# 操作系统

## 进程、线程、协程

进程是系统进行资源调度和分配的基本单位，每个进程都有独立的地址空间用来保存处理执行的代码、变量、动态内存、调用指令等等。

进程有三个状态：

1. 就绪：等待处理的器的调度。就绪进程可以按照多个优先级来划分队列，如果进程由于时间片用完就进入低优先级队列。由I/O操作完成进入高优先及队列。进程创建之后先进入就绪状态。
2. 运行：进程占用处理器资源。
3. 等待：进程等待某种资源，在未得到资源之前无法继续执行。等待态又称为阻塞态或睡眠态。进程进入等待态之后就会让出处理器资源，当获取到等待的资源时，会产生一个中断使得操作系统唤醒在等待态中的进程。

进程三个状态之间的切换：

运行态->等待态：等待外部设备或人工干预。

等待态->就绪态：等待的条件已满足，待分配处理器之后就能运行。

运行态->就绪态：让出处理器，比如时间片用完或者更高优先级的进程抢占处理。

就绪态->运行态：从进程就绪队列中选择一个进程运行。

线程是操作系统进行运算调度的最小单位。同一个进程的多个线程共享进程的资源。线程没有独立的地址空间，而是共享所属进程的地址空间。

多进程的优点：调试简单，可靠性高，单个进程异常不会影响其他进程。进程间数据是分开的，同步简单。

多进程的缺点：创建、销毁开销大，内存和系统资源等占用大。

多线程的优点：创建、销毁开销小，内存和系统资源等占用小。

多线程的缺点：调试复杂，可靠性差，一个线程异常会导致整个进程异常。线程间数据共享，但是要做到数据同步则需要使用锁等操作，复杂度高。

弱相关处理、多机分布式的系统多进程更好。强相关处理、大量的计算任务、多核分布式的系统多线程更好。

协程类似多线程，及在执行一个任务的时候可以中断，转而去执行其他的任务，在适当的时候再回来继续执行中断的任务。但是协程是一个线程而不是多个线程。协程的上下文切换是由程序控制的，而非由操作系统控制。因此协程切换开销更小。

协程比多线程效率更高，而且由于协程是单线程，不需要引入锁或类似机制来进行数据保护。但协程无法利用多核CPU。协程切换时候内容保存在用户栈或者用户堆，协程的切换不会进入内核态。线程和进程切换的时候内容保存在内核栈中，线程和进程的切换都会进入内核态，再回到用户态。协程不适合用于计算密集型的操作，而适合于IO密集型的操作。

## 二、进程线程的API

创建进程fork()

fork执行一次返回两次。在父进程中返回的是子进程的id，在子进程中返回的是0。如果返回-1则表示创建子进程失败。

fork失败可能是操作系统进程数量达到上限，errno为EAGAIN。也有可能是内存不足，errno为ENOMEM。

fork函数做了以下几件事情：

1. 分配新的内存和内核数据给子进程。
2. 父进程的数据空间、堆栈等拷贝给子进程。
3. 子进程不继承父进程的文件锁，但会继承父进程的互斥锁、读写锁和条件变量及它们的状态。
4. 添加子进程到系统进程列表中。
5. 返回，等待系统调度。

调用fork之后，系统会创建一个新的PCB也就是task\_struct结构，与父进程的id不相同，其他都相同。父子进程的内存空间也是映射到相同的地址。

但当其中一个进程试图修改内存空间里的内容时，系统会为修改的内容复制一个新的内存空间。这就是写时复制。写时复制是针对内存页的，而不是针对进程的。

task\_struct是按页分配的，页里面多余的部分则作为该进程的内存堆栈。

fork和vfork的区别：

1. fork父子进程交替运行，vfork保证子进程先运行，父进程阻塞，直到子进程结束，或子进程调用exec或exit。由于这个特性，如果vfork出来的子进程执行过程中依赖于父进程的操作，那么就会造成死锁。
2. fork实现了写拷贝，而vfork是父子进程共用内存空间。
3. vfork必须使用exit或excl退出。

pid\_t wait(int \*status)

进程一旦调用wait就会阻塞自己，wait自动检查是否有僵尸子进程，如果有的话wait会将这个子进程销毁并返回，否则就会一直阻塞。status用来保存被收集进程的一些状态，如果不需要则传入NULL。只要有一个子进程退出，wait就会返回。

pid\_t waitpid(pid\_t pid, int \*status, int options)

