中山大學
SUN YAT-SEN UNIVERSITY

本科生实验报告

实验课程:__________操作系统__________

实验名称:____________________

专业名称:__________网络空间安全__________

学生姓名:__________陈浚铭__________

学生学号:__________20337021__________

实验地点:__________温暖的家__________

实验成绩:____________________

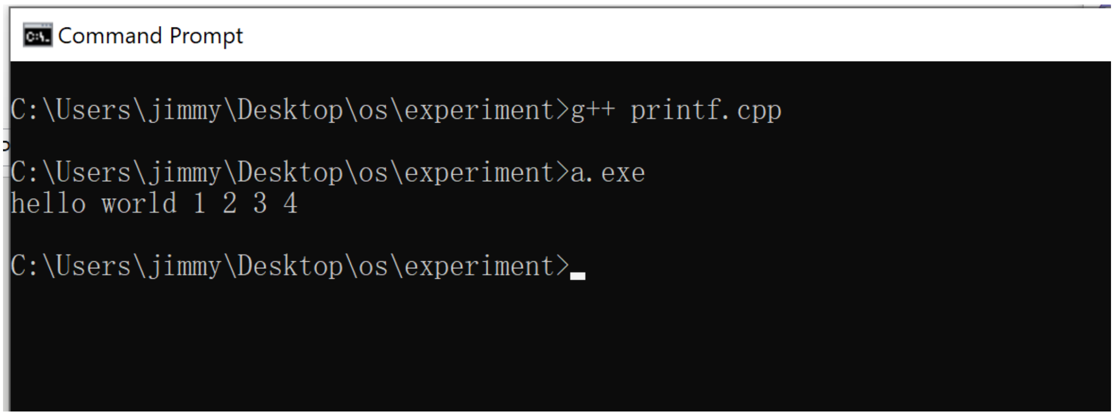
报告时间:____________________

1. 实验要求

深入理解进程在操作系统的作用和重要性

2. 实验过程和实验结果

assignment 1：

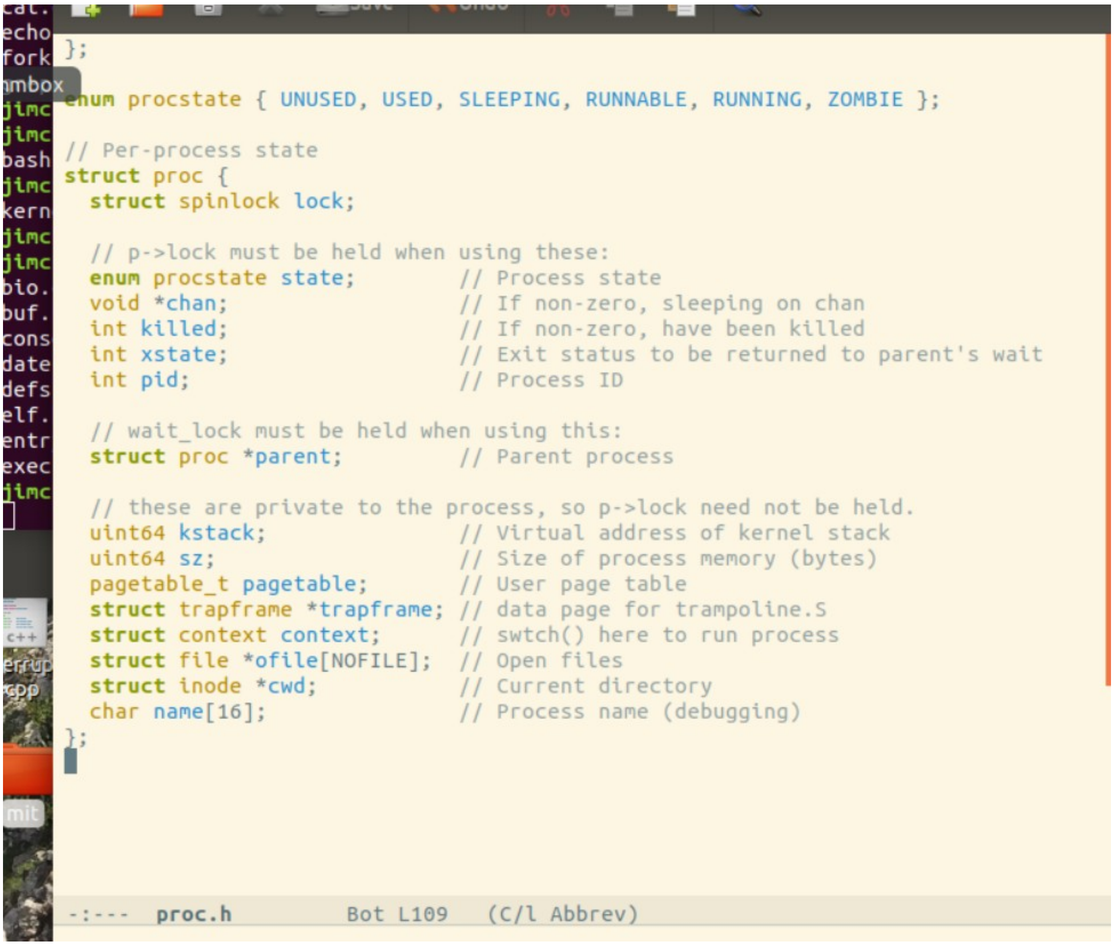


在这里我透过自己实现的 printf 函数打印出 hello world, 以及利用可变参数列表打印出 1 2 3 4.

assignment 2:

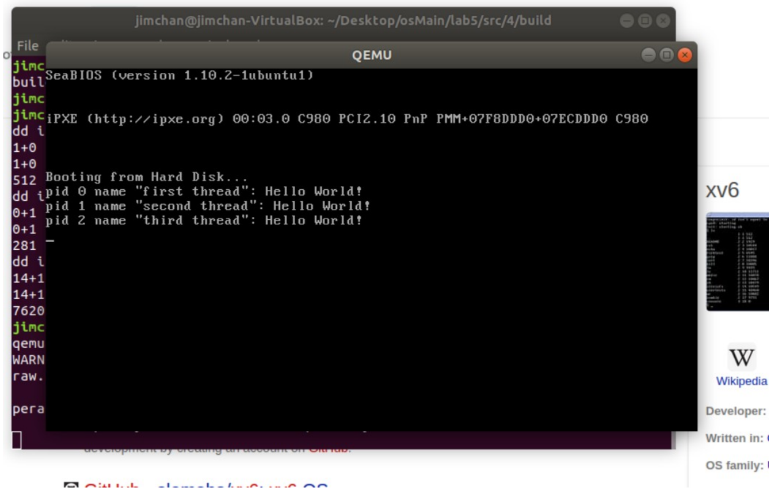
在 PCB 中，一个进程的重要信息为 priority. CPU 能够根据 priority 来调度线程。

由 MIT xv6 这个教学 OS 程序， 我们在 proc.h 可看到一个 PCB 的结构：



它包括了重要信息例如进程状态（process state)，优先度（priority)等等。

我们执行教程中的程序得到结果：



在我事先的PCB，我增加了burstTime，用来代表线程估计的执行时间。

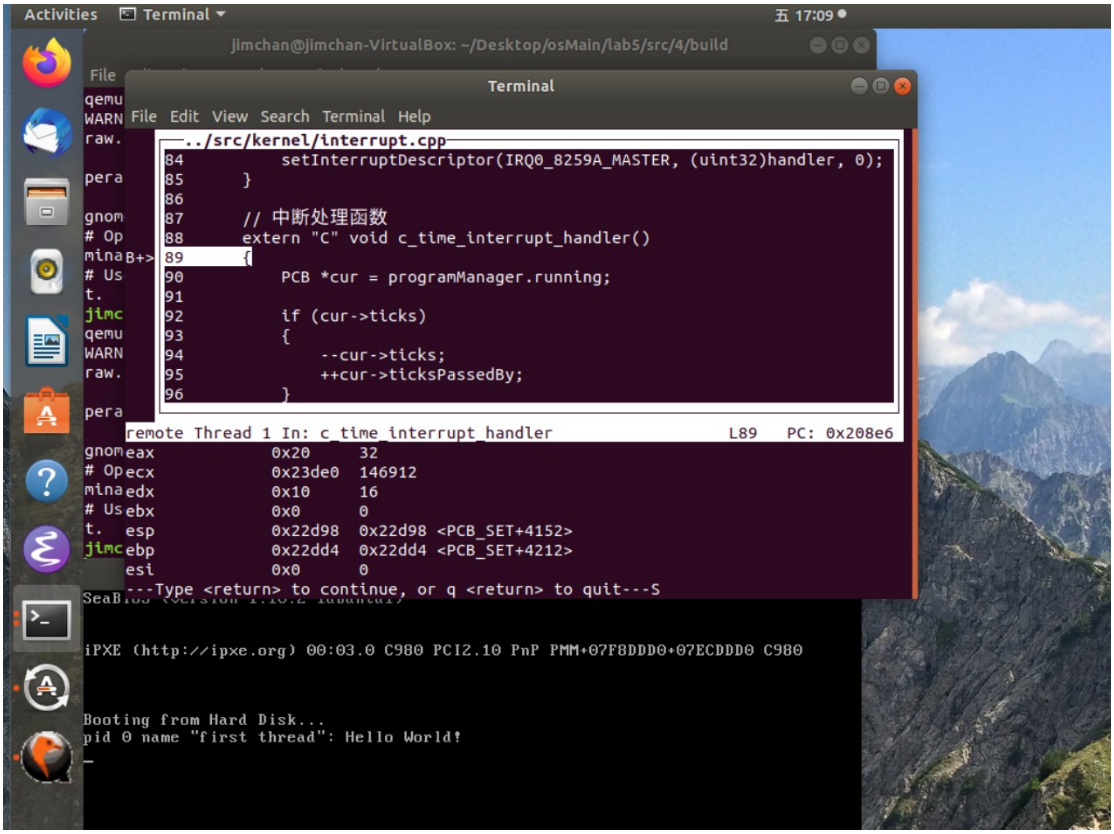
```
struct PCB
{
    int *stack;                      // 栈指针，用于调度时保存esp
    char name[MAX_PROGRAM_NAME + 1]; // 线程名
    enum ProgramStatus status;       // 线程的状态
    int priority;                    // 线程优先级
    int pid;                         // 线程pid
    int ticks;                       // 线程时间片总时间
    int ticksPassedBy; // 线程已执行时间
    int burstTime;
    ListItem tagInGeneralList;       // 线程队列标识
    ListItem tagInAllList;           // 线程队列标识
};
```

assignment 3:

我们执行提供了的代码，并实现了执行线程。

Assignment 3: GDB 调试

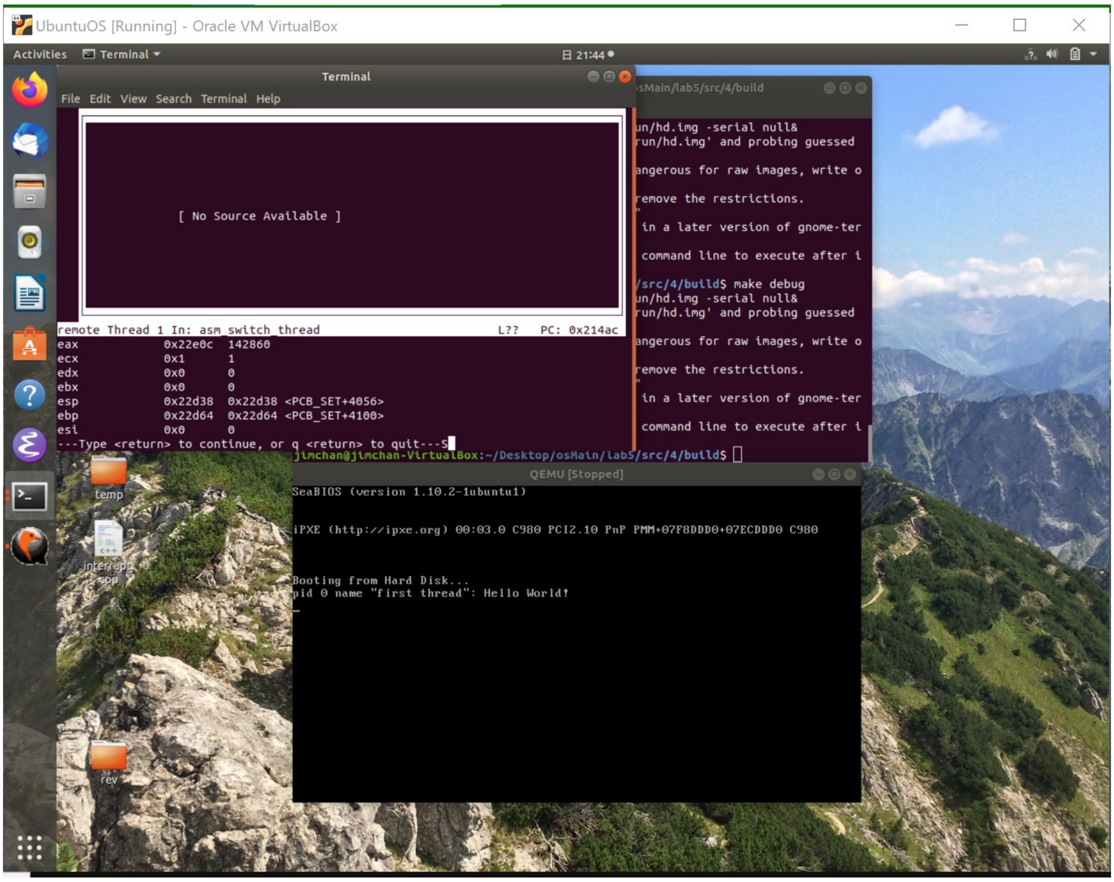
首先设定断点为c_time_interrupt_handler()，并在gdb看寄存器的值：



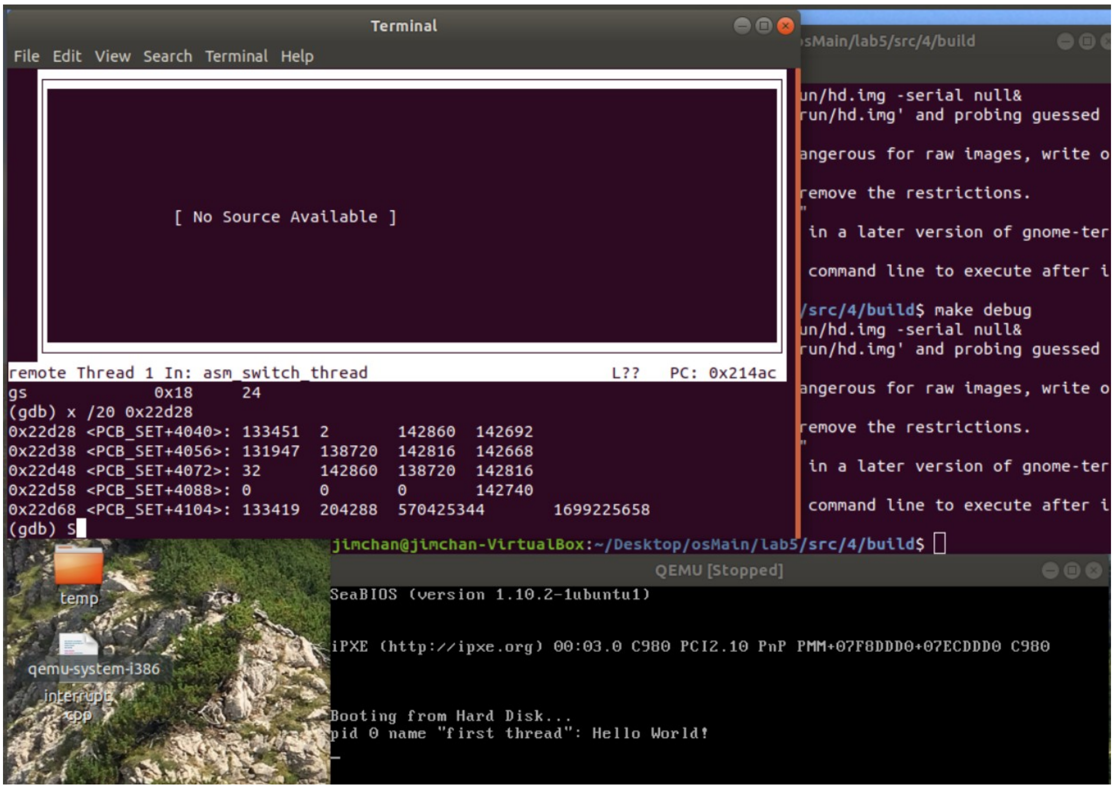
我们可以看到， esp 的值为 PCB_SET + 4152。我们知道对于第 i 个线程的 PCB->stack = PCB_SET + i*4096， 从 ProgramManager::executeThread 的代码，

