



虚存和地址转换

中国科学院大学计算机与控制学院
中国科学院计算技术研究所

2021-10-25





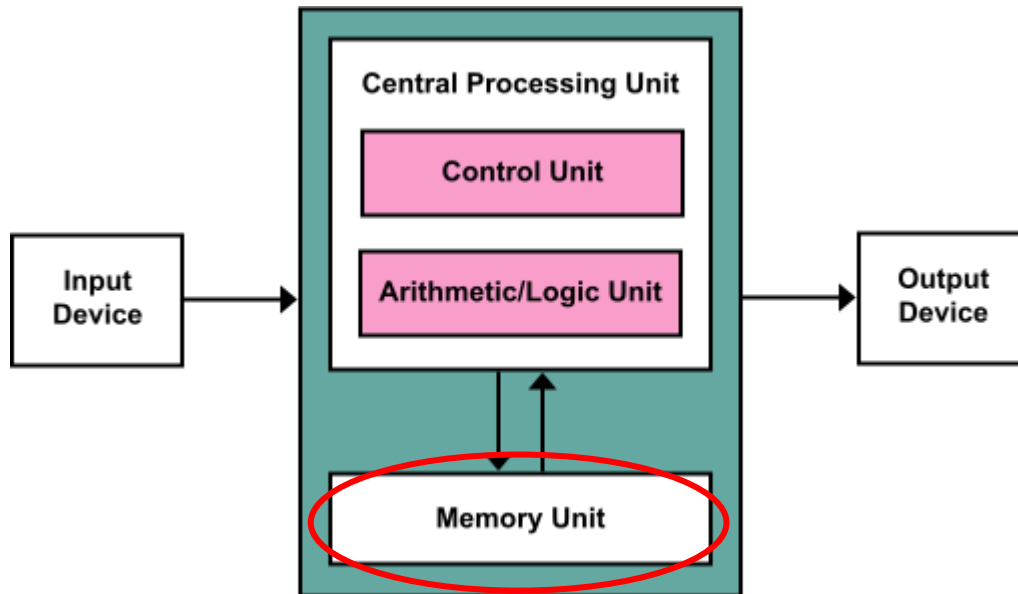
内容提要

- 计算机存储体系结构
- 虚存
 - 虚拟化
 - 保护
- 地址映射
 - 基址+长度
 - 分段
 - 分页
- MMU和TLB



现有计算机体系结构

- 冯·诺伊曼结构



《天才的拓荒者——冯·诺依曼传》
[美] 麦克雷，上海科技出版社
2008年12月

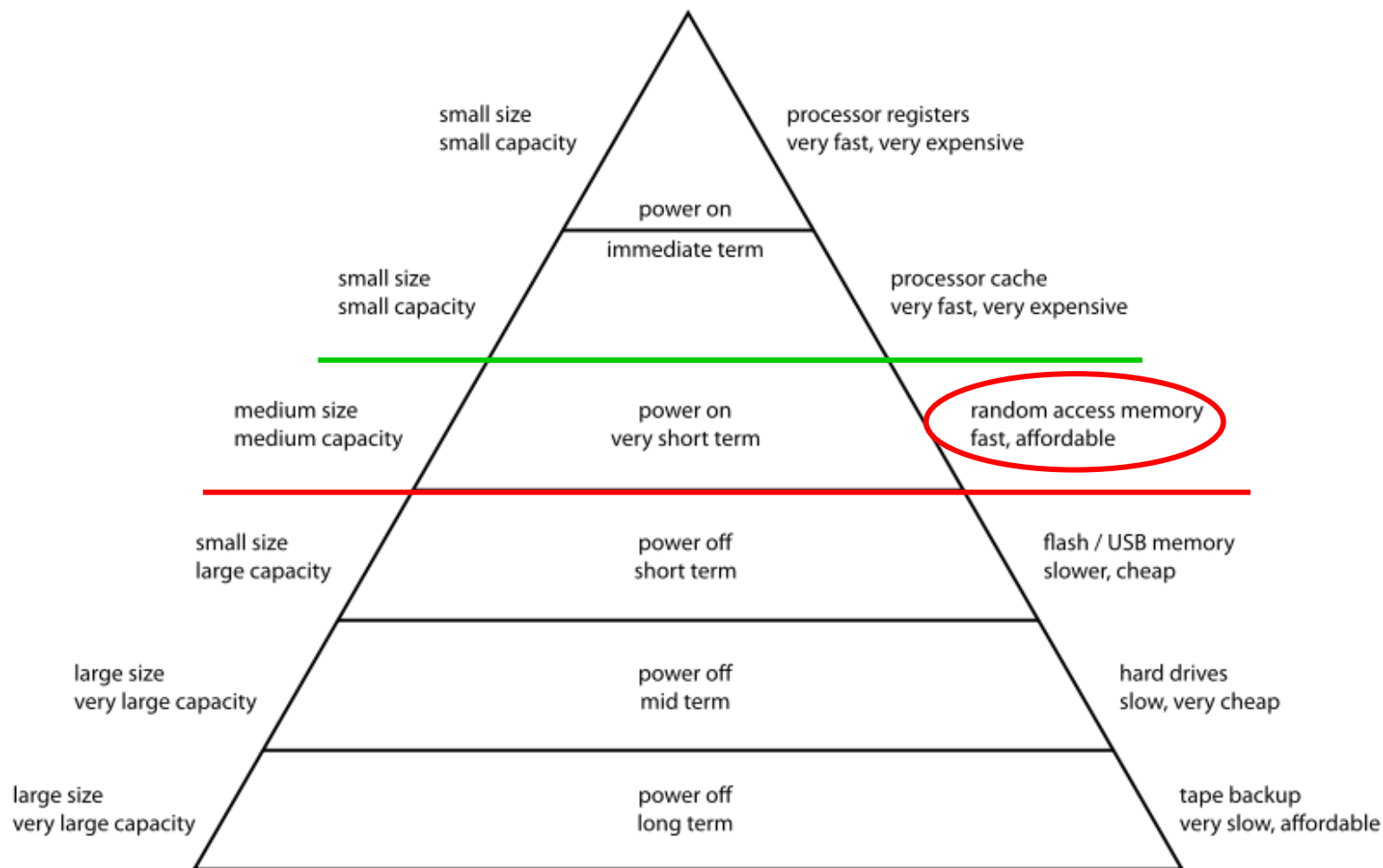
Source: https://en.wikipedia.org/wiki/Von_Neumann_architecture

Jonh von Neumann, **First Draft of a Report on the EDVAC**, 1945



现有计算机体系结构

- 层次化存储结构 Computer Memory Hierarchy



图片来源: https://en.wikipedia.org/wiki/Memory_hierarchy



现有计算机体系结构

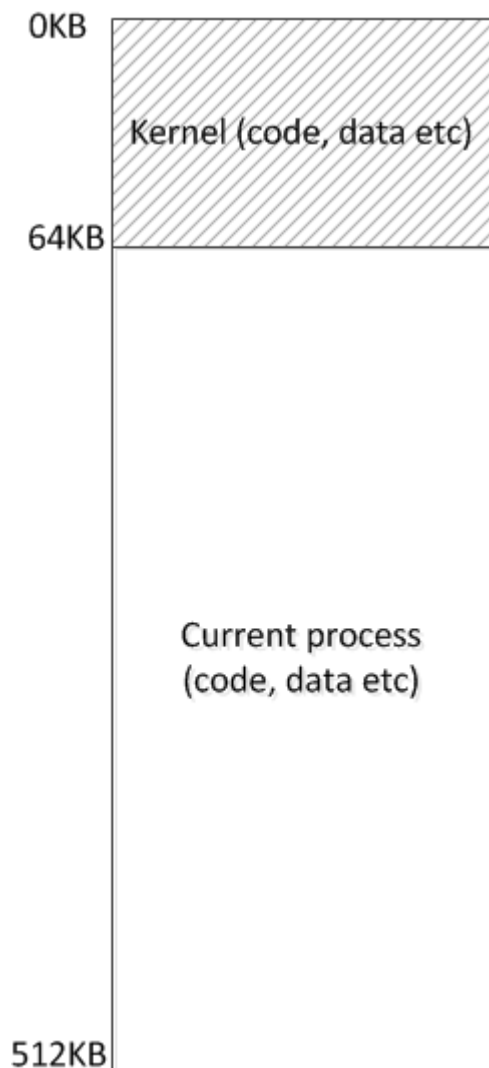
- 内存DRAM：快，但贵，容量小，易失性
- 外存磁盘：持久化，便宜，容量大，但慢

