# 操作系统第一次实验

[操作系统第一次实验 1](#_Toc148794544)

[基础实现 1](#_Toc148794545)

[实现简述 1](#_Toc148794546)

[线程的创建与初始化 2](#_Toc148794547)

[上下文的切换 2](#_Toc148794548)

[FIFO队列的维护 4](#_Toc148794549)

[调度逻辑的实现 4](#_Toc148794550)

[Challenge 5](#_Toc148794551)

[thread\_swtich里只保存了整数寄存器的上下文。如何拓展到浮点数？ 6](#_Toc148794552)

[上面我们只实现了一个1 kthread :n uthread的模型，如何拓展成m : n的模型呢 7](#_Toc148794553)

[上述的实现是一个非抢占的调度器，如何实现抢占的调度呢？ 7](#_Toc148794554)

[在实现抢占的基础上，如何去实现同步原语（例如，实现一个管道channel） 8](#_Toc148794555)

[过程性的思考 9](#_Toc148794556)

[主线程的维护 9](#_Toc148794557)

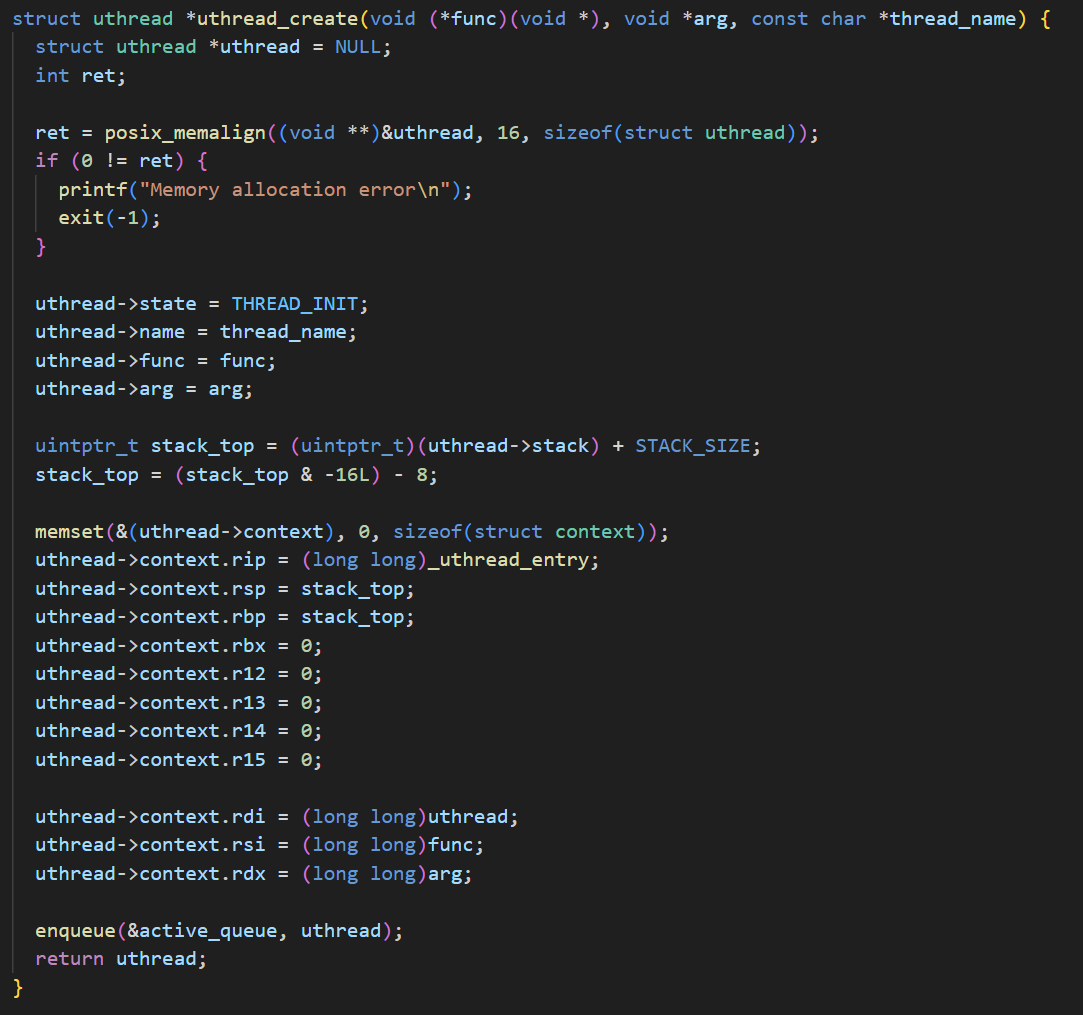
[代码结构的管理 9](#_Toc148794558)

