

OS 0

2020年2月12日 14:34

- 教材

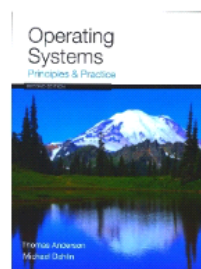
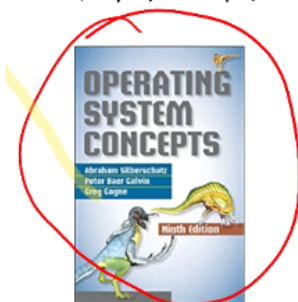
- 唯一指定教材：龙书 OSC
 - osc 上的所有内容都是考试范围
- OSPP 有的project和这本书相关 但是不做要求
- 第三本书是自己写的 部分lab相关 不买也没关系

- ◆ **Operating System Concepts, 9th Edition, Al**
Silberschatz et. al.

- ◆ **Operating Systems Principles & Practice, 1st**
Edition, Thomas Anderson et. al.

- ◆ 操作系统课程设计, 机械工业出版社, 朱敏, 唐博等

-



- 上课

- 大课+lab
- 每节课后读linux kernel (从0.11开始读会比较合适)
- 尝试给kernel做贡献

-

No.	Topic
1	Introduction
2	OS Structures
3	Processes
4	Threads
5	Synchronization
6	Scheduling
7	Main Memory

No.	Topic
8	Virtual Memory
9	Storage Structure
10	I/O
11	File System II
12	File System I
13	Linux System
14	Advanced topics

- 分数

- 作业 50
 - 出勤20 lab20 proj10
 - lab每周一次 要交报告 一共10次 每次lab满分100
- 考试 50
 - 期中20 期末30

- Project

- 选择1: 读OS论文 给一个可以动手的hands on tutorial
 - 可以组队 不超过两个人
 - 讲30min 每个lab时间段内不重复 后面还会再给列表
 - <https://github.com/ISG-ICS/cloudberry/wiki/quick-start>
 - DDL 应该会在后面几个星期

Option 1: I will give a list of system papers, you read some of it and give a **hands-on** tutorial to the students in the same lab session.

- ◆ **Hadoop:** Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified data processing on large clusters." OSDI, 2004
- ◆ **Hive:** Thusoo, Ashish, et al. "Hive: a warehousing solution over a map-reduce framework." VLDB, 2009
- ◆ **Spark:** Zaharia, Matei, et al. "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing." NSDI, 2012
- ◆ **Flink:** Alexandrov, Alexander, et al. "The stratosphere platform for big data analytics." VLDBJ, 2014
- ◆ **Heron:** Kulkarni, Sanjeev, et al. "Twitter heron: Stream processing at scale." SIGMOD, 2015.
- ◆ **Tensorflow:** Abadi, Martin, et al. "Tensorflow: A system for large-scale machine learning." OSDI. 2016.

Tutorial example: <https://github.com/ISG-ICS/cloudberry/wiki/quick-start>

○ 选择2: 根据论文实现FS

- 4~5人 需要邮件申请 评估是否能做
- 可以参考已有的开源实现
- Bonus: 直接拿A/A+ 每周指导

Option 2: Build a File System / Storage System by a published research paper.

- ◆ (1) **PloarFS:** An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database, PVLDB'2018, Alibaba File System for PolarDB
- ◆ (2) **PaxosStore:** High-availability Storage Made Practical in WeChat, PVLDB'2017, Tencent Storage for Wechat.
- ◆ **Team size:** 4-5 per group, assigned by email application.
- ◆ **Bonus:**
 - ◆ Got A or A+ directly (published as an open-source version, e.g., HDFS for GFS), or bonus marks (up to 12%)
 - ◆ One-to-one guidance (at least one afternoon discussion per week)

○ 课程大纲

- 网站: BB

○ Warning

- 会很难
- 没有cheat sheet
- 需要过test case才有分
- 选了之后不要选太多其他课
- 考试会考所有的内容 课上/lab/龙书

OS 1: OS 介绍和引入

2020年2月19日 14:01

- 引入：什么是电脑？
 - 硬件之间的物理连接：总线 bus
 - Memory bus (+DMA), PCI bus, extension bus...
 - 互操作性：driver
 - 如果要print Hello World，执行中发生了什么
 - CPU执行指令 把字符串放到显示控制器对应的内存区域
 - 字符串被通过总线发送到显示控制器
 - 显示控制器更改物理设备的显示
 - 过去：一切都需要手动完成
 - 现在：可以用high level language完成
 - 高级语言编写，编译，执行
 - OS：可执行文件如何被执行（从executable -> computer）
- 引入：学习OS的不同层级
 - Lv1 用OS：使用现有OS的API
 - Lv2 玩OS：根据自己的需求修改OS 扩展功能（有新的硬件了 怎么写driver来使用）
 - Lv3 设计OS
- 引入：CMU/Stanford的OS课
 - Stanford: 先讲concept 然后4个项目 基于Pintos
 - 1 extend threads
 - 2 program with limited syscall
 - 3 implement file system
 - 4 virtual mem memory
 - CMU: 先讲concept 然后给硬件(板子)
 - 然后实现OS
 - 然后编写app运行在OS上
- 什么是OS
 - 给app提供硬件资源访问的特殊的软件中间层
 - 复杂硬件的简单抽象
 - 对共享资源的保护性访问
 - 安全、认证、鉴权
 - 不同实体之间的通信
- OS的组成
 - 核心：kernel：管理硬件设备 提供syscall能力
 - 支持
 - Driver：内核和外部设备的交互
 - shell：简单的用户界面 和内核交互
 - 其他可选程序

- OS的作用
 - 为应用程序提供抽象
 - 管理硬件资源
 - 实现：通过algorithms和techniques
- OS的基础（作用？）
 - Virtual Machine Boundary
 - OS 为底层硬件提供了统一的高层级抽象
 - 把实际的硬件抽象成了虚拟的硬件
 - 程序和进程
 - PCB process control block
 - PID process ID
 - 上下文切换 context switch
 - 在不同的进程间切换
 - 调度 scheduling, protection
 - 制造许多程序同时运行的假象
 - 避免非法访问和数据泄露
 - IO
 - 通过controller bus来控制所有的IO设备
 - 提供抽象供应用程序使用
 - 加载 loading
 - 系统启动的时候 需要把OS从storage装载到memory (booting)
 - 系统运行过程中 也需要加载其他的应用程序
- 进程
 - 运行中的程序示例 execution instance of program
 - 相同的代码 可以由多个进程运行
 - 进程 process 和程序 program的不同
 - 进程有状态：运行到的代码位置 剩余的CPU时间
 - linux中可以用ps/top等查看进程的相关信息
- Shell
 - 一个特殊的程序
 - 运行terminal的时候 实际是在与shell交互
 - 对输入的命令进行语法检查
 - 通过fork来执行程序 and 命令 生成child process
- Process hierarchy
 - 由父-子进程组成的进程树（查看：pstree）
- System Call
 - 由kernel执行的functional call
 - 和底层的执行细节联系
 - 在内核态执行 执行完之后返回用户态
 - 分类：进程、文件系统、内存、安全、设备
 - Man syscall
 - 对比 system call 和 library function call: fopen和open

- fopen为原始的open提供了更方便程序员使用的接口
 - Library call 隐藏了 syscall的复杂性
 - Library function call 被编译和打包在 library file 中: DLL / SO
 - 用户程序 fopen -> library 调用 open -> 内核处理
- 进程
 - syscall
 - 生命周期、调度
 - 信号
 - 同步
- 进程的内存
 - 运行: ./a.out的时候发生了什么
 - 为什么需要 ./: 当前目录不在 \$PATH
 - . 当前目录 .. 上级目录
 - Loading: 从可执行文件到进程的内存空间
 - Segmentation / 内存布局
 - 本地变量 - 栈
 - 动态分配内存 - 堆
 - 全局变量 - data segment
 - 常量 - 常量
 - 代码 - text segment
 - Execute: 调度器给进程分配时间片 将进程置于执行状态
- 内存
 - 虚拟内存
 - 内存相关的 function
 - Stack Overflow
 - malloc的作用
- 文件系统
 - 在存储设备上「组织」文件的方式
 - 数据、元数据、索引
 - FS vs OS
 - 一个磁盘上的多个分区可能有不同的FS
 - 一个OS可以使用多个FS
 - 文件夹: 实际上也是个文件 (linux everything-is-a-file)
 - 著名 FS 的实现
 - 如何删除文件
- OS 的其他部分: programmer -> sys programmer -> program OS
 - 多线程、多进程
 - 引导
 - 锁
 - IO
 - 虚拟化

OS 2: OS 的核心概念

2020年2月26日 15:05

- 操作系统的 component
 - 用户
 - 系统程序和应用程序
 - OS
 - 硬件
- 计算机组成的视角
 - 硬件连接：所有设备和CPU MEM连在总线上 通过共享内存进行交互
 - CPU和外设之间需要竞争内存周期
 - 存储 hierarchy：容量、延迟、带宽、价格
 - Cache: 此处特指 reg 和 mem 之间
 - 一般有 2/3 layer 的 cache
 - L1每个core有自己的 L2和更高层的可能在core之间共享
 - 现代计算机（冯诺依曼结构）：CPU 内存 设备
- 内核中的数据结构
 - 链表：单向、双向、circular
 - 二叉搜索树：log n 的搜索操作
 - Hash map
 - Bitmap: 用n个bit表示n个元素的状态
- OS 简史
 - 变迁：设备越来越便宜 人越来越贵
- 4 个 OS 核心概念
 - 线程：描述了一个程序的运行状态 自己的上下文、寄存器、处理器状态
 - 地址空间：程序执行所在的虚拟地址空间 和物理内存分离
 - 翻译 translate：从虚拟地址空间映射到物理内存
 - 进程：一个地址空间+ 1+个线程
 - 一个进程只能有一个地址空间
 - 双态操作 / 保护
 - 系统保护部分资源 只有OS才能访问 不能被 user program 访问
 - 不同进程之间相互隔离 通过控制 translation
- 最基础的 OS 功能
 - 加载程序、创建内存布局、控制程序执行、提供服务、保护os和其他程序

OS 2

2020年3月4日 14:03

- OS的最基础功能
 - 加载并运行程序
 - 提供服务
 - 保护自己和其他程序
- OS Concept 1: 控制线程
 - reg存储了当前线程的上下文 context: sp, fp, hp...
 - 线程: 独立的执行上下文单元 single unique execution context: PC, reg, execution flag, stack
 - 线程在执行: 上下文被加载到寄存器中 (活在reg中)
 - Program Counter 保存了当前正在被执行的指令的地址
 - reg中只有thread状态最重要的一部分 (root) 其他的状态信息都在内存中
- OS Concept 2: 程序的地址空间
 - 地址空间: 可以被寻址的地址+相关联的状态
 - 地址空间是虚拟的 实际读写的时候可能遇到各种情况: 无操作 (?)、正常 (映射到物理地址上之后读写)、被忽略、IO操作 (memory-mapped IO)、异常 (读写了其他程序的内存)
 - Code segment: 程序的instructions 会被用于执行
 - Static data segments: global constants 在程序运行时不会被修改
 - stack和heap需要动态增长 因此增长方向相对来节省空间 (实现对内存的更灵活应用)
 - 为什么stack往下长 heap往上长
 - stack中存储local variable和frame 由compiler管理 并不需要太大
 - heap用来动态分配 由用户管理 可能会需要很大的内存
 - 对程序员和用户而言 分配的内存地址从小到大符合人类的思维方式 (数组中index大的 addr也大)
 - ?
- OS Concept 3: 进程
 - 进程: 有限权限的执行环境
 - Process = threads + address space
 - 进程有内存空间、文件描述符、文件系统上下文...
 - 这些资源都是被同一个进程内的多个线程共享的
 - Code/data/files在Process中 register/stack每个thread自己处理
 - 为什么要有进程?
 - 最主要: 提供保护 (地址空间隔离), 保护其他进程和OS
 - ?
 - 线程间通信在同一个地址空间内 更简单, 进程间通信跨地址空间 更困难

- 一个应用程序也可以有多个进程
- 进程处理concurrency 地址空间处理protection
- OS Concept 4: 双模式执行
 - 只有OS才能访问特定资源 不能给用户程序直接访问
 - OS也需要控制从虚拟地址到物理地址的转换
 - 硬件提供了两种模式
 - Kernel/supervised/protected/privileged
 - User
 - 硬件实现
 - 有一个bit来标记现在的state
 - 部分操作只能在kernel mode下执行
 - 需要从user mode fail/trap
 - User->kernel 保存user的相关状态 执行操作
 - Kernel->User 恢复user的相关状态
 - 切换模式的3种方式
 - Syscall
 - Interrupt
 - Trap / exception
- 其他问题
 - 为什么ALU里的指令数比decode的指令的数量少?
 - 因为prefetch的存在
 - 如果存在条件执行 decode的指令 未必会被ALU执行
 - 为什么堆栈的增长方向不能互换?
 - <https://gist.github.com/cpq/8598782>
 - 内存读写为什么可能会Nothing?
 - 为什么有了threads还需要process?
 - <https://www.backblaze.com/blog/whats-the-diff-programs-processes-and-threads/>
 - 同一个process的多个threads可以在不同的CPU core上运行吗?
 - <https://www.zhihu.com/question/31683094>
 - 不同的CPU core之间 什么是隔离的 什么是共享的?
 - <https://www.guru99.com/cpu-core-multicore-thread.html>
 - Base & Bound? P38?
 - 超线程? hyperthreading到底在做什么?

OS 3 进程

2020年3月4日 15:15

- 进程：执行中的程序
 - 包含所有相关信息：当前PC,运行时间,文件句柄,page table..
 - 存储在 Process Control Block中
- 进程识别：Process Identifier
 - 可以通过getpid()的syscall获取当前进程的PID
 - C中getpid是syscall还是库函数？
 - 区分：看man page (2是syscall 3是libcall)
 - PID递增 到达最大值之后重新开始 跳过已经分配的部分
- 进程创建：fork
 - fork调用后 子进程从fork return的地方开始执行 不是整个程序的开头
 - fork创建的子进程和父进程执行的是同一个程序
 - fork的实现：通过克隆父进程来创建子进程
 - fork之后 父子进程相同的：PC, 代码, 内存(地址空间), 打开的文件
 - Fork之后 父子进程不同的：fork()返回值(新PID/0), PID(未必+1), 进程的父进程, 运行时间, file locks
 - 系统的首个进程：init
- 执行其他程序：exec
 - exec并不是简单的执行 而是把execution给了目标程序 更改了正在执行的code 而且不会回到原来的code
 - 实质：替换user-space的信息：代码、内存、寄存器值
 - 但是PID、父子进程关系依然是保留的
 - 真正的执行其他程序而不影响源程序：fork + exec
 - shell和c中libcall system()的思想
 - 但是直接fork+exec 源进程和新进程的执行顺序不确定 由scheduler决定
 - 需要用wait来挂起父进程 并在子进程结束的时候重新唤醒父进程
 - Fork + exec + wait = system
 - fork创建进程 exec把program载入内存
- 挂起进程：wait
 - User-space 进程的lifecycle：running, waiting, terminated
 - System-space 还有更多的状态
 - Syscall wait：挂起suspend 当前进程
 - 如果有子进程且某个子进程terminated / 收到signal, 唤醒当前进程
 - 如果没有子进程, 直接return
 - waitpid可以指定等待某个特定的子进程 也可以检测其他状态变化

OS 4 进程2

2020年3月11日 14:01

- 系统调用的不同视角
 - 内存: pc从user->kernel->user区域
 - CPU: trap mode bit 0->1->0
 - 时间: user/sys
- 关于时间的讨论
 - 为什么 $real > user + sys$
 - User/sys 之间的切换也有 overhead, 但是不是主要
 - 如果任务是 IO-intensive 那么就会这样
 - 写到 io 设备很慢
 - Sys 时间只算 cpu 时间, 不计算具体写入 io 的时间
 - Real 计算的是程序开始到结束的 wall-clock time
 - 如果调度了其他程序 也会计算时间
 - 包括了被其他进程占用的时间片
 - 为什么 $real < user + sys$
 - 多核、多线程任务
 - real是物理时间, user/sys是CPU时间
 - 核多, CPU时间成倍增加
 - 根据统计, 这是现代多核计算机的常见状况
 - Overhead
 - Function call 需要压栈出栈
 - Sys call需要模式切换 调用其它进程 (内核)
 - Context switch
- fork到底做了什么
 - 内核中 用一个双向链表作为task list 来存储process
 - Kernel space更新: fork首先把一个进程的全部信息 (PCB) 完全复制,然后更新新进程的PID, running time, parent pointer和原进程的list of child
 - User space更新: 然后把原进程User space内的内存完全复制给新进程
 - 更改返回值: 修改原进程和新进程的fork返回值
 - 注意: opened files在kernel space 也是被复制了的 而且没有修改
 - Array of opened files
 - 0/1/2: stdin, stout, stderr
 - 3+: 其他用fopen
 - 也是父子进程共享terminal output stream的原因
 - TODO: 如果父子进程写入同一个文件 会发生什么
 - 对应到同一个 kernel中的 file description
 - 如果调度适当, 不会互相影响

- exec到底做了什么
 - 清空 User space: 清空local var, dynamic-allocated memory
 - 用新程序 reset user space: 加载新程序的 global var, code + constants, 重设 PC
- wait到底做了什么
 - exit的过程
 - 子进程退出: Kernel释放kernel space内存, 所有opened file关闭
 - 随后free user space memory, 但是 PID 依然保存在 process table 中
 - 需要允许父进程读取 exit status
 - 子进程现在进入了 zombie / terminated: 只有 pid 了, 其他都被释放了
 - Kernel 通知父进程: 发送 SIGCHLD 给父进程
 - Wait的实现
 - 进程默认忽略 sigchld 信号
 - 如果父进程wait()了, 那就会注册一个signal handling routine 在父进程的PCB中
 - 在sigchld发送后, 这个routine被调用
 - 收到信号后, 首先接收信号并移除信号, 然后真正把child process从kernel中移除
 - exit后只会zombie
 - 父进程wait完之后才会完全消失
 - 随后内核解除这个handler的注册, 把子进程的PID作为wait的返回值
 - 父进程此后又会忽略sigchld
 - 异常状态
 - 子进程在父进程wait之前就exit了
 - ◆ sigchld已经发了
 - ◆ 父进程的handler会被立刻触发
 - ◆ 立刻销毁子进程 (但是在此之前子进程都会一直zombie)
 - 父进程不wait: zombie不消失 浪费系统资源
 - 手动模拟了一个sigchld? ?
 - zombie进程会被标记为 defunct
 - wait的重要性
 - 对系统资源管理很重要
 - ◆ 有限的PID
 - ◆ 不wait会耗尽pid: No process left
- 第一个进程
 - 所有进程都fork而来? F
 - 第一个进程init不是
 - 有其他POSIX API来创建进程
 - Vfork: 如果fork完立刻exec 那就不用复制内存了
 - Posix_spawn
 - init的任务: 创建其他进程

