什么是操作系统？请简要概述一下

操作系统是管理计算机硬件和软件资源的计算机程序，提供一个计算机用户与计算机硬件系统之间的接口。

向上对用户程序提供接口，向下接管硬件资源。

操作系统本质上也是一个软件，作为最接近硬件的系统软件，负责处理器管理、存储器管理、设备管理、文件管理和提供用户接口。

操作系统有哪些分类？

操作系统常规可分为批处理操作系统、分时操作系统、实时操作系统。

若一个操作系统兼顾批操作和分时的功能，则称该系统为通用操作系统。

常见的通用操作系统有：Windows、Linux、MacOS等。

什么是内核态和用户态？

为了避免操作系统和关键数据被用户程序破坏，将处理器的执行状态分为内核态和用户态。

内核态是操作系统管理程序执行时所处的状态，能够执行包含特权指令在内的一切指令，能够访问系统内所有的存储空间。

用户态是用户程序执行时处理器所处的状态，不能执行特权指令，只能访问用户地址空间。

用户程序运行在用户态,操作系统内核运行在内核态。

如何实现内核态和用户态的切换？

处理器从用户态切换到内核态的方法有三种：系统调用、异常和外部中断。

1. 系统调用是操作系统的最小功能单位，是操作系统提供的用户接口，系统调用本身是一种软中断。
2. 异常，也叫做内中断，是由错误引起的，如文件损坏、缺页故障等。
3. 外部中断，是通过两根信号线来通知处理器外设的状态变化，是硬中断。

并发和并行的区别

1. 并发（concurrency）：指宏观上看起来两个程序在同时运行，比如说在单核cpu上的多任务。但是从微观上看两个程序的指令是交织着运行的，指令之间交错执行，在单个周期内只运行了一个指令。这种并发并不能提高计算机的性能，只能提高效率（如降低某个进程的相应时间）。
2. 并行（parallelism）：指严格物理意义上的同时运行，比如多核cpu，两个程序分别运行在两个核上，两者之间互不影响，单个周期内每个程序都运行了自己的指令，也就是运行了两条指令。这样说来并行的确提高了计算机的效率。所以现在的cpu都是往多核方面发展。

什么是进程?

进程是操作系统中最重要的抽象概念之一，是资源分配的基本单位，是独立运行的基本单位。

进程的经典定义就是一个执行中程序的实例。系统中的每个程序都运行在某个进程的上下文（context）中。

上下文是由程序正确运行所需的状态组成的。这个状态包括存放在内存中的程序的代码和数据，它的栈、通用目的寄存器的内容、程序计数器、环境变量以及打开文件描述符的集合。

进程一般由以下的部分组成：

1. 进程控制块PCB，是进程存在的唯一标志，包含进程标识符PID，进程当前状态，程序和数据地址，进程优先级、CPU现场保护区（用于进程切换），占有的资源清单等。
2. 程序段
3. 数据段

进程的基本操作

以Unix系统举例：

1. 进程的创建：fork()。新创建的子进程几乎但不完全与父进程相同。子进程得到与父进程用户级虚拟地址空间相同的(但是独立的)一份副本，包括代码和数据段、堆、共享库以及用户栈。子进程还获得与父进程任何打开文件描述符相同的副本，这就意味着当父进程调用 fork 时，子进程可以读写父进程中打开的任何文件。父进程和新创建的子进程之间最大的区别在于它们有不同的 PID。fork函数是有趣的（也常常令人迷惑）， 因为它只被调用一次，却会返回两次：一次是在调用进程（父进程）中，一次是在新创建的子进程中。在父进程中，fork 返回子进程的 PID。在子进程中，fork 返回 0。因为子进程的 PID 总是为非零，返回值就提供一个明 确的方法来分辨程序是在父进程还是在子进程中执行。

```
pid_t fork(void);
```

2. 回收子进程：当一个进程由于某种原因终止时，内核并不是立即把它从系统中清除。相反，进程被保持在一种已终止的状态中，直到被它的父进程回收（reaped）。当父进程回收已终止的子进程时，内核将子进程的退出状态传递给父进程，然后抛弃已终止的进程。一个进程可以通过调用 waitpid 函数来等待它的子进程终止或者停止。