现有计算机系统存储层次	延迟	容量
Register	0.4~1ns	2K
L1 cache	1~4ns	16~32K
L2 cache	5~10ns	64~256K
L3 cache	20~40ns	1~32M
Memory	100ns	4~16G
Flash-based SSD	20~100us	1~4T
Hard disk	3~5ms	1~4T

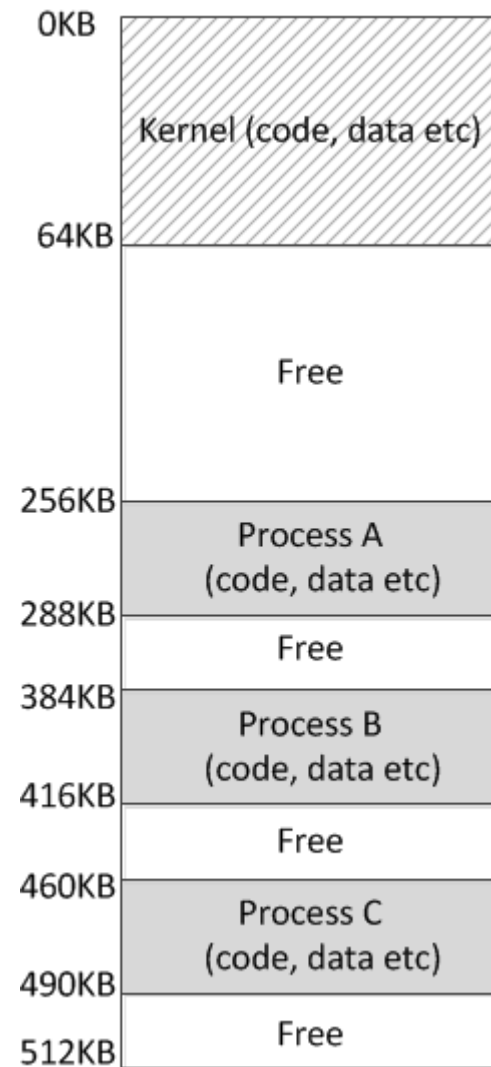


问题

- 如何高效地使用内存空间？



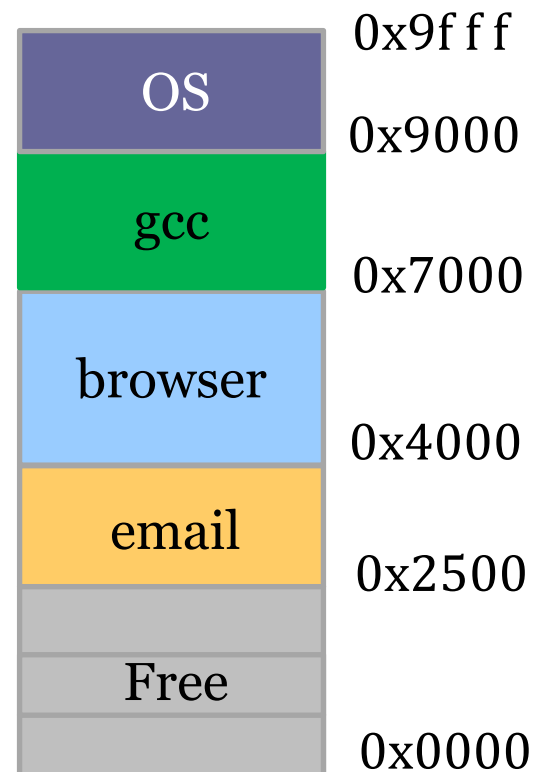
Multiprogramming & Time sharing





最简单的系统

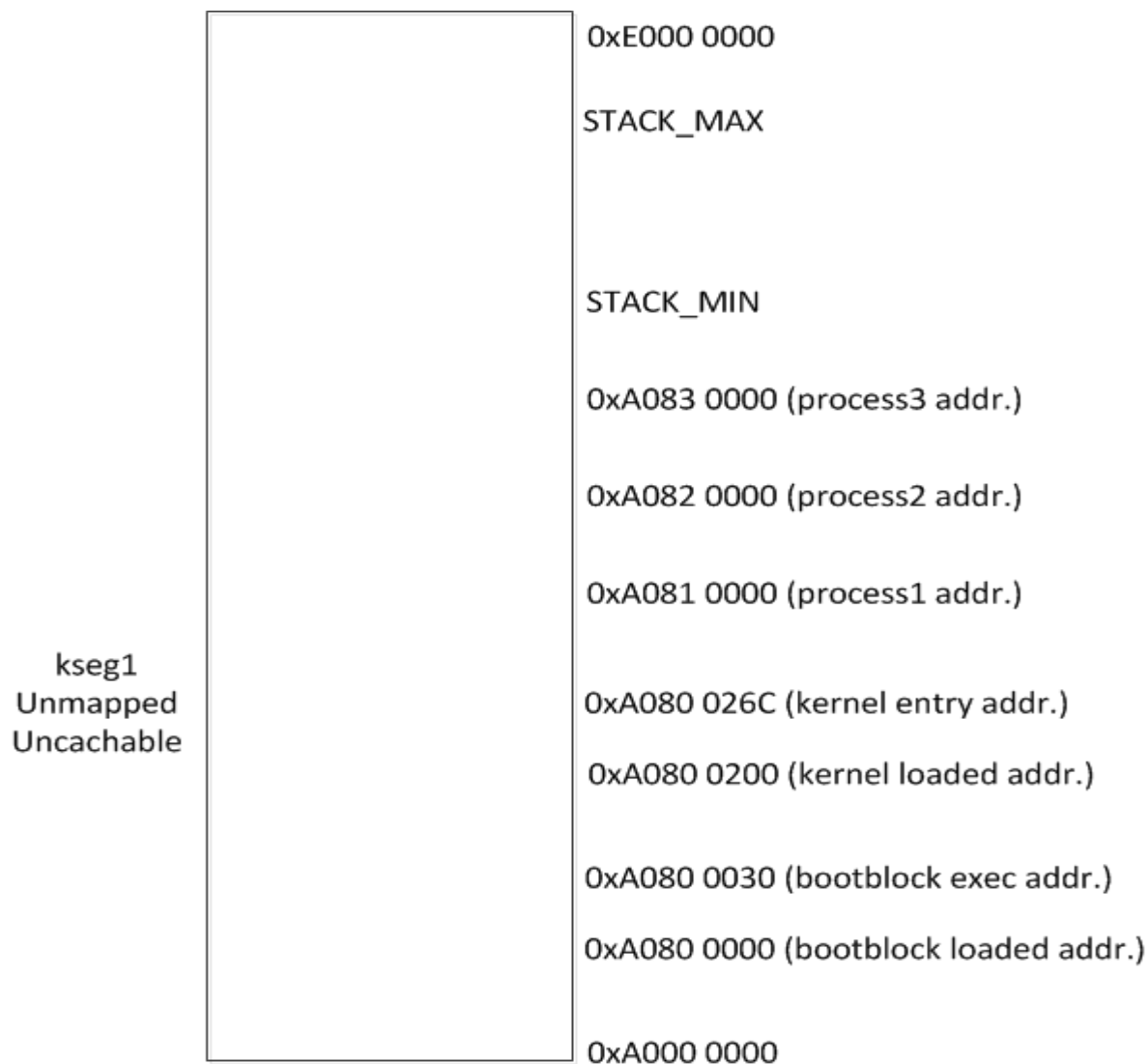
- 只有物理内存
 - 应用程序直接使用物理内存
- 例子：运行3个进程
 - Email , browser, gcc
- 可能会发生以下这些情况
 - email发生地址错误
 - browser对地址0x7050写
 - gcc需要扩展内存
 - browser需要比物理内存更多的内存





