lab3-1718405l4张福翔

联系方式：499480213@qq.com

1. 实验环境

- 自制OS的CPU：Intel 80386
- 模拟80386环境的虚拟机：QEMU
- 交叉编译的编译器：GCC
- 调试工具：GDB
- 运行平台：Ubuntu 18.04
- 编程语言：C，X86 Assembly

2. 实验要求

本实验通过实现一个简单的任务调度，介绍**基于时间中断进行进程切换**以及**纯用户态的非抢占式的线程切换**完成任务调度的全过程。

2.1 实现进程切换机制

在内核中实现进程切换机制，并基于时间中断进行任务调度，具体流程如下

1. Bootloader从实模式进入保护模式,加载内核至内存，并跳转执行
2. 内核初始化IDT，初始化GDT，初始化TSS，初始化串口，初始化8259A，...
3. 启动时钟源，开启时钟中断处理
4. 加载用户程序至内存
5. 初始化内核IDLE线程的进程控制块（Process Control Block），初始化用户程序的进程控制块
6. 切换至用户程序的内核堆栈，弹出用户程序的现场信息，返回用户态执行用户程序

2.2 实现 `FORK`、`SLEEP`、`EXIT` 系统调用和 `pthread` 库

1. 实现 `FORK`、`SLEEP`、`EXIT` 系统调用，并使用用户程序进行测试。
2. 实现要求的 `pthread` 库函数，并使用以下用户程序测试，`main` 函数定义在 `pthread_test.c` 文件中，将 `pthread_test.c` 单独取出后添加相应的头文件是可以独立运行的（如没有 `pthread_yield` 接口，可以换成 `sched_yield`），有兴趣的同学可以尝试。

3. 实验原理

进程为操作系统资源分配的单位，每个进程都有独立的地址空间（代码段、数据段），独立的堆栈，独立的进程控制块；线程作为任务调度的基本单位，与进程的唯一区别在于其地址空间并非独立，而是与其他线程共享；以下为一个广义的进程（包括进程与线程）生命周期中的状态转换图：

- 进程由其父进程利用 `FORK` 系统调用创建，则该进程进入 `RUNNABLE` 状态
- 时间中断到来，`RUNNABLE` 状态的进程被切换到，则该进程进入 `RUNNING` 状态

- 时间中断到来，`RUNNING`状态的进程处理时间片耗尽，则该进程进入`RUNNABLE`状态
- `RUNNING`状态的进程利用`SLEEP`系统调用主动阻塞；或利用系统调用等待硬件I/O，则该进程进入`BLOCKED`状态
- 时间中断到来，`BLOCKED`状态的进程的`SLEEP`时间片耗尽；或外部硬件中断表明I/O完成，则该进程进入`RUNNABLE`状态
- `RUNNING`状态的进程利用`EXIT`系统调用主动销毁，则该进程进入`DEAD`状态

3.1 进程机制

3.1.1 `fork`系统调用

`FORK`系统调用用于创建子进程，内核需要为子进程分配一块独立的内存，将父进程的地址空间、用户态堆栈完全拷贝至子进程的内存中，并为子进程分配独立的进程控制块，完成对子进程的进程控制块的设置。

若子进程创建成功，则对于父进程，该系统调用的返回值为子进程的`pid`，对于子进程，其返回值为`0`；若子进程创建失败，该系统调用的返回值为`-1`。

3.1.2 `sleep`系统调用

sleep系统调用用于进程主动阻塞自身，内核需要将该进程由`RUNNING`状态转换为`BLOCKED`状态，设置该进程的sleep时间片，并切换运行其他`RUNNABLE`状态的进程。

3.1.3 `exit`系统调用

`EXIT`系统调用用于进程主动销毁自身，内核需要将该进程由`RUNNING`状态转换为`DEAD`状态，回收分配给该进程的内存、进程控制块等资源，并切换运行其他`RUNNABLE`状态的进程。

###

3.2 线程机制

3.2.1 `pthread_create`函数

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

`pthread_create`调用成功会返回`0`；出错会返回错误号，并且`*thread`的内容未定义；本实验要求出错返回`-1`。新创建的线程以`pthread_exit`结束。