PCB→stack 的地址 -= 7 = PCB_SET + 1*4096 + 7*8 = PCB_SET + 4152 与 ESP 的值对应。

我们对于 asm_start_thread 设置断点分析：

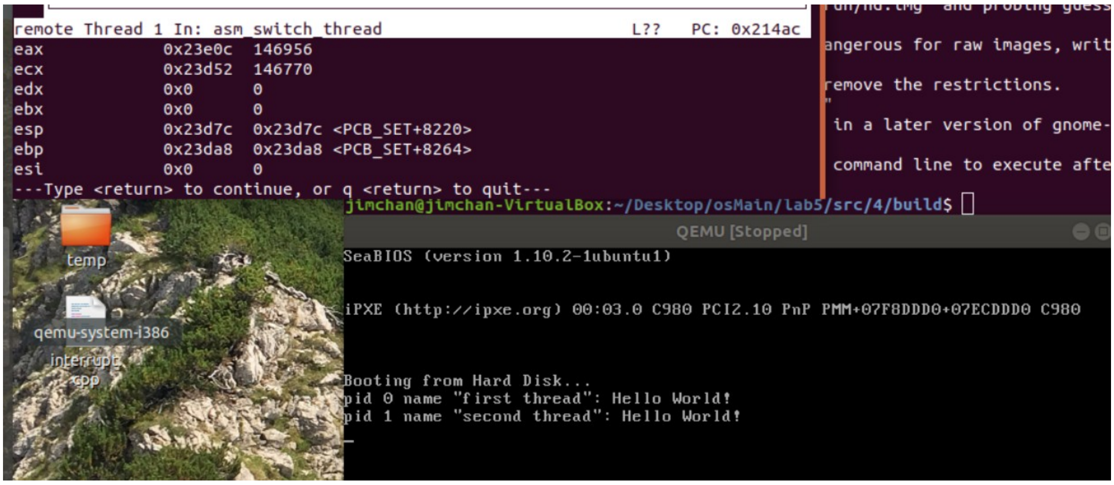


从这里可以看到，断点到 asm_switch_thread 后， 我们有 esp = 0x22d38， ebp = 0x22d64 = 142692，

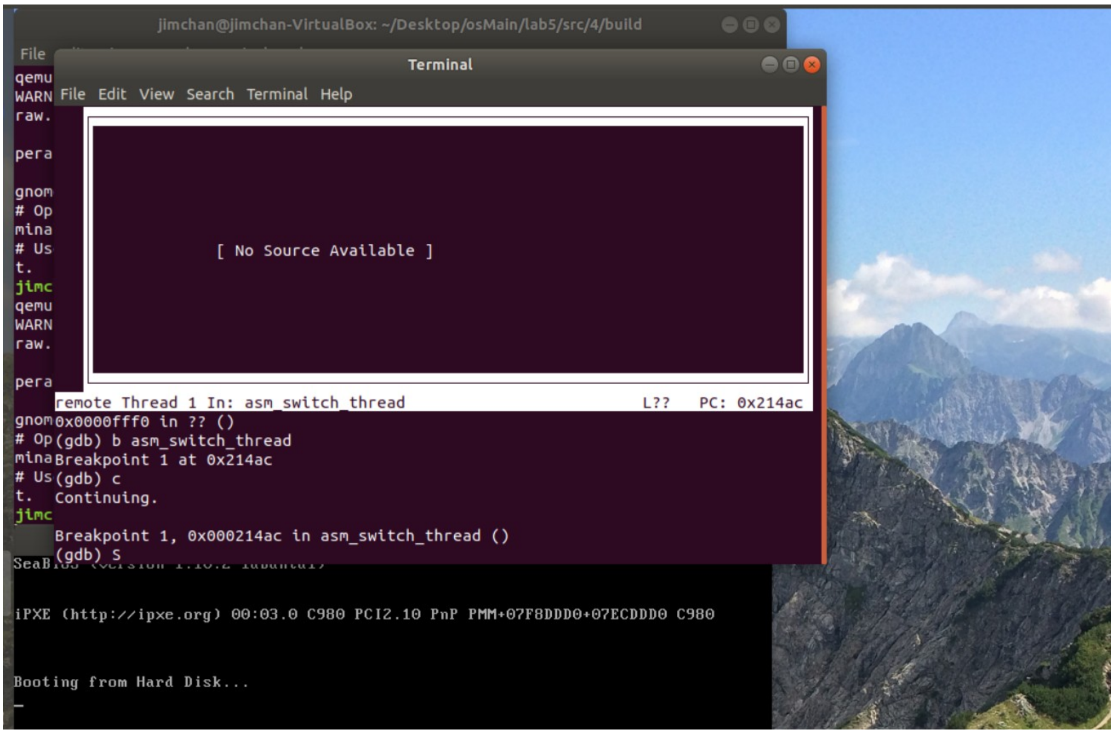


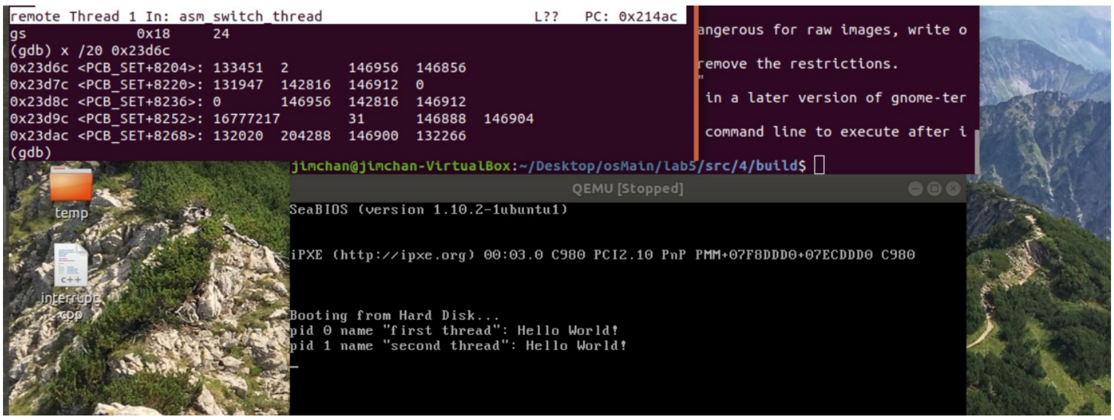
从图上看到， PCB_SET + 4052 的地址里，有值 ebp = 142692，因此我们知道