最简单的系统

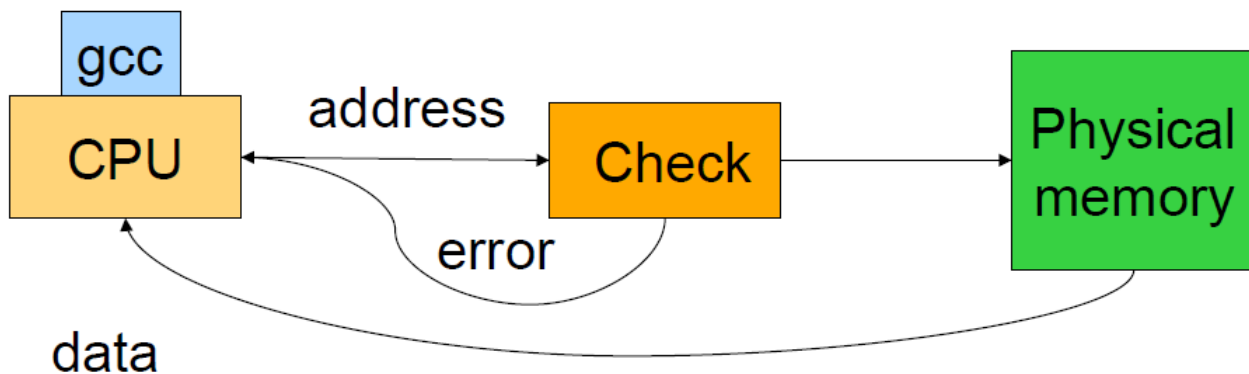
- 物理内存静态划分





需求一：进程保护

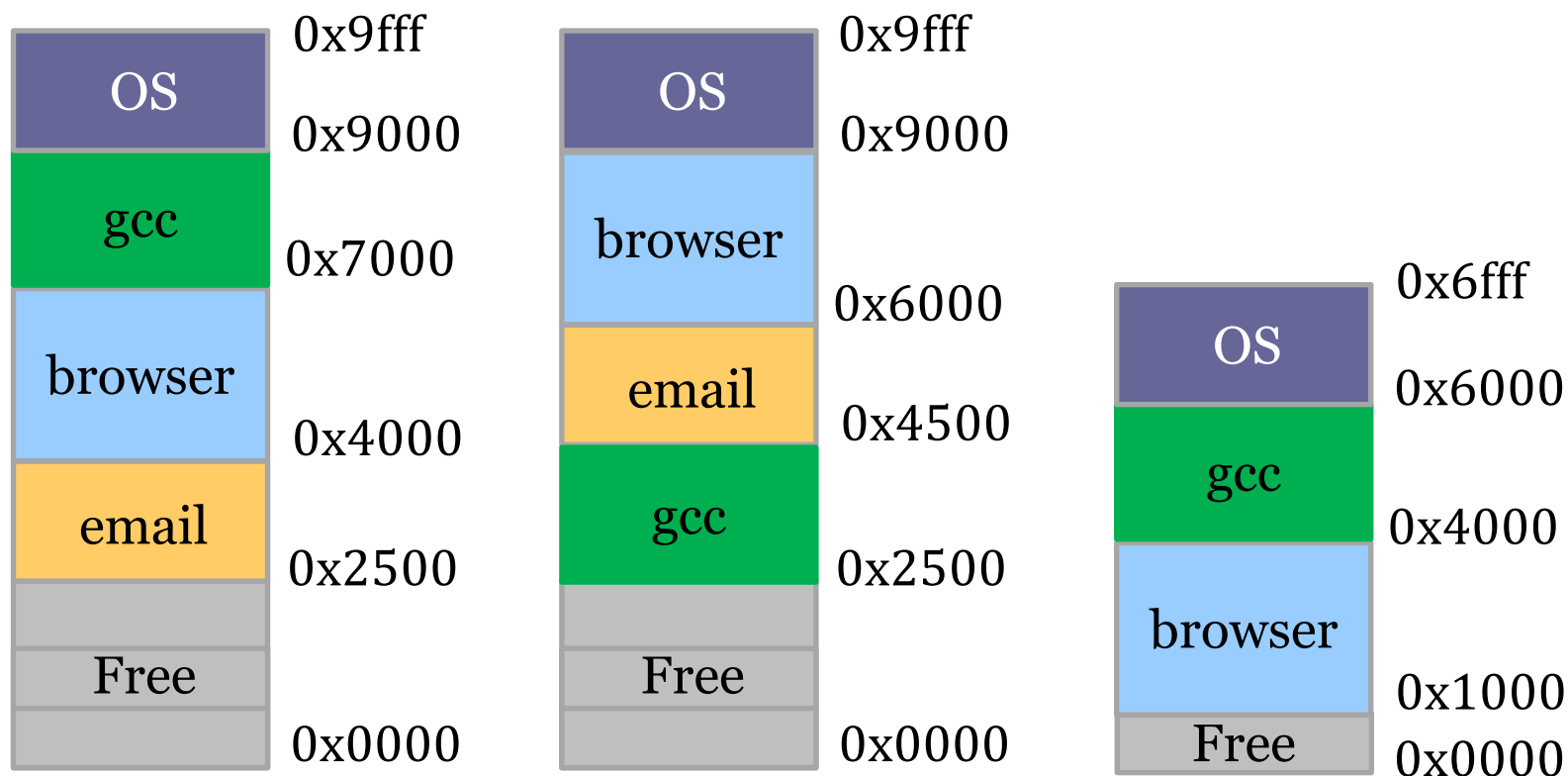
- 一个进程出错不能影响其它进程
- 对每次内存访问都进行检查，只允许合法的内存访问
 - 每个进程的每个load和store指令、隐式访存指令





需求二：扩展内存和应用透明

- 一个进程必须能运行在不同的物理内存区域上
- 一个进程必须能运行在不同的物理内存大小上





问题

- 如何高效地使用内存空间？
 - 目标1：同时运行多个进程
 - 系统运行的进程越多越好
 - 目标2：地址空间足够大
 - 一个大进程，其大小超过物理内存
 - 很多小进程，但它们的总大小超过物理内存
 - 目标3：保护
 - 一个用户进程不能读取、更不能修改另一个用户进程的内存
 - 用户进程不能破坏内核的内存



解决方案

- 基本内存抽象
 - 地址空间：进程的内存视图 → 虚拟内存
 - 透明使用，高效访问，安全保护
- 虚拟内存 vs 虚拟CPU
 - 虚拟CPU
 - 进程不与CPU绑定，可以迁移到任一CPU
 - 进程执行时，以为“独占”CPU
 - 进程CPU状态与切换：上下文
 - 虚拟内存
 - 进程数据不与内存绑定，可以迁移到任意的内存/磁盘位置
 - 进程执行时，以为“独占”内存
 - 进程内存状态：？



