操作系统

- 第一章 计算机架构
  - 课程大纲:熟练使用操作系统编程
  - 主要内容(框架):

    计算机架构 computer architecture 面向硬件的宏观架构,包含程序员可见的硬件特性;
    程序的设计;
    程序的运行;
    操作系统的形成与发展;
    OS对下作为资源管理者,管理硬件资源;对上作为服务者,为应用程序提供服务
    资源管理者体现在:CPU,内存,外设
    此外的特殊位置:硬盘,对应专门提供文件系统以使用硬盘
    应用程序运行于OS上时的并发处理
    应用程序的相互协作(基于OS提供的工具
  - 图灵机与OS和冯诺依曼架构的关系:

    OS是通用图灵机
    冯诺依曼架构来源于图灵机
  - OS最早在冯诺依曼计算机中提供装入作用,即将要运行的APP从外存(程序的装入)内存中,将控制权转交给应用程序;而装入程序则是最开始时直接写入ROM中 #os作用
  - 以EDSAC为例

    开始时:伪指令+暂停(供操作员进行相关操作)
    运行
    结束时:暂停,防止CPU向下读取指令
    现在的操作系统中,由操作系统提供开始和结束时的暂停指令,运行结束后控制权被转交给OS
    也就是操作系统起到控制权转接的作用 #OS作用
  - 指令集架构:计算机硬件的最顶层

    ISA即instruction set architecture
    ISA指令分为:特权指令和非特权指令
    特权指令指:调用关键资源,仅在内核模式中起作用,例如I/O指令\置中断指令
    CPU分为内核模式和用户模式,又称为 **(内核态\核心态\管态)** 和 **(用户态\目态)**
    内核模式拥有更加多的权限,可执行CPU的全部指令,OS拥有特权
    用户模式:CPU的某个状态位为1:只能执行部分指令
    操作系统的内核构成:
    1 与硬件关联较为紧密的模块,例如时钟管理\中断处理\设备驱动等最底层的模块
    2 运行频率较高的程序,例如进程管理,存储器管理和设备管理等
- 第二章 程序设计与操作系统
  - 栈

    栈的修改是从高位向低位增加,即从下到上
    栈中存储着局部变量和参数,子程序调用的返回地址
    是否出栈的检查是通过硬件完成
  - 虚拟地址与物理地址

物理地址:送到地址总线用以访问内存的地址,DMA一定使用物理地址,从0开始的,连续性的线性地址空间.
虚拟地址:由CPU送出的地址(准确来讲,是CPU从指令中解析出来的地址),一般程序中的地址都是虚拟地址.每个程序都运行在自己的虚拟地址空间中
两者的相互转换是通过MMU(内存管理单元)完成,其中含有页表
如果程序中的地址使用的是物理地址,那么它就是绝对代码,不经过OS的地址分配,而是由程序员自己分配
另外,由编译程序和汇编程序生成的也是绝对代码

- 重定位相关

内存管理单元:MMU (memory management unit)
重定位概念:装入程序时,对目标程序中指令和数据地址修改的过程.
静态重定位:在逻辑地址转换为物理地址的过程中,地址变换是在进程装入时一次完成的,之后不再改变.通过软件完成,不需要增加硬件地址转换机构.
动态重定位:动态运行的装入程序把装入模块装入内存后,并不立即把装入模块的逻辑地址转换,而是把地址转换推迟到程序执行时才进行，装入内存后的所有地址都仍是逻辑地址。这种方式需要寄存器的支持(硬件支持)，其中放有当前正在执行的程序在内存空间中的起始地址。

- 汇编相关

.dlign8 代表存在于能够被8整除的存储单元中
%rsp register stack pointer 堆栈指针寄存器
exit 将直接退出程序回到操作系统

- 存储系统

是指能够为应用程序提供统一的存储器

- IO设备

用于输入和输出,包括:键盘,网卡,打印机,机械臂等等

- 子程序
- 栈帧相关的结构
- 程序设计语言
  - 机器语言

    计算机硬件能够直接理解的语言.形式表现为二进制代码
    机器语言是软件和硬件的界面
    机器代码:用机器语言编写的程序
  - 汇编语言

    标识符代替二进制代码
    汇编语言中的语句几乎和机器语言的一一对应.(不对应的包括伪代码,例如abc segment/ends)等等
    汇编程序是将汇编语言写的程序转变为机器代码
  - 高级语言

    与机器语言之间没有明确的映射关系(在编译到形成obj文件过程中会发生许多改变)
    编写出来的程序独立于计算机的体系架构
    远非自然语言
    有专门的程序将之转换为机器代码(编译和解释程序),(编译程序:高级语言->机器代码,静态的),(解释程序:一句话或一个代码块有一个专门的子程序进行执行,例如
    python,java,basic,DOS,Shell)

- 运行时系统
  - 编译程序在将高级语言代码生成可执行文件时,在obj->exe的过程中,添加相应的运行时代码,为用户程序的运行提供一定的管理工作.这部分代码就是运行时系统.例如JVM和沙箱
  - 运行时系统相较于普通的用户程序
    - 其运行中 调用子程序时的压栈入栈,临时变量的内存分配与销毁,参数的传递都由运行时系统完成.
    - 用户程序全部出现在编译后的目标文件中,而运行时系统的代码仅出现在可执行文件中.
  - 运行时系统和程序库的最大区别在于
    - 普通的程序库是用户程序去主动调用完成用户的意图,运行时系统是对用户程序的运行起支持作用,不是用户程序主动调用的.
    - 在OS眼中并不区分运行时系统和程序库.
  - 与操作系统的不同在于
    - 用户程序运行时,其跟随进入内存,程序结束时,其跟随退出.但操作系统一直都在内存中运行.
    - **运行时系统只为特定程序设计语言的用户程序提供更加个性化的\跨平台的服务**,而操作系统面向整个计算机系统
  - 实现运行时系统和系统调用的程序的本质区别
    - 实现运行时系统的程序,服务于应用程序但也属于应用程序,在用户态下运行
    - 实现系统调用的应用程序,服务于应用程序,但是由操作系统提供机制,在内核态下运行.
  - 运行时系统还支持:多线程支持,类型检查,堆栈的创建和释放,垃圾回收
  - 高级语言中,运行时系统负责在程序末尾添加(停机的)系统调用
- 程序的链接
  - 单个模块可以执行编译,但不能运行
  - 程序的链接所要做得,就是在链接过程中,把所有引用他们的地方填上符号名新地址.包括链表法和间接地址法两种方法.

