操作系统

目录

第 1 章 操作系统概述

1.1 什么是操作系统？

操作系统(Operating System)

现代的计算机离不开操作系统(OS)，操作系统是最基本也是最为重要的基础性系统软件。

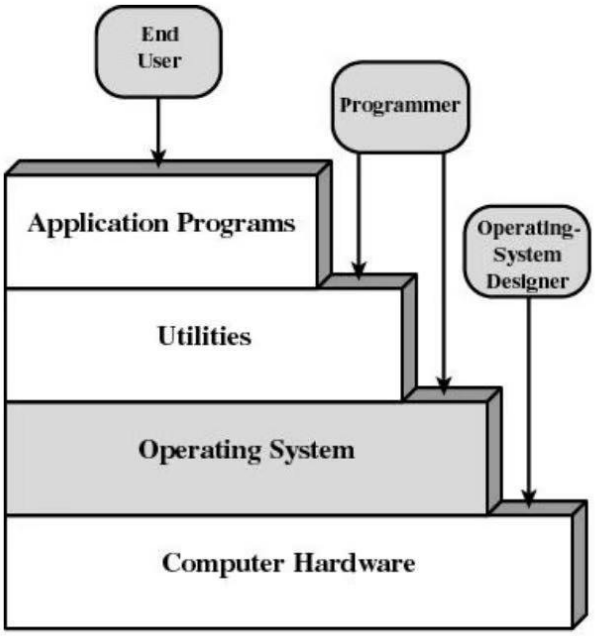
在计算机结构中，**最底层**的是计算机**硬件**，硬件之上就是操作系统。**操作系统是运行在硬件上的第一层软件**，它**扩展了硬件的功能**。但是光有操作系统是不满足的还需要增加新的一些工具，例如开发工具和开发平台等，所以操作系统需要支持用户的程序开发。

计算机结构中用户主要分为三大类：

- 面向硬件的用户被称为**操作系统设计者**，需要熟知硬件的运行机制，以及如何在上面进行系统开发。
- 在操作系统纸上进行程序设计的用户被称为**程序员**，程序员有两种类型，一种是基于操作系统的程序设计，另一种是基于开发平台的

程序设计。

- 终端用户(end user)直接使用应用程序。



操作系统观点

现代操作系统主要有四种基本观点：

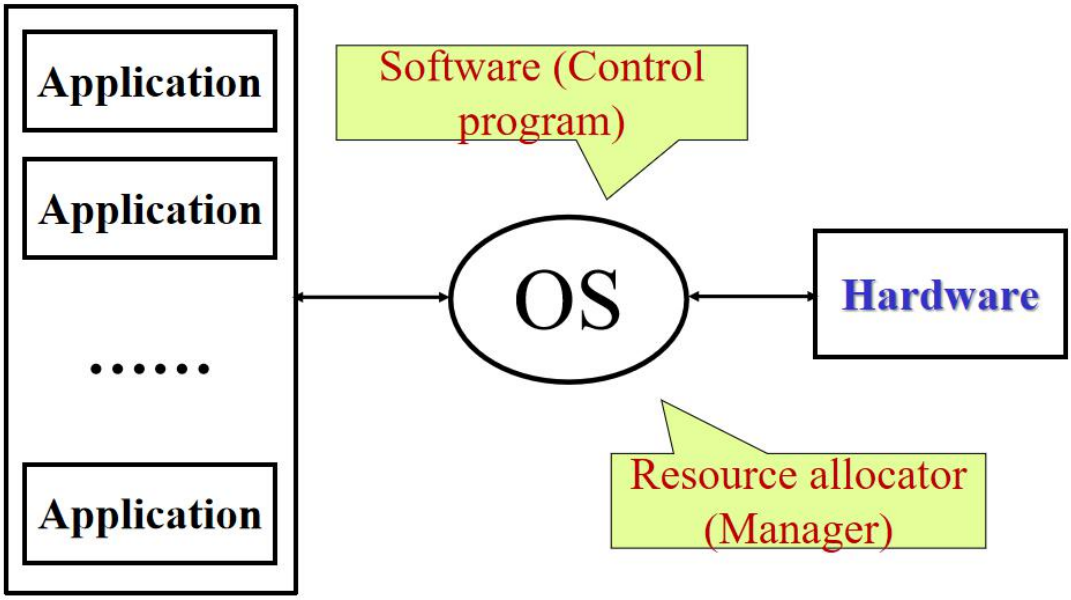
1. 普通用户：OS 是计算机用户使用计算机系统的接口，它为计算机用户提供而方便的工作环境。

iPhone 6 - iOS 8

2. **程序员**：OS 是建立在计算机硬件平台上的**虚拟机器**，它为应用软件提供了许多比计算机硬件功能更强或计算机硬件所没有的功能。

3. **OS 开发者**：OS 是计算机系统中各类**资源的管理者**，它负责分配、回收以及控制操作系统中的软硬件资源（包括 CPU、内存、IO 设备、文件、网络）。

4. **OS 开发者**：OS 是计算机系统**工作流程的组织者**，它负责协调在系统中运行的各个应用软件的运行次序。



1.2 OS 功能性与非功能性需求

功能性需求

OS 的功能性需求主要有两大类：

1. **用户命令**：计算机用户需要使用用户命令进行操作，由 OS 实现的所有用户命令所构成的集合被称为 OS 的**接口(interface)**。

2. **系统调用**(system call)：应用软件需要使用系统调用来实现 OS 所提供的服务，由 OS 实现的所有系统调用所构成的集合被称为**应用编程接口(Application Programming Interface, API)**。

非功能性需求

OS 的非功能性需求主要包括：

1. **性能/效率(performance/efficiency)**：**最大化系统吞吐量(throughput)**、**最小化响应时间(response time)**，在**分时系统**情况下需要满足**尽量多**的用户。

2. **公平性(fairness)**：OS 中的算法不能偏向于某些特定类型的进程，但是在某些时候，如实时系统需要及时响应外部事件。

3. **可靠性(reliability)**

4. **安全性(security)**

5. **可伸缩性(scalability)**：不同预算可购买不同配置，如个人版和企业版。

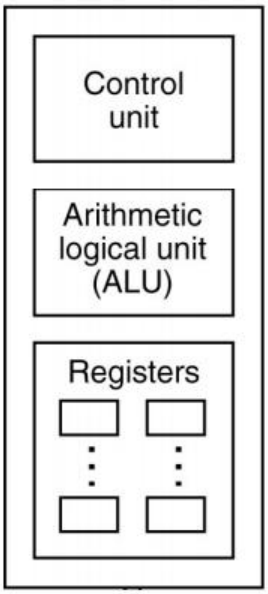
6. **可扩展性(extensibility)**：OS 能够适应新的外部设备的增长。

7. **可移植性(portability)**：不能仅限于某一个平台或某一个厂家生产的硬件设备。

1.3 CPU

中央处理器(Central Process Unit, CPU)

CPU 作为计算机系统的**运算和控制核心**，是信息处理、程序运行的最终执行单元。



Python 中可以通过 multiprocessing 模块获取计算机 CPU 的内核数量。

范例：获取当前 CPU 内核数量

```
import multiprocessing  # 导入多进程模块
# 获取 CPU 的可用数量
print("CPU 内核数量：%d" % multiprocessing.cpu_count())
```

运行结果	CPU 内核数量：12

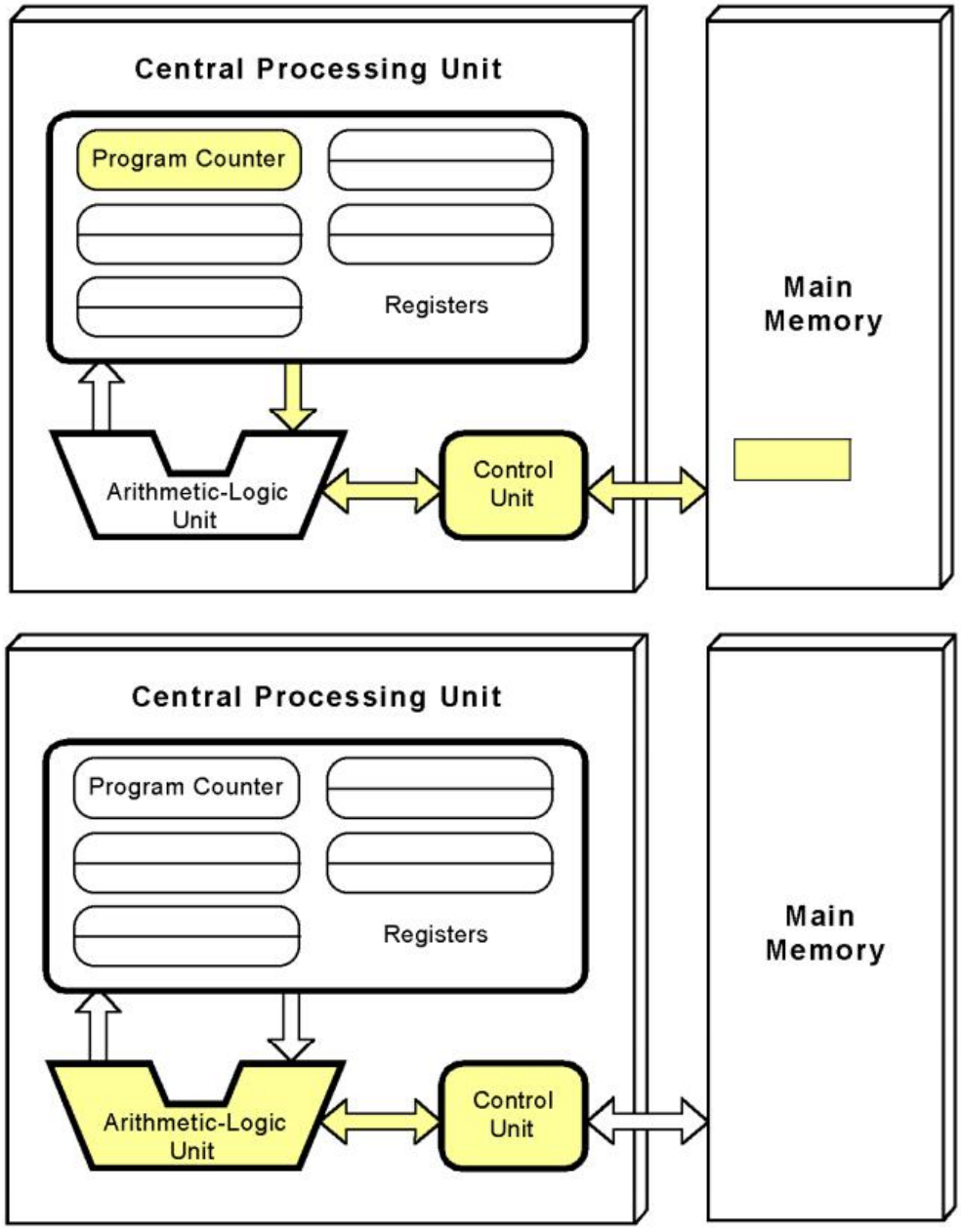
CPU 的核心部分有：

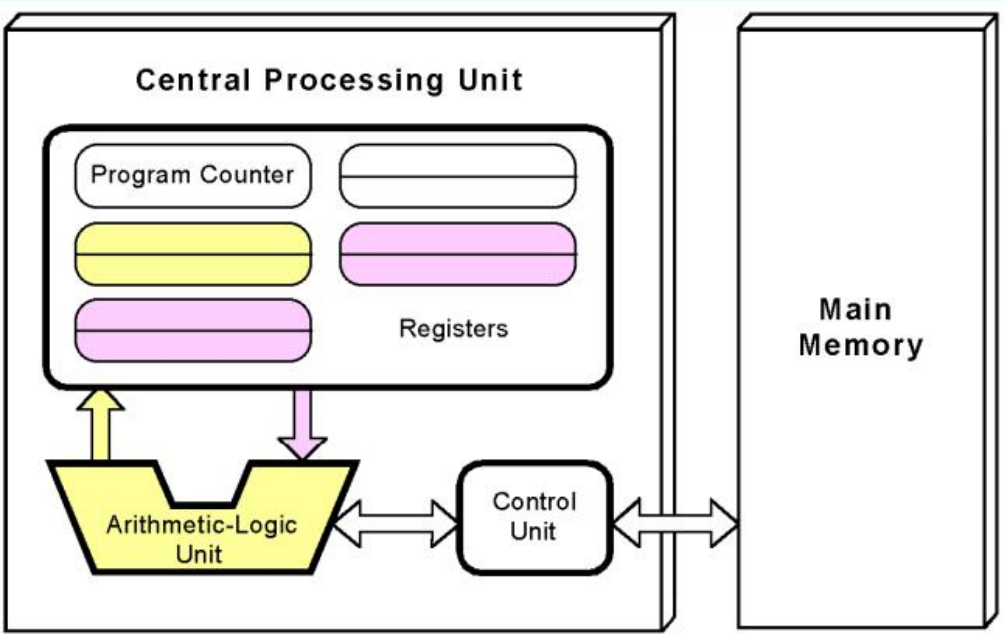
- **控制单元(Control Unit, CU)**：控制单元是 CPU 的子部件，它管理着计算机中所有在这一区域执行的操作。它负责从计算机、指令和数据中获取各种输入，并告诉处理器如何处理它们。
- **算术逻辑单元(Arithmetic and Logic Unit , ALU)**：实现多组算术运算

和逻辑运算的组合逻辑电路。

- **寄存器(Register)**：寄存器是有限存贮容量的高速存贮部件，它们可用来暂存指令、数据和地址。

大多数现代处理器的工作原理是“**取指令-译码-执行**”(Fetch-Decode-Execute Cycle)，也被称为**冯·诺依曼**架构(Von Neumann Architecture)。





Python 中 psutil 模块也提供了**系统硬件**的内容获取。

范例：获取 CPU 信息

```
import psutil

def main():
    # CPU 信息
    print("【CPU】物理数量：%d" % psutil.cpu_count(logical=False))
    print("【CPU】逻辑数量：%d" % psutil.cpu_count(logical=True))
    print("【CPU】用户用时：%f" % psutil.cpu_times().user)
    print("【CPU】系统用时：%f" % psutil.cpu_times().system)
    print("【CPU】空闲时间：%f" % psutil.cpu_times().idle)

if __name__ == "__main__":
    main()
```