## 基础实现

### 实现简述

本次实验要求是实现一个简单的多线程调度功能，重点内容在于线程的创建与初始化、上下文的切换、FIFO队列的维护以及调度逻辑的实现。下面我将对上述几点做详细介绍。

#### 线程的创建与初始化



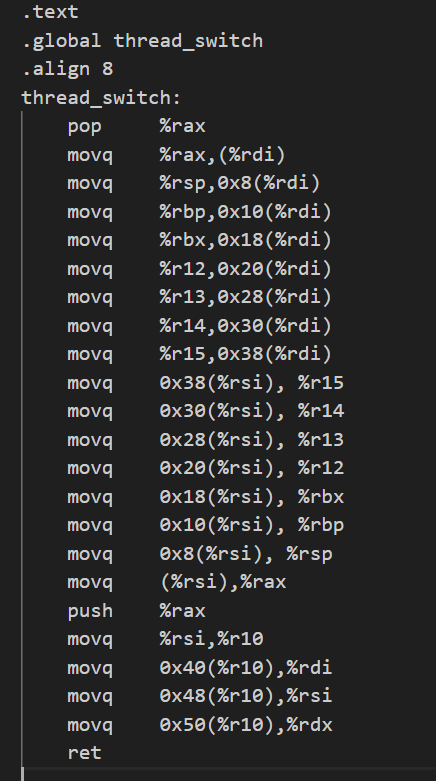
这个功能由函数uthread\_create实现，需要完成两件事：

1. 初始化线程的相关参数，例如state, name, func, arg, context，这些参数在上下文切换时需要保存与恢复。
2. 将初始化的线程加入队列，等待调度。