    链表法:建立一个链表,记住所有引用的位置.当地址确定后再回填.相当于,当我在一个程序中使用了15次外部变量y,当我链接且y地址确定后,要进行15次回填.即依次都填上.
    间接地址法:一个存储单元,存放符号y地址.所有对y得引用都间接访问该单元.当地址确定后,仅回填该单元.对于上述例子,当y地址确定后,仅需要进行一次回填即可.
  - 链接的方式分为两种,静态链接和动态链接.对应于两种链接库:libs和dll

    静态链接:程序运行前连接完成
    动态链接:程序运行中实时链接
- 其他相关内容
  - 栈

    执行函数时,函数内部的局部变量都创建于栈上,函数执行结束时,这些存储单元被自动释放.例如递归函数.(也即,如果在函数内声明较大的存储空间,会造成栈溢出)

- 堆

  动态内存的分配.由程序运行时根据程序中的malloc和new进行内存的申请.由程序员自主释放内存,不释放造成内存泄漏.

- 可执行文件

  包含有链接程序和装入程序、节（具有相同特征的代码或数据）、程序表头
  作用：存访代码和数据。不运行时处于折叠状态，运行时展开。（以节省存储空间）

- 第三章 程序运行与操作系统
  - 程序的装入

    操作系统将exe文件装入内存是指:
    创建一个独立的虚拟地址空间(创建映射函数所需要得相应的数据结构)
    读取可执行文件(磁盘->物理内存),并且建立虚拟空间与可执行文件的映射关系
    将CPU得指令寄存器设置为可执行文件得入口地址,启动运行.
    将程序从磁盘中读入物理内存,为其分配足够的空间

  - 指令的执行

    不同类型中断事件的比较

属性	意外事件	指令流改变	内部中断	外部中断
异常	√	√	√	×
设备中断	√	√	×	√
陷入	×	√	√	×
子程序调用	×	×	×	×

    一条指令是原子的,在执行过程中不可中断.
    指令流的运行环境是?上一条指令留下的全部信息

    - 异常(CPU自身产生的中断信号)也叫做内中断(被动)

      异常分为可以被修复的异常和不可以被修复的异常
      异常处理的意义?(P17)

    - 陷入(程序主动寻求帮助)(访管中断/软件中断)

      例如断点调试和系统调用
      通常由线性的指令引起,在一条指令的执行过程中CPU也可以响应陷入
      陷入处理程序提供的服务为当前进程所用,在当前进程的上下文中执行
      而中断处理程序不是为了当前进程,在系统上下文中执行

    - 中断(设备请求信号)(来自于设备/人的请求)(外中断)

      例如I/O操作完成或出错,打印机和键盘等

    - 中断机制

      暂停当前,转去执行其他
      中断与指令流无关,指的是其与子程序调用的区别

    - 中断向量

      异常\陷入\中断统一编号,是指中断处理程序的入口

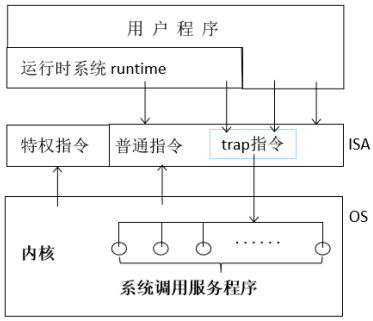
    - 保护断点和上下文

      概念:PSW 程序状态字,包括进位位,中断使能位,CPU模式位等
      上下文:程序运行环境(指寄存器等状态信息)
      流程演视:

      - 保护现场1(主要保存PC和PSW,后面保存时两者已经发生了改变)

        - 查询中断号
        - 计算中断处理程序入口地址
        - 关中断
        - 调用中断处理程序
            - 保护现场2(主要保存寄存器的内容,要先确定保存哪几个寄存器,而不是要全部保存)
    - (陷入)(陷入)

      多任务系统中,OS提供服务与管理.#os作用
      系统调用的服务程序在内核态下运行,当运行完成退出调用时,将恢复到用户态.
      OS会检测发出系统调用的程序的权限.
        - 与API的区别在于:系统调用不可跨平台,但API可以跨平台.例如C中,fopen可以跨平台, 但是open不可以
- 运行时系统
- (陷入)
    - 一种机制,是OS面向应用程序提供的服务,内核态下运行
    - 在#计算机系统 中的地位

    - 可以提供最底层的应用程序接口(API)

      常见的API有POSIX和C库函数(即C语言支持的API)
      一个API函数可以由一个系统调用实现,也可以通过调用多个系统调用来实现.
      C库提供了POSIX的绝大部分API,同时内核提供的**每个系统调用**在C库中都具有相应的封装函数.
      系统命令:位于C库的更上层,是利用C库实现的可执行程序,比如最为常用的ls,cd等命令.
      内核函数:内核函数在内核中实现.应用程序通过系统调用进入内核后,会执行各个操作系统调用对应的内核函数.
      四者的关系:系统命令->C库函数->系统调用->内核函数
    - API与系统调用的区别:看其是否可跨平台.API可以跨平台.
- 程序的终止
    - 停机指令(特权指令,由操作系统执行)(暂停CPU)

      通过(HALT指令或寄存器标志位)使CPU脱离时钟信号,进入暂停状态,但此时整个计算机系统并没有停下来.
      用户程序结束时仅仅将控制权交回操作系统,其本身没有权限执行停机
      用户程序的最后一条指令,是exit(),结束当前的应用程序,将控制权转交给操作系统.
    - 激活CPU(中断信号)

> 中断信号会激活CPU,例如通过按动键盘,产生中断,进而唤醒系统休眠.

