1）OS

一、定义和功能

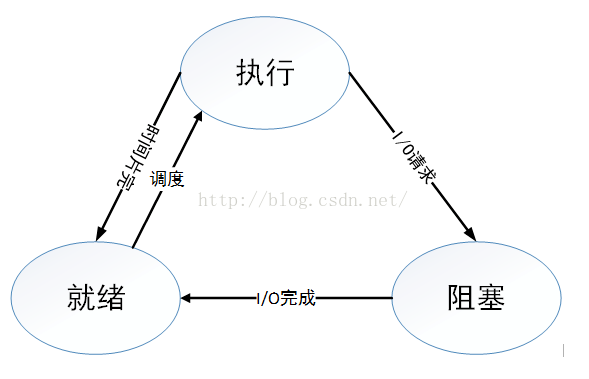
操作系统是管理和控制计算机硬件与软件资源的计算机程序，功能包括管理计算机系统的硬件、软件及数据资源，控制程序运行，改善人机界面，为其它应用软件提供支持，让计算机系统所有资源最大限度地发挥作用，提供各种形式的用户界面，使用户有一个好的工作环境，为其它软件的开发提供必要的服务和相应的接口等。

操作系统主要有五大功能：处理机管理（CPU）、进程管理、内存管理、设备管理和文件系统管理。

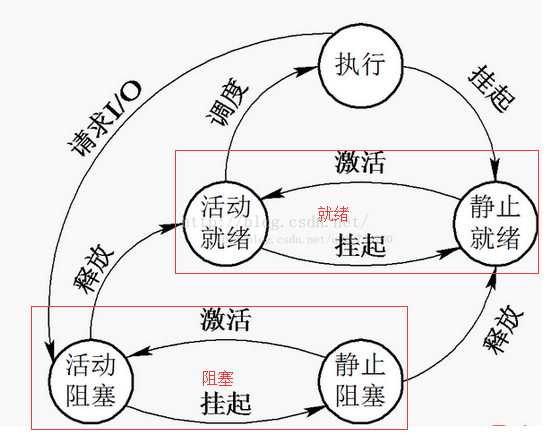
二、调度（进程和线程）

进程是用户提交给操作系统运行的最小单元。进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，是系统进行资源调度和分配的一个独立单位。除了进程，操作系统还提供了更小粒度的调度对象-线程。线程是进程的一个实体，是CPU调度和分派的基本单位，它是比进程更小的能独立运行的基本单位。一个进程可以有多个线程，多个线程也可以并发执行。

进程的内存布局：代码块、数据区、堆、栈段。



进程的三种基本状态转换图



具有挂起状态的进程状态转换图

**进程和线程的区别**：资源分配给进程，同一进程的所有线程共享该进程的所有资源。 同一进程中的多个线程共享代码段(代码和常量)，数据段(全局变量和静态变量)，扩展段(堆存储)。但是每个线程拥有自己的栈段（Java虚拟机栈），用来存放所有局部变量和临时变量、程序计数器（字节码行号指示器）、线程id（标识此线程）、寄存器组的值（原有的线程的寄存器集合的状态保存，以便将来该线程被重新切换到时能得以恢复）。

协程：协程和线程一样共享堆，不共享栈，协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文和栈，直接操作栈（在一个线程内执行）则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。

**协程和线程的区别**是：协程避免了无意义的调度，由此可以提高性能，但也因此，程序员必须自己承担调度的责任，同时，协程也失去了标准线程使用多CPU的能力。

一般情况下，系统按照以下流程创建一个进程：

1. 分配、初始化 PCB
2. 初始化机器寄存器
3. 拷贝、初始化内存页表
4. 从硬盘加载程序代码到内存
5. 将进程加入就绪队列
6. 进程调度时，选择该进程并切换到用户态开始执行进程

系统通过快速切换进程，让每一个进程都有一定的时间片来响应用户提交的请求；在用户的视角，好像每个进程都在同时执行一样。系统切换进程的方法叫做进程调度算法，基本的调度算法有：

（1）先来先服务（FCFS）

算法简单，但效率低；对长作业比较有利，但对短作业不利（相对SJF和高响应比）；有利于CPU繁忙型作业，而不利于I/O繁忙型作业。

（2）时间片轮转

时间片轮转调度算法主要适用于分时系统。在这种算法中，系统将所有就绪进程按到达时间的先后次序排成一个队列，进程调度程序总是选择就绪队列中第一个进程执行，即先来先服务的原则，但仅能运行一个时间片，如100ms。在使用完一个时间片后，即使进程并未完成其运行，它也必须释放出（被剥夺）处理机给下一个就绪的进程，而被剥夺的进程返回到就绪队列的末尾重新排队，等候再次运行。

