

FDU 操作系统 0. Introduction

本文参考以下教材：

- Operating Systems: Three Easy Pieces (R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau) Chapter 2
- 操作系统: 三个简单的部分 (王海鹏 译) 第 2 章

欢迎批评指正!

0.0 Preface

本课程围绕 3 个主题展开讲解：

虚拟化 (virtualization)、**并发** (concurrency) 和**持久性** (persistence).

每个主题都在若干章节中加以阐释 (提出了一个特定的问题，然后展示解决该问题的方法)

- 主页: [Operating Systems: Three Easy Pieces](#)
- 代码: [GitHub: Code from various chapters in OSTEP](#)
- 实验: [GitHub: Projects for an undergraduate OS course](#)

简单来说，程序运行的过程就是不断执行指令的过程。

处理器从内存中**取出** (fetch) 一条指令，

对其进行**解码** (decode) (弄清楚这是哪条指令)，

然后**执行** (execute) 它 (做它应该做的事情，例如两个数相加、访问内存、检查条件、跳转到函数等)。

完成这条指令后，处理器继续执行下一条指令，依此类推，直到程序最终完成。

上述过程就是 Von Neumann 计算机架构的**指令周期** (instruction cycle)。

但实际上，现代处理器为了让程序运行得更快，

做了很多 "可怕" 的事情，例如一次执行 (甚至乱序执行) 多条指令。

但本课程里我们只关心程序运行过程的简单模型：指令有序地逐条执行。

0.1 虚拟化

本课程的重点是学习如何使系统易于使用。

实际上，有一类软件负责让程序运行变得容易 (甚至允许你同时运行多个程序)，

允许程序共享内存，让程序能够与设备交互，以及其他类似的有趣的工作。

这些软件称为**操作系统** (Operating System, OS)，

因为它们负责确保系统既易于使用又正确高效地运行。

要做到这一点，操作系统主要利用一种通用的技术——**虚拟化** (virtualization)。

也就是说，操作系统将**物理资源** (如处理器、内存或磁盘) 转换为更强大且更易于使用的虚拟形式。