另外，在这个函数中还需要实现8字节对齐，在实验代码中，使用stack\_top = (stack\_top & -16L) - 8;实现。

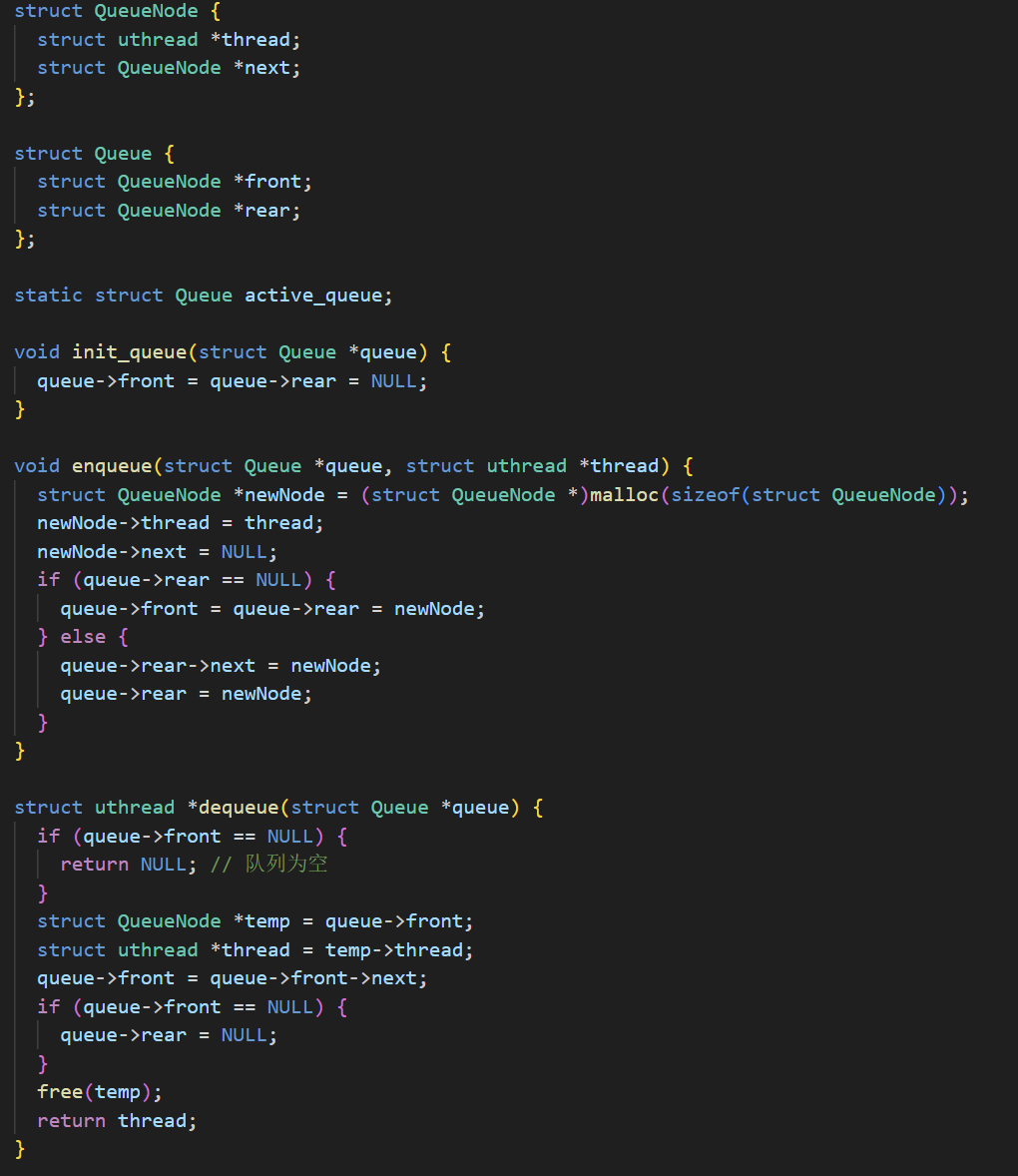
#### 上下文的切换

这个功能在助教学长给出的switch.S文件中已经有非常好的实现，



我在实验中使用了thread\_switch代码。

#### FIFO队列的维护

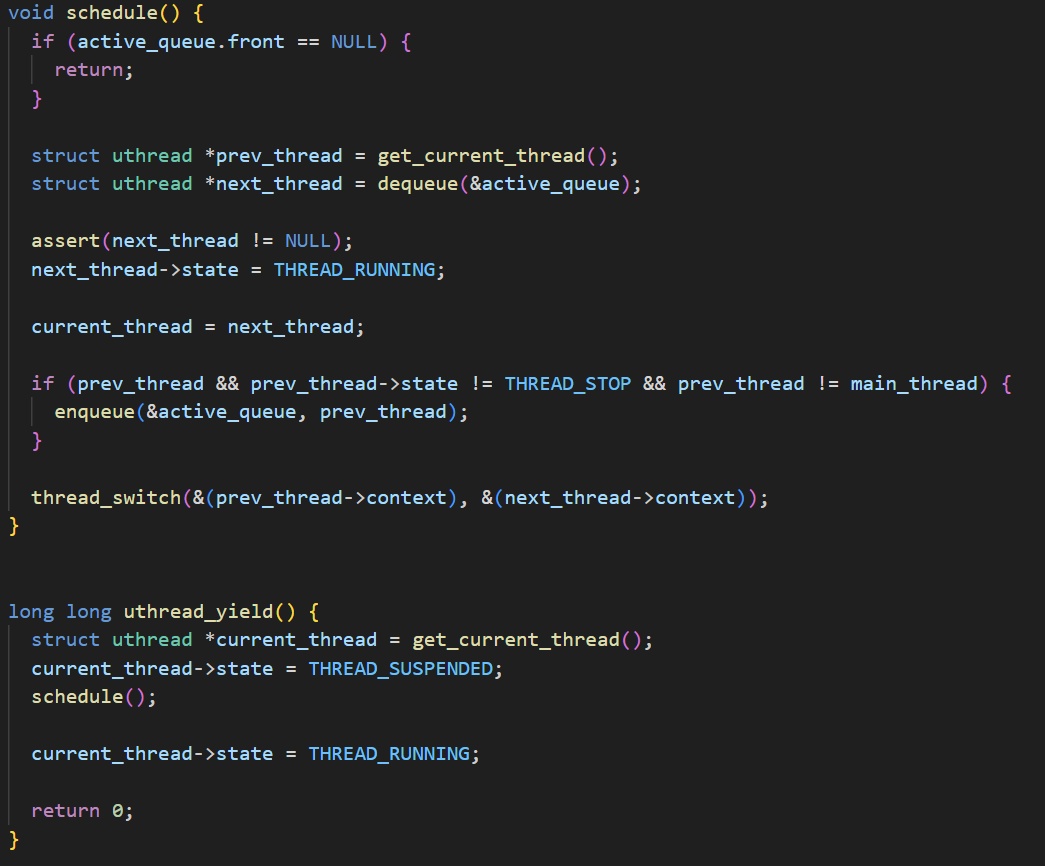


FIFO队列在本次实验的调度逻辑中是最重要的数据结构，它控制了线程执行的顺序，并且以队列是否为空指示子线程是否全部退出。需要说明的是，为了保证代码实现的简洁性，队列不对主线程做维护，与主线程相关的切换在调度器schedule中实现。

