操作系统

一. 操作系统

操作系统（Operating System，OS）是配置在计算机硬件上的第一层软件，是对硬件系统的首次扩充，是管理计算机硬件与软件资源的程序，是计算机的基石。

操作系统本质上是一个运行在计算机上的软件程序 ，主要用于管理计算机硬件和软件资源。 举例：运行在你电脑上的所有应用程序都通过操作系统来调用系统内存以及磁盘等等硬件。

操作系统的存在屏蔽了硬件层的复杂性。操作系统就像是硬件使用的负责人，统筹着各种相关事项。

操作系统的内核（Kernel）是操作系统的核心部分，它负责系统的内存管理，硬件设备的管理，文件系统的管理以及应用程序的管理。内核是连接应用程序和硬件的桥梁，决定着系统的性能和稳定性。

其设计目标是在计算机系统上配置操作系统，，其主要目标是方便性，有效性，可扩充性和开放性。

作用：

① OS 作为用户与计算机硬件系统之间的接口。
② OS 作为计算机系统资源的管理者。
③ OS 实现了对计算机资源的抽象。

二. 分时与实时系统的特征

1. 分时系统

① 多路性
② 独立性
③ 及时性
④ 交互性

2. 实时系统

① 多路性
② 独立性
③ 及时性
④ 交互性
⑤ 可靠性
⑥ 多路性
⑦ 独立性
⑧ 及时性
⑨ 交互性

三. 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。（这是传统的操作系统进程概念，在引入线程的操作系统中，进程只是拥有资源的基本单位）