因此操作系统又称为**虚拟机** (virtual machine)。

典型的操作系统会提供几百个**系统调用** (system call)，供应用程序调用，

用于运行程序、访问内存和设备，以及进行其他相关操作。

可以说，操作系统为应用程序提供了一个**标准库** (standard library)。

因为虚拟化让许多程序可以同时运行 (从而共享 CPU)，

同时访问各自的指令和数据 (从而共享内存)，

同时访问设备 (从而共享磁盘等外设)，

所以操作系统又被称为**资源管理器** (resource manager)，

而每个 CPU、内存和磁盘都是系统的**资源** (resource)。

因此操作系统扮演的主要角色就是管理这些资源，以达到公平、高效或其他基于实际需求的目标。

0.1.1 虚拟化 CPU

以下代码用于定时打印用户命令行输入的字符串：

```
// code for cpu.c
#include <stdio.h>    // 标准输入输出头文件，用于 printf 函数
#include <stdlib.h>   // 标准库头文件，用于 exit 函数
#include <unistd.h>   // 包含 sleep 函数
#include <assert.h>   // 断言头文件，通常用于调试
#include "common.h"  // 包含用户自定义函数或变量的头文件

// 主函数，程序入口点
int main(int argc, char *argv[])
{
    // 如果程序参数个数不等于 2，则输出用法提示信息并退出
    if (argc != 2)
    {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }

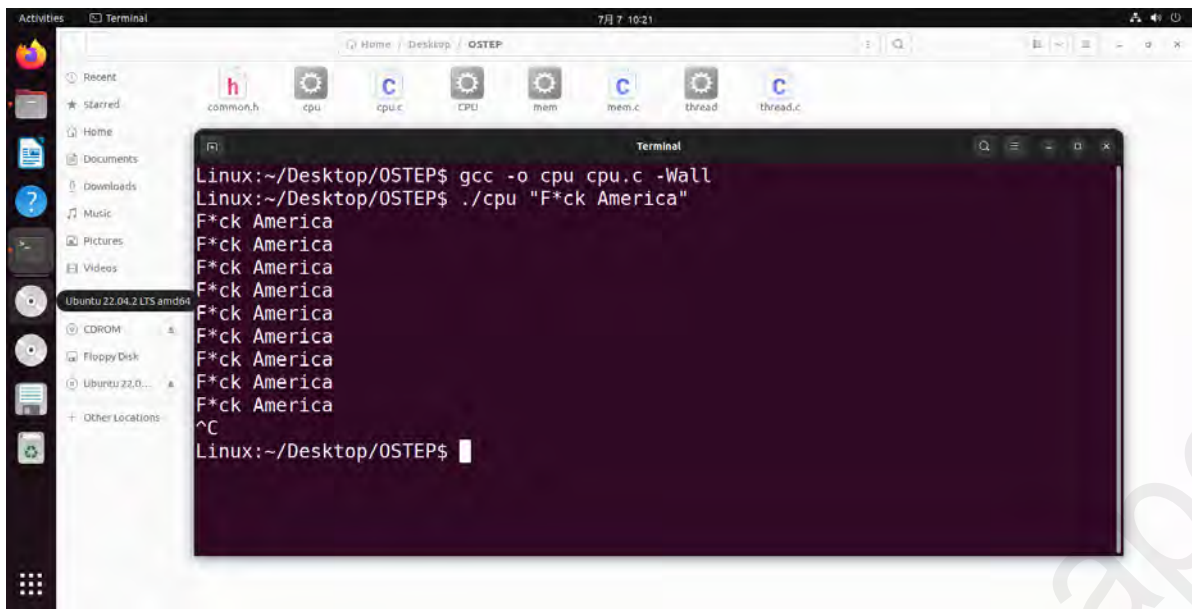
    // 获取程序参数中的字符串
    char *str = argv[1];

    // 无限循环，每隔 1 秒打印一次字符串
    while (1)
    {
        sleep(1);           // 休眠 1 秒
        printf("%s\n", str); // 打印字符串
    }

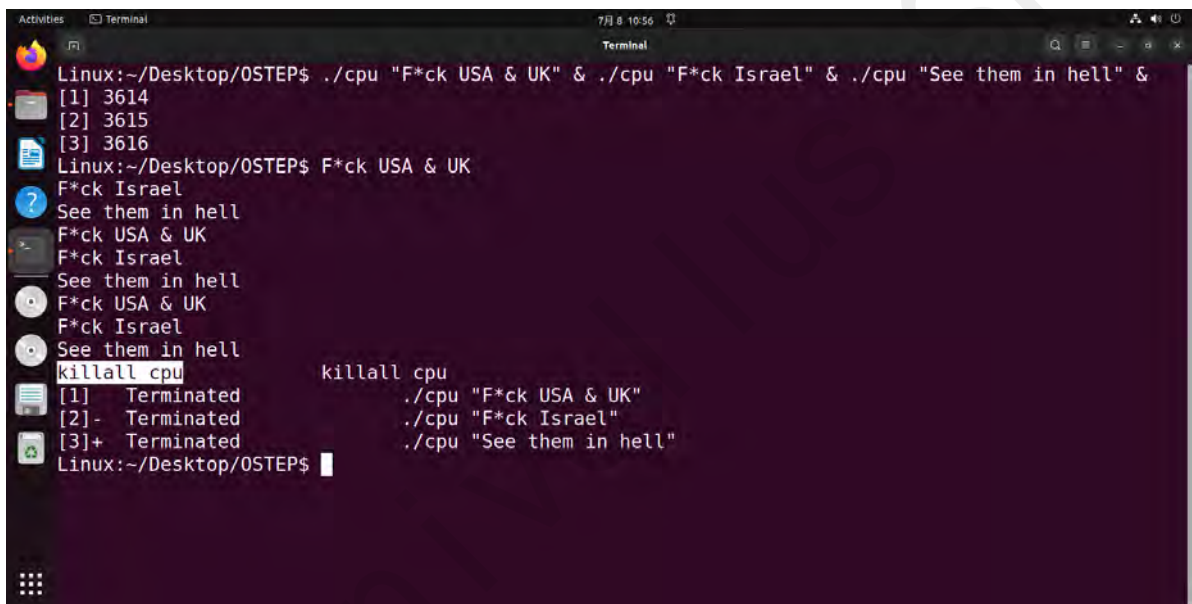
    return 0; // 返回 0 表示程序成功结束
}
```