- 操作系统的运行环境
  -

- 第四章:操作系统的形成和发展
  - 早期人机交互
    - 作业:用户在系统中完成一项相对完整的工作.
    - 作业控制的评价标准:CPU利用率

      > CPU利用率=(CPU执行时间)/(CPU执行时间+CPU空闲时间)
    - 提高CPU利用率是操作系统发展的动力.此时的人工操作是限制条件.
  - 批处理
    - 定义:以作业流为前提 ,将一类作业(例如Fortran作业)放在同一时间段进行处理,这样一次进行多个作业只需要一次编译程序的装入.人工操作仍是限制条件.
  - 作业流的自动控制:
    - 通过监控程序,减少乃至避免了人机交互操作.此时输入输出影响CPU的效率

      > 监控程序:计算机启动后,装入并且执行监控程序;从磁带读入一个作业,装入到内存执行,依次处理直到结束.
      > 对应衍生:作业控制语言JCL(Job Control Language)
  - 脱机输入输出
    - 思想:使用卫星机(专用于I/O的小计算机)连接设备,内容转存到磁带中.主机读写磁带内容完成输入和输出.
    - 批处理程序的优势和劣势P18
    - 局限性:及时性差,自动化程度高,但早期技术
  - 多任务
    - 多个作业;作业之间具有切换,CPU保持忙. (繁忙的售票员
    - 并行与并发:
      - 所谓并行,发生在多CPU系统中.当一个CPU执行一个进程时,另一个CPU可以同时执行另一个程序.两个进行不抢占CPU资源,可以同时进行.

        > ---
        > ---
      - 所谓并发,是指对于一个CPU而言,表面看上去"同时"执行多个任务,但实际上在操作系统中,是指一个时间段中有几个程序都处于已启动运行到运行完毕之间,并且这几个程序都是在同一个处理机上运行.只是由于时间片的划分,导致CPU不停在不同任务之间切换,看起来像是"同时发生".

        > - - - - - - - - - - -
        > - - - - - - - - - -
    - 多任务的实现:
      - CPU切换于多任务之间
      - 内存中同时装入多个作业
      - 设备共享

        - 作业之间不相互影响
- 分时
    - 假定n个交换任务,轮流执行,每次一个时间片Δ.那么每个任务等待时间(n-1)Δ.事实上大于这个值,因为CPU切换任务需要进行保存等操作,花费时间,即系统开销.
    - 意义:提高了CPU利用率,具有足够快的响应时间和交互性,更加友好
- 操作系统角色#os作用
    - 服务
        - 监控程序,装入作用
        - I/O程序库,输入输出服务
    - 管理和控制
        - 多任务
        - 分时
    - 目标:提高CPU利用率和用户友好性
- 内核
    - 相关描述
        - 访问权描述:二元组(A,R) :对A可读
        - 保护域:访问权的集合,描述一个对象能做的所有事
    - 用CPU模式定义访问权
        - 指令集架构特权指令:操作关键资源的指令, 例如控制I/O开关和中断开关
        - CPU运行模式,内核态下可以执行ISA的全部指令.

            初始,系统加电,CPU内核模式.
            进入用户程序之前,更改CPU为用户模式
            退出用户程序之后,更改为内核模式
        - 内核尽量把CPU让给用户程序.

            非必要,不行动.
            通过响应中断和系统调用,控制整个系统,例如时钟中断10ms/次
            激活CPU(中断信号)**是激活内核的手段.内核是中断驱动的**
        -
    - 内核为应用程序提供管理和服务
- 操作系统结构
    - 结构
        - 操作系统代码划分为:内核空间,外部管理程序(系统程序,即装机时自带的如NotePad\播放器等)
        - 此外还有应用程序和用户进程.
        - 操作系统中的外部管理程序和上述,都属于外部空间.
    - 分层设计:

        系统分为若干层,高层建立于低层的基础之上.只能由高层程序调用低层程序
    - 微内核

将内核功能尽可能缩小.小到只负责模块之间的通信.(交互,线程管理,交互通信)
更容易移植和扩展.更加可靠和安全.但由于模块之间通过内核通信,性能会降低,不容易得到良好的整体优化

- 模块化

面向对象的技术.自顶向下设计分模块,模块之间一定要保持低耦合.
例如LKM(loadable Kernel Module) 可装载内核模块
具有分层的特性,将微内核作为最底层
具有微内核特性,区分核心模块和其他

- 操作系统的研发于安装
  - 策略与机制分离

策略指目标
机制指手段

  - 安装方法

重新编译(将原有代码进行一次编译,成为符合当前ISA的二进制文件)
重新连接(动态链接库,随环境变化进行不同的链接操作)
运行时的表驱动(???

  - 操作系统的引导:
    - 多级引导

从ROM获取引导程序1
再到硬盘主引导记录中获取引导程序2和分区表
到硬盘活动分区的第一个记录,获取引导程序3

- 第五章:CPU管理
  - 碎知识点
    - 程序的并发执行,没有顺序性是因为在执行某个程序时,不一定下一步将会继续执行当前程序的后续程序还是并发程序的某一部分程序,这个选择是随机的.

同一个程序的执行是顺序执行,程序间的执行顺序是随机的.

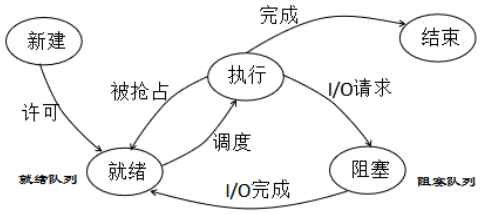
    - n个CPU的并行,可以通过一个CPU的并发实现,并且两者之间的时间具有大致的倍数关系.
    - 顺序执行:

顺序执行条件下,程序执行过程与程序一一对应.
同一个程序在不同的执行过程中,永远都是得到相同的结果和序列(或者说执行过程)

    - 并发执行:

并发执行条件下,程序执行过程与程序不再一一对应.此时同一程序在计算机中的执行可能不再相同.

  - 线程(类比于人)
    - 是OS进行CPU分配的基本单位
    - OS只关心程序的执行过程.程序在虚拟CPU中的执行过程,被称之为线程.
    - 线程的状态:新建,就绪,执行,阻塞,结束



相关内容:
就绪队列和阻塞队列都是线程控制块的链表队列.
调度是通过调度程序.当被选中后,可以进入执行,获得CPU的控制权.
被抢占:当其在执行过程中被更高优先级的抢占时,其状态回到就绪.
I/O请求:发生I/O请求,其会进入等待状态,此时CPU再次被转交给其他就绪线程.
I.O完成:当中断信号发送给CPU并且输入输出完成时,再次转变为就绪状态.