   ```
   pid_t waitpid(pid_t pid, int *statusp, int options);
   ```

3. 加载并运行程序：execve 函数在当前进程的上下文中加载并运行一个新程序。

   ```
   int execve(const char *filename, const char *argv[], const char *envp[]);
   ```

4. 进程终止：

   ```
   void exit(int status);
   ```

简述进程间通信方法

每个进程各自有不同的用户地址空间,任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核,在内核中开辟一块缓冲区,进程A把数据从用户空间拷到内核缓冲区,进程B再从内核缓冲区把数据读走,内核提供的这种机制称为进程间通信。

不同进程间的通信本质：进程之间可以看到一份公共资源；而提供这份资源的形式或者提供者不同，造成了通信方式不同。

进程间通信主要包括管道、系统IPC（包括消息队列、信号量、信号、共享内存等）、以及套接字socket。

进程如何通过管道进行通信

管道是一种最基本的IPC机制，作用于有血缘关系的进程之间，完成数据传递。调用pipe系统函数即可创建一个管道。有如下特质：

1. 其本质是一个伪文件(实为内核缓冲区)
2. 由两个文件描述符引用，一个表示读端，一个表示写端。
3. 规定数据从管道的写端流入管道，从读端流出。

管道的原理: 管道实为内核使用环形队列机制，借助内核缓冲区实现。

管道的局限性：

1. 数据自己读不能自己写。
2. 数据一旦被读走，便不在管道中存在，不可反复读取。

3. 由于管道采用半双工通信方式。因此，数据只能在一个方向上流动。
4. 只能在有公共祖先的进程间使用管道。

进程如何通过共享内存通信?

它使得多个进程可以访问同一块内存空间，不同进程可以及时看到对方进程中对共享内存中数据得更新。这种方式需要依靠某种同步操作，如互斥锁和信号量等。

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操作来实现通信，避免了数据在用户空间和内核空间来回拷贝。
2. 因为多个进程可以同时操作，所以需要进行同步处理。
3. 信号量和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。

什么是信号

一个信号就是一条小消息，它通知进程系统中发生了一个某种类型的事件。 Linux 系统上支持的30 种不同类型的信号。 每种信号类型都对应于某种系统事件。低层的硬件异常是由内核异常处理程序处理的，正常情况下，对用户进程而言是不可见的。信号提供了一种机制，通知用户进程发生了这些异常。

1. 发送信号：内核通过更新目的进程上下文中的某个状态，发送（递送）一个信号给目的进程。发送信号可以有如下两种原因：
    - 内核检测到一个系统事件，比如除零错误或者子进程终止。
    - 一个进程调用了kill 函数， 显式地要求内核发送一个信号给目的进程。一个进程可以发送信号给它自己。
2. 接收信号：当目的进程被内核强迫以某种方式对信号的发送做出反应时，它就接收了信号。进程可以忽略这个信号，终止或者通过执行一个称为信号处理程序(signal handler)的用户层函数捕获这个信号。

如何编写正确且安全的信号处理函数

1. 处理程序要尽可能简单。 避免麻烦的最好方法是保持处理程序尽可能小和简单。例如，处理程序可能只是简单地设置全局标志并立即返回；所有与接收信号相关的处理都由主程序执行，它周期性地检查(并重置)这个标志。
2. 在处理程序中只调用异步信号安全的函数。 所谓异步信号安全的函数(或简称安全的函数)能够被信号处理程序安全地调用，原因有二：要么它是可重入的(例如只访问局部变量），要么它不能被信号处理程序中断。