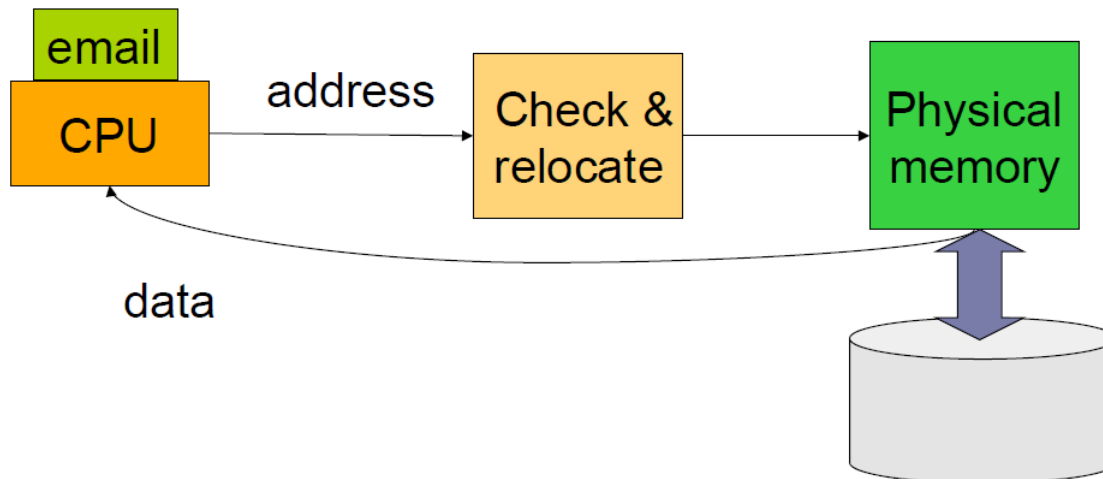
内容提要

- 计算机存储体系结构
- 虚存
 - 虚拟化
 - 保护
- 地址映射
 - 基址+长度
 - 分段
 - 分页
- MMU和TLB



虚拟内存 (Virtual Memory)

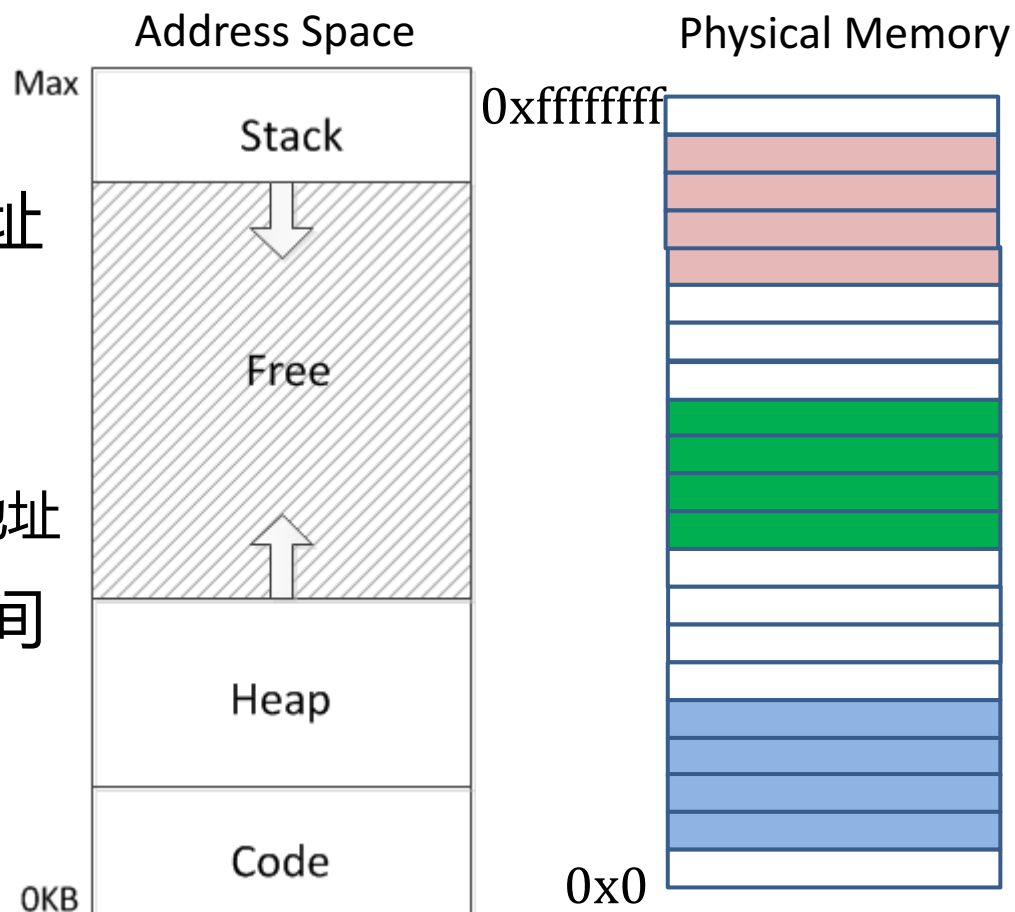
- 独立的 (进程) 地址空间
 - 给每个进程提供一个很大的、静态的 “虚拟” 地址空间
- 虚实地址转换
 - 进程运行时，每次访存通过地址转换 (relocate) 获得实际的物理内存地址
- 磁盘作为内存的延展 (**磁盘交换区**)
 - 按需加载：只装载部分地址空间至内存





地址空间

- 独立的进程地址空间 $[0, \text{max}-1]$
 - 程序员看到的是虚地址
- 运行时装载部分地址空间
- 每次访存：虚地址 \rightarrow 物理地址
 - CPU看到的是虚地址
 - 进程看到的是虚地址
 - 内存与I/O设备看到的是物理地址
- 如果访问到未装载的地址空间
 - 通知OS将它加载进内存





虚存的好处

- 灵活
 - 进程在执行时才放进内存，一部分在内存 & 另一部分在磁盘
- 简单
 - 进程的内存访问变得非常简单
- 高效
 - 20/80原则：20%的地址空间承担80%的访问
 - 将20%地址空间放进物理内存
- 安全
 - 虚实地址转换时进行安全检查，防止非法访问



虚存设计

- 设计问题
 - 如何进行地址转换？
 - 如何划分内存？
 - 如何实施保护？



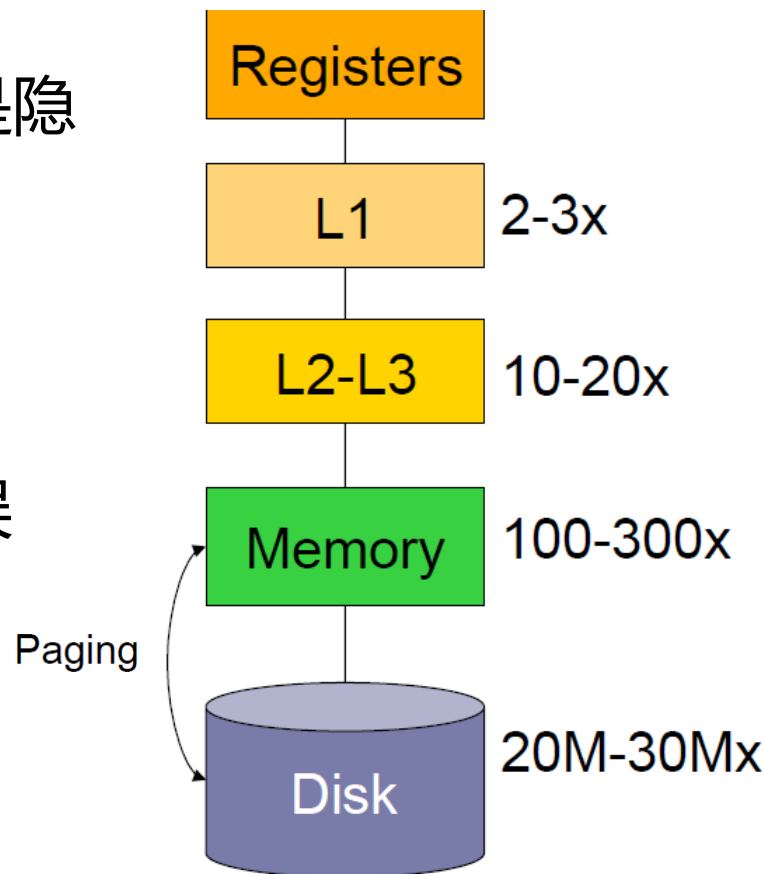
内容提要

- 计算机存储体系结构
- 虚存
 - 虚拟化
 - 保护
- 地址映射
 - 基址+长度
 - 分段
 - 分页
- MMU和TLB



地址转换

- 目标
 - 隐式：对每个内存访问，转换是隐式的
 - 快速：命中内存时必须非常快
 - 例外：不命中时触发一个例外
 - 保护：能够隔离用户进程的错误





地址映射和粒度

- 需要某种“映射”机制
 - 把虚地址空间（大）的内容放进物理内存空间（小）
- 映射必须有合适的粒度
 - 粒度决定灵活性
 - 大粒度映射可能造成内存浪费
 - 细粒度映射需要更多的映射信息（trade-off）
- 极端情况
 - 字节粒度映射：映射表过大
 - 进程粒度映射：内存浪费



