操作系统概念

1. 操作系统定义

   操作系统没有统一的定义，但是可以说操作系统是一个在电脑硬件和电脑用户之间的一个协调者。

2. 操作系统的作用

   - Control and coordinate the use of system resources (hardware and software)
   - Use the computer hardware in an efficient and protected manner
   - Make the computer system convenient to use for users (services)

3. 操作系统的分时系统

   CPU的运行速度远远快于IO或者其他的操作响应，所以串行执行任务会让CPU大部分时间处于空闲状态。让CPU在多任务之间进行切换，让用户觉得每一个任务都是在同时执行。

4. 运行模式

   为了把操作系统和一般的程序进行分离，操作系统会由两种运行状态，一种是user mode，另一种是kernel mode。一些操作只能在kernel mode下执行（比如IO控制，timer，中断管理），如果在user mode 中需要用到kernel mode下才能执行的功能，那可以调用system call进入kernel mode，从system call中返回时进入user mode。

5. timer作用

   - 保证OS拥有对于CPU的控制权
   - 防止用户的code进入infinite loop而不把控制权转还到OS。
   - user program在运行的的时候，timer就会启动，如果counter变成了0，就会产生一个中断，CPU控制权重新回到OS，OS把相应的program杀死。

6. 多处理器和单处理器计算机

   现代电脑基本都是多处理器，也叫并行或者多核系统，多个CPU公用总线，内存，时钟等，但是每一个处理器都有自己的register和cache。

   多处理器的优点：

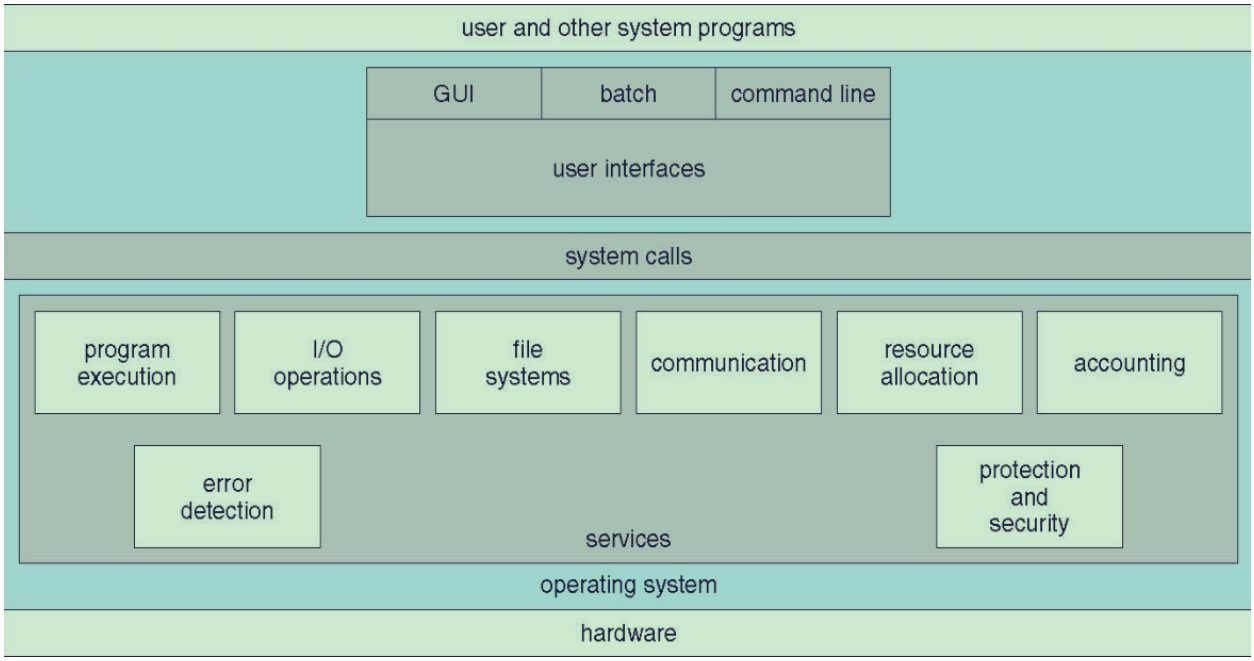
   - 更大的吞吐量
   - 比多个单核计算机更加经济
   - 更可靠

   有两种不同的CPU组织方式，一种是不对称处理——boss-worker，另一种是对称处理。

7. 多核和计算机集群的比较

操作系统结构

- 系统服务



- 系统调用
- 系统软件
- 系统设计和实现
- 系统结构
- 虚拟机

系统进程

1. 什么是进程

   进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位。

   1） 运行状态：进程正在处理器上运行。在单处理器的环境下，每一时刻最多只有一个进程处于运行状态。

   2） 就绪状态：进程已处于准备运行的状态，即进程获得了除CPU之外的一切所需资源，一旦得到处理器即可运行。

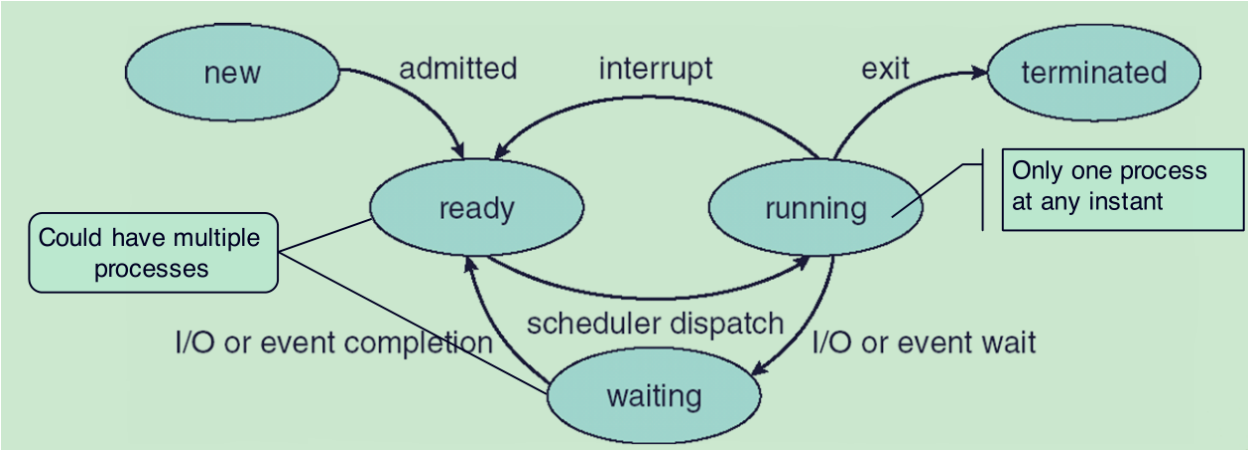
   3） 阻塞状态：又称为等待状态：进程正在等待某一事件而暂停运行，如等待某资源为可用（不包括处理器），或等待输入输出的完成。及时处理器空闲，该进程也不能运行。

   4） 创建状态：进程正在被创建，尚未转到就绪状态。创建进程通常需要多个步骤：首先申请一个空白的PCB，并向PCB中填写一些控制和管理进程的信息；然后由系统为该进程分配运行时所必须的资源；最后把该进程转入到就绪状态。