- 可视化进程树：pstree
- 如果父进程先死了
 - Linux：过继给init，init作为父进程（也可以选择其他进程作为父亲）
 - windows：就这样把 维护forest
- reparenting的实现
 - 父进程exit调用中，把所有子进程的parent pointer改成init, init的list of children 把子进程都加上
- Background job：reparenting的应用
 - 本来shell是父进程
 - 手动把进程过继给了其他进程，父进程就可以安全退出了
- Process lifecycle
 - Ready
 - 刚创建
 - 刚运行完时间片
 - 从Blocked返回
 - Running
 - Zombie/terminated
 - 自己死
 - 被迫死
 - Blocked/waiting ((un)interruptible)
 - Ctrl + C 能不能救
 - 注意：无法直接从 ready -> terminated
 - 需要先变成 running

OS Lab 2: Shell 基础

2020年2月26日 16:29

- 单双引号：双引号有转义 单引号无转义？强弱引用？
 - 单引号 raw string
- awk分列：\$IFS

OS 5: Job Scheduling 作业调度

2020年3月18日 14:02

- Schedule 调度
 - 决定接下来运行哪个进程
 - 为什么需要调度：计算资源（CPU）是有限的
 - 进程分类：CPU/IO Bound
 - CPU Bound 计算密集：user time > sys time
 - IO Bound IO 密集：sys time > user time
 - 进程调度的分类
 - 非抢占式：outdated
 - 抢占式 preemptive / time-sharing
 - 进程一旦拿到 CPU 就会一直占有 直到：自愿等待IO/自愿退出/其他中断发生
 - 好处：保证响应性
 - 坏处：不适用于对完成任务的时间有严格要求的系统
 - 进程调度算法
 - 问题描述
 - 输入：有一系列task 每个task有arrival time和cpu requirement
 - 输出：这些task运行的顺序
 - 两个大类
 - offline：提前知道 只有理论用途
 - online：更实际
 - 不同的优化目标 / 评价标准 metric
 - 上下文切换的次数
 - ◆ 刚开始运行是不算切换的
 - Turnaround time：从任务到达任务结束
 - ◆ 到达后等待+运行中等待+自身cpu req
 - Waiting time：任务的总等待时间
 - ◆ 到达中等待+运行中等待
 - ◆ 也可以用turnaround - cpu req
 - 常见算法
 - FIFO
 - Shortest Job First SJF 最短优先
 - ◆ 非抢占 SJF：不能强行中断 每次取当前队列中需要cpu req最短的执行
 - ◇ 如果平局 FIFO 先到的先处理
 - ◇ 问题：如果有一个cpu req很大的 后面的req都比它小 那就永远调度不到了（饥饿）

- ◇ 就算是抢占式的也会有这个问题
 - ◆ 抢占 SJF: 新task到来时 当前task被暂停 选取有最小*剩余cpu req*的任务继续
 - ◇ 相比非抢占式 平均waiting time和turnaround time都有所下降
 - ◇ 但是代价是更多的上下文切换次数
 - Round-robin RR 轮换
 - ◆ 每个进程都有一个quantum 可用CPU时间总量
 - ◆ 抢占式: 如果当前进程耗尽/退出 就切换到下一个非0剩余时间的进程
 - ◆ 所有进程都耗尽了 那就都重新把剩余时间设定为初始值
 - ◆ 没有优先级的时候就像循环队列
 - ◆ 新进程加到尾部 不会触发调度器立刻调度
 - ◇ 演示中是先加入再调度的
 - ◆ 和 SJF 比较
 - ◇ 似乎平均waiting/turnaround和总上下文切换次数都更大了
 - ◇ 但是响应性上升了 因为每个job都能分到时间
 - 带优先级的调度
 - 每个task都有优先级
 - 分类: static priority / dynamic priority: 一开始确定优先级之后是否会改变
 - 带优先级的RR: 用优先队列实现 每次新process会触发selection
 - 如果进程中间被暂停(抢占了) 会出队再入队
 - 可以直接recharge quantum 也可以之后再一起recharge???
- Context Switch 上下文切换
 - 从一个进程切换到另一个进程的过程
 - 什么情况下会需要暂停当前进程去做切换
 - 当前进程主动进入 Blocking/waiting
 - 信号
 - 外部中断 (键盘输入..)
 - 内部timer中断 -> ready
 - 内部更高优先级事件 -> ready
 - 作用
 - Multi-tasking
 - 充分使用 CPU
 - 实现
 - OS的kernel负责 (kernel中执行的切换过程都是额外的Overhead)
 - 保存当前进程状态到PCB, kernel中其他操作, 从新进程PCB重新加载状态
 - 1 调用 sleep: running -> interruptable wait
 - 2 状态备份到PCB: PC, 其他寄存器...

- 3 调度器选择下一个运行进程
 - 4 从新进程PCB加载状态（上下文）并运行
 - Expensive
 - 直接开销
 - kernel需要保存/恢复寄存器
 - kernel需要执行指令来切换地址空间
 - 间接开销
 - CPU Cache, Buffer Cache, TLB 里的缓存项会miss
 - ◆ 之前进程缓存的东西 新进程未必用得上
 - ◆ CPU cache是OS直接管理的
 - ◆ CPU Cache: CPU的 / Buffer Cache: 内存的
 - Syscall 也会触发上下文切换：从 user process 到 kernel process
 - 所以应该减少 syscall 次数
- 补充文章: <https://www.linuxblogs.cn/articles/18120200.html>
 - hyperthreading: <https://zhuanlan.zhihu.com/p/58448264>

Data/inst level parallelism

SIMD

进程/线程并行

Lab 5

2020年3月25日 14:03

Spin lock: 一直等

Mutex lock: 换下来再唤醒

Mutex try lock: 试一次 不行就算了

Spinlock_try? Mutex_lock_try? 差别?

信号量

Count=0 没有桌子 也没有人在等

P24 brother/dad是while而不是if

Conditional signal和unlock能否交换

作业临时记录

2020年3月29日 17:46

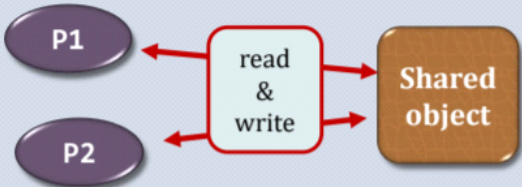
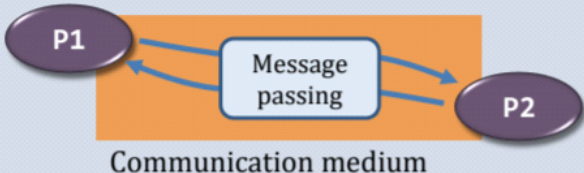
Scheduler.c line84 中文分号

L6 进程间同步

2020年4月1日 14:01

- 进程间通信 IPC
 - Shared object / Message passing

Summary of IPC models

Shared Objects	Message Passing
	
<ul style="list-style-type: none">• shared files (on disk; slow)• pipes (restricted, but OS takes care of synchronization for you)• shared memory (primitive, general, but synchronization is on you)• shared address space (threading)	<ul style="list-style-type: none">• socket programming• message passing interface (MPI) library for computing clusters.
<ul style="list-style-type: none">- Usually single-node communication- More efficient- Need to take great care of synchronization because of sharing the same object	<ul style="list-style-type: none">- Usually multi-node communication- Less efficient- Less troublesome in synchronization- But need to care of other faults (e.g., what if a network link is broken?)

- 问题: buffer size 有限 读写速率不一致? -> 同步
- 竞态条件 race condition
 - 执行结果和共享资源的访问顺序相关
 - 共享对象 + 多个进程 + 同时访问
 - 解决: 互斥 Mutual Exclusion
- 互斥 Mutual Exclusion
 - 共享资源不能被同时访问, 一次只能有一个访问
 - 临界区 CS: 一次只能被一个进程访问的代码段
 - 应该尽可能小, 避免造成性能太大损失
 - 一个CS内可以访问多个共享资源
 - 问题: 如何实现临界区的进入和退出?
 - 实现要求:
 - 互斥: 一次只有一个进程进入 CS
 - 不能多个进程同时进入
 - 其他进程请求后发现 CS 已经被占用, 会被 block, 直到自己可以进入

- 有界等待 Bounded Waiting
 - 请求后，一个进程总会等到自己进入 CS（饥饿）
 - 防止一个进程不断进出，其他进程进不去
- 进展 Progress：如果没有进程在 CS，那么总会有进程进入 CS
 - 不会所有进程都在瞎等（死锁）
 - 厕所没人，但是门被锁了
- 实现
 - 实现0：不实际：CS中禁止context switch
 - 单核虽然可以，但是不现实：如果CS内的进程挂了
 - 多核不正确：一次只锁了一个core，没有禁止所有core
 - 基于锁的实现：悲观锁 lock-based
 - 自旋锁 spin lock
 - ◆ 实现1：basic Spin lock
 - ◇ 用一个共享变量 turn 来标记当前谁可以进入/轮到谁
 - ◇ 如果没到自己，就 while 等
 - ▶ 没到自己 turn != 自己的id
 - ◇ 虽然浪费了一些 CPU 资源，但是在多core且等待时间短（等待时间<上下文切换时间）时还行
 - ◇ 问题：要求 Strict alternation
 - ▶ 要求进程之间必须交替进入 CS
 - ▶ 一个进程无法连续进入 CS，即使当前 CS 没人
 - 违反了 progress 原则
 - 如果一个快，一个慢，快的那个必须等慢的进入完 CS，自己才能进去
 - ◆ 实现2：spin smarter：Peterson's Solution
 - ◇ 多用一个共享对象 interested
 - ▶ 进入前需要设定自己的 interested 为 true 表明意图随后试图获取 turn
 - Turn 在这里的语义不同：标记的是最后一个希望进入 CS 的进程？？？
 - ▶ 如果某个进程没有 interest，那么另一个可以直接进入
 - ▶ 如果两个进程都有 interest，那么按照 turn 来
 - ▶ 退出时直接设定自己的 interest 为 false 就好了
 - ◇ 解决了 basic spin lock 违反 progress 的问题
 - ◇ 可以扩展到更多进程 Filter Algorithm？？？
 - ▶ 坑：考试可能考
 - ◇ 问题：priority inversion problem
 - ▶ 一个低优先级进程 L 进入了 CS
 - ▶ 高优先级进程 H 抢占获得了 CPU 时间，但是无法进入 CS

- ▶ 理论上在 RR+动态优先级下应该没问题？总能调度到 L？
 - Sleep lock
 - ◆ 实现3: sleep-based lock: 信号量 semaphore
 - ◇ 信号量=可用资源计数 s + wait list
 - ◇ 期末问题：哪些方法可以实现 mutual exclusion? 优缺点？
 - ◇ 麻烦之处：需要正确实现 CS 的进入/退出
 - ▶ 需要和内核交互
 - ▶ 需要在进入/退出时 uninterrupted
 - 比实现 0 整个 CS 都锁死已经好很多了
 - ◇ 信号量的操作
 - ▶ Sem_wait: 等待直到获取到资源
 - $s--$
 - 如果可用资源 $s < 0$ 则 sleep
 - ▶ Sem_post: 通知有新的 s 可用
 - $s++$
 - 如果有在等 s 的进程，唤醒一个
 - ◇ 用来实现互斥的时候 S 的初始值设定为 1
 - ◇ 信号量也可以用于更广泛的信号原语
 - ▶ 期末考试：一定会考至少一个 IPC 问题
 - 不需要锁的实现：乐观锁 lock-free
- IPC 问题
 - Producer-consumer 问题 / bounded-buffer problem
 - Bounded buffer + producer + consumer
 - 如果buffer满了 producer应该等待 直到consumer取出一个item后通知 buffer可用
 - 如果buffer空了 consumer应该等待 直到producer放入一个item后通知 buffer可用
 - 问题分解：本质上是两个问题
 - 互斥: buffer 是一个共享对象，一次只能有一个读写
 - ◆ 用一个 binary semaphore (mutex) 解决
 - 同步/协调: buffer 有界，不能溢出
 - ◆ 用两个 semaphore 解决: fill, avail
 - 进一步问题
 - 是否可以把两个 semaphore 合并成一个？
 - ◆ 不可以: 可能会导致从空 buffer 读取
 - 是否可以更换 mutex / avail 顺序
 - ◆ 不可以: consumer 先 mutex 再 avail, 会导致 mutex 一直被持有, producer 就再也拿不到 mutex 了, 导致双方 endless sleep (死锁)
 - Dining philosopher 问题

- 每个哲学家有想和吃两种状态
- 吃的时候需要同时获得左手和右手的筷子才能吃
- 要求：避免饥饿、避免死锁
- 和producer-consumer问题的区别：需要2个共享对象
- 问题分解
 - 互斥：吃的时候筷子不可被其他人用，一个筷子不能多人同时用
 - 同步：避免死锁
- 解法0：强制同步：先13吃，再24吃，不断循环 / 排队
 - 问题：不想吃的人的时间被浪费了
 - 会导致 progress 问题：快的要等慢的
- 解法1：一次等两只筷子，拿两只筷子
 - 所有人同时开始吃的时候会死锁：都尝试拿起左筷，然后开始等待右筷
- 解法2：先尝试拿一只，再尝试第二只，如果第二只失败就放弃第一只
 - 不会死锁了，但是可能饥饿：所有人同时拿起左筷，发现右筷占用，再放弃左筷，下一轮循环...
 - 可能解决：随机启动
- 正确解法0：资源分级 / 顺序修改
 - 某个人按照和其他人不同的顺序拿筷子
 - 问题：效率不高，不实用
- 正确解法1：服务生解法
 - 引入一个服务生来判断筷子的可用性
 - 需要经过服务生允许才能拿筷子
- 正确解法2：把信号量放在人而不是筷子上（Chandy/Misra解法）
 - 如果一个人在吃，那么左手和右手边的人就吃不了
 - Captain：判断自己能不能吃

期中试卷

2020年4月1日 16:12

P5

1. A b e ???
2. C d
3. C ?? A
4. A b c
5. A
6. A b ??
7. D ?? BC
8. A ??
9. A b c d ?? C
10. B ?
11. B ? A
12. B ??
13. ??????
14. C-无答案
15. B ?? A
16. B
17. D
18. A
19. C
20. A b c d ? D

1d: 实现上是, 理论上不是?

3: <https://www.geeksforgeeks.org/multi-threading-models-in-process-management/> (注意问的是不存在的)

6: 先排除C, 有点主观, 但是答案是A: 不应该单纯因为等IO降优先级? (只因为运算降低优先级, 不会因为IO降低优先级)

7: ???

8: paging在user space 但是handler在kernel space

9: 看会不会经过/有没有可能在user space 发生? ??? 只有context switch不会在user space

11: I 的确会wait到内容被读入memory, III read的参数没有filename, 只有fd, 但是open有filename

期中选择14修正: 临界区不是原子操作, 如果A在CS, B不能进入A所在的CS, 但是可以进入程序的其他区域, 这个时候依然是可以做scheduling的 (除非进入CS的时候关闭Interruption), C选项在现在的实现中是可以的, 因此本题没有答案

15: B 资源分配的最小单位是process, C user-level 线程切换是不知道的, D 也是错的 排除法选A

16: 进内核: 中断、异常、syscall (div0是异常)

18: 也可能选B 让IO多的先跑 早点跑早点去等着

20: wait可能会block, malloc失败会block, 请求IO也不一定ready, 只有D了

P1

-4?

time	HRRN	FIFO	RR	P-SJF	NP-SJF	PRI
1	A	A	A	A	A	A
2	A	A	B	B	A	B
3	A	A	A	A	A	A
4	A	A	C	C	A	C
5	B	B	A	C	B	C
6	C	C	C	A	C	D
7	C	C	D	A	C	D
8	D	D	A	D	D	D
9	D	D	D	D	D	A
10	D	D	D	D	D	A
AVG TA	4.5	4.5	4.5	4	4.5	4.25

HRRN 非抢占

RR: 新来的放在后面

SJF: 按照非抢占算

P2

- a. 增加x的操作分为: 读取, 加, 写入。如果A中完成了加但是没有写入, 此时调度到B, B完成了写入, 然后回到A又写入, 则部分写入被无效了, 因此总的值 ≤ 25

a.
$$\begin{cases} x = 2i + 3j \\ i + j \geq 5 \\ 0 \leq i \leq 5 \\ 0 \leq j \leq 5 \end{cases}$$

b. (来自答案：用算术分析)

- b. $x=2$ 要求 x 只被thread A增加了一次，但thread A中语句必定会被执行多次， x 的值一定会被更新??? (实际上 $x \geq 10$? - 的确如此)
- c. 10,12,14,16,18,20: thread A写入前可能被调度到B B更新 x 后又被A用旧的值更新，造成 x 实际上虽然应该+4但是实际只+2，可能出现0~5次此情况
- d. 同process: PC等CPU寄存器, execution stack
不同process: 地址空间、文件描述符、代码段、数据段等资源 and 上下文???
- a. 都要: TCB (reg, PC, sp...)
- b. 不同process: PCB、file-descriptor table
- e. IO密集(多线程读写)，或者多核心下，计算的并行性足够好，不需要复杂的同步操作
- a. Same task, different data: 并行性足够好
- b. Different independent task (CPU+IO): CPU和IO混合?
- c. 题干中 signal processor 不代表单核?

