



操作系统原理实验报告

实验九：线程模型

数据科学与计算机学院 17大数据与人工智能

17341015 陈鸿峰

一、实验目的

- 理解并实现线程模型
- 能够编写运行一些简单的多线程程序

二、实验要求

1. 在内核中实现线程，并在C库中封装相关的系统调用。
2. 利用多线程技术，在进程中创建两个子线程，分别显示5个“Hello”和“world!”。

三、实验环境

具体环境选择原因已在实验一报告中说明。

- Windows 10系统 + Ubuntu 18.04(LTS)子系统
- gcc 7.3.0 + nasm 2.13.02 + GNU ld (Binutils) 2.3.0
- GNU Make 4.1
- Oracle VM VirtualBox 6.0.6
- Bochs 2.6.9
- Sublime Text 3 + Visual Studio Code 1.33.1

虚拟机配置：内存4M，1.44M虚拟软盘引导，1.44M虚拟硬盘。

四、实验方案

本次实验继续沿用实验六保护模式的操作系统。用户程序都以ELF文件格式读入，并且在用户态(ring 3)下执行。

本次实验借鉴了Linux操作系统对线程进程的管理，采用**进程和线程统一管理**的方法，即进程与线程都可以看作是一个**task**，在调度、执行、管理上面都不进行区分。

这一方面大大减少了冗余的代码量，另一方面也使进程和线程的管理更加简单。为了避免进程fork时开销过大的问题，Linux采用了当写时才复制(Copy On Write, COW)机制，即fork时不将进程所对应的全部进程空间进行复制，而是采用以下方案：

- 在GDT表项中将父进程和子进程的数据段都置为只读

- 当实际运行中遇到父进程或子进程对数据段进行修改时，触发异常，硬件中断，进入中断处理程序
- 这时才将进程空间进行拷贝

这将很大程度上降低进程fork的开销，也使进程线程的复制能够统一化管理。

由于Linux任务管理的这种特性，我们也将线程称为轻量级进程(lightweight process)。因此，我们不需要对原有的task.h头文件进行修改，而只需添加一些线程设施。

1. 线程创建

线程创建与进程的fork类似，但要注意下面的函数原型，多了三个参数传递，这是与进程fork非常不同的地方。

```
int do_thread_create(int* tid, uintptr_t func, void* args)
```

其中，

- tid: 返回线程ID
- func: 多线程函数调用入口
- args: 函数参数

具体的步骤如下

- 用proc_alloc分配一个新的任务
- 将当前进程主线程控制块的内容拷贝至子线程中
- 修改栈段指针ebp和user_esp，并拷贝主线程栈中所有内容
- 设置子线程的状态
- 设置函数入口eip=func，同时将函数参数args和返回地址user_pthread_return堆栈，注意这里user_pthread_return必须是用户态(ring 3)程序可以执行的，否则会报GPF错。实现方法即利用用户态中断int 0x81，重返内核。
- 最后设置tid的返回值

完整函数如下所示。

```
int do_thread_create(int* tid, uintptr_t func, void* args)
{
    disable();

    process* child;

    // find empty entry
    if ((child = proc_alloc()) == 0) {
        enable();
        return -1; // fail to create child process
    }
}
```

```

}

// copy PCB, which has been saved by interrupt
child->regImg.eip = curr_proc->regImg.eip;
child->regImg.cs = curr_proc->regImg.cs;
child->regImg.eflags = curr_proc->regImg.eflags;

child->regImg.eax = curr_proc->regImg.eax;
child->regImg.ecx = curr_proc->regImg.ecx;
child->regImg.edx = curr_proc->regImg.edx;
child->regImg.ebx = curr_proc->regImg.ebx;

child->regImg.esp = curr_proc->regImg.esp;
child->regImg.ebp = curr_proc->regImg.ebp + (child->pid * 0x100); // use
    ↪ different stack!
child->regImg.esi = curr_proc->regImg.esi;
child->regImg.edi = curr_proc->regImg.edi;

child->regImg.ds = curr_proc->regImg.ds;
child->regImg.es = curr_proc->regImg.es;
child->regImg.fs = curr_proc->regImg.fs;
child->regImg.gs = curr_proc->regImg.gs;

child->regImg.ss = curr_proc->regImg.ss;
child->regImg.user_esp = curr_proc->regImg.user_esp + (child->pid * 0x100);

// copy stack
memcpy((void*)(child->regImg.user_esp),
      (void*)(curr_proc->regImg.user_esp),
      (curr_proc->regImg.ebp - curr_proc->regImg.user_esp)*2); // each entry 32-
    ↪ bit

// set state
child->regImg.eax = 0;
child->parent = curr_proc;
child->status = PROC_READY;
reset_time(child);

// set function entrance
child->regImg.eip = func;
uintptr_t* stack = (uintptr_t*) child->regImg.user_esp;
stack--;
*stack = (uintptr_t) args; // pass arguments
stack--;
*stack = (uintptr_t) user_pthread_return; // return address