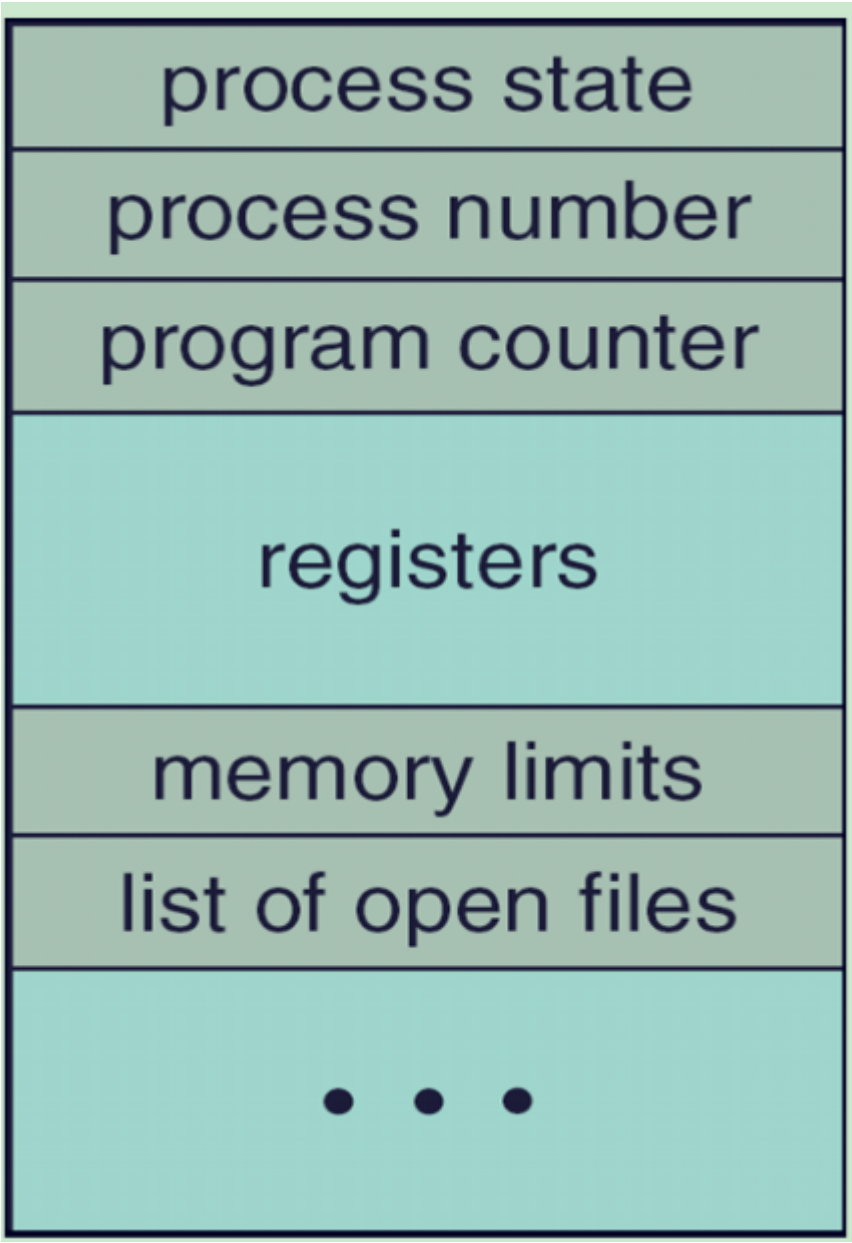
   5） 结束状态：进程正在从系统中消失，这可能是进程正常结束或其他原因中断退出运行。当进程需要结束运行时，系统首先必须置该进程为结束状态，然后再进一步处理资源释放和回收工作。

   注意区别就绪状态和等待状态：就绪状态是指进程仅缺少处理器，只要活得处理器资源就立即执行；而等待状态是指进程需要其他资源或等待某一事件，即使处理器空闲也不能运行。

   进程状态转变：



进程在操作系统中的表示：PCB（Process Control Block）



顺带提一下线程，线代OS一般都允许每一个进程中有多个线程，这样一个程序才能并发运行不同的任务。

- 程序和进程的区别

- 可以说进程不止于程序，一个程序可能只是作为一个部分出现在一个进程的text，code，process state
- 有时候也可以说程序不止一个进程，一个程序可以多次调用形成多个进程。
- 程序应该说是静态的（存在磁盘上的代码行），而进程是有life circle的，始终在某一个状态中。