- 线程控制块TCB:(类比于人的档案),包括标识,状态,上下文(CPU中的全部寄存器).(Thread Control Block
- OS对线程的管理操作:创建\撤销\调度\阻塞\唤醒

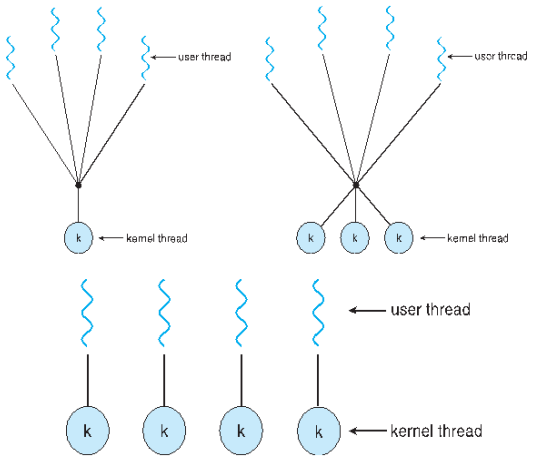
- 进程(类比于家庭).双击exe文件时,实际上是创建了一个进程
  - 是OS进行设备和资源分配的基本单位.
  - 资源保护与共享
    - 多任务系统,应用程序之间共享资源.为了互不影响,进行隔离.即时间或者空间上的分割.隔离实现:为每个任务构造虚拟运行环境.

      常见的有空间上的隔离,例如内存的隔离,通过上下限寄存器完成
      时间上的隔离,通过时间片的分片完成.
    - 虚拟运行环境:虚拟CPU,地址空间(虚拟地址)和设备(设备分配/假脱机技术).
  - 进程:
    - 一组线程及其所依赖的虚拟运行环境构成了进程
    - 一个进程至少有一个线程,还可以通过OS系统调用或者线程库创建新的线程.这些线程之间共享当前的运行环境.因此**不同的线程可以运行同一段代码**
    - 由进程所控制的环境,包括:数据段\代码段\堆\虚拟内存中的地址划分.
    - 由线程控制的环境,包括:运行栈,上下文,代码,数据,堆空间等.即不同的线程之间不共享栈.
    - 进程控制块(PCB):Process Control Block
      - 线程TCB
      - 资源描述(虚拟内存空间,设备)
      - 资源映射关系
      - 进程的档案(进程标识\状态)
  - 多线程进程
    - 多线程进程中,线程之间是并发的.共享的资源包括由进程所控制的环境,包括:数据段\代码段\堆\虚拟内存中的地址划分.例如文件指针 也属于上述.
    - 多线程进程之间,线程相互隔离,进程虚拟机之间没有交集
    - 进程的并发存在限制.

- 多任务
  - 创建新进程的代价要远高于创建线程.因为创建进程对应创建环境,创建线程对应分配资源.
  - 一个线程最多只能运行于一个CPU中.
  - 与之相关的,在一个进程中的两个不同线程,如果是内核线程,那么可以调度到多个CPU上;如果是用户线程,那么就不能在多CPU上并行运算.这是因为,内核级线程由OS创建,系统调用可以实行CPU的分配,但用户级线程由线程库创建,无法完成CPU调度
  - 上下文切换通常指代:同进程中的线程切换. 如果是进程切换,其花费的代价将会很高,例如Cache.TLB.内存等
- 两种不同的线程
  - 内核级线程:由操作系统完成管理的线程.
  - 用户级线程:由用户程序(线程库)自己进行管理(创建\撤销\调度)的线程
  -

	内核级线程	用户级线程
切换方式	抢占式(中断后调度,高优先级优先	非抢占式,通过让出(yield函数)完成切换
切换代价	比较大.Cache等命中降低	绿色线程,代价较小.因为引起的变化相对较小,用到的变量还是同一进程中的全局变量
阻塞	线程阻塞时,只会阻塞本身	I/O阻塞.通过调用yield完成CPU的让出和调度程序的调用,以重新分配
多核	适用于多核\多CPU	不适用于多核,多CPU

  - 两者的关系:(用户级线程-内核级线程)

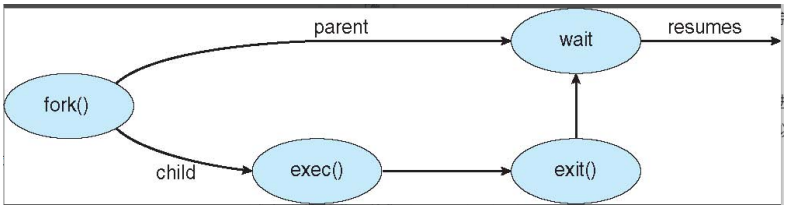


    - 多对一:一个用户级线程阻塞,对应的内核级线程以为自己阻塞,同进程中其他的所有线程都被阻塞.
    - 一对一:所有线程都是内核级线程
    - 多对多:相当于多次的一对多
    -

- 进程创建与撤销
  - 进程的创建

    需要先创建进程控制块,其内包含着进程标识符.
    再创建虚拟运行环境,包括进程的线程,虚拟地址空间和设备等
    最后初始化
  - 程序的撤销:三种情况

    所谓的撤销即:释放进程的全部资源.进程本身无法释放所有资源.例如进程控制块,应该由父进程释放.
    - 1 所有线程执行结束,正常撤销进程
    - 2 进程退出. 进程调用exit(),返回到父进程的wait()
    - 3 父进程想撤销子进程. 又包含三种:子程序运行有问题,调用abort(); 不再需要子进程; 父进程退出
  - 程序撤销造成的结果:
    - 僵尸进程:进程结束但是父进程没有执行wait()为其收尸.可以被OS发现
    - 孤儿进程:进程结束时,父进程就已经结束了.此时的进程将被init进程收留.(即根进程)
  - 父子进程关系
    - 原则上两者并发执行(或者父进程等待子进程结束
    - 原则上彼此独立,有各自的虚拟运行环境
    - 多种意义上的资源共享
  - Unix创建子进程实例COW

    cow: copy on write 当要写入时才进行复制,并且不一定是整块复制.当不写入时,共用一块资源进行读取
    创建实例:
    fork(); //相当于复制
    exec();
- CPU调度
  - CPU调度器的机制:调度与分派.策略:调度算法.
  - 何时执行调度器:
    - 当前线程执行结束
    - 当前线程主动放弃CPU
    - 当前线程阻塞
    - 就绪队列中出现高优先级的线程(抢占式
    - 时钟中断(抢占式

