

Easy lab 实验报告

实验内容


我们已经提供了一个简单的代码框架在：`uthread.h` 和 `uthread.c`。

具体来说，你需要实现这样的用户态线程框架：

- 用户程序调用 `uthread_create` 创建一个协程，放入调度队列中。
- 创建完全部的协程后，主线程调用 `schedule()` 阻塞进入调度程序，开始执行各个协程。调度采用FIFO(先进先出)的顺序
- 线程开始执行的时候，首先跳转到函数 `_uthread_entry` ,然后才进入对应的函数
- 当协程中的函数调用 `uthread_yield` 时，控制权转让给调度器
- 当调度器执行 `uthread_resume` 时，会重新在中断的地方开始执行
- 当调度器发现函数执行结束时，会调用 `thread_destory` 销毁结构体。

我的实现

本次实验的全部代码由Git管理，并已上传至Github私有仓库，如图所示：

**easy_lab_private** Private

Unwatch 1 Fork 0 Star 0 Code

coffee3699 white spce removed (correctly)ebc8499 · last week15 Commits

lab1

Go to file

Add file

.devcontainer	white spce removed (correctly)	last week
img	update document	3 weeks ago
.gitignore	Provided code (without modify)	last week
Makefile	white spce removed (correctly)	last week
README.md	white spce removed (correctly)	last week
demo.c	Provided code (without modify)	last week
pingpong.c	Provided code (without modify)	last week
recursion.c	Undo changes to recursion.c	last week
simple.c	Provided code (without modify)	last week
switch.S	white spce removed (correctly)	last week
uthread.c	remove whitespace	last week
uthread.h	Completion V1.0	last week

About

BUPTOS 2023 | Lab 1: uthread switching

buptos.github.io/homework.html

Readme

1 Branch

0 Tags

Activity

0 stars

1 watching

0 forks

Releases

No releases published

[Create a new release](#)

Packages

No packages published

[Publish your first package](#)

Languages

我的实现主要在uthread.c和uthread.h两个文件中。

uthread.h

这是用户态线程的头文件，它定义了相关的数据结构和接口：

1. 线程的状态 (enum thread_state) :

- `THREAD_INIT` : 表示线程已被初始化。
- `THREAD_RUNNING` : 表示线程正在运行。
- `THREAD_STOP` : 表示线程已停止。
- `THREAD_SUSPENDED` : 表示线程已挂起。

2. 线程上下文 (struct context) :

这个结构体保存了线程上下文，即保存和恢复线程时需要的所有寄存器的值。

3. 线程控制块 (struct uthread) :

- `stack` : 线程的堆栈。
- `context` : 线程的上下文。
- `state` : 线程的当前状态。
- `name` : 线程的名字。

```
1 struct uthread {  
2     char stack[STACK_SIZE];  
3     struct context context;  
4     enum thread_state state;  
5     const char *name;  
6 };
```

4. 队列数据结构 (自己实现的FIFO队列) :

- `QueueNode` : 代表队列中的一个节点，每个节点保存一个 `uthread` 指针。
- `Queue` : 代表整个队列，有一个指向第一个节点的指针和一个指向最后一个节点的指针。

5. 函数原型:

定义了如何创建线程、恢复线程、线程让出、销毁线程等功能的函数。

uthread.c

这个是实现用户态线程切换功能代码的文件，其中主要包含：

1. 全局变量声明

- `current_thread` : 当前运行的线程。
- `main_thread` : 主线程。
- `scheduling_queue` : 用于调度的队列。（后来定义的）

2. thread_switch

外部函数（即 `switch.S`），用于在两个线程之间切换上下文。

3. _uthread_entry

这是所有用户态线程开始执行的入口。首先，它设置线程的状态为 `THREAD_RUNNING`，然后调用用户定义的线程函数。当线程函数返回时，它设置线程的状态为 `THREAD_STOP`，将线程放入调度队列，并交出控制权给调度器。

4. make_dummpy_context

清空给定的上下文结构体。使用 `memset` 来清空内容。

5. uthread_create(自己实现)

创建一个新的用户态线程并返回其指针。

首先，它为新线程分配内存，然后初始化其上下文和栈。

```
1 // Initialize the context structure
2 memset(uthread, 0, sizeof(struct uthread));
3 make_dummpy_context(&uthread->context);
4
5 // Initialize & set up the stack
6 uthread->context.rdi = (long long)uthread; // First argument
7 uthread->context.rsi = (long long)func;    // Second argument
8 uthread->context.rdx = (long long)arg;     // Third argument
9 uthread->context.rip = (long long)_uthread_entry;
10 // 在这里要注意内存的16比特对齐，否则在后面调用入口函数时会出错。
11 uthread->context.rsp = ((long long)uthread->stack + STACK_SIZE) & -16L;
12
13 uthread->context.rsp -= 8;
14
15 // Set the status of the thread
16 uthread->state = THREAD_INIT;
17
18 // Set the thead name
19 uthread->name = thread_name;
```

最后，它将新线程放入调度队列。

```
1 // Enqueue the thread
2 enqueue(scheduling_queue, uthread);
```