-1
-1

P3

- a. Stdout 无输出?? (dup2 的作用: 把new_fd用old_fd覆盖)
stdout输出 starting main
- txt中: child, Ending main:0, parent, Ending main: 123
- b. Stdout: starting main child, Ending main:0
txt中: parent, Ending main: 123
- c. ??? (SIGTEST是啥?)
getpid
apple
orange(pid)
wait(pid, NULL, 0)

-2

(c)

-4

- 1) getpid()
- a. 2) signal(SIGTEST, orange)
- 3) apple()
- 4) waitpid(pid) or wait(NULL) or wait(0)
- d. 父进程开始退出，父进程退出后子进程被reparent给init。父进程可以通过自己设定 signal handler来决定收到sigkill后的行为，例如gracefully exit等。
- a. 注: SIGKILL无法被忽略
- b. Ctrl+c发的是sigint 会把整个前台进程组停掉
- c. 子进程继续运行
- e. Wait会立刻返回，parent process继续运行。
- a. 会返回一个 ECHILD
- f. (kernel stack是啥?)

-1

因为在进入kernel space的时候，kernel的执行有自己的execution stack，为了防止影响user space中的execution stack，以及实现kernel/user space的隔离，所以需要

-2

止影响user space中的execution stack，以及实现kernel/user space的隔离，所以需要切换。每个thread有自己的execution stack, 可能会进行不同的syscall或者其他内核操作，因此需要unique的kernel stack? ? ? ?

(f)

1. OS does not trust user's stack, user's stack may cause unpredictable error. So OS always enter a kernel-allocated stack before entering kernel.
- a. 2. Schedule frequently happens, if every thread has a unique kernel stack, they can easily switch between running and ready state. Otherwise kernel stack has to save and restore in every switching.
- b. 安全问题（需要有kernel stack），效率问题（thread scheduling 频繁发生，防止kernel stack每次都要save restore）

P4

- a. 并行是真实有多段代码在同时执行（多线程或多进程），并发是指通过调度使得多个job看起来是在同时执行，但是实际运行中可能是通过时间片切换等手段，每一时刻同时只有一个job在执行。

Parallelism refers to techniques to make programs faster by performing several computations at the same time. This requires hardware with multiple processing units.

Concurrency refers to techniques that make programs more usable. Concurrency can be implemented and is used a lot on single processing units.

- b. Process: init->ready->running->blocked (intr / unintr)->terminated
Thread: init->ready->running->blocked->terminated? ? ?

线程是轻量级的进程，一个进程中可能有多个线程，每个线程都属于一个进程。线程有自己的execution stack和cpu寄存器状态，进程则在线程之间共享地址空间、数据段、代码段、文件描述符等更多资源。

BASIS FOR COMPARISON	PROCESS	THREAD
Basic	Program in execution.	Part of process.
Memory sharing	Completely isolated and do not share memory.	Shares memory with each other.
Resource consumption	More	Less
Time required for creation	More	Less
Context switching time	Takes more time.	Consumes less time.

- a.

- c. fork执行的时候, kernel先复制当前PCB, 然后修改新PCB中的pid, 父进程等不被共享的属性, 然后复制父进程的内存到另一端空闲内存, 修改PCB中的指针完成新进程的内存分配, 最后修改新进程和父进程的fork返回值。

fork()

create a new process.

copy all user-space data to new process, e.g. PC, code, data and cache.

copy parent's PCB to new process.

update some information in PCB, e.g. PID, parent, running time and return value.

a.

- d. CPU REG, L1, L2, MEM, SSD, HDD 具体数值不确定?

本质上是在速度和容量之间的tradeoff。????

- e. User time: 在user space使用的时间 (例如计算代码)

sys time: 在kernel space使用的时间 (例如完成 IO 操作)

real time: 从进程开始运行到结束的真实世界经过的时间

real < user+sys: 多线程程序, 在IO线程完成IO操作时也在执行计算

real > user+sys: 在执行中调度到了其他进程运行, real 还在继续计时

a. Wait, sleep, IO operation

L7 死锁

2020年4月8日 14:20

- 饥饿和死锁的关系
 - 饥饿：某个线程一直在等
 - 死锁：成环等待 circular waiting for resources
 - 死锁一定造成饥饿，但饥饿不一定是因为死锁
 - 例子：低优先级被高优先级饿死
 - 饥饿是有可能自己解决的（高优先级任务结束），但是死锁是无法自发解决的（必须要外部干预）

- 死锁条件
 - 互斥
 - 边拿边等：拿着已有的资源，去请求剩下的资源
 - 无抢占：只能线程自己释放资源，不能抢占
 - 循环等待

- 判断是否可能死锁：画资源分配图

- 资源分配图是动态的，不能只依据单一时刻的有环来判断死锁
- 看所有任务能否自发结束（是否还有 unfinished)
 - 只要有一个可能顺序让死锁不发生，那就不会检测出死锁

```
[Avail] = [FreeResources]
Add all nodes to UNFINISHED
do {
    done = true
    Foreach node in UNFINISHED {
        if ([Requestnode] <= [Avail]) {
            remove node from UNFINISHED
            [Avail] = [Avail] + [Allocnode]
            done = false
        }
    }
} until(done)
```

◆ Nodes left in UNFINISHED \Rightarrow deadlocked

- 处理死锁的方法
 - 允许系统进入死锁，再从死锁中恢复
 - 需要死锁检测算法
 - 需要可以强制抢占资源（回滚资源请求）
 - 避免系统死锁
 - 需要监测所有锁请求
 - 拒绝可能导致死锁的锁请求
 - 直接忽略，认为不会发生死锁
 - OS不管，需要程序员自己处理
- 如果已经发生死锁了，怎么办？
 - 中止进程，释放资源

- 强制抢占资源
- 回滚死锁进程的占用
- 避免死锁的方式
 - 无限资源：大公司可以，个人不实际
 - 完全禁止资源共享：不可能
 - 禁止等待：如果资源被占用，不断重试，而不是等待资源释放
 - 一开始就申请好所有资源：难以估计，一般会高估
 - 要求按照特定顺序获取资源：在应用层面而不是OS层面完成
 - 问题：现代系统中都无法完美使用
- 银行家算法
 - 去年考了
 - Q：如何用银行家算法解决哲学家筷子问题？
- 期中选择14修正：临界区不是原子操作，如果A在CS，B不能进入A所在的CS，但是可以进入程序的其他区域，这个时候依然是可以做scheduling的（除非进入CS的时候关闭Interruption），C选项在现在的实现中是可以的，因此本题没有答案

L8 地址翻译

2020年4月15日 14:34

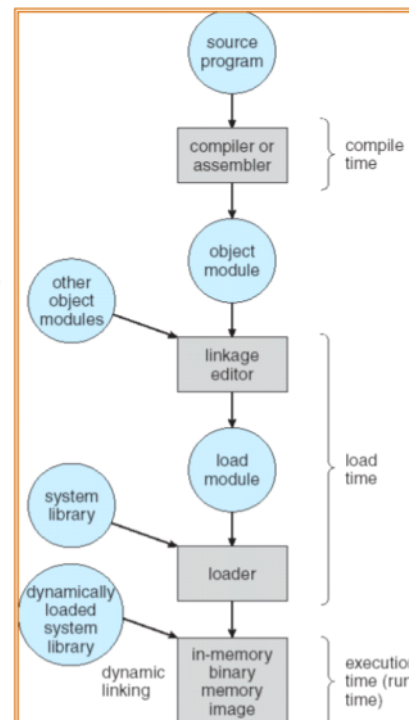
Lec 2~7 进程管理

现在开始地址翻译

- 问题：多个进程如何共存
 - CPU：调度
 - 内存：地址空间和地址翻译
 - 根本方式：资源的虚拟化
- 资源虚拟化
 - 现实中，进程/线程需要共享物理资源
 - 为了实现多路复用（Multiplex），需要把资源虚拟化，然后维护虚拟资源和物理资源的对应
- 回顾：单线程和多线程模型
 - 线程封装concurrency
 - 地址空间实现了保护
- 内存复用的重要部分
 - 保护：保护进程内的私有内存不被其他进程访问
 - 隔离内核和用户程序
 - 每个进程只能读写自己的私有的虚拟内存
 - 有控制的重叠 controlled overlap：允许进程之间共享内存
 - 共享数据或代码
 - 自己的私有区域外，还能读写共享区域
 - 翻译：实现从虚拟地址空间到物理地址空间的转换
 - 可以用来实现以上两者
- 内存复用的时机
 - 可能在 编译/加载/运行 时
 - 程序运行路线图

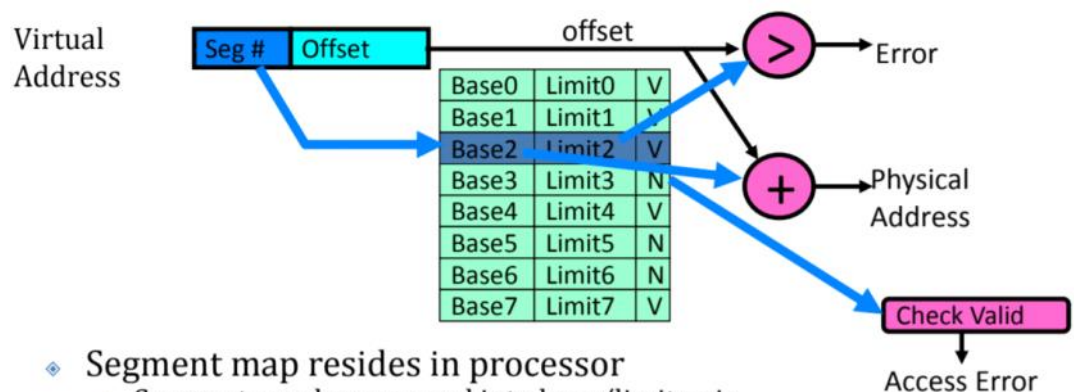
Multi-step Processing of a Program for Execution

- ◆ Preparation of a program for execution involves components at:
 - (1) Compile time (i.e., “gcc”)
 - (2) Link/Load time (UNIX “ld” does link)
 - (3) Execution time (e.g., dynamic libs)
- ◆ Addresses can be bound to final values anywhere in this path
 - ◆ Depends on hardware support
 - ◆ Also depends on operating system
- ◆ Dynamic Libraries
 - ◆ Linking postponed until execution
 - ◆ Small piece of code, *stub*, used to locate appropriate memory-resident library routine
 - ◆ Stub replaces itself with the address of the routine, and executes routine



- 视具体硬件和实现而定
- *stub*：动态加载库加载时被替换成真实的地址
- 实现内存复用的方式
 - Uniprogramming
 - 完全没有翻译和保护

- 一次只运行一个程序
- 运行的程序对内存有完全的访问
- Multiprogramming
 - 有翻译，没有保护
 - 早期os的实现方式
 - 用 loader/linker 实现，加载的时候加载到不同物理地址
 - 没有保护：一个程序crash依然会影响其他程序
 - ◆ 也可以访问其他程序的地址？
 - 有翻译，有保护
 - 依然是加载时完成
 - 但是加入 limitaddr, baseaddr （上下界）
 - 如果访问界限外的，会产生 illegal access
 - switch的时候，从 PCB 加载新的 base/limit
- Virtual memory
 - 在执行时实现翻译和保护
 - 引入了 MMU memory management unit 作为 virtual 和 physical 之间的桥梁
 - 一般是 on-chip （在 CPU 内）
 - 加载的时候都加载到用户地址空间的相同区域
 - 实现1: Base & bound
 - Virt addr加一个base 得到 phy mem
 - 检测 bound 来看是否越界
 - 对应用来说 自己是唯一在运行的程序 地址从0开始
 - 物理空间上的地址更改不会影响虚拟空间内的内存布局
 - 问题
 - ◆ 导致内存碎片
 - ◆ 对spare address space支持不好（虚拟空间内有多块）
 - ◆ 进程间共享难以实现（共享代码和数据）
 - 实现2: multi segment
 - 更灵活的翻译机制
 - 不是以程序为基础 而是以logical segment为基础
 - 每个segment都有自己的base和limit
 - Segment table 段表：在 CPU 内（比较小）：记录了 segid, base, limit
 - ◆ Segid: 内存地址最前面的2个bit（如果有4个seg的话）
 - ◇ 砍掉了前2bit之后，剩下的就是offset 偏移了
 - 翻译过程
 - ◆ 虚拟地址 -> 查找到对应的seg -> 加上 base -> 查看是否超过 limit -> 物理地址



- 期末可能会考手动翻译
- Key observations
 - ◆ 虚拟地址里有洞（未使用的地址），没有对应到实际的物理地址，应用程序不应该直接访问，不然会触发trap
 - ◆ Stack, heap应该可以在valid range外访问，这样才能增长（trap之后kernel增加相应的地址分配，增长stack/heap）
 - ◆ 访问seg table需要进入保护模式，保证安全：code ro,data/stack rw, shared ro或rw
 - ◆ 在context switch的时候，可能会需要把内存存储到硬盘

(swapping)

□ 问题

- ◆ 碎片依然存在，无论是内碎片还是外碎片
 - ◇ 内碎片：分配给了进程但是进程自己没用
 - ◇ 外碎片：分配的物理内存之间的空隙 free gaps
 - ◇ 分段机制下依然会有内碎片：例如一个增长了又缩小的stack
- ◆ 需要在物理内存中存储不同大小的chunk
- ◆ swapping的时候更复杂，需要同时处理多块
 - ◇ Limited options for swapping to disk???
- ◆ 为了fit everything，可能需要多次移动进程???

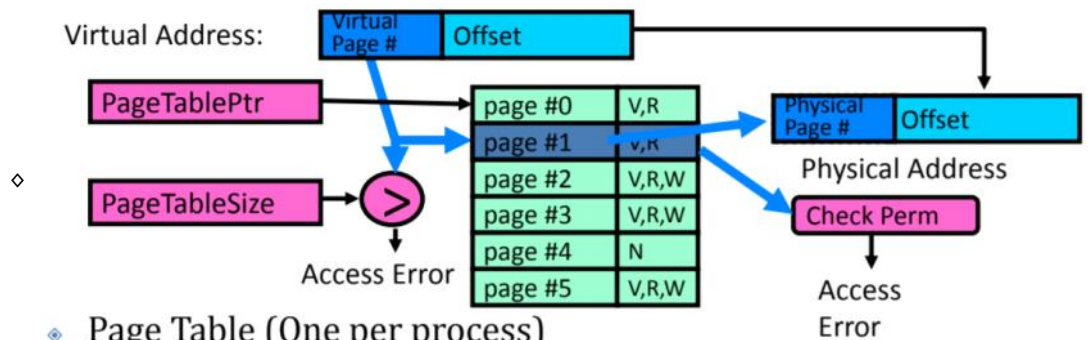
▪ 实现3: paging 分页

- 把物理内存分成大小固定的page
- 每个page都是等价的，用一个bit来指示是否分配
- 问题：page应该多大？

- ◆ 太大：可用的page总数量太少，会导致很多内碎片
- ◆ 太小：page table的entry会太多
- ◆ 现代OS一般是小page：1k~16k
 - ◇ 所以一个程序会需要很多page

□ 实现

- ◆ 每个进程有自己的page table
 - ◇ Page table存储在物理内存中
 - ◇ 第i个位置，存储了物理内存的page号（物理的page地址），valid bit，读写权限bit：page table entry (PTE)
- ◆ 翻译

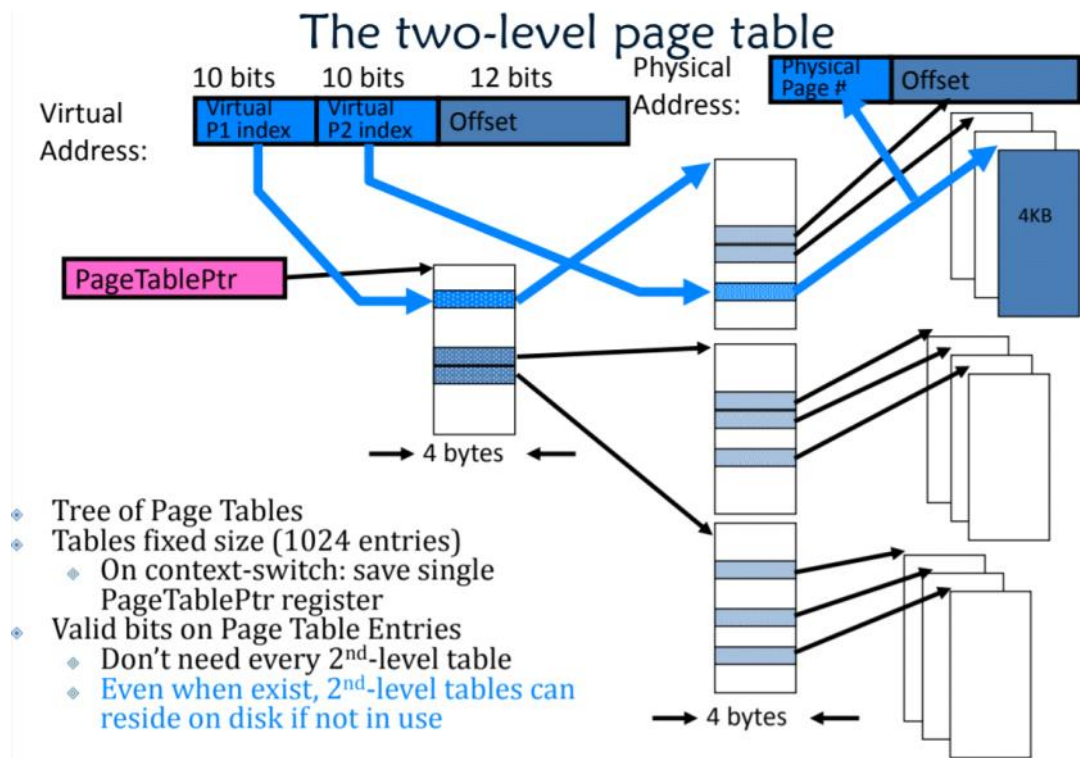


◆ Page Table (One per process)

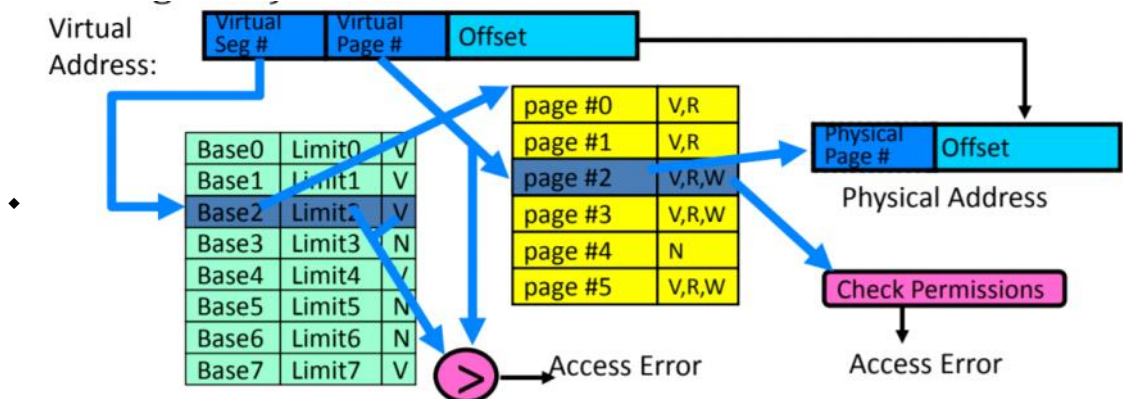
- ◆ 共享内存：在page table中直接指向相同的物理页就好了
- ◆ 在context switch的时候，需要改 page table pointer和limit
 - ◇ limit: valid bit和读写权限bit
 - ◇ limit不是应该和page table本身存储在一起吗???

