Lab: user-level threads and alarm

Git链接：[Jeery1/xv6-6.S081 at syscall](https://github.com/Jeery1/xv6-6.S081/tree/syscall)

**实验目的：**

本实验旨在通过实现 用户级线程切换 和 定时器警报（Alarm） 机制，深入理解以下核心操作系统概念：

1. 用户级线程（User-level Threads）

目标：在用户空间实现线程的创建、切换和调度，不依赖内核线程支持。

关键技能：理解线程上下文（Context）的保存与恢复（寄存器、栈指针等）。

实现线程切换的协作式（Cooperative）或抢占式（Preemptive）调度。管理线程控制块（TCB）和线程栈。

实验任务：实现 thread\_create()：创建新线程并分配栈空间。实现 thread\_schedule()：通过汇编保存/恢复上下文完成线程切换。

2. 定时器警报（Alarm）

目标：实现类似 UNIX alarm() 的功能，允许用户程序注册定时回调。

关键技能：理解硬件定时器（如 RISC-V 的 CLINT）和时钟中断。

处理信号（或回调）与用户程序的交互。

实验任务：实现 sigalarm(interval, handler)：注册定时处理函数。修改内核的时钟中断处理逻辑，在指定时间间隔触发用户态回调。确保回调结束后恢复原程序执行流。

**实验步骤：**

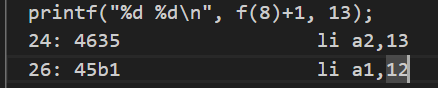
首先回答以下四个问题：

1. 哪个寄存器用于函数传参？对于本例，那个寄存器将13传给了printf



register a2

1. f函数和g函数分别在哪里被调用了？



可以看到，f(8)+1是一个inline function，于是在26: 45b1 li a1,12直接调用了f与g，将值算了出来（为12）

1. printf函数在哪个位置？



5c8

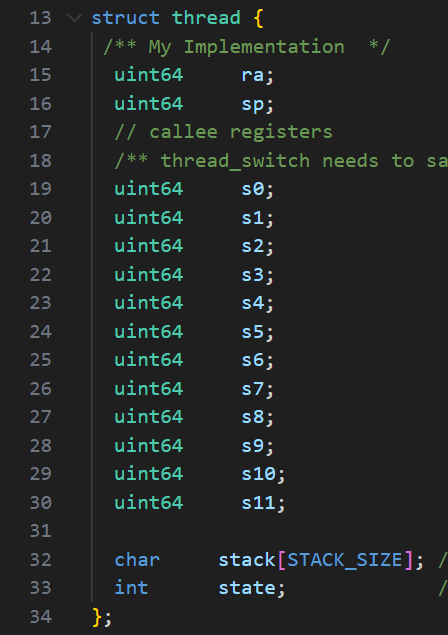
4. 执行完3a: 00000097 auipc ra,0x0后，ra中保存了什么？



