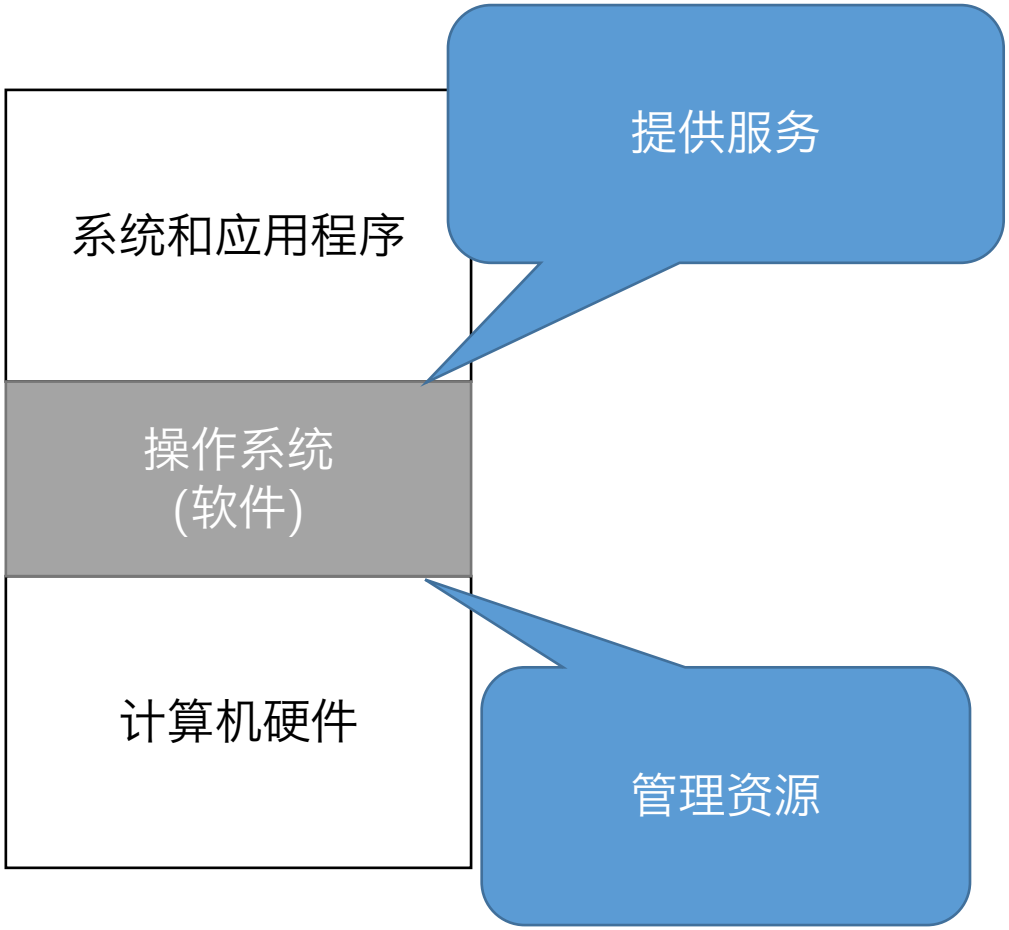
操作系统中的资源分配和调度

蒋炎岩

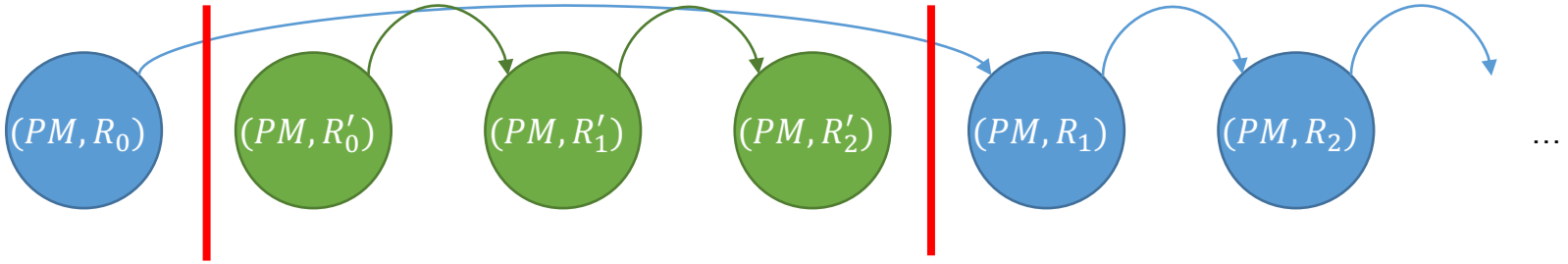
南京大学 | 计算机软件研究所 | 系统与软件分析研究组

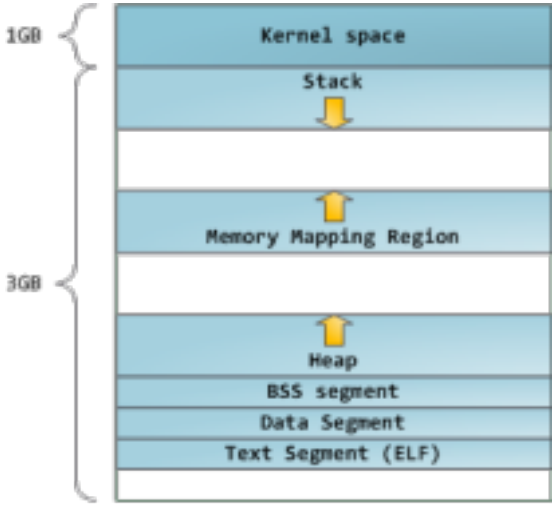
操作系统回顾

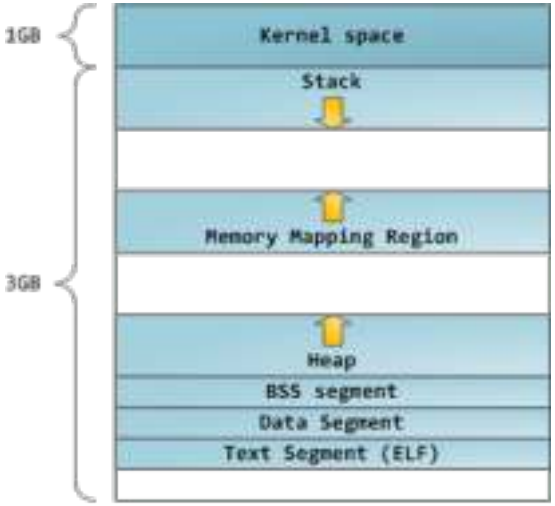


操作系统回顾：虚拟化

- 处理器虚拟化(进程)、内存虚拟化(虚存)



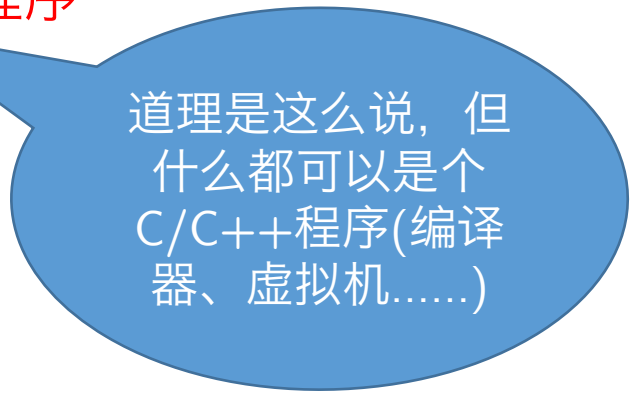




操作系统：到底是什么？

- 另一个视角：
  - 操作系统不过是一个C (或任何语言)程序

- 软件视角的操作系统
  - 从main开始运行
  - 程序执行的结果符合C语言的语义
  - 与硬件合作实现管理应用程序的功能



处理器管理

处理器管理的机制

- 操作系统是个C程序，使用硬件提供的机制
  - 进程：运行的程序；在中断/系统调用时进入操作系统执行
  - 操作系统能决定下一个运行的进程

- 但操作系统怎样更好地管理资源、提供服务?
  - 操作系统中的资源管理问题
  - 对可以分割的资源：如何分配?
  - 对共享的资源：怎样调度?

处理器调度策略：一个处理器、多个程序

- 我们只有一个处理器
  - 但有很多程序要运行
  - 按照什么顺序运行它们?

- 最简单的情况：程序执行到完成，并已知程序执行时间
  - 如果已知任务的时间，怎么“更好”地安排它们?
  - 如果任务并不是一开始就知道，而是随着运行时到达呢?