之后，在执行一次断点到asm_switch_thread 的程序：



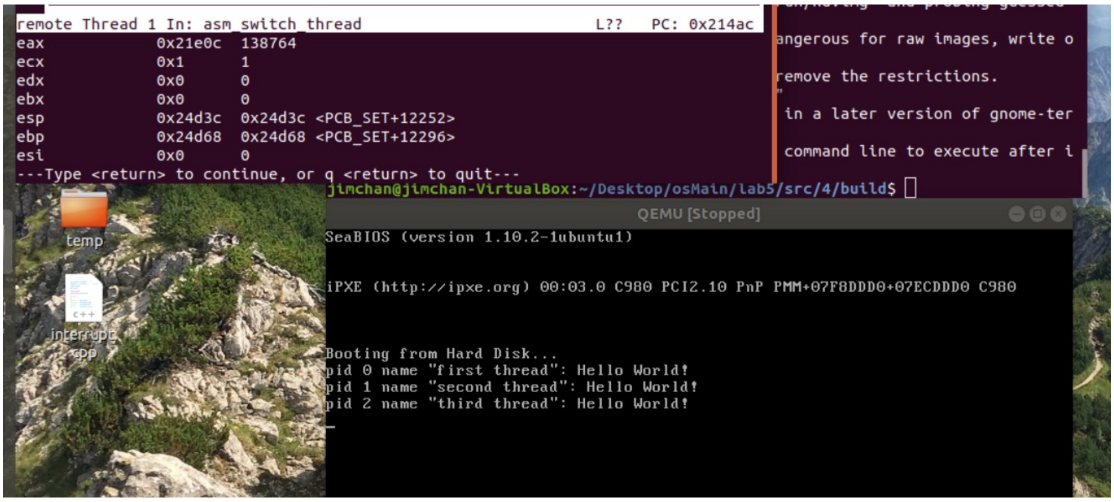
之后， 我们透过

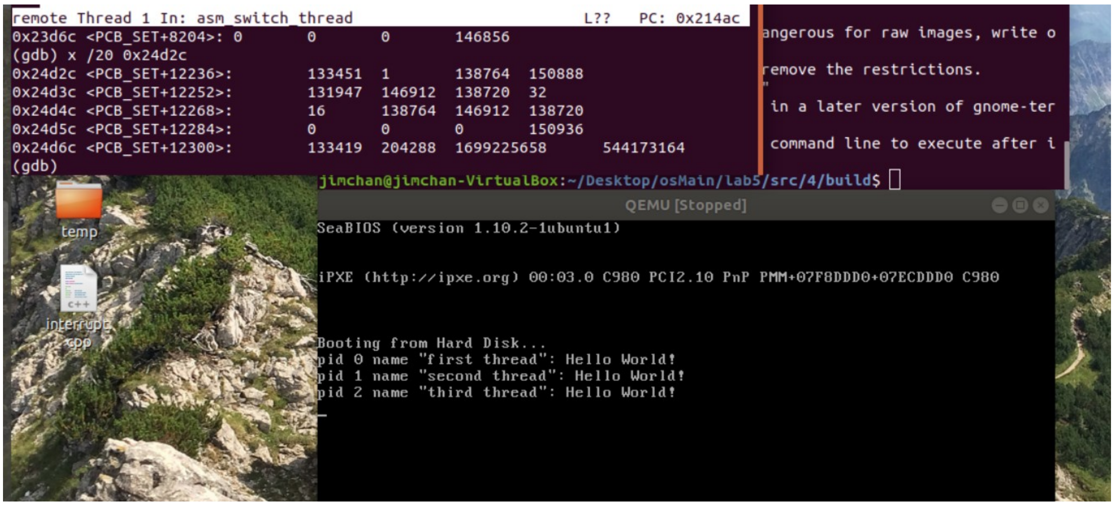




PCB + 8204 放了 ebp 的值

在执行到断点 asm_switch_thread 为止：


remote Thread 1 In: asm_switch_thread L?? PC: 0x214ac
eax 0x21e0c 138764
ecx 0x1 1
edx 0x0 0
ebx 0x0 0
esp 0x24d3c 0x24d3c <PCB_SET+12252>
ebp 0x24d68 0x24d68 <PCB_SET+12296>
esi 0x0 0
---Type <return> to continue, or q <return> to quit---
angerous for raw images, write o
remove the restrictions.
in a later version of gnome-ter
command line to execute after i
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab5/src/4/build$
QEMU [Stopped]
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980
Booting from Hard Disk...
pid 0 name "first thread": Hello World!
pid 1 name "second thread": Hello World!
pid 2 name "third thread": Hello World!
temp
interrupt.cpp


remote Thread 1 In: asm_switch_thread L?? PC: 0x214ac
0x23d6c <PCB_SET+8204>: 0 0 0 146856
(gdb) x /20 0x24d2c
0x24d2c <PCB_SET+12236>: 133451 1 138764 150888
0x24d3c <PCB_SET+12252>: 131947 146912 138720 32
0x24d4c <PCB_SET+12268>: 16 138764 146912 138720
0x24d5c <PCB_SET+12284>: 0 0 0 150936
0x24d6c <PCB_SET+12300>: 133419 204288 1699225658 544173164
(gdb)
angerous for raw images, write o
remove the restrictions.
in a later version of gnome-ter
command line to execute after i
jimchan@jimchan-VirtualBox:~/Desktop/osMain/lab5/src/4/build$
QEMU [Stopped]
SeaBIOS (version 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDD0+07ECDDD0 C980
Booting from Hard Disk...
pid 0 name "first thread": Hello World!
pid 1 name "second thread": Hello World!
pid 2 name "third thread": Hello World!
temp
interrupt.cpp