运行结果

```
【CPU】物理数量：6
【CPU】逻辑数量：12
【CPU】用户用时：1970.000000
【CPU】系统用时：1838.546875
【CPU】空闲时间：150477.968750
```

1.4 中断

中断(Interrupt)

中断是指计算机运行过程中，出现某些**意外情况**需主机干预时，机器能自动停止正在运行的程序并转入处理新情况的程序，处理完毕后又返回原被暂停的程序继续运行。

中断包括鼠标移动、鼠标点击、键盘按键、打印机打印等。

现代计算机中采用中断系统的主要目的是：

1. **提高计算机系统效率**：计算机系统中处理机的工作速度远高于外围设备的工作速度，通过中断可以协调它们之间的工作。当外围设备需要与处理机交换信息时，由外围设备向处理机发出中断请求，处理机及时响应并作相应处理。

2. **维持系统可靠正常工作**：程序员不能直接干预和操纵机器，必须通过中断系统向操作系统发出请求，由操作系统来实现人为干预。在程序运行过程中，如出现越界访问，有可能引起程序混乱或相互破坏信息。为避免这类事件的发生，由存储管理部件进行监测，一旦发生越界访问，向处理机发出中断请求，处理机立即采取保护措施。

3. **满足实时处理要求**：在实时系统中，各种监测和控制装置随机地向处理机发出中断请求，处理机随时响应并进行处理。

4. **提供故障现场处理手段**：处理机中设有各种故障检测和错误诊断的部件，一旦发现故障或错误，立即发出中断请求，进行故障现场记录和隔离，为进一步处理提供必要的依据。

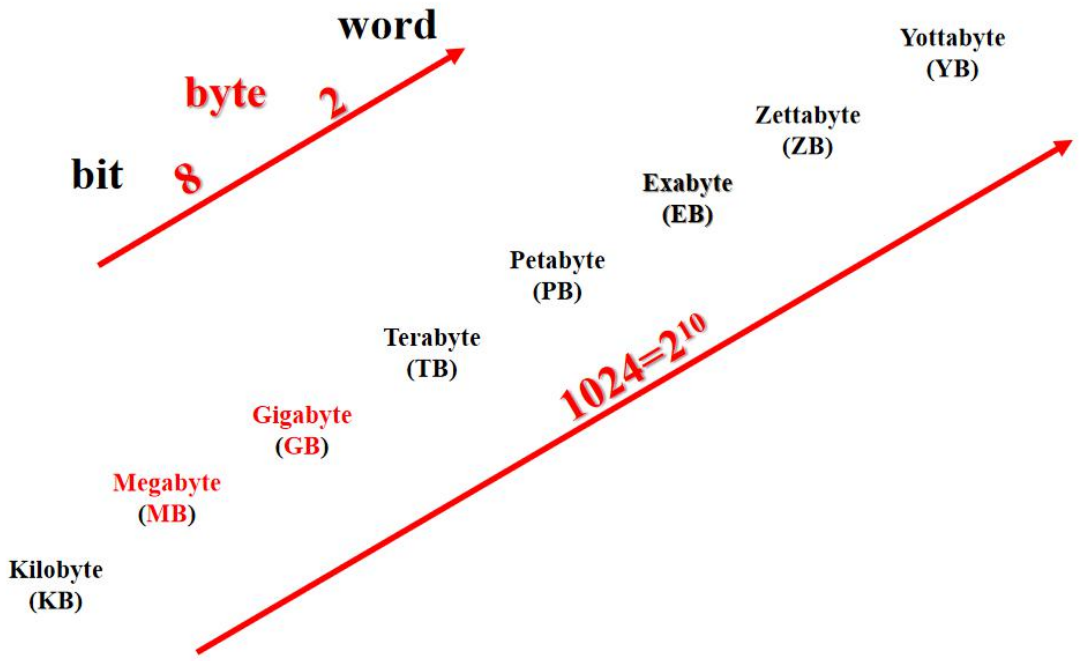
1.5 存储器

存储单位

计算机存储单位一般使用 bit (b)、B、KB、MB、GB、TB、PB、EB、ZB、YB、BB、NB、DB 等符号来表示。

位(bit)是计算机最小的存储单位，只存放一位二进制数。

一个字节(byte)为 8 位二进制数，保存一个英文字母需要一个字节，但保存一个中文需要两个字节字(word)。

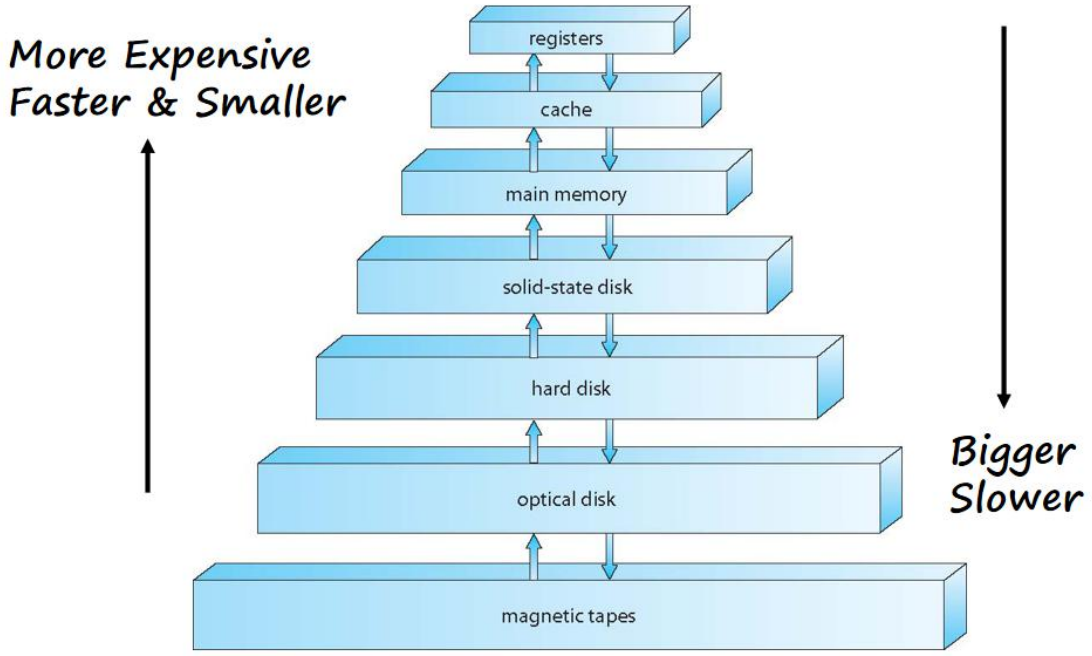


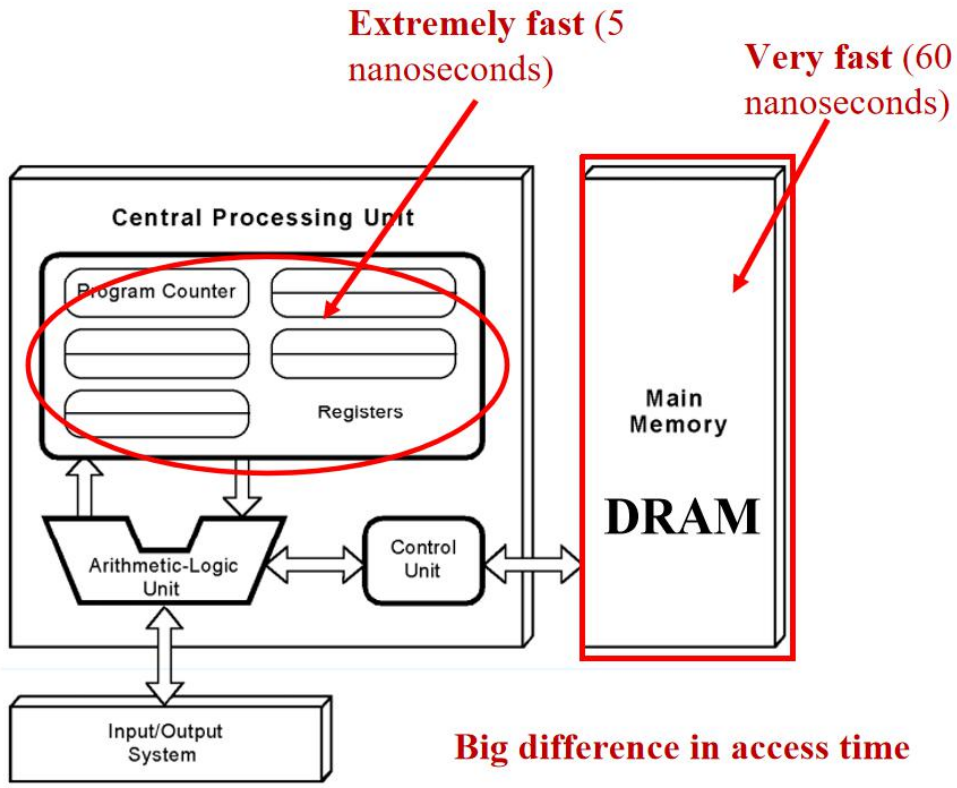
有些硬盘厂商以 1000 进制为单位，这并不是故意弄虚作假，而是为了设计和制造的方便，直接以 1000 计算更方便。

存储设备

存储设备是用于储存信息的设备，通常是将信息数字化后再以利用电、磁或

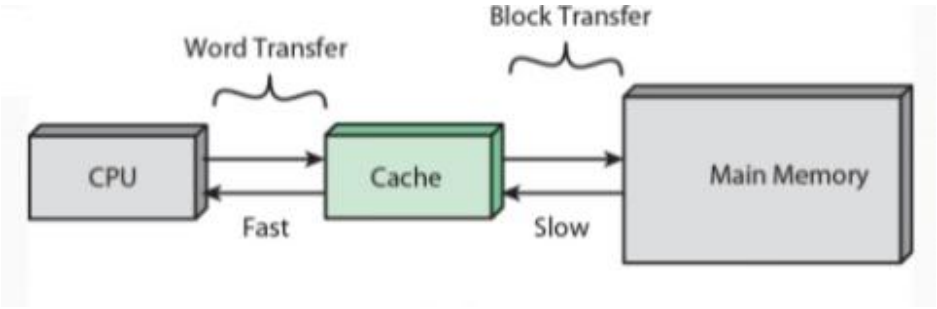
光学等方式的媒体加以存储。





内存缓存(Memory Cache)

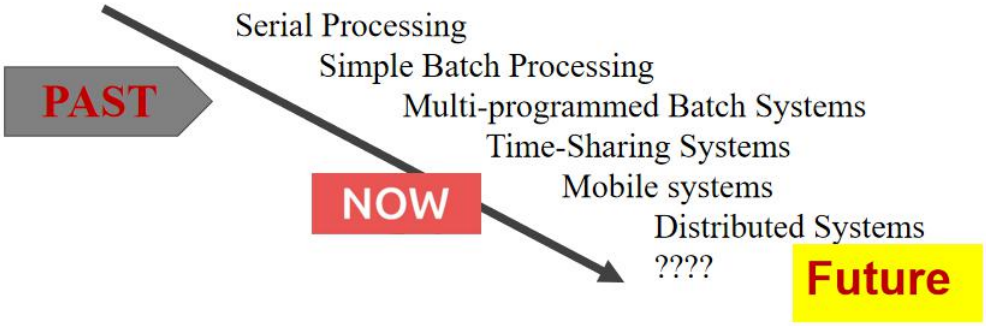
CPU 缓存是位于 CPU 与内存之间的临时存储器，它的容量比内存小的多但是交换速率却比内存要快得多。缓存的出现主要是为了解决 CPU 运算速率与内存读写速率不匹配的矛盾，因为 CPU 运算速率要比内存读写速率快很多，这样会使 CPU 花费很长时间等待数据到来或把数据写入内存。在缓存中的数据是内存中的一小部分，但这一小部分是短时间内 CPU 即将访问的，当 CPU 调用大量数据时，就可避开内存直接从缓存中调用，从而**加快读取速率**。



1.6 操作系统演变

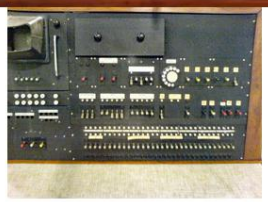
操作系统演变

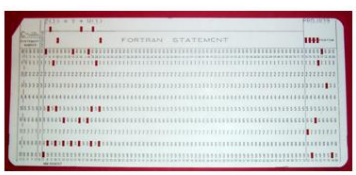
操作系统一直在不断地更新版本，版本升级就是操作系统的改变。操作系统改变的原因包括修复漏洞、支持新的服务、硬件升级、提升性能等。



串行处理(Serial Processing)

这个时期的计算机体积非常大，因为没有操作系统，所以操作的时候需要通过扳动按钮，最终通过灯来进行显示。这种计算机必须全部由专家操作，一般用户是用不了这台计算机的。





这就导致了两个主要的问题：

1. **调度低效**：机器需要等待人扳动按钮，浪费时间。

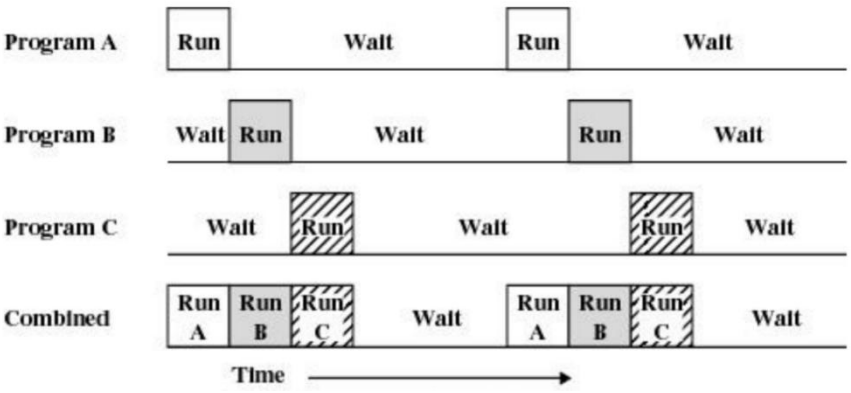
2. **启动时间慢**：没有自动化，需要手工装入编译器、源程序、保存中间代码等。