分时系统中的处理器调度

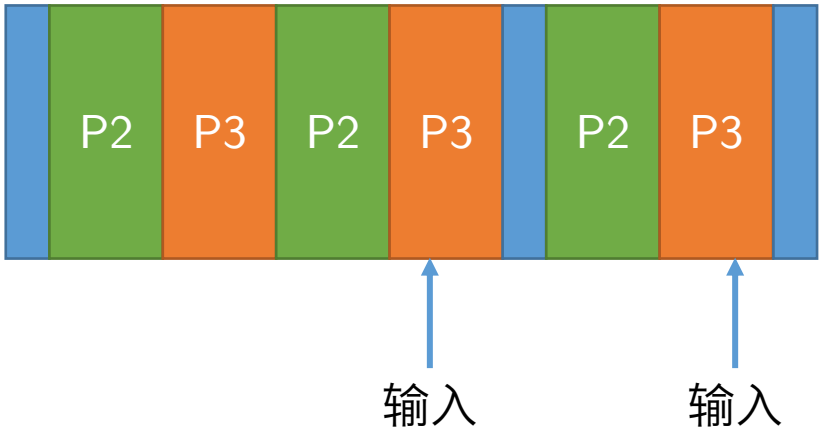
- 分时多任务
  - 让大家分时共享处理器就好了
  - 看起来很公平

- 缺陷：
  - 隐含假设：P1, P2, P3都只使用CPU
  - 而且对短任务不友好：本来10ms就能运行完，硬是拖到了30ms

对短任务不友好：实际情况

- 程序可能等待
  - 等待I/O设备(磁盘、网络、……)
  - 等待另一个进程(wait)
  - 等待其他数据/输入(管道)

- 这就感觉P1吃亏了(想象P1是vim，人打字速度很慢)
  - P2, P3霸占着CPU



一个自然的想法

- 在系统中引入优先级的概念
  - 优先级高的先运行；优先级一样的round-robin
  - 根据进程使用CPU的历史计算它的优先级
    - 用CPU的人(坏人)优先级会变低
    - 不用CPU的人(好人)优先级会变高

- MLFQ (多级反馈队列)
  - 有多个队列(代表优先级)，每个都是round-robin
  - 根据程序运行的情况调整优先级

如何奖励不断放弃处理器的程序?

- 维护每个进程的运行时间$T(P)$
  - 每当中断/系统调用时更新统计$T(P_1) = T(P_1) + \delta$
  - 引入优先级：$\delta = \Delta_t$/优先级 (高优先级进程时间过得慢)
- 公平调度
  - 每次都选$T(P)$最小(运行得最少)的进程执行
  - 如果进程长时间睡眠唤醒后，重置时间为$\min_P T(P) - t$
    - 问题：不这么设置有什么危险?
- 这就是Linux中的CFS (Complete Fair Scheduler)
  - 问题：如何实现CFS?

进程获得CPU的调控

- 好人：愿意把CPU让给别人的人
  - nice值越高，人越好，优先级越低
  - nice命令能设置优先级(man nice)
  - Linux: -20(极坏)到19(极好)

- 例子： nice = 0 vs. nice = 10

```
top - 11:39:23 up  3:33,  4 users,  load average: 1.25, 0.93, 0.62
Tasks: 159 total,   4 running, 155 sleeping,   0 stopped,   0 zombie
%Cpu(s): 88.1 us,  0.3 sy,  9.5 ni,  0.0 id,  0.0 wa,  0.0 hi,  2.0 si,  0.0 st
KiB Mem:   1018256 total,   794236 used,   224020 free,        0 buffers
KiB Swap:  2129916 total,    15632 used,  2114284 free.   185252 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 6919 gary      20   0    9208   2540    492 R  90.0  0.2   0:17.12 matho-primes
 6918 gary      30  10    9208   2540    500 R   9.9  0.2   0:01.83 matho-primes
 6864 root      20   0       0      0      0 S   0.0  0.0   0:00.17 kworker/0:1
    1 root      20   0   52848   4856   2860 S   0.0  0.5   0:04.58 systemd
    2 root      20   0       0      0      0 S   0.0  0.0   0:00.01 kthreadd
```

调度：远远没有解决的问题

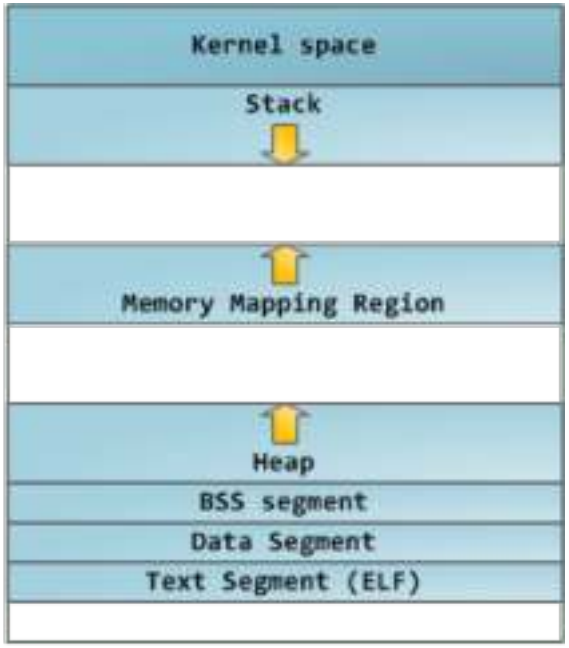
- 如果你有1,000,000台服务器运行1,000,000,000个任务
  - 有些是后台、长时间、批处理、延迟不敏感的任务
  - 有些是延迟非常敏感的任务(购物车刷不出来你就会骂)
  - 所有任务都会影响CPU、缓存、内存、网络、磁盘……
    - 希望一台物理计算机尽可能所有资源都被充分利用(少用10%的的机器就省下1亿美元的硬件成本)
    - 操作系统中网络、磁盘、CPU的调度相对独立(对特定的负载难免照顾不周)
    - 在同一个处理器上调度内存密集任务可能影响高优先级任务的缓存(即便优先级很高，性能也可能受到很大影响)
    - ……