假设我们将这个文件保存为 `cpu.c`，
并在一个单 CPU 的 Linux 系统上编译和运行它，
则我们将看到以下内容：

(除非按下 "Ctrl-C" (这在基于 UNIX 的系统上将终止在前台运行的程序)，否则程序将永远执行下去。)



现在让我们来做同样的事情，但这一次，我们将运行同一程序的不同实例。



现在事情开始变得有趣了。

尽管我们只有一个 CPU，但这 3 个程序似乎在同时运行！

操作系统负责提供这种假象——即系统拥有非常多的虚拟 CPU，从而“同时”运行许多程序。

这就是所谓的**虚拟化 CPU** (virtualizing the CPU)。

0.1.2 虚拟化内存

现代计算机的**物理内存** (physical memory) 就是一个字节数组。

要读取 (read) 内存，必须指定一个地址 (address)；

要写入 (write) 内存，必须同时指定地址和要写入该地址的数据。

程序的指令和数据都在内存中，因此在执行时需要不断访问内存。

让我们来看一个程序 mem.c，它通过调用 malloc() 来分配内存：

```
// code for mem.c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
```

```

#include "common.h"

int main(int argc, char *argv[])
{
    // It dynamically allocates memory for an integer
    // and assigns the pointer p to the allocated memory.
    int *p = malloc(sizeof(int));

    // An assertion is used to check whether the allocation was successful,
    // and if not, the program will terminate with an error message.
    assert(p != NULL);

    // It prints the process ID (PID) and the memory address pointed to by p.
    printf("(%) address pointed to by p: %p\n", getpid(), p);

    // It initializes the integer pointed to by p with the value 0.
    *p = 0;

    // It enters an infinite loop,
    // where it increments the value of the integer pointed to by p every second
    // and prints the updated value along with the PID.
    while (1)
    {
        sleep(1);
        *p = *p + 1;
        printf("(%) p: %d\n", getpid(), *p);
    }
    return 0;
}

```

其中用户头文件 common.h 的内容如下:

```

#ifndef __COMMON_H
#define __COMMON_H

// 基本常用头文件
#include <time.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <strings.h>
#include <stdbool.h>

// 全局错误码
#include <errno.h>

// 文件IO操作
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>

// 数学库
#include <math.h>

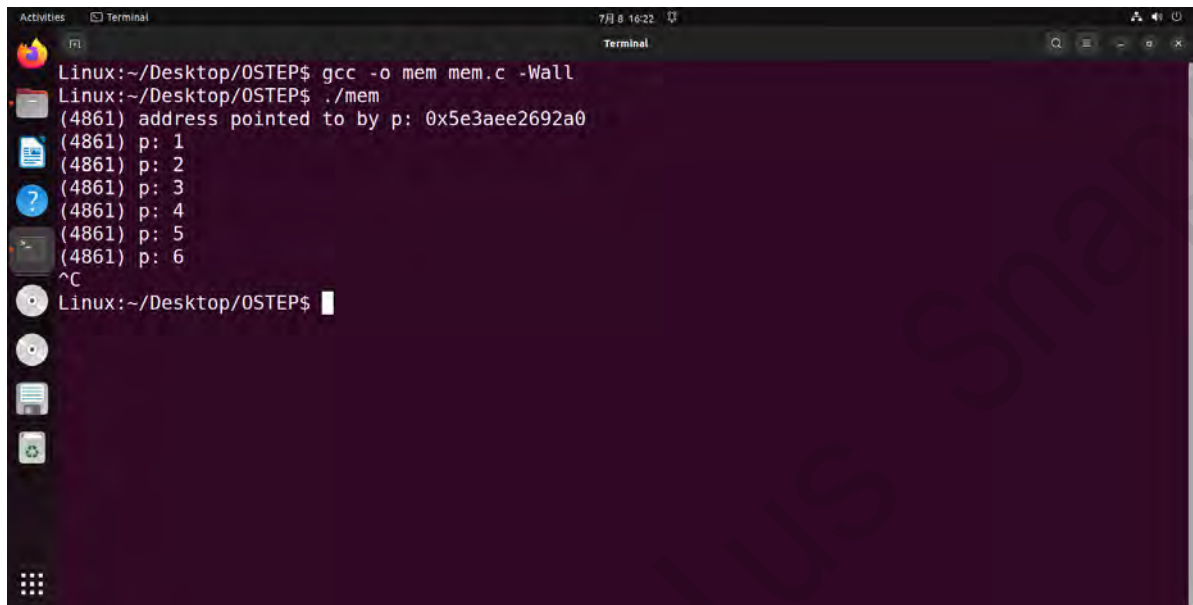
// 目录
#include <dirent.h>