1. 特征

1) 动态性

进程的实质是进程实体的执行过程，它由创建而产生的，因调度而执行，因撤销而消亡。

2) 并发性

多个进程实体同存于内存中，且能在一段时间内运行。

3) 独立性

进程实体是一个能独立运行，独立获得资源和接收调度的基本单位。

4) 异步性

各个进程按各自独立的，不可预知的速度向前推进。

2. 状态转换

进程有三个基本状态，就绪状态、执行状态、阻塞状态。

为了满足进程控制块（PCB）对数据及操作的完整性要求以及增强管理的灵活性，通常在系统中又为进程引入了两种常见状态：创建状态和中止状态。

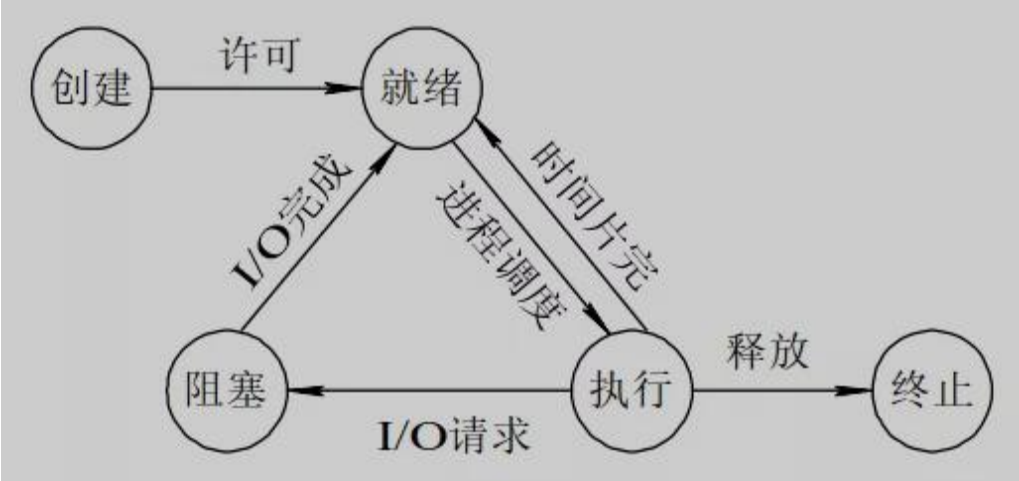

图 1. 进程状态转换

四. 线程

线程是操作系统调度和分派的基本单位。

1. 线程状态

① 执行状态
② 就绪状态
③ 阻塞状态

2. 线程控制块

所有用于控制和管理线程的信息纪录在线程控制块中。数据结构包含：

① 线程标识符
② 一组寄存器
③ 线程执行状态
④ 优先级
⑤ 线程专有存储区
⑥ 信号屏蔽
⑦ 堆栈指针

五. 进程和线程的区别

1. 调度性

在传统 OS 中，拥有资源的基本单位，独立调度和分派的基本单位都是进程。在引入线程的 OS 中，把线程作为调度和分派的基本单位，进程只是拥有资源的基本单位。

2. 并发性

在引入进程的 OS 中，不仅线程间可以并发执行，而且在一个进程内的多线程间，也可以并发执行。

3. 拥有资源

拥有资源的基本单位一直是进程，线程除了一点在运行中必不可少的资源，本身不拥有系统资源，但它可以共享其隶属进程的资源。

4. 独立性

每个进程都能独立申请资源和独立运行，但是同一进程的多个线程则共享进程的内存地址空间和其他资源，他们之间独立性要比进程之间独立性低。

5. 系统开销

在创建或者撤销进程时，系统都要为之分配和回收进程控制块（PCB）以及其他资源，进程切换时所要保存和设置的现场信息也要明显多于线程。由于隶属于一个进程的多个线程共享同一地址空间，线程间的同步与通讯也比进程简单。

6. 支持多处理机系统

传统的进程只能运行在一个处理机上，多线程的进程，则可以将进程中的多个线程分配到多个处理机上，从而获得更好的并发执行效果。

六. 死锁

1. 什么是死锁

死锁（Deadlock）描述的是这样一种情况：多个进程/线程同时被阻塞，它们中的一个或者全部都在等待某个资源被释放。由于进程/线程被无限期地阻塞，因此程序不可能正常终止。

假设有两个进程 A 和 B，以及两个资源 X 和 Y，它们的分配情况如下：

进程	占用资源	需求资源
A	X	Y
B	Y	X

此时，进程 A 占用资源 X 并且请求资源 Y，而进程 B 已经占用了资源 Y 并请求资源 X。两个进程都在等待对方释放资源，无法继续执行，陷入了死锁状态。

① 因为系统资源不足。
② 进程运行推进的顺序不合适。
③ 资源分配不当等。

2. 产生死锁的四个必要条件

1) 互斥

资源必须处于非共享模式，即一次只有一个进程可以使用。如果另一进程申请该资源，那么必须等待直到该资源被释放为止。

2)占有并等待

一个进程至少应该占有一个资源，并等待另一资源，而该资源被其他进程所占有。

3)非抢占

资源不能被抢占。只能在持有资源的进程完成任务后，该资源才会被释放。

4)循环等待

有一组等待进程 {P0, P1,..., Pn}， P0 等待的资源被 P1 占有，P1 等待的资源被 P2 占有，.....，Pn-1 等待的资源被 Pn 占有，Pn 等待的资源被 P0 占有。

注意：这四个条件是产生死锁的必要条件，也就是说只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

3. 解决死锁的方法

解决死锁的方法可以从多个角度去分析，一般的情况下，有预防，避免，检测和解除四种。

预防是采用某种策略，限制并发进程对资源的请求，从而使得死锁的必要条件在系统执行的任何时间上都不满足。

避免则是系统在分配资源时，根据资源的使用情况提前做出预测，从而避免死锁的发生。

检测是指系统设有专门的机构，当死锁发生时，该机构能够检测死锁的发生，并精确地确定与死锁有关的进程和资源。

解除是与检测相配套的一种措施，用于将进程从死锁状态下解脱出来。

1) 死锁的预防

死锁四大必要条件上面都已经列出来了，很显然，只要破坏四个必要条件中的任何一个就能够预防死锁的发生。

破坏第一个条件互斥条件：使得资源是可以同时访问的，这是种简单的方法，磁盘就可以用这种方法管理，但是我们要知道，有很多资源往往是不能同时访问的，所以这种做法在大多数的场合是行不通的。

破坏第三个条件非抢占：也就是说可以采用剥夺式调度算法，但剥夺式调度方法目前一般仅适用于主存资源和处理器资源的分配，并不适用于所有的资源，会导致资源利用率下降。

所以一般比较实用的预防死锁的方法，是通过考虑破坏第二个条件和第四个条件。

a. 静态分配策略

静态分配策略可以破坏死锁产生的第二个条件（占有并等待）。所谓静态分配策略，就是指一个进程必须在执行前就申请到它所需要的全部资源，并且知道它所要的资源都得到满足之后才开始执行。进程要么占有所有的资源然后开始执行，要么不占有资源，不会出现占有一些资源等待一些资源的情况。

静态分配策略逻辑简单，实现也很容易，但这种策略严重地降低了资源利用率，因为在每个进程所占有的资源中，有些资源是在比较靠后的执行时间里采用的，甚至有些资源是在额外的情况下才使用的，这样就可能造成一个进程占有了一些几乎不用的资源而使其他需要该资源的进程产生等待的情况。

b. 层次分配策略

层次分配策略破坏了产生死锁的第四个条件(循环等待)。在层次分配策略下，所有的资源被分成了多个层次，一个进程得到某一次的一个资源后，它只能再申请较高一层的资源；当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源，按这种策略，是不可能出现循环等待链的，因为那样的话，就出现了已经申请了较高层的资源，反而去申请了较低层的资源，不符合层次分配策略。

