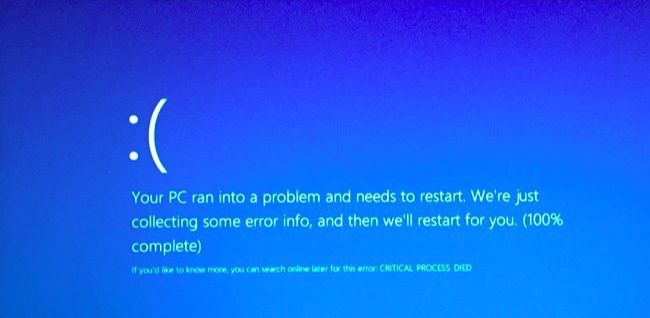
第十三章

这一章我们来聊聊线程吧。在第一章中我们谈到过，操作系统的几大功能，而线程，无疑是其中的重点。Edsger Dijkstra（学过数据结构的肯定熟悉他）曾说过：If debugging is the process of removing software bugs, then programming must be the process of putting them in.

而时至今日，线程之间的通信，资源竞争等问题也没有完全解决。从当年我们熟悉的XP下的错误：



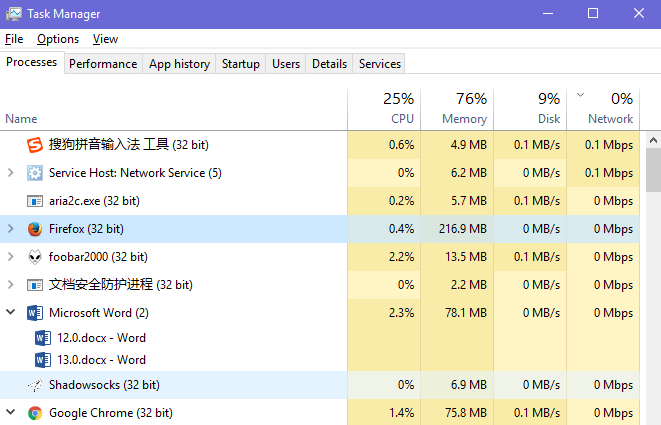
到win10下精致的死机界面：



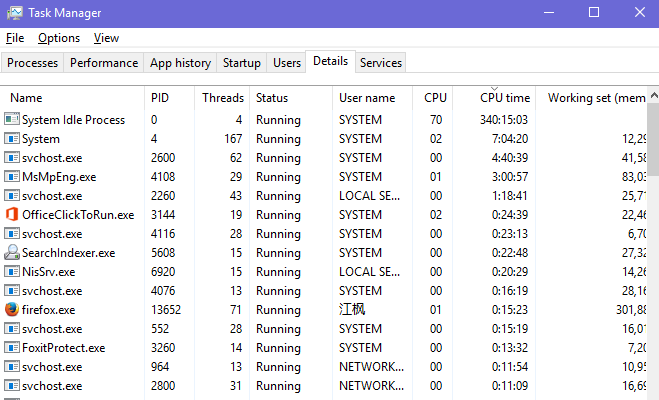
其中线程可以说是贡献了不少的错误。因此对于这部分的工作，历来是操作系统研究的重点。

要做线程的管理，先得看看什么是线程。而要说线程（threads），我们还是先来说说进程(process)。

进程是我们很熟悉的一个字眼了，所谓进程，就是内存中程序或者程序的一部分，在任务管理器里看到的，就是进程：



现在我们所看到的，就是进程。这里的大部分程序都只包括一个进程，而像是有些程序，就需要多个进程，比如chrome，你就可以在这里找到好几个chrome，它们各有不同的作用。而在详细(details)那一栏，我们可以看到它们的线程：



这里的threads，指的就是线程。

所以定义如下：

进程

每个进程提供执行程序所需的资源。 进程具有虚拟地址空间，可执行代码，系统对象的打开句柄，上下文，唯一进程标识符，环境变量，优先级类，最小和最大工作集大小以及至少一个执行线程。 每个进程都是通过单线程启动的，通常称为主线程，但是可以从其任何线程创建其他线程。