虚拟存储管理

虚拟存储管理：机制

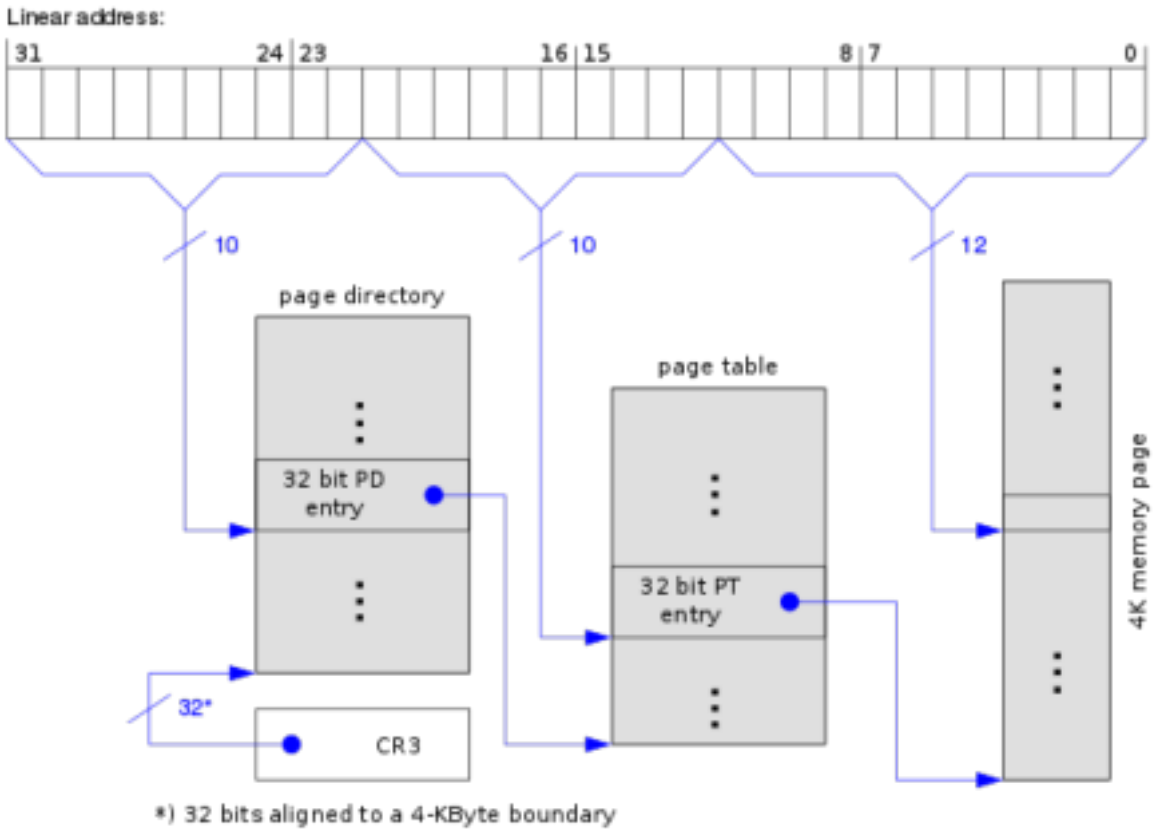
- 就是个映射：$VM(x)$
  - 操作系统能随时设置这个映射(把一个物理页映射到虚拟页)
  - 应用程序运行时，无法观察/修改这个映射，只能访问虚拟内存
  - 在访问错误发生时以异常的形式告知操作系统