```

```

child->regImg.user_esp = (uintptr_t) stack;

// set return values
*tid = child->pid;

return 0;
}

```

2. 线程等待

此函数主要用在主线程，用来等待子线程完成。实现方法即将主线程设置为PROC_WAITING，然后进入任务调度阶段，继续执行子线程。

```

void do_thread_join(int tid, void** ret)
{
    disable();
    curr_proc->status = PROC_WAITING;
    schedule_proc();
    enable();
}

```

3. 线程退出

将当前线程状态设为终止，然后将其父进程唤醒。从这里也可以看出，一个exit函数对应着一个join函数，故有多少个线程，为了同步就应该有多少个join。这与Linux的pthread库实现方式是一样的。

```

void do_thread_exit()
{
    disable();
    curr_proc->status = PROC_TERMINATED;
    if (curr_proc->parent != NULL)
        wakeup(curr_proc->parent->pid);
    enable();
}

```

为方便用户函数调用，也提供了用户态的退出函数user_pthread_return

```

void user_pthread_return() {
    asm volatile (
        "int 0x81\n\t"
        :
        : "a" (3)
        );
}

```

此函数需要添加在每个线程函数的最后，即堆栈返回地址，否则将无法回到主线程继续执行。

4. 系统调用

为了不让单一系统调用太过复杂，本实验的pthread采用int 0x81号中断调用。通过下列函数对IDT进行设置，确保用户态程序可以调用。

```
setvect_user (0x81, (unsigned long) sys_pthread_handler);
```

其中sys_pthread_handler是一个汇编入口，将寄存器信息都保存后，进入sys_pthread_handler_main中断处理程序。先将ebx,ecx,edx寄存器的内容取出，其中含有用户传递的参数；然后依次进入线程管理程序进行处理。注意这里都是以uintptr_t的32位地址格式传递，进入函数后才进行类型转换。

```
extern void sys_pthread_handler ();
int sys_pthread_handler_main (int no) {
    uintptr_t arg1, arg2, arg3;
    asm volatile("":"=b"(arg1:));
    asm volatile("":"=c"(arg2:));
    asm volatile("":"=d"(arg3:));
    if (no == 0) {
        return do_thread_create((int*) arg1, arg2, (void*) arg3);
    } else if (no == 1) {
        do_thread_join((int) arg1, (void**) arg2);
    } else if (no == 2) {
        return do_thread_self();
    } else if (no == 3) {
        do_thread_exit();
    }
    return 0;
}
```

对这些函数进行封装，得到pthread.h头文件，API如下，基本与Linux的POSIX线程库相同：

- int pthread_create(int* tid, uintptr_t func, void* args): 创建线程
- void pthread_join(int tid, void** ret): 线程等待
- int pthread_self(): 获取线程ID
- void pthread_exit(): 线程退出

五、实验结果

1. 多线程Hello world测试

用户程序hello_world_thread.c代码如下。

```
#include "stdio.h"
#include "pthread.h"

int m = 5;

void hello(void* args){
    char* str = (char*) args;
    for(int i = 0; i < m; i++)
        printf("%s\n", str);
}

void main() {
    int tid1, tid2;
    pthread_create(&tid1, (uintptr_t)hello, "Hello");
    pthread_create(&tid2, (uintptr_t)hello, "world!");
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);
    return;
}
```