要实现一个FIFO队列，首先需要实现一个队列的节点QueueNode，这个数据结构包含它指向的一个线程的指针，以及指向下一个节点的指针。然后可以实现队列Queue，在我的实现中，这个数据结构只包含头指针和尾指针。

#### 调度逻辑的实现

调度逻辑主要由两个函数联合实现，分别是schedule和uthread\_yield



1. 最基础的调度器为schedule函数，FIFO的出队、入队操作只在这里进行以保证程序逻辑的清晰。具体来说，这个函数的执行逻辑为：
   1. 判断队列是否为空，若为空则直接返回
   2. 获取当前线程prev\_thread
   3. 以dequeue的方式获取队列的队首指向的线程，也就是要切换到的线程next\_thread
   4. 将需要切换到的线程状态设置为THREAD\_RUNNING，然后把current\_thread设置为next\_thread
   5. 判断线程是否符合入队条件（非NULL、非STOP、非主线程），然后入队prev\_thread
   6. 切换线程
2. 让渡控制权的实现
   1. 在uthread\_yield函数中，首先将current\_thread挂起，也就是将其state设置为THREAD\_SUSPENDED
   2. 然后将控制权转移给调度器schedule
   3. 在调度结束后，重新将current\_thread的state设置为THREAD\_RUNNING

## Challenge

以上是本次实验基础实现部分的代码逻辑实现的介绍，下面将从理论以及代码上对以下四个challenge问题进行回答：

### thread\_swtich里只保存了整数寄存器的上下文。如何拓展到浮点数？

要拓展thread\_switch函数以保存和恢复浮点寄存器的上下文，需要添加对XMM寄存器（用于SSE和AVX指令集中的浮点操作）的保存和恢复操作。以下是一个修改后的thread\_switch函数，包括了XMM寄存器的上下文保存和恢复：

.text

.global thread\_switch

.align 8

thread\_switch:

pop %rax

movq %rax, (%rdi)

movq %rsp, 0x8(%rdi)

movq %rbp, 0x10(%rdi)

movq %rbx, 0x18(%rdi)

movq %r12, 0x20(%rdi)

movq %r13, 0x28(%rdi)

movq %r14, 0x30(%rdi)

movq %r15, 0x38(%rdi)

movaps %xmm0, 0x40(%rdi)

movaps %xmm1, 0x50(%rdi)

movaps %xmm2, 0x60(%rdi)

movaps %xmm3, 0x70(%rdi)

movaps 0x40(%rsi), %xmm0

movaps 0x50(%rsi), %xmm1

movaps 0x60(%rsi), %xmm2

movaps 0x70(%rsi), %xmm3

movq 0x38(%rsi), %r15

movq 0x30(%rsi), %r14

movq 0x28(%rsi), %r13

movq 0x20(%rsi), %r12

movq 0x18(%rsi), %rbx

movq 0x10(%rsi), %rbp

movq 0x8(%rsi), %rsp

movq (%rsi), %rax

push %rax

movq 0x80(%r10), %rdi

movq 0x88(%r10), %rsi

movq 0x90(%r10), %rdx

ret

### 上面我们只实现了一个1 kthread :n uthread的模型，如何拓展成m : n的模型呢

在现有的基础上，如果要实现m:n的多线程模型，需要在主要要在线程初始化与管理、控制权让渡、调度策略上修改。

在初始化时，需要先创建多个kernel\_threads，可以像这样实现：

void init\_uthreads(int num\_kernel\_threads) {

main\_thread = malloc(sizeof(struct uthread));

main\_thread->state = THREAD\_RUNNING;

main\_thread->name = "main";

make\_dummpy\_context(&main\_thread->context);

current\_thread = main\_thread;

init\_queue(&active\_queue);

kernel\_threads = malloc(num\_kernel\_threads \* sizeof(pthread\_t));

for (int i = 0; i < num\_kernel\_threads; i++) {

pthread\_create(&kernel\_threads[i], NULL, kernel\_thread\_func, NULL);

}

}

对于user\_threads的管理需要在uthread\_create做出修改，大致的设计是，将多个user\_threads尽可能均匀地分配给kernel\_threads，在每个kthread内部的调度与现有调度一致，但是还要实现kthread间的调度，一种简单的实现方式是：使用与uthread相同的调度策略来调度m个kthread，在每个kthread被执行时，kthread调度出一个uthread来执行。这样就实现了一个round robin策略。