- 让进程看到独占物理内存的假像



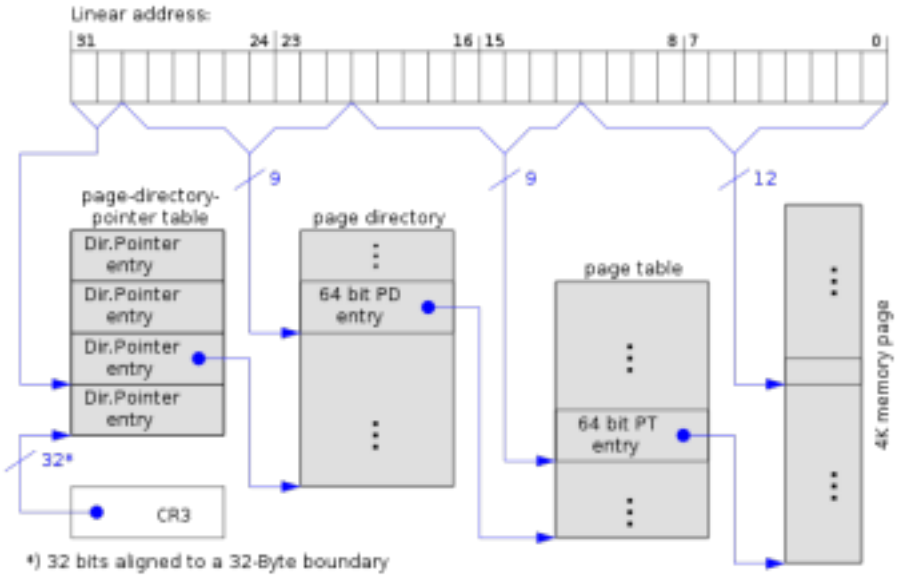
机制：$VM(x)$的实现 – i386

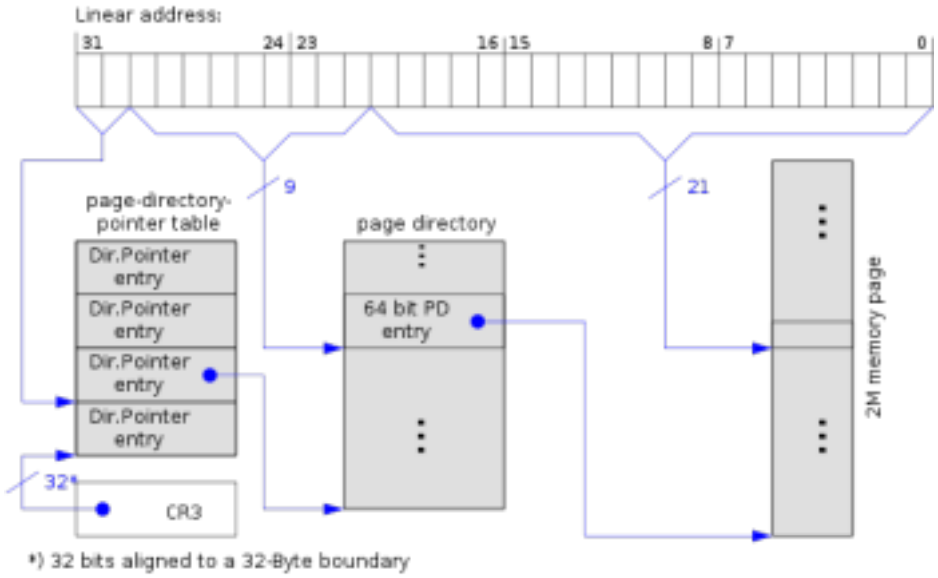
- i386 – 地址转换



机制：$VM(x)$的实现 - PAE

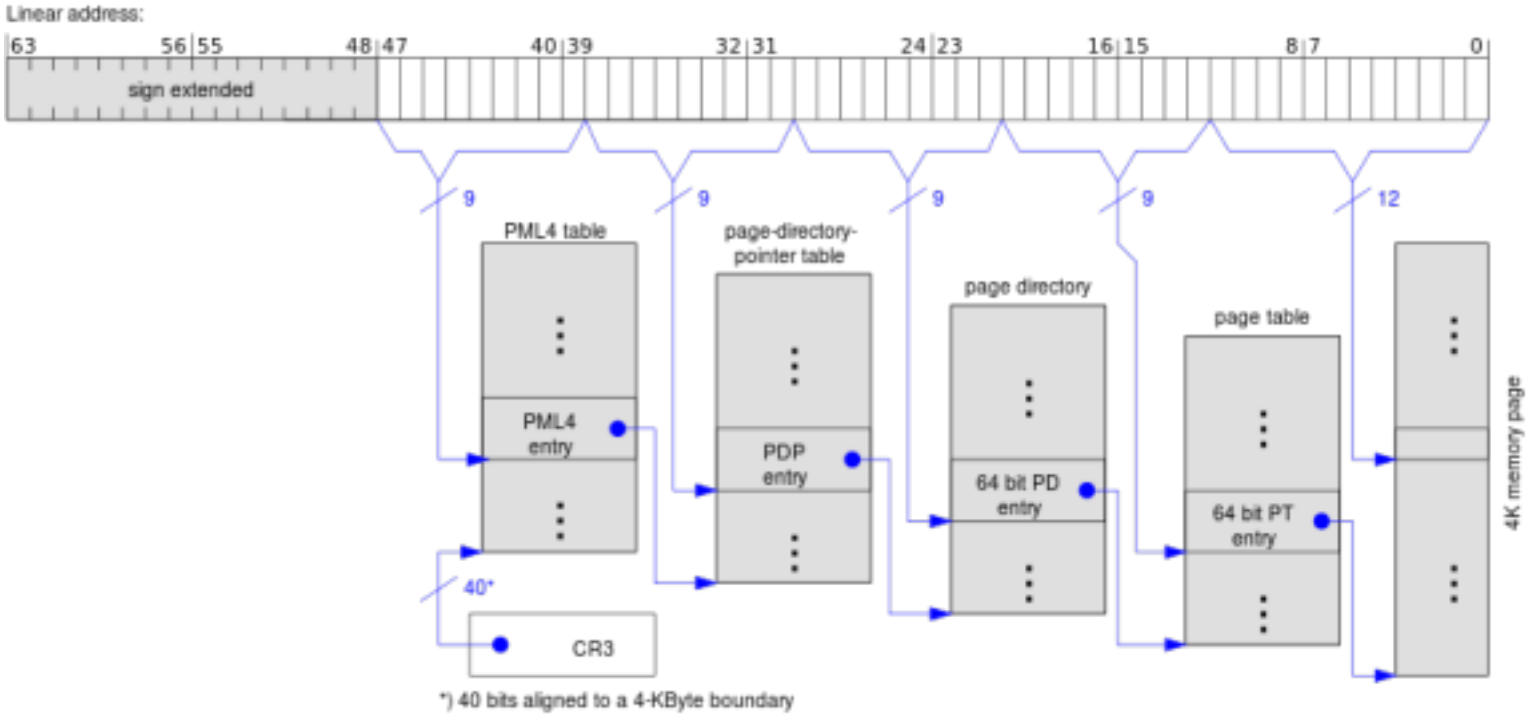
- 没什么，就是RTFM





机制：$VM(x)$的实现 – PML4 (256TB)

- PML4 (page map level 4) @ x86-64 (48bit)
  - 内存更大就需要PML5甚至PML6



(有趣的)机制：$VM(x)$的访问权限

- $VM(x)$还兼顾存储保护
  - 不让用户进程访问(读/写)不该看到的数据

- 可以为每个页目录、页表项设置访问权限
  - P = 0: 禁止访问
  - W = 0: 禁止写
  - K = 1: 禁止低权限运行模式访问

- 硬件会维护页表的信息
  - M = 1: 页被写过
  - A = 1: 页被访问过

感觉可以用来做好玩的事情

VM Tricks：机制与策略

懒(lazy)加载

- 我定义了一个64G的数组，但未必用它
  - 我只要在操作系统中记住这64GB都是合法的内存(记录内存映射)
  - 但在$VM(x)$中故意不分配它们
  - 等到访问缺失的时候再分配也不迟啊
    - 以下程序运行时间：0.00s user 0.00s system 0% cpu 0.002 total