实验结果如图1所示。

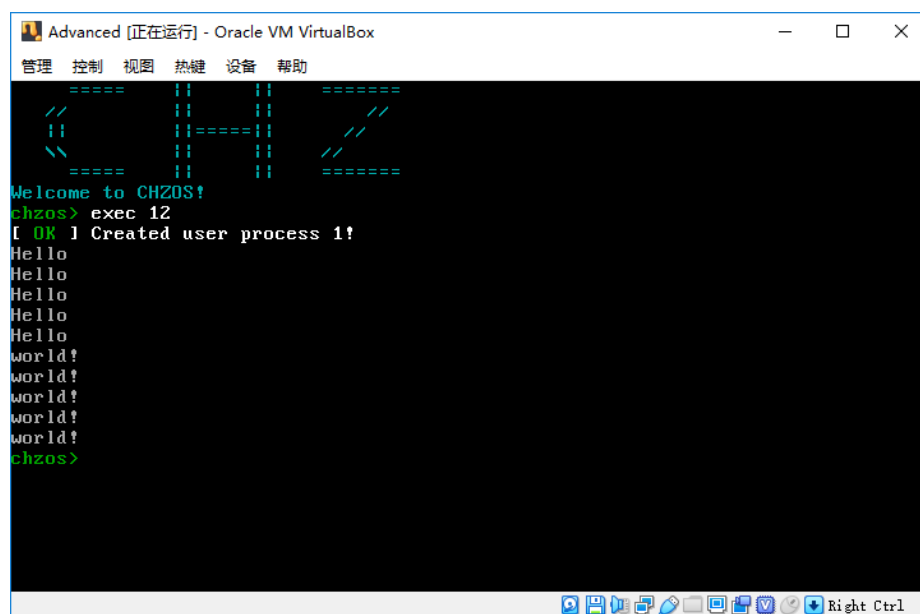


图 1: 多线程Hello world($m = 5$)

由于线程执行的速度非常快，故即使设置轮转时间片大小为1，线程也都执行完了。

为真正展现多线程执行的效果，我将 m 值调大，使其输出多个Hello和多个world!，以便观察多个线程的并发关系。

从图2中可以看出，两个线程确实是交替执行的，Hello和world!相互穿插，甚至有在输出换行期间中断造成线程调度的。从此可以看出多线程程序执行的不确定性，这确实是符合预期的。

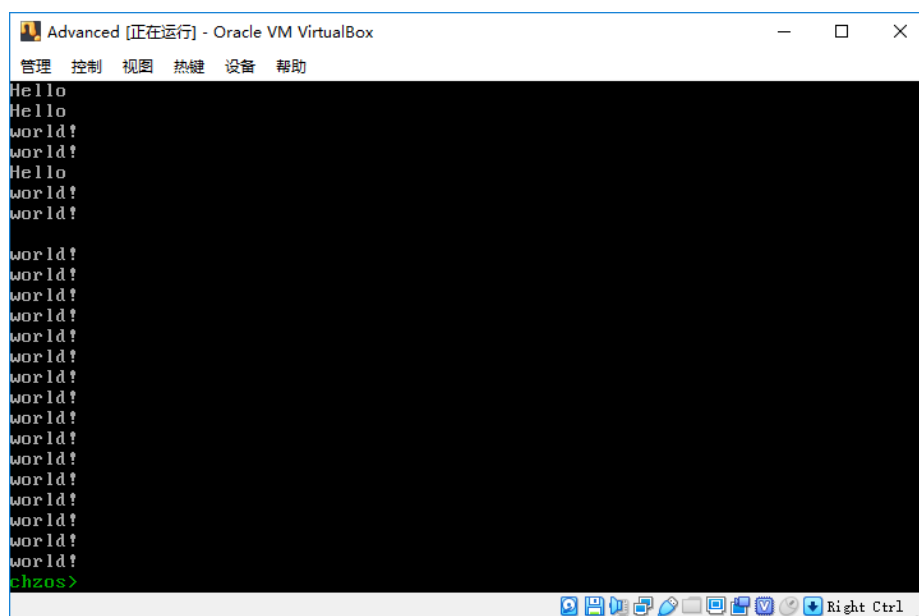


图 2: 多线程Hello world($m = 100$)