□ 分析

- ◆ 优点：实现简单，容易分享
 - ◇ 完全解决外碎片
- ◆ 缺点：地址空间稀疏的时候：Page table entry太多：32bit地址下 1K的page，有 2^{22} 个entry
 - ◇ 实际上不是所有entry都有用，可以只存储有用的entry：working set
 - ◇ 2-level page table：一级不够就再来一级
 - ▶ 每一级长度固定
 - ▶ 第二级表只在需要的时候创建，不用的时候也可以swap出去
 - ▶ 底层连续、高层分散
 - ▶ 最好情况下：page table数 \approx page数 (n page，分散足够稀疏，1个一级，n个2级) ????



- 实现4: paging + seg 分页+分段
 - 第一层用seg, 第二层用page
 - 这样的话第一层的seg table在CPU中, 比第一层page table在内存中会更快
 - Context switch的时候需要存储第一层seg表+指向第一层seg表的指针



- 共享的时候, 甚至可以直接共享二级page table

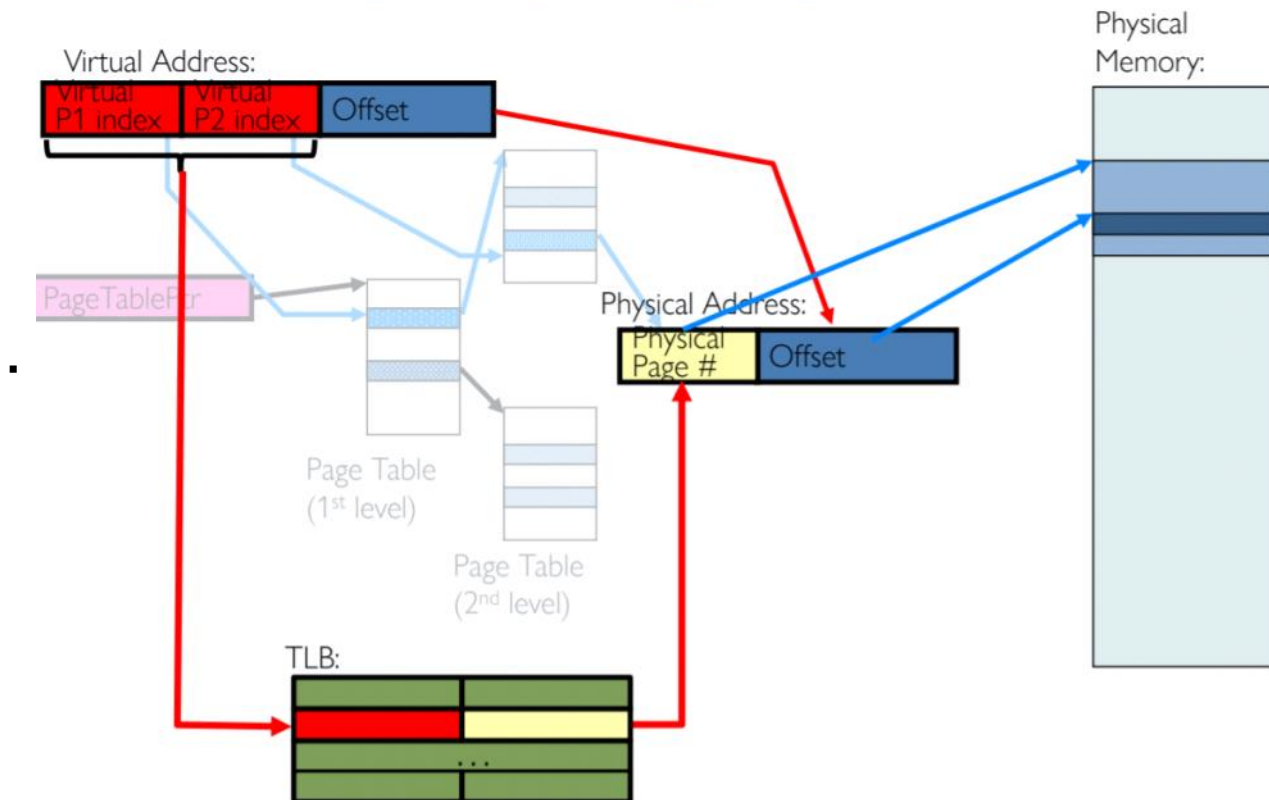
- 问题
 - why do we need V/N in seg table in the approach of seg+page? since we can set the address to null
 - limit不是和page table存在一起吗
 - 动态链接是怎么实现的?

L9 缓存、请求分页

2020年4月22日 15:25

- 缓存
 - 缓存：数据副本的存储
 - 作用：frequent case更快, infrequent case less dominant
 - 缓存有用的前提：frequent case足够快, infrequent case不太慢
 - 量度：平均访问时间：average access time: $(\text{Hit Rate} \times \text{Hit Time}) + (\text{Miss Rate} \times \text{Miss Time})$
 - 缓存有用的原因：局部性
 - 空间局部性
 - 时间局部性
- 地址翻译和缓存
 - 之前单纯用 MMU（内存管理单元）
 - MMU可能是硬件实现，也可能是软件实现
 - 硬件实现：hardware tree
 - 软件实现
 - 翻译过程：虚拟地址 -> seg table -> page table -> 物理内存
 - Seg table 在 CPU 中
 - Page table 在内存/磁盘中（如果是不频繁使用的被swap出去了）
 - 物理地址对应的数据在内存中
 - 如果每次访问都要这么走一次就太慢了，至少要3次内存访问才能拿到数据
 - 1层page table, 2层page table, 实际物理地址
 - Memory access faster than DRAM access? ? ?
 - 考虑使用CPU的L3 cache的情况
 - 如果数据已经在L3 cache内
 - ◆ 如果有TLB, 先TLB拿到物理地址, 然后发现L3有, 就直接拿了
 - ◆ 如果没TLB, 得先用2次内存访问拿到物理地址, 再发现L3有, 这个效果就很糟糕了
 - 加速地址翻译过程：缓存之前的翻译结果 cache translation
 - TLB: translation lookaside buffer (快表)
 - 调MMU之前先看TLB有没有, 调完MMU把结果放入TLB
 - 为什么有用: Page locality是存在的: instruction/stack有, data有一些
 - TLB自身也可以有多层
 - TLB的位置: CPU的L1和L2
 - 有了TLB之后的地址翻译过程
 - 最好情况下一次就可以获得物理地址, 1次内存访问就可以拿到数据了

Putting Everything Together: TLB



- 请求分页 demand paging
 - 问题：现代程序需要很多内存，但很多时候用不上全部
 - 90-10：90%的时间在用10%的代码
 - 如果全部都存入内存 就太浪费了
 - 做法：把main memory作为disk的缓存
 - 大部分page都在disk上，用到了再放入mem
 - 关键要素：
 - Block size：1 page（和page size相同）
 - cache的存储策略：fully associative：任意 virtual -> physical
 - 如何在cache里找到我们要的page
 - Step1: TLB check：如果TLB有，直接返回物理内存地址
 - Step2: page table 遍历
 - Page replacement policy：在ram满了之后，放新page的时候把原来ram中的哪个驱逐出去？
 - LRU, random...
 - Page miss：需要去磁盘重新把miss的page拿回来
 - write：不仅要写cache，也要写到磁盘
 - Write-through：同时写cache和磁盘
 - Write-back：先写到cache并标记dirty bit，被驱逐的时候再写入硬盘
 - 这里得用write-back，才能有足够快的性能，所以需要dirty bit
 - 实现机制
 - PTE: page table entry 帮助实现demand paging
 - Valid/invalid: page是否在mem中
 - valid：PTE在mem, 指向物理内存中的页
 - Invalid: PTE在disk, 需要用PTE去disk里把页找回来
 - 用户访问了一个invalid PTE之后会发生什么？
 - MMU在TLB和page table都没找到：确认PTE是invalid的

- MMU mem management unit 触发page fault, OS介入
- OS的操作
 - ◆ 首先从mem中选择一个old page, 作为替换的目标
 - ◆ 如果old page修改过 (Dirty=1), 把mem中的新版本写入磁盘
 - ◆ 把old page的PTE标记为invalid, 同时把TLB里的cache invalid掉
 - ◆ 从磁盘中加载new page到内存
 - ◆ 更新PTE, 让PTE指向内存中的new page
 - ◆ 从之前中断的地方继续
- 程序继续运行的时候, 新的page就已经加载到内存了
- 从磁盘中加载page的IO操作进行中的时候, OS会跑其他waiting的 process, 来优化CPU的使用率
- Cost model
 - Demand paging基本就是cache, 也可以计算EAT Effective Access Time
 - 如果要EAT不比直接访问内存慢太多, 需要很低的page fault概率
 - 什么情况下会导致 miss?
 - Compulsory miss: 之前没用过的page, 第一次载入内存
 - ◆ 可以通过prefetch减少, 在用到之前就载入
 - Capacity miss: 内存空间不足, 没办法把一个process需要的所有page放入内存
 - ◆ 增大系统的总内存容量
 - ◆ 如果内存中有多个process, 可以修改不同process分到的内存容量 (page数)
 - Conflict miss: 理论上虚拟内存中不存在, 因为这里用的是fully associate, 虚拟内存无限大
 - Policy miss: 根据替换策略被踢出mem的page, 在之后访问的时候miss
- 替换策略
 - 为什么要关心替换策略
 - 回disk的成本很高
 - 应该尽可能把重要的页留在内存中
 - FIFO
 - 直接把最老的扔掉
 - 公平, 每个页在内存中存留的时间都相同
 - 问题: 实际上是更多的是把heavily used的page扔了, 而不是infrequently used
 - MIN minimum:
 - 把最远不会被用到的page扔掉
 - 理论最优, 作为比较基线
 - 但是实际应用中用不上: 因为不知道future
 - RANDOM
 - 每次随机选一个扔掉
 - TLB的做法: 硬件实现很简单
 - 但是难以预测, hard to make real-time guarantee
 - LRU least recently used 最近最远未被使用
 - 换掉当前cache中最久没有用的页
 - 动机: 基于时间局部性: 如果很久都没用到了, 那么大概将来也用不上
 - ◆ LRU看起来是MIN的一个足够好的近似
 - 实现: 用一个list 链表
 - ◆ 一个head, 一个tail, tail指向LRU项
 - ◆ 如果某个page在cache被用了, 从链表中取出然后移到头部
 - ◆ 需要支持的操作: Lookup 查询某页是否在cache中, evict-one 从lru中删除最老的页, update更新某一页的使用时间

- ◇ 可以O(1)实现?
- 问题: 实现起来需要比较多的指令
 - ◆ 实际使用 approx. LRU
- LRU的表现也能很糟糕: ABCD + size=3 会不停替换
- 更多内存一定就会有更少的page miss? 不是的
 - LRU/MIN有这个特性的
 - Bélády' s anomaly: 有些替换策略不满足这个性质
 - ◆ FIFO没有这个性质
 - 核心原因: LRU/MIN 下, 小内存时cache的内容是大内存时的子集, 但是FIFO中不同内存下存储的可能完全不同
- LRU的实际实现
 - LRU的理想实现
 - ◆ 每个page上用timestamp标记访问时间
 - ◆ 把list按照timestamp排序
 - ◆ 不现实, 开销太大: 链表中表项提前需要比较大的constant cost (就算O(1)查询)
 - 实际: clock 算法
 - ◆ Approx. LRU
 - ◆ 只是换掉一个old page, 但是不保证是oldest
 - ◆ 放入的时候use bit=1
 - ◆ 需要替换的时候 看当前指向的item的use bit
 - ◇ 如果use bit=1, 说明最近用过, use bit置0然后移到下一格
 - ◇ 如果use bit=0, 说明最近没使用, 则可以evict 当前item
 - ◆ 不会无限循环: 如果所有都是1, 顶多也就多转一圈, 相当于回落回了FIFO
 - ◆ Hand 移动的很慢: 好事, 说明没有很多page fault / 可以很快找到use=0的page
 - ◇ 移动的快: 很多 page fault / 很多都是use=1
 - ◆ 另一种理解: 把page分成了两种 young (use=1), old (use=0)
 - ◇ 为什么不直接分成两个区?
 - ◆ 变体: N-th chance version
 - ◇ Use bit + counter
 - ◇ 相当于每个page有N次机会不被evict
 - ◇ 如何选择N
 - ▶ N大: 更接近LRU, $N \sim 1k$ 时近似足够好 (实验结果)
 - ▶ N小: 效率更高, 不然得转很多圈
 - ◇ Dirty page 多给N, 在回到全局N的时候做一次write back
 - ◆ 变体: second chance list algorithm
 - ◇ Active list (FIFO) + SecondChance list (LRU)
 - Demand Paging (more details) ???????
 - Allocation of Page Frames (Memory Pages)?????
 - Fixed/Priority Allocation?????

○ 实现细节

- OS怎么选free frame (mem中可以供替换的位置)
 - OS维护一个free list

- unix内存太满的时候 运行reaper机制

L10 I/O和存储设备

2020年5月6日 14:05

- 内存回顾
 - 我们讲过的只是主要原则
 - Memory Management Unit
 - Translation Lookaside Buffer
 - Page Table Entry
 - 实际的os比介绍过的内存知识更复杂
 - Memory zone
 - 多种allocation
 - Canonical hole

图 7.5 描绘了通用的 MMU，以及各级 CPU 缓存和主内存。

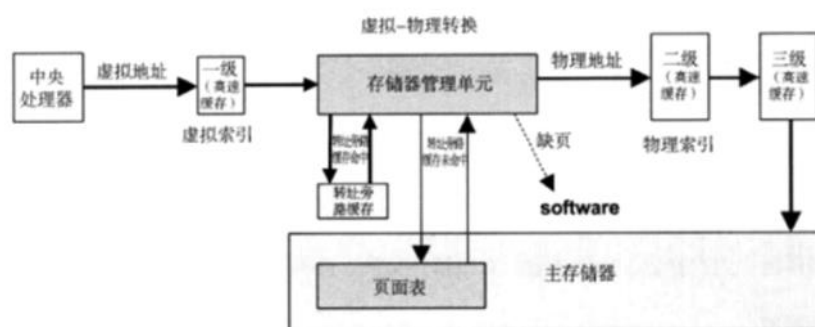
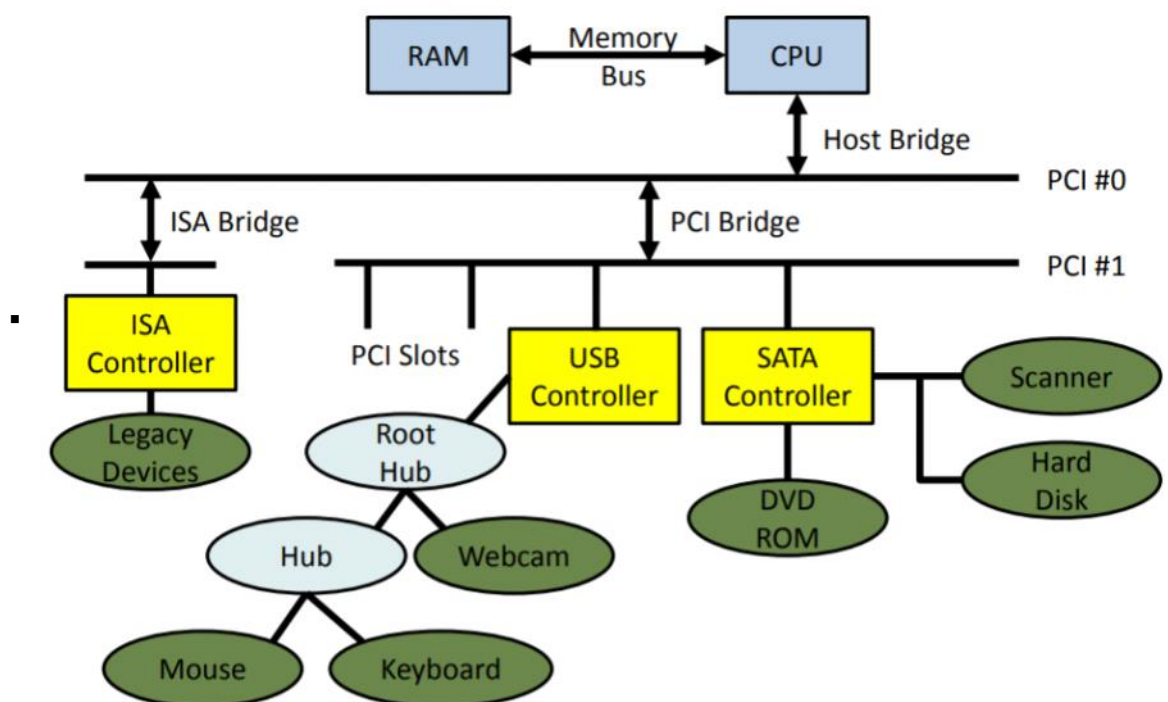