```
 3 #define N (1L << 34)
 4 int a[N]; // 64GB
 5
 6 int main() {
 7   printf("sizeof of a: %ld MB\n", sizeof(a) >> 20);
 8   a[N - 1] = 1;
 9   a[0] = 2;
10   printf("sum = %d\n", a[0] + a[N - 1]);
11 }
```

做得更彻底一些

- 巧媳妇难为无米之炊
  - 我只有4GB内存
  - 怎么算出那么多数字来?

```
1 static uint32_t a[1L << 32]; // 16GB
2 a[0] = 1;
3 for (int i = 0; i < (1L << 32); i++) {
4   a[i] = (a[i-1] + a[i-2]) % 1000000007;
5 }
```

交换/请页调度(Demand Paging)

- 如果我有1,000,000,000GB的SSD磁盘，程序能在磁盘上运行就好了
  - $VM(x)$：当内存空间不足时，把虚拟地址空间中映射的一页保存到磁盘上，然后那一页就又变成自由空间了
    - 在x86的分页机制中，这是如何实现的?
- Demand Paging：机制
  - 在任何时候，操作系统都维护了很多$VM(x)$
  - 操作系统执行时，能任意选择一些映射的页保存到其他设备以缓解内存压力
    - 磁盘、网络、任何地方!