3. 保存和恢复errno。 许多Linux 异步信号安全的函数都会在出错返回时设置errno在处理程序中调用这样的函数可能会干扰主程序中其他依赖于分。解决方法是在进人处理程序时把errno 保存在一个局部变量中，在处理程序返回前恢复它。注意，只有在处理程序要返回时才有此必要。如果处理程序调用_exit终止该进程，那么就不需要这样做了。
4. 阻塞所有的信号，保护对共享全局数据结构的访问。 如果处理程序和主程序或其他处理程序共享一个全局数据结构，那么在访问(读或者写)该数据结构时，你的处理程序和主程序应该暂时阻塞所有的信号。这条规则的原因是从主程序访问一个数据结构d 通常需要一系列的指令，如果指令序列被访问d 的处理程序中断，那么处理程序可能会发现d 的状态不一致，得到不可预知的结果。在访问d时暂时阻塞信号保证了处理程序不会中断该指令序列。
5. 用volatile 声明全局变量。 考虑一个处理程序和一个main 函数，它们共享一个全局变量g 。处理程序更新g，main 周期性地读g， 对于一个优化编译器而言，main 中g的值看上去从来没有变化过，因此使用缓存在寄存器中g 的副本来满足对g 的每次引用是很安全的。如果这样，main 函数可能永远都无法看到处理程序更新过的值。可以用volatile 类型限定符来定义一个变量，告诉编译器不要缓存这个变量。例如：volatile 限定符强迫编译器每次在代码中引用g时，都要从内存中读取g的值。一般来说，和其他所有共享数据结构一样，应该暂时阻塞信号，保护每次对全局变量的访问。

```
volatile int g;
```

6. 用sig_atomic_t声明标志。在常见的处理程序设计中，处理程序会写全局标志来记录收到了信号。主程序周期性地读这个标志，响应信号，再清除该标志。对于通过这种方式来共享的标志，C 提供一种整型数据类型sig_atomic_t对它的读和写保证会是原子的（不可中断的）。
7. 信号的一个与直觉不符的方面是未处理的信号是不排队的。因为 pending 位向量中每种类型的信号只对应有一位，所以每种类型最多只能有一个未处理的信号。关键思想是如果存在一个未处理的信号就表明至少有一个信号到达了。

进程调度的时机

1. 当前运行的进程运行结束。
2. 当前运行的进程由于某种原因阻塞。
3. 执行完系统调用等系统程序后返回用户进程。
4. 在使用抢占调度的系统中，具有更高优先级的进程就绪时。
5. 分时系统中，分给当前进程的时间片用完。

不能进行进程调度的情况

1. 在中断处理程序执行时。
2. 在操作系统的内核程序临界区内。
3. 其它需要完全屏蔽中断的原子操作过程中。

进程的调度策略

1. 先到先服务调度算法
2. 短作业优先调度算法
3. 优先级调度算法
4. 时间片轮转调度算法
5. 高响应比优先调度算法
6. 多级队列调度算法
7. 多级反馈队列调度算法

进程调度策略的基本设计指标

1. CPU利用率
2. 系统吞吐率，即单位时间内CPU完成的作业的数量。
3. 响应时间。
4. 周转时间。是指作业从提交到完成的时间间隔。从每个作业的角度看，完成每个作业的时间也是很关键
    - 平均周转时间
    - 带权周转时间
    - 平均带权周转时间

进程的状态与状态转换

进程在运行时有三种基本状态：就绪态、运行态和阻塞态。

1. 运行（running）态：进程占有处理器正在运行的状态。进程已获得CPU，其程序正在执行。在单处理机系统中，只有一个进程处于执行状态； 在多处理机系统中，则有多个进程处于执行状态。

2.就绪（ready）态：进程具备运行条件，等待系统分配处理器以便运行的状态。 当进程已分配到除CPU以外的所有必要资源后，只要再获得CPU，便可立即执行，进程这时的状态称为就绪状态。在一个系统中处于就绪状态的进程可能有多个，通常将它们排成一个队列，称为就绪队列。

3.阻塞（wait）态：又称等待态或睡眠态，指进程不具备运行条件，正在等待某个时间完成的状态。

各状态之间的转换：

1. 就绪→执行 处于就绪状态的进程，当进程调度程序为之分配了处理机后，该进程便由就绪状态转变成执行状态。

2. 执行→就绪 处于执行状态的进程在其执行过程中，因分配给它的一个时间片已用完而不得不让出处理机，于是进程从执行状态转变成就绪状态。
3. 执行→阻塞 正在执行的进程因等待某种事件发生而无法继续执行时，便从执行状态变成阻塞状态。
4. 阻塞→就绪 处于阻塞状态的进程，若其等待的事件已经发生，于是进程由阻塞状态转变为就绪状态。

什么是孤儿进程？僵尸进程？

1. 孤儿进程： 父进程退出，子进程还在运行的这些子进程都是孤儿进程，孤儿进程将被init进程（1号进程）所收养，并由init进程对他们完成状态收集工作。

2。 僵尸进程： 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 获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的这些进程是僵尸进程。

什么是线程?

