Functions of operating system

- File management
- Device management
- Memory management
- Process management

Functions of operating system

- 文件管理
  - File management system: FAT32 NTFS EXT3 EXT4 ZFS Btrfs
  - 控制文件的访问
  - 管理文件的创建、删除和修改
  - 给文件命名
  - 管理文件的存储

FAT (File Allocation Table):

- 早期个人计算机中常用的文件系统，如MS-DOS和Windows 95/98。
- 支持FAT12、FAT16和FAT32等不同版本。

NTFS (New Technology File System):

- 微软Windows NT操作系统的文件系统，提供了ACL（访问控制列表）、文件加密和其他高级特性。

exFAT:

- 主要用于闪存驱动器和移动设备，支持大容量存储设备和大文件。

ext4:

- 当前Linux系统中广泛使用的文件系统，支持更大的文件和文件系统大小，以及多核处理器。

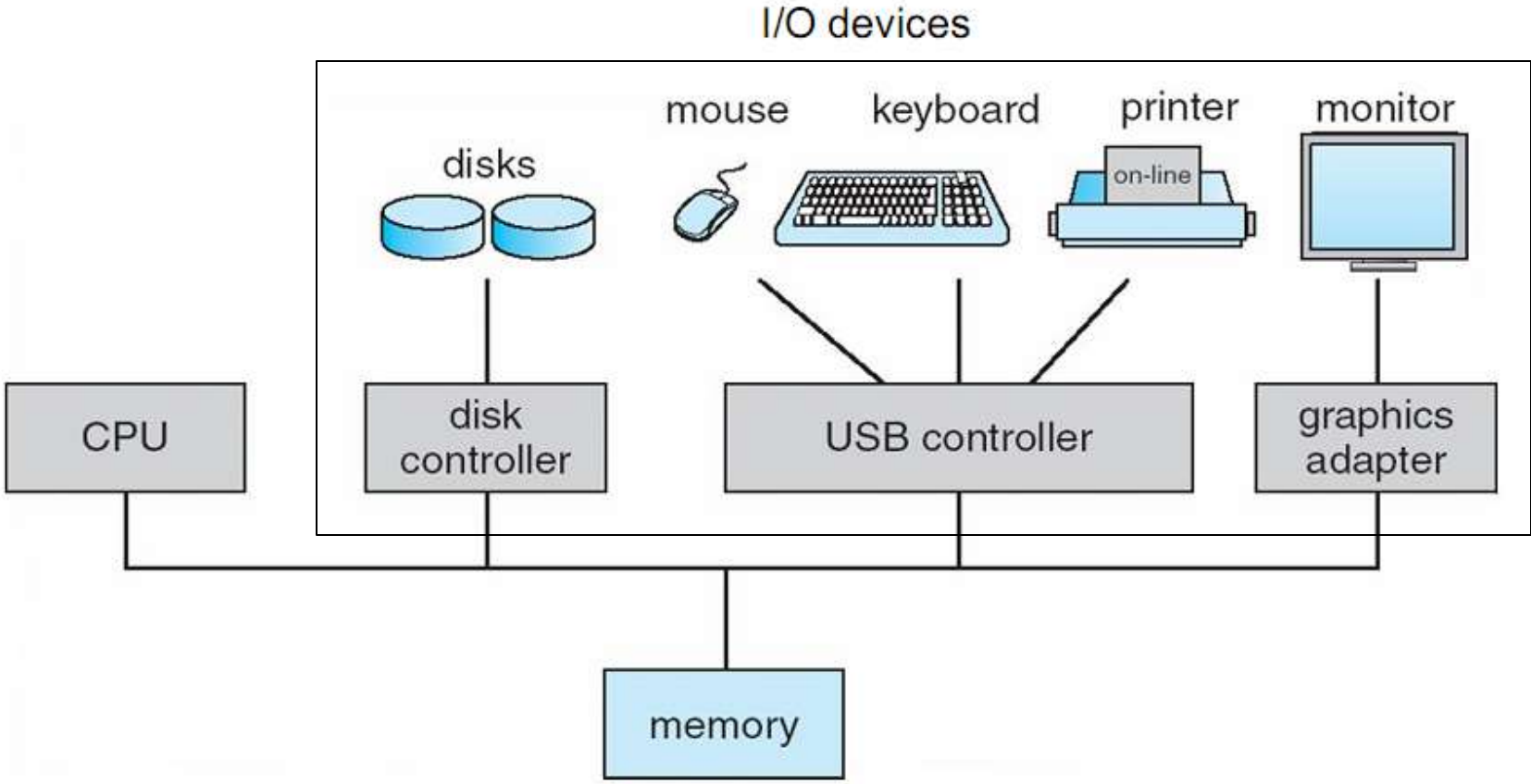
APFS (Apple File System):

- 苹果公司开发的新一代文件系统，用于替代HFS+，提供了更好的性能和安全性。

Functions of operating system

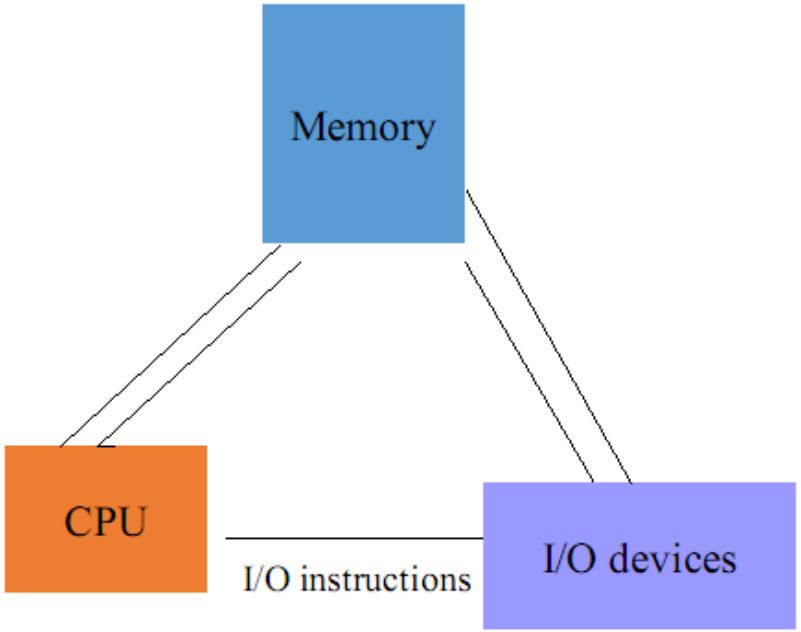
- 设备管理
  - 访问输入输出设备
  - 输入输出设备存在着数量和速度限制
  - 提高设备使用效率

Computer System Organization



Speeding up I/O: Direct Memory Access (DMA)

- Data moved directly between I/O devices and memory
- CPU can work on other tasks



设备管理功能

1. **设备驱动程序**:
   - 操作系统通过设备驱动程序来识别和通信硬件设备。设备驱动程序是操作系统和硬件设备之间的桥梁。
2. **资源分配**:
   - OS负责分配和管理硬件资源，如内存、处理器时间、I/O端口和设备中断。
3. **缓冲管理**:
   - 为了提高数据传输效率，OS会使用缓冲区来暂存从设备传来的数据或要发送到设备的数据。
4. **设备独立性**:
   - 用户和应用程序通常不需要知道硬件设备的具体细节，操作系统提供了设备独立性，允许它们以统一的方式访问不同的设备。
5. **虚拟设备**:
   - 操作系统可以将一个物理设备虚拟化为多个逻辑设备，例如，一个物理硬盘可以被分割为多个逻辑分区。

Functions of operating system

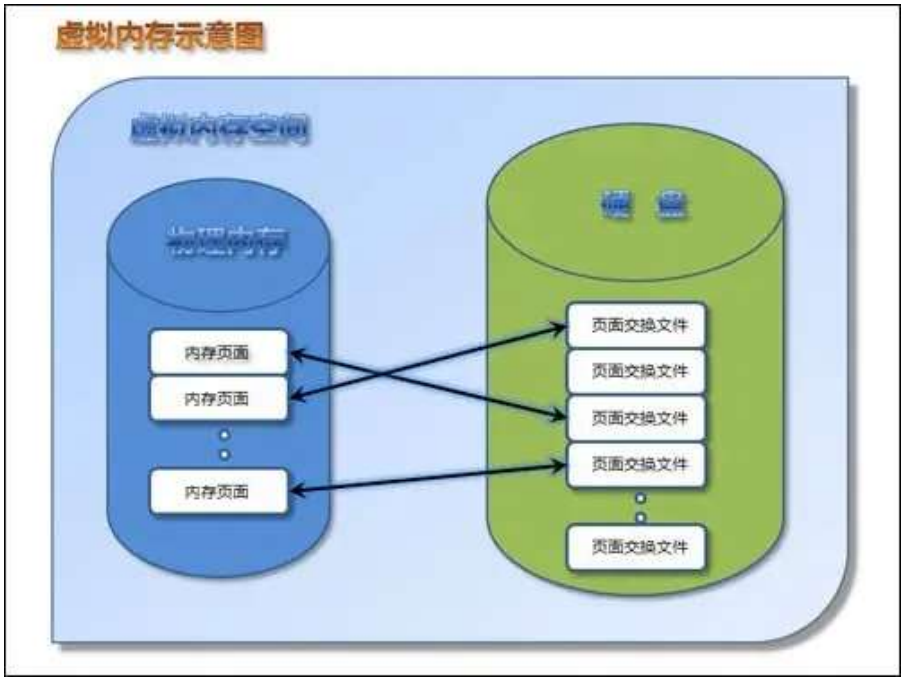
- 内存管理
  - 单道程序
    - 程序需全载入内存
    - 一个程序运行时，其他程序不能运行
  - 多道程序
    - 同一时刻可载入多个程序
    - 模式：非交换、交换
    - 非交换：程序在运行期间始终驻留在内存
    - 交换：运行过程中，程序可以在内存和硬盘间多次交换数据
  - 虚拟内存

