = 0;
    fork(); //创建子进程
    i++;
    printf("%d\n", i);
    while(1);
    return 0;
}
```

父子进程的PID是不同的, 一般情况下PID小的是父进程。

那么在代码中如何区分父子进程呢？

请尝试"man fork"指令。（Tips: 如果遇到困难，查手册）

RETURN VALUE

On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and `errno` is set appropriately.

通过阅读手册，我们得知可以通过fork()的返回值判断“当前”两个进程哪个是父进程，哪个是子进程。

```
#include<stdio.h>
#include<unistd.h>
int main(){
    if(fork()){
        printf("I'm parent process.\n");
    }else{
        printf("I'm child process.\n");
    }
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ gcc -o fork fork.c
sy@sy-OSlab:~/Desktop/lab$ ./fork
I'm parent process.
I'm child process.
```

第二步. 通过getpid()获得进程号

除了区分父子进程，我们经常还需要获得进程的信息，如自己的进程号、父进程的进程号，比较常用的一个函数是getpid()，该函数可以获得自己的进程号。

```
int main(){
    if(fork()){
        printf("I'm parent process %d.\n",getpid());
    }else{
        printf("I'm child process %d.\n",getpid());
    }
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ gcc -o fork fork.c
sy@sy-OSlab:~/Desktop/lab$ ./fork
I'm parent process 5764.
I'm child process 5765.
```

请尝试通过getppid()获得父进程的pid。

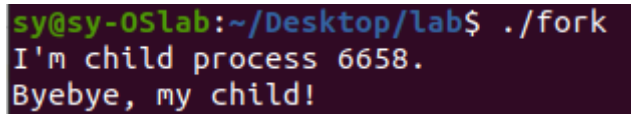
第三步. 学习使用wait()

前面实验的过程中，可以发现父子进程之间是没有固定的执行顺序的，如果我们想要它们以固定的顺序执行，可以使用wait()。

wait()的原型是 `pid_t wait(int *wstatus)`，当wait()被执行时，当前进程会被挂起(suspend)，直到子进程结束，子进程结束后会给父进程发送 `SIGCHLD` 中断，wait()收到中断后会结束阻塞并使系统回收子进程的相关资源。

因此，我们可以通过在父进程中调用wait()来使子进程优先执行。

```
int main(){
    if(fork()){
        wait(NULL);
        printf("Byebye, my child!\n");
    }else{
        printf("I'm child process %d.\n",getpid());
    }
    return 0;
}
```



```
sy@sy-05lab:~/Desktop/lab$ ./fork
I'm child process 6658.
Byebye, my child!
```

注意参数 `int *wstatus` 并非指定某个子进程，而是当子进程结束时用于存储子进程的结束状态值。

如果我们不需要结束状态值，则可以将参数指定为NULL，即 `wait(NULL)`。

那么现在请回答一个问题：

如果一个父进程有多个子进程，它等待的是哪个子进程？是任意一个子进程还是某个子进程还是全部子进程？

请通过 `man wait` 自己找到答案。

另一个常用的wait指令是waitpid()，也请通过 `man wait` 学习如何使用。

第四步. Zombie

前面一步我们学习了wait()的使用，wait()除了可以使子进程优先执行外，还有一个很重要的作用：**回收子进程的相关资源**（比如pid）。这个功能是非常重要的，比如系统的pid是有限的，如果长期有大量进程pid未被回收，可能导致pid被用完从而无法分配pid给新的进程。

关于wait()的执行，这里可以排列组合出几种情况：

1. 父进程执行wait()，父进程比子进程后结束
2. 父进程执行wait()，父进程比子进程先结束（由于wait()的执行，这种情况并不会出现）
3. 父进程不执行wait()，父进程比子进程后结束
4. 父进程不执行wait()，父进程比子进程先结束

情况1，子进程结束后，父进程会通过wait回收子进程的相关资源。

情况3则比较复杂了，会出现一种情况：**子进程虽然已经结束了，但是父进程未结束并且未回收尸体（资源），子进程变成了僵尸（Zombie process）**。那么这样的僵尸进程怎么处理呢？

要解决僵尸进程，我们需要先解决另一个问题：**如果子进程没有结束但是父进程结束了（孤儿进程），那么子进程会变成无父之子吗？** 答案是不会，子进程会被过继给init进程或者注册过的祖父进程（孤儿院）以确保每个进程都一定有父进程。

回到僵尸进程的问题，如果因为父进程不回收而导致出现僵尸进程，那么在**父进程结束后**，僵尸进程会被过继给“继父”，新的父进程会自动回收僵尸进程。

现在，请尝试自己回答情况4的结果会是什么？

第五步. 学习使用exec()

实际编写程序时，我们经常需要fork一个新的进程，但是我们大部分时候并不需要fork一个新的“自己”，因此如何fork一个和原进程不同的进程是很重要的。但我们要创建新的进程只能通过fork()，所以我们需要通过exec()把fork()出的副本进行“重写”以让新的进程执行与父进程不同的功能。

