# 操作系统内核

操作系统内核能力：

* 进程调度：管理进程、线程，决定哪个进程、线程使用CPU。
* 内存管理：管理内存，决定内存的分配和回收。
* 硬件管理：为进程和硬件设备之间提供通信能力。
* 系统调用：作为用户程序与操作系统之间的接口，如果应用程序运行更高权限服务时提供。

Linux内核的设计：

* MultiTask：多任务，对于单核CPU可以并发，对于多核CPU可以并行。（并发指时间段，并行指某一时刻）
* SMP：对称多处理，各个CPU地位平等，每个CPU都可以访问完整的内存和硬件资源。
* ELF：Linux系统的可执行文件链接格式，执行时装载到内存中，CPU读取指令和数据从而执行。
* 宏内核：系统内核的所有模块都运行在系统内核态中。（微内核：只保留进程调度、虚拟内存等基本能力，而将驱动程序、文件系统等放在应用空间，这样功能少，可移植性高，但驱动和硬件设备交互时频繁切换到内核态造成性能损耗）

Windows内核：采取混合内核（宏内核和微内核的结合，内核中会有一个小型的内核，其他模块就在这个基础上搭建，整个内核是个完整的程序），可执行文件为PE（.exe、.dll、.sys）。

# 虚拟内存

在多进程环境下，为了使得进程之间的内存地址不受影响，相互隔离，于是**操作系统就为每个进程独立分配一套虚拟地址空间**，每个程序只关心自己的虚拟地址就可以。

好处：实现进程间的隔离+突破物理内存的限制

**虚拟内存可以使得进程运行内存超出物理内存大小**，对于不常使用的内存将其换到硬盘上。

操作系统会提供一种机制，**将不同进程的虚拟地址和不同内存的物理地址映射起来**。（应该是页表）

程序无需关心物理内存的实际分配，只需操作连续的虚拟地址

允许多个进程共享同一物理内存区域（如库文件）

## 内存分段

将内存分为代码段、数据段、堆段、栈段等。

缺点：会产生外部内存碎片、内存交换效率低。

## 内存分页(核心)

分页是把整个虚拟和物理内存空间切成一段段固定尺寸的大小（通常4KB）。

虚拟内存和物理内存之间通过**页表**来映射。

虚拟地址->物理地址由CPU的内存管理单元（MMU）完成

当进程访问的虚拟地址在页表中查不到时（不在物理内存中），系统会产生一个**缺页中断（/异常）**，系统从用户态切换到内核态，分配物理内存、更新进程页表，最后再返回用户空间，恢复进程的运行。（当空闲的物理内存不足时，首先进行内存回收，利用Swap机制将不常用程序写到磁盘中，若仍不能满足则触发OOM机制(Out Of Memory)）

采用**多级页表**：一级页表覆盖到了全部虚拟地址空间，二级页表在需要时创建，节省空间。（时间换空间）

## 段页式内存管理

先划分为多个段，再对每个段划分多个页。

寻址的过程中，需要有很多层表参与，加大了时间上的开销。（TLB：缓存常用页表项，在CPU内部，访问速度快）

## Linux内存管理

主要采用了分页管理（不可避免的分段管理：把所有段的基地址设为0，即所有程序地址空间都是线性地址空间，段只被用于访问控制和内存保护）。

Linux虚拟空间可分为内核态和用户态（32位中各占1GB、3GB，64位中各占128TB）。

用户态分布（自低到高）：**代码段**、**数据段**（已初始化的全局变量、静态变量）、**BSS段**（未初始化的全局变量、静态变量）、**堆段**、**文件映射段**（动态库、共享内存等）、**栈段**。

**预读失效**：被提前加载进来的页，并没有被访问，反而有可能淘汰 LRU 链表前排的位置。解决方法：使用活跃LRU链表和非活跃LRU链表两个链表，预读页加入非活跃链表，被访问时再加入活跃链表。

**缓存污染**：大量数据在很长一段时间都不会被访问。解决方法：提高进入LRU链表的门槛，在内存页被访问第二次的时候再从非活跃链表升级到活跃链表。

## free如何知道要释放多大空间的内存？

malloc 返回给用户态的内存起始地址比进程的堆空间起始地址多了 16 字节，其中包含有内存块大小等描述信息。free对传来的内存地址向左偏移16字节即可分析出内存块大小。