```

```
#endif
```

该程序的输出如下：

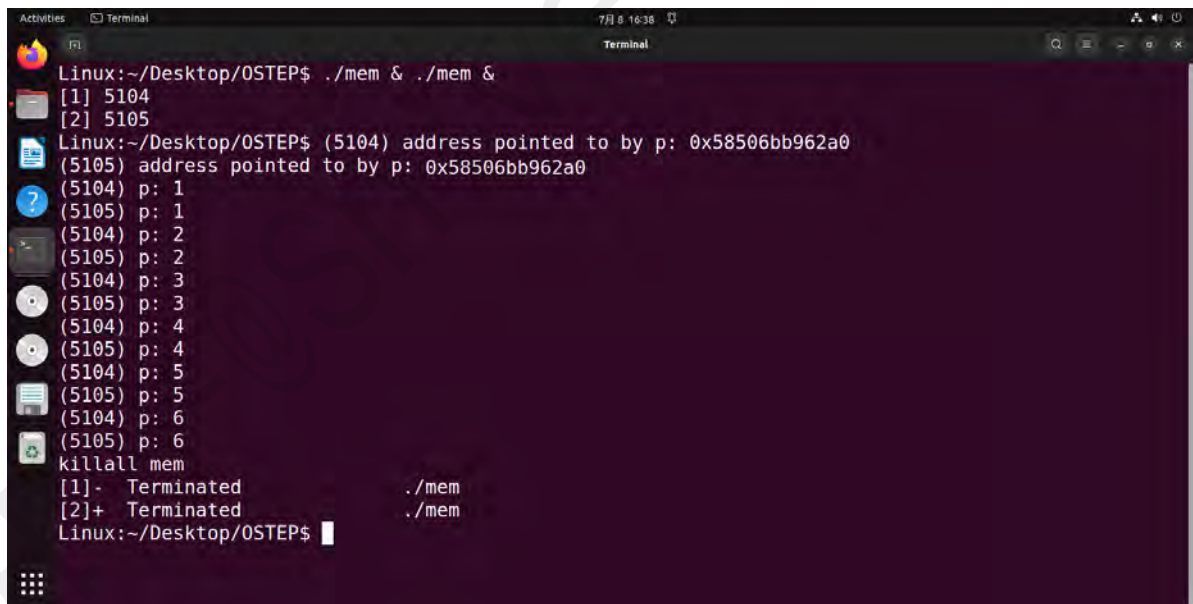
- 首先它分配了一个 `int` 整型数据的内存 (指针为 `p`);
- 然后它打印出指针 `p` 指向的内存地址, 并将数值 0 存入该内存地址;
- 最后程序循环, 每经过一秒递增并打印指针 `p` 指向的内存地址处所保存的值.
- 在每个打印语句中, 它还会打印出所谓的 "正在运行" 程序的进程标识符 (PID).



```
Linux:~/Desktop/OSTEP$ gcc -o mem mem.c -Wall
Linux:~/Desktop/OSTEP$ ./mem
(4861) address pointed to by p: 0x5e3aee2692a0
(4861) p: 1
(4861) p: 2
(4861) p: 3
(4861) p: 4
(4861) p: 5
(4861) p: 6
^C
Linux:~/Desktop/OSTEP$
```

现在我们再次运行同一程序的多个实例, 看看会发生什么:

(要复现以下示例, 需要禁用地址空间随机化: `sudo sysctl -w kernel.randomize_va_space=0`)



```
Linux:~/Desktop/OSTEP$ ./mem & ./mem &
[1] 5104
[2] 5105
Linux:~/Desktop/OSTEP$ (5104) address pointed to by p: 0x58506bb962a0
(5105) address pointed to by p: 0x58506bb962a0
(5104) p: 1
(5105) p: 1
(5104) p: 2
(5105) p: 2
(5104) p: 3
(5105) p: 3
(5104) p: 4
(5105) p: 4
(5104) p: 5
(5105) p: 5
(5104) p: 6
(5105) p: 6
killall mem
[1]- Terminated ./mem
[2]+ Terminated ./mem
Linux:~/Desktop/OSTEP$
```

我们从示例中看到:

每个正在运行的程序都在相同的地址 (58506bb962a0) 处分配了内存,

但每个似乎都独立更新了 58506bb962a0 处的值!

