期末報告

操作系統：三篇文章中文版

作者:黃炯瑞　日期：2020/6/26

本文與圖片大多來自書本約前十章內容

本書簡介：

本書的英文書名為《Operating Systems:Three Easy Pieces》，全篇總計 50 章，圍繞著三個主題元素展開講解。

(1).虛擬化 （virtualization）

(2).併發 （concurrency）

(3).持久性 （persistence）

每個主要概念在若干章節中加以闡釋，其中大部分章節都是提出一个特定的問題，然後展示解决它的方法。

什麼是虛擬化？

虛擬化是透過軟體以虛擬形式呈現物件的過程。

想像一下有一台(實體)車，大家都想開，而且每個人都希望自己有一台。現在用一種特殊技術把車子變成很多(虛擬)車，這樣就可以給所有人了，因為人不可能一直使用車，在沒使用時交給別人去使用。在這種幻覺之下，明明能開的車只有一台，但大家表面上都有車。

CPU 也是，通過讓一個進程只運行一小時間片段，然後切換到其他的進程來提供存在多個虛擬 CPU 的假象。

CH4. 抽象：進程

操作系統為正在運行的程序提供的抽象，就是所谓的進程（process），一個進程只是一個正在運行的程序。

操作系統的所有接口必須包含以下的內容，也就是說，所有現代操作系統都以某種形式提供這些 API。

(1). 創建（create）：操作系統必須包含一些創建新進程的方法。在 shell 中键入命令时，會調用操作系統来創建新進程，運行指定的程序。
(2).銷毀（destroy）：由於存在創建進程的接口，因此系統還提供了强制銷毀進程的接口。用來預防進程不退出的狀況。
(3).等待（wait）：有時等待進程停止運行是有用的，因此經常提供某種等待接口。
(4).其他控制（miscellaneous control）：除了殺死或等待进程外，有時還可能有其他控制。例如，大多數操作系統提供某种方法来暂停進程（停止運行一段时间），然后恢復（繼續運行）。
(5).狀態（statu）：通常也有一些接口可以獲得有關進程的狀態信息，例如運行了多久，或者處於何種狀態。

進程的狀態可能處於以下情況:

(1).初始（initial）:表示進程處於創建時的狀態。
(2). 運行（running）：在此狀態下，進程正在處理器上運行。意味着它正在執行指令。
(3). 就绪（ready）：在就绪狀態下，進程已準備好運行，但出于某種原因，操作系統選擇不在此时運行。
(4). 阻塞（blocked）：在阻塞状态下，一個進程執行了某種操作，直到發生其他事件才準備運行。

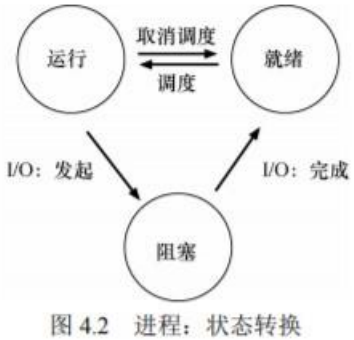

图 4.2 进程：状态转换

狀態映射圖

如果把狀態映射到圖上，如上圖所示。根據系統載量，使其在就緒與運行中互相轉換。想象有兩個正在運行的進程，每個進程只使用 CPU（它们没有 I/O）。表 4.1

表 4.1　跟踪进程状态：只看 CPU

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	
4	运行	就绪	Process0 现在完成
5	—	运行	
6	—	运行	
7	—	运行	
8	—	运行	Process1 现在完成

在下面例子中，第一个進程在運行一段时间后發起 I/O 要求。此時進程阻塞，讓另一進程運行。如表 4.2

表 4.2　跟踪进程状态：CPU 和 I/O

时间	Process0	Process1	注
1	运行	就绪	
2	运行	就绪	
3	运行	就绪	Process0 发起 I/O
4	阻塞	运行	Process0 被阻塞
5	阻塞	运行	所以 Process1 运行
6	阻塞	运行	
7	就绪	运行	I/O 完成
8	就绪	运行	Process1 现在完成
9	运行	—	
10	运行	—	Process0 现在完成