基址+长度 (Cray-1采用的方法)

- **连续分配**

- 为每个进程分配地址连续的内存
- 用一个二元组来限定其内存区域： $\langle \text{base}, \text{bound} \rangle$

- **保护**

- 一个进程只能访问 $[\text{base}, \text{base} + \text{bound}]$ 区间的内存

- **上下文切换**

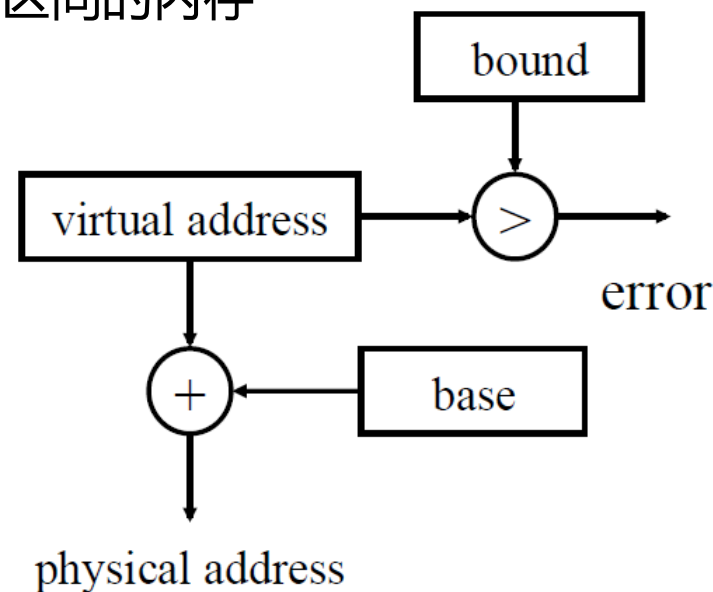
- 保存/恢复基址寄存器及上限寄存器

- **好处**

- 简单：映射时将虚地址与基址相加
- 支持换出(swapping)：多进程并发执行

- **坏处**

- 外部碎片（进程间的碎片）
- 难以支持进程增大
- 难以共享内存

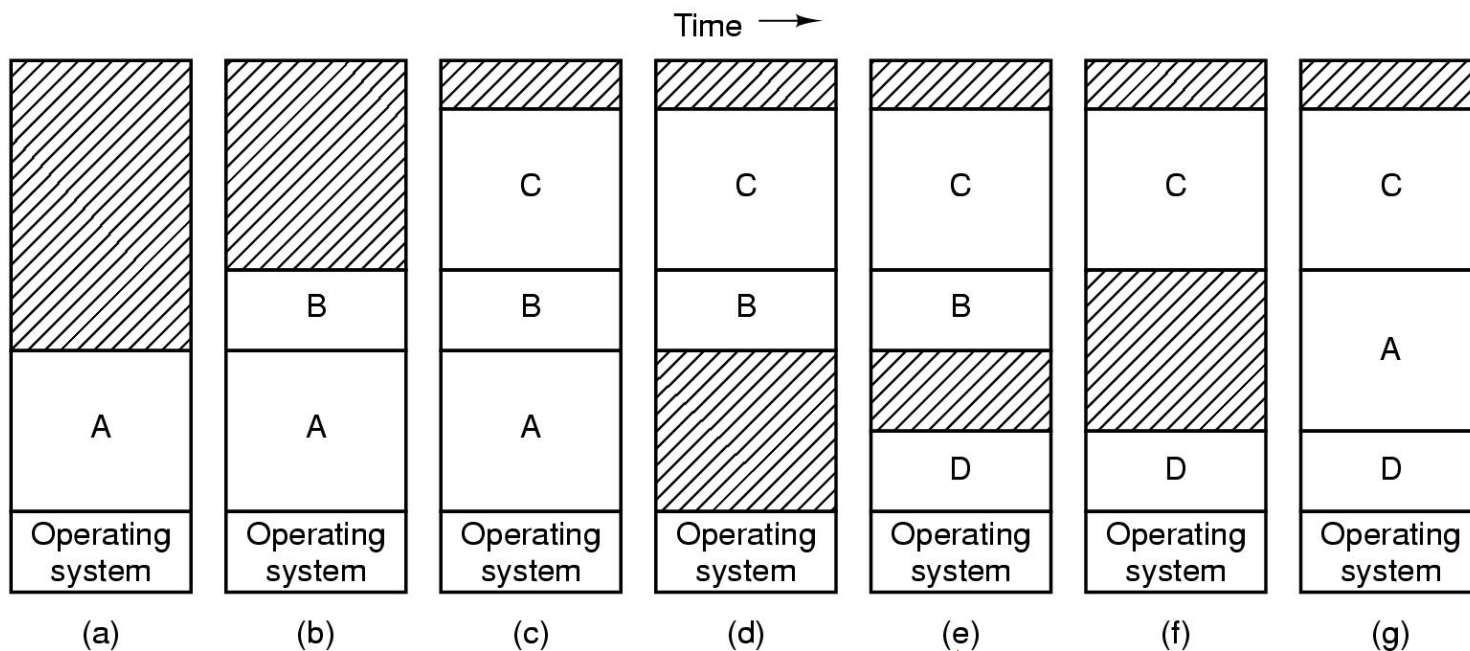




内存碎片

- 碎片问题

- 随着进程的换入与换出，内存产生很多空洞（未使用的小区域）



如果此时A程序要运行

- 要么把另一个进程换出，如(f)(g)所示
- 要么进行碎片聚合（memory compaction）

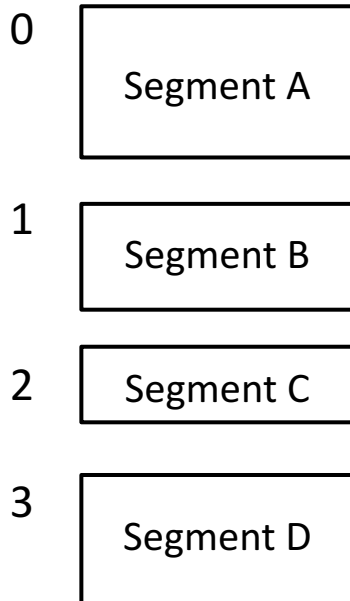


分段

• 不连续分配

- 把程序逻辑上划分为若干段：代码、全局变量、栈、...
- 每个段分配连续内存，段间不必连续
- 每个进程有一张段表：(seg, size)
- 每个段采用基址+长度