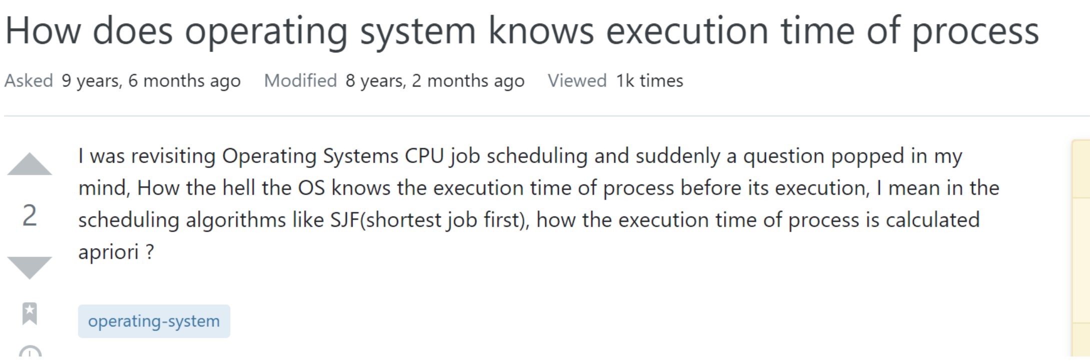
```
asm_switch_thread:
    push ebp
    push ebx
    push edi
    push esi

    mov eax, [esp + 5 * 4]
    mov [eax], esp ; 保存当前栈指针到PCB中，以便日后恢复

    mov eax, [esp + 6 * 4]
    mov esp, [eax] ; 此时栈已经从cur栈切换到next栈

    pop esi
    pop edi
    pop ebx
    pop ebp

    sti
    ret
```

从实验过程中我们可以验证到 asm_switch_thread 的代码。在第 7-8 行， 我们把当前 esp 的值保存到切换下的线程（地址为 eax)，以便下次恢复。

而在第 10-11 行， 我们把要切换上的线程的 next→stack 指针保存在 esp 中。因此，在切换过程中， 我们把当前的线程的状态从栈指针取回， 然后把新的线程的状态写入栈指针。

Assignment 4:

在进行讲解前需要理解一个问题：



How does operating system knows execution time of process

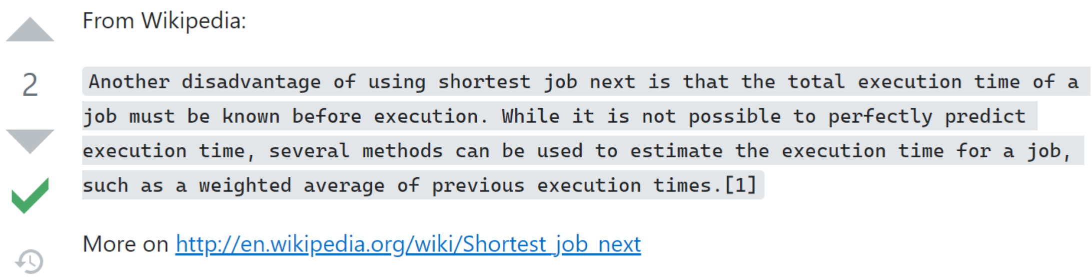
Asked 9 years, 6 months ago Modified 8 years, 2 months ago Viewed 1k times

2

I was revisiting Operating Systems CPU job scheduling and suddenly a question popped in my mind, How the hell the OS knows the execution time of process before its execution, I mean in the scheduling algorithms like SJF(shortest job first), how the execution time of process is calculated apriori ?

operating-system

而答案是我们需要先前知道一个线程的执行时间。



2

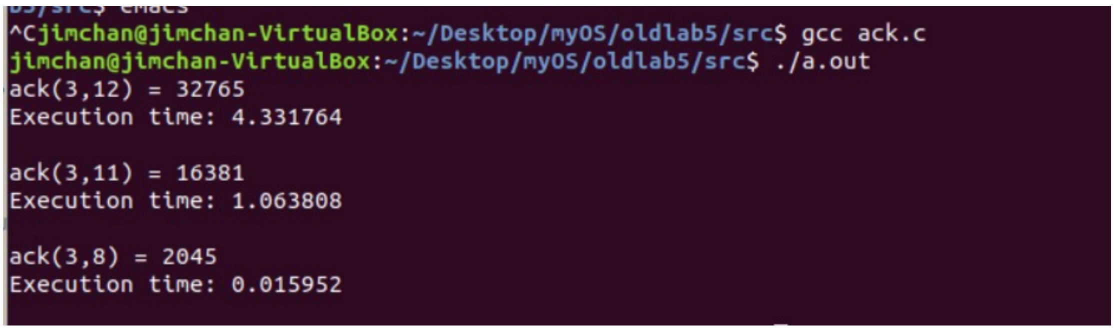
From Wikipedia:

Another disadvantage of using shortest job next is that the total execution time of a job must be known before execution. While it is not possible to perfectly predict execution time, several methods can be used to estimate the execution time for a job, such as a weighted average of previous execution times.[1]

More on http://en.wikipedia.org/wiki/Shortest_job_next

这样我们在写 SJF 的程序的时候， 我们会给一个大概的执行时间（估算）， 也就是我们每个 pcb 又多一个 burstTime 的 member 用来代表线程的执行时间。

为了体会到进程执行的调度和时间的关系， 我们使用 Ackerman 函数来执行在进程里面。



```
^Cjimchan@jimchan-VirtualBox:~/Desktop/myOS/oldlab5/src$ gcc ack.c
jimchan@jimchan-VirtualBox:~/Desktop/myOS/oldlab5/src$ ./a.out
ack(3,12) = 32765
Execution time: 4.331764

ack(3,11) = 16381
Execution time: 1.063808

ack(3,8) = 2045
Execution time: 0.015952
```