- 虚拟内存是计算机系统内存管理的一种技术
- 它使得应用程序认为它拥有连续的可用的内存（一个连续完整的地址空间）
- 实际上，它通常是被分隔成多个物理内存碎片，还有部分暂时存储在外部磁盘存储器上，在需要时进行数据交换
- 目前，大多数操作系统都使用了虚拟内存，如Windows家族的“虚拟内存”；Linux的“交换空间”等

虚拟内存

- VM系统通过将虚拟内存分割为称为虚拟页(Virtual Page，VP)的大小固定的块来处理磁盘和主存之间的传输问题

- 将主存看成是存储在磁盘文件上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传送数据，通过这种方式，它高效地使用了主存



虚拟内存

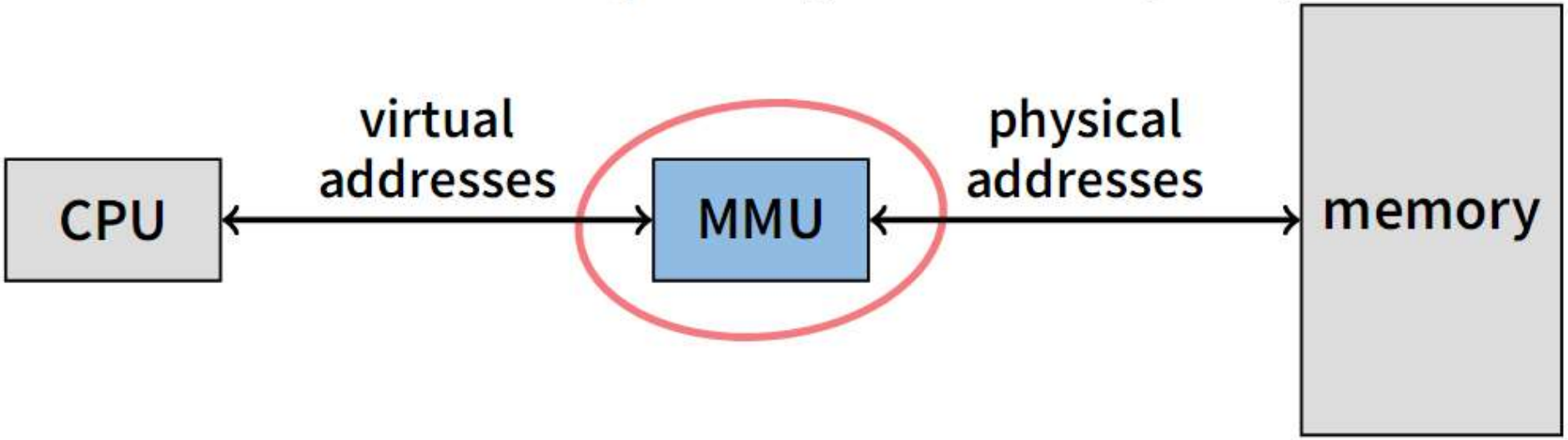
交换技术

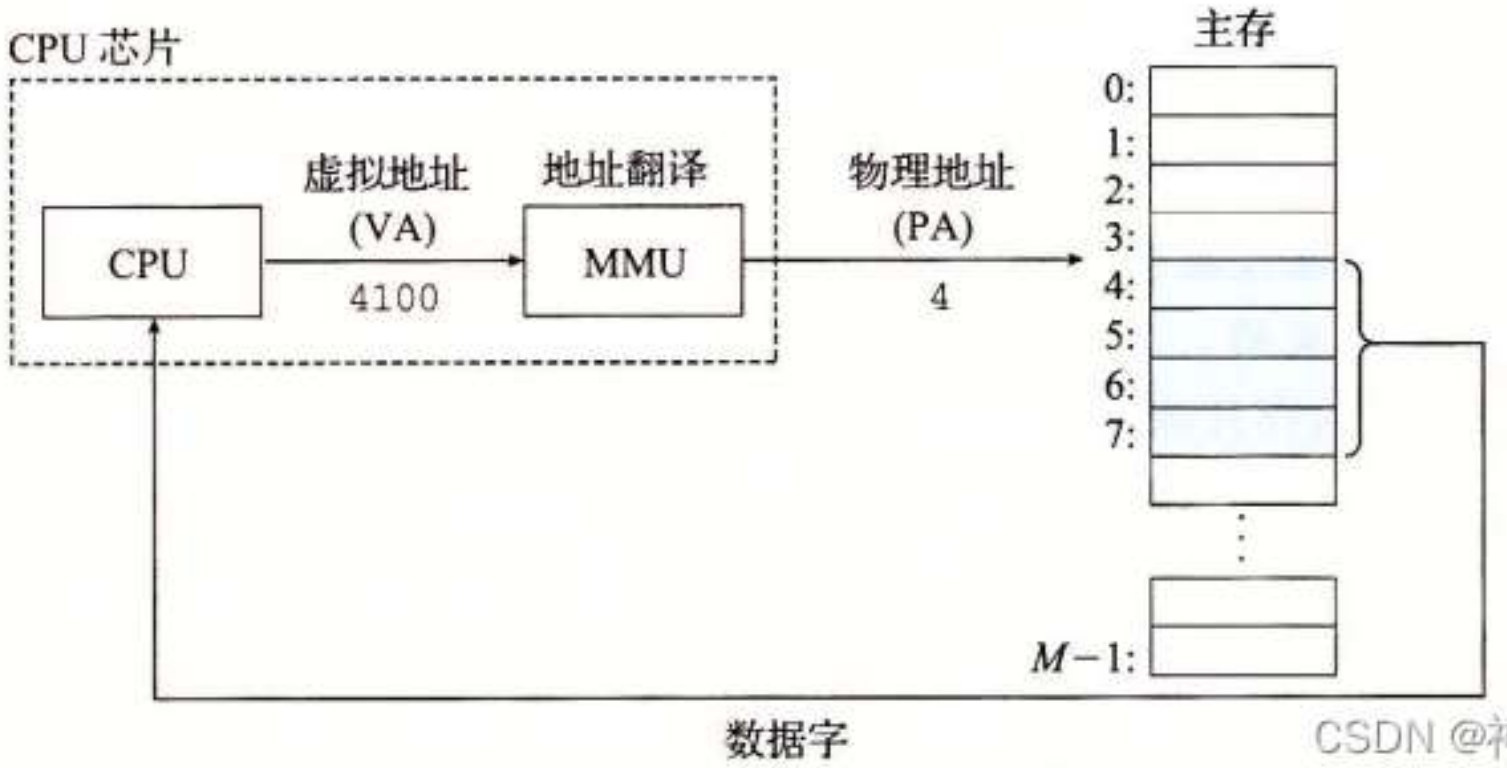
(本图摘自Silberschatz, Galvin and Gagne: "Operating System Concepts")

为何采用虚拟内存

1. **内存扩展**：
   - 允许程序使用比物理内存（RAM）更多的内存。如果应用程序的内存需求超过了物理内存的大小，多余的部分可以临时存储在磁盘上。
2. **内存交换**：
   - 当物理内存不足时，操作系统可以将不活跃的内存页（来自RAM）交换到磁盘上的交换空间或交换文件中，从而为活跃的内存页腾出空间。
3. **内存保护**：
   - 每个进程都有独立的虚拟内存空间，这有助于防止进程间的内存冲突和数据泄露。
4. **内存管理**：
   - 简化了内存管理，因为操作系统可以将虚拟内存视为一大块连续的内存空间，而实际上它可能分散在物理内存和磁盘空间中。