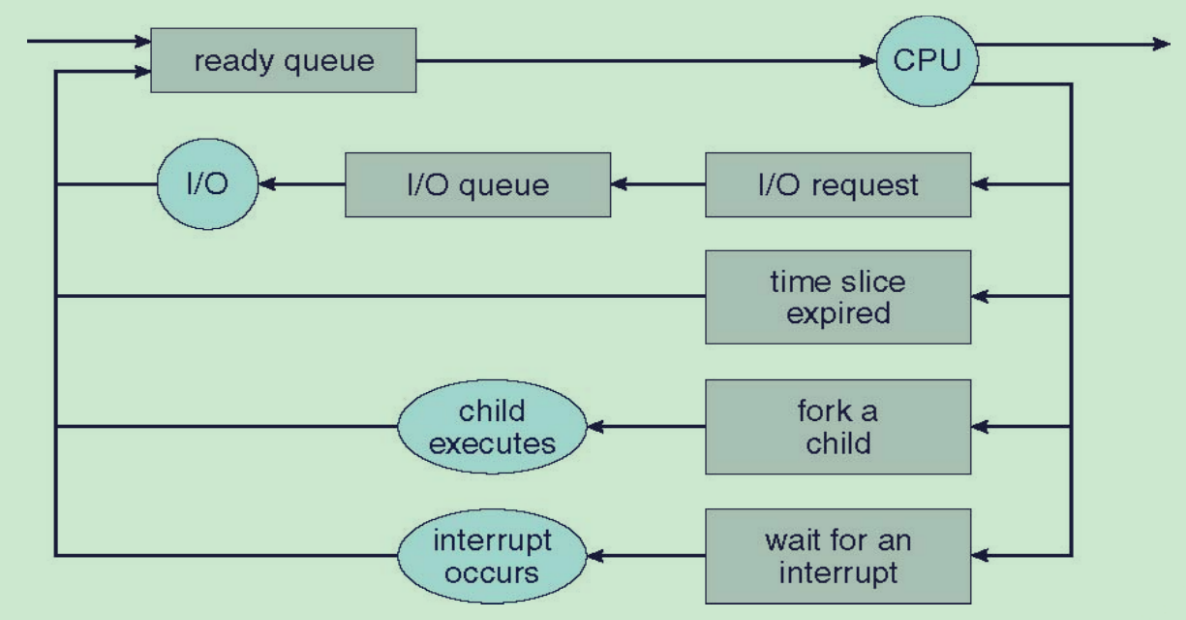
2. 进程的特征，比如进程管理、启动、终止、通信

- 进程管理

  任务队列（Job Queue）：系统所有的进程，是一个进程池

  Ready queue：在内存中准备执行的进程

  Device queues：在等待确定的IO设备的进程



  Long-term scheduler：任务调度，决定什么时候把什么进程放入ready queue/写入内存中，较少被调用。

  Short-term scheduler：CPU调度，决定ready queue 中哪个进程得到CPU的时间，经常被调用。The long-term scheduler controls the **degree of multiprogramming** - the number of processes that can concurrently be running in a system。

  Medium-term scheduler：为了降低系统并发执行进程的度，可以引入Medium-term scheduler的概念，它的主要工作是把内存中进程移除到磁盘上，需要时在调入内存，这个过程叫做**swapping**。

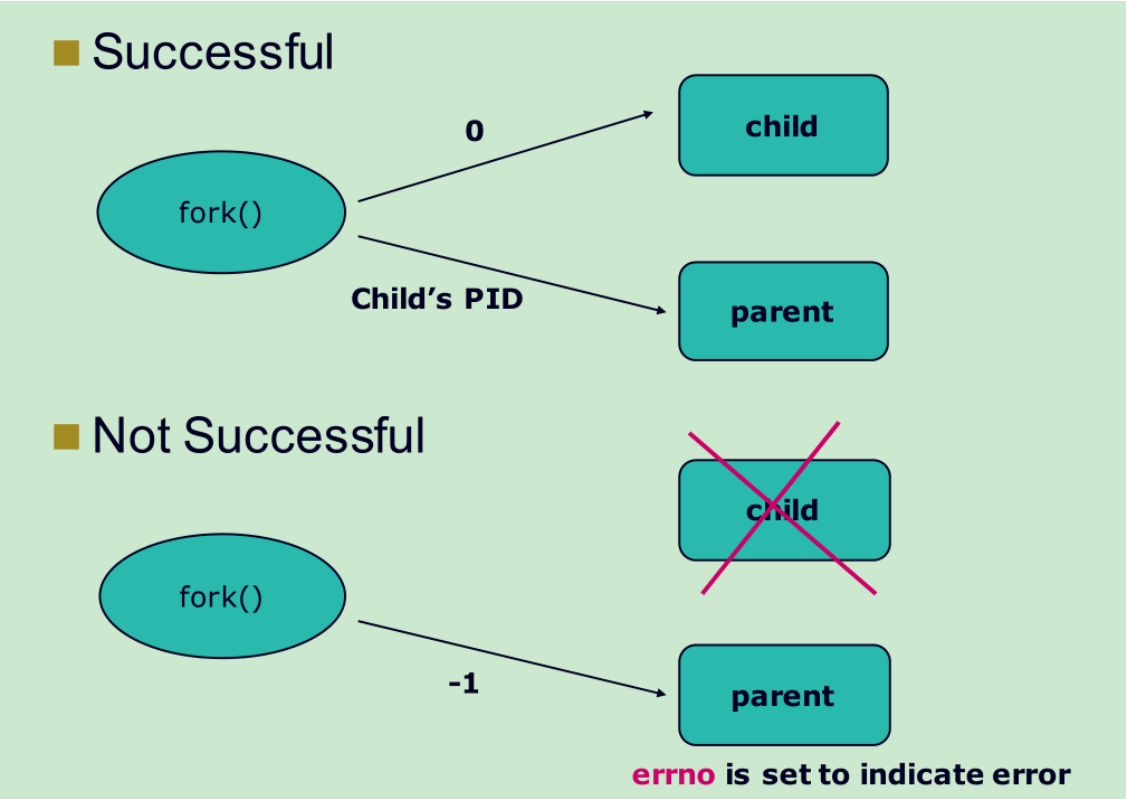
  在进程的转化过程，系统必须保留执行的现场，这种方式叫做**Context Switch**，Context Switch 的信息保存在进程的PCB中。需要保留的信息有程序计数器，寄存器，进程状态，内存管理的信息，打开的文件等等。进程的切换是比较好费时间的操作，而且很依赖于硬件的支持，比如有的硬件能够同时操作多个寄存器。