1. 是进程划分的任务，是一个进程内可调度的实体，是CPU调度的基本单位，用于保证程序的实时性，实现进程内部的并发。
2. 线程是操作系统可识别的最小执行和调度单位。每个线程都独自占用一个虚拟处理器：独自的寄存器组，指令计数器和处理器状态。
3. 每个线程完成不同的任务，但是属于同一个进程的不同线程之间共享同一地址空间（也就是同样的动态内存，映射文件，目标代码等等），打开的文件队列和其他内核资源。

为什么需要线程?

线程产生的原因：进程可以使多个程序能并发执行，以提高资源的利用率和系统的吞吐量；但是其具有一些缺点：

1. 进程在同一时刻只能做一个任务，很多时候不能充分利用CPU资源。
2. 进程在执行的过程中如果发生阻塞，整个进程就会挂起，即使进程中其它任务不依赖于等待的资源，进程仍会被阻塞。

引入线程就是为了解决以上进程的不足，线程具有以下的优点：

1. 从资源上来讲，开辟一个线程所需要的资源要远小于一个进程。
2. 从切换效率上来讲，运行于一个进程中的多个线程，它们之间使用相同的地址空间，而且线程间彼此切换所需时间也远远小于进程间切换所需要的时间（这种时间的差异主要由于缓存的大量未命中导致）。

3. 从通信机制上来讲，线程间方便的通信机制。对不同进程来说，它们具有独立的地址空间，要进行数据的传递只能通过进程间通信的方式进行。线程则不然，属于同一个进程的不同线程之间共享同一地址空间，所以一个线程的数据可以被其它线程感知，线程间可以直接读写进程数据段（如全局变量）来进行通信（需要一些同步措施）。

简述线程和进程的区别和联系

1. 一个线程只能属于一个进程，而一个进程可以有多个线程，但至少有一个线程。线程依赖于进程而存在。
2. 进程在执行过程中拥有独立的地址空间，而多个线程共享进程的地址空间。（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，栈段又叫运行时段，用来存放所有局部变量和临时变量。）
3. 进程是资源分配的最小单位，线程是CPU调度的最小单位。
4. 通信：由于同一进程中的多个线程具有相同的地址空间，使它们之间的同步和通信的实现，也变得比较容易。进程间通信 `IPC` ，线程间可以直接读写进程数据段（如全局变量）来进行通信（需要一些同步方法，以保证数据的一致性）。
5. 进程编程调试简单可靠性高，但是创建销毁开销大；线程正相反，开销小，切换速度快，但是编程调试相对复杂。
6. 进程间不会相互影响；一个进程内某个线程挂掉将导致整个进程挂掉。
7. 进程适应于多核、多机分布；线程适用于多核。

进程和线程的基本API

进程API以Unix系统为例，线程相关的API属于Posix线程(Pthreads)标准接口。

进程原语	线程原语	描述
`fork`	`pthread_create`	创建新的控制流
`exit`	`pthread_exit`	从现有的控制流中退出
`waitpid`	`pthread_join`	从控制流中得到退出状态
`atexit`	`pthread_cancel_push`	注册在退出控制流时调用的函数
`getpid`	`pthread_self`	获取控制流的ID
`abort`	`pthread_cancel`	请求控制流的非正常退出