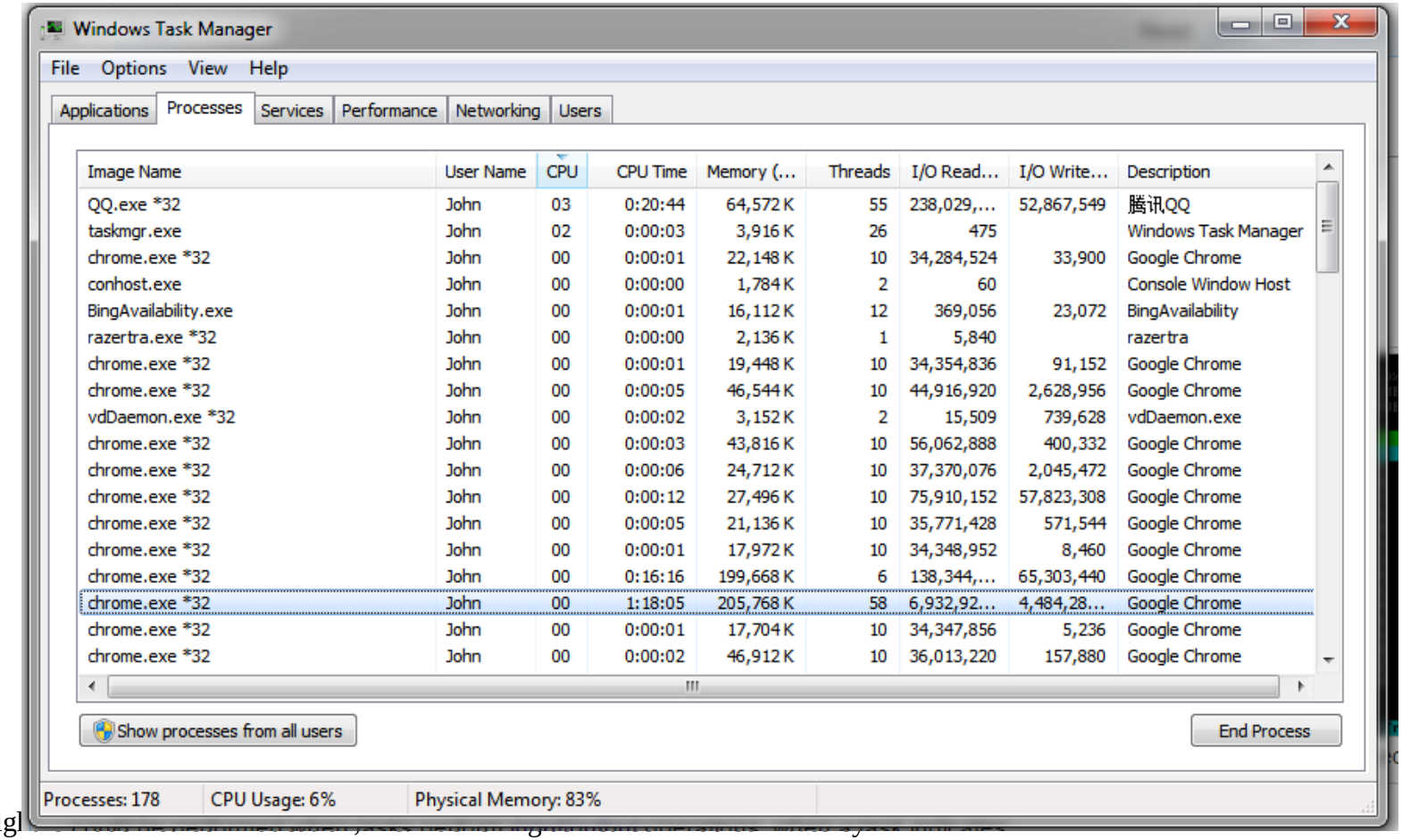
- Programs load/store to virtual addresses
- Actual memory uses physical addresses
- VM Hardware is Memory Management Unit (MMU)





进程管理process management

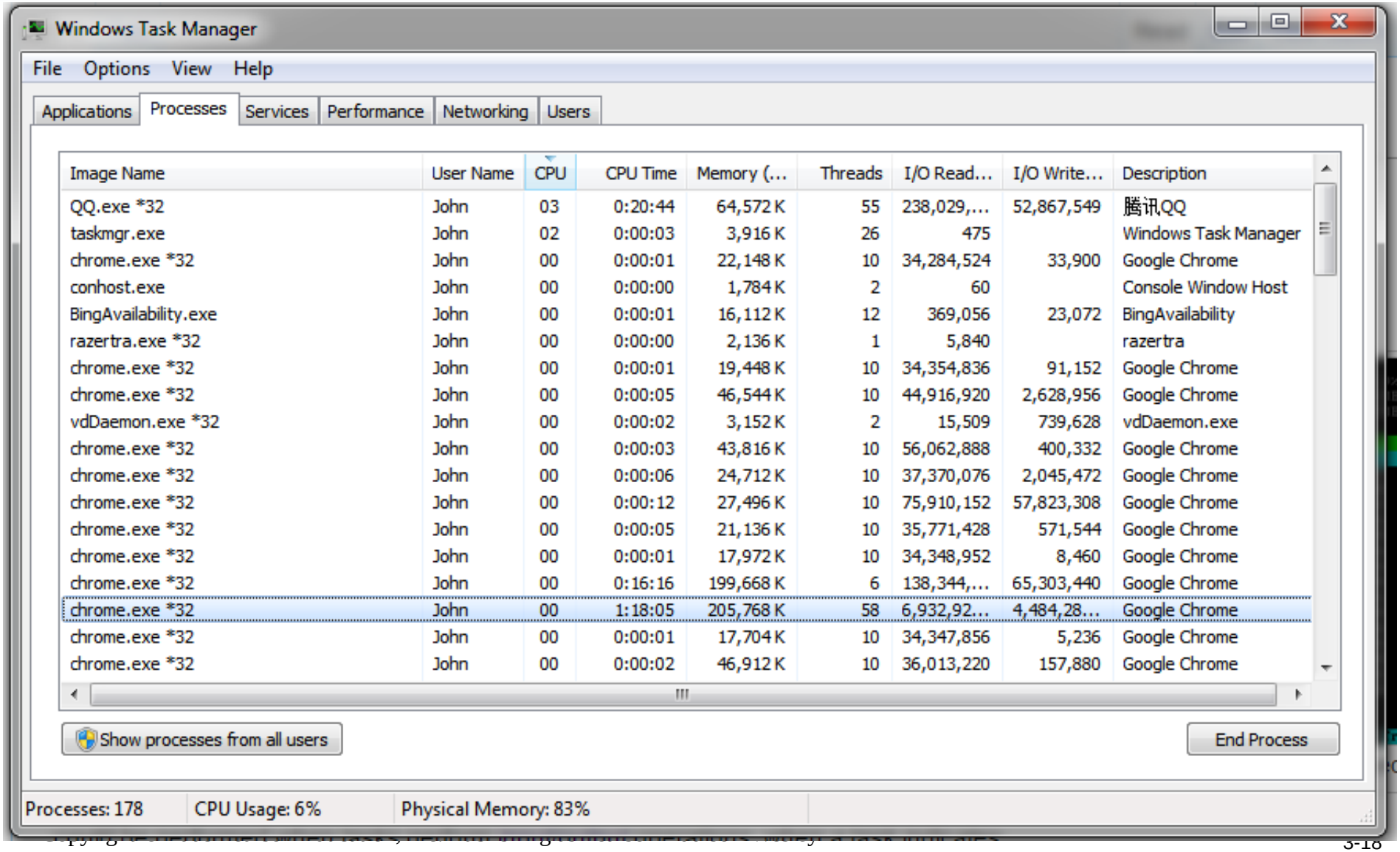
- What is a process



A Program vs. a Process

- Program: a set of instructions, e.g., notepad.c, notepad.exe
- Process: activity of executing a program
- A program can be run multiple times, each instance/activity called a process
- **Process State:** Current status of the activity
  - Program counter
  - General purpose registers
  - Related portion of main memory

Processes


Windows Task Manager
File Options View Help
Applications Processes Services Performance Networking Users
Image Name | User Name | CPU | CPU Time | Memory (... | Threads | I/O Read... | I/O Write... | Description
QQ.exe *32 | John | 03 | 0:20:44 | 64,572 K | 55 | 238,029,... | 52,867,549 | 腾讯QQ
taskmgr.exe | John | 02 | 0:00:03 | 3,916 K | 26 | 475 | | Windows Task Manager
chrome.exe *32 | John | 00 | 0:00:01 | 22,148 K | 10 | 34,284,524 | 33,900 | Google Chrome
conhost.exe | John | 00 | 0:00:00 | 1,784 K | 2 | 60 | | Console Window Host
BingAvailability.exe | John | 00 | 0:00:01 | 16,112 K | 12 | 369,056 | 23,072 | BingAvailability
razertra.exe *32 | John | 00 | 0:00:00 | 2,136 K | 1 | 5,840 | | razertra
chrome.exe *32 | John | 00 | 0:00:01 | 19,448 K | 10 | 34,354,836 | 91,152 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:05 | 46,544 K | 10 | 44,916,920 | 2,628,956 | Google Chrome
vdDaemon.exe *32 | John | 00 | 0:00:02 | 3,152 K | 2 | 15,509 | 739,628 | vdDaemon.exe
chrome.exe *32 | John | 00 | 0:00:03 | 43,816 K | 10 | 56,062,888 | 400,332 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:06 | 24,712 K | 10 | 37,370,076 | 2,045,472 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:12 | 27,496 K | 10 | 75,910,152 | 57,823,308 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:05 | 21,136 K | 10 | 35,771,428 | 571,544 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:01 | 17,972 K | 10 | 34,348,952 | 8,460 | Google Chrome
chrome.exe *32 | John | 00 | 0:16:16 | 199,668 K | 6 | 138,344,... | 65,303,440 | Google Chrome
chrome.exe *32 | John | 00 | 1:18:05 | 205,768 K | 58 | 6,932,92... | 4,484,28... | Google Chrome
chrome.exe *32 | John | 00 | 0:00:01 | 17,704 K | 10 | 34,347,856 | 5,236 | Google Chrome
chrome.exe *32 | John | 00 | 0:00:02 | 46,912 K | 10 | 36,013,220 | 157,880 | Google Chrome
Show processes from all users
End Process
Processes: 178 CPU Usage: 6% Physical Memory: 83%