另外，除了上述的運行、就緒、阻礙之外，還有其他一些進程可以處於的狀態。最終(final)狀態，它允许其他進程檢查進程的返回碼，并查看剛完成的進程是否成功執行。

补充：数据结构——进程列表

操作系统充满了我们将在这些讲义中讨论的各种重要数据结构（data structure）。进程列表（process list）是第一个这样的结构。这是比较简单的一种，但是，任何能够同时运行多个程序的操作系统当然都会有类似这种结构的东西，以便跟踪系统中正在运行的所有程序。有时候人们会将存储关于进程的信息的个体结构称为进程控制块（Process Control Block，PCB），这是谈论包含每个进程信息的 C 结构的一种方式。

CH5. 插叙：進程 API

此章節通過 2 個系統調用:fork()、exec()，並調用 wait()來等待其創建的子進程執行完成。

fork() 系統調用:

```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    //輸出hello world 與 PID
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // 子進程
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // 父進程(main)
        printf("hello, I am parent of %d (pid:%d)\n",
        rc, (int) getpid());
    }
    return 0;
}
```

```
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
```

進程調用了 fork()，这是操作系统提供的創建新進程的方法。過程與調用進程完全一模一樣。新創建的為子進程(child)，原本進程為父進程(parent)。又因為 hello world 只輸出一次，證明了子進程並沒有從 main()開始跑，而是直接跑 fork()，就好像自己調用 fork()一樣。父進程所獲得的值是子進程的 PID，而子進程則為 0。當然，子進程也有可能先行運行，父進程後運行。

wait()系統調用

```c
int
main(int argc, char *argv[])
    {
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    } else { // parent goes down this path (main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

這邊wait()跟上一個fork()相比，在父進程的部分多了

```c
int wc = wait(NULL);
```

這一行，這代表著不管是父進程先運行還是子進程先運行，到父進程時，都會等一下，這也意味著其輸出結果就會被確定如下:

```
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
```

exec()系統調用

```c
int
main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());
    int rc = fork();
    if (rc < 0) { // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) { // 子進程
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc"); // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL; // marks end of array
        execvp(myargs[0], myargs); // runs word count
        printf("this shouldn't print out");
    } else { // 父進程(main)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

```
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (wc:29384) (pid:29383)
```

子進程調用 execvp()来運行字符記數程序 wc。事實上，它針對源代码運行 wc，告訴我們此文件有多少行、多少單辭，以及多少字節。

除了上述三個之外，還有其他進程交互的方式，如利用 kill()發出訊號，要求進程睡眠或是終止等等。

CH6 機制：受限直接執行

為何要受限直接執行?原因有二:

(1)在於如何讓他只單一運行一個程序，操作系统怎么能確保程序不做任何我們不希望它做的事，同時仍然高效地運行它？
(2)當運行一個進程時，操作系统如何讓它停下来并切到另一個進程，從而實現虛擬化 CPU 所需的時分共享？

关键问题：如何高效、可控地虚拟化 CPU

操作系统必须以高性能的方式虚拟化 CPU，同时保持对系统的控制。为此，需要硬件和操作系统支持。操作系统通常会明智地利用硬件支持，以便高效地实现其工作。

提示：采用受保护的控制权转移

硬件通过提供不同的执行模式来协助操作系统。在用户模式（user mode）下，应用程序不能完全访问硬件资源。在内核模式（kernel mode）下，操作系统可以访问机器的全部资源。还提供了陷入（trap）内核和从陷阱返回（return-from-trap）到用户模式程序的特别说明，以及一些指令，让操作系统告诉硬件陷阱表（trap table）在内存中的位置。

有二種模式:

(1)用户模式（user mode）。在用户模式下運行的代碼會受到限制。
(2)内核模式（kernel mode）。在此模式下，代碼可以做它喜歡的事。

(操作系统就以內核模式運行)