2)死锁的避免

上面提到的破坏死锁产生的四个必要条件之一就可以成功预防系统发生死锁，但是会导致低效的进程运行和资源使用率。而死锁的避免相反，它的角度是允许系统中同时存在四个必要条件，只要掌握并发进程中与每个进程有关的资源动态申请情况，做出明智和合理的选

择，仍然可以避免死锁，因为四大条件仅仅是产生死锁的必要条件。

我们将系统的状态分为安全状态和不安全状态，每当在未申请者分配资源前先测试系统状态，若把系统资源分配给申请者会产生死锁，则拒绝分配，否则接受申请，并为它分配资源。

如果操作系统能够保证所有的进程在有限的时间内得到需要的全部资源，则称系统处于安全状态，否则说系统是不安全的。很显然，系统处于安全状态则不会发生死锁，系统若处于不安全状态则可能发生死锁。

那么如何保证系统保持在安全状态呢？通过算法，其中最具有代表性的避免死锁算法就是 Dijkstra 的银行家算法，银行家算法用一句话表达就是：当一个进程申请使用资源的时候，银行家算法通过先试探分配给该进程资源，然后通过安全性算法判断分配后系统是否处于安全状态，若不安全则试探分配作废，让该进程继续等待，若能够进入到安全的状态，则就真的分配资源给该进程。

死锁的避免(银行家算法)改善了资源使用率低的问题，但是它要不断地检测每个进程对各类资源的占用和申请情况，以及做安全性检查，需要花费较多的时间。

3)死锁的检测

对资源的分配加以限制可以预防和避免死锁的发生，但是都不利于各进程对系统资源的充分共享。解决死锁问题的另一条途径是死锁检测和解除 (这里突然联想到了乐观锁和悲观锁，感觉死锁的检测和解除就像是乐观锁 ，分配资源时不去提前管会不会发生死锁了，等到真的死锁出现了再来解决嘛，而死锁的预防和避免更像是悲观锁，总是觉得死锁会出现，所以在分配资源的时候就很谨慎)。

这种方法对资源的分配不加以任何限制，也不采取死锁避免措施，但系统定时地运行一个“死锁检测”的程序，判断系统内是否出现死锁，如果检测到系统发生了死锁，再采取措施去解除它。

4)死锁的解除

当死锁检测程序检测到存在死锁发生时，应设法让其解除，让系统从死锁状态中恢复过来，常用的解除死锁的方法有以下四种：

a. 立即结束所有进程的执行，重新启动操作系统

这种方法简单，但以前所在的工作全部作废，损失很大。

b. 撤销涉及死锁的所有进程，解除死锁后继续运行

这种方法能彻底打破死锁的循环等待条件，但将付出很大代价，例如有些进程可能已经计算了很长时间，由于被撤销而使产生的部分结果也被消除了，再重新执行时还要再次进行计算。

c. 逐个撤销涉及死锁的进程，回收其资源直至死锁解除

d. 抢占资源

从涉及死锁的一个或几个进程中抢占资源，把夺得的资源再分配给涉及死锁的进程直至

死锁解除。

七. 虚拟内存

虚拟内存中，允许将一个作业分多次调入内存，需要时就调入，不需要的就先放在外存。因此，虚拟内存实际是建立在离散分配的内存管理方式的基础上的。虚拟内存的实现有以下三种方式。

① 请求分页存储管理
② 请求分段存储管理
③ 请求段页式存储管理

1. 什么是虚拟内存？解决了什么问题？

虚拟内存是操作系统内存管理的一种技术，每个进程启动时，操作系统会提供一个独立的虚拟地址空间，这个地址空间是连续的，进程可以很方便的访问内存，这里的内存指的是访问虚拟内存。虚拟内存的目的，一是方便进程进行内存的访问，二是可以使有限的物理内存运行一个比它大很多的程序。

虚拟内存的基本思想：每个程序拥有自己的地址空间，这个空间被分割成很多块，每块称为一页，每一页地址都是连续的地址范围。这些页被映射到物理内存，但不要求是连续的物理内存，也不需要所有的页都映射到物理内存，而是按需分配，在程序片段需要分配内存时由硬件执行映射(通常是 MMU)，调入内存中执行。

