Operating Systems: Three Easy Pieces

(1-11 章)

資工二 110710540 劉瑞姿

說明: 本筆記的節錄內容主要來源為-- Operating Systems: Three Easy Pieces、操作系統
由於時間不足然後後面的內容越來越複雜，較難閱讀理解並整理，所以只有先做 1~11 章。(某些部分還不是很懂所以就沒有寫，暑假會繼續讀弄清楚 OSTEP)

操作系统介绍

這本書主要分三個簡單的部分是 virtualization、concurrency、persistence。這三個概念可以瞭解作業系統是如何運作。像是它如何決定接下來哪一個程式使用 CPU，如何在虛擬記憶體系統中處理記憶體使用超載，虛擬機器監控器如何工作，如何管理磁片上的資料，還有如何建構在部分節點失敗時仍然能正常工作的分散式系統。

作業系統是負責讓程式運行變得容易，甚至允許你同時運行多個程式，允許程式共用記憶體，讓程式能夠與設備交互。稱為作業系統，因為它們負責確保系統既易於使用又正確高效能地運行。

虛擬化（virtualization）

作業系統將物理資源（如處理器、記憶體或磁片）轉換為更易於使用的虛擬形式。

作業系統會提供幾百個系統調用，讓應用程式調用。由於作業系統提供這

些調用來運行程式、訪問記憶體和設備，甚進行其他相關操作，有時也會說作業系統為應用程序提供了一個標準庫。 再來因為虛擬化讓許多程式運行從而共用 CPU，讓許多程式可以同時訪問自己的指令和資料從而共用記憶體，讓許多程式訪問設備從而共用磁片等，所以作業系統有時被稱為資源操理器。每個 CPU、記憶體和磁片都是系統的資源， 因此作業系統扮演的主要角色就是操理這些資源，以達到高效或公平，或者實 際上考慮其他的目標。

虛擬化 CPU

假設一個電腦只有一個 CPU，虛擬化要做的就是將這個 CPU 虛擬成多個虛擬 CPU 並分給每一個進程使用，因此，每個應用都以為自己在獨佔 CPU，但實際上只有一個 CPU。

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[]){
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
        }
    char *str = argv[1];
    while (1) {
        Spin(1);            //Spin()函数会反复检查时间甚在运行一秒后返回
        printf("%s\n", str);
        }
    return 0;
}
```