在时间片轮转调度算法中，时间片的大小对系统性能的影响很大。如果时间片足够大，以至于所有进程都能在一个时间片内执行完毕，则时间片轮转调度算法就退化为先来先服务调度算法。如果时间片很小，那么处理机将在进程间过于频繁切换，使处理机的开销增大，而真正用于运行用户进程的时间将减少。因此时间片的大小应选择适当。

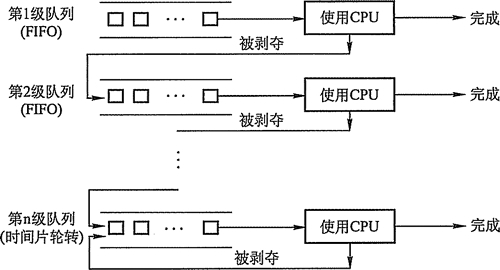
（3）短作业优先（SJF）平均等待时间、平均周转时间最少

* 该算法对长作业不利。更严重的是，如果有一长作业进入系统的后备队列，由于调度程序总是优先调度那些 (即使是后进来的）短作业，将导致长作业长期不被调度（“饥饿”现象，注意区分“死锁”。后者是系统环形等待，前者是调度策略问题）。
* 该算法完全未考虑作业的紧迫程度，因而不能保证紧迫性作业会被及时处理。
* 由于作业的长短只是根据用户所提供的估计执行时间而定的，而用户又可能会有意或无意地缩短其作业的估计运行时间，致使该算法不一定能真正做到短作业优先调度。

（4）优先级调度

* 非剥夺式优先级调度算法。当某一个进程正在处理机上运行时，即使有某个更为重要或紧迫的进程进入就绪队列，仍然让正在运行的进程继续运行，直到由于其自身的原因而主动让出处理机时（任务完成或等待事件），才把处理机分配给更为重要或紧迫的进程。
* 剥夺式优先级调度算法。当一个进程正在处理机上运行时，若有某个更为重要或紧迫的进程进入就绪队列，则立即暂停正在运行的进程，将处理机分配给更重要或紧迫的进程。

（5）多级反馈队列调度（优先级+时间片轮转）



算法的实现思想：

1. 应设置多个就绪队列，并为各个队列赋予不同的优先级，第1级队列的优先级最高，第2级队列次之，其余队列的优先级逐次降低。
2. 赋予各个队列中进程执行时间片的大小也各不相同，在优先级越高的队列中，每个进程的运行时间片就越小。例如，第2级队列的时间片要比第1级队列的时间片长一倍， ……第i+1级队列的时间片要比第i级队列的时间片长一倍。
3. 当一个新进程进入内存后，首先将它放入第1级队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第2级队列的末尾，再同样地按FCFS 原则等待调度执行；如果它在第2级队列中运行一个时间片后仍未完成，再以同样的方法放入第3级队列……如此下去，当一个长进程从第1级队列依次降到第 n 级队列后，在第 n 级队列中便釆用时间片轮转的方式运行。
4. 仅当第1级队列为空时，调度程序才调度第2级队列中的进程运行；仅当第1 ~ (i-1)级队列均为空时，才会调度第i级队列中的进程运行。如果处理机正在执行第i级队列中的某进程时，又有新进程进入优先级较高的队列（第 1 ~ (i-1)中的任何一个队列），则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i级队列的末尾，把处理机分配给新到的更高优先级的进程。

多级反馈队列的优势有：

* + 终端型作业用户：短作业优先。
  + 短批处理作业用户：周转时间较短。
  + 长批处理作业用户：经过前面几个队列得到部分执行，不会长期得不到处理。

三、通信

线程之间共享内存，但拥有各自不同的运行栈；进程之间则相互隔离。线程之间并发需要解决的是线程同步问题，进程之间则是通信问题。

临界区：每个进程中访问临界资源的那段代码称为临界区（Critical Section）（临界资源是一次仅允许一个进程使用的共享资源）。

**3.1 线程同步**

* 原子操作

需要使用线程同步的根本原因在于对普通变量的操作不是原子的，如果操作是原子操作就不用同步进行保证了。

* synchronize / volatile关键字
* 信号量

当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使用“信号量”对象。提供对临界资源的安全分配。如果存在多份临界资源，在多个线程争抢临界资源的情况下，向线程提供安全分配临界资源的方法。如果临界资源的数量为1，将退化为锁。