并把 burst time 放到 PCB 里面。

注意，我们在这里所需要做的调度算法都不需要用到中断所以我把哪里注释到。

```
// 创建第一个线程
int pid1 = programManager.executeThread(first_thread, nullptr, "first thread", 4, 1);
if (pid1 == -1)
{
    printf("can not execute thread\n");
    asm_halt();
}
int pid2 = programManager.executeThread(second_thread, nullptr, "second thread", 3, 2);
if (pid2 == -1)
{
    printf("can not execute thread\n");
    asm_halt();
}
int pid3 = programManager.executeThread(third_thread, nullptr, "third thread", 1, 5);
if (pid3 == -1)
{
    printf("can not execute thread\n");
    asm_halt();
}
```

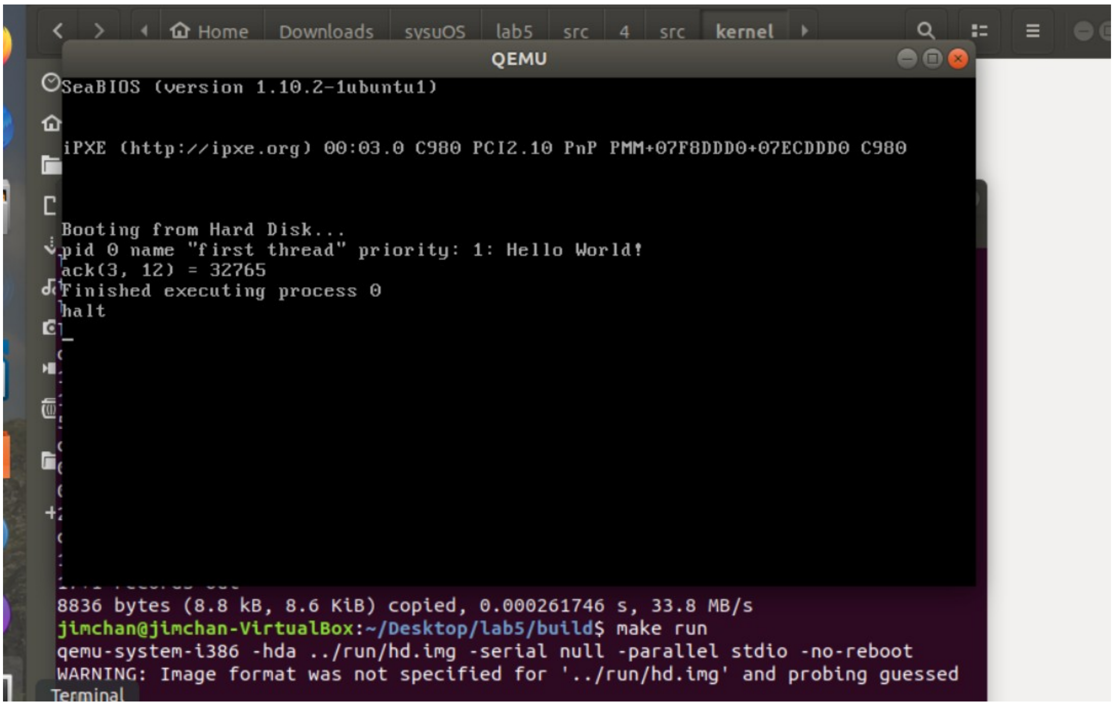
这部分代码设定了：

first_thread 的优先级为 4， 执行时间为5s

second_thread 的优先级为 3， 执行时间为2s

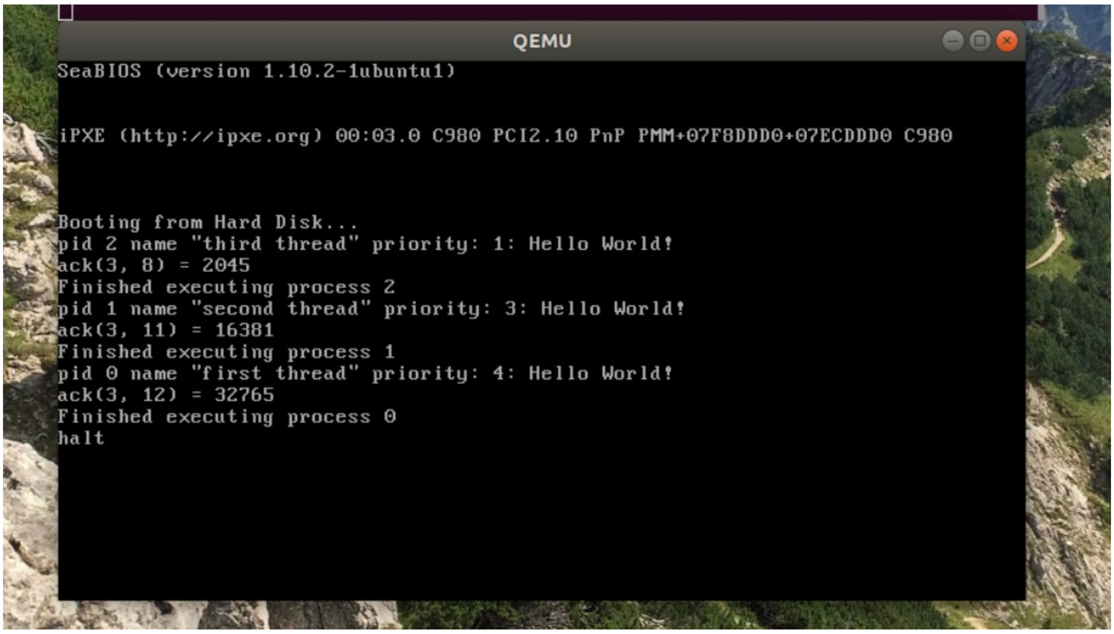
third_thread 的优先级为 1， 执行时间为1s

First come First Serve (FCFS)：



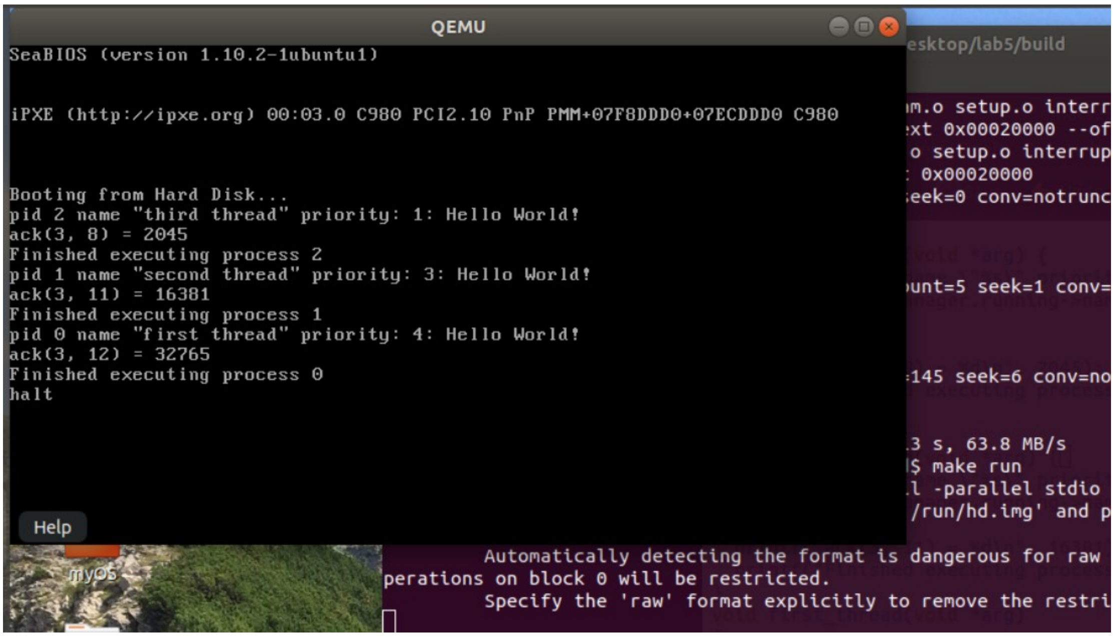
这结果与使用 RR 调度方法一样。

shortest job first (SJF):



注意这里根据执行时间的次序运行。

priority scheduling:



注意这里根据优先级的次序运行。

3. 关键代码

assignment 1:

```
void myPrintf(const char * FormatStr, int numOfArgs, ...)
{
    int formatStrLen = strlen(FormatStr);
    int storeArgFlag = false;
    va_list parameter;
    va_start(parameter, numOfArgs);
    for(int i = 0; i < formatStrLen; i++)
    {
        if(*(FormatStr + i) == '%')
        {
            storeArgFlag = true;
            continue;
        }
        if(storeArgFlag)
        {
            switch(*(FormatStr + i))
            {
                case 'd':
                    std::cout << va_arg(parameter, int);
                    break;
            }
            storeArgFlag = false;
            continue;
        }
        std::cout << *(FormatStr + i);
    }
    va_end(parameter);
    std::cout << std::endl;
}
```

assignment 4:

FCFS 调度：

```cpp
//而RR算法因为要在一段时间片运行所以需要。
void ProgramManager::schedule_FCFS()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

    if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
    else if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }

    ListItem *item = readyPrograms.front();
    PCB *next = ListItem2PCB(item, tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.pop_front();
    asm_switch_thread(cur, next);