Virtual Memory



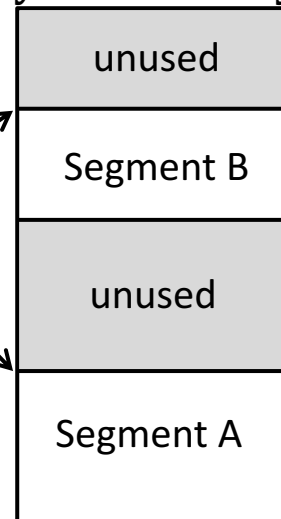
Segment Table

valid	size	base
1	10KB	_____
1	6KB	_____
0	5KB	_____
0	8KB	_____

Virtual address

seg #	offset
-------	--------

Physical Memory

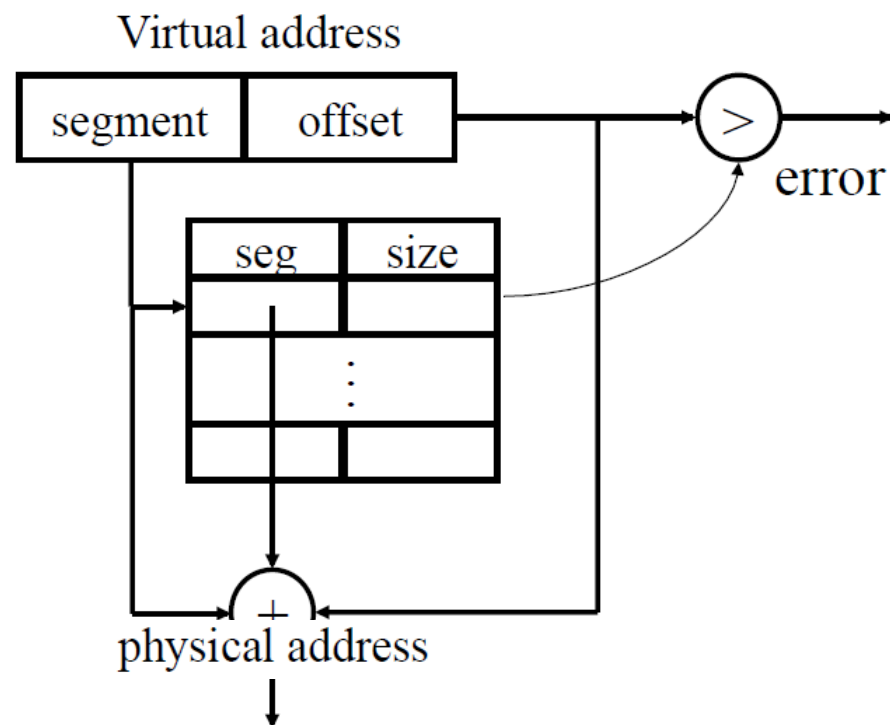


如果访问地址的offset ≥ 段size，则memory violation
如果访问地址所在seg#的valid为0，则segment fault



分段

- 保护
 - 每个段有(read, write, exec)
- 上下文切换
 - 保存/恢复 段表和指向段表的内核指针
- 好处
 - 相比基址+长度，资源使用更高效
 - 易共享
- 不足
 - 管理复杂
 - 外部碎片（段间碎片）

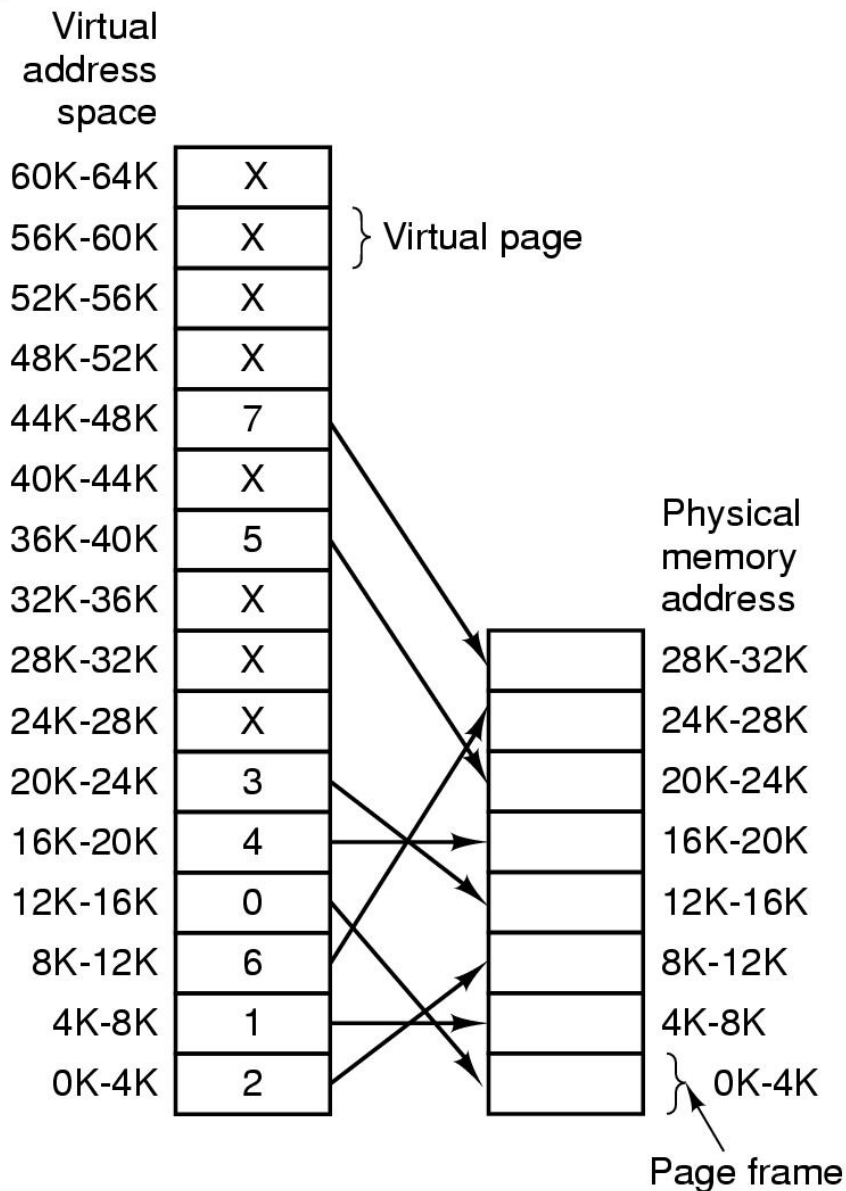




分页

- 分页机制

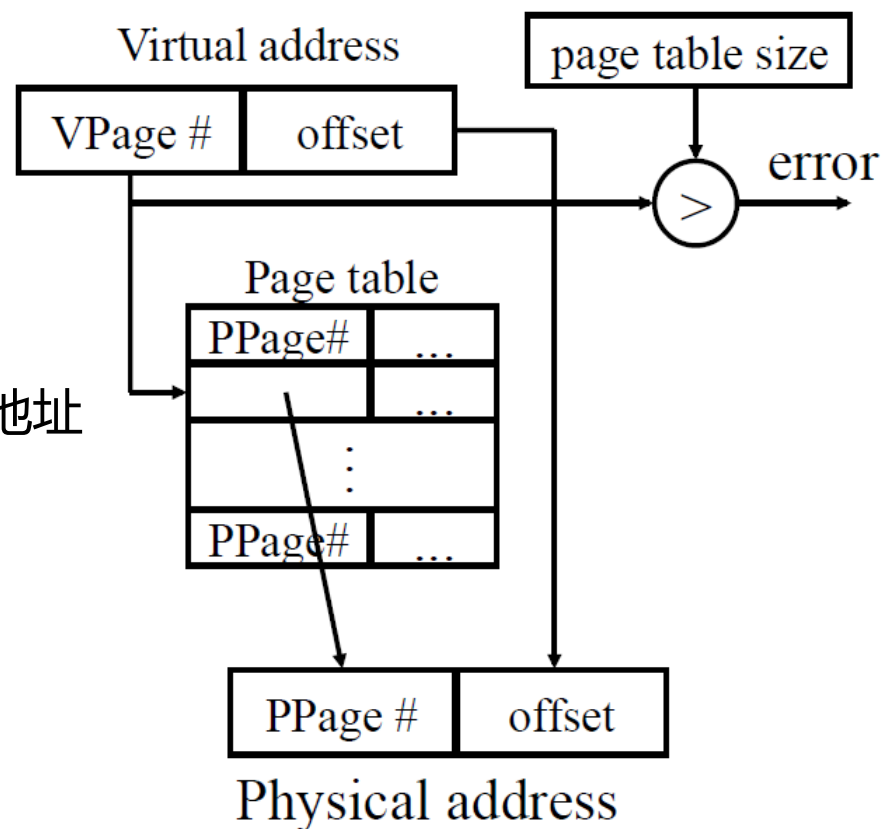
- 使用固定大小的映射单元
- 把虚存划分成固定大小的单元（称为页，page）
- 把物理内存划分成同样大小的单元（称为页框，page frame）
- 按需加载