### 上述的实现是一个非抢占的调度器，如何实现抢占的调度呢？

要实现抢占式调度器，有很多种方法，比如引入计时器中断或者外部中断，总体的设计思想就是在满足某种条件时让当前执行的uthread退出，将控制权让渡到按优先级排序的下一个线程，这里针对引入计时器中断的方式详细介绍。

在uthread.c的实现中加上定时器中断的信号处理程序timer\_interrupt\_handler，

void timer\_interrupt\_handler(int signo) {

uthread\_yield(); // 在定时器中断时切换线程

}

它接收一个signo，然后执行uthread\_yield函数，这样就实现了一个计时器中断控制的抢占式调度。

还要加上初始化定时器中断的函数：

void init\_timer\_interrupt() {

struct itimerval timer;

struct sigaction sa;

memset(&sa, 0, sizeof(sa));

sa.sa\_handler = timer\_interrupt\_handler;

sigaction(SIGALRM, &sa, NULL);

timer.it\_value.tv\_sec = 0;

timer.it\_value.tv\_usec = 100000;

timer.it\_interval.tv\_sec = 0;

timer.it\_interval.tv\_usec = 100000;

setitimer(ITIMER\_REAL, &timer, NULL);

}

然后在需要使用多线程的代码的main函数中加上init\_timer\_interrupt();语句即可使用计时器中断控制的抢占式调度。

### 在实现抢占的基础上，如何去实现同步原语（例如，实现一个管道channel）

对于一个管道而言，可以这样简单定义它的结构：

#define PIPE\_CAPACITY 10

typedef struct {

int buffer[PIPE\_CAPACITY];

int count;

int write\_index;

int read\_index;

pthread\_mutex\_t mutex;

pthread\_cond\_t not\_full;

pthread\_cond\_t not\_empty;

} Pipe;

有三个容易实现的管道的基本方法：init、write、read，可以像这样简单实现：

void pipe\_init(Pipe\* pipe) {

pipe->count = 0;

pipe->write\_index = 0;

pipe->read\_index = 0;

pthread\_mutex\_init(&pipe->mutex, NULL);

pthread\_cond\_init(&pipe->not\_full, NULL);

pthread\_cond\_init(&pipe->not\_empty, NULL);

}

void pipe\_write(Pipe\* pipe, int data) {

pthread\_mutex\_lock(&pipe->mutex);

while (pipe->count >= PIPE\_CAPACITY) {

pthread\_cond\_wait(&pipe->not\_full, &pipe->mutex);

}

pipe->buffer[pipe->write\_index] = data;

pipe->write\_index = (pipe->write\_index + 1) % PIPE\_CAPACITY;

pipe->count++;

pthread\_cond\_signal(&pipe->not\_empty);

pthread\_mutex\_unlock(&pipe->mutex);

}

int pipe\_read(Pipe\* pipe) {

pthread\_mutex\_lock(&pipe->mutex);

while (pipe->count == 0) {

pthread\_cond\_wait(&pipe->not\_empty, &pipe->mutex);

}

int data = pipe->buffer[pipe->read\_index];

pipe->read\_index = (pipe->read\_index + 1) % PIPE\_CAPACITY;

pipe->count--;

pthread\_cond\_signal(&pipe->not\_full);

pthread\_mutex\_unlock(&pipe->mutex);

return data;

}

以上是管道的基本函数，在init时，引入了互斥锁，用于同步线程的读写操作。在实际的多线程任务中，可以通过管道实现线程间通信，由于实验的时间原因，本次实验并没有修改代码来实现线程间通信。

## 过程性的思考

### 主线程的维护

在一开始写代码时，我认为在维护FIFO队列时理论上要把主线程也维护进去，因为这样可以实现这样的调度a->main->b->main->c->…，实际调试过程中发现在上下文切换时还需要额外判断线程是否为主线程，所以维护主线程看起来是多余的，只需要在所有子线程都退出时切换到主线程就可以了。

### 代码结构的管理

纯粹按照逻辑流程写出的代码一般是比较乱的，在最早的实现中，我在schedule、uthread\_yield、\_uthread\_entry三个函数中都进行了enqueue和dequeue操作，导致代码结构十分混乱，调试也十分不方便，为了优化代码结构，便于调试，在提交的版本里，所有的enqueue、dequeue操作均在schedule函数中进行，保证了代码的可读性。