- 进程的启动、生成、终止
  - 启动

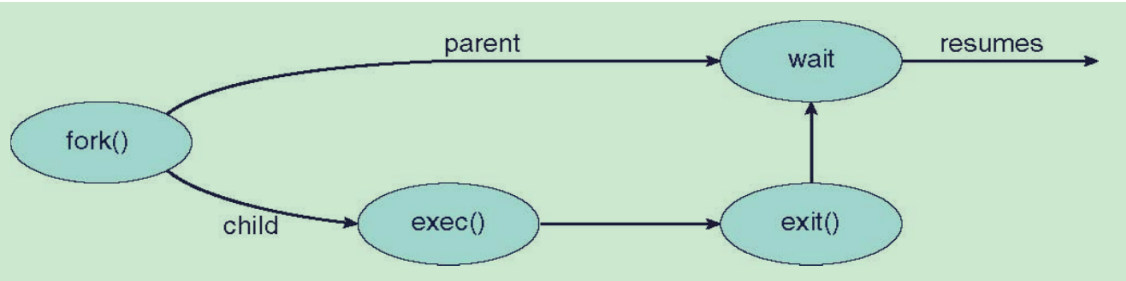
    在Linux中调用fork()函数。

    操作系统会创建一个新的PCB给子进程（如果成功），把父进程的运行状态拷贝一份一模一样的，也就是说子进程的PCB中除了PID和父进程不一样，其余的一致。

    fork()函数的返回值



进程生成之后的执行流程：




```
int main()
{
    for (int i = 0; i < 10; i++)
        if (fork())
            fork();
    return 0;
}
```

这段函数会生成3^{10} 个进程（自己可以画出流程图思考一下）

- 终止进程

  进程的终止有多种情况

  - 进程执行完最后一行代码，请求OS来删除进程（内存，寄存器的使用权等），一般使用exit()函数
  - 父进程可以终止子进程（abort()函数）
    - 子进程的资源已经被OS回收了
    - 子进程的任务被取消了
    - 父进程退出执行了

不同的OS采取的策略可能不同，有的OS不允许子进程在父进程结束之后继续执行。但是有的OS允许，这就造成了子进程没有父进程，这样的进程称为“孤儿进程”，Linux和Unix把孤儿进程全部归到系统的**init**进程的子进程中。init进程会周期性的调用wait()来回收子进程的资源。

- wait()函数的流程说明

  一般一个父进程调用了wait()这个系统函数，父进程会停止执行，等待某个特定（或者所有的子进程）执行退出。子进程exit()后，返回执行的状态码到OS，OS在把返回状态返回给父进程，父进程继续执行。

  如果父进程没有调用wait()函数，而这个时候子进程终止了（通俗讲就是子进程终止的消息没有被父进程接收到，子进程的入口和PID之类的还是存在于父进程的进程目录中），这个子进程就叫做zombie process。

3. IPC的两种方式
   - 共享内存

     不同的进程使用相同的内存地址，都可以调用read() 和write()这样的system calls。但是这样的策略在多核的时候很麻烦（每个核有自己的内存和缓存）。

   - 消息传递

     使用send() 、receive()这样的system calls。使用消息队列来保存管理消息。

     了解消息的队列在操作系统中的使用以及作用的流程。

4. C/S之间的通信
   - 通过网络通信，使用一对Socket
   - pipes

     通过操作系统的管道技术的实现了解一般的管道技术的原理。

多线程编程

1. 多线程的好处
   - 更快的响应速度。如果某一部分的程序block了，那么CPU可以转到其他部分执行，而不会卡死。
   - 资源的共享。一个进程的多个线程是默认共享进程的资源的，着对于进程之间通信来说是简化很多的，更加容易实现。
   - 更轻量级。线程的生成比起进程来说是资源消耗更少，也不会有context switch的困扰。
   - Scalability。能够更好的利用多核的机器，
2. 多进程编程的挑战性
   - 如何切分一个task为多个进程需要很好的设计
   - 数据的分配问题，怎么才是最小的耦合性
3. 并发和并行的区别