就好像每个正在运行的程序都有自己的私有内存 (或者说, 似乎完全拥有物理内存), 而不是与其他正在运行的程序共享相同的物理内存.

实际上, 这正是操作系统虚拟化内存 (virtualizing memory) 时发生的情况.

每个进程访问自己的私有虚拟地址空间 (virtual address space) (有时称为地址空间, address space), 操作系统以某种方式映射到物理内存上.

一个正在运行的程序中的内存引用不会影响其他进程 (或操作系统本身) 的地址空间.

0.2 并发

本课程的另一个主题是**并发** (concurrency),
即在同一内存空间中处理并发执行的线程时出现的一系列问题。
我们来看一个多线程程序 thread.c 的例子:

```
// code for thread.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
// The header file pthread.h includes 'pthread_t', 'pthread_create' and
'pthread_join'
// pthread_t is a data type in C that is used to represent a POSIX thread
identifier,
// which is a standard for creating and managing threads in a multi-threaded
program.
#include "common.h"

volatile int counter = 0;
// The 'volatile' keyword is used to indicate to the compiler
// that the variable may be changed by external factors not known to the
compiler,
// so it should not optimize away reads and writes to it.

int loops;
// Declare a global variable 'loops' without initialization,
// which will be set based on the command-line argument.

// Define a function 'worker' that will be executed by the threads.
void *worker(void *arg)
{
    int i;
    for (i = 0; i < loops; i++)
        counter++;
    return NULL;
}

int main(int argc, char *argv[])
{
    // Check if there is exactly one command-line argument provided
    if (argc != 2)
    {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }

    // Parse the command-line argument as an integer
    // and store it in the 'loops' variable.
    loops = atoi(argv[1]);

    // Create two pthreads ('p1' and 'p2')
    // and execute the 'worker' function in each of them.
    pthread_t p1, p2;

    // Print the initial value of 'counter'
```



```

printf("Initial value : %d\n", counter);

pthread_create(&p1, NULL, worker, NULL);
pthread_create(&p2, NULL, worker, NULL);

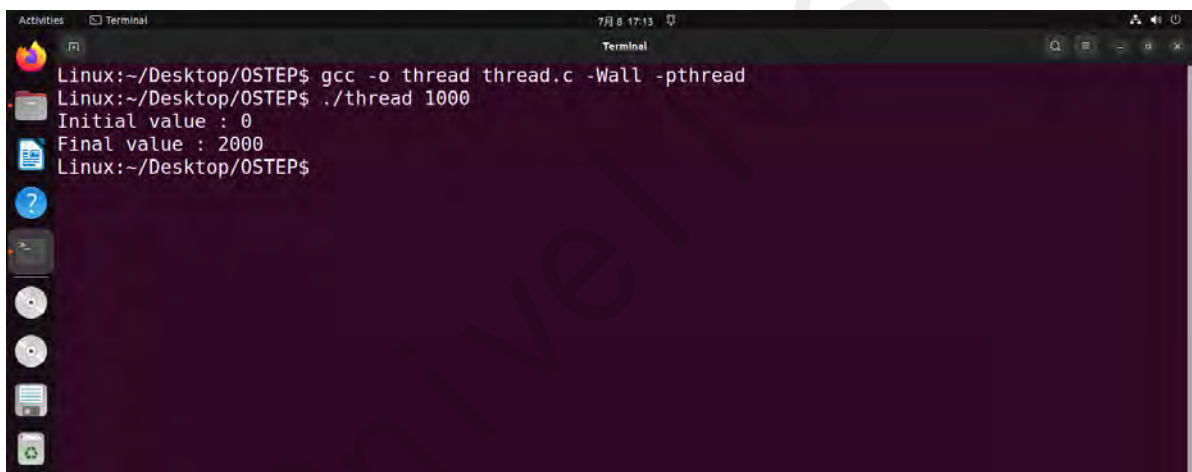
// Wait for both threads to complete using 'pthread_join'.
pthread_join(p1, NULL);
pthread_join(p2, NULL);

// Print the final value of 'counter'.
printf("Final value : %d\n", counter);
return 0;
}