多线程模型

1. 多对一模型。将多个用户级线程映射到一个内核级线程上。该模型下，线程在用户空间进行管理，效率较高。缺点就是一个线程阻塞，整个进程内的所有线程都会阻塞。几乎没有系统继续使用这个模型。
2. 一对一模型。将内核线程与用户线程一一对应。优点是一个线程阻塞时，不会影响到其它线程的执行。该模型具有更好的并发性。缺点是内核线程数量一般有上限，会限制用户线程的数量。更多的内核线程数目也给线程切换带来额外的负担。linux和Windows操作系统家族都是使用一对一模型。
3. 多对多模型。将多个用户级线程映射到多个内核级线程上。结合了多对一模型和一对一模型的特点。

进程同步的方法

操作系统中，进程是具有不同的地址空间的，两个进程是不能感知到对方的存在的。有时候，需要多个进程来协同完成一些任务。
当多个进程需要对同一个内核资源进行操作时，这些进程便是竞争的关系，操作系统必须协调各个进程对资源的占用，进程的互斥是解决进程间竞争关系的方法。 进程互斥指若干个进程要使用同一共享资源时，任何时刻最多允许一个进程去使用，其他要使用该资源的进程必须等待，直到占有资源的进程释放该资源。
当多个进程协同完成一些任务时，不同进程的执行进度不一致，这便产生了进程的同步问题。需要操作系统干预，在特定的同步点对所有进程进行同步，这种协作进程之间相互等待对方消息或信号的协调关系称为进程同步。进程互斥本质上也是一种进程同步。
进程的同步方法：

1. 互斥锁
2. 读写锁
3. 条件变量
4. 记录锁(record locking)
5. 信号量
6. 屏障（barrier）

线程同步的方法

操作系统中，属于同一进程的线程之间具有相同的地址空间，线程之间共享数据变得简单高效。遇到竞争的线程同时修改同一数据或是协作的线程设置同步点的问题时，需要使用一些线程同步的方法来解决这些问题。

线程同步的方法：

1. 互斥锁
2. 读写锁
3. 条件变量
4. 信号量
5. 自旋锁
6. 屏障（barrier）

进程同步与线程同步有什么区别

进程之间地址空间不同，不能感知对方的存在，同步时需要将锁放在多进程共享的空间。而线程之间共享同一地址空间，同步时把锁放在所属的同一进程空间即可。

死锁是怎样产生的?

死锁是指两个或两个以上进程在执行过程中，因争夺资源而造成的下相互等待的现象。
产生死锁需要满足下面四个条件：

1. 互斥条件：进程对所分配到的资源不允许其他进程访问，若其他进程访问该资源，只能等待，直至占有该资源的进程使用完成后释放该资源。
2. 占有并等待条件：进程获得一定的资源后，又对其他资源发出请求，但是该资源可能被其他进程占有，此时请求阻塞，但该进程不会释放自己已经占有的资源。
3. 非抢占条件：进程已获得的资源，在未完成使用之前，不可被剥夺，只能在使用后自己释放。
4. 循环等待条件：进程发生死锁后，必然存在一个进程-资源之间的环形链。

如何解决死锁问题?

解决死锁的方法即破坏产生死锁的四个必要条件之一，主要方法如下:

1. 资源一次性分配，这样就不会再有请求了（破坏请求条件）。
2. 只要有一个资源得不到分配，也不给这个进程分配其他的资源（破坏占有并等待条件）。
3. 可抢占资源：即当进程新的资源未得到满足时，释放已占有的资源，从而破坏不可抢占的条件。
4. 资源有序分配法：系统给每类资源赋予一个序号，每个进程按编号递增的请求资源，释放则相反，从而破坏环路等待的条件

什么是虚拟地址，什么是物理地址?

地址空间是一个非负整数地址的有序集合。

在一个带虚拟内存的系统中，CPU 从一个有N=pow(2,n)个地址的地址空间中生成虚拟地址，这个地址空间称为虚拟地址空间（virtual address space）,现代系统通常支持 32 位或者 64 位虚拟地址空间。

一个系统还有一个物理地址空间（physical address space），对应于系统中物理内存的M 个字节。

地址空间的概念是很重要的，因为它清楚地区分了数据对象（字节）和它们的属性（地址）。

一旦认识到了这种区别，那么我们就可以将其推广，允许每个数据对象有多个独立的地址，其中每个地址都选自一个不同的地址空间。这就是虚拟内存的基本思想。

主存中的每字节都有一个选自虚拟地址空间的虚拟地址和一个选自物理地址空间的物理地址。

什么是虚拟内存?