* wait / notify机制（条件变量）

提供线程之间的一种通知机制，当某一条件满足时，线程A可以通知阻塞在条件变量上的线程B，B所期望的条件已经满足，可以解除在条件变量上的阻塞操作，继续做其他事情。

* 局部变量实现线程同步

使用 ThreadLocal 管理变量，则每一个使用该变量的线程都获得该变量的副本，副本之间相互独立，这样每一个线程都可以随意修改自己的变量副本，而不会对其他线程产生影响。

* 使用阻塞队列实现线程同步

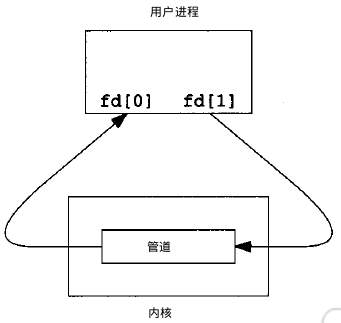
**3.2 进程通信（**IPC，InterProcess Communication**）**

进程之间通信常用的方式有：

**3.2.1 管道**

**1）无名管道**

管道，通常指无名管道，是 UNIX 系统IPC最古老的形式。当一个管道建立时，它会创建两个文件描述符：fd[0]为读文件描述符，fd[1]为写文件描述符。



**特点：**

（1）它是半双工的（即数据只能在一个方向上流动），具有固定的读端和写端。

（2）它只能用于具有亲缘关系的进程之间的通信（也是父子进程或者兄弟进程之间）。

（3）它可以看成是一种特殊的文件，对于它的读写也可以使用普通的read、write 等函数。但是它不是普通的文件，并不属于其他任何文件系统，并且只存在于内存中。

**2） 命名管道**

FIFO，也称为命名管道，它是一种文件类型。FIFO的通信方式类似于在进程中使用文件来传输数据，只不过FIFO类型文件同时具有管道的特性。在数据读出时，FIFO管道中同时清除数据，并且“先进先出”。

**特点：**

1. FIFO可以在无关的进程之间交换数据，与无名管道不同。
2. FIFO有路径名与之相关联，它以一种特殊设备文件形式存在于文件系统中。

**3.2.2 共享内存**

共享内存（Shared Memory），指两个或多个进程共享一个给定的存储区。

**特点：**

1. 共享内存是最快的一种 IPC，因为进程是直接对内存进行存取。
2. 因为多个进程可以同时操作，所以需要进行同步。
3. 信号量+共享内存通常结合在一起使用，信号量用来同步对共享内存的访问。

**3.2.3 信号量**

信号量（semaphore）与已经介绍过的 IPC 结构不同，它是一个计数器。信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据。

**特点：**

1. 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存。
2. 信号量基于操作系统的 P、V 操作，程序对信号量的操作都是原子操作。
3. 每次对信号量的 P、V 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。
4. 支持信号量组。

**3.2.4 消息队列**

消息队列，是消息的链接表，存放在内核中。一个消息队列由一个标识符（即队列ID）来标识。

**特点：**

1. 消息队列是面向记录的，其中的消息具有特定的格式以及特定的优先级。
2. 消息队列独立于发送与接收进程。进程终止时，消息队列及其内容并不会被删除。
3. 消息队列可以实现消息的随机查询,消息不一定要以先进先出的次序读取,也可以按消息的类型读取。

**3.2.5 socket**

更为一般的进程间通信机制，可用于不同机器之间的进程间通信。

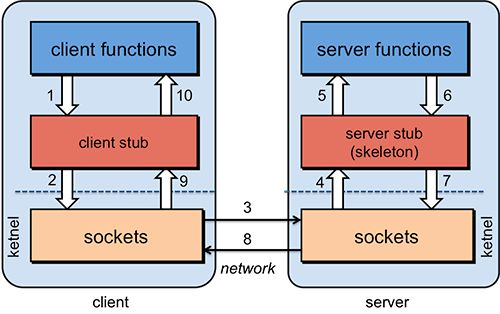
**3.2.6 RPC(Remote Process Call)**

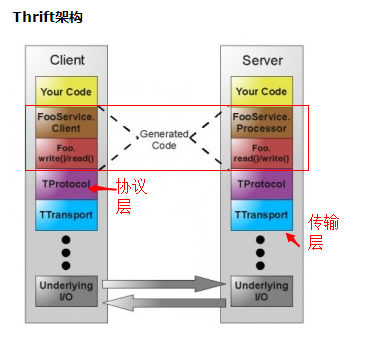
由于不在一个内存空间，不可以直接调用，需要通过网络来表达调用的语义和传达调用的数据。