单道批处理系统(Simple Batch System)

单道批处理系统可以说是**第一代操作系统**，在**外存**中可以有一批**作业等待**，当内存没有作业可运行的时候，**Monitor** 就会从外存去选一个作业进入内存。最简单选择方式就是按**队列顺序**选择，但这不一定是最合理的，这就涉及到了**调度算法**。

多道程序系统(Multiprogramming)

一个支持 **Multiprogramming** 的系统允许多道程序同时**准备**运行。当正在运行的那道程序因为某种原因（比如等待输入或输出数据）暂时不能继续运行时，系统将**自动**地启动另一道程序运行。一旦原因消除（比如数据已经到达或数据已经输出完毕），暂时停止运行的那道程序在**将来某个时候**还可以被系统重新启动继续运行。

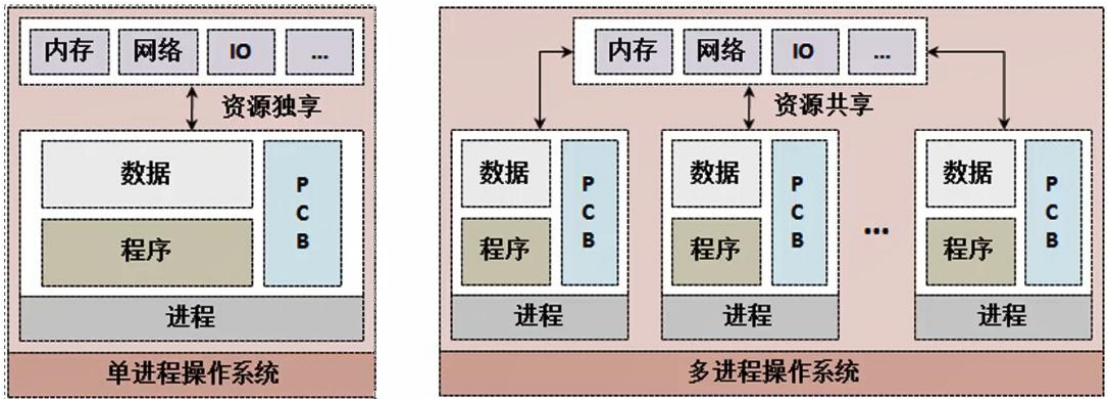


并发编程(Concurrent)

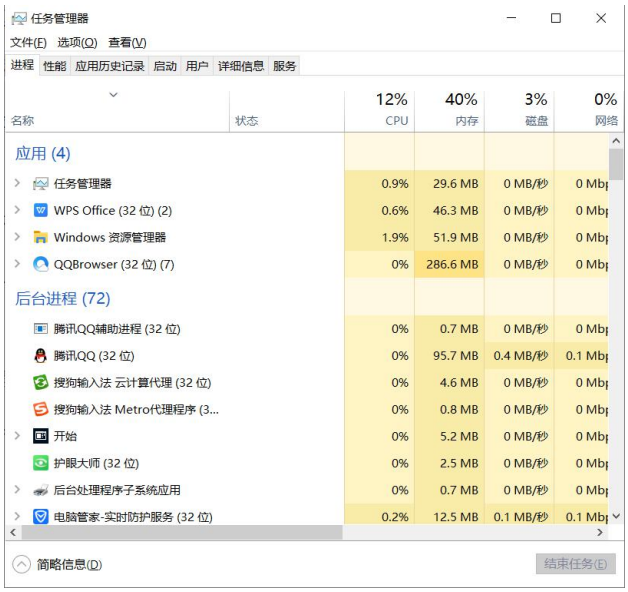
并发编程是一种有效**提高操作系统（服务器）性能**的技术手段，现代的操作系统之中最为重要的代表就是**并发性**，例如现在的 **CPU** 都属于多核 CPU。

早期的 **DOS 操作**系统有一个非常重要的特征：一旦系统沾染了病毒，那么所有的程序就无法直接执行了。因为传统的 **DOS 系统属于**单进程模型，**在同一个时间段上只能运行一个程序**，病毒程序运行了，其它程序自然就无法运行。

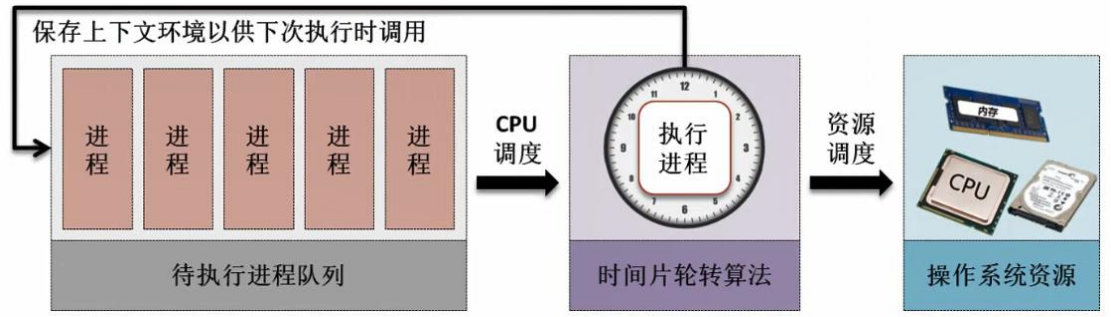
后来到了 **Windows 操作系统**，即便有病毒，也可以正常执行。这采用的是**多进程的编程模型**，**同一时间段上可以同时运行多个程序**。



只要打开 Windows 的任务管理器，就可以直接清楚发现所有正在执行的并行进程。



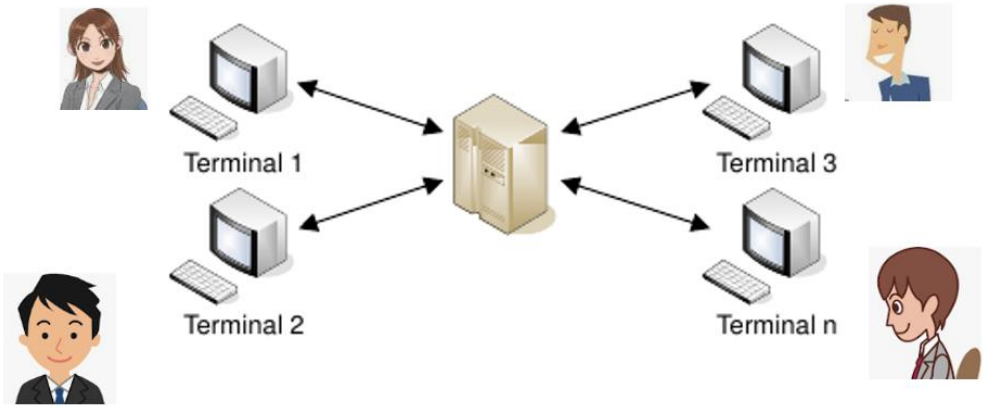
在早期的硬件系统之中由于没有多核 CPU 的设计，利用**时间片的轮转算法（Round Robin），保证在同一个时间段可以同时执行多个进行，但是在某一个时间点上只允许执行一个进程**，可以实现资源的切换。



服务器的硬件性能是有限的，但是对于大部分的程序来讲都属于**过剩**的状态。于是如果按照传统的单进程模式来运行程序，所有的硬件资源几乎都会被**浪费**。

分时系统(Time-Sharing System)

分时系统首先是个多道系统，分时在多道的基础上对每个任务以时间为固定切片，时间到了就切换到下一个任务。这种系统非常适合于**交互式系统**，例如访问网络服务器，采用**时间片轮转**的方式同时为几十个、几百个用户服务。



分时系统会有一些问题，比如时间片太短会**频繁**的**中断**，造成很大的系统开

销，或者有些任务不能被中断，被中断后可能恢复不了。所以分时系统是在某些需求的环境下才会提供。

移动系统(Mobile System)

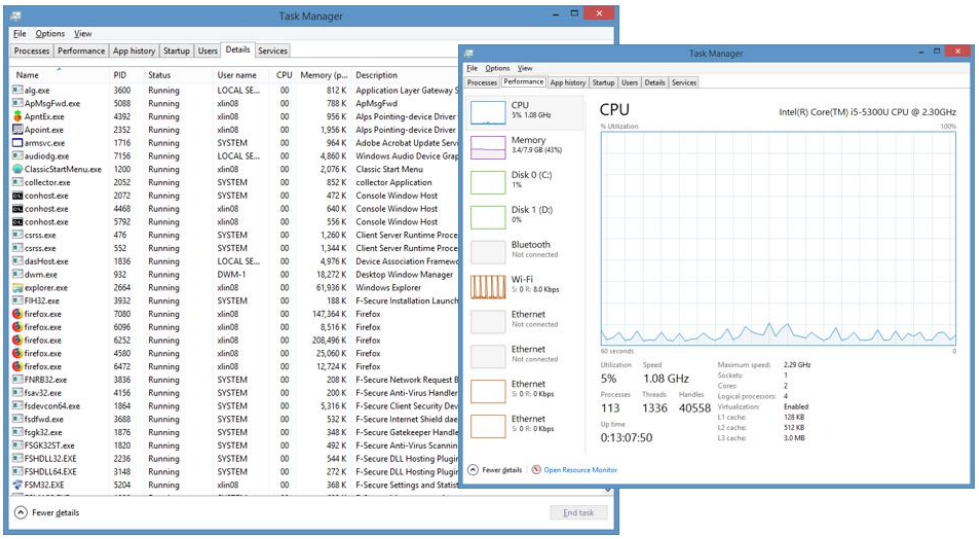
移动设备体积小，可以手持便于携带。移动设备一般都具有 LED 平板界面，提供数字化按钮和键盘的触摸屏界面。移动设备操作系统的例子包括 Apple iOS、Google Android、Research in Motion’s BlackBerry OS 等。

第 2 章 进程管理

2.1 进程的概念

进程(Process)

Windows 任务管理器提供了有关计算机性能的信息，并显示了计算机上所运行的程序和进程的详细信息。

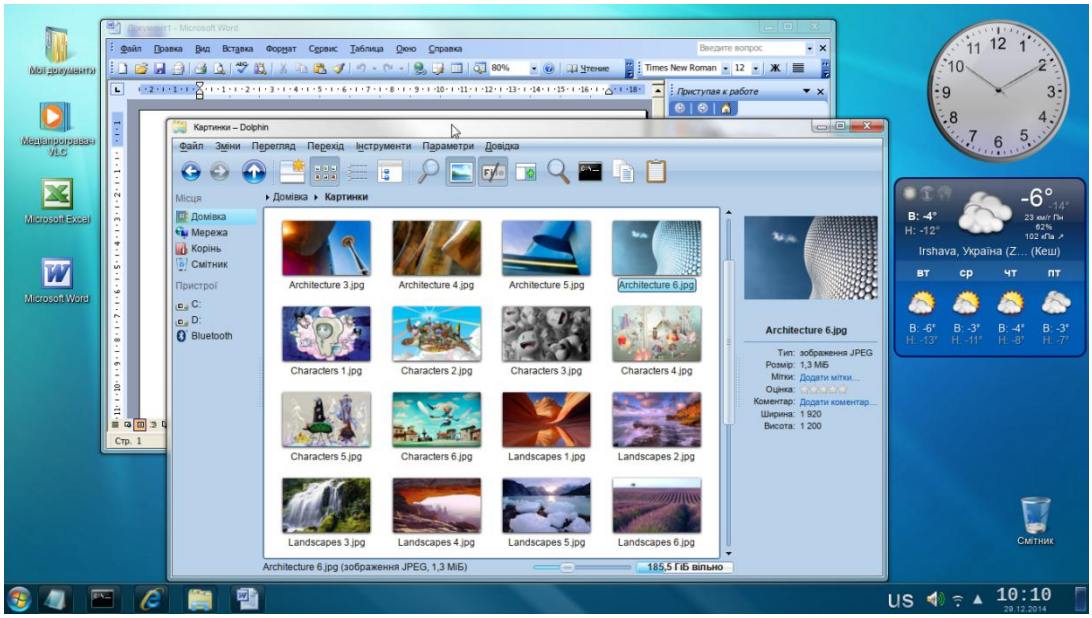


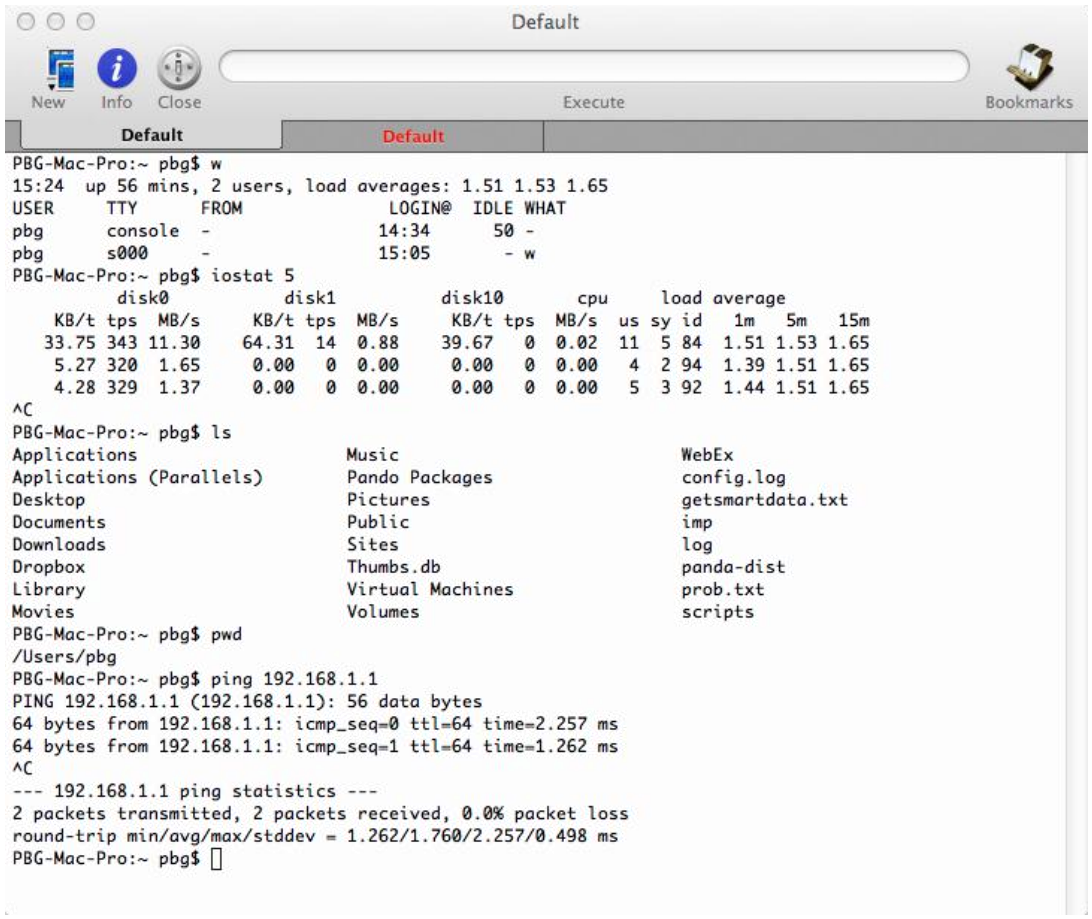
进程指的是一个具有一定独立功能的程序关于某个数据集合的一次运行活动。进程是系统进行资源分配和调度运行的基本单位。进程实体中包含三个组成部分：