2. 多线程矩阵乘法测试

除了前面简单的Hello world测试，我也编写了稍微复杂一点的程序。如下所示是一个多线程的矩阵乘法，采用数据并行的方法进行划分计算。

```
#include "stdio.h"
#include "pthread.h"

// maximum size of matrix
#define MAX 6

// maximum number of threads
#define MAX_THREAD 3

int matA[MAX][MAX];
int matB[MAX][MAX];
int matC[MAX][MAX];
```

```
void multiply(void* arg)
{
    int tid = pthread_self();
    printf("This is thread:%d Arg:%d\n", tid, (int)arg);

    int core = (int) arg;
    // Each thread computes 1/n of the matrix multiplication
    for (int i = core * MAX / MAX_THREAD; i < (core + 1) * MAX / MAX_THREAD; i++)
        for (int j = 0; j < MAX; j++)
            for (int k = 0; k < MAX; k++){
                // printf("%d ", i);
                matC[i][j] += matA[i][k] * matB[k][j];
            }
}

int main()
{
    // Generating values in matA and matB
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++) {
            matA[i][j] = i + j;
            matB[i][j] = i * j;
            matC[i][j] = 0;
        }
    }

    // Displaying matA
    printf("Matrix A\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++)
            printf("%d ", matA[i][j]);
        printf("\n");
    }

    // Displaying matB
    printf("Matrix B\n");
    for (int i = 0; i < MAX; i++) {
        for (int j = 0; j < MAX; j++)
            printf("%d ", matB[i][j]);
        printf("\n");
    }

    // declaring four threads
    pthread_t threads[MAX_THREAD];
```



```

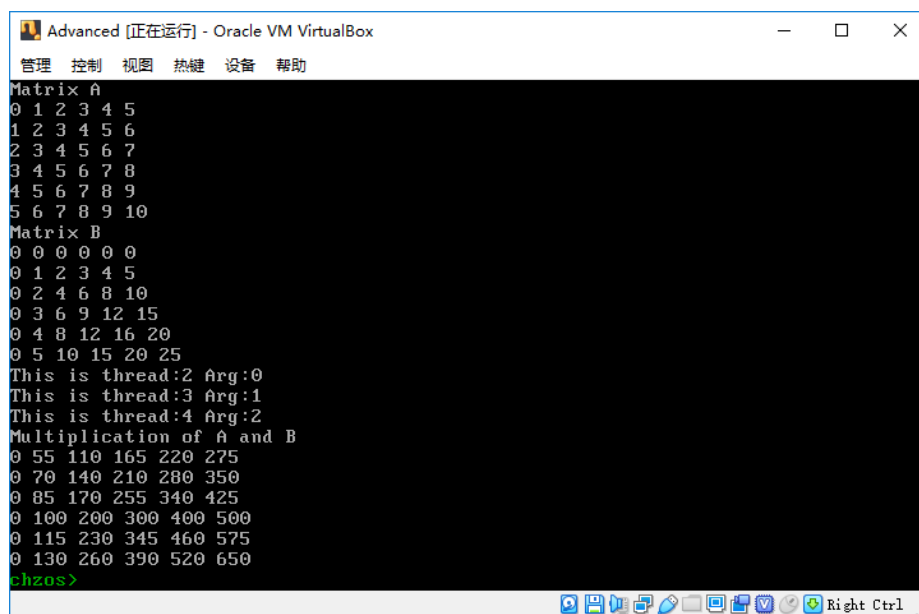
// Creating four threads, each evaluating its own part
for (int i = 0; i < MAX_THREAD; i++)
    pthread_create(&threads[i], (pthread_attr_t){0}, (void*)(i));

// joining and waiting for all threads to complete
for (int i = 0; i < MAX_THREAD-1; i++)
    pthread_join(threads[i], NULL);
pthread_join(threads[MAX_THREAD-1], NULL);

// Displaying the result matrix
printf("Multiplication of A and B\n");
for (int i = 0; i < MAX; i++) {
    for (int j = 0; j < MAX; j++)
        printf("%d ", matC[i][j]);
    printf("\n");
}
return 0;
}

```

从图3中可以看出，即使面向复杂的多线程程序，我的操作系统依然可以正常运行，且结果正确。



```

Advanced [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
Matrix A
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9
5 6 7 8 9 10
Matrix B
0 0 0 0 0 0
0 1 2 3 4 5
0 2 4 6 8 10
0 3 6 9 12 15
0 4 8 12 16 20
0 5 10 15 20 25
This is thread:2 Arg:0
This is thread:3 Arg:1
This is thread:4 Arg:2
Multiplication of A and B
0 55 110 165 220 275
0 70 140 210 280 350
0 85 170 255 340 425
0 100 200 300 400 500
0 115 230 345 460 575
0 130 260 390 520 650
chzos>