```

主函数利用 `pthread_create()` 创建了两个**线程** (thread).
 我们可以将线程看作与其他函数在同一内存空间中运行的函数,
 并且每次都有多个线程处于活动状态.
 在这个例子中, 每个线程开始在一个名为 `worker()` 的函数中运行,
 在该函数中, 它只是递增一个计数器, 循环 `loops` 次.
 (`loops` 的值决定了两个 `worker` 各自在循环中递增共享计数器的次数)

我们将变量 `loops` 设为 1000, 得到的输出结果如下:

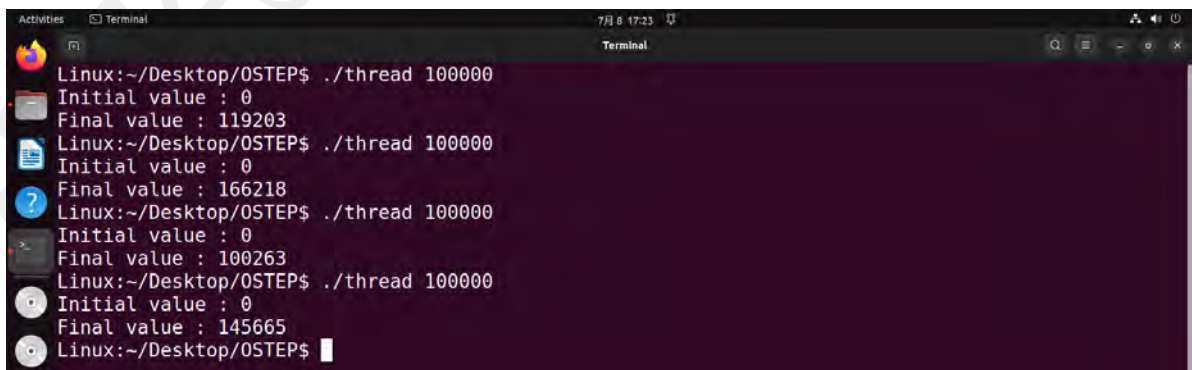


```

Linux:~/Desktop/OSTEP$ gcc -o thread thread.c -Wall -pthread
Linux:~/Desktop/OSTEP$ ./thread 1000
Initial value : 0
Final value : 2000
Linux:~/Desktop/OSTEP$

```

我们很容易想当然地认为, 当 `loops` 的输入值设为 N 时, 程序的最终输出就是 $2N$.
 但事实证明, 事情并不是那么简单.
 让我们将 `loops` 设置为更大的值, 看看会发生什么:



```

Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 119203
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 166218
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 100263
Linux:~/Desktop/OSTEP$ ./thread 100000
Initial value : 0
Final value : 145665
Linux:~/Desktop/OSTEP$

```

奇怪的是, 当我们将 `loops` 的值设为 100000 时,
 并不一定能得到预期的输出 200000, 而且每次运行的输出也不一定相同!

事实证明, 上述奇怪的结果与指令如何执行有关.
 程序每次只执行一条指令,
 但上述程序的关键部分 (`worker()` 中递增共享计数器的部分) 需要 3 条指令:

- 一条将计数器的值从内存加载到寄存器;
- 一条将其递增;
- 一条将其保存回内存;