要執行系統調用，程序必須執行特殊的陷阱（trap）指令。内核通過在啟動時設置陷阱表（trap table）来實現。

LDE 協議有兩階段:

(1)第一个階段（系统引導時），内核初始化陷阱表，并且 CPU 記住它的位置以供随后使用。内核通过特權指令来執行此操作
(2)第二个阶段（運行進程時），在使用从陷阱返回指令开始值行進程之前，内核設置了一些内容（如，在進程列表中分配一個節點，分配内存）。

补充：上下文切换要多长时间

你可能有一个很自然的问题：上下文切换需要多长时间？甚至系统调用要多长时间？如果感到好奇，有一种称为 lmbench [MS96]的工具，可以准确衡量这些事情，并提供其他一些可能相关的性能指标。

随着时间的推移，结果有了很大的提高，大致跟上了处理器的性能提高。例如，1996 年在 200-MHz P6 CPU 上运行 Linux 1.3.37，系统调用花费了大约 4μs，上下文切换时间大约为 6μs[MS96]。现代系统的性能几乎可以提高一个数量级，在具有 2 GHz 或 3 GHz 处理器的系统上的性能可以达到亚微秒级。

应该注意的是，并非所有的操作系统操作都会跟踪 CPU 的性能。正如 Ousterhout 所说的，许多操作系统操作都是内存密集型的，而随着时间的推移，内存带宽并没有像处理器速度那样显著提高[O90]。因此，根据你的工作负载，购买最新、性能好的处理器可能不会像你希望的那样加速操作系统。

表 6.2　　受限直接运行协议

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住系统调用处理程序的地址	
操作系统@运行（内核模式）	硬件	程序（应用模式）
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argv 设置程序栈 用寄存器/程序计数器填充内核栈 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到 main	
		运行 main …… 调用系统调用 陷入操作系统
	将寄存器保存到内核栈 转向内核模式 跳到陷阱处理程序	
处理陷阱 做系统调用的工作 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到陷阱之后的程序计数器	
		……从 main 返回 陷入（通过 exit()）
释放进程的内存将进程 从进程列表中清除		

時間線隨時間向下(表 6.2)。假設每個進程都有一个内核線，在進入内核和離開内核時，寄存器分别被保存和恢復。

表 6.3　　受限直接执行协议（时钟中断）

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 系统调用处理程序 时钟处理程序	
启动中断时钟		
	启动时钟 每隔 *x* ms 中断 CPU	
操作系统@运行（内核模式）	硬件	程序（应用模式）
		进程 A……
	时钟中断 将寄存器（A）保存到内核栈（A） 转向内核模式 跳到陷阱处理程序	
处理陷阱 调用 switch()例程 将寄存器(A)保存到进程结构(A) 将进程结构(B)恢复到寄存器(B) 从陷阱返回（进入 B）		
	从内核栈（B）恢复寄存器（B） 转向用户模式 跳到 B 的程序计数器	
		进程 B……

表 6.3 中顯示了時鐘中斷的狀況，A 運行時中斷，保存 A，跳 B，切換上下文，最後從陷阱中返回，開始運行 B。

CH7 介紹進程調度(以下部分出自 MBA 智庫)

一、什麼是進程調度?

進程調度是指操作系統按某種策略或規則選擇進程占用 CPU 進行運行的過程。

二、內容:

按一定的策略,動態地把處理機分配給處於就緒隊列中的某一個進程,以使之執行。

三、工作負載假设:

(1)·每一個工作運行相同時間
(2)·所有工作同時抵達
(3)·一旦開始,就會保持運行直到完成。
(4)·所有的工作只用 CPU
(5)·每個工作運行時間已知

四、功能:

根據各進程的狀態特征和資源需求等、進程管理模塊還將各進程的 PCB 表排成相應的隊列並進行動態隊列轉接。

五、演算法:

(1).先進先出演算法: 演算法把處理機分配給最先進入就緒隊列的進程

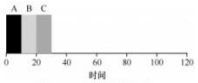

图 7.1 FIFO 的简单例子

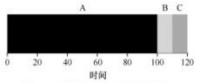

图 7.2 为什么 FIFO 没有那么好

(2).短進程優先: 從就緒隊列中選出下一個“CPU 執行期最短”的進程,為之分配處理機。

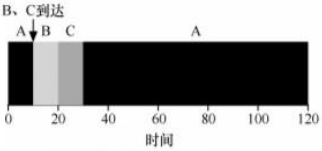

图 7.5 STCF 的简单例子

(3).輪轉法: 系統將所有就緒進程按 FIFO 規則排隊，按一定的時間間隔把處理機分配給隊列中的進程。

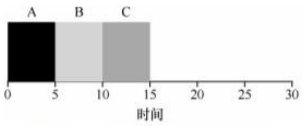

图 7.6 又是 SJF（响应时间不好）

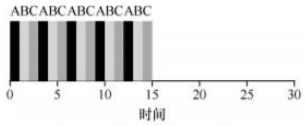

图 7.7 轮转（响应时间好）

(4). 多級隊列方法：將系統中所有進程分成若幹類，每類為一級。

(5). 多級反饋隊列：多級反饋隊列方式是在系統中設置多個就緒隊列，並賦予各隊列以不同的優先權。(下一章)

六、原因:

1. 正在執行的進程執行完畢。這時，如果不選擇新的就緒進程執行，將浪費處理機資源。
2. 執行中進程自己調用阻塞原語將自己阻塞起來進入睡眠等狀態。
3. 執行中進程調用了 P 原語操作，從而因資源不足而被阻塞；或調用了 v 原語操作激活了等待資源的進程隊列。
4. 執行中進程提出 I/O 請求後被阻塞。
5. 在分時系統中時間片已經用完。
6. 在執行完系統調用等系統程式後返回用戶進程時，這時可看作系統進程執行完畢，從而可調度選擇一新的用戶進程執行。
7. 就緒隊列中的某進程的優先順序變得高於當前執行進程的優先順序，從而也將引發進程調度

七、進程調度與作業調度的區別:

（1）作業調度是從作業後備隊列選擇一個或多個作業，為其分配必要的資源，並為之創建進程，做好運行前的準備。按照一定的作業調度演算法，從後備作業隊列中選擇一個或幾個作業調入記憶體。

（2）進程調度是指從已經進入記憶體的進程的就緒隊列中選擇一個進程真正占有 CPU，並為其運行進行上下文轉換，讓其立即運行。按照一定的進程調度演算法，從記憶體的進程中選擇一個進程，將處理機分配給它，使其執行。

（3）作業調度是主要解決進程真正在 CPU 上運行的問題。進程調度是主要解決作業有無資格占有 CPU 的問題。

CH8 調度：多级反饋隊列

一、基本規則:

i. 如果 A 的優先級> B 的優先級，運行 A（不運行 B)。

ii. 如果 A 的優先級= B 的優先級，輪轉運行 A 和 B。

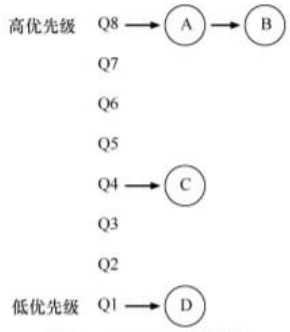

图 8.1 MLFQ 的例子

二、如何改變優先級:

i. 工作進入系统時，放在最高優先级（最上層隊列)。

ii. 工作用完整個時間片後，降低其優先级（移入下一個對列)。如果工作在其時間片以内主動釋放 CPU，則優先級不變。

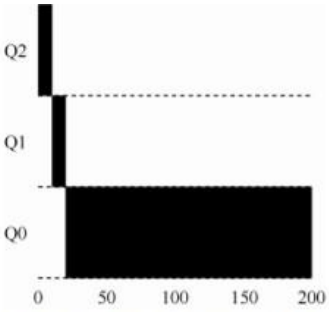

图 8.2 长时间工作随时间的变化

三、提升優先級:

i. 經過一段時間 S，就將系统中所有工作重新加入最高優先級。

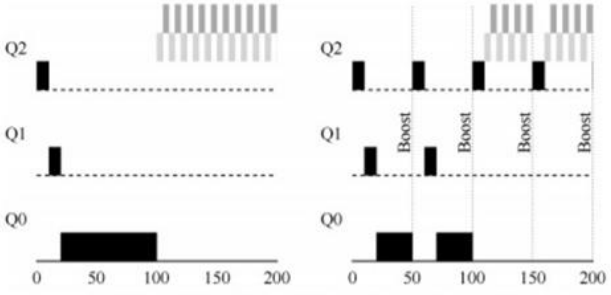

图 8.5 不采用优先级提升（左）和采用（右）

四、更好的計時方式:

i. 一旦工作用完了其在某一層中的時間配額，就降低其優先級

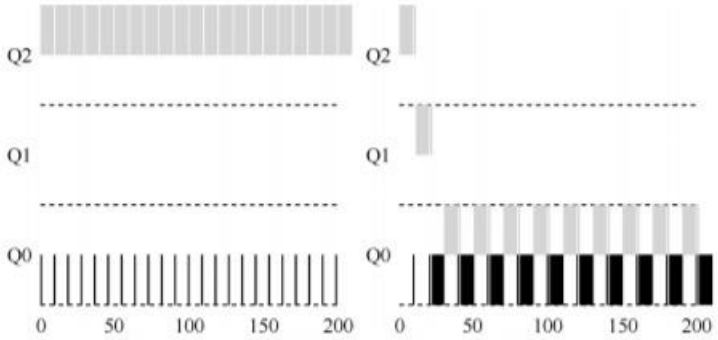

图 8.6 不采用愚弄反制（左）和采用（右）

五、其他問題:

i. 配置多少隊列？

ii. 每一層隊列的時間片配置多大？

iii. 为了避免飢餓問題以及進程行為改變，應該多久提升一次進程的優先級？

提示：避免巫毒常量（Ousterhout 定律）

尽可能避免巫毒常量是个好主意。然而，从上面的例子可以看出，这通常很难。当然，我们也可以让系统自己去学习一个很优化的值，但这同样也不容易。因此，通常我们会有一个写满各种参数值默认值的配置文件，使得系统管理员可以方便地进行修改调整。然而，大多数使用者并不会去修改这些默认值，这时就寄希望于默认值合适了。这个提示是由资深的 OS 教授 John Ousterhout 提出的，因此称为 Ousterhout 定律（Ousterhout's Law）。

CH9 彩票調度

一、基本概念：彩票數表示份額:

彩票数（ticket）代表了進程占有某资源的份额。一個進程擁有的彩票數占總彩票数的百分比，就是它占有资源的份额。

提示：用彩票来表示份额

彩票（步长）调度的设计中，最强大（且最基本）的机制是彩票。在这些例子中，彩票用于表示一个进程占有 CPU 的份额，但也可以用在更多的地方。比如在虚拟机管理程序的虚存管理的最新研究工作中，Waldspurger 提出了用彩票来表示用户占用操作系统内存份额的方法[W02]。因此，如果你需要通过什么机制来表示所有权比例，这个概念可能就是彩票。

二、機制:

i. 彩票貨幣（ticket currency）。这种方式允许擁有一组彩票的用户以他们喜欢的某種貨幣，將彩票分给自己的不同工作。之後操作系统再自動將這種貨幣兑换为正確的全局彩票。

ii. 彩票轉讓（ticket transfer）。通過轉讓，一个進程可以臨時將自己的彩票交给另一个進程。

iii. 彩票通脹（ticket inflation）。利用通脹，一個進程可以臨時提升或降低自己擁有的彩票數。

三、:隨機:

彩票調度算法不需要針對每個進程紀錄全局狀態，只需要用新進程的票數更新全局的總票數就可以了。因此彩票調度算法能夠更合理地處理新加入的進程。