   并行是真正意义上的同时运行，多个核心执行不同的任务；并发只是不断的切换任务，并不是真正的同时。
4. Amdahl's Law （阿姆达尔定律）

   增加更多的处理器核心带来的速度提升比可以用下面的公式描述

$$speedup = \frac{1}{S + \frac{(1-S)}{N}}$$

其中S是描述程序中串行部分比例的一个百分数，N是总共的核数。

比如一个应用程序75%是并行，25%串行，那么使用两个核心最多比一个核心要快1.6倍。

如果$N \to \infty$，那么加速比趋近于$\frac{1}{S}$。

5. 线程的状态

   就像进程间一样，每一个线程都要维护一个状态信息，线程的状态信息就保存在TCB（Thread Control Table）中，包括下面的信息：

   - 执行的状态：CPU的寄存器、程序的计数器、堆栈的位置指针
   - 调度信息等等

6. 用户线程、核心线程

   - 核心线程
     - 内核线程又称为守护进程，内核线程的调度由内核负责，一个内核线程处于阻塞状态时不影响其他的内核线程，因为其是调度的基本单位。这与用户线程是不一样的；
     - 这些线程可以在全系统内进行资源的竞争；
     - 内核空间内为每一个内核支持线程设置了一个线程控制块（TCB），内核根据该控制块，感知线程的存在，并进行控制。在一定程度上类似于进程，只是创建、调度的开销要比进程小。有的统计是1：10。
     - 内核线程切换由内核控制，当线程进行切换的时候，由用户态转化为内核态。切换完毕要从内核态返回用户态，**即存在用户态和内核态之间的转换，比如多核cpu，还有win线程的实现。**

   - 用户线程

     用户线程一般是有runtime library支持的，最主要的的三个的线程库是

     ●POSIX Pthreads – POSIX standard (IEEE 1003.1c) define an API for thread creation and synchronization. Most UNIX-type systems such as Linux, Mac OS X, and Solaris. ● Win32 threads – Window thread library. ● Java thread – any system that provides a JVM such as Window, Linux, and Mac OS X.

     - 用户线程在用户空间中实现，内核并没有直接对用户线程进程调度，内核的调度对象和传统进程一样，还是进程（用户进程）本身，内核并不能看到用户线程，内核并不知道用户线程的存在。
     - 不需要内核支持而在用户程序中实现的线程，其不依赖于操作系统核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。
     - 内核资源的分配仍然是按照进程（用户进程）进行分配的；**各个用户线程只能在进程内进行资源竞争。**
     - 用户级线程内核的切换由用户态程序自己控制内核切换（通过系统调用来获得内核提供的服务）,不需要内核干涉，少了进出内核态的消耗，但不能很好的利用多核Cpu。**目前Linux pthread大体是这么做的。**
     - 每个用户线程并不具有自身的线程上下文。因此，就线程的同时执行而言，任意给定时刻每个进程只能够有一个线程在运行，而且只有一个处理器内核会被分配给该进程。

   - 用户级线程和内核级线程的区别

     - **内核支持：**用户级线程可在一个不支持线程的OS中实现；内核支持线程则需要得到OS内核的支持。亦即内核支持线程是OS内核可感知的，而用户级线程是OS内核不可感知的。
     - **处理器分配：**在多处理机环境下，对用户级线程而言主，内核一次只为一个进程分配一个处理器，进程无法享用多处理机带来的好处；在设置有内核支持线程时，内核可调度一个应用中的多个线程同时在多个处理器上并行运行，提高程序的执行速度和效率。
     - **调度和线程执行时间：**设置有内核支持线程的系统，其调度方式和算法与进程的调度十分相似，只不过调度单位是线程；对只设置了用户级线程的系统，调度的单位仍为进程。

    - 用户级线程执行系统调用指令时将导致其所属进程被中断，而内核支持线程执行系统调用指令时，只导致该线程被中断。
    - 在只有用户级线程的系统内，CPU调度还是以进程为单位，处于运行状态的进程中的多个线程，由用户程序控制线程的轮换运行；在有内核支持线程的系统内，CPU调度则以线程为单位，由OS的线程调度程序负责线程的调度。
  - 内核线程和用户线程的联系
    - 一对一模型

      一个用户进程绑定一个内心线程，一旦用户的线程结束，内核线程也要结束。例如，linux使用clone()创建的线程，以及win下使用CreateThread()创建的线程。