# 32位和64位操作系统的区别？

运行能力不同：64位可以一次性处理8个字节的数据量而32位一次性只能处理4个字节的数据量。

内存寻址不同：32位处理器最大只支持4GB内存，而64位理论可以很大。

运行软件不同：为保证兼容性，64位可以运行32位程序，但32位不可以运行64位程序。

# 线程、进程与协程

* 线程是CPU调度的单位，进程是资源分配的单位。
* 进程是程序的一次动态执行过程，线程是进程当中的一条执行流程，一个进程中可以有多个线程
* 进程拥有一个完整的资源平台（环境），而线程只独享必要资源（栈和寄存器等）。
* 同一进程的各线程之间共享内存和文件资源
* 线程和进程同样具有就绪、阻塞、执行三种基本状态，同样具有状态之间的转换关系。
* 多个进程可以并发，一个进程内多个线程可以并发，线程的并发开销更小（多线程共享空间，线程创建、销毁快，线程间数据传输快，无需页表切换所以上下文切换快）。
* 协程是一种用户态的轻量级线程，与操作系统调用的线程不同，协程的切换完全由用户控制。创建和切换的成本较低

协程在主线程中执行，可以在不阻塞主程序的情况下启动另一段逻辑执行，适合对某个任务进行分时处理。不会阻塞主程序

什么时候用多线程，什么时候用多进程：需要频繁创建和销毁的优先使用多线程、任务相关性较强共享地址空间和可用数据的用多线程、程序需要扩展到多个设备的（分布式）用多进程。

（UnityEngine所使用的API不能被多线程调用。）

# 进程的状态

**运行状态**：该时刻进程占用CPU。

**就绪状态**：可运行，由于其他进程处于运行状态而暂时停止运行。

**阻塞状态**：该进程正在等待某一事件发生（例如IO事件）。（通常会被换出到磁盘：阻塞挂起状态）

**创建状态**：进程正在被创建的状态。

**结束状态**：进程正在结束的状态。

使用**链表**的方式，将具有相同状态的进程控制块（PCB）组成队列。

进程上下文切换：从一个进程切换到另一个进程。

# 线程的实现

用户线程：在用户空间的线程，不由内核管理，多个用户线程对应一个内核线程。

内核线程：在内核空间的线程，由内核管理，一个用户线程对应一个内核线程。

轻量级进程：在内核中支持用户线程，多对多。

# 进程之间通讯的方式？

① 管道：单向传输无格式流数据。匿名管道只能在父子关系进程之间进行通信。数据缓存在内核中。简单，通信效率低，先入先出，**不适合频繁交换数据**。

② 消息队列：存放在内核中的消息链表由标识符标识。进程可以添加信息或读取信息。**不适合大数据传输**，存在内核态和用户态之间的数据拷贝开销。

③ 共享内存：一块虚拟内存被映射到相同的物理内存中。通常与信号量配合使用。多个进程访问同一块内存空间

④ 信号量：一个计数器，用于实现进程间的互斥与同步。P操作：信号量-1，若信号量 < 0 则资源已被占用，进程阻塞等待，否则进程继续正常执行；V操作：信号量+1，若信号量 <= 0则表明当前有进程阻塞，将进程唤醒运行。互斥：信号量初始化为1；同步：信号量初始化为0，保证进程执行顺序。

⑤ 信号：通知进程某个事件已经发生。如硬件来源（键盘按下ctrl+c）、软件来源（kill命令）。进程响应信号：执行默认操作、捕捉信号、忽略信号。

⑥ Socket：用于不同设备间的进程通信，本地也可作为两个进程通信方式。

# 锁

**自旋锁/忙等待锁**

struct lock\_t {

int flag;

}

int CompareAndSwap(int\* ptr, int value) {

int old = \*ptr;

\*ptr = value;

return old;

}

void lock(lock\_t\* lock) {

while (CompareAndSwap(&lock->flag, 1)) {}

}

void unlock(lock\_t\* lock) {

lock->flag = 0;

}

当请求上锁时，若没有其他线程持有锁，即flag为0，进行上锁，将flag置为1；若由其他线程持有锁，即flag为1，则会循环等待直至锁被释放，然后再执行上锁。

**互斥锁/无等待锁**

当没获取到锁时，释放CPU给其他线程。

会有两次线程上下文切换的成本，当被锁代码执行时间很短时不应使用。

**读写锁**

当写锁没有被线程持有时，多个线程能并发持有读锁；而当写锁被持有时，写线程的读操作被阻塞，其他线程的写操作也被阻塞。

根据优先级分为读优先锁、写优先锁、公平读写锁（可基于队列实现）。

**乐观锁与悲观锁**

自旋锁、互斥锁、读写锁均为悲观锁。即访问共享资源前要上锁。

乐观锁：先修改共享资源，再验证有无冲突，若没有其他线程修改资源则操作完成，否则放弃操作。（无锁编程，当冲突概率低且加锁成本非常高时考虑，例如在线文档多人同时编辑）