图 7.5 内存管理单元

- https://sylvanassun.github.io/2017/10/29/2017-10-29-virtual_memory/
- 地址翻译
 - Tlb
 - 实际上是cpu内mmu里的缓冲区
 - 和l1,l2,l3是独立的
 - Page table (lvl 1, lvl 2)
 - Physical memory addr
- 请求分页 demand paging
 - Physical addr -> page # + offset
 - 检查内存中是否存在（可以把内存看作cache）：找page table，看page的valid bit
 - 每个进程有自己的page table
 - 每个page table内有应用用过的所有page的page entry
 - 触发page fault
 - Page fault handler
 - schedule其他进程
 - 加载page到内存，并替换掉一个旧page：page replacement policy
 - 更新page table，设定对应的page table entry valid=1
 - 重新执行指令
- I/O
 - IO设备可以用IO控制器支持
 - 处理器可以先写到内存再写入IO设备，也可以直接写IO寄存器

- IO虚拟化的问题
 - 设备多种多样，如何标准化接口
 - 设备可能是不可靠的
 - 速度慢、难以预测性能表现
 - 不同的IO设备，速度差异12个数量级
 - 需要能够处理各种不同速度的IO设备
 - 快的设备不应该有太多额外开销
 - 慢的设备不用浪费时间等
- IO的操作参数
 - 数据粒度：byte / block
 - 键盘：每次提供一个byte (char)
 - 其他设备：每次提供一个block (disk, network...)
 - 访问模式：顺序/随机
 - 顺序：tape
 - 随机：其他设备 disk...
 - Continual monitoring / interrupt
 - 传输机制：programmed IO / DMA
 - DMA: direct memory access
- IO子系统的目标：为不同的设备提供统一的接口
- PCI总线架构

Example: PCI Architecture



- 处理器和设备的操作过程
 - CPU和controller交互
 - Controller里面有register, cpu可以读取和写入
 - 也有可能memory来管理队列、bitmap...
 - 两种交互模式
 - IO instruction：用指令来读写controller的register，完成IO操作
 - ◆ pro：硬件实现简单，容易编程

- ◆ con: IO操作需要cpu cycle处理, cpu cycle数和数据量成正比
- Memory mapped IO: IO设备直接映射到物理内存地址, 直接像内存一样用
 - ◆ 可以被地址翻译保护
 - ◆ DMA: 允许 controller 访问 memory bus, 直接传输 data blocks, 不需要额外cpu cycle, 在完成后通过终端通知cpu

Memory-mapped I/O allows the CPU to control hardware by reading and writing specific memory addresses. Usually, this would be used for low-bandwidth operations such as changing control bits.

DMA allows hardware to directly read and write memory *without* involving the CPU. Usually, this would be used for high-bandwidth operations such as disk I/O or camera video input.

来自 <<https://stackoverflow.com/questions/3851677/what-is-the-difference-between-dma-and-memory-mapped-io>>

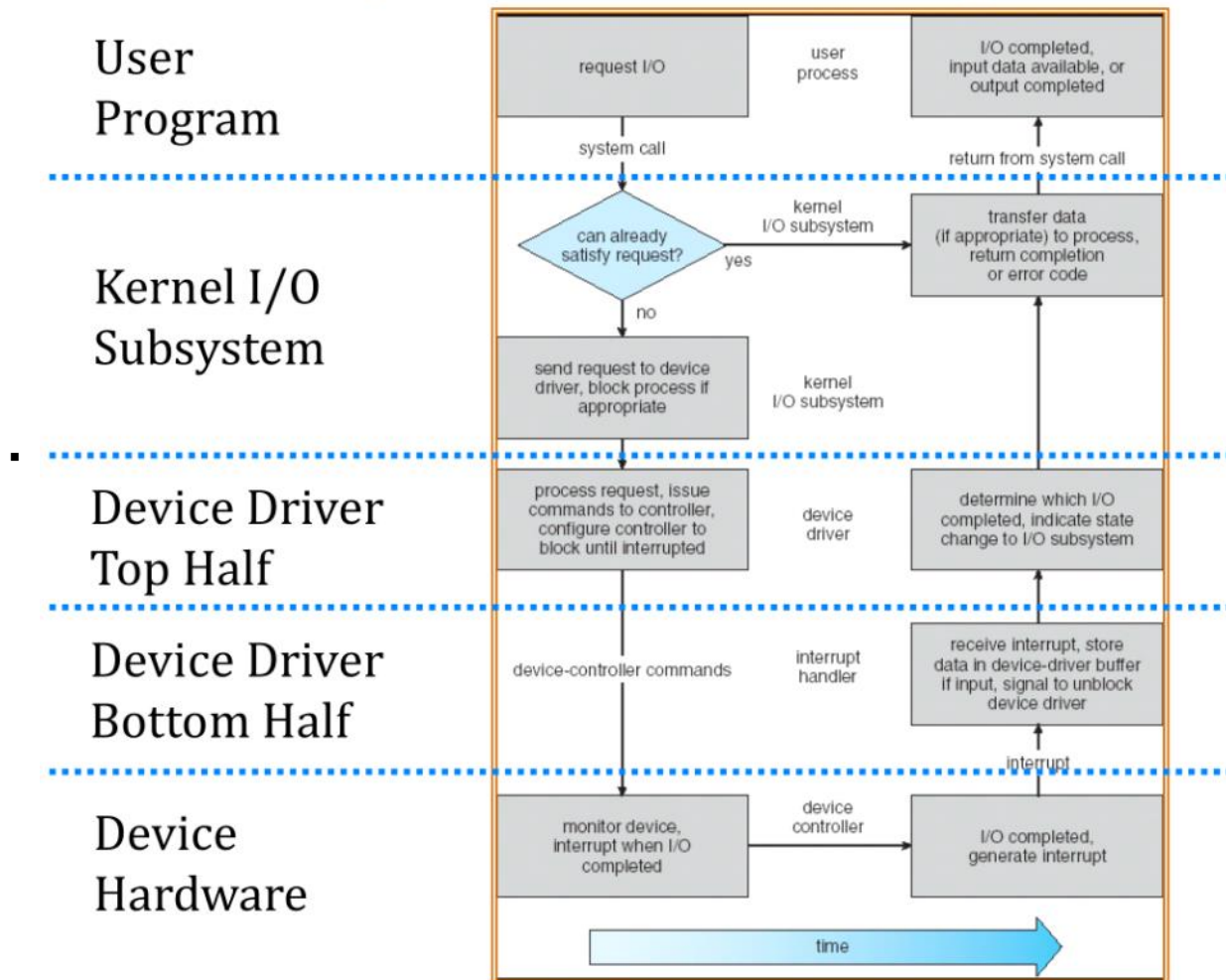
○ IO 设备如何通知OS

- 需要在完成/出错的时候通知OS
- IO 中断
 - 需要OS介入的时候触发中断
 - pro: 不用一直等, 便于unpredictable的event
 - con: OS处理中断的开销高
- 轮询 polling
 - OS定时检查一个Status register
 - pro: 额外开销小
 - con: 要一直检查, 如果是infrequent/unpredictable 的IO操作, 会浪费很多cpu cycle
- 实际中是两种方法一起用的
 - 例子: 网卡
 - ◆ 中断通知第一个数据包到达
 - ◆ 轮询读取接下来的数据包直到清空queue

○ 设备驱动 device driver

- 内核中直接和硬件交互的代码
 - 直接实现了标准的内部接口
 - 设备特定操作可以用ioctl system call完成
- 两个部分
 - Top half: system call会调用到的
 - ◆ 启动IO过程, 可能会把相关thread挂起至IO完成
 - Bottom half: 中断实现
 - ◆ 实际去完成transfer
 - ◆ IO操作完成后唤醒thread

Life Cycle of An I/O Request



io性能评价标准

- 响应时间/延迟：完成操作的时间
- 带宽/吞吐量：操作执行的速度
 - MB / Mb / MiB ?????
 - Transfer capacity? 用MB计算?
- 启动时间/overhead：初始化一个操作到开始操作的时间
- 许多io操作的总时间，基本上是关于n线性的
 - $\text{Latency}(n) = \text{Overhead} + n/\text{TransferCapacit}$
- 考虑到启动时间的存在，我们可以计算出一个真实的带宽
 - $\text{Bandwidth} = n/(S + n/B) = B*n/(B*S + n) = B/(B*S/n + 1)$
 - 随着数据量n的增加，这个带宽会越来越靠近理论的带宽
- Half-power point：此时真实带宽是理论带宽的一半
 - $n = S*B \rightarrow \text{Bandwidth} = B/2$
- 决定理论带宽（峰值带宽）的因素
 - 总线速度
 - 设备自身的带宽
 - ◆ 磁盘的旋转速度
 - ◆ NAND flash的读写速度
 - ◆ 网络信号传输的速度

磁盘驱动器

- 硬盘、SSD
- 柱面、磁道、磁头...
- 磁盘读写需要的时间
 - 实际上开始前还有 Queueing time, Controller time
 - Queueing time: 多个读写请求可能请求同一个设备, 请求会排队
 - Controller time: 从软件请求转换成硬件指令
 - 一般很短, 常常被忽略
 - 寻道时间: 把磁头移过去
 - 一般平均寻道时间是完整寻道时间的 1/3 (积分)
 - 旋转延迟: 等待目标扇区旋转到磁头下
 - 完整旋转时间 可以用 RPM 计算: $60000 \text{ ms/min} / 7200 \text{ rev/min} \approx 8 \text{ ms/rev}$
 - ◆ Revolution Per Minute
 - 一般用完整旋转时间的一半作为旋转延迟
 - 传输时间: 真正传输数据, 读取一个 block 的时间
 - 可以用传输速率计算: $\text{sector size} = 1\text{KB}$, 速率 4MB/s
 - $1024 \text{ Byte} / (4 \times 10^6 \text{ Byte/s}) \approx 0.26 \text{ ms}$
 - 从时间计算实际传输速率
 - ◆ Read sector from random place on disk:
 - ◆ Seek (5ms) + Rot. Delay (4ms) + Transfer (0.26ms)
 - ◆ Approx 10ms to fetch/put data: 100 KByte/sec
 - ◆ Read sector from random place in same cylinder:
 - ◆ Rot. Delay (4ms) + Transfer (0.26ms)
 - ◆ Approx 5ms to fetch/put data: 200 KByte/sec
 - ◆ Read next sector on same track:
 - ◆ Transfer (0.26ms): 4 MByte/sec
- SSD
 - Flash-based
 - 写入磨损
- Startup cost of IO
 - Syscall overhead
 - OS processing
 - Controller overhead
 - Device startup
 - Queueing
- 性能表现
 - 响应时间, 吞吐量, 实际带宽
 - 实际带宽 = 传输大小 / 响应时间
 - $N / (\text{startup time} + n/\text{bandwidth})$
- 磁盘调度 Disk scheduling
 - 合理排序磁盘请求的顺序
 - 目的: 最小化 seek time -> 最小化总磁头移动距离
 - 策略
 - FIFO
 - Shortest Seek Time First: 选离当前位置最近的
 - ◆ 类似于 SJF, 可能导致饥饿
 - SCAN: 类似于电梯的做法, 从一头到另一头, 然后再反向回来

- ◆ 如果两个请求分别在磁盘两端，可能会等很久
- ◆ 忽视了磁盘的旋转开销
- C-SCAN：视作循环边界，到达一头后直接跳到另一头
 - ◆ 两端的请求处理更好一些
- LOOK, C-LOOK：不是到边界，只到最大的请求位置
- 选择策略
 - 对于大负载下，SCAN, C-SCAN表现更好

L11 文件系统

2020年5月13日 14:44

- 文件操作层级
 - High level io, low level io, syscall, filesystem, driver
- 回顾：C中的IO操作，都是基于 file descriptor，作为文件的 handle
 - Flags: access mode, open flags, operating mode
 - Mode: permission bit
- 文件系统
 - 把块设备的 Block interface 转换为 文件/目录 的 OS 层
 - 组件
 - Naming: 可以用路径和文件名定位文件，而不是 block（用户视角）
 - 磁盘管理：管理可用空间，碎片处理???（块设备视角）
 - 保护：避免权限外的访问，keep data secure
 - 可靠性：在异常时保护数据，keep file durable
- 对文件的不同视角
 - 用户：持久化的数据结构
 - Syscall：一长串 bytes，和底层设备无关
 - Inside OS：一系列 block 的集合
 - block：logical transfer unit
 - Sector：physical transfer unit
 - Block size > sector size
 - 1 block = multiple sector
 - 从用户视角到OS视角：用户视角处理的是 bytes，但是实际处理都是 block
 - 读整个block，返回需要的bytes
 - 写bytes到block，再写回整个block
- 实现文件系统
 - 目录
 - Hierarchical structure
 - 文件和目录的集合
 - Mapping names to files
 - 每一项有名字和属性
 - DAG而不是tree：有hard link
 - hard link *
 - soft link **
 - 文件
 - Named permanent storage
 - 数据
 - 元数据
 - ◻ 文件名不是文件 entry里存储的元数据之一

- 磁盘上存储的entry
 - 文件
 - 目录
- 用顺序方式访问磁盘上的sector
 - 物理块：用 cylinder, surface, sector 来定位
 - 逻辑块寻址：每个块编号 +1
- 挑战
 - 处理坏 sector
 - Hardware 对 OS 隐藏实际的 sector 结构
 - 需要记录 free block: free block link -> bitmap
 - 需要 structure files: file header?????
 - 文件头，记录一个文件到底被放在了哪些 block 里
 - 可以利用访问/使用的pattern来优化文件的存放
- 文件系统的 layout
 - 种类
 - Contiguous allocation
 - Linked allocation -> FAT
 - Inode allocation -> NTFS
 - Contiguous allocation 连续分配
 - 直接连续放
 - 删除和插入都还行
 - 问题：外碎片严重：需要整理磁盘碎片
 - 问题：不好增长
 - 现实世界使用：CD
 - Linked Allocation
 - 把磁盘分成很多个固定大小的block
 - 每个block可以指定下一个block的位置
 - 可以处理文件大小变化了，解决了外碎片的问题
 - 问题：内碎片，最后一个block可能用不完
 - 问题：不好确定大小 -> 改进：把大小和文件元数据一起存储
 - 问题：随机访问差 -> 改进：把next block的链表用数组方式直接存在一起
 - FAT
 - 现实世界使用：FAT
 - Linked-list approach
 - DOS里 block被叫做cluster
 - 16, 32: block的bit length, 决定了寻址空间的大小
 - ◆ FAT32实际上是28个地址位，FS自己预留了4个
 - 计算可用总容量：block的个数 * 每个block的大小
 - ◆ FAT32, block size=32kb: $2^{28} * 2^5 * 2^{10} = 2^{43}$ (8TB)
 - 读取时，一层一层读
 - Directory entry: 32 byte

- ◆ 文件名: 8 filename + 3 ext
- ◆ 最大文件大小: 4G ($2^{32}-1$)
- ◆ 为什么需要存储 first cluster address: 知道从哪里开始读
- ◆ 为什么需要存储 size: 知道读到哪里结束 (+方便常见的统计任务?)
- ◆ 头里面的数值都是 little-endian, 需要反一下
- LFN: Long File Name: 把文件名存在 directory entry 里
 - ◆ 一个可以存 13 个 unicode char (2byte): 5+6+2
 - ◆ 一个文件最多可以用20个LFN (结尾的和 0x40 OR, bit6设定为 1)
 - ◇ 理论最大文件名长度是260, 但是MS限制了文件名长度是 255 (现有255再有20的限制)
 - ◆ 反着存
- FAT读文件: traverse directory
 - ◆ 1 根据路径找到对应的 directory entry
 - ◆ 2 从 directory entry中读出start cluster, size
 - ◆ 3 读第一个 cluster
 - ◆ 4 从 FAT 表中找到 start cluster 的下一个, 读
 - ◆ 5 结束判定: FAT表中next=EOF, 读到的总数=size
 - ◇ 读最后一个cluster的时候不用读完, 可以用size来确定最后还要读多少
- FAT 写文件
 - ◆ 1 一直读, 直到最后一个 cluster
 - ◆ 2 开始写 non-full cluster
 - ◆ 3 如果最后一个 cluster 还不够写, 会从 fs 申请新的 cluster (FSINFO)
 - ◇ 找下一个 free cluster 是 circular, next-availble search, 有空间局部性??
 - ◆ 4 写完了, 更新 FAT1, FAT2 和 FSINFO
 - ◆ 5 最后更新 directory entry 中的 file size
- FAT 删文件: lazy delete
 - ◆ 1 把所有的 cluster 在 FAT 表中 de-allocate, 更新 FSINFO
 - ◆ 2 把 dir entry 的第一个byte改成0xE5, 标记为已删除
 - ◆ 注意: 并没有删除实际数据, 只是标记为可以被使用
 - ◆ 文件恢复: 首先找到dir entry, 读第一个cluster, 然后circular search (用free cluster往后重建)
- 使用: CF/SD/USB 设备
 - ◆ Most commonly used FS: Space Efficient & Simple
 - ◆ Poor performance, no security
- 注: FAT的性能问题: FAT表需要能够快速随机读写, 除非加载到内存不然速度会受限 (见lab)