```
int main(){
    printf("before execl ...\n");
    execl("/bin/ls", "/bin/ls", NULL);
    printf("after execl ...\n");
    return 0;
}
```

```
sy@sy-OSlab:~/Desktop/lab$ ./exec
before execl ...
exec execl.c fork fork.c sample1 sample1.c
```

需要特别注意exec()一旦执行，原本的进程将不复存在，进程的代码段全部被替换，并且会分配新的资源，因而原进程exec()之后的代码不会再被执行（即使exec()执行结束）。

exec家族有很多不同参数列表的兄弟姐妹都可以实现exec的功能，请自己 man 一下。

第六步. 学习使用kill(), signal()

第一周我们学习过通过kill -9来强制终止进程。这里的9即9号中断信号SIGKILL，该信号用于强制结束进程，并且这个信号不能被阻塞、处理和忽略。可以通过 kill -l 查看中断号对应的中断信号。

```
sy@sy-OSlab:~/Desktop/lab$ kill -l
1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
6) SIGABRT     7) SIGBUS     8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV    12) SIGUSR2    13) SIGPIPE    14) SIGALRM    15) SIGTERM
16) SIGSTKFLT  17) SIGCHLD   18) SIGCONT    19) SIGSTOP    20) SIGTSTP
21) SIGTTIN    22) SIGTTOU    23) SIGURG     24) SIGXCPU    25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF   28) SIGWINCH   29) SIGIO      30) SIGPWR
31) SIGSYS     34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

而本周的学习中我们提到子进程结束时会发送SIGCHLD中断给父进程，这个信号是第17号信号，SIGCHLD是允许被忽略的。（可以通过 man wait 查看“如果父进程忽略这个信号会出现什么情况”，看不懂请问老师）

现在，我们学习下如何发送、处理、忽略中断信号。

发送中断信号：

代码中可以通过 int kill(pid_t pid, int sig) 发送中断信号，其中pid代表要接收进程的进程号，sig代表要发送的中断信号的中断号。

处理中断信号：

signal()虽然长得非常像发送信号的方法，但我们已经有kill()了，所以signal()并不是用来发送信号的。并且signal()也不是用来获取信号的。

signal()实际上是用**来注册信号的处理方式**的，`sighandler_t signal(int signum, sighandler_t handler)`; 中signum代表中断号，handler则是接到signum对应的中断信号后的**响应方式**，handler的值可以是SIG_IGN，SIG_DFL 或者一个函数。当handler是一个函数时，一旦进程接收到signum对应的中断信号，则handler会代替默认的中断处理方式进行处理。

忽略中断信号：

通过 `signal(signum, SIG_IGN)`; 可以忽略signum对应的中断信号。请注意 SIGKILL 和 SIGSTOP 不能被忽略。

请阅读并运行以下代码

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
void child_ding(){
    printf("child ding~~~~~\n");
}

void ctrlc_ding(){
    printf("ctrl+c ding~~~~~\n");
}

int main(){
    if(fork()){
        signal(SIGALRM, child_ding);
        signal(SIGINT, ctrlc_ding);
        signal(SIGCHLD, SIG_IGN);
        while(1){}
    }else{
        while(1){
            sleep(2);
            kill(getppid(),SIGALRM);
        }
    }
}
```

观察运行结果，尝试点击ctrl+c并观察结果。（还可以尝试用另一个terminal通过kill发送中断）

第七步. 创建ucore process

在之前的实验中，我们的ucore代码只是简单的启动了一个最小化的操作系统。接下来我们希望可以帮助让操作系统运行不同的进程，这节课我们来学习一下ucore是如何**创建并管理进程**的。（注：在本次lab代码中包含有关于内存管理的代码，这部分代码会在之后的实验中进行介绍，本次不做介绍）

定义进程控制块PCB

在ucore中，使用了大量的结构体用于管理各项资源，以下是用于管理进程的结构体：

```
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。（具体的实现会在之后的实验中介绍。）
- `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。（关于页表会在之后的实验中具体介绍）

