- 操作系统的基本特征：并发、资源共享
    - 资源：硬件资源、信息资源
- **操作系统提供给用户的界面**
    - GUI, 图形用户界面
    - 批处理
    - 命令行
- 中断：外部中断、内部中断（陷入）、异常
- 操作系统双重模式：内核模式、用户模式
- 系统调用类型：
    - Process control 进程控制
    - File manipulation 文件管理
    - Device manipulation 设备管理
    - Information maintenance 信息维护
    - Communications 通信
- 操作系统分为：微内核、宏内核
- 程序与进程的区别
    - 程序是被动执行的
    - 进程是主动的
- 进程是资源分配的最小单位；线程是CPU调度的最小单位
- 进程与线程的不同点
    - 调度：线程作为调度的基本单位，同进程中线程切换不引起进程，当不同进程的线程切换才引起进程切换；进程作为拥有资源的基本单位。
    - 并发性：一个进程间的多个线程可并发
    - 拥有资源：线程仅拥有属于进程的资源；进程是拥有资源的独立单位
    - 系统开销：进程开销较大，线程小
- **PCB里包含**
    - 进程状态
    - 程序计数器
    - CPU寄存器
    - CPU调度信息
    - 内存管理信息
    - 记账信息
    - I/O状态信息
- 上下文切换
    - 内核会将旧进程的状态保存在其PCB中
    - 而后内核后装入经调度要执行的并已保存的新进 程的上下文
- 进程通信
    - 消息传递：通常用**系统调用**来实现，因此需要更多的内核介入的时间消耗
    - 共享内存：仅在建立共享内存区域时需要系统调用，一旦建立了共享内存，所有的访问都被处理为常规的内存访问，不需要来自内核的帮助
- 并发和并行
    - 并发：在一个CPU上执行多道程序设计
    - 并行：在多个CPU上同时进行多道程序
- 线程与属于同一进程的其他线程共享：代码段、数据、文件

- 线程模型：多对一、一对一、多对多

- **多线程的优势**
  - 高响应度：如果对一个交互程序采用多线程，那么即使其部分阻塞或执行较冗长的操作，该程序仍能继续执行，从而增加了对用户的响应程度
  - 资源共享：线程默认共享它们所属进程的内存和资源。代码和数据共享的优点是它能允许一个应用程序在同一地址空间有多个不同的活动线程
  - 经济：进程创建所需要的内存和资源的分配比较昂贵。由于线程能共享它们所属进程的资源，所以创建和切换线程会更为经济
  - 多处理器体系结构的利用：多线程的优点之一是能充分使用多处理器体系结构，以便每个进程能并行运行在不同的处理器上。不管有多少CPU，单线程进程只能运行在一个CPU上。在多CPU上使用多线程加强了并发功能

- 解决临界区问题：互斥、前进、有限等待

- 死锁特征
  - 占有并等待：一个进程必须占有至少一个资源，并等待另一个资源，而该资源为其他程序占有
  - 互斥：至少有一个资源必须处于非共享模式，即一次只有一个进程使用。如果另一个进程申请该资源，那么申请进程必须等到该资源被释放为止。
  - 非抢占：资源不能被抢占，即资源只能在进程完成任务后自动释放
  - 循环等待：有一组等待进程｛P0，P1，……，Pn｝，P0等待的资源为P1所占有，P1等待的资源为P2所占有，……Pn - 1等待的资源为Pn所占有，Pn等待的资源为P0所占有。

- 解决死锁
  - **死锁预防**：破坏上文死锁的四个特征条件之一即可，但这种破坏并不合理
  - 死锁避免：寻找安全序列——银行家算法
  - 死锁检测+死锁恢复：死锁检测采用银行家算法

- 重定位
  - 定义：重定位就是把程序的逻辑地址空间变换成内存中的实际物理地址空间的过程。
  - 编译时：如果编译时就知道进程在内存中的驻留地址，那么就可以生成绝对代码——即生成绝对地址，不可更改
  - 加载时：如果编译时不知道进程将驻留在内存的什么地方，那么编译器就必须生成可重定位代码。
  - 运行时：如果进程执行时可以从一个内存段移动到另一个内存段，那么绑定必须延迟到执行时才进行。
  - **重定位最合适的时机**：运行时重定位（基地址加偏移量）

- MMU（内存管理单元）
  - 作用：完成从虚拟地址到物理地址的映射，将虚拟地址送往总线之前转换成物理地址
  - 为什么使用：地址绑定在运行时重定位，每次都需要根据基地址和偏移地址计算物理地址，这一个过程采用硬件的方式速度更快。

- 碎片

- 外部碎片：内存块加起来能满足一个请求，但是由于不连续（中间有断层），不能用来连续分配
    - 解决方案：紧缩、分页、分段
- 内部碎片：分出去的分区略大于请求的内存长度。这个余下的小内存块属于该分区，但是无法利用

- **页表很大的解决**
    - 层次页表/多级页表
        - 实现思路
            - 将页表再分页，将页表的逻辑地址拆分成多张页表
            - 顶层页表常驻内存，不需要映射的逻辑地址不需要建立页表项
        - 特点：提高空间利用率
    - 哈希页表
        - 实现思路
            - 虚拟地址（逻辑地址）的页号经过哈希函数转换后，指向页表中某个页表项
            - 哈希函数值相同的虚拟页号，指向同一个页表项，它们在那个页表项下组成一个链表
            - 地址翻译时，由虚拟页号哈希后锁定对应链表，搜索链表得到虚拟页号的匹配项
            - 如果找到匹配项，则找到了虚拟页号对应的物理页帧
        - 特点
            - 当一个程序占用大的虚拟地址空间的一小部分时，哈希页表非常合适。
            - 但页面较多会引起冲突，导致其相应的哈希入口负担过重，从而在链表上搜索的代价非常高。
    - 反向页表
        - 实现思路
            - 为每一个物理块设置一个也表项并按物理块号排序，其中的内容则是页号及属于的进程的标志符。
            - 整个系统只有一个页表，对每个物理内存的页只有一条相应的条目。
        - 特点：
            - 减少页表占用的内存空间
            - 有可能需要查找整个表来寻求匹配，增加了查找页表所需要的时间