线程

线程是进程内的一个实体，可以安排执行。 进程的所有线程共享其虚拟地址空间和系统资源。 此外，每个线程都维护异常处理程序，调度优先级，线程本地存储，唯一的线程标识符以及系统将用于保存线程上下文直到调度的一组结构。 线程上下文包括线程的进程的地址空间中的线程的机器寄存器，内核栈，线程环境块和用户堆栈。 线程还可以拥有自己的安全上下文，可用于模拟客户端。

这个定义有点晦涩，但简单来讲，线程是进程的一部分，一个进程内部的线程共享资源，而进程之间不共享资源。**在我们这里，实现的是内核级的线程，但实际上也是进程**，因为之间没有共享资源，我们通过最简单的单向循环链表来调度。

从进程的角度来看，它一般具备动态分配内存和使用堆栈空间的调用方法。操作系统要做的一个工作，就是为进程堆栈和堆内存分配空间以供进程使用。对于栈来说，如果进程没有声明，分配的空间大小就都是一致的。而对于堆来说，每个进程在用户模式下都有自己的堆分配器（heap allocator）。 这是使用C运行时库（C Runtime Library，CRT）中类似malloc，free，realloc，brk和sbrk这样的接口实现的。当程序与CRT链接时，就可以调用这些库函数来为自己分配堆内存。或者程序也可以自己实现类似CRT的库来实现。

我们知道现代操作系统都是按照时间片来给进程分配计算资源的。通过这种方式实现多任务（multi task）的同时处理，而一种常用的实现方式就是**抢占式多任务处理**（Preemption）。这个也是现代操作系统广泛使用的一种方式，抢占就是抢占是暂时中断计算机系统正在执行的任务的行为，并在稍后恢复该任务。 所执行的任务的这种改变就被称为上下文切换（context switches）。 它通常由特权任务或系统的抢占式调度器来执行。而在这一过程中为了对进程的特定信息做管理，我们使用进程管理块（process control block,PCB）来实现。

进程管理块（PCB）是用于存储关于过程或任务的信息的数据结构。 PCB包含诸如中断描述符指针，页目录基址寄存器（PDBR）等信息。 保护级别，运行时间，进程状态，进程标志，VM86标志，优先级和进程ID（Process ID，PID）等信息。 PCB的具体信息和操作系统有关。

我们这里的进程切换也使用PCB来实现，看一下我们的头文件：

/include/task.h

1. #ifndef INCLUDE\_TASK\_H\_
2. #define INCLUDE\_TASK\_H\_
3. #include "types.h"
4. #include "pmm.h"
5. #include "vmm.h"
6. // PID结构定义
7. typedef int32\_t pid\_t;
8. // 进程状态描述
9. typedef
10. enum task\_state {
11. TASK\_UNINIT = 0, // 未初始化
12. TASK\_SLEEPING = 1, // 睡眠中
13. TASK\_RUNNABLE = 2, // 可运行(也许正在运行)
14. TASK\_ZOMBIE = 3, // 僵尸状态
15. } task\_state;
16. // 内核线程的上下文切换保存的信息
17. struct context {
18. uint32\_t esp;
19. uint32\_t ebp;
20. uint32\_t ebx;
21. uint32\_t esi;
22. uint32\_t edi;
23. uint32\_t eflags;
24. };
25. // 进程内存地址结构
26. struct mm\_struct {
27. pgd\_t \*pgd\_dir; // 进程页表
28. };
29. // 进程控制块 PCB
30. struct task\_struct {
31. volatile task\_state state; // 进程当前状态
32. pid\_t pid; // 进程标识符
33. void \*stack; // 进程的内核栈地址
34. struct mm\_struct \*mm; // 当前进程的内存地址映像
35. struct context context; // 进程切换需要的上下文信息
36. struct task\_struct \*next; // 链表指针
37. };
38. // 全局 pid 值
39. extern pid\_t now\_pid;
40. // 内核线程创建
41. int32\_t kernel\_thread(int (\*fn)(void \*), void \*arg);
42. // 线程退出函数
43. void kthread\_exit();
44. #endif // INCLUDE\_TASK\_H\_