该函数的作用和wait函数是一样的，只是提供了更多的参数能够让使用者根据需要进行使用。

pid > 0，等待指定pid的进程退出，不管其他进程是否退出，只要指定的pid没有退出就一直等。

pid = 0，等待同一个进程组中的任意子进程退出。

pid = -1，作用和wait完全一样。

pid < -1，等待指定进程组的任意子进程退出，进程组的id等于pid的绝对值。

linux支持的options有两个WNOHANG和WUNTRACE，两个可以同时使用。

WNOHANG会让waitpid在即使没有进程退出的情况下返回，不会像wait一样一直阻塞。

创建线程

int pthread\_create(pthread\_t \*thread, const pthread\_arrt\_t \*attr, (void \*)(\* start\_rtn)(void\*), void \*arg);

返回0表示线程创建成功，创建失败则返回错误号。

第一个参数是创建好的线程标识符。

第二个参数用来设置线程属性，如果传入NULL，则使用默认属性。

1. 四个参数则是线程的回调函数以及回调函数的参数。

终止线程

int pthread\_cancel(pthread\_t thread);

thread就对应上面pthread\_create的第一个参数，一个线程可以调用pthread\_cancel终止同进程内的另一个线程。返回值0为成功，失败返回错误码。

void pthread\_exit(void \*retval);

线程调用pthread\_exit终止自己。参数一般传入空指针。

线程等待

void thread\_join(pthread\_t thread, void \*retval);

当前线程挂起等待目标线程thread终止。

僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid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。这种进程称之为僵死进程。

孤儿进程：父进程先于子进程退出，子进程被pid等于1的init进程收养，并有init进程完成其状态收集工作。

## 三、进程间通信

管道：半双工、有固定的写端和读端。

消息队列：内核中的消息链表，通过ID来标识一个消息队列。消息队列独立于收发进程，进程结束消息队列及其内容不会删除。消息队列里的消息可以按条件查询，不一定要按顺序读取。

// 创建或打开消息队列：成功返回队列ID，失败返回-1

int msgget(key\_t key, int flag);

// 添加消息：成功返回0，失败返回-1

int msgsnd(int msqid, const void \*ptr, size\_t size, int flag);

// 读取消息：成功返回消息数据的长度，失败返回-1

int msgrcv(int msqid, void \*ptr, size\_t size, long type,int flag);

// 控制消息队列：成功返回0，失败返回-1

int msgctl(int msqid, int cmd, struct msqid\_ds \*buf);

信号

共享内存

socket

## 四、线程间数据同步

1、锁

读写锁

pthread\_rwlock相关函数

读写锁有三种状态：读模式加锁，写模式加锁，不加锁。

只有一个线程可以占用写模式的读写锁，但是可以有多个线程占用读模式的读写锁。

即如果某个线程申请了读锁，那么其他线程也可以申请读锁，可以读数据。但是不能申请写锁。

而如果某个线程申请了写锁，其他线程不能申请读锁也不能申请写锁。

互斥锁

pthread\_mutex相关函数

得不到的锁的线程会被挂起，让出CPU，等待别的线程把锁释放之后再被唤醒。

自旋锁

pthread\_spin相关函数

得不到锁的线程不会被挂起，一直会占用CPU，直到得到锁。

可重入锁

外层函数获得锁之后，内层递归函数仍然可以获得该锁。即同一个线程可以多次获取已获取到的锁。不会导致死锁的发生。不可重入锁则会导致死锁的发生。

1. 信号量

信号量是一个确定的二元组(s,q)，其中s是一个具有非负初值的整形变量。q是一个初始状态为空的队列。s>=0表示系统中当前可用资源的数目。s<0时，其绝对值表示系统中因为请求该资源而被阻塞的进程数目。

p(s)操作：如果s的值大于零，就给他减1；如果值小于零，就挂起该进程。

v(s)操作：给s的值加1。

如果s<=0，每次进行v操作，就从q队列中移出第一个进程，使其变为就绪态。

int sem\_init(sem\_t \*sem, int pharsed, uint value);

pharsed如果为0表示局部信号量，只在当前进程内有效。如果是其他只则可以在不同的进程中使用。nginx中这个值传入的是1。value是sem的初始值，一般是0。

int sem\_wait(sem\_t \*sem);

这个就是P操作。

int sem\_post(sem\_t \*sem);

这个就是V操作。

int sem\_destroy(sem\_t \*sem);

所有的函数成功时返回0，失败时返回-1。

1. 条件变量

初始化条件变量

int pthread\_cond\_init(pthread\_cond\_t \*restrict cond, const pthread\_condattr\_t \*restrict attr);

attr一般直接传NULL

等待条件变量

int pthread\_cond\_wait(pthread\_cond\_t \*restrict cond, pthread\_mutex\_t \*restrict mutex);