请页调度：策略

- 把一页换走需要代价
  - 决定换哪一页需要时间；换出一页需要磁盘/网络带宽
  - (尴尬)如果刚换出去的那一页下一条指令就访问了……
- 请页调度策略
  - 何时换出一页?
  - 换出哪一页?
- 问题的本质：预测未来的访问模式
  - 计算机系统里非常多见这个问题：分支预测, cache, TLB, …

程序执行的局部性

- 程序执行有 “惯性”

```
1 static uint32_t a[1L << 32]; // 16GB
2 a[0] = 1;
3 for (int i = 0; i < (1L << 32); i++) {
4   a[i] = (a[i-1] + a[i-2]) % 1000000007;
5 }
```

- 时空局部性
  - 某个资源如果被访问，未来有可能再被访问(i)
  - 某个资源如果被访问，附近的东西也可能被访问(a)
  - 推论：如果某个东西很久没访问，未来可能也不访问了(LRU)

- 如果没有局部性……
  - 分页都不work了(为什么？)

LRU：我是最优的吗?

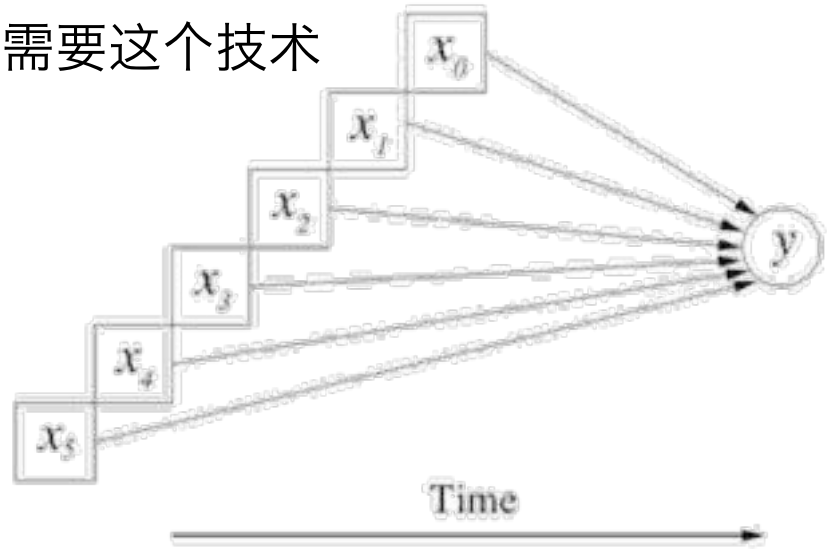
- 访问顺序：
  - 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4
  - 但同时只能容纳3个物理页
    - (1) {1, 2, 3} [4无法容纳，LRU] → {2, 3, 4}
    - (2) {2, 3, 4} [1无法容纳, LRU] → {3, 4, 1}
    - (3) {3, 4, 1} [2无法容纳, LRU] → {4, 1, 2}
    - … (聪明反被聪明误)

- 问题求解习题：给定访问顺序和同时能容纳的页数k，问最少交换的次数
  - 在上面的例子里，把{1, 2}始终留下是最好的

答案：最优替换

- 当必须换出一页时，总是换出未来最远被访问的那个

- 所以我们需要不断对程序的执行进行训练
  - 预测一个资源下次访问的时间
  - 分值预测、TLB、Cache更新需要这个技术

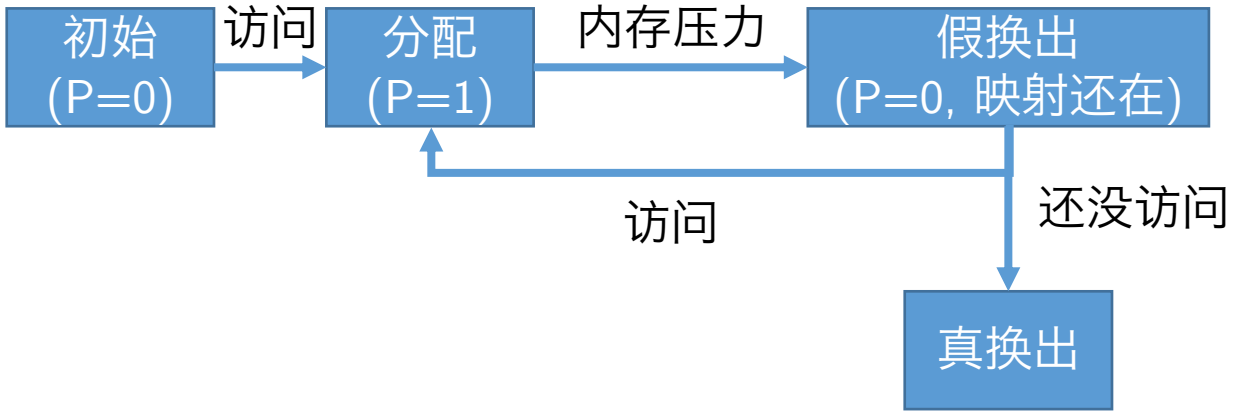


Fast Path-Based Neural Branch Prediction (MICRO03)