- 线程的执行过程实际上分为CPU执行期和I/O执行期

  CPU执行期花费的时间是可预估的.公式如下:
  假设首个CPU执行期预测为S0,实际为T0
  则下一个CPU执行期的预测时间表示为 `a*(T0)+(1-a)*S0`
  同理,后续可表示为
  `Sn=a*T( n-1)+(1-a)*Sn`
  - 抢占式与非抢占式

    抢占式:CPU什么时候\什么位置离开当前进程,是程序员无法预料的
    两者的比较:
    抢占式及时,但系统开销比较大,切换频繁
  - 空闲进程:当就绪队列为空时,CPU执行空闲进程,此进程不断获得CPU控制权然后立刻放弃(yield()).
  - 调度(三级调度)
    - 作业调度(高级调度) (外存->内存) (发生频率最低,长期调度)

      选择**部分**作业进入CPU运行
      在所有想进入系统运行的作业中,选择**一批作业**进入系统(进入后不一定立即执行,可能还在就绪队列)
      按照某种算法在**外存**中处于后备队列的作业中挑选一个(或多个)作业,给它分配内存等必要资源,并建立相应的进程(即PCB),使之获得竞争的权力.
    - 交换调度(中级调度) (外存->内存) (发生频率中等,中期调度)

      被调到外存中等待的进程处于**挂起态**.此时该进程的代码段和数据段被**调回**外存,但PCB依旧存储于内存中.当满足一定条件 吧,(例如被**从挂起队列中选中时**)会被调入内存.
      选择哪些程序或者数据留在内存,这种技术称之为交换
    - CPU调度(低级调度) (内存->CPU) (发生频率最高,短期调度)

      按照算法,从就绪队列中选择一个进程,为其分配处理机资源
  - 其他相关调度:例如I/O调度
- 调度算法
  - 调度算法的相关概念

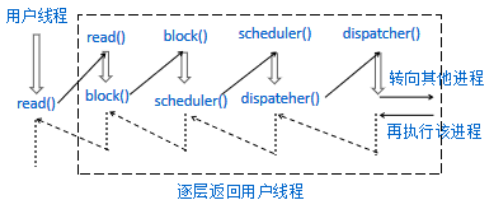
    CPU利用率: `(CPU运行总时间-CPU运行空闲时间)/CPU运行总时间`
    吞吐量:单位时间内完成的进程个数(宏观概念)
    等待时间:线程在就绪队列中的时间
    响应时间:从用户提交请求到系统做出反馈的时间
    运转周期:进程等待时间+运行时间/或者说是: 从进入就绪状态开始到进入阻塞状态(或执行完毕)为止,所花费的时间
  - 综合指标

    平均等待时间W=等待时间之和/n
    平均周转时间T
    平均带权周转时间(T/C) C为实际执行时间
  - CPU调度算法
- CPU分派
  - 线程的上下文:即CPU内的寄存器(通用和PC等等)+CPU外的寄存器(如中断屏蔽寄存器)

- 在进行线程的切换时,要调用CPU调度程序和CPU分派程序
- 下图为调用过程

CPU分派程序的调用过程

- CPU分派程序的调用实例
  - -read()
  - -block()
  - -scheduler()
  - -dispatcher()
- 虚拟CPU
  - -程序员仅看到了read()层



- 分派程序实例:
  - SWITCH函数,含有两个参数P1进程指针,P2进程指针.

- 进程的通信
  - 线程之间共享:多线程进程中,线程之间是并发的.共享的资源包括由进程所控制的环境,包括:数据段\代码段\堆\虚拟内存中的地址划分.例如文件指针 也属于上述.
  - 进程通信的两种模式
    - 消息传输

      通常是指消息队列传输,每一个进程控制块都对应拥有一个消息队列
      消息传输通常由可以细分为两类
      - 直接通信

        要求对方在线,每个进程都有一个消息队列.
        发送消息时显示指定对方ID,即发送给进程A:send(A,message)
      - 间接通信

        在通信前需要建立邮箱
        双方可以不同时在线
    - 共享内存

      共享内存的特点:多个进程都可以访问各自的虚拟内存区域
      进程虚拟地址映射到相同的物理地址
      可以理解为:进程A将块x作为共享内存,告诉OS.此时B告诉OS想要用x作为贡献内存,则OS在进行映射时,会将x作为B地址的一部分映射到B的虚拟地址
      - 共享内存需要特别注意,内存的共享区访问存在互斥,也就是需要在一个进程执行写操作的时候对该共享区加保护

- 第六章 内存管理
  - 连续内存分配
    - 固定分区

      操作系统事先将内存划分成多个区域，分区个数一般是固定的
      每个分区在每个时刻只能分配给一个作业（运行一道程序）
      这些分区大小、位置是固定的
      分区大小可以相等，也可以不等
      - 相关概念

内碎片:被分配出去但是无法被利用的内存空间
作业装入分区:每个作业被分配到一个**空间最小**,且能容纳该作业的**空闲分区**

- 缺点

分区大小固定,会导致小作业占用大空间,导致大作业找不到合适的分区
分区的个数是固定的,限制了系统中进程的并发程度

- 实现

通过地址变换机构实现固定分区
所谓的地址变换机构即:基址寄存器(base register),重定位寄存器(relocation register),限长寄存器(limit register)
这里涉及到重定位,重温一下重定位

- 重定位相关

- 动态分区

事先不对内存进行划分，而是根据作业的实际大小为其分配内存

- 相关概念

外碎片:当某程序退出后的空闲空间又装进了一个更小的进程,那么会产生空闲分区
如果这个空闲分区太小,以至于无法满足任何作业的需求,就会产生外碎片

- 实现

动态分区的实现,例如分区分配表\地址变换机构\地址变换过程,以及存储保护措施(**基址寄存器和限长寄存器**) 都和固定分区的实现相同或者相似

- 动态分区分配算法
  - 首次适配(FF:first-fit):在空闲区列表中按照分区地址序检索,第一个能够容纳的被选中.开销小,算法简单.
  - 最佳适配(BF:best-fit):在所有能够容纳的分区中,选择最小的空闲分区,该方法**容易产生碎片**
  - 最差适配(WF:worst-fit):在所有空闲分区中,选择最大的空闲分区.该方法不容易产生碎片
  - 碎片整理

为了处理系统运行过程中产生的碎片.通过调整进程占用的分区位置,来减少或避免分区碎片的.其中包含很多种方式,包括**碎片紧凑,分区对换.**

- 碎片紧凑

应用条件:所有应用程序都可以重定位.即可以等到执行命令的时候才生成内存地址.
时机:进程处于等待状态时搬动
开销:移动开销

- 分区对换

抢占并回收处于等待状态进程的分区,即将等待状态进程的数据存储到外存中.
把即将要运行的进程移动到内存中.完成置换,内存空间被重复使用.