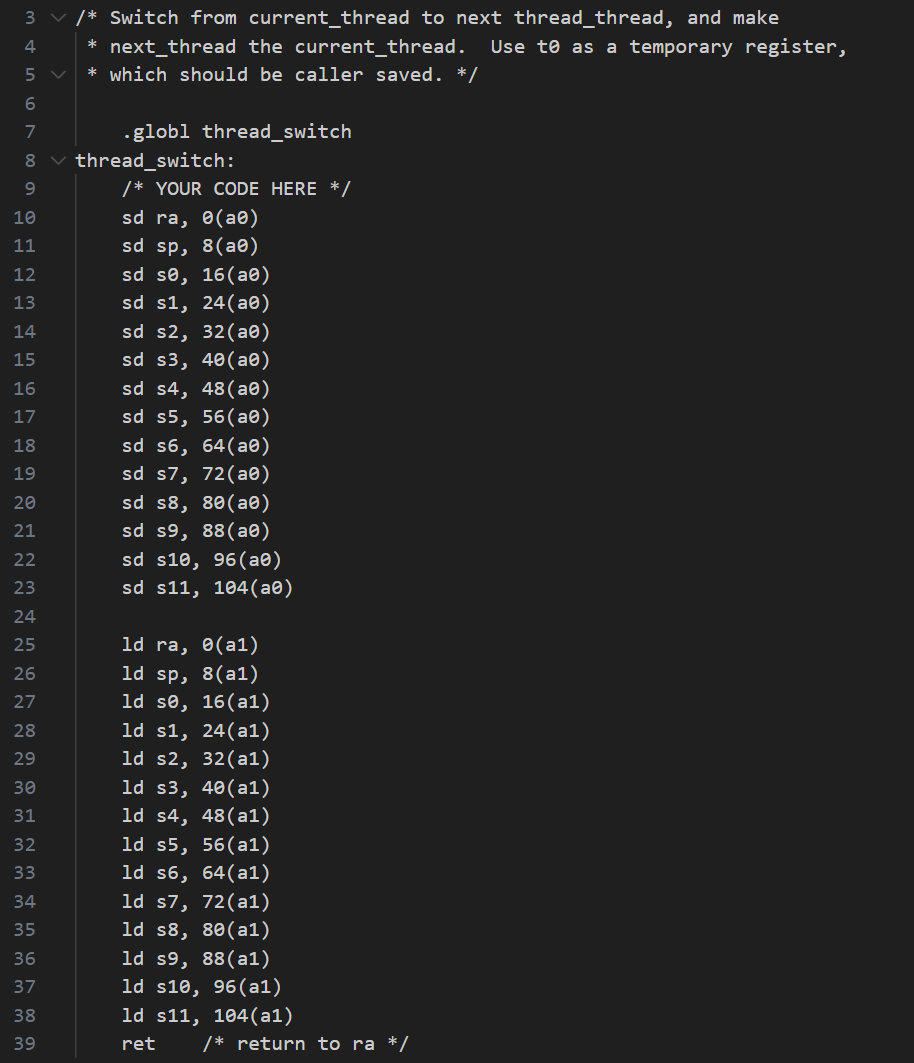
将当前pc的值加上0x0送入ra，故此时ra为3a。

Uthread: switching between threads

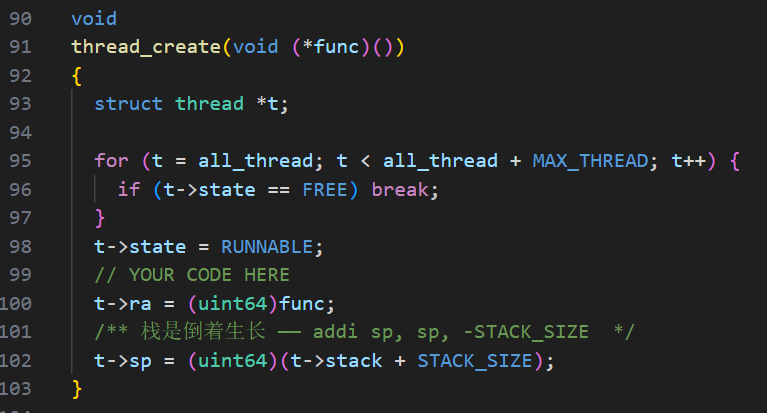
我们可以参考进程上下文切换的方式来完成线程上下文的切换。首先修改uthread.c中的thread定义如下：



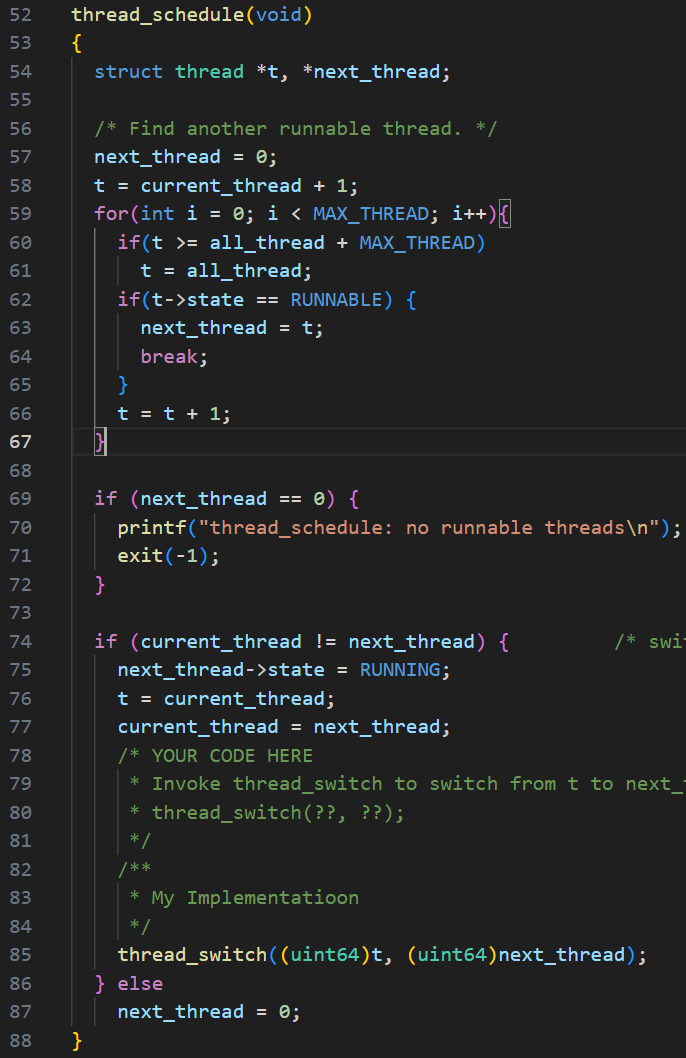
然后模仿进程上下文的切换，在uthread\_Switch.S中补充线程上下文的切换：