分页

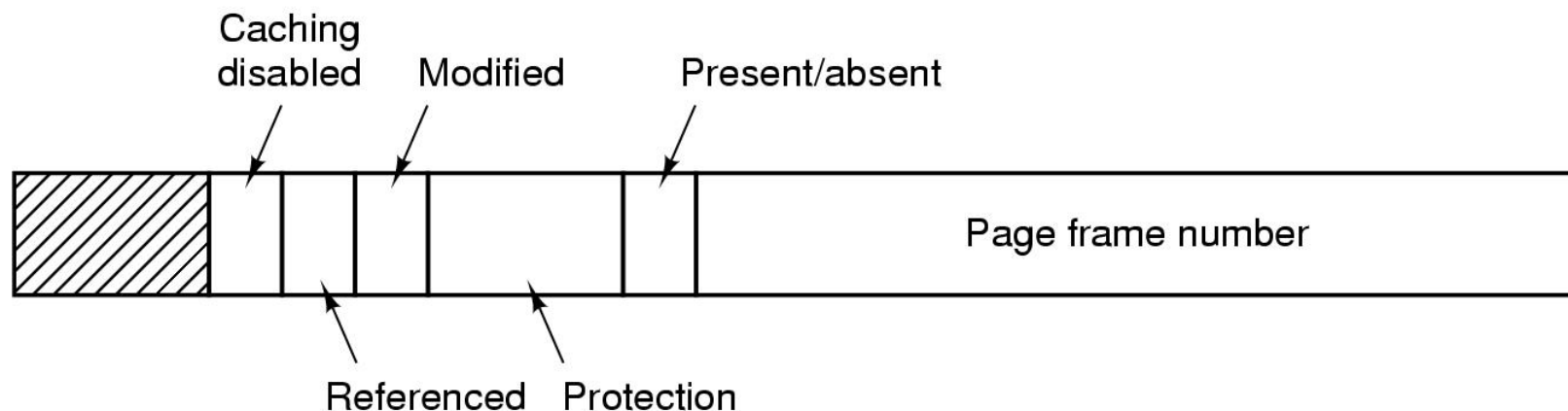
- 用**页表**来记录映射
 - 虚页 → 物理页
- 每个表项有若干控制位
 - 按页保护 (read, write, exe)
- 上下文切换
 - 与分段类似：保存/恢复页表及其地址
- 好处
 - 分配简单
 - 易共享
- 坏处
 - 页表很大
 - 进程地址空间有很多空洞：对应的页表项无用





页表项

- 表达一个映射关系 (Page Table Entry , PTE) : 虚页号→物理页号



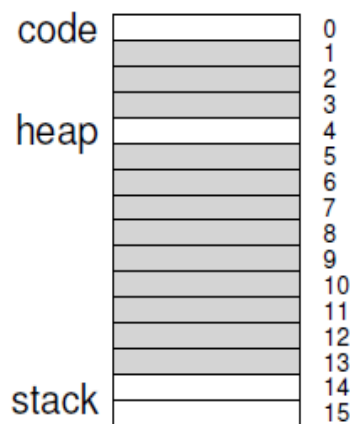
- PP#
- 控制位
 - 有效位(Valid) : 1位 , 标识该页在内存(Present)或不在内存(absent) , V位
 - 保护位(Protection) : 1~3位 , read, write, exe
 - 修改位(Modified) : 1位 , M位
 - 访问位(Referenced) : 1位 , R位
 - 缓存位(Cacheable) : 1位 (讲设备时会用到)



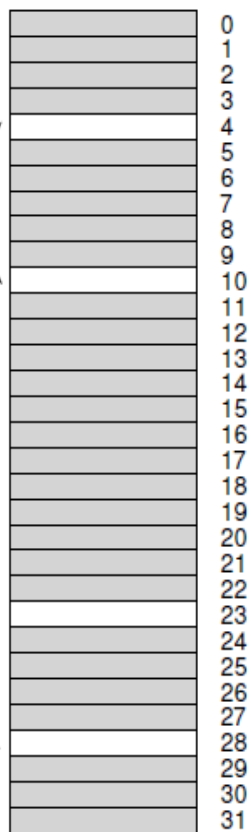
线性页表

- 线性页表

Virtual Address Space



Physical Memory



PFN	valid	prot	present	dirty
10	1	r-x	1	0
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
23	1	rw-	1	1
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
-	0	—	-	-
28	1	rw-	1	1
4	1	rw-	1	1



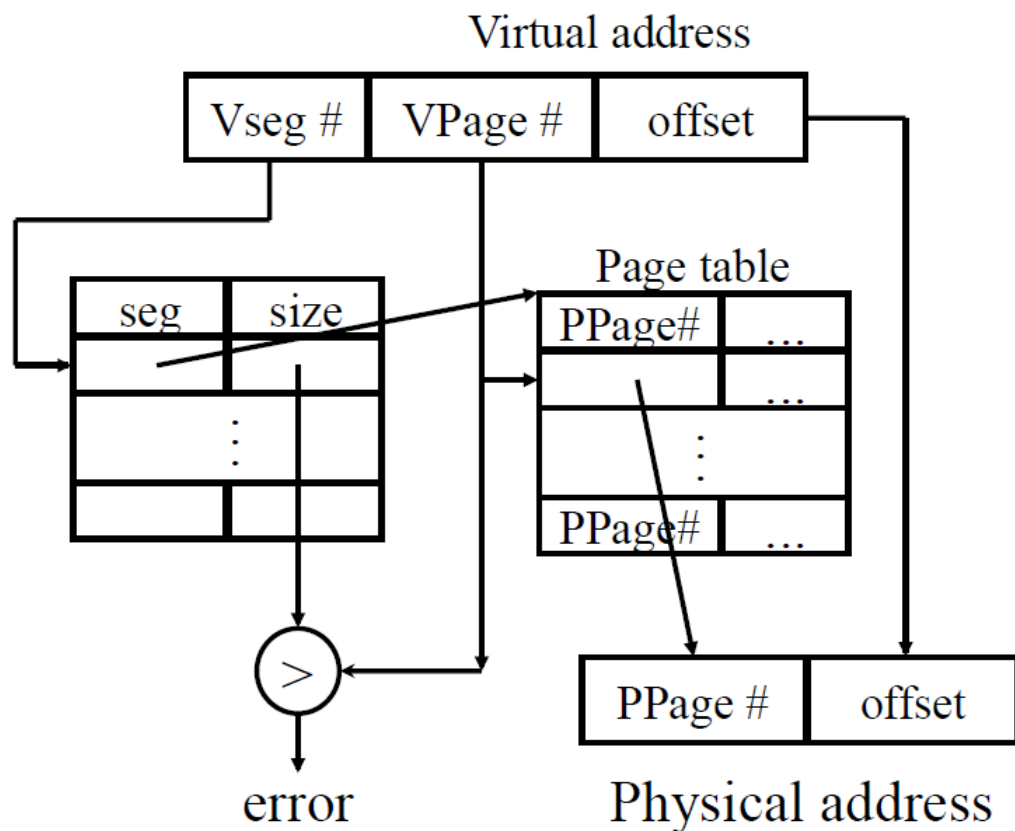
页表项 (PTE) 数量

- 假设4KB的页
 - 低12位为页内偏移
- 32-位地址的机器
 - 每个进程的页表有 2^{20} 个页表项，每一项4B (~4MiB)
 - 页表所需内存空间 = 进程数量 x 4MB
 - 如果有10K个进程，内存放不下所有的页表
- 64-位地址的机器
 - 每个进程的页表有 2^{52} 个页表项
 - 页表所需内存空间 = 进程数量 x 2^{52}
 - 一个进程的页表可能磁盘都存不下 (2^{52} PTEs = 16PiBytes!)