3.2.2 `pthread_join`函数

```
int pthread_join(pthread_t thread, void **retval);
```

实验要求`pthread_join`调用成功返回`0`;出错返回`-1`.实际测试只考虑调用成功的情况。

`pthread_join`函数会等待`thread`指向的线程结束。

3.2.3 `pthread_yield`函数

```
int pthread_yield(void);
```

实验要求 `pthread_yield` 调用成功返回 `0` ;出错返回 `-1` .实际测试只考虑调用成功的情况。

`pthread_yield` 函数会使得调用次函数的线程让出CPU。

3.2.4 `pthread_exit` 函数

```
void pthread_exit(void *retval);
```

`pthread_exit` 函数会结束当前线程并通过 `retval` 返回值，该返回值由同一进程下 `join` 当前线程的线程获得。

4. 实验框架

```
.
├── app                                 # 用户程序
│   ├── main.c
│   ├── Makefile
│   └── pthread_test.c                  # 用户线程测试文件
├── bootloader                          # BootLoader
│   ├── boot.c
│   ├── boot.h
│   ├── Makefile
│   └── start.S
├── kernel                              # 内核程序
│   ├── include
│   │   ├── common
│   │   │   ├── assert.h
│   │   │   ├── const.h
│   │   │   └── types.h
│   │   ├── common.h
│   │   ├── device
│   │   │   ├── disk.h
│   │   │   ├── serial.h
│   │   │   ├── timer.h
│   │   │   └── vga.h
│   │   ├── device.h
│   │   ├── x86
│   │   │   ├── cpu.h
│   │   │   ├── io.h
│   │   │   ├── irq.h
│   │   │   └── memory.h
│   │   └── x86.h
│   ├── kernel
│   │   ├── disk.c                      # 硬盘外设
│   │   ├── doIrq.S                     # 中断处理程序接口
│   │   ├── i8259.c                     # 8259中断控制器
│   │   ├── idt.c                       # 中断描述符表
│   │   ├── irqHandle.c                 # 具体中断服务例程与系统调用
```

```
|   |   ├── kvm.c                       # 内核初始化程序
|   |   ├── serial.c                    # 串口外设
|   |   ├── timer.c                     # 时钟外设
|   |   └── vga.c                       # 显示外设
|   ├── lib
|   |   └── abort.c
|   ├── main.c                          # 内核主程序
|   └── Makefile
├── lib
|   ├── lib.h
|   ├── pthread.c                       # 用户级线程库
|   ├── pthread.h
|   ├── syscall.c                       # 系统调用
|   └── types.h
├── Makefile
└── utils                               # 镜像文件os.img生成工具
    ├── genBoot.pl
    └── genKernel.pl
```

5. 解决思路

5.1 时钟中断处理程序

本实验中时钟中断号为 `0x20`，可通过时钟自身触发中断引脚和自陷指令 `int $0x20` 进入时钟中断处理程序。如在文件 `kvm.c` 中，`kernel` 在完成了所有初始化工作后，就通过一句自陷指令：

```
asm volatile("int $0x20"); // trigger irqTimer
```

主动引发时钟中断，从而将处理器资源让渡给用户进程。

在中断处理程序中，我们通过 `tss` 切换到内核栈，并保存当前进程的现场，根据中断号，我们选择调用时钟中断的处理程序。在时钟中断的处理程序中，我们需要进行如下操作：

1. 对进程控制块表 `TCB` 进行遍历，将处于阻塞状态的进程（即调用 `sleep` 系统调用阻塞自身的进程）的 `sleepTime` 字段数值减少 `1`，如果减少后，该进程的 `sleepTime` 已经变为 `0` 了，表示该进程已经可以重新投入运行了，将其状态设置为可运行的（`RUNNABLE`）。

   ```
   int i = (current+1) % MAX_PCB_NUM;
   for (; i != current; i = (i + 1) % MAX_PCB_NUM)
   {
       if (pcb[i].state == STATE_BLOCKED)
       {
           if (pcb[i].sleepTime > 0)
               pcb[i].sleepTime--;
           if (pcb[i].sleepTime == 0)
               pcb[i].state = STATE_RUNNABLE;
       }
   }
   ```

