easy_lab1

1. 用户态线程实现

1.1 线程队列

调度器采用 FIFO 的顺序进行管理，所以我们需要实现一个线程队列，我通过链表这一数据结构来实现，核心函数为入队和出队操作。

```c
// 队列节点

struct QueueNode {

    struct uthread *thread;

    struct QueueNode *next;

};

// 队列结构
struct Queue {

    struct QueueNode *front;

    struct QueueNode *rear;

};

// 初始化队列
void initQueue(struct Queue *q) {

    q->front = q->rear = NULL;

}

// 入队列
void enqueue(struct Queue *q, struct uthread *thread) {
```

```
    struct QueueNode *newNode = (struct QueueNode
*)malloc(sizeof(struct QueueNode));

    if (newNode == NULL) {

        perror("Failed to allocate memory for queue node");

        exit(EXIT_FAILURE);

    }

    newNode->thread = thread;

    newNode->next = NULL;

    if (q->rear == NULL) {

        q->front = q->rear = newNode;

        return;

    }

    q->rear->next = newNode;

    q->rear = newNode;

}

// 出队列
struct uthread *dequeue(struct Queue *q) {

    if (q->front == NULL) {

        return NULL;

    }

    struct QueueNode *temp = q->front;

```

```
    q->front = q->front->next;

    if (q->front == NULL) {

        q->rear = NULL;

    }

    struct uthread *thread = temp->thread;

    free(temp);

    return thread;

}
```

1.2 线程创建（uthread_create）

- 这个函数用于创建一个协程，为每个协程分配内存，并初始化协程的状态和名称，同时设置协程的上下文（context）。
- 设置每个协程的上下文的寄存器状态 rsp（栈指针）、rip（函数入口地址）、rdi、rsi、rdx（传递参数），以便后续程序调用汇编代码。
- 最后将创建的协程被添加到调度队列中。

```c
struct uthread *uthread_create(void (*func)(void *), void
*arg,const char* thread_name) {

  struct uthread *uthread = NULL;

  int ret;



  // 申请一块16字节对齐的内存

  ret = posix_memalign((void **)&uthread, 16, sizeof(struct
uthread));

```

```
16    if (0 != ret) {
17
18      printf("error");
19
20      exit(-1);
21
22    }
23
24
25
26    //          +------------------------+
27
28    // low      |                        |
29
30    //          |                        |
31
32    //          |                        |
33
34    //          |          stack         |
35
36    //          |                        |
37
38    //          |                        |
39
40    //          |                        |
41
42    //          +------------------------+
43
44    //  high    |    fake return addr    |
45
46    //          +------------------------+
47
48
49
50    /*
51
52      TODO: 在这里初始化uthread结构体。可能包括设置rip,rsp等寄存
53    器。入口地址需要是函数_uthread_entry.
54
55          除此以外，还需要设置uthread上的一些状态，保存参数等
56
```

```
等。

        你需要注意rsp寄存器在这里要8字节对齐，否则后面从
context switch进入其他函数的时候会有rsp寄存器

        不对齐的情况（表现为在printf里面Segment Fault）

    */

    uthread->state = THREAD_INIT;

    uthread->name = thread_name;



    make_dummpy_context(&uthread->context);



    uthread->context.rsp = ((long long)&uthread->stack +
STACK_SIZE) & (~(long long)15);

    uthread->context.rsp -=8;

    uthread->context.rip = (long long)_uthread_entry;



    uthread->context.rdi = (long long)uthread;

    uthread->context.rsi = (long long)func;

    uthread->context.rdx = (long long)arg;



    //把新创建的线程加入到队列中

    enqueue(&readyQueue, uthread);
```

```
    return uthread;

}
```