1. 程序
2. 数据
3. PCB（进程控制块）

程序(program)与进程是有区别的，程序是静态的，进程是动态的。当程序的可执行文件被装入内存后就会变成进程。进程可以认为是执行中的程序，进程需要一定资源（如 CPU 时间、内存、文件、I/O 设备）完成任务。这些资源可

以在创建的时候或者运行中分配。一个程序可以通过 **GUI (Graphic User Interface)图形用户界面**的鼠标点击、**CLI (Command-line Interface)命令行界面**输入程序名等方式运行。



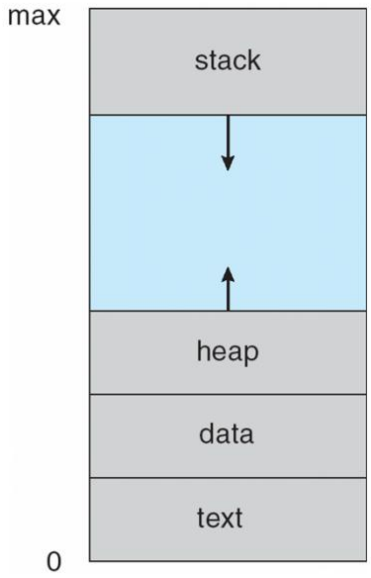



```
PBG-Mac-Pro:~ pbg$ w
15:24  up 56 mins, 2 users, load averages: 1.51 1.53 1.65
USER     TTY      FROM              LOGIN@  IDLE WHAT
pbg      console  -                 14:34     50 -
pbg      s000     -                 15:05      - w
PBG-Mac-Pro:~ pbg$ iostat 5
          disk0           disk1          disk10       cpu     load average
    KB/t tps  MB/s     KB/t tps  MB/s     KB/t tps  MB/s  us sy id   1m   5m   15m
   33.75 343 11.30    64.31  14  0.88    39.67   0  0.02  11  5 84  1.51 1.53 1.65
    5.27 320  1.65     0.00   0  0.00     0.00   0  0.00   4  2 94  1.39 1.51 1.65
    4.28 329  1.37     0.00   0  0.00     0.00   0  0.00   5  3 92  1.44 1.51 1.65
^C
PBG-Mac-Pro:~ pbg$ ls
Applications                    Music                           WebEx
Applications (Parallels)        Pando Packages                  config.log
Desktop                         Pictures                        getsmartdata.txt
Documents                       Public                          imp
Downloads                       Sites                           log
Dropbox                         Thumbs.db                       panda-dist
Library                         Virtual Machines                prob.txt
Movies                          Volumes                         scripts
PBG-Mac-Pro:~ pbg$ pwd
/Users/pbg
PBG-Mac-Pro:~ pbg$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1): 56 data bytes
64 bytes from 192.168.1.1: icmp_seq=0 ttl=64 time=2.257 ms
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.262 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 1.262/1.760/2.257/0.498 ms
PBG-Mac-Pro:~ pbg$
```

进程内存分配

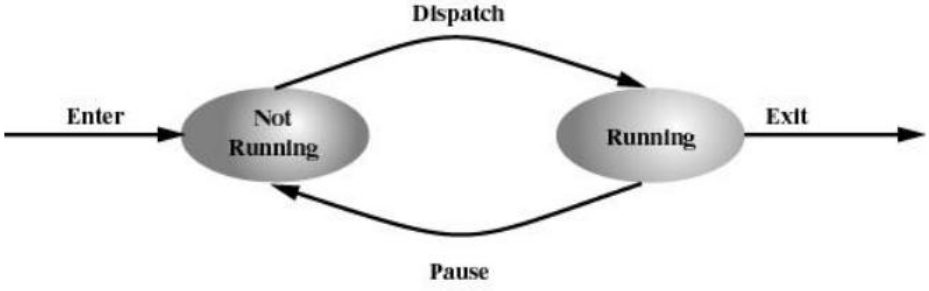
一个进程通常包括了**栈区(stack)、堆区(heap)、数据区、程序代码区**：

- **栈区：由编译器自动分配和释放，存放函数的参数值、局部变量的值等。**
- **堆区：一般由程序员分配和释放，若程序员不释放，程序结束后被 OS 回收。**
- **数据区：存放全局变量和静态变量，程序结束后由系统释放。**
- **程序代码区：存放函数体的二进制代码。**



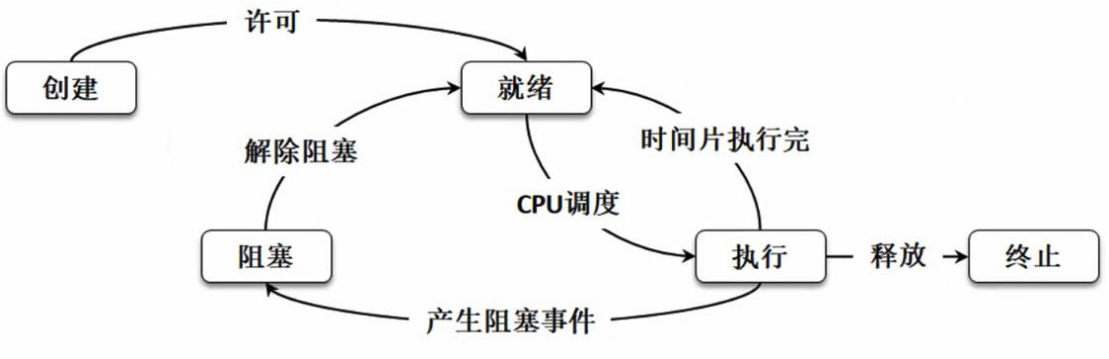
进程状态模型

在两状态进程模型中，进程被分为运行态(running)和非运行态(not-running)。



并非所有进程只要是非运行态就一定处于就绪状态，有的需要阻塞等待 I/O 完成。因此非运行态又可分为就绪态(ready)和阻塞态(block)。

所有的进程从其创建到销毁都有各自的**生命周期**，进程要经过如下几个阶段：



1. **创建状态**：系统已经为其分配了 PCB（可以获取进程的而信息），但是所需要执行的进程的上下文环境(context)还未分配，所以这个时候的进程还无法被调度。

2. **就绪状态**：该进程已经分配到除 CPU 之外的全部资源，并等待 CPU 调度。

3. **执行状态**：进程已获得 CPU 资源，开始正常提供服务。

4. **阻塞状态**：所有的进程不可能一直抢占 CPU，依据资源调度的算法，每一个进程运行一段时间之后，都需要交出当前的 CPU 资源，给其它进程执行。

5. **终止状态**：某一个进程达到了自然终止的状态，或者进行了强制性的停止，那么进程将进入到终止状态，进程将不再被执行。

多进程

Python 中在进行多进程开发的时候可以使用 **multiprocessing 模块**进行多进

程的编写，这个模块内容提供有一个 **Process 类**，利用这个类可以进行多进程的定义。所有的 Python 程序执行都是通过**主进程**开始的，所有通过 Process 定义的进程都属于**子进程**。

范例：创建多进程

```
import multiprocessing

def worker():
    """
        进程处理函数
    """
    print("【进程】id：%d，名称：%s" % (
        multiprocessing.current_process().pid,
        multiprocessing.current_process().name))

def main():
    print("【主进程】id：%d，名称：%s" % (
        multiprocessing.current_process().pid,
        multiprocessing.current_process().name))

    # 创建 3 个进程
    for i in range(3):
        process = multiprocessing.Process(target=worker, name="进程%d" % i)
        process.start()

if __name__ == "__main__":
    main()
```

运行结果	【主进程】id：4476，名称：MainProcess 【进程】id：14216，名称：进程 0 【进程】id：1424，名称：进程 1 【进程】id：16636，名称：进程 2

psutil 是一个进程管理的**第三方模块**，该模块可以**跨平台**（Linux、UNIX、MaxOS、Windows 都支持）地进行**进程管理**，可以极大地简化不同系统中的进程处理操作。

范例：获取全部进程信息

```
import psutil
```

```
def main():
    # 获取全部进程
    for process in psutil.process_iter():
        print("【进程】id：%d，名称：%s，创建时间：%s" % (
            process.pid, process.name,
            process.create_time()))

if __name__ == "__main__":
    main()
```

2.2 进程控制块 PCB

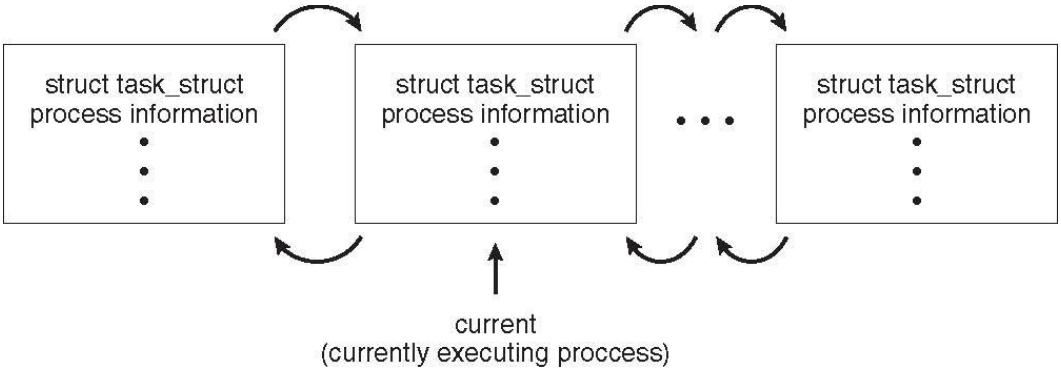
PCB (Process Control Block)

进程控制块 PCB 是 OS 控制和管理进程时所用的基本数据结构，PCB 是相关进程存在于系统中的唯一标志，系统根据 PCB 而感知相关进程的存在。

PCB 中包含了关于进程的一些信息，如**进程标识（pid）、程序计数器、状态信息、CPU 调度信息、内存管理信息、I/O 状态信息**等。

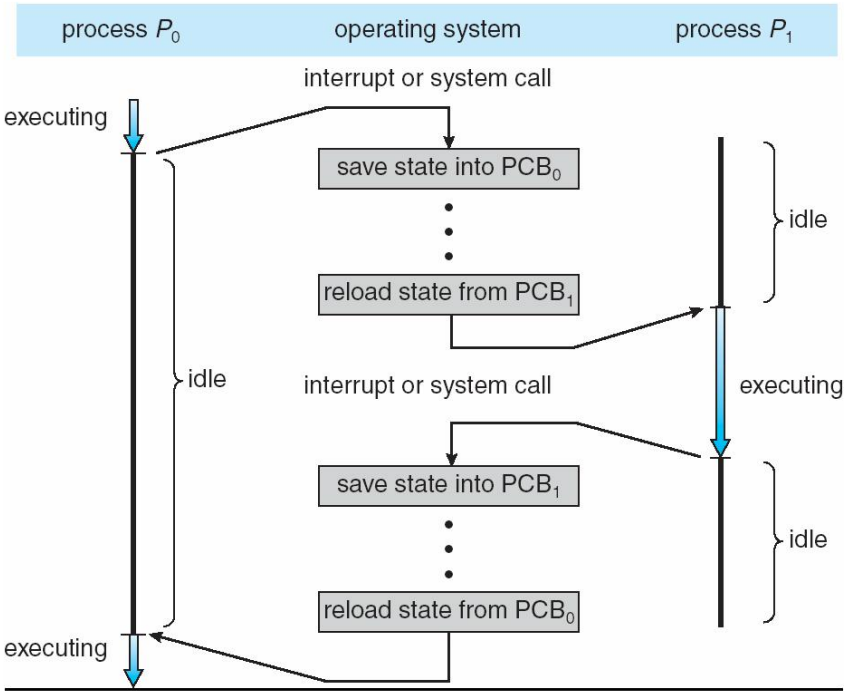
Linux 中 C 语言<linux/sched.h>头文件中定义的进程结构如下：

```
pid t_pid;                    /* process identifier */
long state;                   /* state of the process */
unsigned int time_slice;      /* scheduling information */
struct task_struct *parent;   /* this process's parent */
struct list_head children;    /* this process's children */
struct files_struct *files;   /* list of open files */
struct mm_struct *mm;         /* address space of this process */
```