这个头文件核心的就是我们对于PCB的定义了，可以看出来，在我们这个小内核里，定义了它6方面的信息。我们这里做的，是内核级的线程（和当年XP下的硬件驱动一样，所以有一点小错误也可能会导致宕机，ps，宕机比死机好听多了），线程之间共享全局资源，共用相同的页表，只有栈是私有的。所以这里就不涉及到创建另一个页目录了。

PCB定义之后，后面的内容，包括PID值，线程的创建和退出函数等下再讲。

除去PCB之外，我们还需要一个调度程序（scheduler），调度程序有不同的算法来实现调度，通常的通用调度算法包括但不限于先进先出，最短剩余时间，固定优先级抢占，循环和多级队列等。这些调度算法的实现需要在PCB中设置对应的优先级等信息结构。使用最广的，也是 Windows和Linux使用的算法是多级反馈队列（multilevel feedback queue）。但我们这里的实现就是一个单向循环列表，没什么trick的算法。

先看下头文件吧：

/include/sched.h

1. #ifndef INCLUDE\_SCHEDULER\_H\_
2. #define INCLUDE\_SCHEDULER\_H\_
3. #include "task.h"
4. // 可调度进程链表
5. extern struct task\_struct \*running\_proc\_head;
6. // 当前运行的任务
7. extern struct task\_struct \*current;
8. // 在entry.c里定义的栈顶变量
9. extern uint32\_t kern\_stack\_top;
10. // 初始化任务调度
11. void init\_sched();
12. // 任务调度
13. void schedule();
14. // 任务切换准备
15. void change\_task\_to(struct task\_struct \*next);
16. // 任务切换
17. void switch\_to(struct context \*prev, struct context \*next);
18. #endif // INCLUDE\_SCHEDULER\_H\_

这里的内容其实都很简单，首先我们定义了两个PCB格式的结构体，一个用来存放我们当前正在运行的进程，一个存放我们要进行调度的进程。前面说过我们要做的调度很简单，就是单向循环链表，所以就是这两个进程来回转换。

13行里我们声明了外部变量，kern\_stack\_top，这个是在初始化调度里用到的一个变量。

16行是我们调度程序的一个初始化。

18行到25行都是我们的调度程序，为了方便把它分成了三个函数而已。

看一下我们具体的调度函数吧：

/kernel/sched.c

1. #include "sched.h"
2. #include "heap.h"
3. #include "print.h"
4. // 可调度进程链表
5. struct task\_struct \*running\_proc\_head = NULL;
6. // 当前运行的任务
7. struct task\_struct \*current = NULL;
8. void init\_sched()
9. {
10. // 为当前执行流创建信息结构体 该结构位于当前执行流的栈最低端
11. current = (struct task\_struct \*)(kern\_stack\_top - STACK\_SIZE);
12. current->state = TASK\_RUNNABLE;
13. current->pid = now\_pid++;
14. current->stack = current; // 该成员指向栈低地址
15. current->mm = NULL; // 内核线程不需要该成员
16. // 单向循环链表
17. current->next = current;
18. running\_proc\_head = current;
19. }
20. void schedule()
21. {
22. if (current) {
23. change\_task\_to(current->next);
24. }
25. }
26. void change\_task\_to(struct task\_struct \*next)
27. {
28. if (current != next) {
29. struct task\_struct \*prev = current;
30. current = next;
31. switch\_to(&(prev->context), &(current->context));
32. }
33. }