为了更加有效地管理内存并且少出错，现代系统提供了一种对主存的抽象概念，叫做虚拟内存(VM)。虚拟内存是硬件异常、硬件地址翻译、主存、磁盘文件和内核软件的完美交互，它为每个进程提供了一个大的、一致的和私有的地址空间。通过一个很清晰的机制，虚拟内存提供了三个重要的能力：

1. 它将主存看成是一个存储在磁盘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。
2. 它为每个进程提供了一致的地址空间，从而简化了内存管理。
3. 它保护了每个进程的地址空间不被其他进程破坏。

为什么要引入虚拟内存?

1. 虚拟内存作为缓存的工具
    - 虚拟内存被组织为一个由存放在磁盘上的N个连续的字节大小的单元组成的数组。
    - 虚拟内存利用DRAM缓存来自通常更大的虚拟地址空间的页面。
2. 虚拟内存作为内存管理的工具。操作系统为每个进程提供了一个独立的页表，也就是独立的虚拟地址空间。多个虚拟页面可以映射到同一个物理页面上。
    - **简化链接：** 独立的地址空间允许每个进程的内存映像使用相同的基本格式，而不管代码和数据实际存放在物理内存的何处。
        - 例如：一个给定的 `linux` 系统上的每个进程都是用类似的内存格式，对于64为地址空间，代码段总是从虚拟地址） `0x400000` 开始，数据段，代码段，栈，堆等等。
    - **简化加载：** 虚拟内存还使得容易向内存中加载可执行文件和共享对象文件。要把目标文件中.text和.data节加载到一个新创建的进程中，Linux加载器为代码和数据段分配虚拟页VP，把他们**标记为无效（未被缓存）** ，将页表条目指向目标文件的起始位置。
        - **加载器从不在磁盘到内存实际复制任何数据，在每个页初次被引用时，虚拟内存系统会按照需要自动的调入数据页。**

    - **简化共享：** 独立地址空间为OS提供了一个管理用户进程和操作系统自身之间共享的一致机制。
        - 一般：每个进程有各自私有的代码，数据，堆栈，是不和其他进程共享的，**这样OS创建页表，将虚拟页映射到不连续的物理页面。**
        - 某些情况下，需要进程来共享代码和数据。例如每个进程调用相同的操作系统内核代码，或者C标准库函数。**OS会把不同进程中适当的虚拟页面映射到相同的物理页面。**
    - **简化内存分配：** 虚拟内存向用户提供一个简单的分配额外内存的机制。当一个运行在用户进程中的程序要求额外的堆空间时（如 `malloc` ），OS分配一个适当k大小个连续的虚拟内存页面，并且将他们映射到物理内存中任意位置的k个任意物理页面，**因此操作系统没有必要分配k个连续的物理内存页面，页面可以随机的分散在物理内存中。**

- 虚拟内存作为内存保护的工具。不应该允许一个用户进程修改它的只读段，也不允许它修改任何内核代码和数据结构，不允许读写其他进程的私有内存，不允许修改任何与其他进程共享的虚拟页面。每次CPU生成一个地址时， `MMU` 会读一个 `PTE` ，通过在 `PTE` 上添加一些额外的许可位来控制对一个虚拟页面内容的访问十分简单。

常见的页面置换算法

当访问一个内存中不存在的页，并且内存已满，则需要从内存中调出一个页或将数据送至磁盘对换区，替换一个页，这种现象叫做缺页置换。当前操作系统最常采用的缺页置换算法如下：