- 设计一个 FS
 - 考虑 critical point
 - 读写
 - Size
 - 文件/文件夹
 - Allocate/free blocks

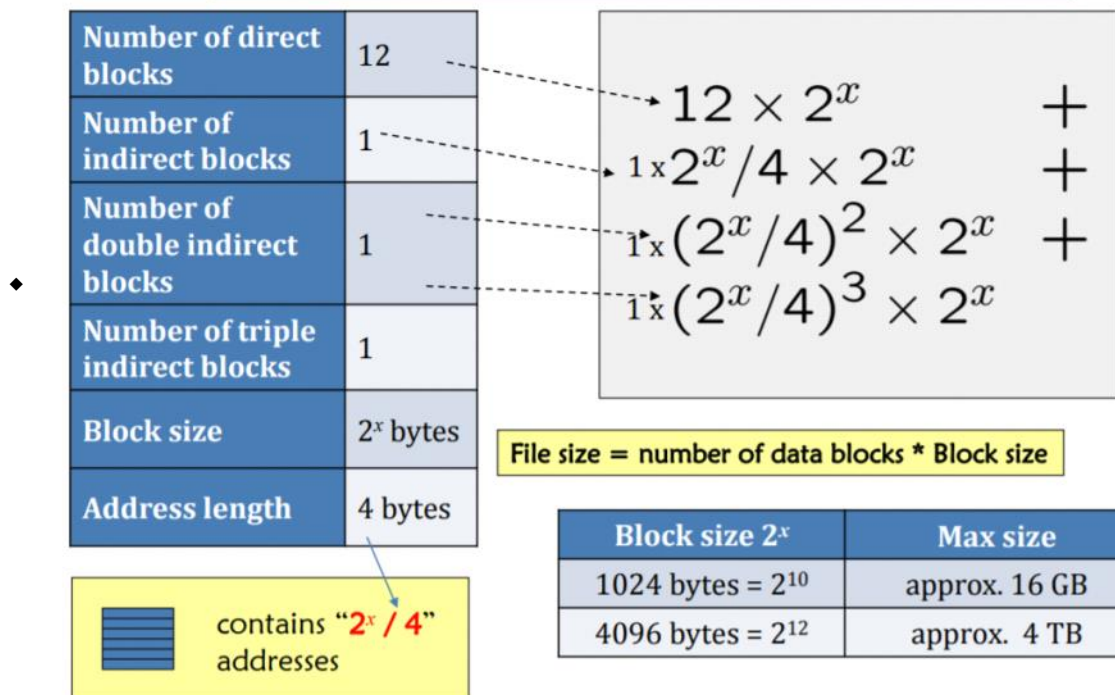
L12 文件系统-续

2020年5月20日 15:30

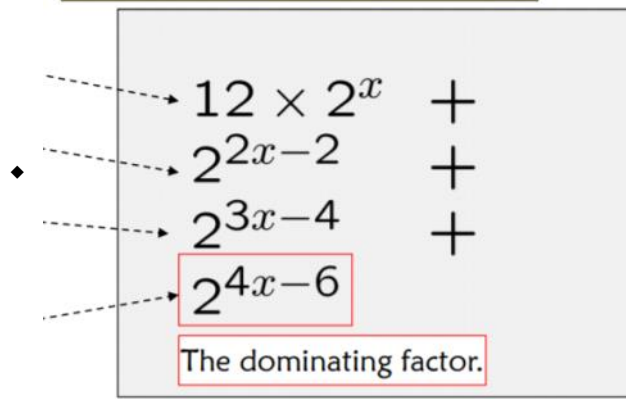
- Unix 的文件系统
 - Inode - BSD 引入
 - Inode array 记录了 file number
 - 用 inode 而不用 FAT 表
- Inode
 - 每个文件/目录一个 inode
 - 存储了 metadata 和 指向 data block 的指针
 - 指针可能是多层的
 - 小文件: 12个direct ptr (12*4KB blocks = 48KB)
 - 还有 indirect ptr, double indirect ptr, triple indirect ptr
 - 指向一个只有ptr的disk block, 4KB->1024ptr
 - 4MB @lv2, 4GB @lv3, 4TB @lv4
 - 计算最大文件大小
 - ◆ 1kb inode -> ~60/16GB??
 - ◆ Max file size: 单个文件的最大大小
 - ◆ FS size: 整个 FS 能容纳的容量

Index-node – file size

Reminder: Max file size != FS size



$$\text{File size} = \text{number of data blocks} \times 2^x$$



-
- Inode table
 - Array of inodes
- EXT 文件系统
 - 最新 ext4
 - Ext2, ext3
 - Block: block addr 4byte, 地址空间 2^{32}
 - 不同的block size, 支持的最大FS大小也不一样
 - 1024 block size \rightarrow 4TB
 - FS Size = 每个 block 的大小 * 所有 block 的数量
 - Max file size = 数direct block和所有的 indirect block
 - Block group
 - FS被分成多个 block group, 每个group的结构一样
 - ◆ 为什么要分成多个 block group: 性能和可靠性
 - ◇ 性能: 空间局部性, 把 inode和相关的data block放的很近
 - ◇ 可靠性: superblock, GDT 在每个 block group 中都有副本 (每个block group中的superblock,GDT都是一致的, 但是剩下的是每个group自己的)
 - 每个 block group 内的结构
 - ◆ Superblock
 - ◇ 标识FS, 存储本group内的block数
 - ◆ Group Descriptor Table (GDT), Reserved GDT
 - ◇ 存储bitmap, table的位置, free block/inode数
 - ◇ 为什么有 reserved GDT: 实际上block group里的GDT是当前fs内所有group的GDT的副本, 为了处理未来增长而保留
 - ◆ Block bitmap
 - ◆ Indoe bitmap
 - ◇ 一开始就固定好的bitmap, 说明Fs能存储的文件总数是固定的 (有上限的?)
 - ◆ Inode table: 根据inode号排序的inode array
 - ◆ Data blocks: 真正存储数据的Block
- Disk layout
 - disk被分为多个block group
 - 每个group有两个bitmap: block/inode bitmap
 - 在格式化的时候设定了 block的大小
- 目录 directory
 - 每个 inode总长度128byte
 - 文件大小的数值: 用64bit存(4-7, 108-111), 单位byte
 - ◆ 实际上能否达到, 要考虑 direct/indirect data block之和能否达到

- Link count用2byte 存
 - Directory entry
 - ◆ inode号, entry长度 (2byte) , filename长度 (1byte 最长255字符) , file type, ASCII文件名
 - ◇ 为什么同时存了 entry 长度和filename长度: entry 长度实际上是到下一个entry的offset, 在删除entry的时候, 会修改entry长度?????????
 - ◆ 每个directory实际上会占一个 (?) inode, 不过里面和一般的file的布局不一样, 存储的是一个 directory entry
 - ◇ 文件夹内的文件数量有上限? ? ? ? ?
 - ◇ 4 byte对齐? ? ? ?
 - 链接 link
 - 文件系统的link是DAG, 而不是一个简单的 tree
 - Hard link: 两个 dir entry指向同一个inode
 - ◆ 不会增加新的 inode
 - ◆ 会增加文件的 link count: 有多少个 dir entry指向这个inode
 - ◇ 子目录有 .. (实际上就是对父目录的hard link)
 - ◇ 当前目录下有 . (对自己的 hard link)
 - ◇ root下实际上还要加1 (root下的..依然指向自己)
 - ◇ 删除文件的时候的syscall是unlink, 把 link count -1, 如果 link count=0就说明没有指向文件的引用了, inode和data block可以被重新分配了
 - ▶ GC 引用计数: 如果成环了就再也删不掉了
 - ◆ 相当于文件的引用
 - Soft/symbolic link: 快捷方式
 - ◆ 创建一个新的 inode / 一个新的文件
 - ◆ inode的内容是目标文件的文件系统路径 pathname
 - ◇ 如果路径很短, 就直接用 direct + indirect block ptrs 存 ((12+3)*4=60 bytes, ASCII 1 byte/char -> 60 char)
 - ◇ 如果路径很长, 用一个 normal inode + 一个 direct data block: 为了省空间 (如果路径本身很短, 足够放在inode里面, 就不需要额外再分配一个data node了)
 - ◇ Soft link 的 path 有最大长度限制? ? ? ?
 - ◆ 如果源文件删除了, 那么soft link会失效
- NTFS
 - New Technology File System
 - 基础特性
 - 可变长 extent
 - 固定大小: FAT32-32byte, inode-128byte
 - Attr:value pair
 - 无论是数据还是元数据
 - 混合 direct/indirect
 - 用B树组织目录
 - 结构
 - Master File Table: 实际上是一个DB
 - 1KB最小块
 - 长度可变
 - extent: 可变长的连续区段
 - 带日志 journaling
 - 文件存储

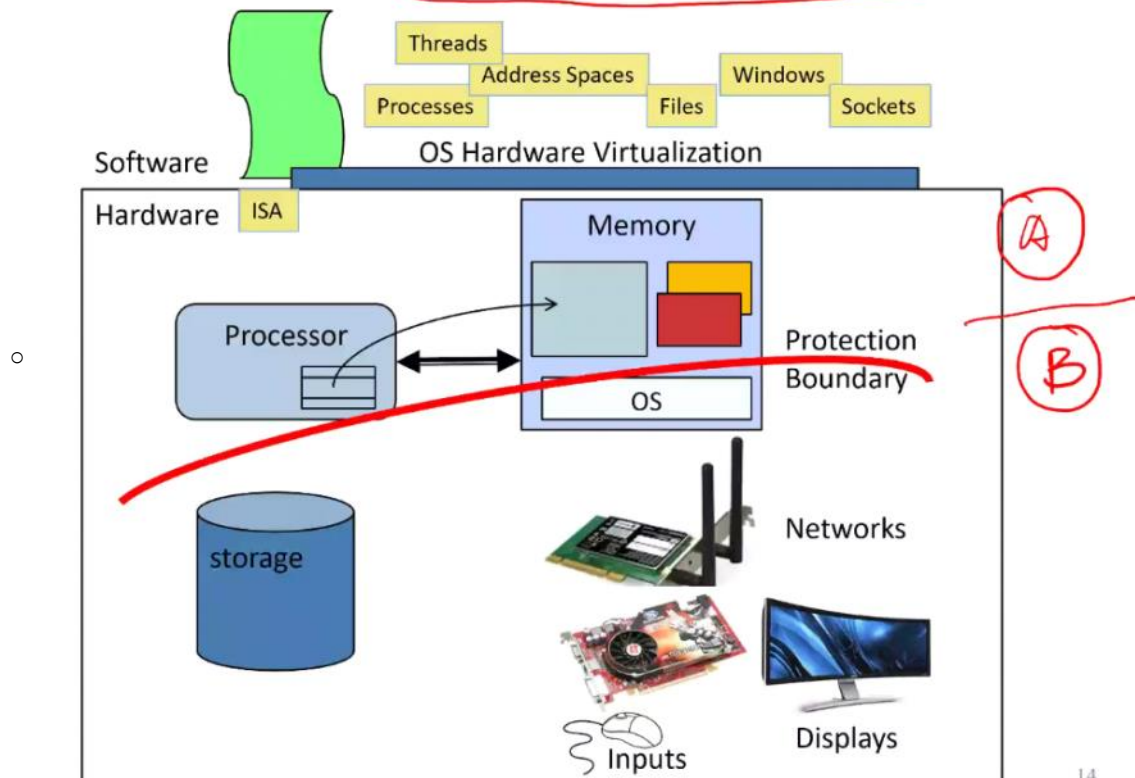
- 小文件：直接把数据存在 MFT 的 record 中的 data 段 (resident)
- 中等大小：record 中 data 段存储指向数据真正存在的 extent 的指针 (start+length)
- 大文件：和 inode 中 indirect 做法一样，用二层 extent 存储，最高层 attr list 中存储的是指向其他 mft record 的指针，中间层没有 attr 段，只有 data 段，存储指向 data extent 的指针，最底层是 data extent
- 超大文件：最多三层
 - 最大文件大小：几乎和 FS 大小一样（当前实现是 8PB）
- 内存映射文件 Memory Mapped Files
 - 之前要用 buffer，要多次操作 cache，多次 syscall
 - 现在可以直接把一个硬盘上的文件映射到内存，读的时候 page in，写完 page out
 - 可执行文件在执行的时候实际上就是这么做的
 - Mmap syscall
 - 可以直接像操作内存那样操作文件，会被自动同步回硬盘
- 文件系统总结
 - FS
 - Blocks -> file, dir
 - 为 size, access, usage pattern 优化
 - 最大化顺序读，但是随机读写的效率也要高
 - Header: inode/block
 - Naming: names -> resources
 - Dir
 - Link
 - Multi level index scheme
 - Inode -> data block ptr (indirect)
 - NTFS
 - Free space management
 - Memory map

OS 复习

2020年6月3日 14:00

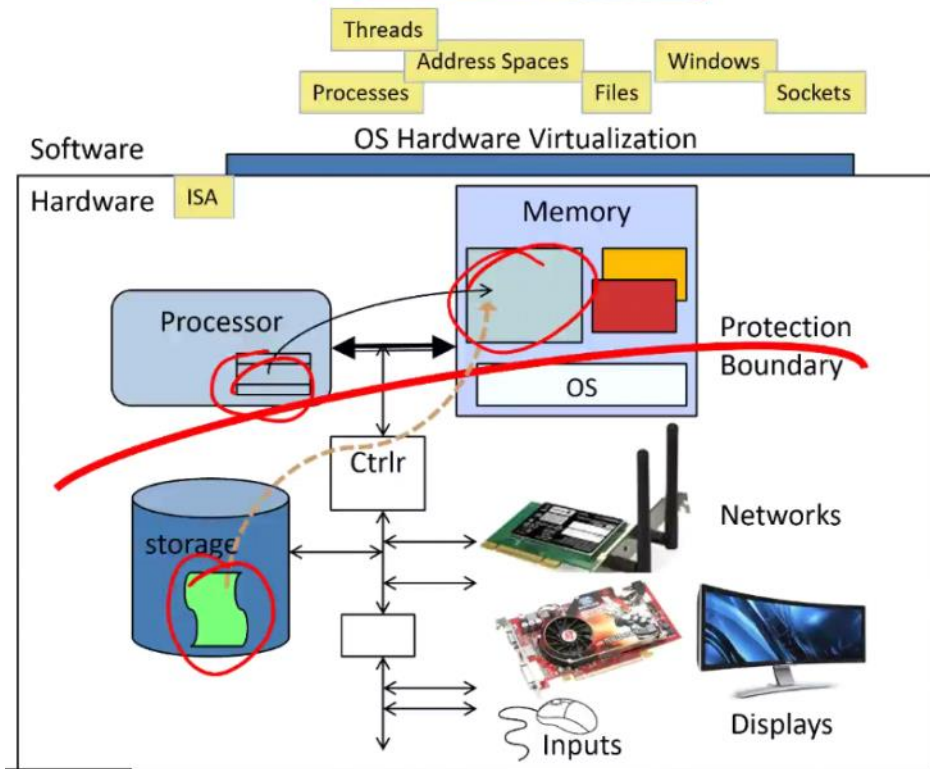
- OS 的功能 / 作用
- OS 基础: 资源抽象 / 虚拟化
- OS 基础: 程序和进程
- OS 基础: 上下文切换
- OS 基础: 调度和保护

OS basics: Scheduling, Protection



- OS 基础: IO
- OS 基础: 程序加载 loading

OS basics: loading



- 4个核心OS概念
 - 线程：最小的执行单元
 - 地址空间 (+ 地址翻译)
 - 翻译：使用 MMU
 - 进程：线程+地址空间
 - Dual mode operation / protection
 - Kernel + user mode/space
 - 通过控制地址翻译来实现程序之间的保护

Four Fundamental OS Concepts

Thread

- Single unique execution context: fully describes program state
- Program Counter, Registers, Execution Flags, Stack

Address space (with translation)

- Programs execute in an *address space* that is distinct from the memory space of the physical machine

Process

- An instance of an executing program is *a process consisting of an address space and one or more threads of control*

Dual mode operation / Protection

- Only the "system" has the ability to access certain resources
- The OS and the hardware are protected from user programs and user programs are isolated from one another by *controlling the translation* from program virtual addresses to machine physical addresses

• OS 概念1: thread of control

- Reg 中有 thread 的 context
 - Stack ptr, frame ptr, heap ptr, data
- Thread: single unique execution context
 - 有自己的: Pc, reg, execution flags, stack
- reg 中是 thread 的 root state, 其他的在内存中

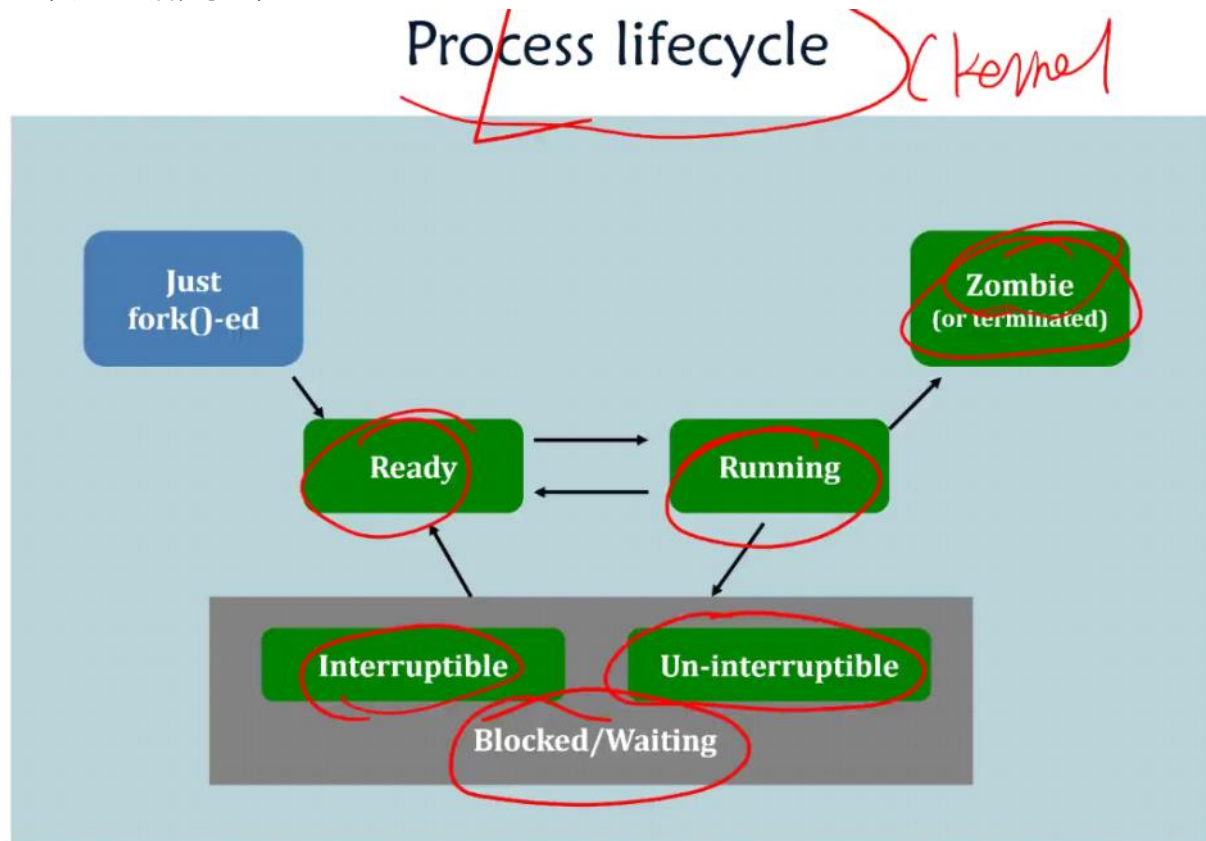
• OS 概念2: 地址空间

- 地址空间: 可寻址的地址 + 相关状态
 - 32bit: 2^{32}
- 读写地址的时候可能发生
 - Nothing
 - 正常内存读写
 - 忽略写入 (只读)
 - IO 操作 (memory-mapped IO)
 - Exception (page fault, seg fault)
 - 什么是 exception: 引发 trap, 进入 OS 规定的 exception handler 处理

• OS 概念3: 进程

- 权限受限的执行环境: execution env with restricted rights
 - 一个或多个线程的地址空间 (内存)
 - 文件描述符、FS context
 - 实际上是对线程之间共享资源的封装
- 为什么需要进程?
 - 进程互相隔离, 和 OS 隔离
 - 进程级别的抽象提供了内存保护
 - 线程比进程更 efficient
- Tradeoff: 保护 - 效率
 - 进程之间比线程之间更难
 - 进程间通信的方法: 6种? ? ?