2. 考虑当前进程的已运行时间字段`timeCount`，如果该运行时间没有达到OS分配的时间片（本实验中设置为`16`），就说明该进程还可以继续投入运行。于是我们从时钟中断处理函数返回，恢复该进程的现场，调用`iret`指令返回到用户态继续执行。

```
if (pcb[current].timeCount < MAX_TIME_COUNT)
{
    // continue
    pcb[current].timeCount++;
    return;
}
```

3. 如果字段`timeCount`的数值已经超过了OS分配的时间片，说明OS需要调度下一个进程投入运作了，我们从该进程的下一个进程开始，寻找一个状态为`RUNNABLE`的进程：

```
for (i = (current + 1) % MAX_PCB_NUM; i != current; i = (i + 1) % MAX_PCB_NUM)
{
    if (i == 0)
        continue;
    if (pcb[i].state == STATE_RUNNABLE)
        break;
}
pcb[i].state = STATE_RUNNING;
```

在遍历时我们需要注意，除非确实没有其他可调用的用户进程，不然我们不应该调用内核进程执行，因为内核进程本身不进行任何工作，只会降低CPU的利用率。当然，如果遍历一次其他进程没有发现可执行的进程，那么也就有`i=current`，将会继续运行原来的进程。

选中进程`i`后，我们就将当前运行的进程改成`i`，并内联汇编`mov`指令使得寄存器`esp`切换到进程`i`对应的内核栈顶。另外，我们还需要设置好`TSS`中的`esp0`和`ss0`字段（实际上，由于本实验中各个进程的内核栈对应的段是一致的，因此不修改`ss0`字段不会对实验产生实际影响）。

```
/* XXX recover stackTop of selected process */
current = i;
tmpStackTop = pcb[current].stackTop;
asm volatile("movl %0, %%esp"::"m"(tmpStackTop));
pcb[current].stackTop = pcb[current].prevStackTop;
// setting tss for user process
tss.esp0 = pcb[i].stackTop;
tss.ss0 = KSEL(SEG_KDATA);
```

最后，我们采取类似文件`doIrq.s`中的中断返回程序代码，直接通过内联汇编让该进程退出内核态：

```
// switch kernel stack
asm volatile("popl %gs");
asm volatile("popl %fs");
asm volatile("popl %es");
asm volatile("popl %ds");
asm volatile("popal");
asm volatile("addl $8, %esp");
asm volatile("iret");
```

通过上述的步骤，我们实现了一次时钟中断的全过程，现在程序已经可以在内核进程和用户进程1（即`pid=1`）的用户进程中来回切换。

5.2 进程相关系统调用实现

5.2.1 `fork`系统调用实现

`fork`系统调用用于创建一个子进程，其拥有自己的独立内存空间，并拷贝来自父进程的内存内容执行。内核为子进程提供一个独立的进程控制块，并完成对子进程进程控制块的设置。

其代码实现思路大致如下：

1. 首先，我们仍需要通过对进程控制块的一次迭代，检查该计算机系统的进程分配情况是否达到了上限，如果已经达到上限，就不能`fork`新的进程了，我们需要对父进程直接`return -1`。
2. 如果在进程控制块中找到一个可以分配的空位（即找到一项状态为`DEAD`的控制块），我们就可以将其作为新的进程。
    - 我们首先将父进程用户内存空间的所有内容复制到子进程内存空间中，由于这个过程可能比较长（我们需要对`0x100000`字节的内容进行拷贝），因此我们在该过程中打开中断，允许这个过程被其他中断嵌套：

      ```
      enableInterrupt();
      for (j = 0; j < 0x100000; j++) {
          *(uint8_t *)(j + (i+1)*0x100000) = *(uint8_t *)(j + (current+1)*0x100000);
      }
      disableInterrupt();
      ```

    - 将用户内存空间内容全部拷贝完成后，我们开始设置进程控制块中的信息，包括`pid`、`sleepTime`、`timeCount`，设置其状态为`RUNNABLE`，将其段寄存器设置为对应的段表索引（用户进程`1`对应段表`3`和`4`，进程`2`对应段表`5`和`6`，进程`3`对应段表`7`和`8`），并拷贝父进程的通用寄存器和`EFLAGS`寄存器。