- 小结

存储在连续的空间中
容易实现
支持多道程序设计技术
无法实现内存共享
不支持虚拟存储

易产生碎片
对于大作业不友好

- 分段

引入:物理地址空间是线性的,程序的结构是模块化\非线性的.
因为模块之间的无序性,程序员可以将不同模块装入不同的分区中.
以程序的一个或者几个模块作为内存分配单位.
分段与连续分配的不同:分段以段为单位进行内存分配;连续内存以整个作业为整体进行内存分配

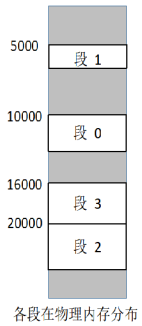
- 段

用户作业中**具有逻辑意义**的一部分,段内代码和数据形成线性存储空间
每个段有一个段号
虚拟地址表示为[段号:变量名] 或者 [段号:段内偏移量]
虚拟地址是二维的(段号相当于行号,段内偏移量相当于列号)

- 段表(每个进程一个)

段号	段基址	段长
0	10000	3000
1	5000	1600
2	20000	5000
3	16000	4000

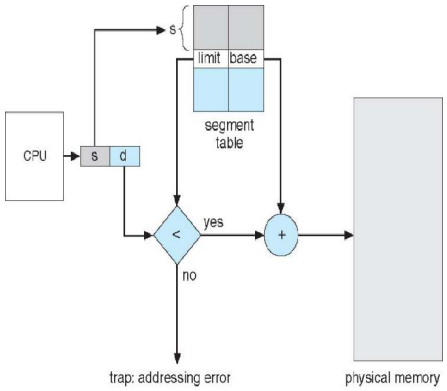
段表



各段在物理内存分布

段表保存在**内存中** (为了提高访问速度),记录每段在内存中的实际位置.
包含**两个字段**,段基址和段长
硬件支持:段基址寄存器\段长寄存器

- 地址变换



MMU实现
使用两个寄存器
(段表基址寄存器):**段表**的起始地址
(段表长度寄存器):**段表**的长度
(注意和段基址寄存器和段长寄存器区分开)
地址变换过程中**OS的作用**:填写段表,在切换进程时负责相关寄存器内容的保存和加载

- 过程

(1)比较段号s和段表长度寄存器，判断段号s是否超出了段长，若是，则产生异常
(2)通过段表基地址寄存器，用段号s查段表得到段基址和段长

(3)判断指令中给出的段内偏移量d是否小于段长，若不小于，则引发异常
(4)将段基址和偏移量相加，得到物理地址

- 分段的特点

程序员可见,由程序员划分的段
段具有逻辑意义
作业在内存中不连续存放
段的大小不等
段内连续分配,仍存在碎片问题

- 分页

物理地址和虚拟地址按统一尺度分割
一般而言,页的大小都是2的整数次方字节
虚拟地址空间中的每一段称之为页
物理地址空间中每一个划分称之为页框/帧(frame)
作业装入,首先被划分成页,然后每个页装入一个页框

- 页表(每个进程一个)

给出页和页框的对应关系,实现逻辑地址空间与物理地址空间的映射
存放在**内存**中
页表只有一个字段,即页框号字段

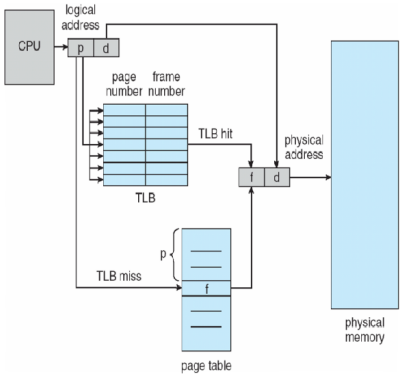
- 地址变换

页大小为2^n.虚拟地址假设为add.
可以得到页内地址为:add % 2^n 即后n位
页号:add / 2^n 即前len(add)-n位

- 代价

由页号查页表得到页框号
每一次地址变换都消耗一次访存时间

- 解决措施TLB(快表)

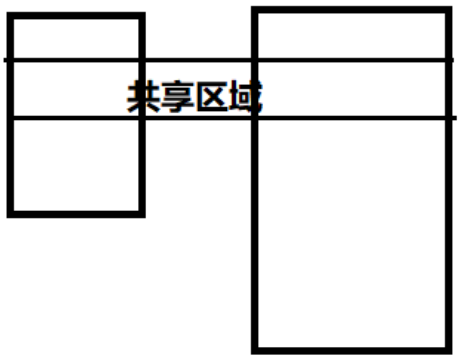


TLB(translation lookaside buffer)
其速度相当于寄存器一级的速度,存放于CPU中的一个专用存储区
只存放部分页表,按关键字查找
为了提高速度,采用并行查找技术(内部并行查找,与页表同时查找)
基于局部性原理
有两个字段(页号,页框号)
由**CPU负责填充**

- 页的大小

- 碎片:最后一页有1/2的空间被浪费,称之为页内碎片

        - 页越大,碎片大,但是页表就小,占用内存少.TLB覆盖范围增大,命中率提高
        - 一般页相对较小,不关心这点碎片
    - 页的共享
        - 同一程序建立的不同进程可以共享代码,即将代码部分映射到相同的页框中
        - 页不是代码的逻辑单元,但共享是针对逻辑单元的
        - 解决:逻辑单元对齐页边界,要共享的区域在两个进程中的各自相对位置必须相同



    - 页的保护
        - 页表保护位:在页表中添加字段,在地址变换中进行检查

            有效位(是否是空洞,即没有初始化过的内存)
            读/写/执行位
            存在位(是否缺页的标志)

        - 违规操作导致陷入
    - 多级页表
        - 常见的二级页表:页目录和页表
        - 段页式:先分段再分页
            - 先查段表(依据段号),从段表中可以得到页表位置
            - 根据页号查页表,得到物理地址
            - 将页内偏移量和物理地址结合,得到结果
- 分页与分段的比较
    - 不同点