6. schedule(自己实现)

调度函数负责选择下一个要运行的线程。它首先检查调度队列是否为空。如果为空，则程序退出。否则，它取出一个线程并检查其状态。如果线程的状态为 `THREAD_STOP`，则销毁它，否则使用 `uthread_resume` 恢复该线程。

```
1 struct uthread *next_thread = NULL;
2
3 // Dequeue the next thread to run (if the next thread's state is THREAD_STOP,
4 // destroy it, then dequeue again)
5 do {
6     if (isEmpty(scheduling_queue)) {
7         exit(0);
8     }
9     next_thread = dequeue(scheduling_queue);
10    if (next_thread->state == THREAD_STOP) {
11        thread_destroy(next_thread);
12        continue;
13    } else {
14        break; // Break out of the loop if the next thread is valid and not in the
15               // THREAD_STOP state
16    }
17 } while (1);
18
19 // Use uthread_resume to resume the next thread
20 uthread_resume(next_thread);
```

7. uthread_yield(自己实现)

当前线程让出控制权并放入调度队列。然后调用调度函数。

```
1 // Set the current thread's state to suspended
2 current_thread->state = THREAD_SUSPENDED;
3
4 // Enqueue the current thread
5 enqueue(scheduling_queue, current_thread);
6
7 // Call the scheduler to switch to another thread
8 schedule();
9
10 // The scheduler will switch to another thread, and when this thread is
11 // resumed, it will continue from here
```

8. uthread_resume(自己实现)

将控制权交给给定的线程。首先，它设置线程的状态为 `THREAD_RUNNING`。然后，它保存当前线程的上下文，并恢复给定线程的上下文。

```
1 // Set the next thread's state to running
2 tcb->state = THREAD_RUNNING;
3
4 struct uthread* prviou_thread = current_thread;
5
6 // Set the current thread to the next thread (tcb)
7 current_thread = tcb;
8 // printf("Resume--->Previous thread is thread %d.\n", (int)(intptr_t)prviou_t
9 // printf("Resume--->Current thread is thread %d.\n", (int)(intptr_t)current_t
10 // Call the scheduler to switch to the next thread
11 thread_switch(&prviou_thread->context, &current_thread->context);
```

9. thread_destroy

释放给定线程的内存。

10. init_uthreads(自己实现部分)

初始化用户态线程库。首先，为主线程分配内存并初始化其上下文。然后，它设置 `current_thread` 为 `main_thread`。最后，它初始化调度队列。

```
1 main_thread = malloc(sizeof(struct uthread));
2 make_dummpy_context(&main_thread->context);
3
4 // Set the current thread to the main thread
5 current_thread = main_thread;
6
7 // Initialize the queue for thread scheduling
8 scheduling_queue = createQueue();
```

11. createQueue, enqueue, dequeue, isEmpty (自己实现)

这是一个简单的队列实现。`createQueue` 创建一个新的空队列，`enqueue` 将一个项添加到队列的尾部，`dequeue` 从队列的前面移除一个项，而 `isEmpty` 检查队列是否为空。

```
1 Queue* createQueue() {
2     Queue* queue = (Queue*)malloc(sizeof(Queue));
3     if (queue == NULL) {
4         fprintf(stderr, "Memory allocation error.\n");
5         exit(1);
6     }
```

```

6     }
7     queue->front = queue->rear = NULL;
8     return queue;
9 }
10
11 void enqueue(Queue* queue, struct uthread* data) {
12     QueueNode* newNode = (QueueNode*)malloc(sizeof(QueueNode));
13     if (newNode == NULL) {
14         fprintf(stderr, "Memory allocation error.\n");
15         exit(1);
16     }
17     newNode->data = data;
18     newNode->next = NULL;
19     if (queue->rear == NULL) {
20         queue->front = queue->rear = newNode;
21         return;
22     }
23     queue->rear->next = newNode;
24     queue->rear = newNode;
25 }
26
27 struct uthread* dequeue(Queue* queue) {
28     if (isEmpty(queue)) {
29         fprintf(stderr, "Queue is empty. Cannot dequeue.\n");
30         exit(1);
31     }
32     struct uthread* data = queue->front->data;
33     QueueNode* temp = queue->front;
34     queue->front = queue->front->next;
35     if (queue->front == NULL) {
36         queue->rear = NULL;
37     }
38     free(temp);
39     return data;
40 }
41
42 int isEmpty(Queue* queue) {
43     return queue->front == NULL;
44 }