      ```
      pcb[i].pid = i;
      pcb[i].sleepTime = pcb[current].sleepTime;
      pcb[i].prevStackTop = (uint32_t)&(pcb[i].stackTop);
      pcb[i].stackTop = (uint32_t)&(pcb[i].regs);
      pcb[i].state = STATE_RUNNABLE;
      /*XXX set regs */
      pcb[i].regs.cs = USEL(2*i + 1);
      pcb[i].regs.ds = USEL(2*i + 2);
      pcb[i].regs.es = USEL(2*i + 2);
      pcb[i].regs.fs = USEL(2*i + 2);
      pcb[i].regs.ss = USEL(2*i + 2);
      pcb[i].regs.eflags = pcb[current].regs.eflags;
      pcb[i].regs.edx = pcb[current].regs.edx;
      pcb[i].regs.ecx = pcb[current].regs.ecx;
      pcb[i].regs.ebx = pcb[current].regs.ebx;
      pcb[i].regs.esp = pcb[current].regs.esp;
      pcb[i].regs.ebp = pcb[current].regs.ebp;
      pcb[i].regs.edi = pcb[current].regs.edi;
      ```

```
pcb[i].regs.esi = pcb[current].regs.esi;
pcb[i].regs.eip = pcb[current].regs.eip;
```

- 最后我们设置返回值，成功情况下对父进程返回子进程的`pid`，对子进程返回`0`。

```
pcb[i].regs.eax = 0;
pcb[current].regs.eax = i;
```

由此我们即完成了系统调用`fork`的流程。中断处理结束后，系统将返回父进程继续执行，直到父进程的时间片用尽，内核再调度其他进程进行执行。

5.2.2 `sleep`系统调用实现

`sleep`系统调用用于一个进程主动向内核提出请求，在一定时间内阻塞自己，内核记录调用`sleep`的进程的阻塞时间，每经过一个时钟中断，内核将该时间减少`1`，减少到`0`时该进程重新恢复可运行状态。`sleep`系统调用内要执行的操作包括：

1. 获取用户进程调用`sleep`的参数，由`syscall`函数我们可知该参数存储在寄存器`ecx`中，我们由此在进程控制块中设置好`sleepTime`字段，并修改该进程状态进入阻塞态。

```
pcb[current].sleepTime = sf->ecx;
pcb[current].state = STATE_BLOCKED;
```

2. 接下来，由于当前进程已经陷入阻塞，我们需要选择一个新的进程投入运作，进程调度的算法已经在时钟中断处理程序中描述过，在此不多做赘述，我们选择好下一个执行的进程后，恢复这个进程的现场，从而将其投入运作。

由此我们完成`sleep`系统调用的流程。`sleep`系统调用的返回值没有强制规定，在此没有实现。

5.2.3 `exit`系统调用实现

`exit`系统调用用于终止一个进程的执行，内核回收该进程的进程控制块和内存空间。实际上，对内核来说，完成这个操作只需要一步操作，即设置该进程在进程控制块中的状态为`DEAD`：

```
pcb[current].state = STATE_DEAD;
```

随后我们需要调度下一个进程投入运行，该方法同前文描述的完全一致，在此省略。

`exit`的返回值在此没有处理，因为这个进程实际上已经结束工作了。

5.3 用户级线程库相关函数实现

用户级线程库的实现基于一个线程控制块的结构`TCB`，其大体结构类似于进程控制块，但完全处于用户级，因此不会有中断时间的发生，各个线程之间通过调用函数`pthread_yield`、`pthread_join`等等来主动让渡出处理器资源。线程控制块的结构如下：

```
struct ThreadTable {
    uint32_t stack[MAX_STACK_SIZE];
    struct Context cont;
    uint32_t retPoint;              // the entry to exit the thread
    uint32_t pthArg;                // the arg to pass into thread
    uint32_t stackTop;
    int state;
    uint32_t pthid;
    uint32_t joinid;
};
```

其中`stack`即为当前线程使用的栈空间，主线程除外，因为主线程使用的栈空间是创建进程时分配的栈空间。

主线程通过函数`pthread_initial`引入线程机制，激活线程控制块：

```
void pthread_initial(void){
    int i;
    for (i = 0; i < MAX_TCB_NUM; i++) {
        tcb[i].state = STATE_DEAD;
        tcb[i].joinid = -1;
    }
    tcb[0].state = STATE_RUNNING;
    tcb[0].pthid = 0;
    current = 0; // main thread
    return;
}
```