实现策略：残酷的现实

- 记录页面的访问是代价很大的
  - 每个页面凭空多出来那么些个信息，存在哪里？谁来更新？
  - M/A bit可以用来快速找到一段时间里没访问过的页面，这已经不错了

- 想得到更多信息：trick访问控制(但是有代价)



VM Tricks：写时复制

fork的难题

- fork有两个场景
  - 真的拷贝了一份：算一个素数表，然后让100个子进程接着玩
  - 假的拷贝了一份：fork-execve，华丽的变身
  - 同时要维持fork的语义：拷贝得到两个独立的进程
    - 一个进程随便往内存里写什么，对另一个进程都没有影响

- 原则上：我们需要把内存全拷贝一份
  - 否则如果父进程先写了内存，fork语义就错了

```
1 int x = 0, pid = fork();
2 if (pid == 0) {
3   assert(x == 0);
4 } else {
5   x = 1;
6 }
```

写时复制的机制

- 稍微动一下已经生锈的脑子……
  - 如果父/子之中有一个不再访问x，那就不用拷贝了
  - 但只要其中有一个写了x，这一页就必须拷贝
    - 因为不能预测另一个进程还会不会访问了

- 有了很直观的办法：等到写的时候再拷贝嘛 (copy-on-write)
  - fork以后，得到两个进程，共享一个$VM(x)$
  - 但所有的内存都只读不能写(完美解决素数表)
  - 如果写了共享的页面，就复制一份(维持fork语义必须)

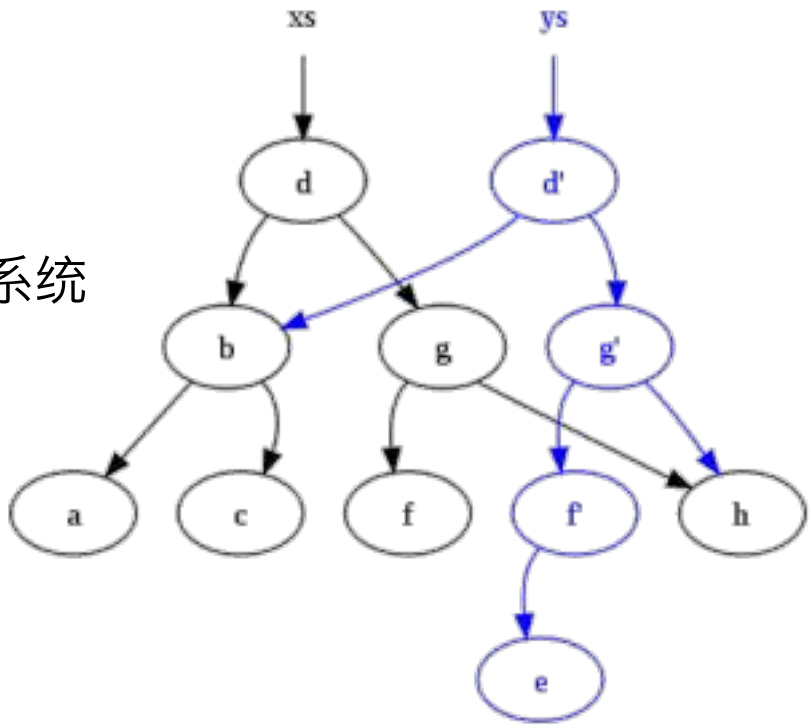
写时复制：策略

- fork
  - 子进程一开始没有映射任何页
  - 父进程可以任意读；子进程读异常、更新共享引用计数
  - 写页面引发异常，就拷贝一份单独映射，引用计数-1
  - 引用计数回到1时变回可写

- fork-execve
  - 让子进程优先级比父进程略高一些
  - 子进程会先执行少量代码后execve
    - execve会释放所有页面(引用计数-1)
    - 父进程写时那一页就可以写了

Copy-on-Write

- Copy-on-Write还是实现Functional Data Structure的重要方法
- 在计算机系统中的应用
  - 并发数据结构
  - 数据库
  - 支持快照的文件系统(XFS, btrfs)
  - 在单次写入的CD里实现读/写文件系统



为什么系统难实现?

- 如果我们只是管理一个$VM(x)$，那是相当好办了
- 但系统中有很多进程，很多进程会共享页面……
  - 比如libc的代码就只有一份
- 带来的麻烦
  - 如果访问P1的一页，P2页表的access bit是否会被更新?
  - 如果有多个进程共享同一个页面，它该不该被换出去呢?
    - 换出去将会引起很多页表的修改，还会并发……
- 所以操作系统理论大多是bullshit，大家还是要靠实践