```

困难和思考

由于这是我第一次接触git，并且在之前的课程中从来没有接触过底层代码的编写和阅读，因此这个lab对我来说挑战很大，当然在做完之后也带给我很多收获，包括但不限于：

1. 熟练使用git进行代码管理

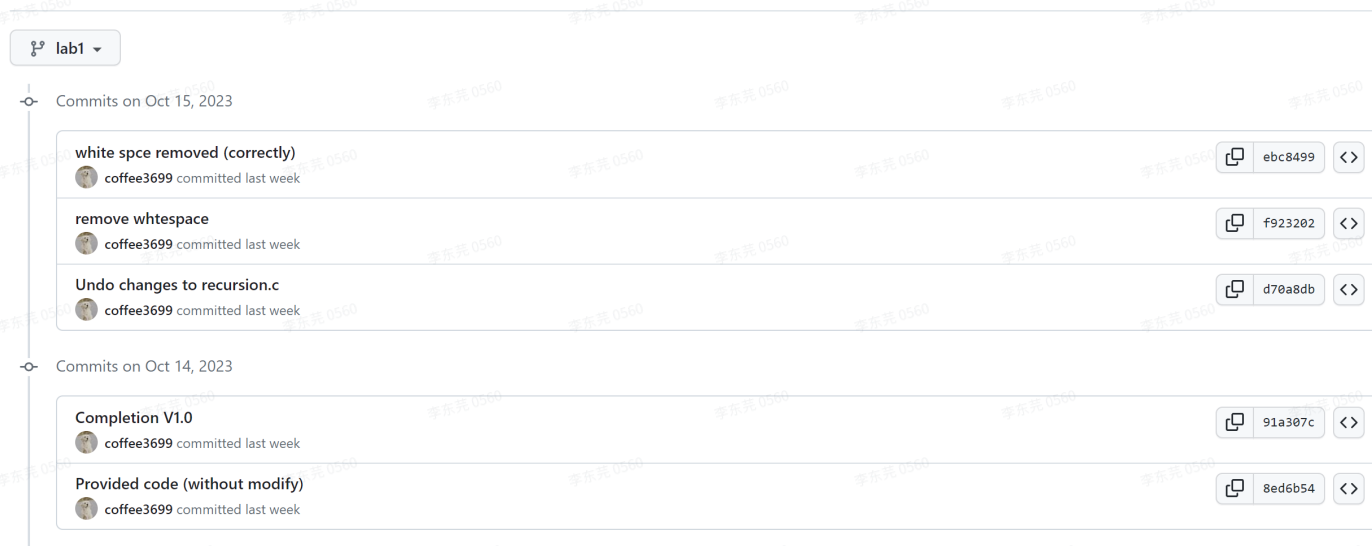
2. 使用docker和dev container管理开发环境
3. 熟练gdb的使用，以及vscode对gdb的可视化支持
4. 熟悉基本的汇编代码语法（AT&T syntax）
5. C语言中指针的应用
6. 用户态线程管理和切换的大致流程

在实现lab的过程中，我经历了学习课件，编写代码，思路卡壳，搜索相关博客，debug，de不出来等多个轮回，在做lab的过程中从一开始的框架代码到最后整体跑通，通过测试，极大地加深了我对于操作系统线程切换，内存管理等多个方面的认识，并且成功地激发了我对于操作系统的兴趣

我遇到的bug包括，不止一次的segfault，没有输出，程序意外停止，无法进入线程入口函数，无法提交patch等，其中的艰辛如图所示（这个图也不全面，我后来重新clone过一遍仓库）

熟练使用gdb对于debug非常有帮助（特别是backtrace），我目前也只是入门水平，还需要在以后继续熟练；

Commits



在这期间，我发现了来自网络上多篇优质的操作系统博客：https://xiayingp.gitbook.io/build_a_os/并且和助教保持了密切的沟通，在git的操作上给了我很多有用的建议和帮助，在此一并感谢

Challenge

以下是我对于lab中没有涉及到的问题的思考：

- **thread_switich里只保存了整数寄存器的上下文。如何拓展到浮点数？**

在线程上下文结构中添加一个新的字段 `fpu_state`，大小为512字节。

在switch.S中添加以下操作：

- 使用 `fxsave [destination]` 将当前的FPU（浮点单元）状态保存到目标内存位置。
- 使用 `fxrstor [source]` 从内存位置恢复FPU状态。

- **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1:k模型中，有一个内核线程支持多个用户态线程。要拓展到m:n模型，可以创建多个内核线程来执行用户态线程，我们可以引入一个线程池机制来管理内核线程。

- 设计一个内核线程池结构，其中包含内核线程的列表、状态信息等。
- 用户态线程在这些内核线程上调度和运行。
- 修改调度器，使其可以选择一个内核线程来执行用户态线程。当一个内核线程阻塞（如IO操作），另一个内核线程可以接管和继续执行用户态线程。

- **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？**

实现抢占式调度需要硬件支持（通常是时钟中断）以及操作系统级别的支持。

- 使用并初始化硬件定时器：x86提供了多种时钟源，一个常用的是 `Programmable Interval Timer` (PIT)。我们需要首先对其进行初始化，让其能够定期产生中断。
- 中断处理程序：在中断处理程序中，执行上下文保存，并调用调度器选择下一个线程进行调度（需要重新考虑调度策略，线程的优先级等）。

- **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）**

- 设计 `channel` 结构，它应该有一个缓冲区、读写指针、以及与此管道相关的等待线程的队列。
- 使用互斥锁来保护对管道的并发访问。
- 当管道为空且线程试图从中读取时，它应该被阻塞并加入到管道的等待线程队列中。
- 当有数据被写入管道时，检查等待队列，如果有线程在等待，则将其唤醒。