        页系统划分,对程序员透明;段程序员划分
        页没有逻辑意义,一个子程序或者数据单元可能被分到两页中;段具有逻辑意义
        页的大小固定,为2^n;段大小不固定
        页宏观无碎片(最后一页可能有碎片);段有碎片
        分页模式下,程序员访问存储单元使用一维地址;分段是二维地址

    - 共同点

        作业在内存中不连续存访
        页的装入和地址变换和段非常类似
- 请求调页

- 虚拟存储器

  进程执行时不必要完全装入内存，由系统实现对换功能
  将内存抽象成一个巨大的、统一的存储数组
  实现了用户所看到的逻辑内存与物理内存的分离
  程序员无需考虑物理内存空间的限制，只需关注问题求解
  容易实现文件与内存的共享
  实现困难，可能会降低系统的性能
- 两种作业的内存分配方式
  - 连续内存分配:全部装入.
  - 非连续内存分配:访问哪一部分就把哪一部分装入内存.没有访问到的就先不装入
- 请求调页

  执行一个进程前，仅将PCB装入内存
  程序和数据并没有实际装入内存
  程序执行过程中，如果访问到的页还没有装入内存，就将其装入内存，称为缺页异常处理
  继续执行
- 页故障异常处理

  ①缺页异常，由MMU检测存在位引发(存在位表明该页是否被真正存放在内存里)
  ②缺页异常处理，进入OS
  ③写回换出页，检测脏位(被换出的页是否发生过更改了,如果更改,需要写回,否则丢弃(覆盖))
  ④读入换入页
  ⑤修改页表：页框号、存在位(被分配的具体物理地址)
  ⑥重新执行，PC不变
- 与纯分页方式的比较（先说硬件再说软件）
  - 纯分页:一次性装入作业的所有页
  - 请求调页

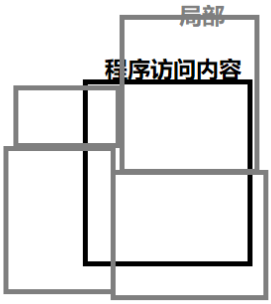
    ISA架构的改变：页故障异常、页表中增加存在位和脏位等设计
    操作系统的改变：页故障异常处理程序(页置换)
    硬盘的改变：对换区
    页表改变。
- 内存映射文件
- 请求调页的效率分析

- 缺页率
  - 局部性原理
    - 时间局部性:当前被访问得信息项,那么近期内它很有可能被再次访问
    - 空间局部性:如果访问了某存储单元,那么该存储单元附近的其他存储单元也可能被访问
  - 局部模式
    - 概念

      局部:是指进程访问的页的集合,是一个抽象的概念,一段程序或者数据由若干局部组成
      局部模式:一种运行方式,指程序执行时,其访问的内存空间从一个局部跃迁到另一个局部.

这种迁移是通过页置换完成.
抖动:CPU利用率低,缺页率高.(通常由于为进程分配的内存较小,导致进程每次访问内存时,访问内容没有命中,不断进行页置换,**时间开销大,系统繁忙于页的换入/换出,CPU利用率低**). 此时如果引入更多进程,缺页率进一步提高.

- 进程的局部与其缺页率的关系

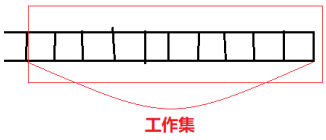


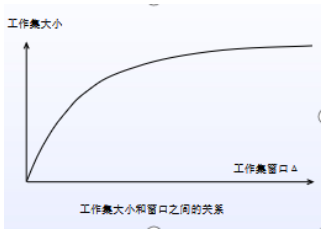
进程局部与缺页率的关系:
在上图的情况下,进程同时访问多个局部,形成了一个大局部,局部性差如果进程访问内容始终在一个局部中,则其局部性好

- 降低缺页率的方法

充分利用程序的局部性
为进程分配足够的内存,即足够的页框数量
采用适合的置换算法

- 工作集(通过过去使用的页集合去预测将来要使用的页集合,某种程度上可以看作是OS局部的一种表现方法)
  - 相关概念





工作集(WorkingSet):(按进程)
进程在某个时刻t之前进行Δ次 访问,这其中访问的页的集合.
记为WS(t,Δ). Δ成为工作集的窗口
窗口越大,工作集越大,但其增长速率是一定的,增长到某个值之后就趋于稳定

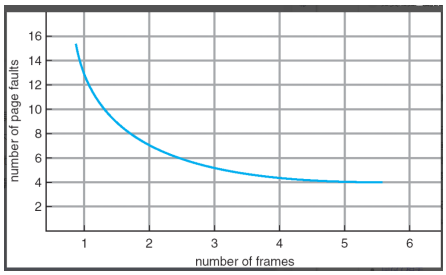
  - 相关性质

如果将WS中的页都装入内存,那么其缺页率会达到最小值
(因为过去使用的页如果都装入内存,那么依据局部性原理,接下来要使用的页大概率在加载进的WS中)
当Δ增大,|WS|也会增大,但其增加速率必然小于Δ.
(这是因为Δ越大,其包含的集合必然会越大,那么从过去向后看,过去使用到的页已经被包含在了未来使用到的页集中了,那么从后向前遍历时,前面的大概率已经被包含在页集中了)

当|WS|/Δ足够小时,可以认为WS是进程当前的局部

当Δ增大到某个值后,工作集变得相对稳定.
(相对稳定是指,对于t不敏感,这样当前的工作集可以作为将来近期的工作集;
同时对于Δ也不敏感,也就说明此时工作集已经包含了这个进程的一个局部了)

- 分配页框
  - 页框分配与缺页率的关系



  - 分配页框考虑的因素

需要**降低缺页率\防止抖动**出现\当前进程的数量\每个进程各自的工作集大小
系统中当前可用的内存数量\固定分配还是可变分配\所有进程中需要的最少页框数

  - 进程需要的最少页框数

最简单的一条一字节指令,对应需要一个页框(因为对应于一个访存操作)
如果一条指令两字节长,那么即便理论上只访问一次内存,但如果由于两个字节被分在两个页框中,也对应需要分配两个页框,**两次**访存
如果一条指令间接寻址:并且最终的指令要访问的两个字节大小的数据被分在两个内存页框中,可能会需要**4**次访存

  - 页框分配方法
    - 固定分配方法(进行置换时,只能换出自己的那一部分页,即局部置换)