```
PS D:\Desktop\110710540\sp\Final>> ./cpu A & ; ./cpu B & ; ./cpu C & ; ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
C
B
D
```

儘管只有一個處理器，但我執行了 4 個程式似乎在同時進行，將單個 CPU 轉換為看似無數個 CPU， 從而讓許多程式同時運行，這就是所謂的虛擬化 CPU。

行程（process）

行程是一個實體。每一個行程都有它自己的地址空間，包括文本區域（text region）、數据區域（data region）和堆栈（stack region）。文本區域存儲處理器執行的代碼；數据區域存儲變量和行程執行期間使用的動態分配的内存；堆棧區域存儲著活動過程調用的指令和本地變量。行程是一個“執行中的程序”。程序是一個有生命的實體，只有處理器賦予程序生命時，它才能成為一個活動的實體，我们稱其為行程。

使用者下達執行程式的命令後，就會產生行程。同一程式可產生多個行程，以允許同時有多位使用者執行同一程式，卻不會相衝突。

行程需要一些資源才能完成工作，像是 CPU 使用時間、記憶體、檔案以及 I/O 裝置，且為依序逐一進行，也就是每個 CPU 核心任何時間內僅能執行一項行程。

希望同時能運行多個程式。一個正常的系統可能會有上百個行程同時在運行。雖然只有少量的物理 CPU 可用，但是作業系統如何提供幾乎有無數個 CPU 可用的假像？作業系統通過虛擬化 CPU 來提供這種假像。通過讓一個行程只運行一個時間片，然後切換到其他行程，作業系統提供了存在多個虛擬 CPU 的假像。

這就是分時處理(time sharing)CPU 技術，允許使用者如願運行多個併發行程。潛在的開銷就是性能損失，因為如果 CPU 必須共用，每個行程的運行就會慢一點。

介紹一下作業系統的所有接口必須包含哪些內容

- 創建（create）：作業系統必須包含一些創建新進程的方法。在 shell 中鍵入命令或按兩下應用程式圖示時，會調用作業系統來創建新行程，運行指定的程式。

- 銷毀（destroy）：由於存在創建行程的介面，因此系統還提供了一個強制銷毀進程的介面。當然，很多行程會在運行完成後自行退出。但是，如果它們不退出，用戶可能希望終止它們，因此停止失控進程的介面非常有用。
- 等待（wait）：有時等待行程停止運行是有用的，因此經常提供某種等待介面。 其他控制：除了殺死或等待行程外，有時還可能有其他控制。例如，大多數作業系統提供某種方法來暫停行程，停止運行一段時間，然後恢復繼續運行。
- 狀態（statu）：通常也有一些介面可以獲得有關進程的狀態資訊，例如運行了多長時間，或者處於什麼狀態。

創建行程(Create Process)

在多道程序環境中，只有行程才能在系統中運行。因此，為使程序能運行，就必須為它創建行程。導致一個行程去創建另一個行程的典型事件，可以有以下四類：

用戶登錄

在分時系統中，用戶在終端鍵入登錄命令後，如果是合法用戶，系統將為該終端建立一個行程，並把它插入到就緒隊列中。

作業調度

在批處理系統中，當作業調度程序按照一定的算法調度到某作業時，便將該作業裝入到內存，為它分配必要的資源，並立即為它創建行程，再插入到就緒隊列中。

提供服務

當運行中的用戶程序提出某種請求後，系統將專門創建一個行程來提供用戶所需要的服務，例如，用戶程序要求進行文件打印，操作系統將為它創建一個打印行程，這樣，不僅可以使打印行程與該用戶行程並發執行，而且還便於計算出為完成打印任務所花費的時間。

應用請求

在上述三種情況中，都是由系統內核為它創建一個新行程，而這一類事件則是基於應用行程的需求，由它創建一個新的行程，以便使新行程以並發的運行方式完成特定任務。

行程狀態(Process State)

一個行程的生命期可以劃分為一組狀態，系統根據 PCB 結構中的狀態值控制行程。行程在生命消失前處於且僅處於三種基本狀態之一。

行程的基本狀態有三種：

1. 就緒狀態(ready)：當行程已分配到除 CPU 以外的所有必要資源後，只要在獲得 CPU，便可立即執行，行程這時的狀態稱為就緒狀態。處於該狀態的行程構成緒列隊。
2. 執行狀態(running)：行程正在處理器上運行的狀態，該行程已獲得必要的資源，也獲得了處理器，用戶程序正在處理器上運行。
3. 等待(阻塞)狀態(blocked)：正在執行的行程由於發生某事件而暫時無法繼續執行時，變放棄處理器而處於暫停狀態，即行程的執行收到阻塞，成為阻塞狀態，也成為等待狀態。

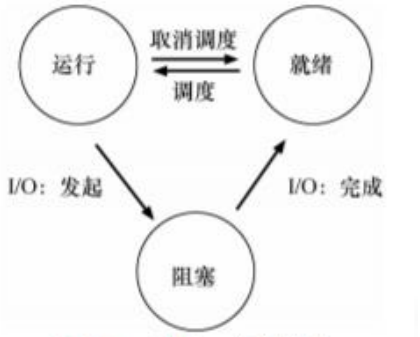


图 4.2 进程：状态转换

根據作業系統的載量，讓行程在就緒狀態和運行狀態之間轉換。從就緒到運行意味著該行程已經被調度。從運行轉移到就緒意味著該行程已經取消調度。一旦行程被阻塞，OS 將保持行程的這種狀態，直到某件事件發生。此時，進程再次轉入就緒狀態也可能立即再次運行。

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

進程 API

這裡採用了 UNIX 系統中的行程創建。UNIX 系統採用了一種非常有趣的創建新行程的方式，就是通過一對系統調用：fork()和 exec()。行程還可以通過系統調用 wait()，來等待其創建的子行程執行完成。

fork()

用於從已存在的行程中創建一個新行程。新進程稱為子行程，而原進程稱為父行程。兩個行程分別獲得其所屬 fork()的返回值，其中在父行程中的返回值是子行程的進程號，而在子行程中返回 0。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

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)
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }
    return 0;
}
```

```
PS D:\Desktop\110710540\sp\Final> ./p1
hello world (pid:26175)
hello, I am child (pid:26176)
hello, I am parent of 26176 (pid:26175)
```

Wait()