进程切换

当 CPU 切换到另一个进程时，系统必须保存当前执行进程的**上下文**（context），并且加载需要被执行进程的上下文。进程的上下文都被保存在 PCB 中。

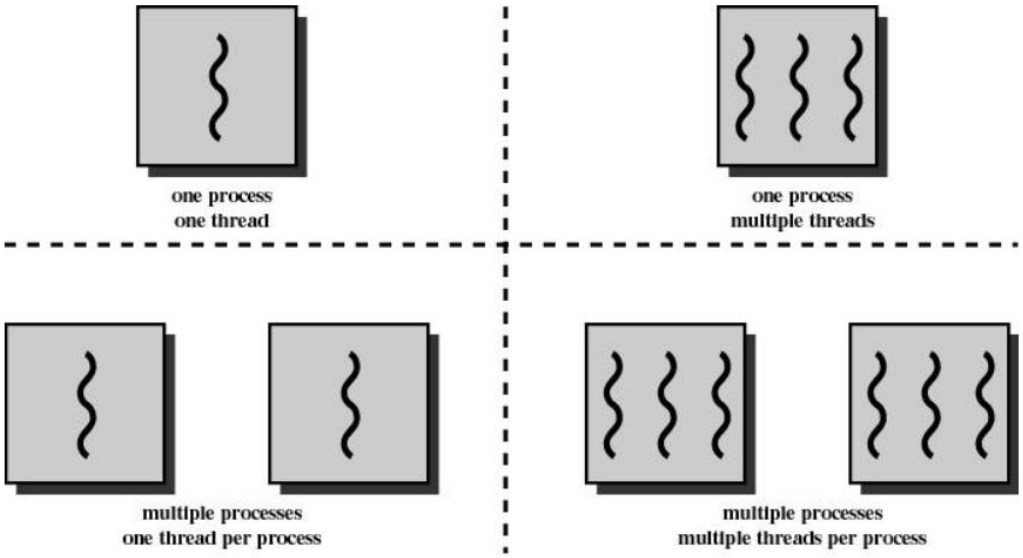

process P_0
operating system
process P_1
interrupt or system call
executing
save state into PCB_0
idle
reload state from PCB_1
idle
interrupt or system call
executing
save state into PCB_1
idle
reload state from PCB_0
executing

2.3 线程

线程(Thread)

60 年代，在 OS 中能拥有资源和独立运行的基本单位是进程，然而随着计算机技术的发展，进程出现了很多弊端。由于进程是资源拥有者，创建、撤消与切换存在较大的时空开销。因此在 80 年代，出现了能独立运行的基本单位“线程”。

线程是操作系统进行运算调度的最小单位，它被包含在进程中，是进程中的实际运作单位。一个进程中可以并发多个线程，每条线程并行执行不同的任务。



由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统内多个程序间并发执行的程度，从而显著提高系统资源的利用率和吞吐量。因此使用线程可以带来很多好处：

1. 创建一个线程相比创建一个进程花费更少的时间。
2. 结束一个线程相比结束一个进程花费更少的时间。
3. 在同一个进程中切换不同线程花费更少的时间。
4. 线程与资源分配无关，它属于某一个进程，与进程内其它线程一起共

享进程资源。

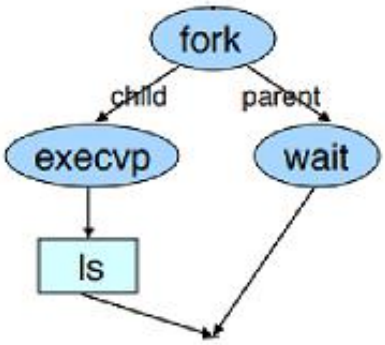
子进程(Child Process)

fork()函数是 UNIX 或类 UNIX 中的分叉函数，fork()函数将运行着的程序分成两个几乎完全一样的进程，每个进程都启动一个从代码的同一位置开始执行的线程。这两个进程中的线程继续执行，就像是两个用户同时启动了该应用程序的两个副本。

范例：创建子进程

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

int main() {
    pid_t pid;

    pid = getpid();
    printf("Before fork(): pid is %d\n", pid);

    // fork a child process
    pid = fork();

    if(pid < 0) {               // error
        fprintf(stderr, "Fork failed.\n");
        return 1;
    } else if(pid == 0) {       // child process
        printf("Child process: pid is %d\n", getpid());
        execlp("/bin/ls", "ls", NULL);
    } else {                    // parent process
        printf("Parent process: pid is %d\n", getpid());
        wait(NULL);
        printf("Child completed.");
    }

    return 0;
}
```

运行结果	Before fork(): my pid is 2250 Parent Process: my pid is 2250 Child Process: my pid is 2251 main main.c Child Complete



正常情况下，子进程由父进程创建，子进程可以再创建新的进程。父子进程是一个异步过程，**父进程永远无法预测子进程的结束**，所以，**当子进程结束后，它的父进程会调用 wait()或 waitpid()取得子进程的终止状态，回收掉子进程的资源。**

但是一些情况下会产生**僵尸进程(zoobie process)**和**孤儿进程(orphan process)**的情况：

1. **僵尸进程**：子进程退出了，但是父进程没有用 wait()或 waitpid()去获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中，这种进程称为僵尸进程。

2. **孤儿进程**：父进程结束了，而它的一个或多个子进程还在运行，那么这些子进程就成为了孤儿进程。子进程的资源由 init 进程（pid=1）回收。

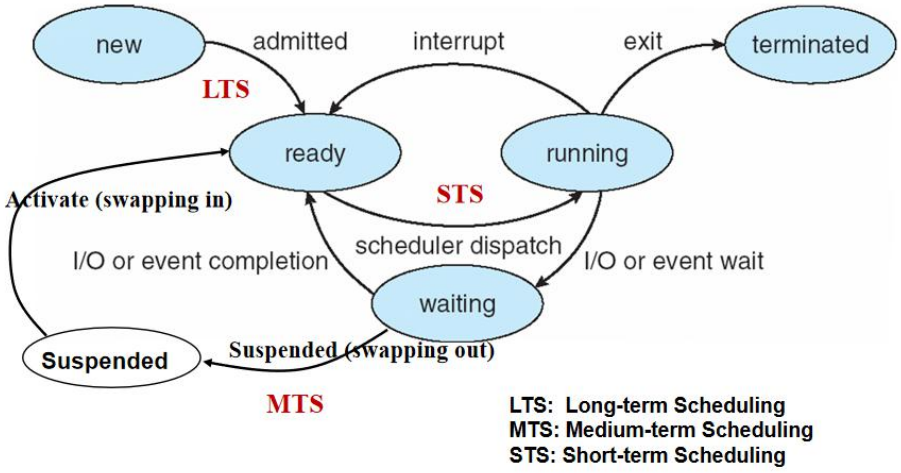
2.4 进程调度

进程调度(Scheduling)

短程调度（short-term scheduling）：从准备队列中选择进程送到 CPU 执行。

中程调度（medium-term scheduling）：从将外存中挂起的进程中选择进程送到内存。

长程调度（long-term scheduling）：从外存中选择一个作业送到内存中，为其创建进程，并加入准备队列。



调度准则(Scheduling Criteria)

调度的基本准则分为**面向用户**和**面向系统**两方面。

面向用户准则包括：

1. **响应时间（response time）快**：一般采用响应时间作为衡量调度算法的重要准则之一。从用户角度看，调度策略应尽量降低响应时间，使响应时间处在用户能接受的范围之内。

2. **周转时间（turnaround time）短**：周转时间是指从作业被提交给系

统开始，到作业完成为止的这段时间间隔。通常把周转时间的长短作为评价批处理系统的性能、选择作业调度方式与算法的重要准则之一。

3. **截止时间（deadline）的保证**：截止时间是指某任务必须开始执行的最迟时间，或必须完成的最迟时间。截止时间是用于评价实时系统性能的重要指标。

4. **优先权（priority）准则**：让某些紧急的作业能得到及时处理，在要求较严格的场合，往往还须选择抢占式调度方式（preemptive），才能保证紧急作业得到及时处理。

面向系统准则包括：

1. **系统吞吐量（throughput）高**：吞吐量是指在单位时间内系统所完成的作业数。

2. **处理机利用率高**：使处理机的利用率成为衡量系统性能的十分重要的指标。

3. **各类资源的平衡利用**：有效地利用其它各类资源，如内存、外存和 I/O 设备等。

4. **公平性（fariness）**：不能让某一些进程遭受饥饿。

决策模式(Decision Mode)

OS 调度算法的决策模式分为非抢占式（nonpreemptive）和**抢占式（preemptive）**。

非抢占式：一个进程一旦开始执行就会一直占用处理机，直到进程结束或发生阻塞。非抢占式主要用于批处理系统。

抢占式：当前正在运行的进程可以被打断，并转移到就绪态，可防止单一进程长时间独占 CPU。抢占式主要用于实时性要求较高的实时系统以及性能要求较高的批处理系统和分时系统。

先来先服务 FCFS (First Come First Serve)

最简单的 CPU 调度算法是 FCFS 算法，FCFS 策略可以通过 FIFO 队列容易地实现。当一个进程进入就绪队列时，它的 PCB 会被链接到队列尾部。当 CPU 空闲时，它会分配给位于队列头部的进程，并且将这个进程从队列中移去。

FCFS 策略的缺点是平均等待时间往往很长。

假设有一组进程，它们在时间 0 到达：

进程	执行时间（ms）
P1	24
P2	3
P3	3

如果进程按 P1、P2、P3 的顺序到达，Gantt 图如下：

进程 P1 的等待时间为 0，进程 P2 的等待时间为 24，而进程 P3 的等待时间为 27。因此，平均等待时间为(0 + 24 + 27) / 3 = 17ms。

不过，如果进程按 P2、P3、P1 的顺序到达，Gantt 图如下：

现在平均等待时间为(0 + 3 + 6) / 3 = 3ms，这个减少是相当大的。

FCFS 算法总结	
算法思想	主要从“公平”的角度考虑（类似生活中的排队）
算法规则	按照进程到达的先后顺序进行服务
决策模式	非抢占式
优点	公平、算法实现简单
缺点	排在长作业后面的短作业需要等待很长时间，带权周转时间很大，对短作业来说用户体验不好。

最短作业优先 SJF (Shortest Job First)

SJF 调度算法是**最优**的，因为进程的**平均等待时间最小**。通过将短进程移到长进程之前，短进程的等待时间减少大于长进程的等待时间增加。因而，平均等待时间减少。

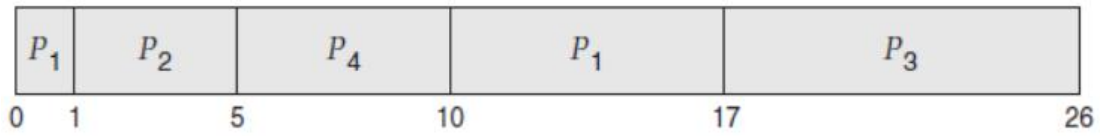
SJF 算法的真正**困难**是如何知道下次 CPU 执行的长度。一种方法是**预测**下一个 CPU 执行的长度，可以认为下一个 CPU 执行的长度与以前的相似。因此，通过计算下一个 CPU 执行长度的**近似值**，可以选择具有预测最短 CPU 执行的进程来运行。

SJF 算法可以是抢占或非抢占的。当一个新进程到达就绪队列而以前进程正在执行时，就需要选择了。新进程的下次 CPU 执行，与当前运行进程的尚未完成的 CPU 执行相比，可能还要小。抢占 SJF 算法会抢占当前运行进程，而非抢占 SJF 算法会允许当前运行进程先完成 CPU 执行。**抢占 SJF 调度有时称为最短剩余时间优先调度（Shortest Remaining Time First）**。

假设有一组进程：

进程	到达时间	执行时间（ms）
P1	0	8
P2	1	4
P3	2	9
P4	3	5

使用抢占 SJF 调度算法的 Gantt 图如下：



进程 P1 在时间 0 开始，因为这时只有进程 P1。进程 P2 在时间 1 到达。进程 P1 剩余时间（7ms）大于进程 P2 需要的时间（4ms），因此进程 P1 被抢占，而进程 P2 被调度。

抢占 SJF 调度的平均等待时间为[(10-1) + (1-1) + (17-2) + (5-3)] / 4 = 26 / 4 = 6.5ms。

如果使用非抢占 SJF 调度，平均等待时间为 7.75ms。

优先级调度(Priority Scheduling)

优先级调度算法的原理是给每个进程赋予一个**优先级**，每次需要进程切换时，找一个优先级最高的进程进行调度。这样，如果赋予长进程一个高优先级，则该进程就不会再“**饥饿（Starvation）**”。

优先级调度的优点是可以赋予重要的进程高优先级以确保重要任务能够得到 CPU 时间。**其缺点是低优先级的进程可能会“饥饿”**。不过，只要**动态地调**

节优先级即可。例如，在一个进程执行特定 CPU 时间后将其优先级降低一个级别，或者将处于等待进程的优先级提高一个级别。这样，一个进程如果等待时间很长，其优先级将因持续提升而超越其他进程的优先级，从而得到 CPU 时间。

假设有一组进程：

进程	执行时间（ms）	优先级
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

使用抢占优先级调度算法的 Gantt 图如下：

平均等待时间为(6 + 0 + 16 + 18 + 1) / 5 = 8.2ms。

轮转调度 RR (Round Robin)

RR 调度算法是专门为分时系统设计的，它类似于 FCFS 调度，但是增加了抢占以切换进程。RR 算法中，将一个较小时间单元定义为**时间量（time quantum）或时间片（time slice）**，时间片的大小通常为 10 ~ 100ms。

就绪队列作为循环队列，CPU 调度程序循环整个就绪队列，为每个进程分配不超过一个时间片的 CPU。有两种情况可能发生：

1. 进程可能只需少于时间片的 CPU 执行：进程本身会自动释放 CPU，

调度程序接着处理就绪队列的下一个进程。

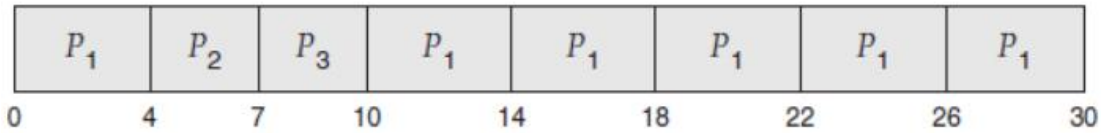
2. 当前运行进程的 CPU 执行大于一个时间片：时间片用完，进行上下文切换，再将进程加到就绪队列的尾部，接着 CPU 调度程序会选择就绪队列内的下一个进程。

不过采用 **RR 策略的平均等待时间通常较长**。

假设有一组进程，它们在时间 0 到达：

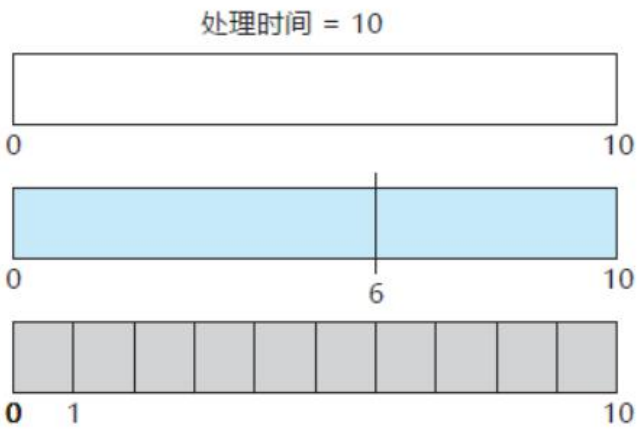
进程	执行时间（ms）
P1	24
P2	3
P3	3

如果使用 4ms 的时间片，那么 P1 会执行最初的 4ms。由于它还需要 20ms，所以在第一个时间片之后它会被抢占，而 CPU 就交给队列中的下一个进程。由于 P2 不需要 4ms，所以在其时间片用完之前就会退出。CPU 接着交给下一个进程 P3。在每个进程都得到了一个时间片之后，CPU 又交给了进程 P1 以便继续执行。



平均等待时间为[(10-4) + 4 + 7] / 3 = 17 / 3 = 5.66ms。

RR 算法的性能很大程度取决于时间片的大小。在一种极端情况下，如果时间片很大，那么 RR 算法与 FCFS 算法一样。相反，如果时间片很小，那么 RR 算法会导致大量的上下文切换。

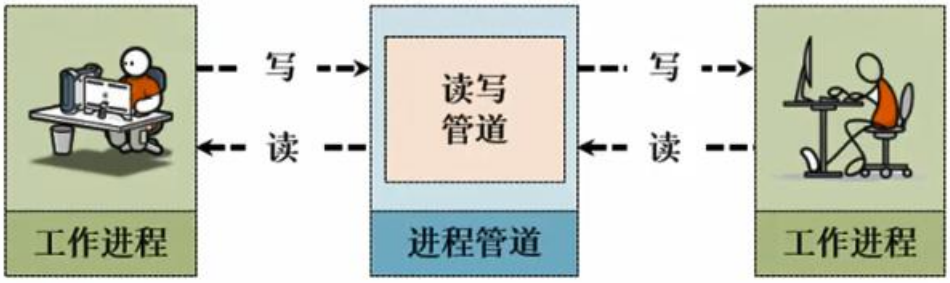


时间片	上下文切换
12	0
6	1
1	9

2.5 进程间通信

进程管道通信

进程是程序运行的基本单位，每一个程序内部都有属于自己的存储数据和程序单元，每一个进程都是完全独立的，彼此之前不能直接进行访问。但是可以通过一个特定的管道实现 IO。



范例：创建进程通讯管道

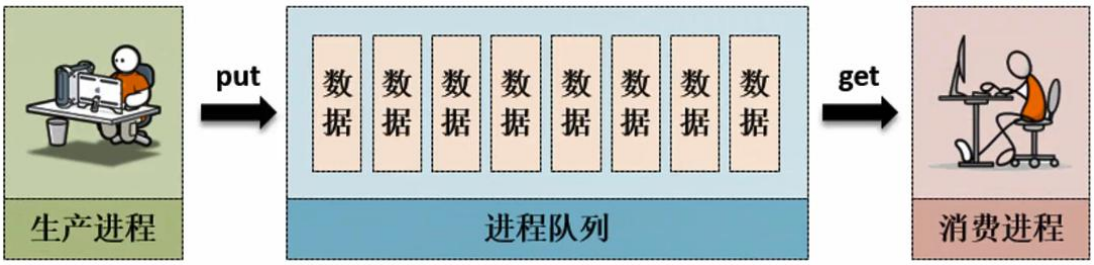
```
import multiprocessing