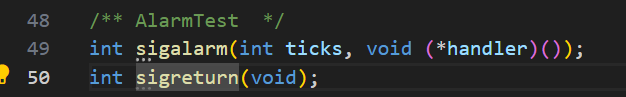
接下来，修改thread\_create，使之能够记录线程的返回地址ra与栈地址sp。其中，返回地址意味着当切换线程时，线程应该返回到什么哪个地址。这里应该是传入的函数的入口func。代码如下：



最后，在thread\_schedule中添加thread\_switch调用：



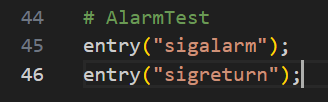
至此，我们便完成了Uthread。



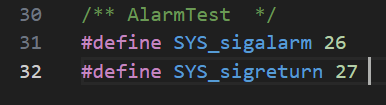
首先，我们在user/user.h声明Alarm系统调用接口：

然后，参考user/usys.pl、 kernel/syscall.h、 kernel/syscall.c的写法，添加sigalarm与sigreturn的系统调用：

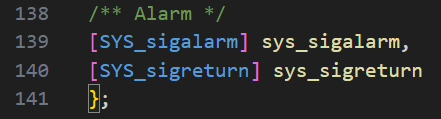
user/usys.pl



kernel/syscall.h



kernel/syscall.c



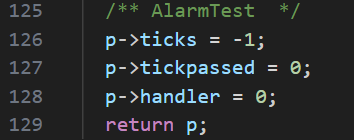
然后，我们可以在 kernel/sysfile.c中实现sigalarm与sigreturn的系统调用：

下面，我们将sys\_sigalarm补充完整。首先要明确sigalarm(int ticks, void (\*handler)())函数的作用：每ticks秒，执行一次handler函数。现在，我们遇到了第一个问题，如何传参？参考kernel/sysfile.c中其他系统调用函数的实现，我们可以很容易知道依靠argint、argaddr等函数即可实现参数传递。接下来，根据指导书给出的几个Hints，我们可以很快完成sys\_sigalarm的实现。

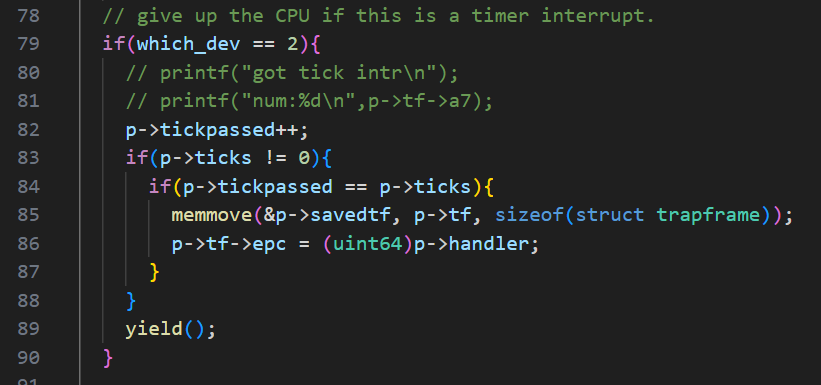
首先在在proc.h中对proc添加如下定义：



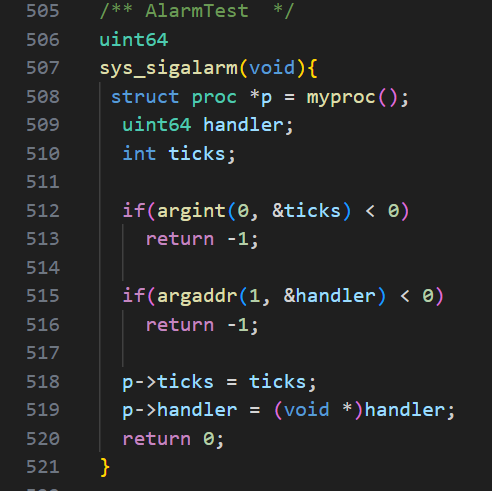
接着，在proc.c中的allocproc初始化这几个字段：



然后再trap.c中的时钟中断里修改进程proc的时钟状态，当流逝的时间等于预设的时间时，我们便调用handler函数，需要注意的是，由于中断发生在内核态，因此我们不能直接用p->handler()调用handler函数，因为用户态与内核态的函数地址不同，这里只能通过p->tf->epc = (uint64)p->handler将下一条指令的地址修改为handler地址才能正常运行。