Process

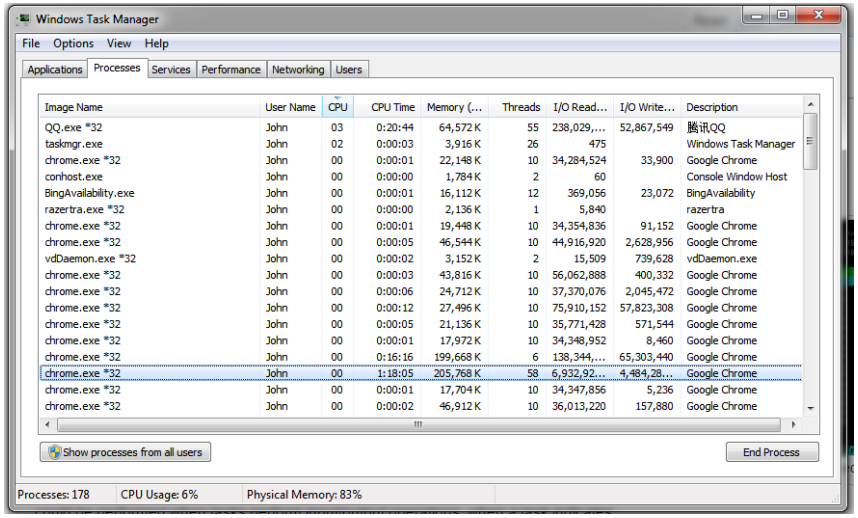
- Each process represents a task
- Jobs vs Tasks
  - A collection of tasks that is used to perform a computation is known as a *job* (MSDN)

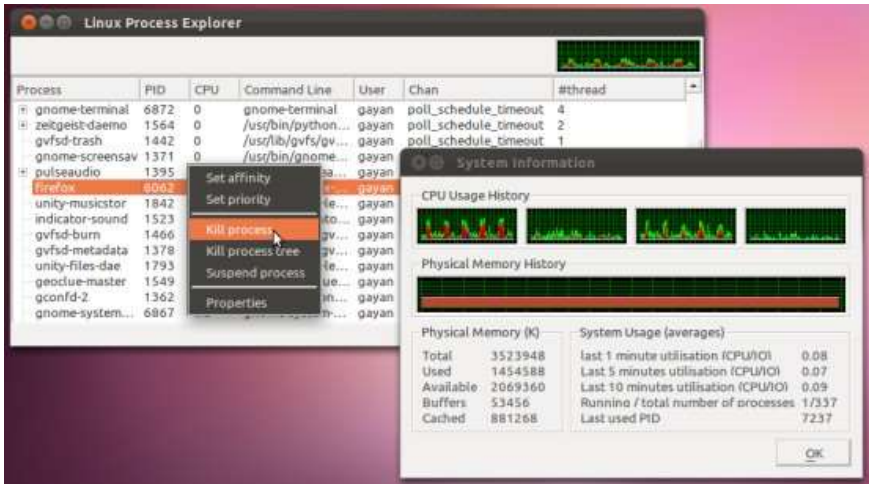
进程管理主要任务

- 在多道程序环境下，计算机内同时运行多个进程
- 进程何时创建、何时撤销、如何管理、如何避免冲突、合理共享就是进程管理的最主要的任务

Process table

- A table of all the processes maintained by the operating system.
- Each entry is a process and its descriptions (PCB).

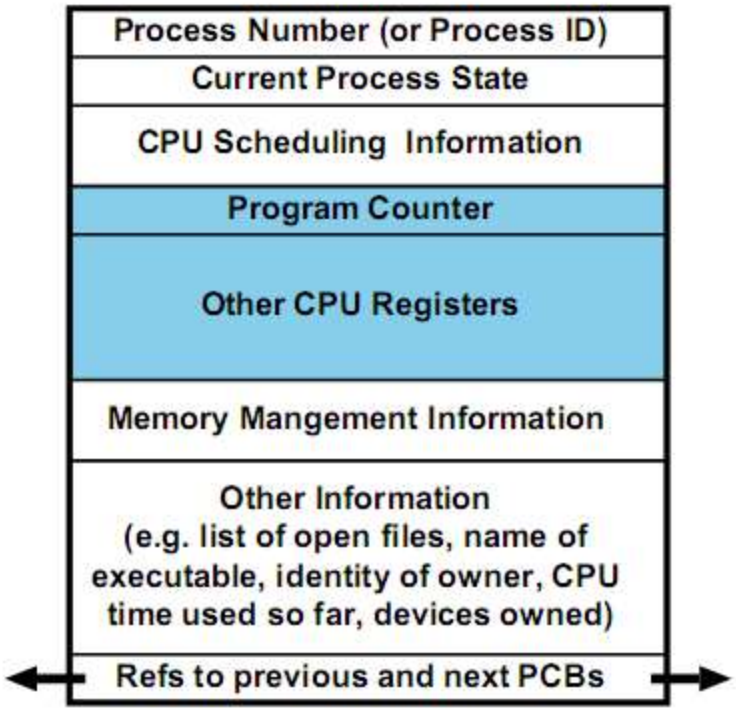




Process control block (PCB)

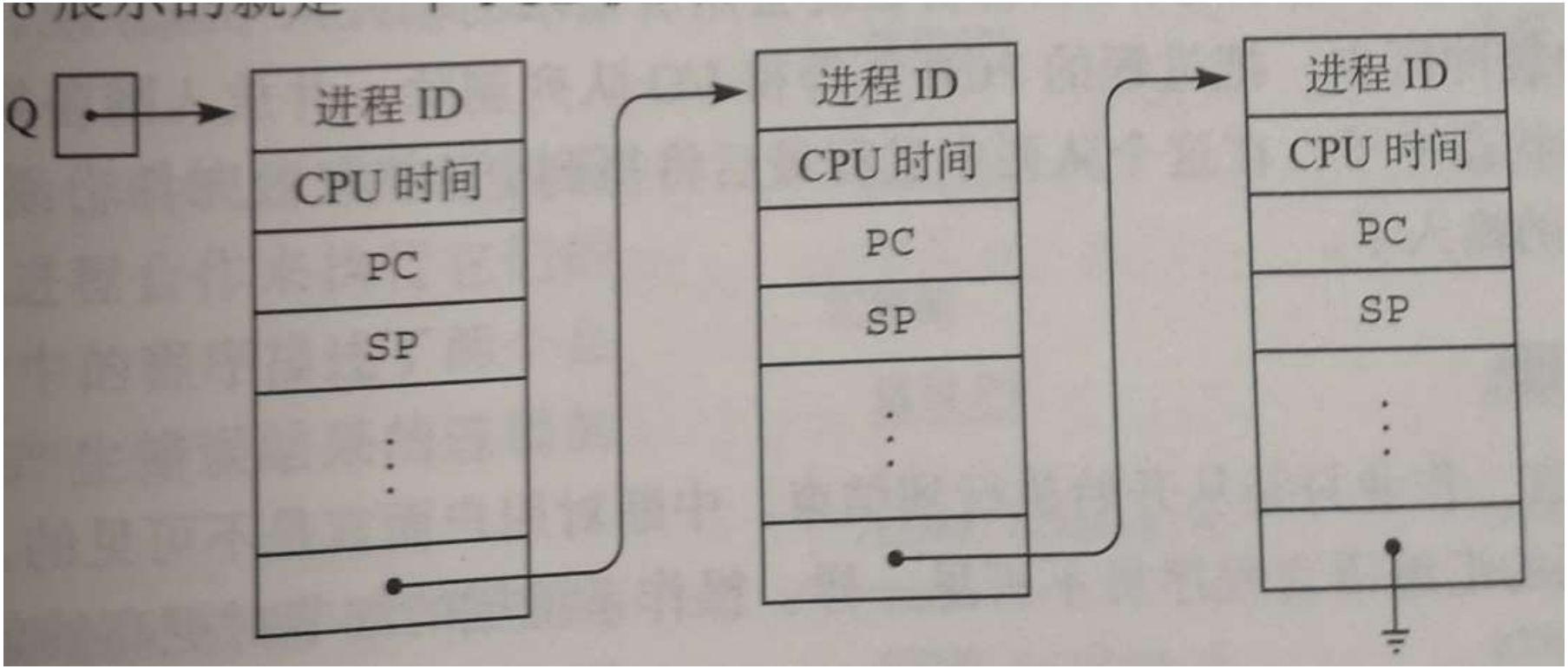
- "the manifestation of a process in an operating system"
  - Process identification
  - Processor state data (the status of a process, saved registers, program counter etc.)
  - Process control data (scheduling state, privileges etc.)