**RMI**：JAVA自带的远程方法调用工具，不过有一定的局限性，毕竟是JAVA语言最开始时的设计，后来很多框架的原理都基于RMI。

**Hessian**：基于HTTP协议传输，在性能方面还不够完美，负载均衡和失效转移依赖于应用的负载均衡器，Hessian的使用则与RMI类似。

**Dubbo**：基于Netty的高性能RPC框架，是阿里巴巴开源的，繁多的配置，设置不当都会让人烦脑。





黄色部分是用户需求实现的业务逻辑，褐色部分是根据 Thrift 定义的服务接口描述文件生成的客户端和服务器端代码框架，红色部分是根据 Thrift 文件生成代码，实现数据的读写操作。TProtocol（协议层），定义数据传输格式（二进制、json等）。TTransport（传输层），定义数据传输方式，可以为TCP/IP传输，内存共享或者文件共享等。

其中管道、信号、共享内存和消息队列只能运行在一台机器上，而 socket 和 RPC 则提供了远程支持。当然，也有在 socket 或 RPC 基础上实现消息队列的。一般需要实现进程间通信，可以直接考虑 socket 或 RPC，毕竟以后的业务场景有可能扩展到多机。

四、死锁

**1. 概念**

两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。

**2. 死锁产生的原因**

（1）因竞争资源发生死锁

现象：系统中供多个进程共享的资源的数目不足以满足全部进程的需要时，就会引起对诸资源的竞争而发生死锁现象

（2）进程推进顺序不当发生死锁

**3. 死锁产生的四个必要条件**（有一个条件不成立，则不会产生死锁）

（1）互斥条件：一个资源一次只能被一个进程使用

（2）请求与保持条件：一个进程因请求资源而阻塞时，对已获得资源保持不放

（3）不剥夺条件：进程获得的资源，在未完全使用完之前，不能强行剥夺

（4）循环等待条件：若干进程之间形成一种头尾相接的环形等待资源关系

**4. 处理死锁的方法**

只要上述一个条件不成立，就不会产生死锁，所以解决死锁的基本方法有：预防死锁、避免死锁、检测死锁、解除死锁。

（1）预防死锁（破坏四个必要条件）：

资源一次性分配：（破坏请求和保持条件）

可剥夺资源：即当某进程新的资源未满足时，释放已占有的资源（破坏不可剥夺条件）

资源有序分配法：系统给每类资源赋予一个编号，每一个进程按编号递增的顺序请求资源，释放则相反（破坏环路等待条件）

（2）避免死锁（银行家算法）:

预防死锁的几种策略，会严重地损害系统性能。因此在避免死锁时，要施加较弱的限制，从而获得较满意的系统性能。由于在避免死锁的策略中，允许进程动态地申请资源。因而，系统在进行资源分配之前预先计算资源分配的安全性。若此次分配不会导致系统进入[不安全状态](http://metc.gdut.edu.cn/os/oscai/chapter2/pages/ch29.htm#%E6%A6%82%E5%BF%B5)，则将资源分配给进程；否则，进程等待。其中最具有代表性的避免死锁算法是银行家算法。

（3）检测死锁

通过将资源分配图简化的方法来检测系统状态S是否为死锁状态。

（4）解除死锁

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剥夺资源：从其它进程剥夺足够数量的资源给死锁进程，以解除死锁状态。

②撤消进程：可以直接撤消死锁进程或撤消代价最小的进程，直至有足够的资源可用，死锁状态消除为止。

五、内存管理

**5.1 内存管理的方式**

（1）块式管理：把主存分为一大块一大块的，当所需的程序片段不在主存时就分配一块主存空间，把程序片段load入主存，就算所需的程序片段只有几个字节也只能把这一块分配给它。这样会造成很大的浪费，平均浪费了50%的内存空间，但是易于管理。