`pthread_initial`函数中，将所有其他线程状态设置为`DEAD`，并让全局变量`current`指向主线程`0`，表示当前正在执行主线程，然后返回。

5.3.1 `pthread_create`函数实现

`pthread_create`函数的原型如下：

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *
(*start_routine)(void *), void *arg)
```

其中`pthread_attr_t`的线程属性`attr`在本实现中没有讨论，创建的均为默认线程，参数`start_routine`为创建的线程所调用的函数，其接受一个`void *`类型的参数，返回值为`void *`类型，第四个参数`arg`即为函数接收的参数。

以下是`pthread_create`函数的执行流程：

1. 遍历线程控制块，找到一个可以分配的线程号，如果不能再分配新的线程，则直接返回`-1`。
2. 如果找到一个可以分配的线程号，让第一个参数`thread`指针所指向内存的值赋值为这个线程号。将原线程的状态设置为`RUNNABLE`，准备切换到新线程执行。

```
tcb[current].state = STATE_RUNNABLE;
*thread = i;
```

3. 我们需要先保存原线程的现场信息，我们直接内联汇编两条指令`pusha`和`mov`，分别用于保存通用寄存器信息和栈顶`esp`寄存器信息：

```
asm volatile("pushal");
asm volatile("movl %%esp, %0": "=m"(tcb[current].stackTop)); // store current esp
```

4. 随后我们对新线程的线程控制块进行处理，设置好线程`id`，状态，参数，返回值指向`pthread_exit`，这样如果函数中没有显式调用`pthread_exit`也可以使得该线程正常退出。考虑到程序栈中正常调用一个函数时，栈帧情况应该是这样的：

```
+------------------------------+
|            函数参数           |  (stack[MAX-1])
+------------------------------+ <- %ebp + 8
|            返回地址           |  (stack[MAX-2])
+------------------------------+ <- %ebp + 4
|            %ebp的旧值         |
+------------------------------+ <- %ebp
|                              |
|            函数栈帧           |
|                              |
+------------------------------+ <- %esp
```

由于程序进入到一个新的函数时，其执行的前两句指令为`push %ebp`和`mov %esp, %ebp`，因此在准备进入函数之时，其`esp`寄存器的位置应当是指向返回地址的，即我们应该令`esp`的位置处于`stack的顶部地址 - 8`，这就是栈顶的位置，随后我们将正确的函数参数和返回地址设置到栈中，并通过`jmp`汇编指令跳入调用函数的地址，即完成了`create`函数中的线程切换：

```
current = i; // change current thread to i
tcb[i].pthid = i;
tcb[i].state = STATE_RUNNING;
tcb[i].pthArg = (uint32_t)arg; // store function arg
tcb[i].retPoint = (uint32_t)(void *)pthread_exit; // return to pthread_exit
tcb[i].cont.eip = (uint32_t)(void *)start_routine;

// use stack[MAX_STACK_SIZE - 1] as the stack top
tcb[i].stack[MAX_STACK_SIZE - 1] = tcb[i].pthArg;
// return address stores at the bottom of pthread Argument
tcb[i].stack[MAX_STACK_SIZE - 2] = tcb[i].retPoint;