      弊端：

      内核线程数量有限 许多操作系统内核线程调用的时候，上下文切换的开销很大。
    - 多对一模型

      多个用户线程绑定到一个内核线程。这种线程切换起来很快，用户代码来控制，内核线程并不需要改变。

      缺点：一旦一个线程阻塞了，其他线程就不能执行了。（这不是没有并发的功能了？）

      现代的OS基本不使用这种模式
    - 混合模型
    - 多对多模型

      多个用户映射到多个内核线程上，比起多对一的并发性更好了。在利用多处理器效果上更好，但是效果还是比不上一对一。

7. 线程带来的问题
  - fork()和exec()函数调用的作用

    我们知道fork函数能够新建一个新的进程，拷贝父进程的状态信息。但是如果一个父进程的某一个线程调用了fork函数，是不是父进程的所有线程们都要拷贝呢?这是一个设计问题而不是技术的问题了。
    - 有的UNIX系统采用了两套fork机制，让调用者进行选择
    - 如果exec函数在fork函数之后立即执行，那么拷贝所有的线程信息是多余的，exec指定要执行的program会替换整个进程。所以只要创建一个单线程的进程就可以了。
  - 信号处理

    信号是用来通知进程系统中某件事件的发生。信号一旦产生，必须要处理，处理的流程都按照下面：
    - 某件事件产生引发了该信号
    - 信号被传到某一进程
    - 信号被接受后，必须要处理

    信号的处理机制有同步和异步两种，取决于信号的产生地点
    - 如果在进程执行的内部代码产生的信号，那么信号就要同步处理。比如非法内存操作、零作除数c
    - 进程执行时，外部事件引起的信号异步处理。

    ## signal handler

    有两种signal handler：系统默认、用户自定义的

    所有的信号默认系统处理，用户定义的handler可以覆盖系统默认的
  - 目标线程的取消

当某项task完成之后，相应执行该任务的线程就可以取消了。取消线程也是分同步和异步两种方式。

- 异步取消会立即终止目标线程
- 延迟取消会让线程check自己是否需要取消，让线程更有序、更安全的终止。

- 线程存储空间

  每个线程有自己的数据空间。和函数的本地变量不同，每个线程的数据区可以让多个函数共同使用，更像是 static data。

CPU调度

1. 学习目标
   - 基本概念和名词

     进程的执行模式是：**CPU burst + I/O burst。**CPU执行和IO等待时间交替进行的，这就为多线程运行提供了实行的空间。

     这一章节介绍的是短期调度，也就是怎么合理把内存中的任务分配到CPU执行。CPU调度决策发生在以下的情况：

     - 进程从running状态变成ready状态
     - 进程从running变成waiting状态
     - 进程从waiting变成ready状态
     - 进程terminate

     上面这几种都是非抢占式的，其他的任务调度都是抢占式的。

     ### Dispatcher

     在short-term scheduler决定了CPU下一个要执行的进程后，Dispatcher便开始转交CPU的控制权，这包括下面的动作：

     - 转换Context
     - 转换成User mode
     - 跳转到指定的内存空间重新启动那个程序

     所以说调度是一定的延迟或者说是代价的。

     ### 设计CPU调度程序的准则

     - CPU的利用率越高越好，keep CPU as busy as possible
     - 吞吐量 。单位时间处理的进程数量
     - Turnaround Time，CPU执行一个进程的时间
     - 进程等待时间。一般采用的平均等待时间来衡量，算法要尽量让等待时间较少
     - 响应时间，一个请求从产生到产生响应的时间，这一点对于图形用户很重要，一般这种控制用户视觉响应的进程在OS都会比较高的优先级，而后台处理的进程优先级会比较小一些。

     但是我们需要了解到OS设计的时候不可能兼顾所有指标，应该说有些指标的要求是相互冲突的，我们要根据用户的需求来确定那种算法是比较适用的。

   - 了解几种分配CPU时间的算法
     - FCFS（First Come, First Served）

       这应该是最简单的一种调度思想，先到先得。实现起来也是比较容易的。这种算法的性能和进程来到顺序有很大的关系。

考虑先后来到的三个进程：P_1, P_2, P_3，三个进程完成分别需要的CPU时间：24，3，3

可以知道平均等待时间：$P_{av} = (0 + 24 + 27)/3 = 17$

如果三个进程完成需要的时间分别是：3，3，24

平均等待时间$P_{av} = (0 + 3 + 6)/3 = 3$。

可以看出耗时较长的进程如果先到的话，对于后面的进程是有很大的影响的，会很大程度的影响OS的响应性能。

- SJF（Shortest Job First）

  受到上面的安排思想的启发，我们可以把最容易完成的进程先完成，最耗时的进程放在最后，平均的等待时间会有很大的提升。实际我们可以通过数学证明：SJF在平均等待时间方面是最优的。