（2）页式管理：把主存分为一页一页的，每一页的空间要比一块一块的空间小很多，这种方法的空间利用率要比块式管理高很多

（3）段式管理：把主存分为一段一段的，每一段的空间又要比一页一页的空间小很多，这种方法在空间利用率上又比页式管理高得多，但是也有另外一个缺点。一个程序片段可能会被分为几十段，这样很多时间就会被浪费在计算每一段的物理地址上。

（4）段页式管理：结合了段式管理和页式管理的优点。把主存先分成若干段，每个段又分成若干页。段页式管理每取一次数据，要访问3次内存。

**5.2 分页和分段的区别**

页是信息的物理单位，分页是为了实现离散分配方式，以消减内存的外零头，提高内存的利用率；或者说，分页仅仅是由于系统管理的需要，而不是用户的需要。

段是信息的逻辑单位，它含有一组其意义相对完整的信息。分段的目的是为了能更好地满足用户的需要。页的大小固定且由系统确定，把逻辑地址划分为页号和页内地址两部分，是由机器硬件实现的，因而一个系统只能有一种大小的页面。段的长度却不固定，决定于用户所编写的程序，通常由编译程序在对源程序进程编辑时，根据信息的性质来划分。

分页的作业地址空间是一维的，即单一的线性空间，程序员只需利用一个记忆符，即可表示一地址。分段的作业地址空间是二维的，程序员在标识一个地址时，既需给出段名，又需给出段内地址。

**5.3 虚拟内存**

虚拟内存简称虚存，是计算机系统内存管理的一种技术。它是相对于物理内存而言的，可以理解为“假的”内存。它使得应用程序认为它拥有连续可用的内存（一个连续完整的地址空间），允许程序员编写并允许比实际系统拥有的内存大得多的程序，这使得许多大型软件项目能够在具有有限内存资源的系统上实现。而实际上，它通常被分割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。虚存比实存有以下好处：

（1） 扩大地址空间。无论段式虚存，还是页式虚存，或是段页式虚存，寻址空间都比实存大。

（2）内存保护。每个进程运行在各自的虚拟内存地址空间，互相不能干扰对方。另外，虚存还对特定的内存地址提供写保护，可以防止代码或数据被恶意篡改。

（3）公平分配内存。采用了虚存之后，每个进程都相当于有太阳大小的虚存空间。

（4）当进程需要通信时，可采用虚存共享的方式实现。

不过，使用虚存也是有代价的，主要表现在以下几个方面：

（1）虚存的管理需要建立很多数据结构，这些数据结构要占用额外的内存。

（2）虚拟地址到物理地址的转换，增加了指令的执行时间

（3）页面的换入换出需要磁盘I/O，这是很耗时间的。

（4）如果一页中只有一部分数据，会很浪费内存。

六、案例分析

1. 生产者消费者模型（阻塞队列实现）

public class Resource3 {

private BlockingQueue resourceQueue = new LinkedBlockingQueue(10);

/\*\*

\* 向资源池中添加资源

\*/

public void add(){

try {

resourceQueue.put(1);

System.out.println("生产者" + Thread.currentThread().getName()

+ "生产一件资源," + "当前资源池有" + resourceQueue.size() +

"个资源");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

/\*\*

\* 向资源池中移除资源

\*/

public void remove(){

try {

resourceQueue.take();

System.out.println("消费者" + Thread.currentThread().getName() +

"消耗一件资源," + "当前资源池有" + resourceQueue.size()

+ "个资源");

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public class ProducerThread3 extends Thread {

private Resource3 resource3;

public ProducerThread3(Resource3 resource) {

this.resource3 = resource;

}

public void run() {

while (true) {

try {

Thread.sleep((long) (1000 \* Math.random()));

} catch (InterruptedException e) {

e.printStackTrace();

}

resource3.add();

}

}

}

public class ConsumerThread3 extends Thread {

private Resource3 resource3;

public ConsumerThread3(Resource3 resource) {

this.resource3 = resource;

}

public void run() {

while (true) {

try {

Thread.sleep((long) (1000 \* Math.random()));

} catch (InterruptedException e) {

e.printStackTrace();

}

resource3.remove();

}

}

}

public class TestProducerAndConsumer {

public static void main(String[] args) {

Resource3 resource = new Resource3();

//生产者线程

ProducerThread3 p = new ProducerThread3(resource);

//多个消费者

ConsumerThread3 c1 = new ConsumerThread3(resource);

ConsumerThread3 c2 = new ConsumerThread3(resource);

ConsumerThread3 c3 = new ConsumerThread3(resource);

p.start();

c1.start();

c2.start();

c3.start();

}

}