1.3 线程调度（schedule）

- 这个函数负责协程的调度。
- 采用 FIFO（先进先出）调度策略，主线程首先被调度，然后不断从调度队列中取出协程进行执行。直到队列为空。
- 每次调度，将执行 resume 函数恢复协程的上下文。
- 当发现协程执行结束后，调用 thread_destory 函数销毁协程，并释放其内存。

```c
void schedule() {

  /*

  TODO: 在这里写调度子线程的机制。这里需要实现一个FIFO队列。这意味着你需要一个额外的队列来保存目前活跃

        的线程。一个基本的思路是，从队列中取出线程，然后使用resume恢复函数上下文。重复这一过程。

  */

  if (current_thread == NULL) {

    // 第一次调度，从主线程开始

    current_thread = main_thread;

    current_thread->state = THREAD_RUNNING;

  }

  //printf("%s\n",current_thread->name);


```

```
26
27      //不断取出队头，直到队列为空
28      while(1)
29      {
30        struct uthread *next_thread = dequeue(&readyQueue);
31
32
33        if (next_thread) {
34              uthread_resume(next_thread);
35              if(current_thread->state == THREAD_STOP)
36              {
37                //销毁线程
38                //printf("线程结束：%s\n",current_thread->name);
39                thread_destory(current_thread);
40              }
41              current_thread=main_thread;//将当前线程设置为主线程
42        } else {
43          //printf("队列为空\n");
44          break;
45        }
46      }
47
```

```
}
```

1.4 线程让出（uthread_yield）

- 这个函数用于协程主动让出 CPU 控制权。
- 当一个协程调用此函数时，它的状态被设置为挂起，然后将其放回调度队列，以便稍后再次执行。
- 同时调用线程切换函数将控制权转交给调度器，让调度器切换到其他协程执行。

```c
long long uthread_yield() {

    /*

    TODO: 用户态线程让出控制权到调度器。由正在执行的用户态函数来调用。记得调整tcb状态。

    */

    //printf("线程主动让出\n");

    // 设置当前线程的状态为挂起

    current_thread->state = THREAD_SUSPENDED;

    // 将当前线程加入到队列中

    enqueue(&readyQueue, current_thread);

    // 控制权转交回调度器

    //printf("线程切换回主线程\n");

    thread_switch(&current_thread->context, &main_thread->context);

    return 0;

}
```

1.5 线程恢复（uthread_resume）

- 这个用于从调度队列中恢复协程的执行。
- 调度器选择下一个要执行的协程后，调用这个函数，将其状态设置为运行，并调用线程切换函数切换到其上下文。

```c
void uthread_resume(struct uthread *tcb) {

    /*

    TODO: 调度器恢复到一个函数的上下文。

    */

    //printf("当前线程: %s\n",current_thread->name);

    tcb->state = THREAD_RUNNING;



    // 在恢复时，将当前线程设置为被恢复的线程

    //printf("下一线程: %s\n",tcb->name);

    current_thread = tcb;



    // 执行上下文切换，恢复线程的上下文

    thread_switch(&main_thread->context, &tcb->context);

}
```

1.6 函数入口

- 所有协程函数开始执行的入口，在这个函数中将 tcb 的状态设置为运行，并执行相应的函数，函数执行完毕后，将 tcb 状态设置为停止，切换回主线程。

```c
void _uthread_entry(struct uthread *tcb, void (*thread_func)
(void *),

                    void *arg) {

  /*

  TODO: 这是所有用户态线程函数开始执行的入口。在这个函数中，你
需要传参数给真正调用的函数，然后设置tcb的状态。

  */

  tcb->state = THREAD_RUNNING;

  thread_func(arg);

  tcb->state = THREAD_STOP;

  thread_switch(&tcb->context, &main_thread->context);

}
```

2. 实验中遇到的困难

2.1 前置知识学习成本大

在一开始做这个实验的过程中，因为一开始不熟悉汇编代码，不理解 thread_switch 这个函数怎么使用，即使询问了 gpt 也是一知半解，导致经常报错 Segmentation Fault。后来通过助教更新的 demo 阅读代码后才有了比较清晰的认知。而且因为之前没有学过计组和 CSAPP ，以及人工智能专业的课程里很少用到 C 语言，在看到这么多指针也是很头大，但是通过不断的摸索以及 GPT 的帮助下慢慢熟络并完成作业。