这段程序里，我们首先声明了我们的两个PCB，然后就是初始化我们的调度程序了。那么我们为什么要初始化呢？又是怎样初始化的呢？

先说说为什么要初始化，我们既然要把我们的程序（或者说是代码）用进程/线程的形式进行管理，那么我们就不能漏掉我们的前面的内容，在我们进入进程管理时，也应当把前面执行的部分使用一个或多个进程描述。我们的初始化就是做的这个工作，我们模仿了Linux早期的做法，把我们前面执行流的PCB放置在了线程栈的最低处（如果对这里的代码有疑问，回去看看我们前面内存管理初始化那部分的工作）。然后我们更新了PCB的其他元素。最后我们把这个链表指向了自己，因为我们这里还没有创建新的进程，当我们创建新的进程时，会更新这个链表的。

下面的几个函数就是进程调度函数了，这里有两个，还有一个用汇编写的：

/kernel/switch\_to.s

1. [global switch\_to]
2. ; 具体的线程切换操作，重点在于寄存器的保存与恢复
3. switch\_to:
4. mov eax, [esp+4]
5. mov [eax+0], esp
6. mov [eax+4], ebp
7. mov [eax+8], ebx
8. mov [eax+12], esi
9. mov [eax+16], edi
10. pushf
11. pop ecx
12. mov [eax+20], ecx
13. mov eax, [esp+8]
14. mov esp, [eax+0]
15. mov ebp, [eax+4]
16. mov ebx, [eax+8]
17. mov esi, [eax+12]
18. mov edi, [eax+16]
19. mov eax, [eax+20]
20. push eax
21. popf
23. ret

这部分的代码主要实现了我们对寄存器内容的保护，然后我们切换了栈，这样在ret指令后返回的地址就是我们转换后的栈的位置了，所以也就切换了我们的进程。

这些内容都完备了，我们来看看如何创建一个进程吧：

/kernel/task.c

1. #include "gdt.h"
2. #include "pmm.h"
3. #include "vmm.h"
4. #include "heap.h"
5. #include "task.h"
6. #include "sched.h"
7. #include "string.h"
8. #include "print.h"
9. // 全局 pid 值
10. pid\_t now\_pid = 0;
11. // 内核线程创建
12. int32\_t kernel\_thread(int (\*fn)(void \*), void \*arg)
13. {
14. struct task\_struct \*new\_task = (struct task\_struct \*)kmalloc(STACK\_SIZE);
15. error\_detect(new\_task != NULL, "kern\_thread: kmalloc error");
16. // 将栈低端结构信息初始化为 0
17. bzero(new\_task, sizeof(struct task\_struct));
18. new\_task->state = TASK\_RUNNABLE;
19. new\_task->stack = current;
20. new\_task->pid = now\_pid++;
21. new\_task->mm = NULL;
22. uint32\_t \*stack\_top = (uint32\_t \*)((uint32\_t)new\_task + STACK\_SIZE);
23. \*(--stack\_top) = (uint32\_t)arg;
24. \*(--stack\_top) = (uint32\_t)kthread\_exit;
25. \*(--stack\_top) = (uint32\_t)fn;
26. new\_task->context.esp = (uint32\_t)new\_task + STACK\_SIZE - sizeof(uint32\_t) \* 3;
27. // 设置新任务的标志寄存器未屏蔽中断，很重要
28. new\_task->context.eflags = 0x200;
29. new\_task->next = running\_proc\_head;
31. // 找到当前进任务队列，插入到末尾
32. struct task\_struct \*tail = running\_proc\_head;
33. error\_detect(tail != NULL, "Must init sched!");
34. while (tail->next != running\_proc\_head) {
35. tail = tail->next;
36. }
37. tail->next = new\_task;
38. return new\_task->pid;
39. }
40. void kthread\_exit()
41. {
42. register uint32\_t val asm ("eax");
43. printk("Thread exited with value %d\n", val);
44. while (1);
45. }

<https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx>

<http://www.brokenthorn.com/Resources/OSDev24.html>

https://en.wikipedia.org/wiki/Preemption\_(computing)