Process Control Block



进程控制块的队列

- 用链表中指针把PCB连接在一起形成队列
- CPU时间:暂停进程目前使用的CPU时间总量



例：ID=782，kill(782)，搜索PCB队列，找对应PCB删除

The secret of concurrent execution

- What do you have to do when switching from one ongoing task to another?
  - Simply stop the current task and turn to another
  - Record the status of the current task and suspend it and turn to another
- Context switch

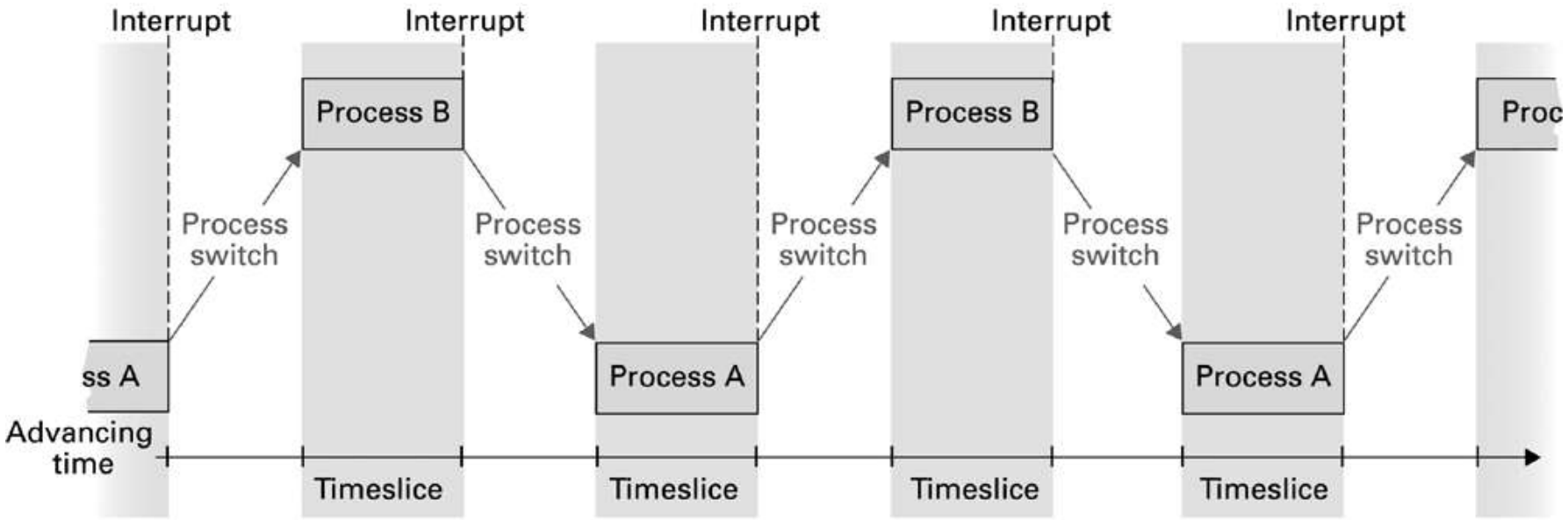
What is a context?

- Snapshot of current status of a process (PCB)
  - A process identifier, or PID
  - Register values, Program Counter value
  - The memory space, I/O, files for the process
  - Can be saved and resumed as if the process is not interrupted
- Another meaning: execution *state* of the process
  - Ready: ready for execution
  - Waiting: waiting for some I/O
  - Complete: finished process

Context switch

- The process of storing and restoring the state (context) of a process so that execution can be resumed from the same point at a later time.
- This enables multiple processes to share a single CPU and is an essential feature of a multitasking operating system.

Figure 3.6 Time-sharing between process A and process B



Who is responsible for context switching?

- **Process management**
  - **Scheduler (调度):** Adds new processes to the process table and removes completed processes from the process table
  - **Dispatcher (分派):** Controls the allocation of time slices to the processes in the process table

When to switch?

- **Interrupt** a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.
- The *interrupt handler* (part of dispatcher) starts after the interrupt to perform context switch
- Modern architectures are interrupt driven.
  - Software interrupt (I/O)
  - Hardware interrupt (press a key)

Scheduler

- Determines which processes should be considered for execution based on some priorities or concerns
  - Using process table for administration
- Process table
  - Ready or waiting
  - Priority
  - Non-scheduling information: memory pages, etc.

Dispatcher

- Gives time slices to a process that is ready
- Executes a context switch when the running process's time slice is over
  - *Time slice*: a time segment for each execution
  - *Interrupt*: the signal generated by a hardware timer to indicate the end of a time slice.
  - The *interrupt handler* (part of dispatcher) starts after the interrupt to perform context switch

Context Switch (process switch)

1. Get an interrupt from timer
2. Go to the interrupt handler
   a. Save the context of process A
   b. Find a process ready to run (Assume that is process B)
   c. Load the context of process B
3. Start (continue) process B

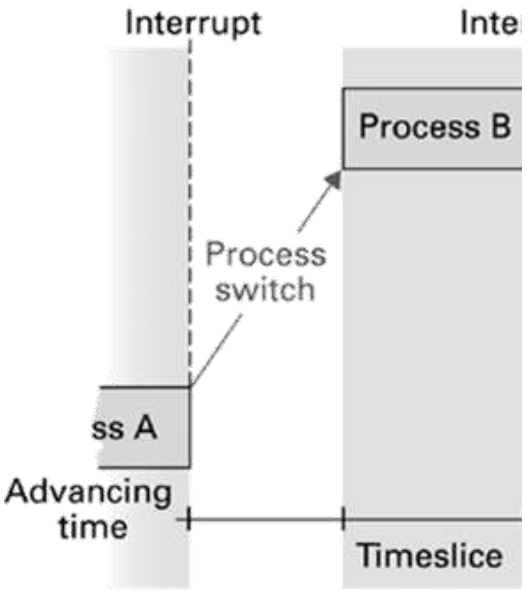
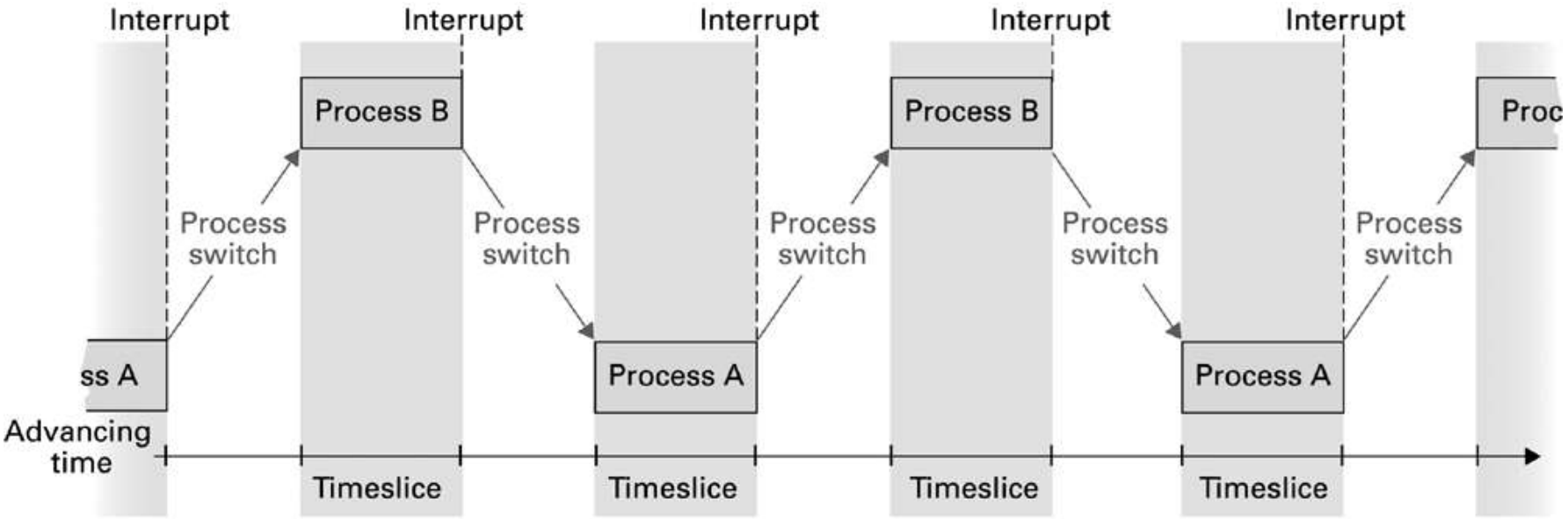