    interruptManager.setInterruptStatus(status);
}
```

根据线程进入就绪队列的次序执行。

Shortes job first 调度：

```
void ProgramManager::schedule_SJF()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

    if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }
    int shortestJobTime = ListItem2PCB(readyPrograms.front(), tagInGeneralList)->burstTime;
    int indexOfShortestJob = 0;

    for(int i = 1; i < readyPrograms.size(); i++)
    {
      int tempTime = ListItem2PCB(readyPrograms.at(i), tagInGeneralList)->burstTime;
      if(tempTime < shortestJobTime)
      {
    indexOfShortestJob = i;
    shortestJobTime = tempTime;
      }
    }


    PCB *next = ListItem2PCB(readyPrograms.at(indexOfShortestJob), tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.erase(indexOfShortestJob);

    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

通过找出在就绪对列上最小执行时间的线程执行，我是参考了在数组找出最小的元素的代码。

我在 schedulingAlgorithm.cpp 放了这些调度算法的代码，并有相关注释。

Ack.c 用来估计线程的执行时间。

priority-based 调度;

```
void ProgramManager::schedule_priority()
{
    bool status = interruptManager.getInterruptStatus();
    interruptManager.disableInterrupt();

    if (readyPrograms.size() == 0)
    {
        interruptManager.setInterruptStatus(status);
        return;
    }

    if (running->status == ProgramStatus::DEAD)
    {
        releasePCB(running);
    }

    ListItem *nextItem = nullptr;

    while(nextItem == nullptr)
    {

      ListItem *tempItem;

      for(int i = 0; i < readyPrograms.size(); i++)
      {
    tempItem = readyPrograms.at(i);
    if(ListItem2PCB(tempItem, tagInGeneralList)->priority == currentPriority)
    {

      nextItem = tempItem;
      break;
    }
      }
      if(nextItem == nullptr) ++currentPriority;
    }
    PCB *next = ListItem2PCB(nextItem, tagInGeneralList);
    PCB *cur = running;
    next->status = ProgramStatus::RUNNING;
    running = next;
    readyPrograms.erase(nextItem);

    asm_switch_thread(cur, next);

    interruptManager.setInterruptStatus(status);
}
```

我们知道优先度的值小就是更优先要处理的。因此，我在 ProgramManager 类增加了 int currentPriority 用来记录现在执行的线程的优先度 schedulePriority()每次会找目前优先度的线程，如果找不到，就会 increment currentPriority，从而执行下一个优先度的线程，如此类推。

4. 总结

通过实验了解上下切换和线程调度的重要概念。