唤醒一个等待条件变量的线程

int pthread\_cond\_signal(pthread\_cond\_t \*cond);

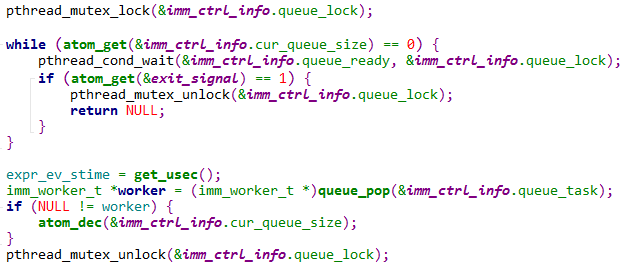
唤醒所有等待条件变量的线程

int pthread\_cond\_broadcast(pthread\_cond\_t \*cond);

销毁一个条件变量

int pthread\_cond\_destroy(pthread\_cond\_t \*cond);

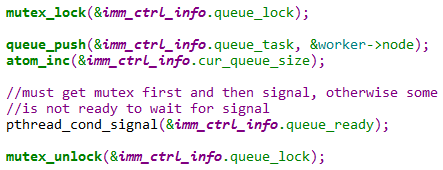
条件变量要配合互斥锁一起使用。



为什么在调用pthread\_cond\_wait之前要加锁？因为要保证对量长度为0的判断和条件等待为一个原子操作。如果不加锁的话，有可能出现在长度判断和wait之间，别的线程已经发出了条件满足的信号，这样会导致信号收不到。

pthread\_cond\_wait如果条件不满足，会阻塞，释放之前加的锁。当条件满足时候，会重新获取锁，执行后续的操作。

为什么要用while循环而不直接使用if判断队列是否为空？在多线程的系统里面，会有多个线程同时在等待一个条件满足。当条件满足时，信号会发送给所有等待的线程。线程就开始争抢互斥锁。获得互斥锁的线程才能够接下来的处理。其他的线程则被锁阻塞住。当处理完成锁被释放，又会有其他线程获得锁，这时候需要重新判断队列是否为空，因为刚刚有一个线程已经处理了队列中的成员。



数据加入队列的之前也需要加锁，加入完成之后调用pthread\_cond\_signal发送时间条件满足信号。然后释放锁。这里的锁和上面的锁是同一个锁。

## 五、进程调度策略

1、先进先出FIFO

先进入就绪队列的进程先被调度。等待CPU当前作业完成之后选取先进入就绪队列的进程执行。

优点：公平，简单，非抢占。但不适合交互式系统。

1. 短进程优先

选出CPU执行期最短的进程进行调度。如何知道进程执行要花费的时间？根据历史进行推测。

优点：可以保证最小的平均等待时间。

1. 时间片轮转

将一定的cpu时间分配给队列中的进程，均匀使用。

优点：定时响应，等待时间取决于时间片长度。

缺点：如果时间片太短，上下文切换次数多。如果时间片太长，又会使得每个进程的等待时间边长。

1. 优先级执行

抢占式：选择当前优先级最高的进程执行，执行过程中如果出现优先级更高的进程，则转而执行优先级更高的进程。

非抢占式：选择当前优先级最高的进程执行到结束。

缺点：会导致优先级低的进程一直得不到执行。不适合交互式系统。

1. 多级反馈队列

设置多个优先级队列，优先级越高的队列中的进程分到的CPU时间片越少。当第一队列空闲时，CPU才会执行第二队列，依次类推。第一队列中时间片内没有执行完的进程被加入第二队的队尾，第二队时间片内没有执行完的进程被加入第三队的队尾，依次类推。

最通用的调度算法。

## 六、用户态内核态

用户态和内核态的区别：

用户态的进程存取自己的指令和数据。内核态的进程能够存取内核和用户地址。

内核态CPU可以访问内存的所有数据，包括外围设备。

在内核态运行的指令称之为特权指令，可以访问用户存储和系统存储。

在内核态执行中的进程是不允许被抢占的，而在用户态执行中的进程是可以被抢占的。

用户态切换到内核态的三种方式：

1、系统调用：用户态进程主动要求切换到内核态的一种方式，用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作。

2、异常：当CPU在执行运行在用户态下的程序时，发生了某些不可知的异常，这时会触发由当前运行进程切换到处理此异常的内核相关程序中，也就转到了内核态，比如缺页异常

3、外围设备的中断：当外围设备完成用户请求的操作后，会向cpu发出相应的中断信号，这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序，

