中山大学

OS Lab 12

多线程

Author:
赖少凡

Instructor:
凌应标

2015/06/10

Contents

1 实验简述

本实验模仿 Linux 的多线程实现方式，利用（轻量级）进程与线程一一对应的方法，让操作系统支持多线程。用户的多线程库模仿目前流行的多线程库进行设计，利用 join 代替实验要求中的 exit 和 wait。此外，与原实验要求的设计不一样，我的设计中系统内核只提供 clone 的系统调用，其他线程管理由外部库完成。

2 实验目的

1. 学习多线程技术，掌握内核线程实现方法。
2. 修改和扩展内核，实现内核线程模型和调度。
3. 利用多线程技术，实现一个简单的多线程应用。

3 实验内容

1. 根据线程原理，实现多线程模型。并实现 thread_create, thread_join 这两个功能。
2. 实现用户的多线程库。提供 thread_create, thread_join, gettid 等功能。
3. 编写一个多线程应用的 C 程序，进程中创建 4 个线程分工合作完成一项任务。

4 整体设计

根据 [2]，Linux 使用一种叫做“轻量级进程”的进程与线程一一对应，通过系统内核进行多线程间的切换，而在外部库进行其他调度。模仿该原理，我设计了一种“简化”的进程，该种进程只有独立的栈段，而其他的段是共享其他进程的。一个简化进程对应着一个线程。

4.1 用户内存管理

为了从用户的堆栈空间中分出空间给新的线程做栈，我需要对进程的内存进行管理。我们所熟知的栈管理通过 esp、push、pop 等汇编指令就可以做到，而在此需要实现的是类似 malloc 的堆管理函数。在一进程初始化的时候，我会执行：

```
    heap_top = get_task_length(); // 计算堆顶
    heap_ptr = heap_top - get_stack_size(); // 计算堆起始位置

```

对堆进行初始化管理，而 malloc 的代码为：

```
void * malloc(uint32_t size) {
    if (heap_ptr == 0) { // 如果堆未初始化，报错
        printf(">> Error: The heap must be initialized before using!\n");
        return 0;
    }
    register int esp asm("esp"); // 获得当前的栈指针
    if (heap_ptr + size > esp) { // 栈指针与堆指针相矛盾（内存用尽），报错
        printf(">> Error: The free space was all used!\n");
        return 0;
```

```
    }

    void * tmp = (void *)heap_ptr; // 当前堆指针
    heap_ptr += size; // 堆指针移动
    return tmp;
}

```

4.2 clone——通过共享创建一个轻量级进程

```
int clone_for_syscall(exception_status_t * t) {
    // t->ebx is eip, t->ecx is esp, t->edx
    // 传入的三个参数分别为线程的eip, esp, id
    task_t * new_task = create_empty_task(cur_task->stack_size, 0); // 创建一个空的进程

    // task
    new_task->length = cur_task->length; // 复制进程信息
    new_task->base = cur_task->base;
    new_task->estatus.ds = new_task->estatus.ds = new_task->estatus.es = new_task->estatus.fs
    = new_task->estatus.gs = new_task->estatus.ss = cur_task->estatus.ds; // 共享数据段
    new_task->estatus.cs = cur_task->estatus.cs; // 共享代码段
    new_task->estatus.eflags = 202;

    new_task->estatus.eip = t->ebx; // 线程入口
    new_task->estatus.ebp = t->ecx; // 栈指针的独立
    new_task->estatus.esp = t->ecx; // 栈指针的独立

    // thread
    new_task->is_thread = true; // 标志为专属于线程的轻量级进程
    new_task->tid = t->edx; // 线程编号
}

```

4.3 thread_create——创建线程

```
int thread_create(void * begin, int stack_size) {
    void * stack = malloc(stack_size); // 在用户空间中申请一个栈
    *((uint32_t*)(stack+stack_size)-1) = (uint32_t)thread_end; // 将线程终止的地址先写入栈中，
    在线程执行完执行ret时就会跳到线程终止的地方
    ++tid_counter; // 增加线程计数器
    syscall_clone(begin, stack+stack_size-4, tid_counter); // 创建一个对应的轻量级进程，-4
    是为了模拟压入线程终止地址这个动作。
}

```

4.4 thread_end——线程终止

```
void thread_end() {
    tid_counter--; // 将计数器减一
    exit(0); // 结束对应的轻量级进程
}
```

4.5 thread_join——等待所有线程结束

```
int thread_join() {
    while (tid_counter) {} // 当线程计数器不为零时等待
}
```

