网络操作系统，

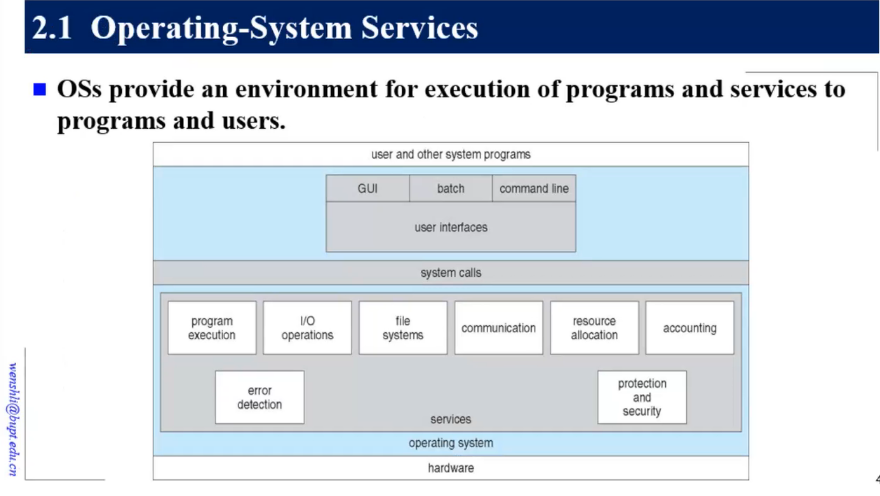
分布式操作系统，

批处理操作系统，

分时操作系统，

实时操作系统:硬实时（刚性实现），软实时（优先级，多媒体，虚拟现实）

嵌入式操作系统



**面向用户：**

UI：

User interface (UI)

Command-Line (CLI)

Batch interface

Grapics User Interface(GUI)