Figure 3.6 Time-sharing between process A and process B



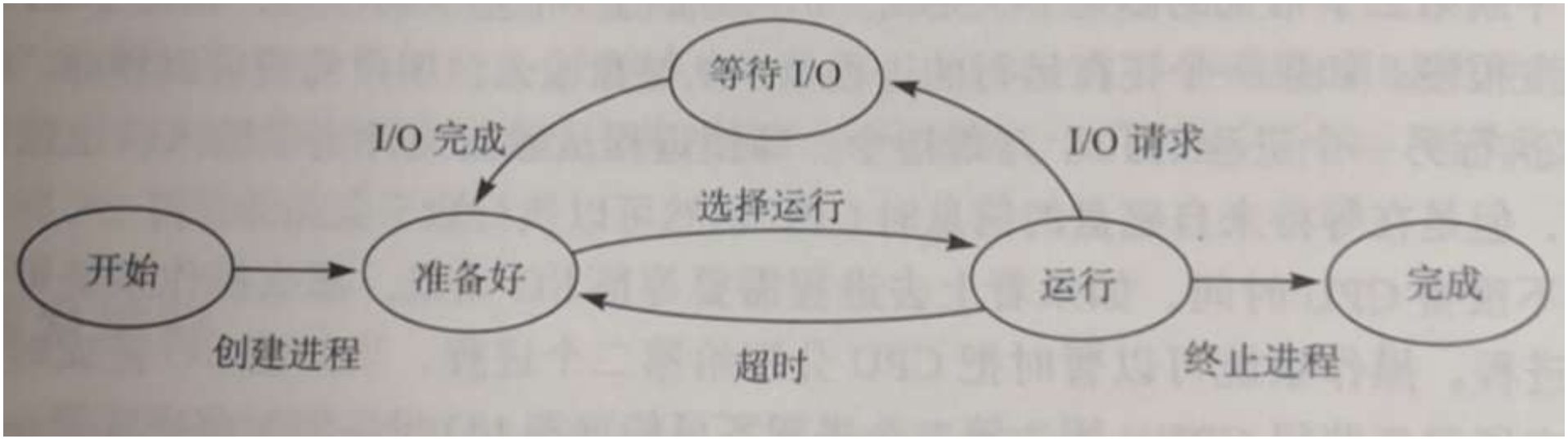
操作系统的并发特征

并行：同一时刻同时进行

- 并发
  - 并发：两个或多个事件在同一时间间隔内发生。
  - 操作系统的并发性是指计算机系统中同时存在多个运行的程序，因此它具有处理和调度多个程序同时执行的能力。
  - 在多道程序环境下，一段时间内， 宏观上有多道程序在同时执行，而在每个时刻，单处理机环境下实际仅能有一道程序执行，因此微观上这些程序仍是分时交替执行的。操作系统的并发性是通过分时得以实现的

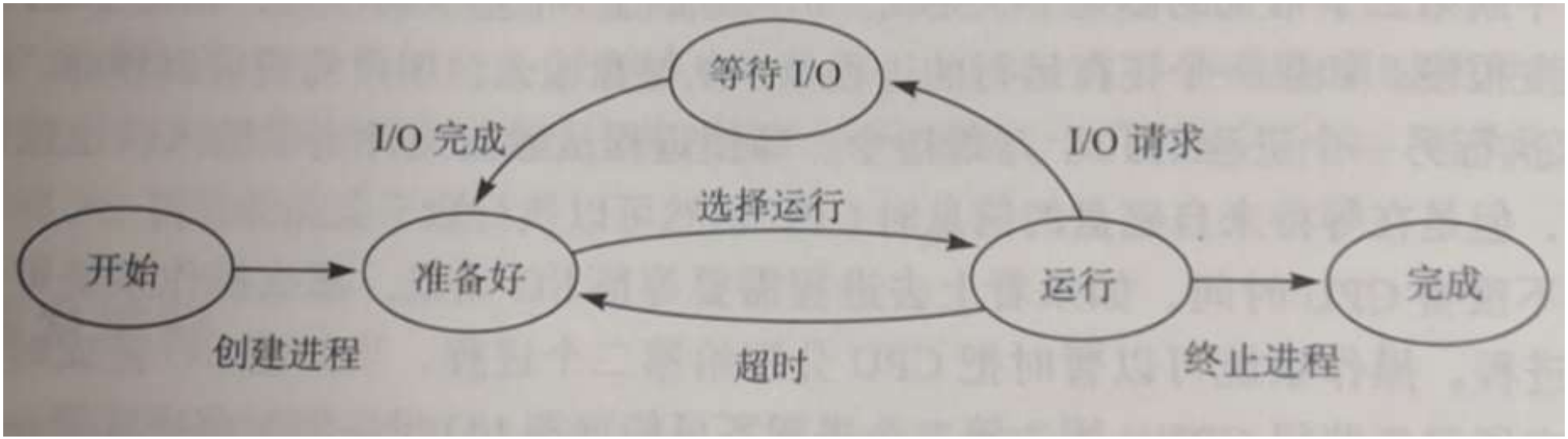
我们以现实生活中的直观例子来认识并发和并行的区别。例如，如果你在 9:00～9:10 仅吃面包，在 9:10～9:20 仅写字，在 9:20～9:30 仅吃面包，在 9:30～10:00 仅写字，那么在 9:00～10:00 吃面包和写字这两种行为就是并发执行的；再如，如果你在 9:00～10:00 右手写字，左手同时拿着面包吃，那么这两个动作就是并行执行的。

操作系统中作业的状态转换图



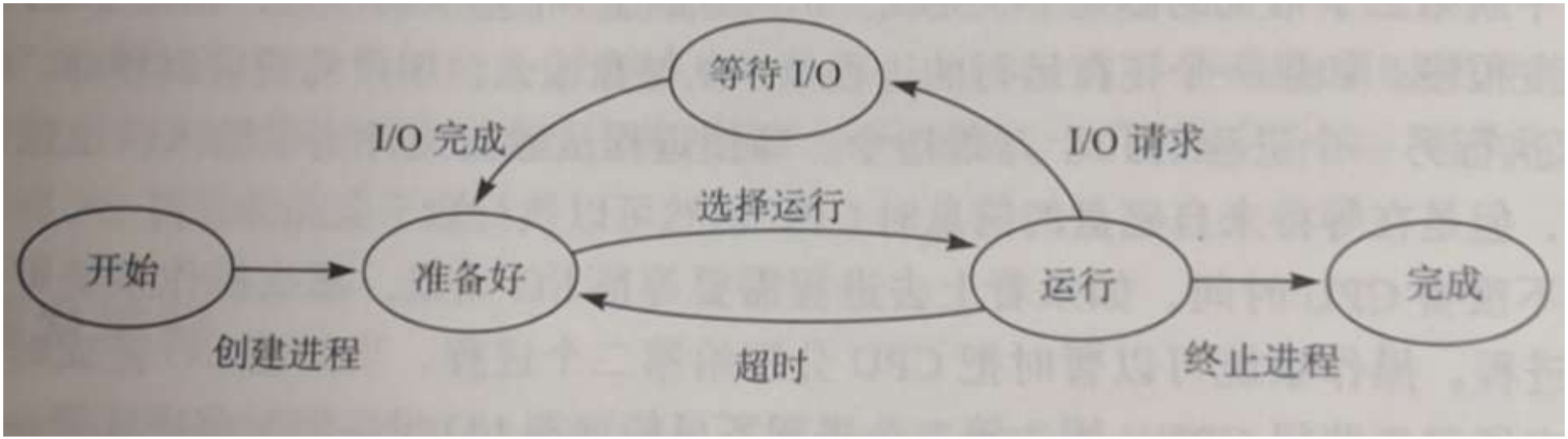
- 用户提交作业，OS生成进程，分配新的PCB
- 将PCB加入等待CPU的进程队列，把程序装载到主存
- 把PCB中PC的副本设置为进程第一条指令地址
- 作业处于ready状态