接下来，开始正式实现sys\_sigalarm。很简单，利用argint与argaddr获得参数后更新进程字段即可。

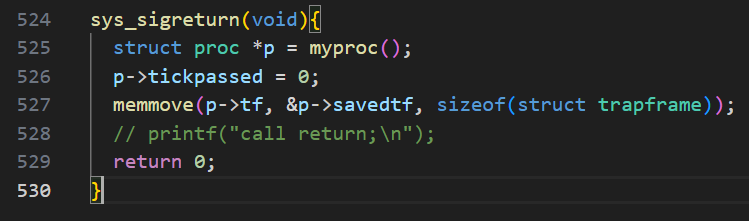


在这一阶段，我们主要需要实现sigreturn。根据实验指导书的Hints，我们需要在执行handler之前将用户进程的地址保存起来，然后在调用sigreturn后恢复到调用handler之前的状态。为了实现这一点，指导书给出了一个非常重要的Hint：

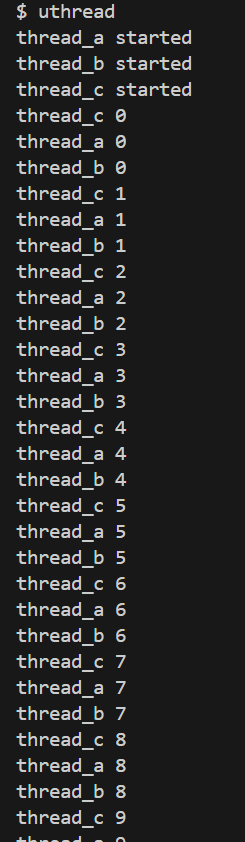
Your solution will require you to save and restore registers—what registers do you need to save and restore to resume the interrupted code correctly? (Hint: it will be many).

Hint中讲到，要保存许多寄存器。于是我的想法是干脆直接将整个trapframe保存起来，那里面含有足够多的寄存器。于是，向proc添加了新的字段，该字段用于保存trapframe:

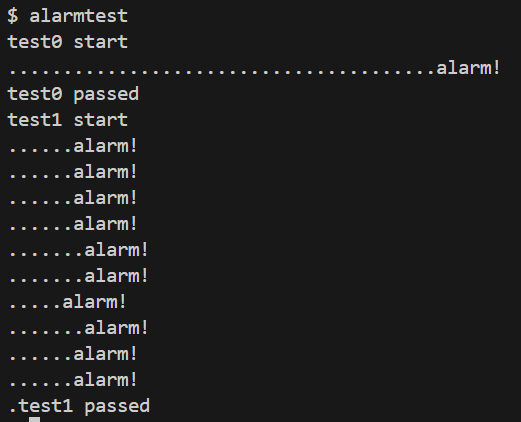
最后，填充sigreturn，使之能够将保存的trapframe重新恢复回来，代码如下：



**实验目的：**



运行uthread



运行alarmtest

**实验中遇到的问题和解决办法：**

在trap.c中的时钟中断里，当流逝的时间等于预设的时间时，我们应该调用handler函数，但是一开始调用发生错误。

分析后发现，由于中断发生在内核态，因此我们不能直接用p->handler()调用handler函数，因为用户态与内核态的函数地址不同，这里只能通过p->tf->epc = (uint64)p->handler将下一条指令的地址修改为handler地址才能正常运行。



**实验心得：**  
通过本次xv6用户级线程和定时器实验，我对操作系统的并发机制有了更深刻的认识。在实现用户级线程库的过程中，最让我印象深刻的是上下文切换的精细控制——通过精心设计context结构体和汇编级的寄存器操作，实现了轻量级的线程切换。调试过程中遇到的栈指针错误让我深刻理解了线程执行环境的脆弱性，必须确保所有关键寄存器都能被正确保存和恢复。而定时器中断与信号机制的配合使用，则让我体会到用户态和内核态如何协同工作来实现抢占式调度。这次实验不仅让我掌握了线程调度、临界区保护等核心概念，更让我认识到并发编程中资源管理的复杂性，比如线程退出时的内存回收问题就让我对资源生命周期有了新的思考。这些经验为我后续学习更高级的并发模型（如协程、多核调度）打下了坚实基础，也让我对操作系统如何优雅地管理并行执行有了更深层次的理解。