def send_data(pipe, data):
    """
        往管道发送数据
        Args:
            pipe (Pipe): 管道
            data (str): 发送的数据
    """
    pipe.send(data)
    print("【进程%d】发送数据：%s" % (
        multiprocessing.current_process().pid,
        data
    ))

def recv_data(pipe):
    """
        从管道接收数据
        Args:
            pipe (Pipe): 管道
    """
    print("【进程%d】接收数据：%s" % (
```

```
        multiprocessing.current_process().pid,
        pipe.recv()
    ))

def main():
    # 管道分为发送端和接收端
    send_end, recv_end = multiprocessing.Pipe()
    # 创建两个子进程，将管道传递到对应的处理函数
    sender = multiprocessing.Process(target=send_data,
                                     args=(send_end, "Hello!"))
    receiver = multiprocessing.Process(target=recv_data,
                                       args=(recv_end,))
    sender.start()
    receiver.start()

if __name__ == "__main__":
    main()
```

运行结果	【进程 11664】发送数据：Hello! 【进程 1032】接收数据：Hello!

生产者/消费者问题(Producer/Consumer Problem)

不同的进程彼此之间可以利用管道实现数据的发送和接收，但是如果发送的数据过多并且接收处理缓慢的时候，这种情况下就需要引入队列的形式来进行缓冲的操作。



multiprocessing.Queue 是 Python 多进程编程中提供的进程队列结构，该队列采用 **FIFO** 的形式实现不同进程间的数据通讯，这样可以保证多个数据可以按序实现发送与接收处理。

范例：进程队列

```
import multiprocessing
import time

def produce(queue):
    """
        生产数据
        Args:
            queue (Queue): 进程队列
    """
    # 生产 3 条数据
    for item in range(3):
        time.sleep(2)
        data = "data-%d" % item
        print("【%s】生产数据：%s" % (
            multiprocessing.current_process().name,
            data
        ))
        queue.put(data)

def consume(queue):
    """
        消费数据
        Args:
            queue (Queue): 进程队列
    """
    while True:      # 持续消费
        print("【%s】消费数据：%s" % (
            multiprocessing.current_process().name,
            queue.get()
        ))

def main():
    queue = multiprocessing.Queue()
    producer = multiprocessing.Process(
                target=produce, name="Producer", args=(queue,))
    consumer = multiprocessing.Process(
                target=consume, name="Consumer", args=(queue,))
    producer.start()
    consumer.start()

if __name__ == "__main__":
    main()
```

运行结果	【Producer】生产数据：data-0

	【Consumer】消费数据：data-0 【Producer】生产数据：data-1 【Consumer】消费数据：data-1 【Producer】生产数据：data-2 【Consumer】消费数据：data-2

2.6 互斥与同步

互斥与同步

计算机运行过程中，大量的进程在使用有限、独占、不可抢占的资源，由于进程无限，资源有限，这种矛盾称为竞争（Race）。

竞争条件分为两类：

1. 互斥（Metex）：两个或多个进程彼此之间没有内在的制约关系，但是由于要抢占使用某个临界资源（不能被多个进程同时使用的资源，如打印机）而产生制约关系。

2. 同步（Synchronization）：两个或多个进程彼此之间存在内在的制约关系（前一个进程执行完，其他的进程才能执行）。

在整个操作系统之中每一个进程都有自己独立的数据存储单元，也就是说不同进程之间无法直接实现数据共享。通过管道流可以实现进程之间的数据共享，相当于打通了不同进程之间的限制。但是不同的进程操作同一个资源就必须考虑**数据同步**的问题。

要理解同步概念，首先要清楚进程不同步所带来的的问题。

范例：售票操作（Bug 版本）

```
import multiprocessing
import time

def sell_ticket(dict):
    while True:      # 持续售票
        # 获取当前剩余票数
        num = dict.get("ticket")

        if num > 0:          # 如果还有票剩余
            time.sleep(1)    # 模拟网络延迟
```

```
            num -= 1          # 票数减 1
            print("【售票员%d】售票成功，剩余票数：%d" % (
                multiprocessing.current_process().pid,
                num
            ))
            dict.update({"ticket":num})      # 更新票数
        else:                 # 已经没有票了
            break

def main():
    # 创建共享数据对象
    manager = multiprocessing.Manager()
    # 创建一个可以被多个进程共享的字典对象
    ticket_dict = manager.dict(ticket=5)    # 默认有 5 张票

    # 创建多个售票进程
    sellers = [
        multiprocessing.Process(
            target=sell_ticket, args=(ticket_dict,))
        for _ in range(5)
    ]

    for seller in sellers:
        seller.start()
    for seller in sellers:
        seller.join()    # 进程强制执行

if __name__ == "__main__":
    main()
```

运行结果

```
【售票员 9732】售票成功，剩余票数：4
【售票员 1640】售票成功，剩余票数：4
【售票员 5976】售票成功，剩余票数：4
【售票员 8048】售票成功，剩余票数：4
【售票员 10516】售票成功，剩余票数：4
【售票员 9732】售票成功，剩余票数：3
【售票员 1640】售票成功，剩余票数：3
【售票员 5976】售票成功，剩余票数：3
【售票员 8048】售票成功，剩余票数：3
【售票员 10516】售票成功，剩余票数：3
【售票员 9732】售票成功，剩余票数：2
【售票员 1640】售票成功，剩余票数：2
【售票员 5976】售票成功，剩余票数：2
【售票员 8048】售票成功，剩余票数：2
```

	【售票员 10516】售票成功，剩余票数：2 【售票员 9732】售票成功，剩余票数：1 【售票员 1640】售票成功，剩余票数：1 【售票员 5976】售票成功，剩余票数：1 【售票员 8048】售票成功，剩余票数：1 【售票员 10516】售票成功，剩余票数：1 【售票员 1640】售票成功，剩余票数：0 【售票员 9732】售票成功，剩余票数：0 【售票员 5976】售票成功，剩余票数：0 【售票员 8048】售票成功，剩余票数：0 【售票员 10516】售票成功，剩余票数：0

多个进程同时进行票数判断的时候，在没有及时修改票数的情况下，就会出现**数据不同步**的问题。这套操作由于没有对同步的限制，所以就造成了不同步的问题。

解决互斥方法有两种：

1. 忙等待（Busy Waiting）：等着但是不停地检查测试，直达能进行为止。

2. 睡眠与唤醒（Sleep and Wakeup）：引入 Semapgore 信号量，为进程睡眠而设置，唤醒由其它进程引发。

临界区(Critical Section)

临界资源是各进程采取互斥的方式，一次仅允许一个进程使用的共享资源。属于临界资源的有打印机、变量、缓冲区等。

每个进程中访问临界资源的那段代码称为临界区，每次只允许一个进程进入临界区，进入后，不允许其它进程进入。不论是硬件临界资源还是软件临界资源，多个进程必须互斥的对它进行访问。使用临界区时，一般不允许其运行时间过长，只要运行在临界区的线程还没有离开，其它所有进入此临界区的线程都会被挂起而进入等待状态，因此会在一定程度上影响程序的运行性能。

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

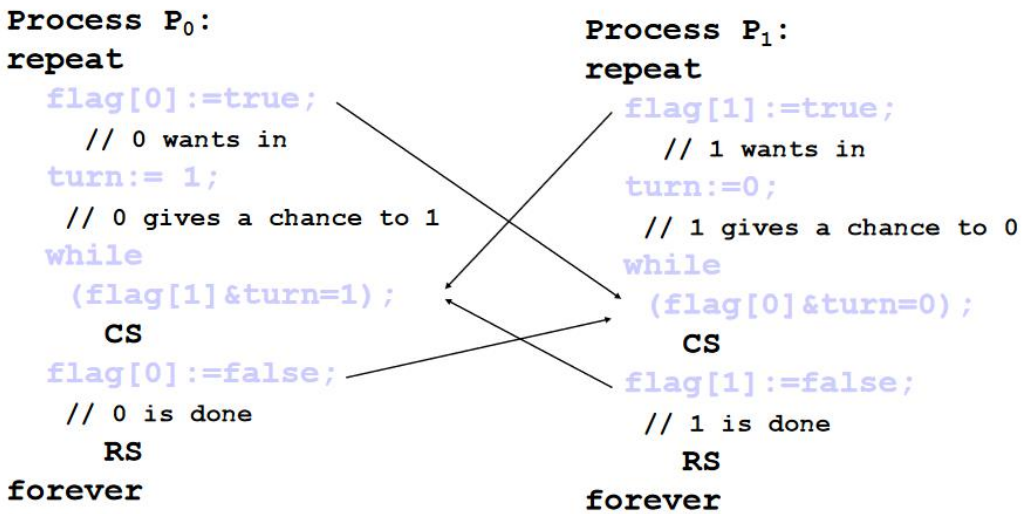
Peterson 算法

Peterson 算法是由 Gary L. Peterson 于 1981 年提出的一个实现**互斥锁**的并发算法，可以控制**两个进程**访问一个共享的单用户资源而不发生访问冲突。

Peterson 算法要求两个进程**共享两个数据项**：

```
int turn;
boolean flag[2];
```

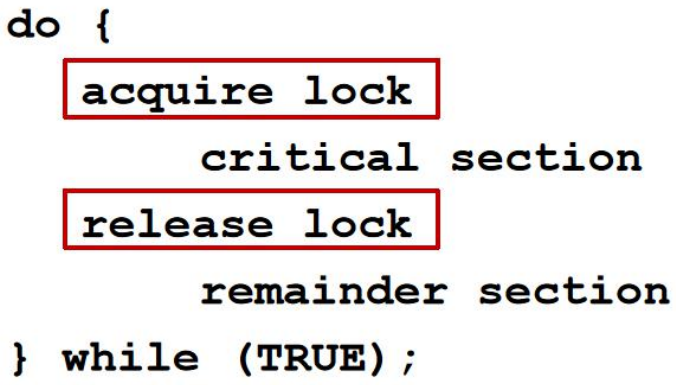
变量 turn 表示哪个进程可以进入临界区，数组 flag 表示哪个进程准备进入临界区。



互斥锁(Mutux Locks)

互斥锁是用一种简单的**加锁**方法来控制对共享资源的访问，互斥锁只有两种状态，即**上锁（lock / acquire）**和**解锁（unlock / release）**。

互斥锁必须设定为一个**原子操作（atomic operation）**，这意味着操作系统保证了如果一个进程锁定了一个互斥量，没有其它进程在同一时间可以成功锁定这个互斥量。

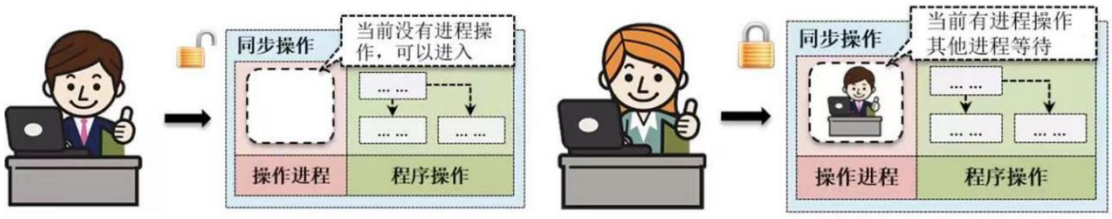


范例：互斥锁