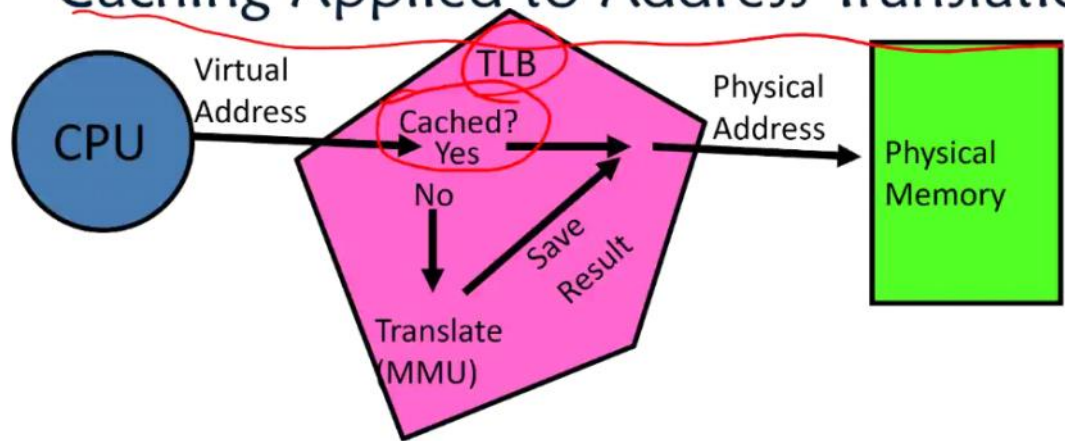
- 大类: Shared obj / message passing
- Signal <- OS 用的
 - ◆ Kill
 - ◆ 父进程等待子进程
- Pipe
- OS 概念4: dual mode operation
 - 至少提供两种模式: kernel / user
 - 进入/退出 kernel mode 都要 save/load PC
 - 进程在 user space 的三种状态
 - Running...?
 - Fork的实现过程
 - ??
 - [期末会考]
 - 进程的完整 lifecycle
 - Fork, ready, running, zombie/terminated, blocked/waiting ([un] interruptible)
 - 进程结束的时候会有一段时间 wondering? Hang up?
 - 父进程死了之后, 子进程过继?



- 上下文切换
- 进程调度
 - 核心问题: 计算资源 CPU 有限
 - 两类进程: CPU bound / IO bound
- 调度算法
 - Task: arrival time + requirement (实际上不实际)
 - Online / offline
 - SJF, FIFO, 抢占式
- Kernel 中的 pipe 实现
 - Ls | less
 - Pipe() syscall: 在 kernel sapce 建了一个共享FIFO queue

- Fixed size
- 同步问题
 - Producer-consumer
 - 可以用 信号量/mutex/cond val
 - Reader-writer
 - 哲学家就餐
- 死锁四条件/要素
 - 互斥
 - 持有并等待
 - 无抢占
 - 循环等待
- 解决死锁
 - Allocation graph
 - 银行家算法
 - 解除某个要素
- 从虚拟地址到物理地址
 - uniprogramming: 一次运行一个, 直接用物理内存
 - General address translation: 使用 MMU 实现
 - MMU memory management unit
 - translation简化了protection的实现
 - 程序可以被放入相同的虚拟地址? ? ? ?
 - Translation 的不同实现
 - 最简单: base & bound
 - 可以在运行的时候在每个load/store指令给地址+base
 - 每个程序有大块连续的物理内存
 - 问题: 碎片
 - 问题: 不支持稀疏地址空间 (code, data, stack 互相是分离的)
 - 问题: 难以实现进程间内存共享
 - ◆ 共享code segment, 共享特定的memory range
 - 更有弹性: 分段 segment
 - 每段独立分配连续内存, 有自己的base + limit
 - 问题: 每一段 varied size
 - 问题: 可能要移动进程多次才能把一个程序的所有segment都放入内存
 - 问题: swap的时候受限? ? ? ? ?
 - ◆ 移动的时候需要考虑移动哪些segment?
 - 问题: 内碎片? ? ?
 - 解决稀疏问题: 2层页表
 - 每个page大小相同
 - ◆ 小page - 小内碎片 - 需要更多page
 - 用 pagetableptr指向一层页表地址
 - 每个entry有valid bit
 - 2层页表也可以一起swap, 放到硬盘上
 - 地址翻译的 caching
 - TLB: Translation Lookaside Buffer

Caching Applied to Address Translation



- 是否真的存在局部性？
 - 连续访问的时候，很多时候会访问同一个page
 - stack肯定有局部性
 - data可能有局部性
- ? ? ? ? ?
- Demand paging
 - 问题：内存很大，用的没这么多（90-10）
 - 把所有的代码都放在内存里太浪费了
 - 可以把部分page放回硬盘，给其他程序腾空间
 - 实质：把内存作为硬盘的cache
 - 换页算法
 - FIFO
 - MIN / OPT
 - Random
 - LRU: Least Recently Used
 - ◆ [期末会考]
 - ◆ Insert, delete, lookup
 - ◆ 太贵了
 - 近似LRU
 - ◆ Clock
 - ◆ Second-chance
 - ◆ N-chance
- IO
 - 数据传输
 - programmed IO
 - 手动用指令操作
 - 简单，耗CPU时间
 - DMA: Direct Memory Access
 - IO 请求的lifecycle
 - 磁盘的结构
 - 时间计算
 - 磁盘调度
 - FIFO
 - SSTF
 - SCAN
 - LOOK
- 文件系统？
 - FAT
 - 读过程

- Dir entry
 - FS 设计要素
 - Inode
 - Inode array
 - Dir entry
 - Data block ptr, indirect ptr, doubly-indirect ptr
 - 最大文件大小和最大FS大小
 - mmap: 基于 paging 实现
 - 建立了对应的页表项, 但是实际上是对应到文件
 - 读写内存地址的时候会被映射回文件的读写
- 考试内容
 - 都在今天复习内容里

OS 期末复习 - 图形笔记

2020年8月29日 15:51

复习范围

问题记录

已解决

课件 x12

lab x?

期中试卷 x1

作业 x? report x9

pop quiz

信号的处理过程?

死锁的处理? 银行家算法?

多个程序共享同一个对象的时候, 什么算CS?

A's CS != B's CS

L6 进程间通信的方式?

Shared file

Pipe/FIFO

Shared memory

Semaphore

Shared file?

Message passing

Signal

Message queue

Socket

MPI

Dining philosopher – the final solution.

Shared object	Main function	<pre>void wait(semaphore *s) { disable_interrupt(); *s = *s - 1; if (*s < 0) { enable_interrupt(); sleep(); disable_interrupt(); } enable_interrupt(); }</pre>
<pre>#define N 5 #define LEFT ((i+N-1) % N) #define RIGHT ((i+1) % N) int state[N]; semaphore mutex = 1; semaphore p[N] = 0;</pre>	<pre>1 void philosopher(int i) { 2 think(); 3 take_chopsticks(i); 4 eat(); 5 put_chopsticks(i); 6 }</pre>	
Section entry	Section exit	<pre>void post(semaphore *s) { disable_interrupt(); *s = *s + 1; if (*s <= 0) wakeup(); enable_interrupt(); }</pre>
<pre>1 void take_chopsticks(int i) { 2 wait(&mutex); 3 state[i] = HUNGRY; 4 captain(i); 5 post(&mutex); 6 wait(&p[i]); 7 }</pre>	<pre>1 void put_chopsticks(int i) { 2 wait(&mutex); 3 state[i] = THINKING; 4 captain(LEFT); 5 captain(RIGHT); 6 post(&mutex); 7 }</pre>	
Extremely important helper function		
<pre>1 void captain(int i) { 2 if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) { 3 state[i] = EATING; 4 post(&p[i]); 5 } 6 }</pre>		

96

Producer-consumer problem: semaphore

Shared object

```
#define N 100
semaphore mutex = 1;
semaphore avail = N;
semaphore fill = 0;
```

Note

The size of the bounded buffer is “N”.

fill : number of occupied slots in buffer

avail: number of empty slots in buffer

Producer function

```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6         wait(&avail);
7         wait(&mutex);
8         insert_item(item);
9         post(&mutex);
10        post(&fill);
11    }
12 }
```

Consumer Function

```
1 void consumer(void) {
2     int item;
3
4     while(TRUE) {
5         wait(&fill);
6         wait(&mutex);
7         item = remove_item();
8         post(&mutex);
9         post(&avail);
10        //consume the item;
11    }
12 }
```

71

Semaphore **logical** view

```
typedef struct {
    int value;
    list process_id;
} semaphore;
```

Section Entry: sem_wait()

```
1 void sem_wait(semaphore *s) {
2     disable_interrupt();
3     *s = *s - 1;
4     if ( *s < 0 ) {
5         enable_interrupt();
6         sleep();
7         disable_interrupt();
8     }
9     enable_interrupt();
10 }
```

Initialize **s** = 1

“sem_wait(s)”

- I wait until I get **an s**
(i.e., **wait(s)** only returns when I get **an s**)
- Implementation:
of **s--**;
sleep if # of **s** < 0;

Important 1

s can be a plural

Important 2

This wait is different from parent's folk wait(child). When programming, it is sem_wait()

“sem_post(s)”

- I notify the others that one **s** is added
- Implementation:
of **s++**;
If someone is waiting **s**, wakeup one of them

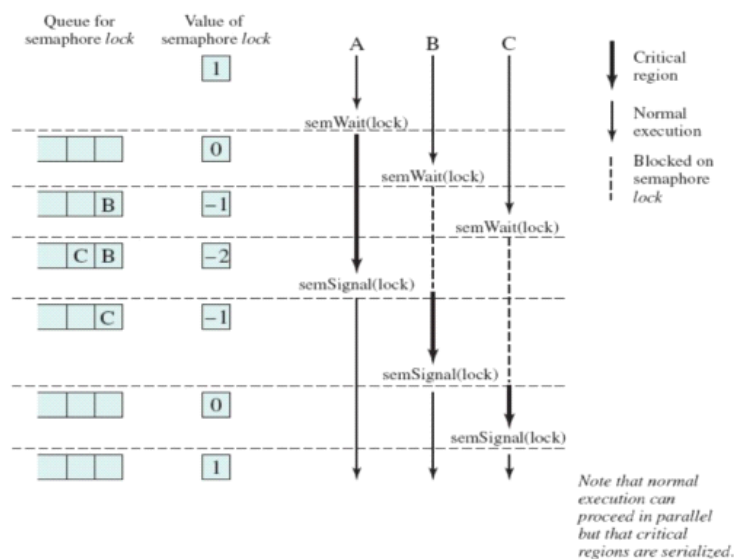
Section Exit: sem_post()

```
1 void sem_post(semaphore *s) {
2     disable_interrupt();
3     *s = *s + 1;
4     if ( *s <= 0 )
5         wakeup();
6     enable_interrupt();
7 }
```

#2: Spin Smarter (by Peterson's solution)

```
1  int turn;                                /* who is last enter cs */
2  int interested[2] = {FALSE,FALSE}; /* express interest to enter cs*/
3
4  void lock( int process ) { /* process is 0 or 1 */
5      int other;                /* number of the other process */
6      other = 1-process;        /* other is 1 or 0 */
7      interested[process] = TRUE; /* express interest */
8      turn = process;
9      while ( turn == process &&
              interested[other] == TRUE )
10         ; /* busy waiting */
11 }
12
13 void unlock( int process ) { /* process: who is leaving */
14     interested[process] = FALSE; /* I just left critical region */
15 }
```

Mutual Exclusion using Semaphores



<https://www.geeksforgeeks.org/readers-writers-problem-set-1-introduction-and-readers-preference-solution/>

```

do {
    // writer requests for critical section
    wait(wrt);

    // performs the write

    // leaves the critical section
    signal(wrt);
} while(true);

```

```

do {

    // Reader wants to enter the critical section
    wait(mutex);

    // The number of readers has now increased by 1
    readcnt++;

    // there is atleast one reader in the critical section
    // this ensure no writer can enter if there is even one reader
    // thus we give preference to readers here
    if (readcnt==1)
        wait(wrt);

    // other readers can enter while this current reader is inside
    // the critical section
    signal(mutex);

    // current reader performs reading here
    wait(mutex); // a reader wants to leave

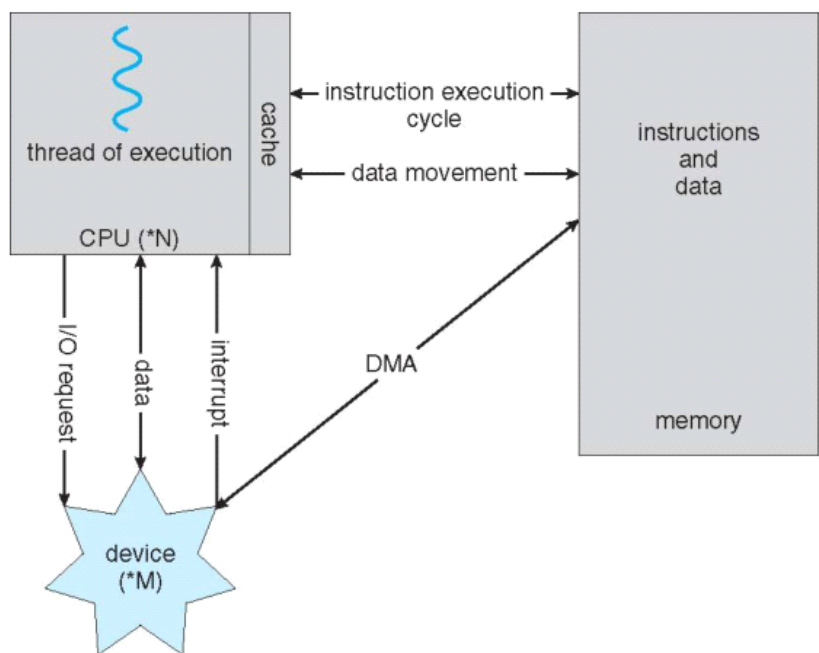
    readcnt--;

    // that is, no reader is left in the critical section,
    if (readcnt == 0)
        signal(wrt); // writers can enter

    signal(mutex); // reader leaves
} while(true);

```


How a Modern Computer Works




A von Neumann architecture

Index-node – file size

Reminder: Max file size != FS size

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes



contains " $2^x / 4$ " addresses

$$\begin{aligned}
 &12 \times 2^x && + \\
 &1 \times 2^x / 4 \times 2^x && + \\
 &1 \times (2^x / 4)^2 \times 2^x && + \\
 &1 \times (2^x / 4)^3 \times 2^x
 \end{aligned}$$

File size = number of data blocks * Block size

Block size 2^x	Max size
1024 bytes = 2^{10}	approx. 16 GB
4096 bytes = 2^{12}	approx. 4 TB

Index-node – file size

File size = number of data blocks x 2^x

Index-node – file size

File size = number of data blocks $\times 2^x$

Number of direct blocks	12
Number of indirect blocks	1
Number of double indirect blocks	1
Number of triple indirect blocks	1
Block size	2^x bytes
Address length	4 bytes



contains " $2^x / 4$ " addresses

$$\begin{array}{rcl}
 12 \times 2^x & + & \\
 2^{2x-2} & + & \\
 2^{3x-4} & + & \\
 2^{4x-6} & &
 \end{array}$$

The dominating factor.

Block size 2^x	Max size
1024 bytes = 2^{10}	approx. 16 GB
4096 bytes = 2^{12}	approx. 4 TB

Reminder: Max file size != FS size

OS indigrid 笔记 - 复习大纲

2020年9月2日 20:53

OS 复习

大架构

进程

syscall

用户视角、内核视角

生命周期、调度

信号

同步、死锁

内存

虚拟内存、地址翻译

缓存

请求分页

存储

I/O

文件系统

复习内容

课件 x12

lab x?

期中试卷 x1

作业 x? report x9

pop quiz

onenote 图形笔记

去看看 unix call youtube视频

考前确认

L5, schedule算法手动画图

L6, 模拟p-c, 哲学家就餐问题

lab7 银行家算法模拟

L8 P33 地址翻译模拟

L9 P23 缓存模拟

L10 P38 磁盘速度计算

L10 P52 disk scheduling 模拟

未解决问题列表

信号的处理过程?