I/O 相关：

键盘打印机

文件读写：

信息交互：

Eg，共享内存空间

消息传递方式

错误检测：

**面向所有计算机用户：**

资源分配模块：

多个进程共享，提出资源请求

（cpu调度模块，内存管理模块，i/o设备使用方式管理模式，不会出现死锁）

监控资源使用状态

收集资源使用信息模块：

记录信息，并分析，对系统改进升级

多进程，并发：

操作系统相互隔离，不能相互干扰

保证受控

抵御外来系统侵入等，授权，权限认证，外来非法访问等。

关于CLI：

需要借助底层，命令解释程序，来支撑使用CLI(命令行方式)使用操作系统。

关于GUI：

随着硬件发展，用户提出需求，实现技术使得操作系统越来越方便，发展起来

触摸屏接口

语音界面（VUI）：

语音识别，语音翻译，语音合成

图像识别（计算机视觉）：

车牌号识别，车辆，视觉理解，车辆行为理解与监控，手势输入

脑机接口（BCI）

分类：侵入式，非侵入式

外核思想：保证资源安全的前提下，减少对应用程序的限制，充分满足应用程序对硬件资源的不同需求。

多核，重核：一个CPU核对应一个操作系统内核，多个内核并行运行，不共享内存。

特点：操作系统以节点形式存在于每一个核上

设计指导原则：

明确内核之间通信方式

每个内核保存一份状态，而不是共享

离散核结构，分布式结构：根据设备划分（还有待讨论）

虚拟机：多用户，多任务并发操作

实现：通过cpu调度模块，将cpu资源分配不同作业，分时复用。

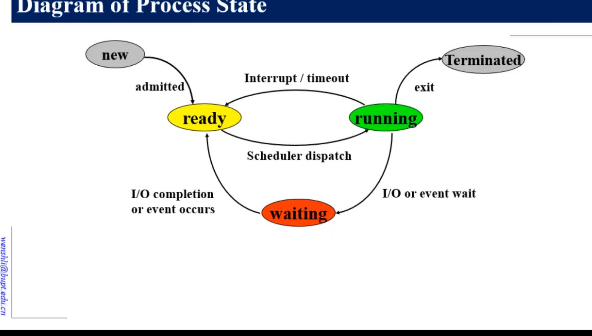
在单处理环境中，每次最多只有一个进程处于running状态

进程：执行中的程序

管理进程，需要获得进程信息，进程管理信息在PCB里面

状态迁移图：

New—被系统接收🡪ready—（被cpu调度模块选中）🡪running(中断返回)—io请求等待—>waiting（条件具备，执行重新排队）



调度：

长期调度：作业，job调度

中期调度：交换

短期调度：CPU调度

进程创建

进程调度，进程迁移

七种调度算法

进程状态迁移

产生阻塞的情况，

没有就绪的模块等待就绪的模块，就会产生阻塞

解决办法：

阻塞是指一个线程在等待某个事件发生时被挂起，无法继续执行下去。解决阻塞的方法取决于阻塞的原因和具体情况，以下是一些常见的解决阻塞的方法：

1. \*\*使用多线程\*\*：将阻塞操作放在单独的线程中执行，避免主线程被阻塞。这样可以保持主线程的响应性，提高系统的并发性能。

2. \*\*异步编程\*\*：使用异步编程模型，将阻塞操作改为非阻塞的异步操作。可以使用回调函数、Promise、Future等机制来处理异步操作的结果。

3. \*\*超时机制\*\*：设置一个超时时间，在等待事件发生的过程中定时检查是否超时，如果超时则放弃等待并执行相应的处理逻辑。

4. \*\*非阻塞IO\*\*：使用非阻塞IO操作，当IO操作无法立即完成时，不会阻塞线程，而是返回一个状态或错误码，可以通过轮询或事件通知的方式等待IO操作完成。

5. \*\*使用线程池\*\*：使用线程池来管理线程的创建和销毁，避免频繁创建线程导致的性能开销。

6. \*\*事件驱动编程\*\*：采用事件驱动的编程模型，通过事件和回调函数来处理异步操作，避免阻塞主线程。

7. \*\*并发编程\*\*：使用并发编程技术，如信号量、互斥锁、条件变量等来控制线程的并发访问，避免线程间的阻塞。

以上是一些常见的解决阻塞的方法，根据具体情况选择合适的方法可以提高系统的性能和响应速度。

内存管理模式都有哪些

内存管理模式是指操作系统或编程语言中用于管理内存分配和释放的不同策略和机制。常见的内存管理模式包括以下几种：

1. \*\*手动内存管理\*\*：程序员需要显式地分配和释放内存，例如在C语言中使用malloc()和free()函数。这种模式需要程序员自行管理内存，容易出现内存泄漏和内存溢出等问题。

2. \*\*自动内存管理\*\*：由编程语言或运行时系统自动管理内存的分配和释放。常见的自动内存管理方式包括垃圾回收（Garbage Collection）和引用计数（Reference Counting）等。

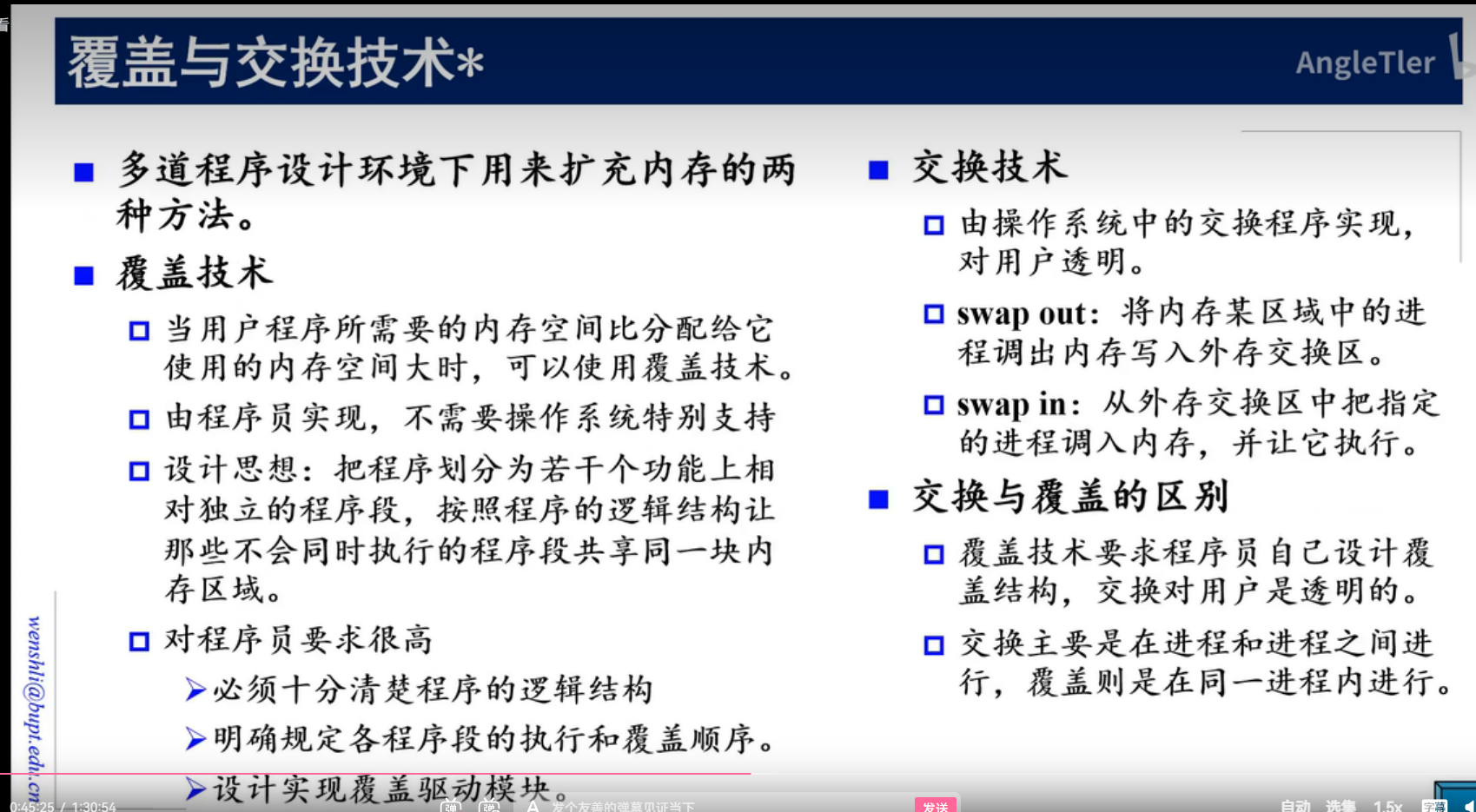
3. \*\*分区式内存管理\*\*：将内存分为不同的区域，每个区域用于存储不同类型的数据。常见的分区式内存管理包括固定分区、动态分区和伙伴系统等。