操作系统中作业的状态转换图



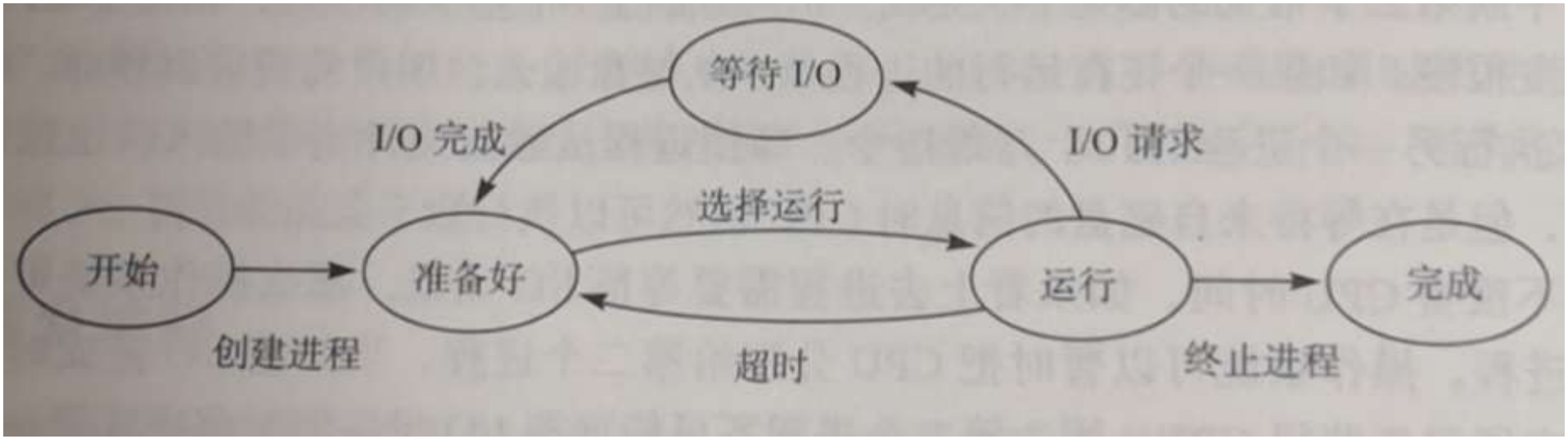
- 中断产生后，OS把寄存器副本从PCB放入CPU
- 作业处于运行状态

操作系统中作业的状态转换图



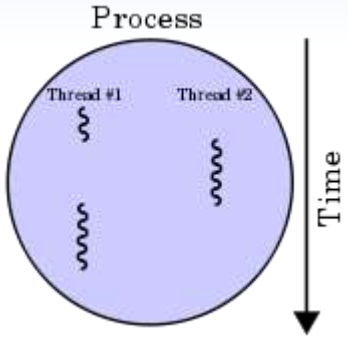
- 在运行状态会发生3类事件
  - 超时：时间片结束，PCB放入准备好队列，进入ready状态
  - 完成：执行最后一条指令请OS终止进程
  - I/O请求

操作系统中作业的状态转换图



- I/O请求
  - OS把请求转给对应I/O设备
  - 把PCB放入等待I/O完成队列，程序在等待I/O状态
  - I/O操作后，I/O设备中断系统，系统把数据放在内存缓冲区
  - 将进程PCB从等待I/O队列删除，放入ready队列，进程进入ready状态

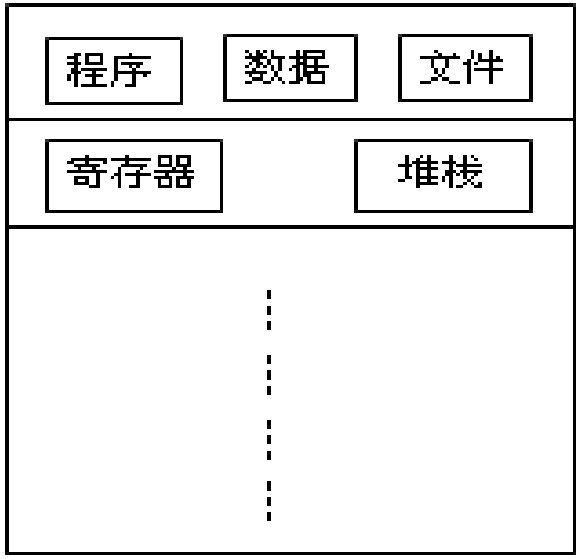
Thread线程



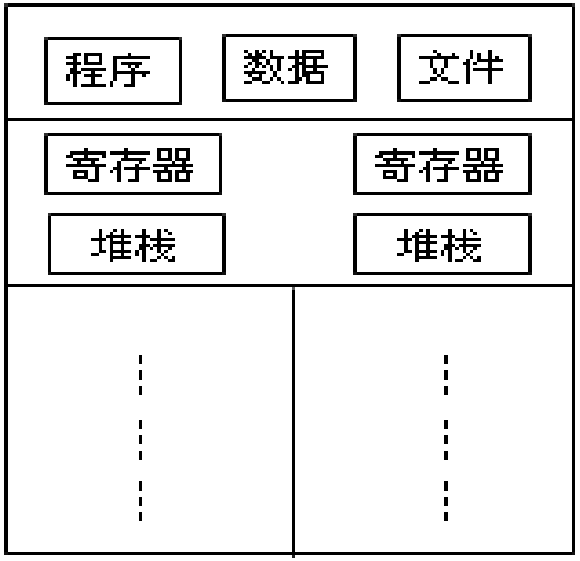
- A task existing within a process that allows multiple independent instances to be executed concurrently
  - Multiple threads share resources such as memory, program code, …
  - Each thread has its own program counter, registers, and stack (local memory)
- The context switch of threads is much faster than that of processes

- 引入进程的目的是更好地使多道程序并发执行，提高资源利用率和系统吞吐量
- 引入线程的目的则是减小程序在并发执行时所付出的时空开销，提高操作系统的并发性能。

A Thread vs. a Process



单线程



多线程

Share more

同一进程中的多个线程之间可以并发执行

High Concurrence

Fast Switch

由于有了线程，线程切换时，有可能会发生进程切换，也有可能不发生，平均而言每次切换所需的开销就变小了，因此能够让更多的线程参与并发，而不会影响到响应时间等问题

Exercises

- Suppose an OS allocates time slices in 10 millisecond units and the time required for a context switch is negligible. How many processes can obtain a time slice in one second?

Competition for Resources

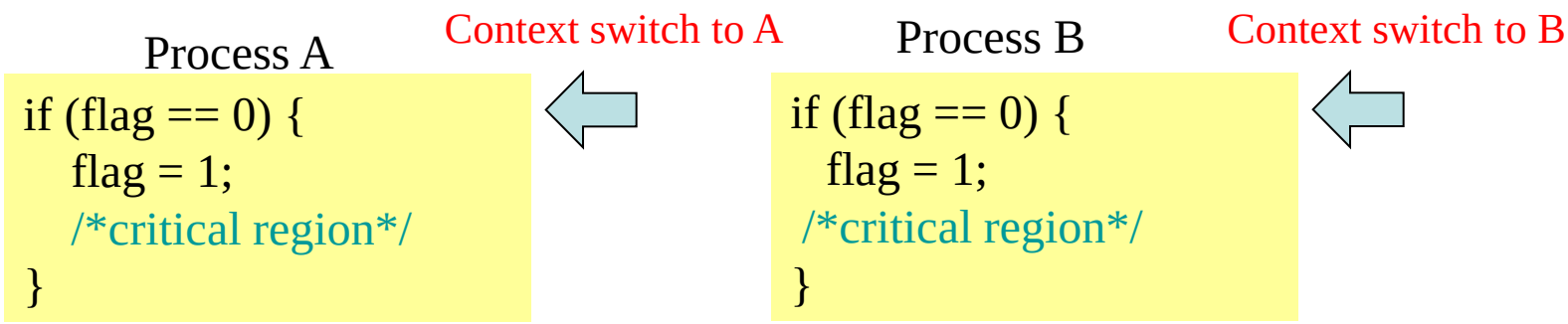
- What are *resources*?
  - CPU, memory, files, peripheral devices, …
- In a multitasking system, resources are shared by processes
  - Some resources should not be employed by more than one process simultaneously
  - E.g., printer

Handling Competition for Resources

- **Semaphore信号量:** A “control flag”
- **Critical Region临界区:** A group of instructions that should be executed by only one process at a time
- **Mutual exclusion互斥:** Requirement for proper implementation of a critical region

First Algorithm

- Use a flag (a global memory address)
  - flag=1: the critical region is occupied
  - flag=0: no process is in the critical region
- Problem:



  - Both processes get into the critical region

Solutions

- Testing&setting the flag must be completed without interruption

1. Use disable_Interrupt() to prevent context switch during the flag test and set process.

```
Diable_Interrupt();
if (flag == 0) {
    flag = 1;
    Enable_Interrupt();
    / *critical region*/
}
Enable_Interrupt();
```

2. A machine instruction called “test-and-set” which cannot be interrupted

- Semaphore: a properly implemented flag

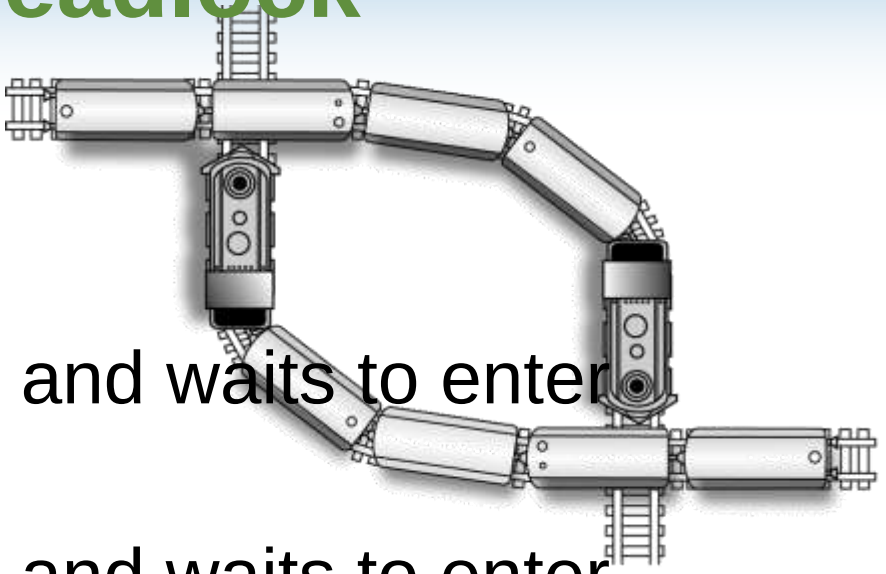
现代操作系统中，不能进行进程的调度与切换的情况有以下几种：

1）在处理中断的过程中。中断处理过程复杂，在实现上很难做到进程切换，而且中断处理是系统工作的一部分，逻辑上不属于某一进程，不应被剥夺处理机资源。

2）进程在操作系统内核临界区中。进入临界区后，需要独占式地访问，理论上必须加锁，以防止其他并行进程进入，在解锁前不应切换到其他进程，以加快临界区的释放。

3）其他需要完全屏蔽中断的原子操作过程中。如加锁、解锁、中断现场保护、恢复等原子操作。在原子过程中，连中断都要屏蔽，更不应该进行进程调度与切换。

Another Problem: Deadlock



- Example:
  - A is in critical region 1, and waits to enter critical region 2
  - B is in critical region 2, and waits to enter critical region 1