// now the esp address should be &cont -8
tcb[i].stackTop = (uint32_t)&tcb[i].cont - 8;
asm volatile("movl %0, %%esp"::"m"(tcb[i].stackTop));
asm volatile("jmpl *%0"::"m"(tcb[i].cont.eip));
```

5. 关于返回值，成功情况下该函数需要返回 `0`，但由于我们在该函数中已经实现了线程的切换，因此在该函数最后添加 `return` 已经没有意义了（因为 `jmp` 汇编指令已经使得程序脱离了这个函数，`return` 语句不会得到执行）。我们需要在其他函数中考虑这个返回值的处理。

由此我们实现了 `pthread_create` 函数，可以正常的创建一个新的线程使其执行一个指定的函数，并切换到这个创建的新线程进行执行。接下来我们讨论一下如何进行不同线程的切换，以便描述接下来几个函数的实现。

5.3.2 线程切换的流程

想要实现线程的切换，我们需要保留每个未执行完成但又暂时不处于执行状态下的线程的上下文，其至少需要包括以下信息：

- 所有通用寄存器的信息，其中栈顶 `esp` 的信息我们额外用字段 `stackTop` 进行处理。
- 程序计数器（`eip`）的信息。

其中通用寄存器的信息保存可以通过 `pusha` 指令完成，而栈顶信息可以通过 `mov` 指令完成，关键的是对于 `eip` 信息的保存，我们不应该直接保存当前位置的 `eip`，这是由于我们在函数中做了线程切换，如果只保存当前的 `eip`，那么重新恢复现场时，线程会从该指令继续往下做，即继续执行切换线程的操作，这不是我们需要的，我们想要的是恢复现场后线程从对应的 `pthread` 函数中返回，并保证返回值的正确。好在对于所有的 `pthread` 函数，其执行成功的返回值都是 `0`，因此我们可以直接在恢复待执行的线程的现场后，通过汇编指令让线程从相应函数中返回：

```
leave
movl $0, %eax
ret
```

需要注意的是，只执行上述这几条指令不一定能保证程序的正确性，这是因为在调用函数时，根据 `x86` 系统规范，我们将通用寄存器划分为两类：调用者保护寄存器 `eax`、`edx` 和 `ecx` 以及被调用者保护寄存器 `ebx`、`esi` 和 `edi`，对于后者而言，是需要调用的函数在调用开始时进行入栈保护，并在返回时出栈恢复状态的。由于本实验中的线程控制块 `tcb` 是一个复杂的数据结构，对该数据结构的寻址过程中极有可能**需要用到包含被调用者保护寄存器在内的大量通用寄存器**（当然，具体使用哪些寄存器是由编译器 `gcc` 决定的），因此 `gcc` 在编译代码时会对其自动加入保护过程（即函数运行前的入栈保护和返回前的出栈恢复工作），而我们直接通过上述的三句汇编指令返回，就失去了这样的出栈恢复。修改方法是，我们直接使用C语言编译语句 `return 0;`，`gcc` 会自动帮我们将对应的语句编译进去。

考虑这种方法，我们就不需要保存对应的 `eip` 了，因为返回地址本身就在每个线程控制块中的栈被保存了。但需要注意的是，采用这样的方法，由于一个线程返回时所处在的函数结构并非一定是调用时的函数，而对于不同函数，`gcc` 运用的被调用者保护寄存器个数可能不同，这可能导致了一些寄存器没有得到正常恢复，从而产生各种异常。因此，我们**手动添加了一些无关的复杂寻址指令**，目的是使得 `gcc` 编译该函数时使用到所有的3个被调用者保护寄存器，从而保证每个 `pthread` 函数的行为一致。

当然，我们还有一些其他的方法，下面是一个在实验过程中所想到的实现方法：

我们可以通过以下方法保存当前的 `eip`：通过指令 `call` 调用一个函数，这样在调用的函数中，`%ebp+4` 对应的就是 `call` 指令的下一条指令的地址值，我们使用内联汇编取出这个指令地址，借助一个辅助寄存器，把它存在一个变量中：

```
asm volatile("movl 0x4(%ebp), %eax");
asm volatile("movl %%eax, %0": "=m"(eip_tmp));
```

这样我们在变量 `eip_tmp` 中存储的就是所需的 `eip` 地址，我们再将其保存到当前线程的线程控制块中，作为现场上下文信息的一部分。恢复现场时，我们通过 `jmp` 指令跳转到这个地址继续执行：

```
asm volatile("jmp *%0":: "m"(tcb[i].cont.eip));
```

但需要注意的是，如果不加以限制，跳转后程序做的仍然是切换线程的工作，而不会从当前函数中返回，因此我们需要通过一个 `if` 控制语句，先在执行线程切换前用临时变量存储一下当前线程号 `current`，判断此时全局变量 `current` 的值与此前存储的是否一致，如果一致说明此时处于切换完成状态，可以直接返回；如果不一致，表明 `current` 被切换到即将运行的线程。