4. \*\*页式内存管理\*\*：将内存划分为固定大小的页（Page），程序在运行时将数据存储在页中。页式内存管理可以提高内存的利用率和管理效率。

5. \*\*段式内存管理\*\*：将内存划分为不同大小的段（Segment），每个段用于存储不同类型的数据。段式内存管理可以提供更灵活的内存管理方式。

6. \*\*虚拟内存管理\*\*：通过虚拟内存技术将物理内存和虚拟地址空间进行映射，使得程序能够访问超出物理内存容量的数据。虚拟内存管理可以提高系统的稳定性和安全性。

以上是常见的内存管理模式，不同的系统和编程语言可能会采用不同的内存管理策略来满足不同的需求。在实际开发中，选择合适的内存管理模式可以提高程序的性能和可靠性。



关于重定位

重定位是指将程序加载到内存中的不同位置而不需要修改程序本身的地址引用，通常由操作系统负责完成。在了解重定位时，需要注意以下细节：

1. \*\*静态重定位和动态重定位\*\*：静态重定位是在程序加载到内存之前完成地址的修改，而动态重定位是在程序运行时根据需要动态修改地址引用。了解两者的区别和实现方式有助于理解重定位的原理。

2. \*\*地址空间布局\*\*：了解操作系统的地址空间布局对重定位很重要。通常，操作系统将内存划分为代码段、数据段、堆、栈等区域，不同区域的重定位方式可能有所不同。

3. \*\*重定位表\*\*：重定位表是记录程序中需要重定位的地址引用和对应的偏移量的数据结构。操作系统通过重定位表来完成地址的修改。

4. \*\*地址解析\*\*：了解操作系统是如何解析程序中的地址引用，并通过重定位表来找到对应的内存地址的。这涉及到地址转换的过程，包括基地址寄存器、限制寄存器等。

5. \*\*共享库和动态链接\*\*：在动态链接的情况下，程序在运行时需要动态加载共享库，这也涉及到重定位的问题。了解动态链接的原理对于理解重定位很有帮助。

6. \*\*内存保护\*\*：重定位也涉及到内存保护的问题，操作系统需要确保程序只能访问自己的内存空间，防止越界访问和内存泄漏等问题。

7. \*\*性能影响\*\*：重定位可能会带来一定的性能开销，特别是在动态重定位的情况下。了解重定位对程序性能的影响有助于优化程序设计。

总的来说，了解重定位需要深入理解操作系统的内存管理机制、地址转换过程、重定位表的结构和作用等方面的知识。这些知识对于理解程序加载和运行的过程非常重要。