多个程序共享同一个对象的时候, 什么算CS?

A's CS != B's CS: L6-P28

银行家算法处理哲学家就餐问题?

资源分配图是啥?

没有环路不会死锁, 出现环路可能死锁

如果只有一个资源实例，又出现了环路，就一定会死锁
资源分配图化简？

找一个只有分配边的进程节点（资源->进程）

找不到怎么办？

移除该节点和分配边，把资源分配给等待的进程

重复

如果能完全被化简，就没有死锁

程序从存储设备加载到内存的过程？

进程的调度方式？

缓存的替换方式？

进程和线程的区别？

为什么stack往下长 heap往上长

concurrency? parallel?

concurrency 并发，parallel 并行

“并发”指的是程序的结构，“并行”指的是程序运行时的状态

并行计算、并发编程

判断程序是否处于并行的状态，就看同一时刻是否有超过一个“工作单位”在运行就好了。所以，单线程永远无法达到并行状态。

正确的并发设计的标准是：使多个操作可以在重叠的时间段内进行

并发设计让并发执行成为可能，而并行是并发执行的一种模式。

<https://laike9m.com/blog/huan-zai-yi-huo-bing-fa-he-bing-xing,61/>

fork的 file, file locks?

file descriptor: process中的一个int

file description: kernel中描述file的一个struct, 包括当前offset

file lock: advisory 加在 file 上, mandatory 加在 inode 上

之前process lab上遇到的诡异问题是什么？bg/fg?

Threads in a Process? L05P57

程序编译/运行全过程？L08-P14

L8-P33 用到的 x86 汇编指令？

L9-P19 Effective Access Time? Avg Access Time?

$EAT = Hit\ Time + Miss\ Rate \times Miss\ Penalty$

L9-P28 LRU/MIN 下，小内存时cache的内容是大内存时的子集？

L9最后的页面部分：P33-P35？

L10-P4 Linux Memory Canonical Hole？

L10-P36 数值错误？

对比 ext2/3 和 fat？

L12-P4：VS. FAT: pointers of a file are

已经解决

system = fork + exec + wait

stack在上的内存布局被叫做什么：L1P29 segmentation

PCB存储在kernel中的数据结构：双向链表

wait, waitpid

wait: 任何一个children, 只管termination

waitpid: 指定某一个child, 可以监听多种状态改变

作用: 暂停parent、清理child

time 的计时问题, 什么时候多, 什么时候少

real 真实耗时, user 用户模式下CPU时间, sys kernel模式下CPU时间

user+sys只是CPU时间

这三个值都是包含子进程的(如果可能, 例如用了 wait/waitpid)

real < user+sys: 多线程+多核

real > user+sys: IO-intensive, 调度其他程序, mode切换overhead

syscall可能同时使用user和sys: 可能需要数据处理:

<https://stackoverflow.com/a/556411>

scheduling, context switching

schedule: 决定下一个要运行的进程

context switch: 真正的切换过程

hyperthreading

在OS层面上, 线程的实现本身就是软件上降低依赖性, 提升并发度的方法。

而“假装自己是两个处理器”是最简单的硬件解耦, 于是HT技术就这么产生了。

首先解码逻辑核心1的机器码, 送入执行单元, 等待结果的同时解码逻辑核心2的机器码, 如果出现的空闲的执行单元恰好是逻辑核心2的微码需要的, 就直接送入执行单元不必一定要等待逻辑核心1的代码完全执行完毕。这一来一去, 两个逻辑核心变得相对独立了。

<https://zhuanlan.zhihu.com/p/58448264>

Multiprocessing, Multiprogramming, Multithreading

Multiprocessing Multiple CPUs

Multiprogramming Multiple Jobs or Processes

Multithreading Multiple threads per Process

进程间通信的方式? 一共有几种?

Shared file

Pipe/FIFO

Shared memory

Semaphore

Shared file?

Message passing

Signal

Message queue

Socket

MPI

sem_close, sem_unlink

sem_close: close's a semaphore, this also done when a process exits. the semaphore still remains in the system.

sem_unlink: will be removed from the system only when the reference count reaches 0 (that is after all processes that have it open, call sem_close or are exited).

前者是语义上的, 后者是文件系统上的

binary semaphore, mutex

- Mutex is more about resource protection.
- Semaphore is more about resource assignment.

- Mutex can only be unlock by container.
- Semaphore can be assigned by anyone, including caller itself.
- Mutex lock will be released, if the holder is terminate.
- Semaphore will not add up, if the holder is terminate.

conditional variable

Cond = a condition + a mutex

使用时间： when If/else is unbalance

Reader/Writer 问题

W: semaphore wrt=1

writer: wait wrt, write, signal wrt

R: int readcnt=0, semaphore mutex=1

这个mutex只是锁readcnt的

wrt才是控制文件写入的

因为reader之间是不干扰的

reader

wait mutex, readcnt++, 如果是第一个就wait wrt阻止写入, 然后signal mutex

read

wait mutex, readcnt--, 如果没有剩下的reader就signal wrt允许写入, 然后signal mutex

奇怪的知识点列表

OS: kernel + driver, shell + util

存储 hierarchy: 容量、延迟、带宽、价格

OS的4个核心概念: thread, addr space (+trans), process, dual mode op/protection

thread: single unique execution context + state

process: execution instance of a program, execution environment with Restricted Rights, 有状态

protection: control translation

读写内存地址的可能结果: nothing, 正常, ignore, I/O操作, page fault, seg fault

触发context switch的因素: timer, I/O interrupt, signal, priority, voluntary yield, other

OS的保护目标: reliability, security, privacy, fairness

切换Kernel/User的3种方式: syscall, interrupt, trap/exception

fork的4变与5不变

不变 (copy) : pc/register, code, memory (copy on write), opened file

变: return value, PID, parent, running time, file locks

进程相关syscall实现

fork的实现过程: PCB复制、PCB修改、内存复制(可能CoW)、改返回值

exec实现: 清空内存 (userspace: local var + heap)、用新程序加载 (reset: global var, code, constant)、重设PC

exit实现: 1内核释放PCB、关闭opened files, 1a如果有children、过继给

init (改child parent_ptr, 改init child_list, 应用background job), 2user space memory释放, 3进程zombie、发送SIGCHLD给parent

wait实现: 1注册SIGCHLD handler、2接收SIGCHLD信号、3接受并移除信号, 把child完全从kernel-space移除、4删除handler, 返回PID

process 生命周期: created, ready, running, waiting/blocked([un]interruptable),

terminated/zombie

thread 生命周期: init, ready, running, waiting, finished

context switch 的开销

direct: 保存/加载register, 模式切换, 地址空间切换

indirect: CPU cache, buffer cache?, TLB miss

schedule算法: FIFO, Shortest-Job-First(Preemptive, NP), Round-Robin, Priority + Multiple Queue, Priority + Multi Queue/Scheduler

RC: 共享对象 多个进程 同时访问

CS实现: Mutual Exclusion, Bounded Waiting (饥饿), Progress (死锁)

死锁条件: Mutual Exclusion, Hold and Wait, no Preemption, Circular Wait

处理死锁策略: 允许死锁恢复、避免死锁、忽略

已经死锁了: 中止进程、抢占资源、回滚操作

避免死锁: 无限资源、禁止共享、禁止等待、一开始申请好全部、特定顺序获取

死锁的检测: 银行家算法

remain

如果分配是否超过max

是否可以安全结束

Memory Multiplexing的重点: Protection, Controlled Overlap, Translation

虚拟内存的实现方式: Base & Bound, Segmentation, Paging, Segment + Paging

缓存有用的前提: freq case 足够频繁, infreq case不太expensive

缓存有用的原因: spatial / temporal locality

cache的关注点: block size, organization, 如何查找, replacement policy, miss处理, write处理

organization: direct-mapped, set-associative, fully-associative, 见L9-P41

write处理: write-through, write back

page fault处理: 根据替换策略选择被替换的页、如果dirty写回、改PTE/TLB为invalid, 加载新页, 更新PTE并invalid旧TLB项、从错误位置恢复

belady anomaly: 增加cache size反倒会增加miss, 看size小是否是size大的子集, FIFO会, LRU/MIN不会

cache miss的种类: compulsory, capacity, conflict, policy

page replacement policy: FIFO, MIN, RANDOM, LRU, approx LRU(clock, n-th clock, second chance [FIFO+LRU])

page frame allocation: Equal, Proportional, Priority

IO设备的参数: 数据粒度(byte, block), 访问模式(seq, random), 传输方式(programmed IO, DMA)

CPU控制IO设备的方式 (连接方法): io instruction (in/out), memory-mapped IO (load/store)

CPU和IO设备之间数据传输的方式 (传输方法): programmed IO, Direct Memory Access

IO设备通知OS: interrupt, polling

IO评价标准: response time/latency, bandwidth/throughput, start up/overhead

硬盘的物理组成: sector, track, cylinder

硬盘读写时间: (queuing time, controller time,) seek time (移到柱面), rotational

latency (旋转到扇区), transfer time

SSD 无 seek/rotational time

比较SSD/HDD: pro: 延迟/吞吐, 移动部件, 读速度 con: 贵、asymmetric block write
perf、limited lifetime

Startup Cost for I/O: syscall overhead, OS processing, controller overhead, driver
startup, queuing

disk scheduling: FIFO, Shortest Seek Time First, SCAN, Circular-SCAN, LOOK, CLOOK

FS的组成部分: naming, disk management, protection, reliability

FS Layout: contiguous allocation, linked allocation, inode allocation

期末问题

fork的实现过程

哪些方法可以实现 mutual exclusion? 优缺点?

Filter Algorithm Peterson's_algorithm: 多进程 turn/interested

期末考试: 一定会考至少一个 IPC 问题

银行家算法

LRU

Thinking Time - chapter 11

```

chapter 11 object
# define N 5
# define LEFT ((i+N-1) % N)
# define RIGHT ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore p[N] = 0;

Section Entry
void take_chopsticks (int i) {
    wait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    post(&mutex);
    wait(&p[i]);
}
    
```

```

think()
take_chopsticks(i)
eat()
put_chopsticks(i)
    
```

```

Section Exit
void put_chopsticks (int i) {
    wait(&mutex);
    state[i] = THINKING;
    captain(LEFT);
    captain(RIGHT);
    post(&mutex);
}
    
```

```

semaphore
void init_semaphore (int i) {
    disable_interrupt();
    s = s - 1;
    if (s == 0) {
        enable_interrupt();
        sleep();
        disable_interrupt();
    }
    enable_interrupt();
}

void post (semaphore *s) {
    disable_interrupt();
    *s = *s + 1;
    if (*s == 0) {
        wakeup();
    }
    enable_interrupt();
}
    
```

```

Helper
void captain (int i) {
    if state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING {
        state[i] = EATING;
        post(&p[i]);
    }
}
    
```

```

typedef struct {
    int value;
    list process_id;
} semaphore;
    
```

Producer - Consumer

Shared Object

```

# define N 100
semaphore mutex = 1;
semaphore avail = N;
semaphore full = 0;

// full: occupied slot
// avail: empty slot
    
```

Producer

```

void producer (void) {
    int item;
    while (TRUE) {
        item = produce();
        wait(&avail);
        wait(&mutex);
        insert_item(item);
        post(&mutex);
        post(&full);
    }
}
    
```

Consumer

```

void consumer (void) {
    int item;
    while (TRUE) {
        wait(&full);
        wait(&mutex);
        item = get_item();
        remove();
        post(&mutex);
        post(&avail);
    }
}
    
```

using only avail
如果 consumer 的 get null
D 的 buffer
Buffer 满了, producer 的 avail 被
sleep, consumer 的 mutex 被
deadlock.

Spin smarter - (Peterson)

```

int turn;
int interested[2] = {FALSE, FALSE};
void lock(int process) {
    in order;
    other = 1 - process;
    interested[process] = TRUE;
    turn = process;
    while (turn == process &&
        interested[other] == TRUE) {
        ;
    }
}

void unlock (int process) {
    interested[process] = FALSE;
}
    
```

← 每个 P 一个 CS

```

P0    P1
lock
turn = 0
...
[CS]
    
```

```

lock
interested[1] = TRUE
turn = 1
while {
    // wait
}
[CS]
    
```

```

P0    P1    P1
lock
interested[0] = TRUE
turn = 0
    
```

```

turn = 1
while {
    no wait
}
[CS]
    
```

```

lock
interested[1] = TRUE
turn = 1
    
```

```

[CS]
unlock
interested[0] = FALSE
// wait done
[CS]
    
```

Riter Algo.

```

level[N] = {-1}
last-to-enter[N-1] = {-1}
    
```

lock: i = process no.
from 0 to N-1 exclusive.

while...
wait...
level[i] = -1
等待方向
等待 CS
等待所有
高同。

1. $turn = \text{互斥 Mutual Exclusion.}$
 $interent = \text{解决 strict alternation.}$
 2. $\text{priority inversion problem}$

lock: $\in \text{process no.}$
 for l from 0 to $N-1$ exclusive.
 $level[l] \leq l$
 $(last-to-enter[l] \leq i)$
 while $(last-to-enter[l] = i \text{ and } \text{exist } k \neq i, \text{ where } level[k] \geq l)$: 按顺序
 wait
 1. 若没有别的线程在要访问临界区
 2. 则的线程进入我的房间. 为 $wait$, 被阻塞

unlock: $level[i] = -1$ 唤醒所有等待.

Shared Object.
 Semaphore: $mutex = 1$
 semaphore: $write \rightarrow 1$
 int read count $\rightarrow 1$

lock: $\text{write}()$
 // WRITE
 $\text{post}()$ (write)
 while (true).

lock: $\text{read}()$
 $\text{readcount}++$
 if $(\text{readcount} == 1)$ $\text{wait}()$ (mutex);
 $\text{post}()$ (mutex);
 // READ
 $\text{wait}()$ (mutex);
 $\text{readcount}--$
 if $(\text{readcount} == 0)$ $\text{post}()$ (mutex);

Race condition:
 shared object.
 Multiple Process/Thread.
 Access concurrently.
 After Deadlock Attitude:
 Allow Recover.
 Prevent Ignore.

Critical Section Check.
 Mutual Exclusion.
 Bounded Waiting (starvation)
 Progress (Deadlock)
 After Deadlock:
 Terminate Process.
 Preempt w/o killing.
 Rollback action

Dead Lock:
 Mutual Exclusion.
 Hold And wait
 No preemption.
 Circular wait

Prevent Deadlock:
 Infinite Resource.
 No sharing
 NO priority: allow Preempt.
 Request All need out seging.
 Force Request Order.

Achieve MR:
 Lock: Spin-based
 Lock-free: Sleep-based.
 CAS: versioning.

#0: Disable Interrupt for
 Mode $\neq CS$.
 uncore: correct but not
 permissible.
 (user x kernel V)
 multicores: incorrect.
 (process on other core).
 may still change
 shared object.

#1: Basic Lock.
 $\text{while}(\text{True})$
 $\text{while}(\text{turn} == 0)$:
 CS
 $\text{turn} = 1$
 $\text{remainder}()$
 correct.
 OK for short
 spin < context
 switch.
 strict alternation.

#2: Spin Snafter.
 correct.
 priority inversion problem.
 #3: Semaphore.
 need interact with scheduler
 unturntable section entry/exit.
 used as ~~mutex~~ synchronization
 primitive.

Bauker Algo
 $[Avail] = [FreeResources]$
 RM node $\text{cod} \rightarrow \text{unfinishe}$.
 do {
 $\text{done} = \text{true}$.
 foreach node in unfinished {
 if $\frac{[MAX] - [thoc]}{\text{node}} \leq [Avail]$ {
 remove node from UNFINISHED.
 $[Avail] = [Avail] + [thoc]$
 $\text{done} = \text{false}$.
 IO ctrl: IO instruction, non-mapped IO.
 IO trans: Programmed IO, DMA.
 HDD: sector, track, cylinder.
 Disk Time: Queue, ctrl, seek Time.
 rotational delay, Transfer.

FORK:
 1. copy PCB
 2. kernel space update.
 parent - Add new child.
 child - update PZD, runj time.
 ptr to parent.
 3. user space update.
 copy memory context
 (COW).
 4. finish.
 update parent/child return vale.

Don't change (v)
 PC/Register, Code, Memory,
 Open File
 Change (v)
 return vale.
 PZD, parent.
 runj time.
 file codes.

File size
 $12 \cdot 2^x + 1 \cdot 2^x \cdot 2^x + 1 \cdot \left(\frac{2^x}{4}\right)^2 \cdot 2^x + 1 \cdot \left(\frac{2^x}{4}\right)^3 \cdot 2^x$
 $= 12 \cdot 2^x + 2^{2x-2} + 2^{3x-4} + 2^{3x-6}$
 2^x : block size.

4 Fundamental OS:
 Thread
 Address Space.
 Process.
 Dual mode / Protection
 feat \square User+Sys
 <: multi-process.
 multi-core.
 >: IO-intensive.
 Scheduling.
 overhead.

ZPC:
 Shared File:
 Pipe / FIFO
 Shared Pipe.
 Shared File
 Shared memory
 Semaphore.
 Binary semaphore
 semaphore: resource
 protection
 By container
 By anyone.
 will not
 add up.
 Message Pass:
 Signal.
 Message Queue.
 Select
 API Library.

2x: stack size.

Parallel/Concurrency

Concurrency: structure. Sequence.

Parallel: running status. Hardware. multiple units.

Thread/Process

Process

Program in execution

Don't share memory

More context switch. Less.

Execution environment with restricted right.

instance of a program

Address space ✓ have its x don't.

Thread

Single unique.

Part of progress

share memory.

Less.

Execution context

+ state.

Multi-core.

> IO-intensive.

Scheduling.

Mode. Change Overhead.

Life Cycle

Init → Ready → Run → Finished

Wait.

Just Forked.

Ready

Run

Wait

Blocked.

Terminated

Zombie

Save of CS

Thread: TCB, reg, PC, stack ptr.

Process: +

Used Thread

Same task.

different data.

different task.

independent task (CPU/D).

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

different

</