这里特别注意一下`list_entry_t`链表，这里通过以下两个宏定义实现了当得到一个`list_link`时获得整个`proc_struct`结构体指针的功能：

```

#define le2proc(le, member)              \
    to_struct((le), struct proc_struct, member)

#define to_struct(ptr, type, member)     \
    ((type *)((char *) (ptr) - offsetof(type, member)))

```

那么为什么不直接为结构体增加一个`next`指针使之成为一个链表呢？在之后的实验中，我们还需要管理各种各样的资源，现在的实现方式可以让各种资源结构体都包含一个`list_entry_t`就可以实现各种资源的管理链表了，且不用为各链表单独实现插入删除等方法。

管理0号进程idle进程

操作系统启动后执行了`proc_init()`，在这个方法中创建了一个进程结构体`idleproc`用于**管理初始进程0号进程**。代码首先分配了结构体需要的内存空间，分配成功后对idle进程的控制块进行了初始化。

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。

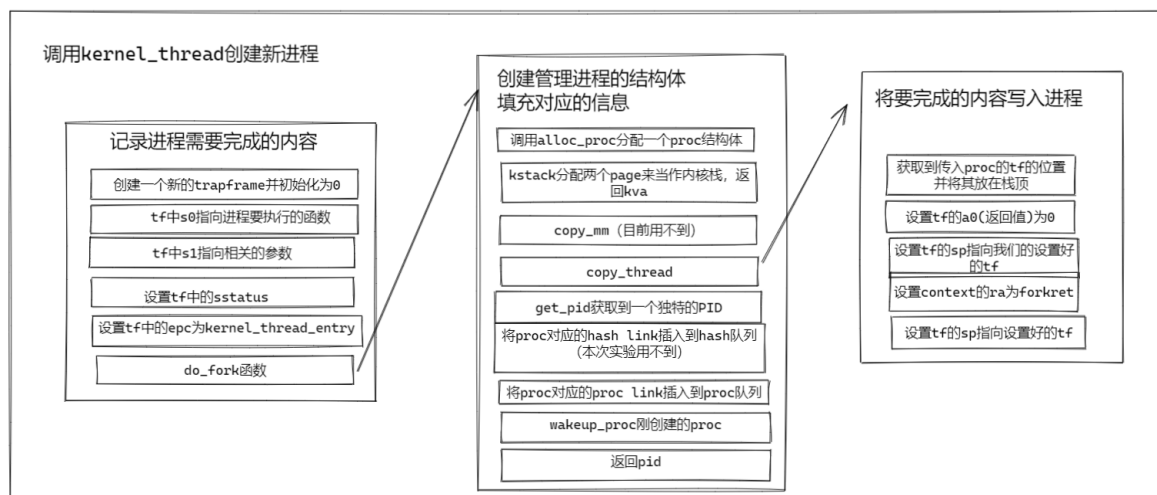
1号内核进程

接下来我们来创建一个1号内核进程。在本次实验中，我们让1号进程输出一个"Hello world!!"。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。