主要用於掛起正在運行的行程進入等候狀態，直到有一個子行程終止。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

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;
}
```

```
PS D:\Desktop\110710540\sp\Final> ./p2
hello world (pid:29266)
hello, I am child (pid:29267)
hello, I am parent of 29267 (wc:29267) (pid:29266)
```

exec()

提供了一個在行程中啟動另一個程式執行的方法。它可以根據指定的檔案名或目錄名找到可執行檔，並用它來覆蓋原調用行程的資料段、程式碼片段和堆疊段，在執行完之後，原調用行程的內容除了進程號外，其他全部被新的行程取代了。

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

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());
        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 { // 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;
}
```

```
PS D:\Desktop\110710540\sp\Final> ./p3
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)
```

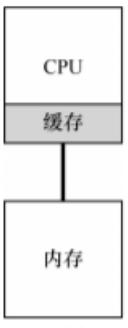
多處理器調度

作業系統應該如何在多 CPU 上調度工作？多處理器調度

多核 CPU 帶來了許多困難。主要困難是典型的應用程式都只使用一個 CPU，增加了更多的 CPU 並沒有讓這類程序運行得更快。為了解決這個問題，不得不重寫這些應用程式，使之能並行執行，也許使用多線程。多線程應用可以將工作分散到多個 CPU 上，因此 CPU 資源越多就運行越快。除了應用程式，作業系統遇到的一個新的問題是多處理器調度。

多處理器架構

多核 CPU 與單 CPU 之間的基本區別。區別的核心在於對硬體緩存的使用，以及多處理器之間共享數據的方式。



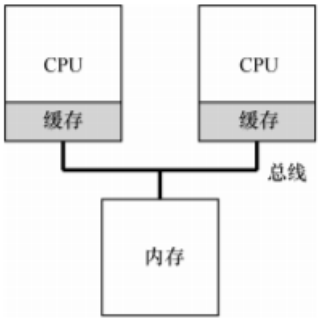
在單 CPU 系統中，存在多級的硬體緩存，說會讓處理器更快地執行程序。緩存是很小但很快的存儲設備，通常擁有內存中最熱的數據的備份。相比之下，內存很大且擁有所有的數據，但訪問速度較慢。通過將經常訪問的數據放在緩存中，系統似乎擁有又大又快的內存。就是程序第一次被讀取數據的時候，數據在內存裡，所以

會花比較多的時間。假如這數據被放在 CPU 緩存中，之後程序再次要用到數據時，CPU 會先到緩存找，因為會在緩存中到，因此取得數據的速度快了不少，這樣程序也會運作的快。

緩存是基於局部性的概念，局部性有兩種:

1. 時間局部性:有數據被訪問，他有可能在沒過多久之後又再被訪問
2. 空間局部性:程序訪問地址是 0 的數據時，有可能在 0 附近的數據會被訪問

多 CPU 的情況下緩存會複雜得多。假設一個運行在 CPU A 上的程序從內存地址 X 讀取數據。由於不在 CPU A 的緩存中，所以系統直接訪問內存，得到值 D。程序然後修改了地址 X 處的值，只是將它的緩存更新為新值 D'。將數據寫回內存比較慢，所以系統會稍後再做。假設這時作業系統中斷了該程序的運行，並將其交給 CPU B，重新讀取地址 X 的數據，由於 CPU B 的緩存中並沒有該數據，所以會直接從內存中讀取，得到了舊值 D，而不是正確的值 D'。上述的問題稱為緩存一致性。



硬體提供了解決這個問題的方法：通過監控內存訪問，硬體可以保證獲得正確的數據，並保證共享內存的唯一性。在基於總線的系統中，一種方式是使用總線窺探。每個緩存都通過監聽連結所有緩存和內存的總線，來發現內存訪問。如果 CPU 發現對它放在緩存中的數據的更新，會作廢本地副本從緩存中移除，或更新它。回寫緩存，如上面提到的，讓事情更複雜由於內存的寫入稍後才會看到。

緩存親和度

一個進程在 CPU A 上運行時，會在 CPU A 的緩存中維護狀態。下次再運行到 CPU A 時，因緩存中的數據所以會運行很快，假如運行在不是 CPU A 上，則會需要重新加載數據所以會變慢。因此多處理器調度應考慮到上面所說的緩存親和度的問題，讓進程盡量在同一 CPU 上。