```
acquire() {
    while(!available) {
        ;   //busy wait
    }
    available = false;
}

release() {
    available = true;
}

do {
    acquire()
    //critical section
    release()
    //remainder section
} while(true);
```

并发进程的执行如果要进行同步处理，那么就必须对一些核心代码进行同步。Python 中提供了一个 **Lock 同步锁机制**，利用这种锁机制可以实现部分代码

的**同步锁定，保证每一次只允许有一个进程**执行这部分的代码。



范例：售票操作

```
import multiprocessing
import time

def sell_ticket(lock, dict):
    while True:      # 持续售票
        # 请求锁定，如果 5 秒没有锁定则放弃
        lock.acquire(timeout=5)

        # 获取当前剩余票数
        num = dict.get("ticket")

        if num > 0:          # 如果还有票剩余
            time.sleep(1)    # 模拟网络延迟
            num -= 1         # 票数减 1
            print("【售票员%d】售票成功，剩余票数：%d" % (
                multiprocessing.current_process().pid,
                num
            ))
            dict.update({"ticket":num})      # 更新票数
        else:                # 已经没有票了
            break

        lock.release()       # 释放锁

def main():
    lock = multiprocessing.Lock()    # 同步锁
    # 创建共享数据对象
    manager = multiprocessing.Manager()
    # 创建一个可以被多个进程共享的字典对象
    ticket_dict = manager.dict(ticket=5)    # 默认有 5 张票

    # 创建多个售票进程
    sellers = [
        multiprocessing.Process(
```

```
            target=sell_ticket, args=(lock, ticket_dict))
        for _ in range(5)
    ]

    for seller in sellers:
        seller.start()
    for seller in sellers:
        seller.join()   # 进程强制执行

if __name__ == "__main__":
    main()
```

运行结果	【售票员 14612】售票成功，剩余票数：4 【售票员 15868】售票成功，剩余票数：3 【售票员 13972】售票成功，剩余票数：2 【售票员 10844】售票成功，剩余票数：1 【售票员 2872】售票成功，剩余票数：0

一旦程序中追加了同步锁，那么程序的部分代码就只能以**单进程**执行了，这样势必会造成程序的执行**性能下降**，只有在考虑**数据操作安全**的情况下才会使用锁机制。

硬件实现

test_and_set()函数是用**硬件**保持的**原子操作**，这操作**不会被打断**。

范例：test_and_set()

```
boolean test_and_set (boolean *target) {
    boolean rv = *target;
    *target = true;
    return rv;
}
```

共享锁 lock 初始化为 false。while(Test_And_Set_Lock(&lock));这句后面是的分号表示会**循环等待解锁**。

范例：test_and_set 实现互斥锁

```
do {
    while(test_and_set(&lock)) {
        ;       // do nothing
```

```
    }
    // critical section
    lock = false;
    // remainder section
} while(true);
```

compare_and_swap()函数也是按**原子操作**执行，**不接受中断**。

范例：compare_and_swap()

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if(*value == expected) {
        *value = new_value;
    }
    return temp;
}
```

互斥锁 lock 初始化为 0。

范例：test_and_set 实现互斥锁

```
do {
    while(compare_and_swap(&lock, 0, 1) != 0) {
        ;       // do nothing
    }
    // critical section
    lock = 0;
    // remainder section
} while(true);
```

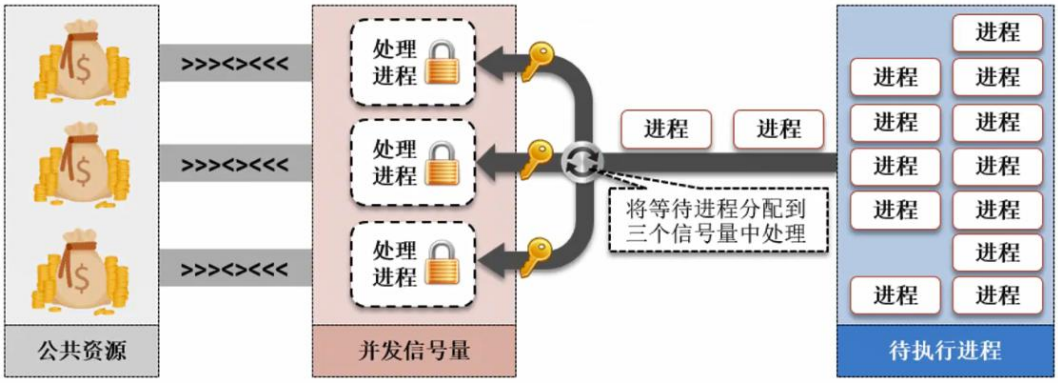
使用**机器指令**的**优点**是简单且易于证明，可以**支持多个临界区**，每个临界区都可以用它自己的变量定义。**缺点**是需要**忙等待**，并且可能会引发**饥饿**，当一个进程离开一个临界区并且有多个进程正在等待时，选择哪一个进程是任意的，因此可能会有进程被无限地拒绝进入。

2.7 Semaphore

信号量(Semaphore)

Semaphore（**信号量**）是一种**有限资源**的**进程同步管理机制**。例如银行的业务办理，所有客户都会拿到一个号码，而后号码会被业务窗口叫号，被叫号的办理者就可以办理业务。

Semaphore 类本质上是一种带有计数功能的进程同步机制，**acquire()减少计数**，**release()增加计数**。当**可用信号量的计数为 0** 时，后续进程将被**阻塞**。



Lock 一般是针对于一个资源同步的，而 Semaphore 是针对有限资源的并行访问。

范例：信号量同步处理

```
import multiprocessing
import time

def work(sema):
    if sema.acquire():       # 获取信号量
        print("【进程%d】开始办理业务" %
            multiprocessing.current_process().pid)
        time.sleep(2)        # 模拟办理业务
        print("【进程%d】结束办理业务" %
            multiprocessing.current_process().pid)
        sema.release()       # 释放资源
```

```python
def main():
    # 允许 3 个进程并发执行
    sema = multiprocessing.Semaphore(3)
    workers = [
        multiprocessing.Process(target=work, args=(sema,))
        for _ in range(10)
    ]

    for worker in workers:
        worker.start()
    for worker in workers:
        worker.join()

if __name__ == "__main__":
    main()
```

运行结果	【进程 7880】开始办理业务 【进程 3032】开始办理业务 【进程 5412】开始办理业务 【进程 7880】结束办理业务 【进程 2876】开始办理业务 【进程 3032】结束办理业务 【进程 14076】开始办理业务 【进程 5412】结束办理业务 【进程 8816】开始办理业务 【进程 2876】结束办理业务 【进程 14076】结束办理业务 【进程 7900】开始办理业务 【进程 7860】开始办理业务 【进程 8816】结束办理业务 【进程 16252】开始办理业务 【进程 7860】结束办理业务 【进程 7900】结束办理业务 【进程 972】开始办理业务 【进程 16252】结束办理业务 【进程 972】结束办理业务

一个信号量 Semaphore 是一个整型量，除对其初始化外，它只能由两个原子操作 wait()和 signal()进行访问。wait()和 signal()即早前使用的 P/V 操作，P/V 的名称来源于荷兰文 proberen（测试）和 verhogen（增量）。

范例：wait() / signal()

```
wait(s) {
```

```
    while(s <= 0) {
        ;       // busy waiting
    }
    s--;
}

signal(s) {
    s++;
}
```

但这并不是信号量的最终实现，最终的信号量实现最好是能**解决两个问题**：

1. **不能忙等。**
2. **记录处于等待状态的进程数量。**

为了避免进程忙等，wait()和 signal()的定义需要进行修改。**当一个进程执行 wait()但发现信号量 s <= 0 时，它必须等待，这里的等待不是忙等，而是阻塞自己。当一个进程阻塞且等待信号量时，在其它进程执行 signal()之后被重新执行。**

信号量的意义为：

- **s > 0：表示有 s 个资源可用。**
- **s = 0：表示无资源可用。**
- **s < 0：表示等待队列中有|s|个进程。**

为了定义基于**阻塞（block）**和**唤醒（wakeup）**的信号量，可以将信号量定义为如下结构：

范例：Semaphore 结构

```
typedef struct {
    int value;
    struct process *list;
} Semaphore;
```

范例：无需忙等的 wait() / signal()

```
wait(Semaphore *s) {
    s->value--;
    if(s->value < 0) {
        add this process to s->list;
        block();
    }
}

signal(Semaphore *s) {
    s->value++;
    if(s->value <= 0) {
        remove a process p from s->list;
        wakeup(p);
    }
}
```

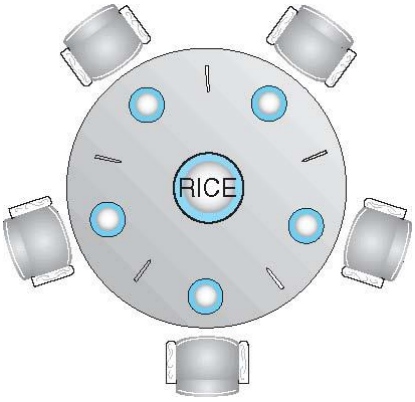
2.8 死锁的概念

死锁(Deadlock)

计算机系统中有很多**独占性**的资源，在同一时刻只能每个资源只能由一个进程使用。**死锁**是指两个或两个以上的进程在执行过程中，由于**竞争资源**或者由于彼此通信而造成的一种永久阻塞的现象。若无外力作用，它们都将无法推进下去，此时称系统处于死锁状态。

哲学家就餐问题(Dining Philosophers Problem)

假设有五位哲学家围坐在一张圆形餐桌旁，哲学家只做两件事情：吃饭或思考。餐桌上有一碗食物，每两个哲学家之间有一根筷子，哲学家必须用两根筷子才能吃东西。他们只能使用自己左右手边的那两根筷子。



将 5 位哲学家分别编号为 0~4，第 i 位哲学家左手边的筷子编号为 i。

范例：哲学家就餐问题（死锁）

```
do {
    wait(chopstick[i]);              // 申请左筷子
    wait(chopstick[(i + 1) % 5]);    // 申请右筷子
    // eat
    signal(chopstick[(i + 1) % 5]); // 释放右筷子
    signal(chopstick[i]);            // 释放左筷子
} while(true);
```

但是这个算法存在死锁的问题。每个哲学家都拿着左筷子，永远都在等右筷子（或者相反）。

在实际的计算机问题中，缺乏筷子可以类比为缺乏**共享资源**。一种常用的计算机技术是资源**加锁**，用来保证在某个时刻，资源只能被一个程序或一段代码访问。当一个程序想要使用的资源已经被另一个程序锁定，它就等待资源解锁。当多个程序涉及到加锁的资源时，在某些情况下就有可能发生死锁。例如，某个程序需要访问两个文件，当两个这样的程序各锁了一个文件，那它们都在等待对方解锁另一个文件，而这永远不会发生。

产生死锁的条件

产生死锁必须要同时满足 **4 个必要条件**：

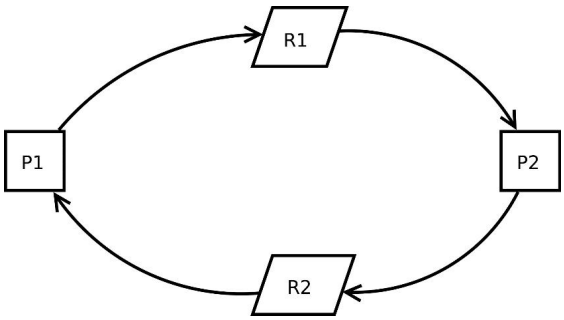
1. **互斥（mutual exclusion）**：进程要求对所分配的资源进行排它性控制，即在一段时间内某资源仅有一个进程所占用。

2. **占有并等待（hold and wait）**：当进程因请求资源而阻塞时，对已获得的资源保持不放。

3. **不可剥夺（no preemption）**：进程已获得的资源在未使用完之前，不

能剥夺，只能在使用完时由自己释放。

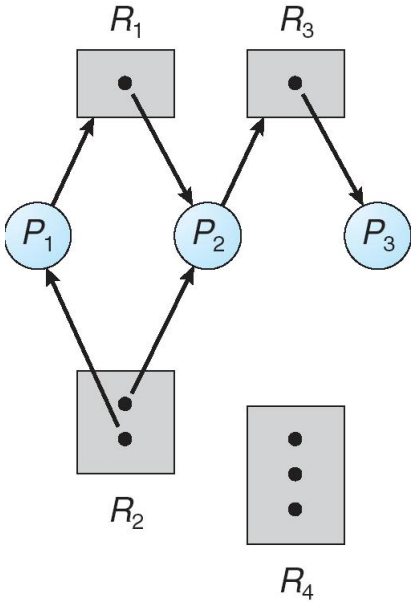
4. **循环等待（circular wait）**：一定会有一个环互相等待。



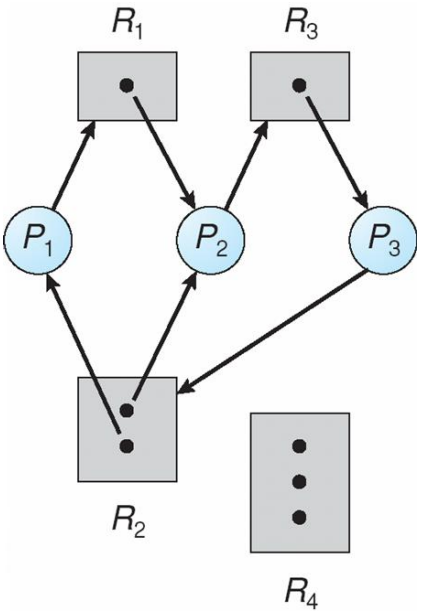
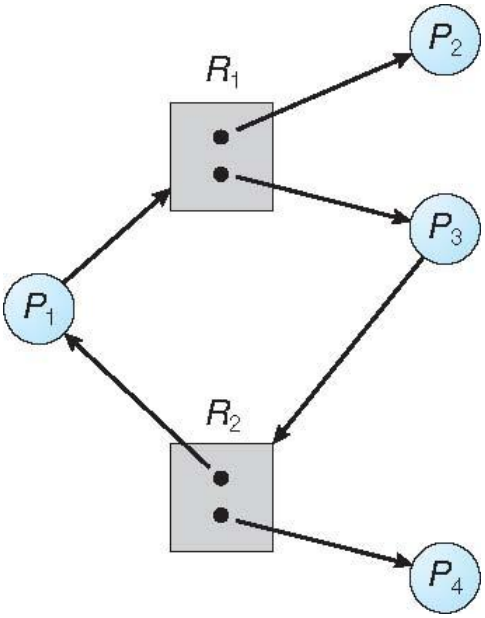
资源分配图(Resouce Allocation Graph)