Process A

```
if (test_set(flag1)) {
    /*critical region 1*/
   while(!test_set(flag2));
   /*critical region 2*/
 }
```

Context switch to A

Process B

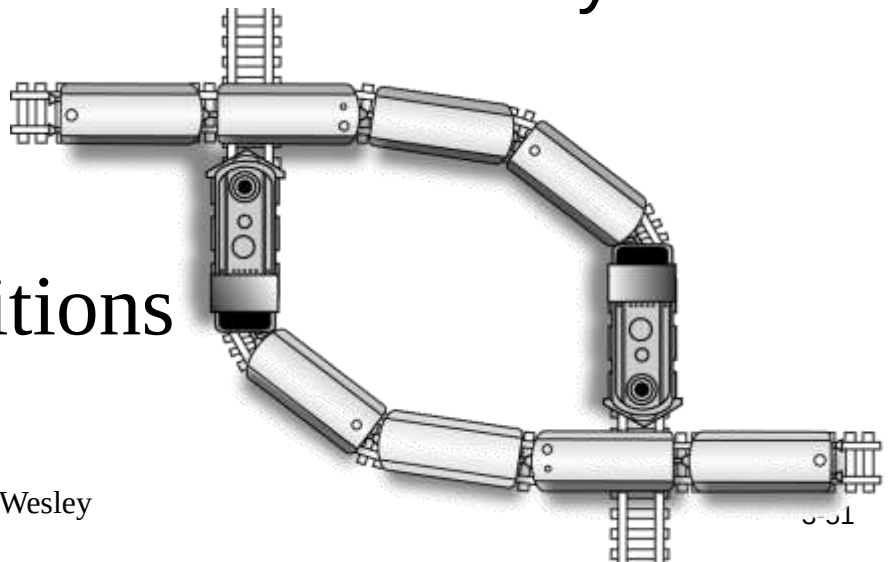
```
if (test_set(flag2)) {
    /*critical region 2*/
   while (!test_set(flag1));
     /*critical region 1*/
   }
```

Context switch to B

死锁是指多个进程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进

Deadlock

- Processes block each other from continuing
- Conditions required for deadlock
  1. Competition for non-sharable resources
  2. Resources requested on a partial basis
  3. An allocated resource can not be forcibly retrieved



Remove any one of the conditions can resolve the deadlock.

Solutions

Which condition is removed?

1. Kill one of the process
2. Processes need to request all the required resources at one time
3. Spooling
   - For example, stores the data to be printed and waits the printer available
4. Divide a file into pieces so that it can be altered by different processes

Exercises

- There is a bridge that only allows one car to pass. When two cars meet in the middle, it causes “deadlock”. The following solutions remove which conditions
  - Do not let a car onto the bridge until the bridge is empty.
  - If cars meet, make one of them back up.
  - Add a second lane to the bridge.
- What’s the drawback of solution 1?

操作系统的共享特征

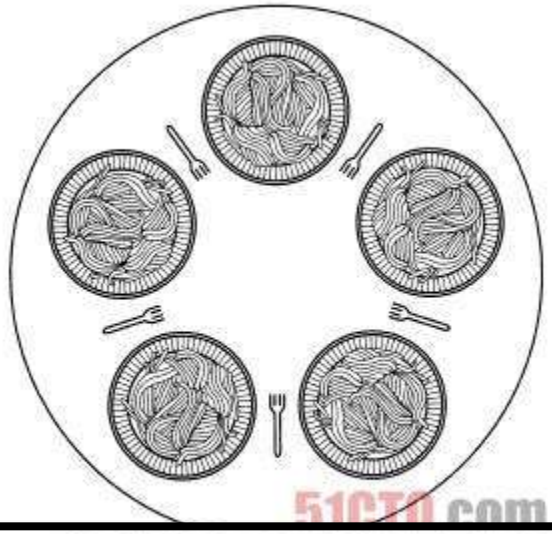
- 资源可供内存中多个并发执行的进程共同使用
- 两种方式：互斥共享 + 同时访问
- 互斥共享
  - 在一段时间内只允许一个进程访问该资源（临界资源）
  - 仅当某进程访问完并释放该资源后，才允许另一个进程对该资源进行访问（如：打印机）
- 同时访问
  - 资源允许一段时间内由多个进程“同时”访问，通常是宏观上同时，在微观上，进程可能交替地对该资源进行访问即“分时共享"。
  - 一个请求分几个时间片段间隔地完成
  - 可供多个进程“同时”访问的典型资源是磁盘设备

Dinning philosophers

Five philosophers are sitting at a round table. In front of each is a plate of spaghetti. There are five forks on the table, one between each plate. Each philosopher wants to alternate between thinking and eating. To eat, a philosopher requires possession of both the forks that are adjacent to the philosophers plate. Identify the possibility of deadlocks and starvation that are present in this problem.

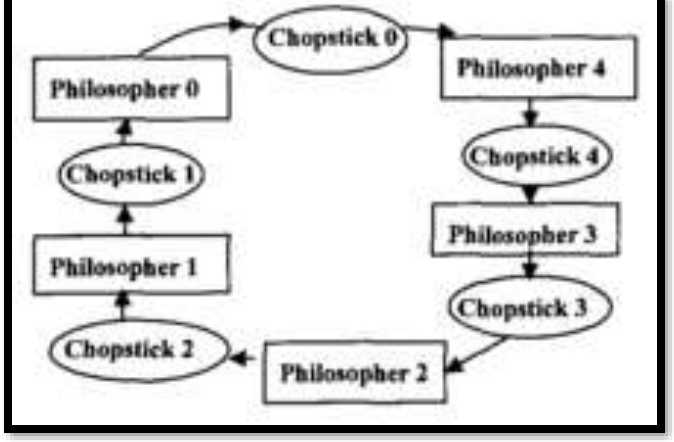
Starvation: a process that is waiting for a time slice is said to suffer starvation if it is never given a time slice.
如果等待时间过长，则导致进程使命没有意义

在1971年，著名的计算机科学家艾兹格·迪科斯彻提出了一个同步问题，即假设有五台计算机都试图访问五份共享的磁带驱动器。稍后，这个问题被托尼·霍尔重新表述为哲学家就餐问题。这个问题可以用来解释死锁和资源耗尽。



一组进程处于死锁状态是指组内的每个进程都在等待一个事件，而该事件只可能由组内的另一个进程产生。这里所关心的主要是事件是资源的获取和释放。与死锁相关的另一个问题是无限期阻塞（Indefinite Blocking）或饥饿（Starvation），即进程在信号量内无穷等待的情况。

- 设有5个哲学家，共享一张放有5把椅子的桌子，每人分得一把椅子，但是，桌子上共有5只筷子，在每人两边各放一只，哲学家们在肚子饥饿时才试图分两次从两边拿起筷子就餐。
  条件:
  1）拿到两只筷子时哲学家才开始吃饭。
  2）如果筷子已在他人手上，则该哲学家必须等他人吃完之后才能拿到筷子。
  3）任一哲学家在自己未拿到两只筷子前却不放下自己手中的筷子。



- 五位哲学家只思考和吃饭并不交谈，若同时拿起左面的叉子/筷子，就没有人能够拿到他们右面的叉子，于是发生死锁。
- 防止死锁的方法：
- （1）规定在拿到左侧的筷子后，先检查右面的筷子是否可用。如果不可用，则先放下左侧筷子， 等一段时间再重复整个过程。如果同时拿起左边筷子，则右边的筷子都不可用，则放下，然后再次拿起，。。。，否则谁都无法就餐，

  问题：发生饥饿现象(starvation)
- （2）最多允许四个哲学家同时进餐,以保证至少有一个哲学家能够进餐,最终总会释放出他所使用过的两支筷子,从而可使更多的哲学家进餐。
- （3）将拿左筷子，与拿右筷子当做一个原子操作：（即只有当左右筷子均可以拿到时，才拿筷子。）
- （4）规定奇数号的哲学家先拿起他左边的筷子,然后再去拿他右边的筷子;而偶数号的哲学家则相反， 最后总会有一个哲学家能获得两支筷子而进餐。而申请不到的哲学家进入阻塞等待队列，根FIFO原则，则先申请的哲学家会较先可以吃饭，因此不会出现饿死的哲学家