  但是这个算法需要解决的一个问题就是进程的执行时间是不可知的，实际上不运行一次的话，谁也无法准确知道进程的运行耗时。所以我们需要一种算法来预测下一个进程的运行时间。下面介绍这种算法的主要思想。

  - 预测一个进程的运行时间

    我们假设CPU是按照一个个时间片来运行的，要预测进程的运行时间，我们需要知道两个数据——上一个进程的运行的准确时间，上一次预测的下一个进程时间。

    1. t_n = CPU实际的n^{th} 时间片的长度
    2. τ_n 是n^{th} 时间片长度的预测值
    3. 定义一个α,，$0 < \alpha < 1$
    4. 预测$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$

    通常来说$\alpha = 1/2$

  至于为什么这个能够比较预测一个时间片的长度，？

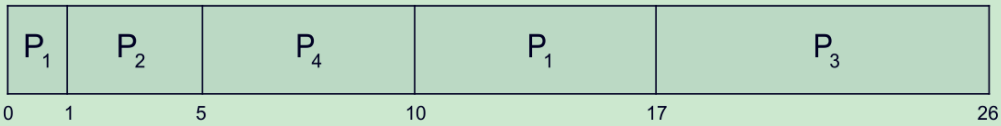
  SJF还有一个抢占式的版本——shortest-remaining-time-first

  简单的来说就是假定每个进程进入ready queue的时间并不是一致的，这样就会有一个问题，如果一个需耗时8的进程运行了4单位时间——剩余4个单位时间之后，有一个预测需要耗时5的进程进入了ready queue，这时候是应该立即剩余耗时4的进程还是等剩余耗时4的进程执行完呢。在抢占式的CPU调度中，耗时短的进程优先度是高的，可以抢夺优先级低的进程的执行时间，所以这里还是要继续执行剩余耗时最短为4的进程。

- Now we add the concepts of varying arrival times and preemption to the analysis

Process	*Arrival* Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

- *Preemptive* SJF Gantt Chart



- Average waiting time = [(10-1)+(1-1)+(17-2)+5-3)]/4 = 26/4 = 6.5 msec

- Round Robin (RR)

  RR调度决策方法中，把CPU时间分割成等长的时间片——长度为 q，每个进程得到一个时间片。如果一个时间片的时间该进程没有结束，进程由队头转入队尾；如果一个时间片中进程结束，那么下一个进程立即开始下一个时间片。

  在这种情况下，q 的大小很重要

  - q 太大时，就变成了FIFO的模式（大部分进程可以在一个时间片内完成）
  - q 太小时，进程切换的代价不可忽略，造成很大的CPU时间浪费。

- 优先级调度

  也就是根据一些信息确定进程的优先级，直接通过排序优先级，CPU每次都是调用最高优先级的进程。但是这里的优先级的确定方法就有很多的评价方法，每一种都不一定是最完美的，在某些情况下还是会存在某些缺陷。上面SJF本质也是一种优先级调度，优先级的标准是预测的进程CPU占用时间。

  优先级调度方法一般都有两种策略：

  - 非抢占式
  - 抢占式

  产生的问题：低优先级的进程有可能永远都没有机会运行。

  解决方法：老化——低优先级进程随着等待时间变长，优先级提升。

- 多层次队列

  在一种调度方案都会有不可避免的缺陷的情况下，使用多种方案共同决策能够扬长避短

  1. 比如Ready queue 分成两个队列
     - foreground （用于和用户交互的进程）
     - background

     foreground 使用RR算法，background使用FCFS算法。进程使用固定的优先级，一般来说是foreground的优先级比background进程高，这样有可能造成的就是有些后台进程starvation。然后固定一定比例的foreground 和background进程时间片比例，比如80%用作交互，20%给后台。

2. Multilevel Feedback Queue

   进程可以在不同的队列中转移，这样就可以使用老化的策略。

   一个多层次带反馈的队列需要确定下面东西：

   - 要多少不同等级的队列
   - 每个队列使用什么算法进行调度
   - 决定什么情况进程改变队列（包括升级和降级）的算法
   - 当一个进程需要某种服务的时候，决定该进程进入哪个等级的队列

- 真是操作系统的做法
  - Solaris
  - Windows
  - Linux