资源分配图是一种**有向图**，一个图 G 可以由**结点集 V** 以及**边集 E** 组成。

在资源分配图中，结点集 $V = P \cup R$，边集 $E = \{(P_i, R_i) \cup (R_i, P_i)\}$，其中 P 为系统中所有进程的集合，R 为系统中所有资源类的集合。(P_i, R_i)表示一条由进程 P_i 到资源类 R_i 的有向边，即进程 P_i 申请资源 R_i，(R_i, P_i)则是表示资源的分配。



如果一个图中没有环路（回路），则系统中不存在死锁。若有环路，系统可能存在死锁。

死锁	有环路但不是死锁
	

2.9 死锁的预防与避免

死锁预防

可以通过破坏死锁产生的 4 个必要条件来预防死锁：

1. **破坏“互斥”条件**：由于资源互斥是资源使用的固有特性是**无法改变**的。

2. **破坏“占有并等待”条件**：每个进程在开始执行时就申请所需要的全部资源，只要有一个请求的资源不可用，其它可用资源就都不分配给它。采用该方法对系统来说是**非常浪费**的。

3. **破坏“不可剥夺”条件**：一个已拥有资源的进程，若它再提出新资源要求而不能立即得到满足时，它必须释放已经拥有的所有资源，以后需要时再重新申请。该方法实现复杂且要**付出很大的代价**，会导致之前的**工作失效**。

4. **破坏“循环等待”条件**：系统将所有资源按类型进行线性排序，并赋予不同的序号，所有进程对资源的请求必须严格按照资源序号递增的次序提出。该方法的**缺点**是进程实际需要资源的顺序不一定与资源的编号一致，因此仍会造成**资源浪费**。同时资源的序号必须相对稳定，从而限制了新设备的增加。

鸵鸟算法(Ostrich Algorithm)

传说中鸵鸟看到危险就把头埋在地底下。当你对某一件事情没有一个很好的解决方法时，那就忽略它，装作看不到。这样的算法称为“**鸵鸟算法**”。

鸵鸟算法可以称之为不是办法的办法。在计算机科学中，鸵鸟算法是**解决潜**

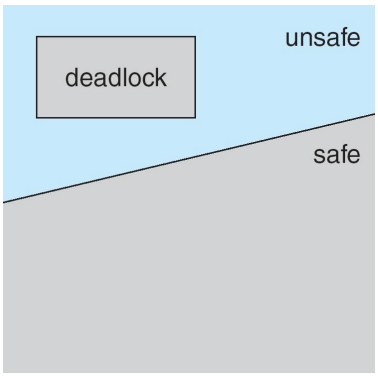
在问题的一种方法。假设的前提是，这样的问题出现的概率很低。比如，在操作系统中，为应对死锁问题，可以采用这样的一种办法。大多数操作系统，包括UNIX、MINUX 和 Windows，处理死锁问题的办法仅仅是忽略它。其假设前提是大多数用户宁可在极偶然的情况下发生死锁也不愿接受性能的损失。因为解决死锁的问题，通常代价很大。所以鸵鸟算法，是平衡性能和复杂性而选择的一种方法。当死锁真正发生且影响系统正常运行时，采取手动干预——重新启动。

死锁避免

死锁避免的基本思想就是在进行系统资源分配之前，先计算此次资源分配的安全性。若此次分配不会导致系统进入不安全状态，则将资源分配给进程，否则让进程等待。

若系统能按某种顺序如<P_1, P_2, ..., P_n>来为每个进程分配其所需资源，直至最大需求，使每个进程都可顺序完成，则称系统处于安全状态（safe state）。若系统不存在这样一个安全序列，则称系统处于不安全状态（unsafe state）。

只要系统处于安全状态，系统便不会进入死锁状态。当系统处于不安全状态时，并非所有不安全状态都必然转换为死锁。



银行家算法(Banker's Algorithm)

银行家算法可用于银行发放一笔贷款前，预测该笔贷款是否会引起银行资金周转问题。这里，银行的资源就类似于计算机系统的资源，贷款业务类似于计算机的资源分配。**该算法可以预测一次资源分配对计算机系统是否是安全的**。

银行家算法需要设置以下数据结构：

1. **可利用资源向量 Available**：一个具有 m 个元素的数组，其中的每一个元素代表一类可利用资源的数目，其初始值为系统中该类资源的最大可用数目。其值将随着该类资源的分配与回收而动态改变。Available[j] = k 表示系统中现有 R_j 类资源 k 个。

2. **最大需求矩阵 Max**：一个 n * m 的矩阵，定义了系统中 n 个进程中每一个进程对 m 类资源的最大需求。Max[i, j] = k 表示进程 i 对 R_j 类资源的最大需求数目为 k 个。

3. **分配矩阵 Allocation**：一个 n * m 的矩阵，定义了当前分配给每个进程的各类资源数量。Allocation[i, j] = k 表示进程 i 当前已分得 R_j 类资源的数目为 k 个。

4. **需求矩阵 Need**：一个 n * m 的矩阵，表示每一个进程尚需的各类资源数。Need[i, j] = k 表示进程 i 还需要 R_j 类资源 k 个方能完成任务。

矩阵间的关系为 **Need[i, j] = Max[i, j] - Allocation[i, j]**。

设 $Request_i$ 是进程 P_i 的请求向量，$Request_i[j] = k$ 表示进程 P_i 需要 k 个 R_j 类资源。当进程 P_i 发出资源请求后，系统按下述步骤进行检查：

1. 如果 $Request_i <= Need_i$，则跳转到步骤 2；否则，出错。

2. 如果 $Request_i <= Available$，则跳转到步骤 3；否则，表示尚无足够资源可供分配，进程 P_i 必须阻塞等待。

3. 系统**试探性地**将 P_i 申请的资源分配给它，并修改下列数据：

```
Available = Available - Request[i];
Allocation = Allocation + Request[i];
Need[i] = Need[i] - Request[i];
```

4. 系统利用**安全性算法**，检查此次资源分配以后，系统是否处于安全状态。若安全，才正式讲资源分配给进程 P_i。否则，试探分配失效，让进程 P_i 阻塞等待。

例如已知系统中进程的资源需求状况：

最大需求矩阵 Max					分配矩阵 Allocation				需求矩阵 Need			
	R1	R2	R3			R1	R2	R3		R1	R2	R3
P1	3	2	2	-	P1	1	0	0	= P1	2	2	2
P2	6	1	3		P2	6	1	2	P2	0	0	1
P3	3	1	4		P3	2	1	1	P3	1	0	3
P4	4	2	2		P4	0	0	2	P4	4	2	0

资源向量 Resource				可用资源向量 Available		
R1	R2	R3		R1	R2	R3
9	3	6		0	1	1

当前的可用资源为(0, 1, 1)，如果将这些资源分配给 P_1、P_3、P_4，并不能满足满足它们的需求。因为 P_2 只需要一份 R_3 资源就可以执行，执行结束后 P_2 会释放所有资源，那么可用资源变为(6, 2, 3)。此时可用资源可以满足剩余任意一个进程，而都不会进入不安全状态。因此，<P_2, P_1, P_3, P_4>就是其中的一个安全序列。

如果无法找到一个安全序列，那么系统就处于不安全状态，便有可能进入死锁状态。

2.10 死锁的检测与解除

死锁检测与解除

检测死锁不同于预防死锁，不限制资源访问方式和资源申请。OS 可以周期性地执行死锁检测例程，检测系统中是否出现环路等待。

当发现有进程死锁后，便应立即把它从死锁状态中解脱出来，常采用的方法有：

1. **撤销进程法（abort）**：强制撤销部分、甚至全部死锁进程并剥夺这些进程的资源。撤销的原则可以按进程优先级和撤销进程代价的高低进行。

2. **进程回退法（rollback）**：让一个或多个进程回退到足以回避死锁的地步，进程回退时自愿释放资源而不是被剥夺。要求系统保持进程的历史信息，设置还原点。

哲学家就餐问题(Dining Philosophers Problem)

之前对于哲学家就餐问题的解决算法可能产生死锁。为了使全部哲学家都能进餐，算法必须避免死锁和饥饿。可行的解决方案是**最多只允许 4 个哲学家同时进餐厅**，则至少有 1 个哲学家可以拿到两根筷子进餐。进餐完毕后放下筷子，其他哲学家就可进餐，这样就不会出现死锁和饥饿。

范例：哲学家就餐问题

```
Semaphore room = 4;

Process phiosopher(i) {
    do {
        wait(room);                    // 第 5 位哲学家将被阻塞
        wait(chopstick[i]);            // 申请左筷子
```

```
        wait(chopstick[(i + 1) % 5]);   // 申请右筷子
        // eat
        signal(chopstick[(i + 1) % 5]); // 释放右筷子
        signal(chopstick[i]);           // 释放左筷子
        signal(room);                   // 唤醒被阻塞的哲学家
    } while(true);
}
```

第 4 章 I/O 设备管理

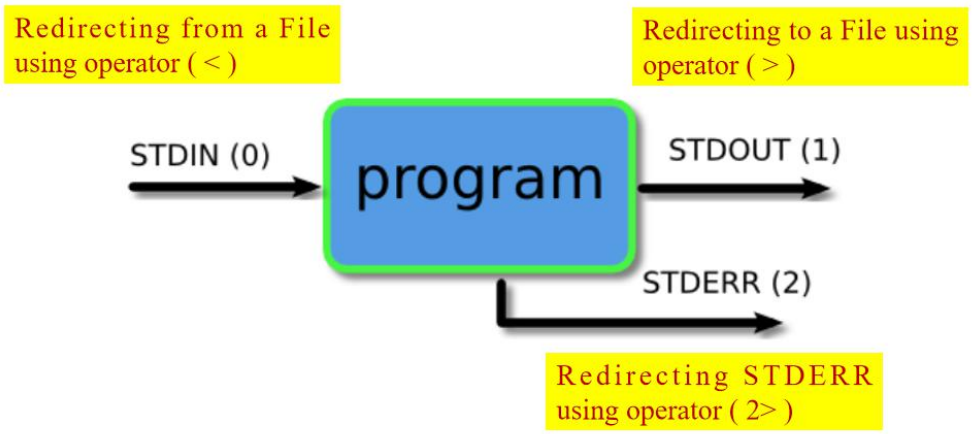
4.1 I/O 重定向

标准 I/O (Standard I/O)

程序对读入的数据进行处理，再输出数据。**数据的输入(Input)和输出(Output)简称为 I/O，在没有指定输入输出的情况下，默认为标准输入和标准输出。**

打开的文件都有一个**文件描述符（fd: file descriptor）**，表现为一个数字：

1. **标准输入 stdin（键盘）：fd = 0。**
2. **标准输出 stdout（显示器）：fd = 1。**
3. **标准错误输出 stderr（显示器）：fd = 2。**



范例：标准 I/O

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int num;
    fscanf(stdin, "%d", &num);
    fprintf(stdout, "[stdout] num = %d\n", num);
    fprintf(stderr, "[stderr] This is an error message.\n");
    return 0;
```

```
}
```

运行结果	123 [STDOUT] num = 123 [STDERR] This is an error message.

范例：文件 I/O

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

void writeFile(const char *filename) {
    FILE *fp = fopen(filename, "w");
    if(!fp) {
        fprintf(stderr, "File open failed.\n");
        exit(1);
    }
    char *name = "小灰";
    int age = 17;
    double height = 182.3;
    fprintf(fp, "姓名：%s\n 年龄：%d\n 身高：%.2f\n",
                name, age, height);
    fclose(fp);
}

void readFile(const char *filename) {
    FILE *fp = fopen(filename, "r");
    if(!fp) {
        fprintf(stderr, "File open failed.\n");
        exit(1);
    }
    char name[32];
    int age;
    double height;
    fscanf(fp, "姓名：%s\n 年龄：%d\n 身高：%lf\n",
                name, &age, &height);
    printf("name: %s\n", name);
    printf("age: %d\n", age);
    printf("height: %.2f\n", height);
    fclose(fp);
}

int main(int argc, char *argv[]) {
```

```
    const char *filename = "info.txt";
    writeFile(filename);
    readFile(filename);
    return 0;
}
```

运行结果	name: 小灰 age: 17 height: 182.30

I/O 重定向(I/O Redirection)

I/O 重定向就是改变标准输入与输出的默认位置。标准输入默认是键盘，通过改成其它输入，就是输入重定向，例如从文本文件里输入。标准输出默认是显示器，通过改成其它输出，就是输出重定向，例如输出到文件。

输出重定向用“>”表示，若文件不存在，则创建；若文件已存在，则覆盖。使用“>>”时若文件不存在，则创建，若文件已存在，则追加。

错误输出重定向用“2>”和“2>>”表示。

输入重定向用“<”表示，但是在输入重定向中“<<”可不是表示输入追加。

范例：I/O 重定向

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    int data;
    scanf("%d", &data);
    printf("data = %d\n", data);
    return 0;
}
```

编译	gcc -Wall io_redirection.c -o io_redirection
命令行	Windows cmd: io_redirection < info.txt > output.txt Linux terminal: ./io_redirection < input.txt > output.txt
input.txt	12345

output.txt	num = 12345