2. 虚拟内存的意义

1) 虚拟内存可以使得物理内存更加高效

虚拟内存使用置换方式，需要的页就置换进来，不需要的置换出去，使得内存中只保存了需要的页，提高了利用率，也避免了不必要的写入与擦除；

2) 使用虚拟地址可以使内存的管理更加便捷

在程序编译的时候就会生成虚拟地址，该虚拟地址并不是对应一个物理地址，使得也就极大地减少了地址被占用的冲突，减少管理难度；

3) 提高了系统的封装性

在使用虚拟地址的时候，暴露给程序员的永远都是虚拟地址，而具体的物理地址在哪里，这个只有系统才了解。这样就提高了系统的封装性。

3. 分页和分段的机制

分页是实现虚拟内存的技术，虚拟内存按照固定的大小分为页面，物理内存也会按照固定的大小分成页框，页面和页框大小通常是一样的，一般是 4KB，页面和页框可以实现一对一的映射。分页是一维的，主要是为了获得更大的线性地址空间。但是一个地址空间可能存在很多个表，表的数据大小是动态增长的，由于多个表都在一维空间中，有可能导致一个表的数据覆盖了另一个表。

分段是把虚拟内存划分为多个独立的地址空间，每个地址空间可以动态增长，互不影响。每个段可以单独进行控制，有助于保护和共享。

八. 常规问题

1. 字节对齐

字节对齐是计算机系统中一种内存分配方式。由于内存的基本单元是字节，而不是位，所以在内存中存储不同类型的数据时，需要按照一定的规则对数据进行排列，以便于提高计算机系统的运行效率和访问速度。

1) 什么是字节对齐

这跟读取数据有关，cpu 读取一次能读取到的内存大小跟数据总线的位数有关，如果数据总线为 16 位，那么 cpu 一次能够读取 2 字节；如果为 32 位那么 cpu 一次可以读取 4 字节，而读取数据是需要消耗时间的，为了提高效率，尽量让同一个数据（变量）能使用最少次数将其读取出来，那么解决办法就是要求每个数据（变量）在其自然边界上，比如说一个 int 类型的变量占 4 字节，那么在存储这个 int 变量的时候编译器将让这个变量的起始地址能够被 4 整除，那么这样就不会导致这个 int 类型的变量明明没有超过数据总线位数（假设是 32 位）却需要 cpu 两次才能将其读取出来，这就是字节对齐。

2) 为什么要进行字节对齐

前面也说了，为了提高效率。进行字节对齐还有另一个原因，那就是平台的原因，其实不是所有的硬件平台都能够访问任意地址上的任意数据，所以这就导致了进行字节对齐的必要性。尤其是结构体，结构体会包含不同类型的数据，如果不进行人为的（编译器做的事）字节对齐，那么就会导致数据不在他的自然边界上，会影响效率，甚至会引发硬件错误。

3) 数据成员对齐规则

结构(struct)(或联合(union))的数据成员，第一个数据成员放在 offset 为 0 的地方，以后每个数据成员存储的起始位置要从该成员大小或者成员的子成员大小（只要该成员有子成员，比如说是数组，结构体等）的整数倍开始(比如 int 在 32 位机为 4 字节，则要从 4 的整数倍地址开始存储。

e. 字节对齐

字节对齐是针对结构体内的数据的对齐，程序员可以使用预处理指令# parama pack(n)来设定默认对齐数值，其中 n 值就是设置的大小(值位 1,2,4,8...)，数据成员本身也有一个字节大小，编译器会选择这两个中小的那个数值作为对齐大小。第一个数据成员从 offset（偏移量）为 0 的地方开始存储。

比如：

PS：

```
#parama pack(2)
int a;
```

a 为整型，占 4 字节，比预处理指令指定的 2 大，所以按照 2 来对齐，也就是说，存储 a 的起始地址必须要能被 2 整除。

f. 整体对齐

整齐对齐也很简单，整体对齐也会比较两个数值的大小，第一个同样还是预处理指令指定的大小 n，第二个是所有结构体的数据成员中字节最大的那个，编译器会选择两个中小的那个，我们设它为值 m。选取之后，编译器会去查看结构体的最后一个数据成员存储之后的后一个地址是否为选取 m 的倍数，如果是 m 的倍数则什么都不需要做，如果不是，那么需要补齐，使得补齐之后的地址能被 m 整除（为 m 倍数)。

PS：

```
#include<iostream>
using namespace std;
```

```
#pragma pack(2)
struct AA{
    int a;
    char b;
    short c;
    char d;
};
#parama pack();
```