- 先进先出(FIFO)算法：
    - 思路：置换最先调入内存的页面，即置换在内存中驻留时间最久的页面。
    - 实现：按照进入内存的先后次序排列成队列，从队尾进入，从队首删除。
    - 特点：实现简单；性能较差，调出的页面可能是经常访问的
- 最近最少使用（ `LRU` ）算法:
    - 思路： 置换最近一段时间以来最长时间未访问过的页面。根据程序局部性原理，刚被访问的页面，可能马上又要被访问；而较长时间内没有被访问的页面，可能最近不会被访问。
    - 实现：缺页时，计算内存中每个逻辑页面的上一次访问时间，选择上一次使用到当前时间最长的页面
    - 特点：可能达到最优的效果，维护这样的访问链表开销比较大

当前最常采用的就是 `LRU` 算法。

- 最不常用算法（ `Least Frequently Used, LFU` ）
    - 思路：缺页时，置换访问次数最少的页面
    - 实现：每个页面设置一个访问计数，访问页面时，访问计数加1，缺页时，置换计数最小的页面
    - 特点：算法开销大，开始时频繁使用，但以后不使用的页面很难置换

请说一下什么是写时复制?

- 如果有多个进程要读取它们自己的那部门资源的副本，那么复制是不必要的。每个进程只要保存一个指向这个资源的指针就可以了。只要没有进程要去修改自己的“副本”，就存在着这样的幻觉：每个进程好像独占那个资源。从而就避免了复制带来的负担。如果一个进程要修改自己的那份资源“副本”，那么就会复制那份资源，并把复制的那份提供给进程。不过其中的复制对进程来说是透明的。这个进程就可以修改复制后的资源了，同时其他的进程仍然共享那份没有修改过的资源。所以这就是名称的由来：在写入时进行复制。
- 写时复制的主要好处在于：如果进程从来就不需要修改资源，则不需要进行复制。惰性算法的好处就在于它们尽量推迟代价高昂的操作，直到必要的时刻才会去执行。
- 在使用虚拟内存的情况下，写时复制（Copy-On-Write）是以页为基础进行的。所以，只要进程不修改它全部的地址空间，那么就不必复制整个地址空间。在fork()调用结束后，父进程和子进程都相信它们有一个自己的地址空间，但实际上它们共享父进程的原始页，接下来这些页又可以被其他的父进程或子进程共享。

实时操作系统的概念

实时操作系统（Real-time operating system, RTOS），又称即时操作系统，它会按照排序运行、管理系统资源，并为开发应用程序提供一致的基础。 实时操作系统与一般的操作系统相比，最大的特色就是“实时性”，如果有一个任务需要执行，实时操作系统会马上（在较短时间内）执行该任务，不会有较长的延时。这种特性保证了各个任务的及时执行。

优先级反转是什么？如何解决

由于多进程共享资源，具有最高优先权的进程被低优先级进程阻塞，反而使具有中优先级的进程先于高优先级的进程执行，导致系统的崩溃。这就是所谓的优先级反转(Priority Inversion)。其实,优先级反转是在高优级(假设为A)的任务要访问一个被低优先级任务(假设为C)占有的资源时,被阻塞.而此时又有优先级高于占有资源的任务(C)而低于被阻塞的任务(A)的优先级的任务(假设为B)时,于是,占有资源的任务就被挂起(占有的资源仍为它占有),因为占有资源的任务优先级很低,所以,它可能一直被另外的任务挂起.而它占有的资源也就一直不能释放,这样,引起任务A一直没办法执行.而比它优先低的任务却可以执行。

目前解决优先级反转有许多种方法。其中普遍使用的有2种方法：一种被称作优先级继承(priority inheritance)；另一种被称作优先级极限(priority ceilings)。

1. 优先级继承(priority inheritance) 优先级继承是指将低优先级任务的优先级提升到等待它所占有的资源的最高优先级任务的优先级.当高优先级任务由于等待资源而被阻塞时,此时资源的拥有者的优先级将会自动被提升。

2. 优先级天花板(priority ceilings)优先级天花板是指将申请某资源的任务的优先级提升到可能访问该资源的所有任务中最高优先级任务的优先级.(这个优先级称为该资源的优先级天花板)。

我是小牛，非科班转行的微软程序员新人一枚。

我会在这里分享一下自己的转行学习路线、个人经历、内推信息等等！欢迎大家转载我的原创文章，可在公众号下留言！