2.2 不会使用 GDB 调试

虽然之前程序设计课有部署过一个项目到服务器上有接触过 linux，知道一些基本的 linux 命令，但是从来没有使用过 GDB 这类调试工具，以至于每次报错 Segmentation Fault 的时候，我只能通过断点和在程序中 print 输出来慢慢调试错误，不能很快的定位错误。

2.3 内存管理

在创建链表队列和协程的时候，因为对内存管理的疏忽，经常导致运行时 Segmentation Fault。

3. 思考

通过这次 easy_lab（一开始其实并不 easy ），基于助教给出的代码，我从零到有的实现了一个基础的有栈协程框架，允许用户创建、调度和销毁协程。在这次实验中，我还学习到了汇编、寄存器等相关知识，对操作系统的理解更进了一步（有空真的要学 CSAPP，很有用），同时提高了 C 语言的编程水平。总而言之，这是一次十分考验综合能力很强的实验，布置这个实验的老师和助教老师们都非常厉害，很感谢有机会能完成这次实验!

4. Challenge

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

添加对XMM寄存器的保存和恢复操作。

```X86asm
.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)

    // 保存XMM寄存器上下文
    movaps  %xmm0, 0x40(%rdi)
    movaps  %xmm1, 0x50(%rdi)
    movaps  %xmm2, 0x60(%rdi)
    movaps  %xmm3, 0x70(%rdi)
    movaps  %xmm4, 0x80(%rdi)
    movaps  %xmm5, 0x90(%rdi)
    movaps  %xmm6, 0xA0(%rdi)
    movaps  %xmm7, 0xB0(%rdi)
    movaps  %xmm8, 0xC0(%rdi)
    movaps  %xmm9, 0xD0(%rdi)
```

```
23      movaps  %xmm10, 0×E0(%rdi)
24      movaps  %xmm11, 0×F0(%rdi)
25      movaps  %xmm12, 0×100(%rdi)
26      movaps  %xmm13, 0×110(%rdi)
27      movaps  %xmm14, 0×120(%rdi)
28      movaps  %xmm15, 0×130(%rdi)
29
30      // 恢复XMM寄存器上下文
31      movaps  0×40(%rsi), %xmm0
32      movaps  0×50(%rsi), %xmm1
33      movaps  0×60(%rsi), %xmm2
34      movaps  0×70(%rsi), %xmm3
35      movaps  0×80(%rsi), %xmm4
36      movaps  0×90(%rsi), %xmm5
37      movaps  0×A0(%rsi), %xmm6
38      movaps  0×B0(%rsi), %xmm7
39      movaps  0×C0(%rsi), %xmm8
40      movaps  0×D0(%rsi), %xmm9
41      movaps  0×E0(%rsi), %xmm10
42      movaps  0×F0(%rsi), %xmm11
43      movaps  0×100(%rsi), %xmm12
44      movaps  0×110(%rsi), %xmm13
45      movaps  0×120(%rsi), %xmm14
46      movaps  0×130(%rsi), %xmm15
47
48      movq    0×130(%rsi), %rax
49      push    %rax
50
51      movq    %rsi, %r10
52      movq    0×140(%r10), %rdi
53      movq    0×148(%r10), %rsi
54      movq    0×150(%r10), %rdx
55      ret
```

4.2 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?

1. 线程管理：扩展线程管理器以支持多个用户态线程。您需要创建一个线程管理数据结构，用于跟踪和管理所有用户态线程。这包括线程的创建、销毁、状态管理等。

2. **调度器**：实现一个调度器，以便在多个用户态线程之间进行切换。调度器应该能够根据一定的调度策略（例如，抢占式或协作式调度）选择下一个要运行的线程。
3. **上下文切换**：扩展上下文切换机制，以保存和恢复所有线程的上下文。这包括整数寄存器和浮点寄存器的上下文，以及堆栈。
4. **线程同步**：在多个用户态线程之间引入线程同步机制，以处理并发和避免竞态条件。这包括互斥锁、信号量、条件变量等。
5. **资源管理**：确保多个用户态线程正确地管理共享资源，例如堆内存分配、文件句柄等。
6. **异常处理**：处理线程中的异常，例如在用户态线程内的错误处理以及线程取消。
7. **初始化和清理**：编写初始化和清理函数，确保线程管理器和调度器在程序启动和结束时能够正确初始化和清理资源。

4.3 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?