第一步：a 占 4 字节，比预设的 2 大所以将 2 作为对齐大小，起始地址 0 是 2 倍数，所以 a 存储的位置区间为[0,3]。

第二步：b 占 1 字节，比 2 小，所以将 1 作为对齐大小，如果紧挨着 a 存储 b，那么起始地址为 4，4 是 1 的倍数，所以存储的 b 位置区间为[4]。

第三步：c 占 2 字节，所以对齐大小为 2，如果紧挨着 c 存储，那么起始地址为 5，而 5 不是 2 的倍数，6 是 2 的倍数，所以需要补一位，从 6 开始存储 c，所以存储 c 的位置区间为[6,7]。

第四步：字节 d 占 1 字节，将 1 作为对齐大小，8 是 1 的倍数，所以可以紧挨着 c 存储 d，所以存储 d 的位置区间为[8]。

第五步：所有数据成员存储完后，需要进行整体对齐，数据成员中占最大字节的是 int a，占 4 字节，比预设的 2 大，所以将 2 作为对齐大小，而存储完最后一个数据成员 d 之后的地址为 9，9 不是 2 的倍数，需要补齐。

所以最后结构体占的区间为[0,9]，也就是说结构体的 size 为 10。

因为是按照顺序存储结构体中的成员，所以即使结构体中的数据成员完全一样，而他们的相对位置顺序不一样的话，那么他们所占的大小也很可能不同。

比如下面的结构体和上面的结构体中的数据成员一模一样，只是顺序位置不一样，下面结构体的 size 却是 8。

PS：

```
#include<iostream>
using namespace std;

#pragma pack(2)
struct AA{
    int a;
    char b;
    char d;
    short c;
};
#parama pack();
```

4)主要目的

提高计算机系统的运行效率和访问速度。在计算机系统中，数据是通过内存总线进行读取和写入的。如果数据没有进行字节对齐，那么在读取和写入数据时，计算机需要进行额外的内存操作，这将影响计算机的运行效率和访问速度。而进行字节对齐后，数据就可以按照整数倍的地址存储，从而减少内存操作的次数，提高系统的运行效率和访问速度。

5)结构体作为成员

如果一个结构里有某些结构体成员，则结构体成员要从其内部最大元素大小的整数倍地

址开始存储。(struct a 里存有 struct b，b 里有 char、int 、double 等元素，那 b 应该从 8 的整数倍开始存储。)

6)收尾工作

结构体的总大小，也就是 sizeof 的结果，必须是其内部最大成员的整数倍，不足的要补齐。

2. 并发和并行有什么区别

并发就是在一段时间内，多个任务都会被处理；但在某一时刻，只有一个任务在执行。单核处理器可以做到并发。比如有两个进程 A 和 B，A 运行一个时间片之后，切换到 B，B 运行一个时间片之后又切换到 A。因为切换速度足够快，所以宏观上表现为在一段时间内能同时运行多个程序。

并行就是在同一时刻，有多个任务在执行。这个需要多核处理器才能完成，在微观上就能同时执行多条指令，不同的程序被放到不同的处理器上运行，这个是物理上的多个进程同时进行。

3. 进程和线程的区别

调度：进程是资源管理的基本单位，线程是程序执行的基本单位。

切换：线程上下文切换比进程上下文切换要快得多。

拥有资源：进程是拥有资源的一个独立单位，线程不拥有系统资源，但是可以访问隶属于进程的资源。

系统开销： 创建或撤销进程时，系统都要为之分配或回收系统资源，如内存空间，I/O 设备等，OS 所付出的开销显著大于在创建或撤销线程时的开销，进程切换的开销也远大于线程切换的开销。

4. 协程与线程的区别

线程和进程都是同步机制，而协程是异步机制。

线程是抢占式，而协程是非抢占式的。需要用户释放使用权切换到其他协程，因此同一时间其实只有一个协程拥有运行权，相当于单线程的能力。

一个线程可以有多个协程，一个进程也可以有多个协程。

协程不被操作系统内核管理，而完全是由程序控制。线程是被分割的 CPU 资源，协程是组织好的代码流程，线程是协程的资源。但协程不会直接使用线程，协程直接利用的是执行器关联任意线程或线程池。

协程能保留上一次调用时的状态。

5. 什么是虚拟内存

虚拟内存就是说，让物理内存扩充成更大的逻辑内存，从而让程序获得更多的可用内存。虚拟内存使用部分加载的技术，让一个进程或者资源的某些页面加载进内存，从而能够加载更多的进程，甚至能加载比内存大的进程，这样看起来好像内存变大了，这部分内存其实包含了磁盘或者硬盘，并且就叫做虚拟内存。