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。

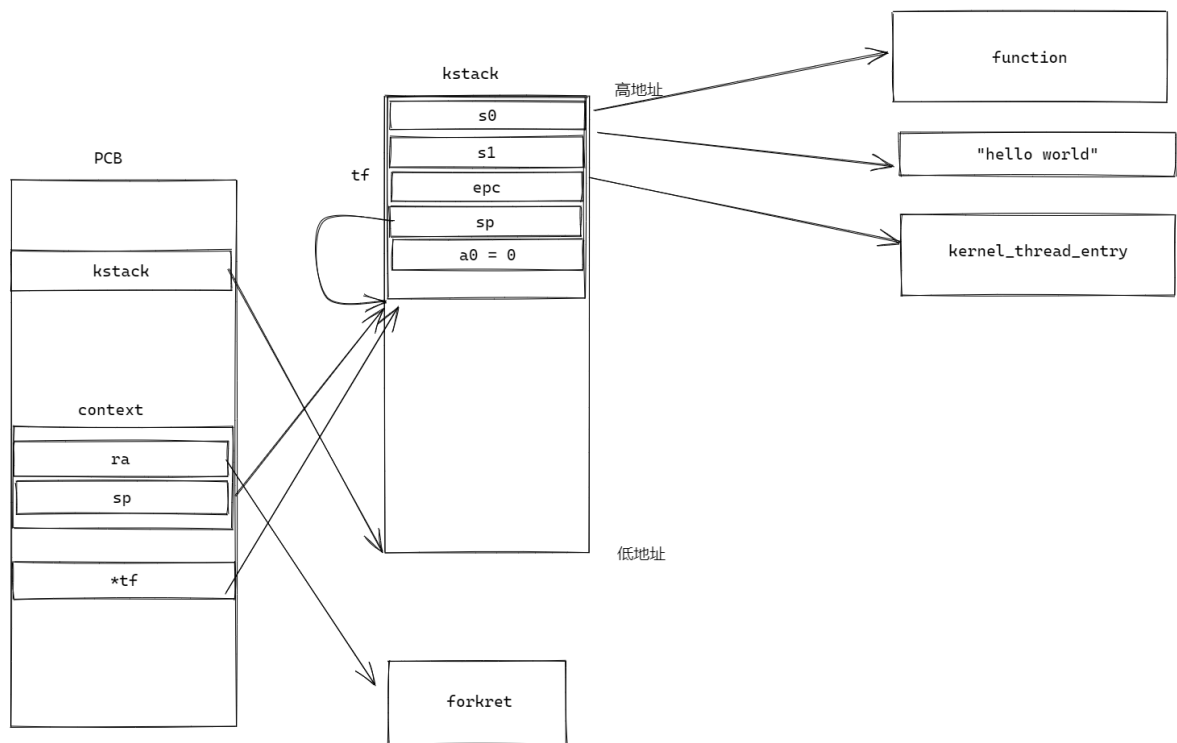
最后，调用proc.c中的 do_fork 函数把当前的进程复制一份。

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

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。

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。

进程切换

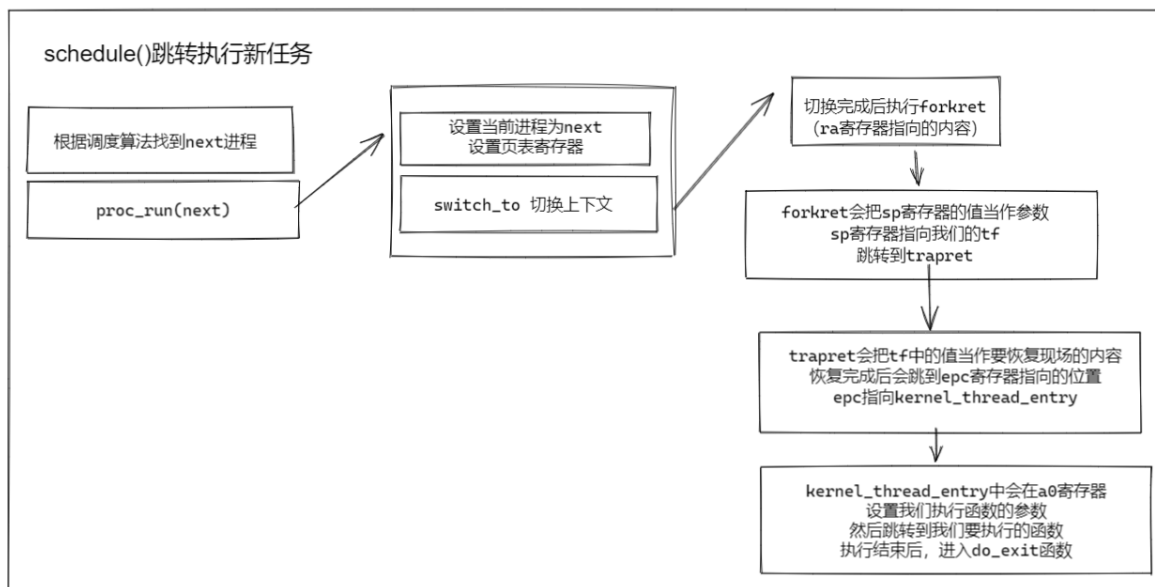
proc_init()执行结束后, 我们拥有了两个进程0号进程idleproc和1号进程init_main, 操作系统启动后运行的进程是0号进程, 1号进程存在于我们的[进程管理链表](#)中。

接下来操作系统会执行cpu_idle()

```
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。

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;
}
```

六、本节知识点回顾

在本次实验中，你需要了解以下知识点：

1. 如何创建进程
2. 父子进程的区别
3. 如何回收子进程的资源
4. 如何使用exec家族
5. 发送、处理、忽略中断信号
6. ucore进程创建与管理

七、下一实验简单介绍

在下次实验中，我们将创建用户进程。