# 生产者消费者问题

* 生产者在生成数据后放在缓冲区中。
* 消费者从缓冲区取出数据处理。
* 任何时刻只能有一个生产者或消费者可以访问缓冲区。

基于信号量（semaphore）进行处理，采用三个信号量：互斥信号量mutex、资源信号量fullBuffers（用于消费者询问缓冲区是否有数据）、资源信号量emptyBuffers（用于生产者询问缓冲区是否有空位）。

semaphore mutex = 1;

semaphore fullBuffers = 0;

semaphore emptyBuffers = n;

void producer() {

while (true) {

P(emptyBuffers);

P(mutex);

生成数据并放到缓冲区中

V(mutex);

V(fullBuffers);

}

}

void consumer() {

while (true) {

P(fullBuffers);

P(mutex);

从缓冲区读取数据

V(mutex);

V(emptyBuffers);

}

}

# 哲学家进餐问题

五个哲学家五把叉子，只有拿到两把叉子才能进食。

通常的只用五个信号量的解决方法：会发生死锁。

解决方法：

1. 添加一个互斥信号量，只要有一个哲学家准备拿叉子，其他就不能动，一次仅有一位哲学家用餐。

2. 让奇数编号的哲学家先拿左边的叉子再拿右边的叉子，偶数编号的哲学家先右再左。

3. 对每个哲学家记录状态：饥饿、进餐、思考。当饥饿时只有左右两个邻居都不处于进餐状态时用餐。

# 读者-写者问题

读读允许、读写互斥、写写互斥。

即同一时刻允许多个读者读，有写者时不允许读，有读者或写者时不允许写。

解决方法：

1. 写信号量、读者计数、读者计数信号量。（读者优先）

semaphore wMutex = 1;

semaphore rCountMutex = 1;

int rCount = 0;

void writer() {

while (true) {

P(wMutex);

write();

V(wMutex);

}

}

void reader() {

while (true) {

P(rCountMutex);

if (rCount == 0) P(wMutex);

rCount++;

V(rCountMutex);

read();

P(rCountMutex);

rCount--;

if (rCount == 0) V(wMutex);

V(rCountMutex);

}

}

2. 添加flag信号量：读写公平，优先级相同。

semaphore wMutex = 1;

semaphore rCountMutex = 1;

semaphore flag = 1;

int rCount = 0;

void writer() {

P(flag);

P(wMutex);

write();

V(wMutex);

V(flag);

}

void reader() {

P(flag);

P(rCountMutex);

if (rCount == 0) {

P(wMutex);

}

rCount++;

V(rCountMutex);

V(flag);

read();

P(rCountMutex);

rCount--;

if (rCount == 0) {

V(wMutex);

}

V(rCountMutex);

}

# 

# 

# 

# LRU缓存结构、LFU缓存结构

LRU：最近最少使用，使用一个哈希表和一个双向链表表示，哈希表存储键到结点的映射，双向链表按先后顺序存储结点。

LFU：最不经常使用，使用双哈希表表示，其中一个哈希表存储键到结点的映射，一个哈希表存储使用频率到结点链表的映射。

# 死锁

两个或两个以上线程在执行过程中由于竞争资源造成的阻塞现象，两个线程在各自有锁的情况下又去尝试获取对方的锁。

死锁的必要条件：

① 互斥：一个资源只能被一个线程所占有，当资源被占用后其它线程就只能等待。

② 不可抢占：线程不主动释放资源，资源将一直被线程占有。

③ 持有并请求：线程持有一个资源后又请求新的资源。

④ 环路等待：线程资源占有与请求形成了一个环。

死锁的处理方法：

① 程序运行前预防死锁，破坏四个条件之一，如允许抢占资源、给资源编号、规定线程当可以请求到所有资源时再执行上锁。

② 程序运行时避免死锁，银行家算法：判断对请求的满足是否会进入不安全状态。

③ 可以允许系统进入死锁状态，然后检测它，并加以恢复（抢占、回滚、杀死进程）。

④ 鸵鸟算法。当死锁发生概率很低时，可以忽视这个问题，认为死锁不可能在系统内发生。

# 进程调度算法

① 先来先服务：按照请求的顺序进行调度（非抢占式）。

② 短作业优先：按估计运行时间最短的顺序进行调度（非抢占式）。

③ 最短剩余时间优先：按剩余运行时间的顺序进行调度。当一个新的作业到达时，其整个运行时间与当前进程的剩余时间作比较。（抢占式）

④ 时间片轮转：将所有就绪进程按到达先后顺序排成一个队列，每次调度时，每次使队首进程执行一个时间片，然后计时器发出时钟中断，将其调至队列的末尾。

⑤ 优先级调度：为每个进程分配一个优先级，按优先级进行调度。

⑥ 多级反馈队列：时间片轮转调度算法和优先级调度算法的结合。

# 页面置换算法

当发生缺页中断时操作系统必须在内存中选择一个页面将其换出内存以便为即将调入的页面腾出空间。

① ~~最佳置换法~~（OPT）：优先淘汰最长时间内不会被访问的页面。

② 先进先出置换算法（FIFO）：优先淘汰最先进入内存的页面。

③ 最近最久未使用置换算法（LRU）：优先淘汰最近最久没访问的页面。

④ 时钟置换算法（CLOCK）：环形链表，为每个页面设置一个访问位，当访问时将其置为1，淘汰一个页面时检查访问位：若为0则选择该页换出；若为1则置为0并向前移动表针。