依据的是一个进程固定的\整体的静态特征.
例如:
1 平均分配:大家都相等,方法简单
2 依据进程大小分配(比例):(看似合理,但实际上没有太大意义,因为大进程也是要依据局部性原理,使用到的不会特别多;小进程有可能访问的更加跳跃)
3 基于优先级分配

    - 可变分配方法 (全局置换)

根据进程的实际情况,决定分配的内存数量
当要进行置换时,要在所有进程中选择要置换的页
缺页率高的进程占用缺页率低的进程的页框

    - 置换策略与抖动的关系

可能会消除抖动
也可能会传播抖动(抖动蔓延),例如被换出的页所在进程也发生抖动

  - Windows XP示例
    - 相关定义

页簇(将某个页以及这个页所在的后续若干页调入).也即**预调入方法**,源于局部性
预调入:对于页簇中的其他没有用到的页,为什么不使用到的时候再进行调入,而要一次性将其调入呢?
因为对于调页来讲,一次读取磁盘的时间,**最大花费**在于移动磁头所花费的时间.也就

是我们认为,一次读磁盘花费的时间为(t(移动磁头等开销)+M(传输))
如果使用一次读一个页簇(假定是4个页),时间花费为:t+4M
如果一次读一页,读4页,时间花费为:4(t+M)
与预调入方法相关的还有延迟写.即将要写入的数据攒够一定的大小再统一写入磁盘中.

- 空闲页框的设置与一个阈值有关系,保证当空闲页框少于这个阈值,会自动调整工作集,从页框数量多于最小工作集的进程中获得.

- 置换算法
  - 置换算法概述(策略)
    - 置换:选择要淘汰的页,调入新的页
    - 目标:降低缺页率;在保证内存足够的**前提**下,优先保留进程工作集中的页
    - 算法抽象模型

      系统分配给进程页框数量为n
      进程访问内存行为通过页号序列表示
      连续多次的访问同一页,页号不重复
      系统可记录已经访问的页,但不知道进程将要访问哪些
      进程执行前,所有页框为空
  - 最优置换算法(理想化模型,实际上完全做不到)
    - 算法思想:淘汰将来最久不用的页(最优算法)

      也就是,缺页时,将-从现在开始到后面第一次被用到时距离当前最远的那个页-置换出去
    - #作业 [证明]该算法为啥最优,为啥没有Belady异常
    - 关于算法:无法实现,其意义在于可以用来衡量评价其他算法的好坏.(以该算法的结果为参照);一般来说,页框数越多,缺页率越低
  - 先进先出置换算法(FIFO)
    - 算法思想:置换出在内存中驻留时间最长的页
    - Belady异常

      页框数量越多,缺页率反而越高.
      这时个别现象,一般情况下,还是页框数量越多,缺页率越低
  - 最久未用置换算法(LRU:Least Recently Used 不-最近-使用)
    - 算法思想:依据局部性原理,最近使用到的,将来还会使用到.而最久没有用到的,将来可能不用.
    - 与最优置换算法是对称的,同样没有Belady异常
    - 算法缺点

      在**TLB**中记录最后一次使用的时间:开销
      替换时,在TLB中寻找最小的时间:开销
    - 算法改进

      在TLB中设一个引用位,每隔一段时间,将TLB引用位全部清零
      如果对TLB中某一块访问,将其置1
      替换时,替换引用位为0的
  - CLOCK置换算法

        - 算法思想:二次机会法.将进程在内存中的页组织成一个环形链表.
            - 链表中的每一个单元都有两个标志位.
            - 引用位R:近期是否被使用过,如果被使用,标1.如果没被使用,保持为0
            - 修改位M:当前页是否被修改过,修改过标1,未被修改标0
            - 是基于引用位的硬件设计
        - 需要进行替换操作时,需要先从里面优先寻找R=0&&M=0的.
        - 如果当前环形链表指针指向的单元,R=1,则将其置1,循环寻找下一个单元.除非所有单元R都为1,否则当前的单元不会被置换出去.(二次机会法的名称由来)
        - 评价:该算法与LRU方法近似.最早应用在Multics操作系统
- 拓展小知识
    - 概念
        - 透明:(分段)起作用但看不到
        - 磁盘属于计算机外存.
            - 一般磁盘分为两种:硬盘和软盘

              硬盘空间大,速度快,但不容易移动
              软盘空间小,速度慢,但容易移动
            - 硬盘又分为机械硬盘和固态硬盘
    - OS发展相关
        - 世界上第一个操作系统1956年,GM-NAA特征采用批处理系统,监控程序.系统吞吐量增加10倍.
        - 第一个多任务系统 1961年,LEO III计算机系统,实现了并发:多任务,多道程序,多进程,多线程
        - 第一个分时系统CTSS,1959年提出概念.1961年创造,创始人麦卡锡.
        - 分时系统的发展:CTSS->MULTICS->UNIX->Windows->Linux->Android
        - 1959年出现分页存储管理的思想;1962年在曼彻斯特大学研发的Atlas计算机上实现
    - 时钟中断
        - 时钟硬件:包括RTC(实时时钟)和OS时钟.

          RTC实时时钟是PC主板上的一块芯片.独立于操作系统,电池供电,也被称为硬件时钟,最底层.
          OS时钟只在开机时有效,完全由操作系统控制,所以也被称之为软时钟或系统时钟.
        - 时钟原型机制

          RTC是OS时钟的时间基准,操作系统通过RTC来初始化OS时钟,此后两者保持同步运行
          初始化完成后将完全由操作系统控制,和RTC脱离关系,因为OS时钟完全是一个软件问题.
          Linux中,计算机的时间是以节拍为单位的,每一次时钟节拍,系统时间就加一
        - 时钟中断

          本质上来说,时钟中断只是一个周期性的信号,完全是硬件行为,该信号触发CPU去执行一个中断服务程序.
    - 上课提到过的函数
        - fork()

创建一个子进程

- yield()

  进程主动放弃当前的CPU使用权,同时调用CPU调度器,将CPU控制权分配给某个进程

- 上课提到过的题
  - 要说明一个 存储结构(连续存储或者啥)的特点,需要从那几个方面进行考察?
    - 适合的设备
    - 存取文件的速度(快/慢)
    - 读写方式(顺序/随机)
    - 文件长度(可变/不可变)
    - 空间利用率(碎片问题)
    - 应用

- 实验相关内容:
  - GCC相关
  - MakeFile相关

以上内容整理于 幕布文档