```
// 线程切换前
int tmpThread = current;

/* 线程切换代码 */
// 保存旧线程现场

if (current == tmpThread)
{
    return 0;
}

// 切换到新线程
```

通过这种方法也可以实现线程的切换，`gcc` 会在编译时将 `return 0;` 语句编译成所需的汇编代码。这种方法有一些好处，如在 `if` 语句中，我们可以进行更多的处理操作，但就本次实验而言，我们的需求只需进行 `return 0` 即可。

实际上交的代码中我们使用了第一种的线程切换机制，原因是因为其实现方式更为简单，而对于我们的实验来说，由于同时涉及到了C语言代码、汇编代码和 `gcc` 的汇编过程，采用简单的代码更易于保证不出现 `bug`，也容易保证一些外部因素（如 `gcc` 版本等）对实验的影响。由于各个 `pthread` 库函数的实现没有更为详细的需求，因此我们完全可以采取简单的线程切换机制来完成实验。

5.3.3 `pthread_join` 函数实现

`pthread_join` 函数用于一个线程阻塞自己，直到所指定的线程号对应的线程运行结束为止，再重新恢复可运行状态，其原型为：

```
int pthread_join(pthread_t thread, void **retval);
```

参数 `retval` 在本实验的测试用例中没有涉及到。

在本实验中，如果该函数所指定的线程已经处于结束状态（`DEAD`），该函数会直接返回 `-1` 值。否则，我们将该指定线程的线程控制块中的 `joinid` 字段设置成调用线程号，将当前线程状态阻塞（`BLOCKED`），然后调度下一个线程运行：

```
if (tcb[thread].state == STATE_DEAD)
{
    return -1;
}
tcb[current].state = STATE_BLOCKED;
int join = thread;
tcb[join].joinid = current;
// 调度下一个线程执行
```

其中线程的切换与调度在上面已经提到，在此不做赘述。

5.3.4 `pthread_yield` 函数实现

`pthread_yield` 用于一个线程主动让渡出处理器资源，供其他线程运作，其原型为：

```
int pthread_yield(void);
```

我们遍历线程控制块，如果找不出其他可执行的线程，就返回 `-1`，否则选择一个可运行的线程进行切换。

```
int i;
for (i = (current + 1) % MAX_TCB_NUM; ; i = (i + 1)%MAX_TCB_NUM)
{
    if (tcb[i].state == STATE_RUNNABLE)
        break;
}
if (i == current)
{
    return -1;
}
tcb[current].state = STATE_RUNNABLE;
// 切换到线程i
asm volatile("pushal");
asm volatile("movl %%esp, %0": "=m"(tcb[current].stackTop));

current = i;
tcb[i].state = STATE_RUNNING;

asm volatile("movl %0, %%esp":: "m"(tcb[i].stackTop));
asm volatile("popal");
return 0;
```

切换部分同上文所述，包含保存原线程现场和恢复待执行线程现场的步骤。另外，由于上文所述原因，函数中还有一些无关代码，在此没有列出。

5.3.5 `pthread_exit` 函数实现

`pthread_exit` 函数用于一个线程主动退出，其原型为：

```
void pthread_exit(void *retval);
```

其中参数 `retval` 对应于返回值的指针，在本实验中没有进行实现。

同进程系统调用函数 `exit` 一样，对于原线程而言，我们只需要将其状态设置为 `DEAD` 即可。但对于线程库来说，由于 `pthread_join` 函数的存在，我们还需要额外检查原线程的 `joinid`，观察是否有线程 `join` 到了这个线程上，如果有的话，让 `joinid` 指向的那个线程状态更改为可运行的（`RUNNABLE`）：

```
tcb[current].state = STATE_DEAD;
int join = tcb[current].joinid;
tcb[join].state = STATE_RUNNABLE;
```

随后我们选择下一个线程，调度其投入运行即可。

需要注意的是，`pthread_exit` 函数的返回类型是 `void`，为了保证其与其他线程函数的同步性，我们在恢复现场并返回前额外通过内联汇编手动将 `eax` 寄存器赋值为 `0`：

```
asm volatile("movl $0, %eax");
```

由此我们实现了所有线程相关函数，执行线程测试用例，就可以看到正确的结果了。