5 验证实验

5.1 实验代码

与实验七相同，我写了一个代码来统计一段文字中的小写字母数字。程序会将字符串分割成 N（默认 N=4）份，然后交给 N 个进程分别统计，最后输出答案。假设每个线程要统计 25 个字符，1 号线程则负责统计 0 到 24 的字符，2 号线程负责统计 25 到 49 的字符，以此类推。

```
#include <program_header.h>
#include <string.h>
#include <stdio.h>
#include <thread.h>

#define THREAD_COUNT 4  // 线程个数
static int len_per_group; // 每个线程要处理的长度
static int count; // 计数器
static char *s = "12
    oi922kg3209kg20g39k203g9k2039gbj32kcb9c32n023c92n0cn30929c0i230c9k2r203fk09fk320932kf02939f0k209ka
    "; // len = 100

void calc() {
    int tid = gettid(); // 得到当前线程的编号
    printf("[Thread %d] I am a thread, my process-id is %d\n", tid, getpid()); //
    打印当前线程编号，以及其对应的轻量级进程编号

    int start = (tid-1) * len_per_group, end = tid * len_per_group; //
    计算该线程要从哪里统计到哪里
    printf("[Thread %d] I am going to calculate \"", tid); // 开始统计

    int i;
    for (i = start; i < end; ++i) {
        printf("%c", s[i]); // 边统计边输出自己统计的内容
        if ('a' <= s[i] && s[i] <= 'z')
            ++count; // 累计
    }
    printf("\"\n");
```

```
25 }
27 int main() {
       printf("Lab 12\n  current stack size: %x\n  current task length: %x\n", get_stack_size(),
       get_task_length()); // 输出用户空间的大小、程序的总长度
29     int len = strlen(s); // 得到字符串长度
       len_per_group = len / THREAD_COUNT;  // 计算一个线程要统计多少
31     count = 0; // 清空计数器

33     int i = 0; // 开始创建线程
       for (i = 0; i < THREAD_COUNT; ++i) // 循环创建线程
35         thread_create(calc, 0x500); // 0x500 是栈大小，calc是线程入口
       thread_join(); // 等待线程们统计完毕
37
       printf("[Main] The answer is %d\n", count); // 输出统计结果
39 }
```

5.2 实验结果

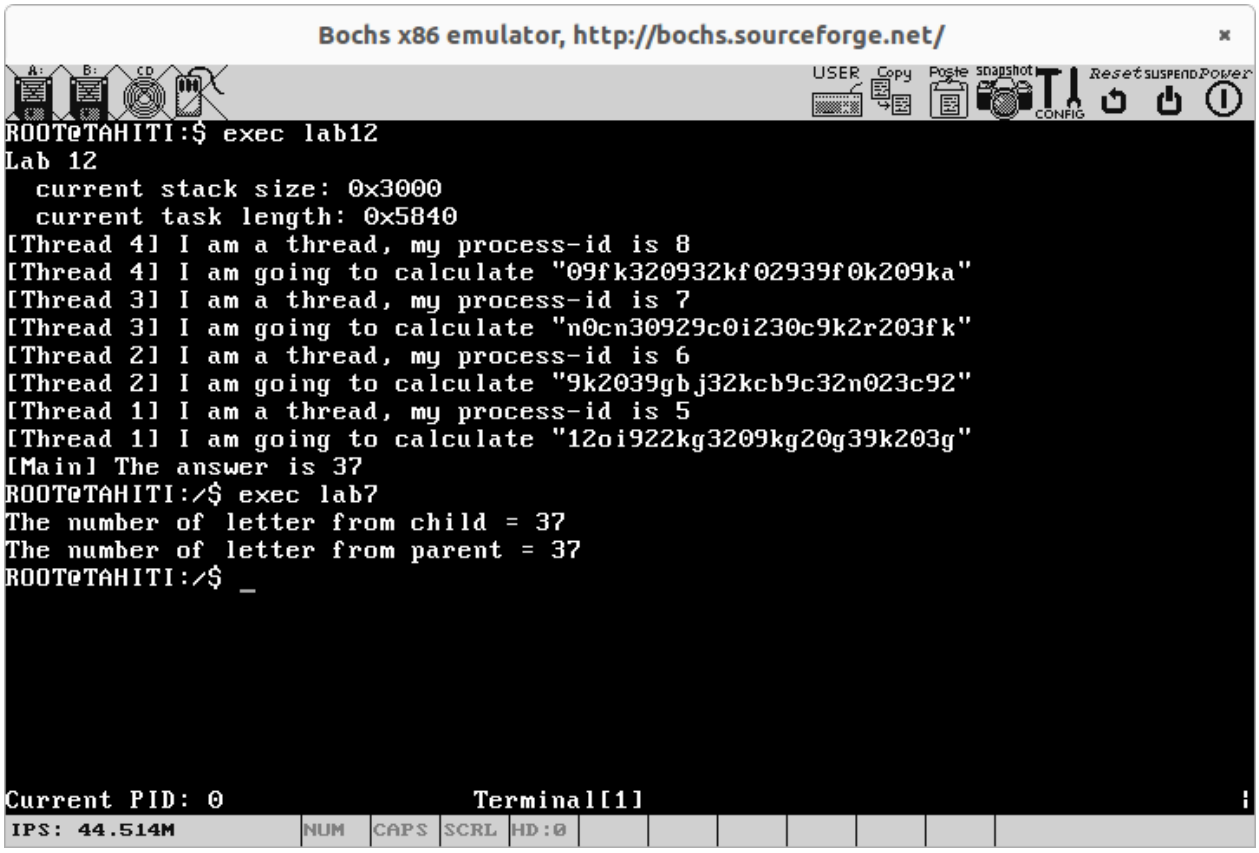

Figure 1: 执行 lab12 时，可见四个进程分别处理 25 个字符，最后输出 37；利用实验七的代码统计的结果也是 37

6 实验心得

本实验的难度在于理解线程在内核和用户层面的关系。在哪个层面进行切换、调度、生成、毁灭都是值得讨论的，并且在不同的操作系统中有不同的设计。理论上来说，用户线程应该对内核不可见，但是完全的用户级线程现仅仅存在于理论中。个人觉得原实验要求对线程的设计也不太合理，因为线程信息对系统内核是完全可见的。我模仿了 linux 早期的多线程想法，通过一个轻量级线程来联系线程与内核，实验结果是可行的。此外值得注意的就是如何对用户内核做有效的管理。

References

[1] http://www.ibm.com/developerworks/cn/linux/kernel/l-thread/ Linux 多线程模型的简析

[2] http://www.cnblogs.com/zhaoyl/p/3620204.html Linux 多线程模型的简析