```

图 3: 多线程矩阵乘法

六、实验总结

本次实验主要在前期的线程模型方面纠结了较长时间，不知道线程控制块应该怎么设置。后来了解到Linux的进程线程管理模型，瞬间醍醐灌顶，不得不说Linux的实现真的非常妙。

传统教材中都将进程和线程分开来处理，而Linux却将这两者看成是等价的，既能实现更细粒度的管理，又能结合COW等机制将进程管理的开销降到最低。因此，基于这个理论，我操作系统的线程实现就非常迅速了。不需要对原有的进程模型进行修改，而只需添加对多线程的支持。

以前常想操作系统已经发展了几十年了，Windows也已到最后一代Win 10，这样操作系统还有什么可以做的呢。这其实是不对的，操作系统是一个充满活力的学科，课本上的理论都是上个世纪的东西，随着操作系统的不断发展，很多实现也与当初的理论大相径庭，但是却能实现性能、稳定性、安全性等的大幅提升。本次实验实现了类Linux的进程模型，相较课本上的进程线程模型更为紧凑而优美，这也算是我迈向现代操作系统的第一步吧。

由并行分布式课程会用POSIX编写一些多线程程序，再到操作系统能自己实现pthread库，看着多线程程序在我眼前飞舞，我能感受到从上到下一脉打通的快感，也真真切切感受到了现代计算机的无比魅力。

七、参考资料

1. OS Development Series, <http://www.brokenthorn.com/Resources/OSDevIndex.html>
2. Roll your own toy UNIX-clone OS, http://www.jamesmolloy.co.uk/tutorial_html/
3. The little book about OS development, <http://littleosbook.github.io/>
4. Writing a Simple Operating System from Scratch, http://www.cs.bham.ac.uk/~exr/lectures/opsys/10_11/lectures/os-dev.pdf
5. Intel® 64 and IA-32 Architectures Software Developer's Manual
6. UCore OS Lab, https://github.com/chyyuu/ucore_os_lab
7. CMU CS 15-410, Operating System Design and Implementation, <https://www.cs.cmu.edu/~410/>
8. 李忠，王晓波，余洁，《x86汇编语言-从实模式到保护模式》，电子工业出版社，2013

附录 A. 程序清单

1. 内核核心代码

序号	文件	描述
1	bootloader.asm	主引导程序
2	kernel_entry.asm	内核汇编入口程序
3	kernel.c	内核C入口程序
4	Makefile	自动编译指令文件
5	bootflpy.img	引导程序/内核软盘
6	mydisk.hdd	虚拟硬盘
7	bochsrc.bxrc	Bochs配置文件

2. 内核头文件

序号	文件	描述
1	disk_load.inc	BIOS读取磁盘
2	show.inc	常用汇编字符显示
3	gdt.inc	汇编全局描述符表
4	gdt.h	C全局描述符表
5	idt.h	中断描述符表
6	hal.h	硬件抽象层
6.1	pic.h	可编程中断控制器
6.2	pit.h	可编程区间计时器
6.3	keyboard.h	键盘处理
6.4	tss.h	任务状态段
6.5	ide.h	硬盘读取
7	io.h	I/O编程
8	exception.h	异常处理
9	syscall.h	系统调用
10	task.h	多进程设施
11	user.h	用户程序处理
12	terminal.h	Shell
13	scancode.h	扫描码
14	stdio.h	标准输入输出
15	string.h	字符串处理
16	elf.h	ELF文件处理
17	api.h	进程管理API
18	semaphore.h	信号量机制
19	systhread.h	线程模型
20	pthread.h	线程管理API

3. 用户程序

用户程序都放置在usr文件夹中。

序号	文件	描述
1-4	prgX.asm	飞翔字符用户程序
5	box.asm	画框用户程序
6	sys_test.asm	系统中断测试
7	fork_test.c	进程分支测试
8	fork2.c	进程多分支测试
9	bank.c	银行存取款测试
10	fruit.c	父子祝福水果测试
11	prod_cons.c	消费者生产者模型测试
12	hello_world_thread.c	多线程Hello_world测试
13	matmul.c	多线程矩阵乘法测试

附录 B. 系统调用清单

int 0x80功能号	功能
0	输出OS Logo
1	睡眠100ms
10	fork
11	wait
12	exit
13	get_pid
20	get_sem
21	sem_wait
22	sem_signal
23	free_sem
100	返回内核Shell

int 0x81功能号	功能
0	pthread_create
1	pthread_join
2	pthread_self
3	pthread_exit