而这 3 条指令并不是以**原子方式** (atomically) 执行的 (即不是作为一个不可拆分的单元执行的)。

0.3 持久性

本课程的第三个主题是**持久性** (persistence)。

现代计算机的主存通常由 DRAM 芯片构成 (Dynamic Random Access Memory, 动态随机存取存储器)。

其数据是易失 (volatile) 的, 即若遭遇断电或系统崩溃, 则主存中所有数据都会丢失。

因此我们需要硬件和软件来持久地 (persistently) 存储数据。

所需的硬件即输入/输出 (Input/Output, I/O) 设备,

例如磁盘驱动器 (hard drive) 或更先进地, **固态硬盘** (Solid-State Drive, SSD)。

而操作系统中管理磁盘的软件通常称为**文件系统** (file system),

它负责以可靠和高效的方式, 将用户创建的所有文件存储在系统的磁盘上, 并在不同的进程之间共享。

下面的代码用于创建包含字符串 "hello world" 的文件 (/tmp/file)

```
#include <stdio.h>      // for standard input/output functions.
#include <unistd.h>      // for POSIX system calls and constants.
#include <assert.h>      // for assertion checks.
#include <fcntl.h>       // for file control options.
#include <sys/types.h>   // for data types used in system calls.

// main function takes command-line arguments (argc and argv)
int main(int argc, char *argv[])
{
    // Opens a file named "/tmp/file" (or creates it if it doesn't exist)
    // for writing (O_WRONLY) with permissions set to read, write, and execute
    int fd = open("file", O_WRONLY|O_CREAT|O_TRUNC);

    // Checks if the file descriptor (fd) is valid (greater than -1)
    // using the assert macro. If not, the program will terminate.
    assert(fd > -1);

    // Writes the string "hello world\n" to the opened file using the write
    // system call.
    // It writes 13 bytes (the length of the string) to the file.
    int rc = write(fd, "hello world\n", 13);

    // Checks if the return value of the write function (rc) is equal to 13 (the
    // number of bytes written).
    // If not, it means the write operation didn't complete as expected, the
    // program will terminate.
    assert(rc == 13);

    // Closes the file using the close system call to release the associated
    // resources.
    close(fd);
    return 0;
}
```


输出结果:

```
Linux:~/Desktop/OSTEP$ gcc -o io io.c -Wall
Linux:~/Desktop/OSTEP$ ./io
Linux:~/Desktop/OSTEP$
```

在当前目录下得到一个名为 file 的文件，修改文件名为 file.txt 后打开内容如下:

```
hello world
\00
```

为了完成这个任务，该程序向操作系统发出 3 个调用。

第一个是对 open() 的调用，它打开文件并创建它。

第二个是 write()，将字符串 hello world\n 写入文件。

第三个是 close()，只是简单地关闭文件，从而表明程序不会再向它写入更多的数据。

这些系统调用被转到称为**文件系统** (file system) 的操作系统部分，然后该系统处理这些请求。

文件系统必须做很多工作:

首先确定新数据将存储在磁盘上的哪个位置，然后在文件系统所维护的各种结构中对其进行记录。

这样做需要向底层存储设备发出 I/O 请求，以读取现有结构或更新 (写入) 它们。

幸运的是，操作系统提供了一种通过系统调用来访问设备的标准和简单的方法。

因此 OS 有时被视为**标准库** (standard library)

关于如何访问设备还有更多细节。

出于性能方面的原因，大多数文件系统首先会延迟这些写操作一段时间，希望将其批量分组为较大的组。

为了处理写入期间系统崩溃的问题，

大多数文件系统都包含某种复杂的写入协议，例如日志 (journaling) 或写时复制 (copy-on-write)，

仔细排序写入磁盘的操作，以确保如果在写入期间发生故障，系统可以在恢复到合理的状态。

为了使不同的通用操作更高效，文件系统采用了许多不同的数据结构和访问方法，从简单的列表到复杂的 B 树。

0.4 目标

现在我们已经了解了操作系统实际上做了什么:

它管理 CPU、内存和磁盘等物理资源，并对它们进行虚拟化 (virtualize)。

它处理与并发 (concurrency) 有关的麻烦且棘手的问题。

它持久地 (persistently) 存储文件，从而使它们长期安全。

鉴于我们希望建立这样一个系统，所以要有一些目标，以帮助我们集中设计和实现，甚至在必要时进行折中。

找到合适的折中是建立系统的关键。

一个最基本的目标，是建立一些**抽象** (abstraction)，让系统方便和易于使用。

抽象使得编写一个大型程序成为可能，将其划分为小而且容易理解的部分。

另一个目标是在应用程序之间以及在 OS 和应用程序之间提供**保护** (protection)

保护是操作系统基本原理之一的核心，即**隔离** (isolation)，让进程彼此隔离是保护的关键。

The End