⑤ 最不常用算法（LFU）：优先淘汰访问次数最少的页面。

# 动态分区分配算法

① 首次适应算法：每次都从低地址开始查找，找到第一个能满足大小的空闲分区。

② 下次/临近适应算法：由首次适应演变而来，每次从上次查找结束位置开始查找

③ 最佳适应算法：优先使用更小的分区，以保留更多大分区。

④ 最坏适应算法：优先使用更大的分区，以防止产生太小的不可用的碎片。

# 磁盘调度算法

① 先来先服务：先到来的请求，先被服务。

② 最短寻道时间优先：先选择从当前磁头位置所需寻道时间最短的请求。

③ 扫描算法：磁头在一个方向上移动，访问所有未完成的请求，直到到达磁盘末端，然后调换方向。

④ 循环扫描算法：在扫描算法的基础上返回时不处理请求，而是立即复位，只处理一个方向上的请求。

⑤ LOOK/C-LOOK：在扫描算法、循环扫描算法的基础上，移动到最远请求处而不是磁盘末端。

# 文件I/O

**缓冲与非缓冲I/O**：是否利用标准库缓冲。

**直接与非直接I/O**：是否利用操作系统的缓存。

**阻塞与非阻塞I/O**：

阻塞等待的是「内核数据准备好」和「数据从内核态拷贝到用户态」这两个过程。

非阻塞I/O：read请求在数据未准备好时立即返回。应用程序不断轮询内核直至数据准备好。

I/O多路复用：通过 I/O 事件分发，当内核数据准备好时，再以事件通知应用程序进行操作。

**同步与异步I/O**：

以上三种均为同步I/O，内核将数据从内核空间拷贝到应用程序空间的过程需要等待。

aio\_read：异步I/O，内核自动将数据从内核空间拷贝到应用程序空间。

# 键盘输入字母，期间经历了什么？

输入字符后，键盘控制器将数据缓存在寄存器中，向CPU发送中断，CPU保存被中断进程的上下文，然后调用键盘的中断处理程序。CPU从键盘控制器的寄存器中读取到扫描码后，转换成ASCII码，将其放到读缓冲区队列，由显示设备将其再读取到写缓冲区队列，从而进行显示。然后CPU恢复被中断进程的上下文。

# I/O多路复用

一个进程/线程来处理多个Socket（I/O），进程可以通过一个系统调用函数从内核中获取多个事件。

select/poll：使用「线性结构」来存储进程关注的 Socket 集合，需要遍历文件描述符来找到可读或可写的Socket，时间复杂度为O(n)。（BitMap/链表）

epoll：使用「红黑树」存储进程关注的Socket集合；使用事件驱动（回调）的方式，维护链表来记录就绪事件，只将有事件发生的Socket集合传递给应用程序。支持边缘触发（只有在第一次条件满足的时候触发，仅在FD状态变化时通知一次），直接处理就绪事件，无需遍历所有Socket

# 一致性哈希

传统哈希在节点数量发生变化的时候，最坏情况下所有数据都需要迁移。一致性哈希是指将「存储节点」和「数据」都映射到一个首尾相连的哈希环上，如果增加或者移除一个节点，仅影响该节点在哈希环上顺时针相邻的后继节点，其它数据也不会受到影响。

# 

# 

# 字节序：大端模式和小端模式

大端：低地址位存高位字节（更符合阅读习惯）；

小端：低地址位存低位字节。

如果需要逐位运算，或者需要到从个位数开始运算，都是小端序占优势。反之，如果运算只涉及到高位，或者数据的可读性比较重要，则是大端序占优势。

一般操作系统都是小端，而通讯协议是大端的。受硬件影响。

判断大端序还是小端序：

int a = 0x1234;

int d = (char)a;

if (d == 0x12) {

cout << "大端序";

} else if (d == 0x34) {

cout << "小端序";

}

# 浮点数存储结构

32位：1bit符号位，8bit指数部分，23bit尾数部分。范围±(2-2^(-23))×2^127。

64位：1bit符号位，11bit指数部分，52bit尾数部分。范围±(2-2^(-52))×2^1023。