1. **时钟中断**：使用操作系统提供的时钟中断或定时器来触发调度器的运行。时钟中断定期发生，即使线程没有主动让出 CPU，也可以强制进行调度。时钟中断通常以毫秒或微秒为单位配置，具体取决于操作系统和硬件。
2. **信号处理程序**：在接收到时钟中断时，操作系统可以调用一个特殊的信号处理程序（例如，`SIGALRM`），该处理程序用于触发线程调度。
3. **抢占**：当信号处理程序被触发时，它可以请求执行线程切换。这通常通过将当前执行的线程放入就绪队列并选择下一个要运行的线程来实现。
4. **上下文切换**：调度器可以执行上下文切换以切换到下一个要运行的线程。这涉及保存当前线程的上下文并加载下一个线程的上下文。
5. **线程同步**：为了防止竞态条件，必须在执行上下文切换时对线程同步进行妥善处理。例如，锁定共享资源以防止多个线程同时访问。
6. **状态管理**：确保调度器维护每个线程的状态，并可以正确地切换线程的状态，例如，从运行状态切换到挂起状态。
7. **优先级调度**：如果需要，可以实现线程的优先级调度，以确保高优先级线程在抢占时有更高的机会运行。

4.4 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）？

1. **数据结构**： 首先，您需要定义一个数据结构来表示管道 channel。这个数据结构应该包括缓冲区（用于存储数据项）、互斥锁（用于控制对缓冲区的访问）、条件变量（用于线程之间的通信）等。
2. **初始化**： 在创建管道 channel 时，需要初始化相关数据结构，包括分配缓冲区、初始化互斥锁和条件变量等。
3. **发送数据项**： 为了发送数据项到管道中，您可以编写一个发送函数，该函数将数据项添加到缓冲区中，然后通知等待的接收函数。

4. **接收数据项：** 为了从管道中接收数据项，您可以编写一个接收函数，该函数等待数据项的到来（使用条件变量），然后从缓冲区中取出数据项。
5. **线程同步：** 在发送和接收函数中，需要使用互斥锁来保护对共享资源的访问，以避免竞态条件。
6. **阻塞和唤醒：** 当没有数据项可供接收时，接收函数应该被阻塞，等待发送函数通知它有数据可供接收。这可以使用条件变量来实现。
7. **错误处理：** 考虑线程的退出和错误处理。当发送和接收函数执行时，需要处理线程退出和错误情况，以确保线程的正确终止和资源的释放。

一个伪代码示例：

```c
// 数据结构表示管道 channel
struct Channel {
    int buffer[MAX_SIZE];
    int size;
    int in;
    int out;
    Mutex mutex;
    CondVar notEmpty;
    CondVar notFull;
};

// 初始化管道
void initChannel(Channel* ch) {
    ch->size = 0;
    ch->in = 0;
    ch->out = 0;
    initMutex(&ch->mutex);
    initCondVar(&ch->notEmpty);
    initCondVar(&ch->notFull);
}

// 发送数据项到管道
void send(Channel* ch, int data) {
    acquireMutex(&ch->mutex);
    while (ch->size == MAX_SIZE) {
        waitCondVar(&ch->notFull, &ch->mutex);
    }
    ch->buffer[ch->in] = data;

```

```
    ch->in = (ch->in + 1) % MAX_SIZE;
    ch->size++;
    signalCondVar(&ch->notEmpty);
    releaseMutex(&ch->mutex);
}

// 从管道中接收数据项
int receive(Channel* ch) {
    int data;
    acquireMutex(&ch->mutex);
    while (ch->size == 0) {
        waitCondVar(&ch->notEmpty, &ch->mutex);
    }
    data = ch->buffer[ch->out];
    ch->out = (ch->out + 1) % MAX_SIZE;
    ch->size--;
    signalCondVar(&ch->notFull);
    releaseMutex(&ch->mutex);
    return data;
}
```