如果先前执行的执行时用户态下的程序，那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成，系统会切换到硬盘读写的中断处理程序中执行后续操作等。

系统调用本质也是一种中断。外围设备的中断叫硬中断，系统调用属于软中断。

## 七、系统调用

进程控制：

fork、getpid、exit、wait、waitpid、sched\_yeild

文件操作：

open、creat、fcntl、close、read、write

套接字：

socket、bind、listen、accept、connect、send、recv、sendto、recvfrom

## 八、死锁

两个或多个进程无限期的阻塞、相互等待的一种状态。

产生条件：

1、互斥：一个资源一次只能被一个进程使用。

2、占有并等待：一个进程占有一个资源，同时等待另一个资源。而这个资源为其他进程占有。

3、不抢占：一个进程只能等待其他进程释放资源，而不能主动抢占该资源。

4、循环等待：若干进程之间形成一种头尾相接的环形等待资源关系。

## 九、32位系统和64位系统各变量类型的大小

32位系统

      char ：1个字节

      char\*（即指针变量）: 4个字节

      short int : 2个字节

      int：  4个字节

      unsigned int : 4个字节

      float:  4个字节

      double:   8个字节

      long:   4个字节

      long long:  8个字节

      unsigned long:  4个字节

64位系统

char \*：8字节

long：8字节

unsigned long：8字节  
其他不变。

## 十、中断

所谓的中断就是在计算机执行程序的过程中，由于出现了某些特殊事情，使得CPU暂停对程序的执行，转而去执行处理这一事件的程序。等这些特殊事情处理完之后再回去执行之前的程序。中断一般分为三类：

1、由计算机硬件异常或故障引起的中断，称为内部异常中断。

2、由程序中执行了引起中断的指令而造成的中断，称为软中断（这也是和我们将要说明的系统调用相关的中断）。

3、由外部设备请求引起的中断，称为外部中断。简单来说，对中断的理解就是对一些特殊事情的处理。

中断处理程序是在中断发生时，由操作系统内核执行的特定函数。

中断优先级：

机器错误 > 时钟 > 磁盘 > 网络设备 > 终端 > 软件中断。

## 十一、内存管理

**1、全局/静态存储区**

又分为data区和bss区

data存放的是初始化的静态变量或全局变量。

bss存放的是未初始化的静态变量或全局变量。

**2、栈**

用户存放局部变量，栈是一块连续的内存区域，栈的容量是有限制的。栈内存也有动态分配，但栈内存的动态分配是由编译器管理的，释放也是由编译器进行释放。

函数形参的压栈顺序是从左到右。

ulimit -s 查询栈空间的大小

ulimit -s 数字 修改栈空间的大小

栈内存的增长是从高地址到低地址。

**3、堆**

动态申请的内存就是从堆内存中申请的。堆内存不连续，需要人工释放。

堆内存的增长是从低地址到高地址。

**4、代码段**

存放程序执行代码的区域，只读。

代码段和data段在编译的时候就已经分配了空间。而BSS不占用可执行文件的大小。

**5、常量区**

字符串常量放在该区域。const修饰的变量不能算作严格意义上的常量。const修饰符只是修饰了变量名，使其不能作为左值。但是并没有限定变量对应的内存只能读不能写。可以通过指针的方式修改内存中的值。

## 十二、虚拟内存和物理内存

每个进程都有自己独立的4G内存空间，这个内存是虚拟内存。当访问内从空间的某个地址，需要转换成物理内存的地址。所有的进程共享物理内存，每个进程只把自身当前需要的虚拟内存映射和存储到物理内存。

通过页表来记录哪些虚拟内存的数据在物理内存上以及在物理内存的什么位置。页表的表项分为两个部分，一部分记录此页是否在物理内存上，一部分记录物理内存页的地址。如果进程查看某个虚拟内存发现其不在物理内存上，则出现缺页异常。

进程常见的时候，内核只是创建了虚拟内存并建立了虚拟内存和磁盘文件之间的映射（通过mmap）。在实际运行程序的时候才会通过缺页异常来拷贝数据。程序在运行的时候，调用malloc等也只是分配虚拟内存。

页面置换算法：

1、FIFO算法

先入先出，即淘汰最早调入的页面。

2、OPT(MIN)算法

选未来最远将使用的页淘汰，是一种最优的方案，可以证明缺页数最小。

可惜，MIN需要知道将来发生的事，只能在理论中存在，实际不可应用。

3、LRU(Least-Recently-Used)算法

用过去的历史预测将来，选最近最长时间没有使用的页淘汰(也称最近最少使用)。

LRU准确实现：计数器法，页码栈法。