分段+分页

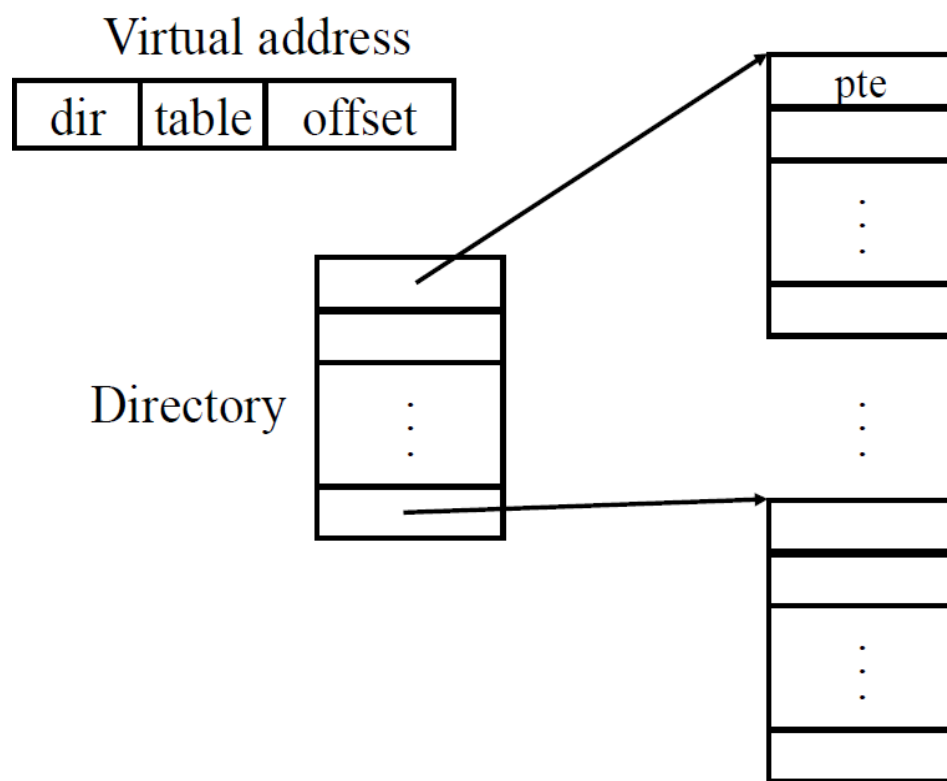
- 先将进程划分为若干段
- 每个段采用分页
- 段表记录它的页表地址
- 不足
 - 分段的不足仍然存在





多级页表

- 虚地址除去offset之外的部分划分为多个段
 - 每段对应一级页表
 - 多个页表
- 好处
 - 节省空间
- 分段+分页和多级页表
 - 区别？





示例：两级页表

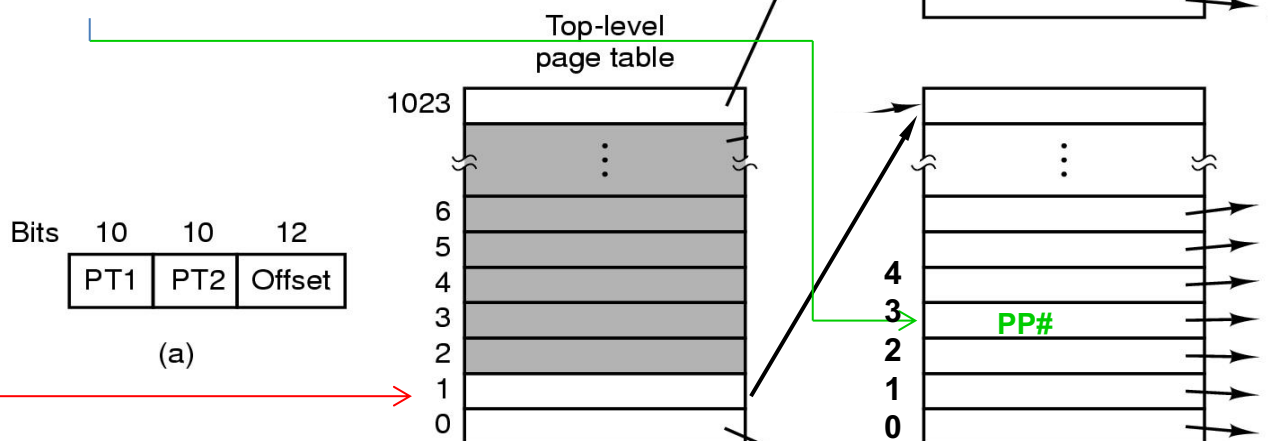
虚地址：0X00403004（32位虚地址空间，4KiB/页）

0000 0000 0100 0000 0011 0000 0000 0100

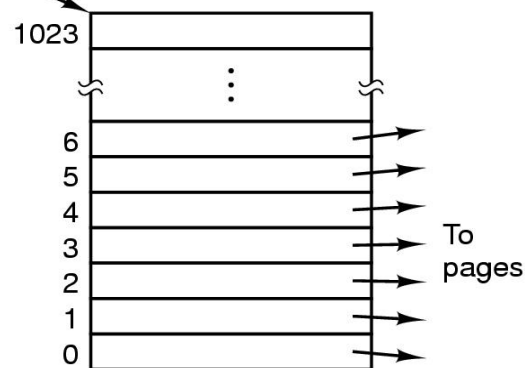
PT1=1

PT2=3

Offset=4



- 每个页表4KiB，1024个表项
- 下级页表的每一项映射1页（4KiB）
- 上级页表的每一项映射4MiB的地址空间
- 对于大地址空间，大部分程序只需要几个页表

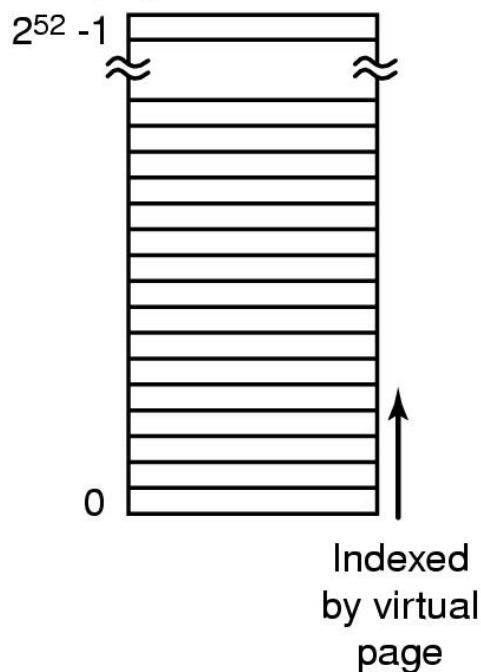




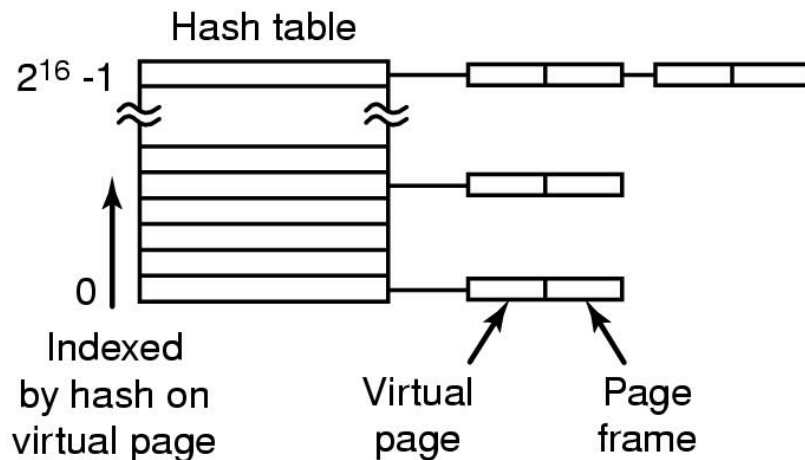
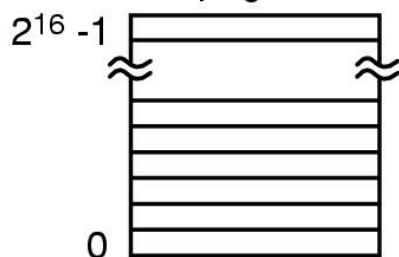
反向页表

- 64位地址空间，4KiB页，总共有 2^{52} 个页
- 256MiB物理内存，总共有 2^{16} 个页框
- 按物理页索引，记录每个物理页对应的进程ID及虚页

Traditional page
table with an entry
for each of the 2^{52}
pages



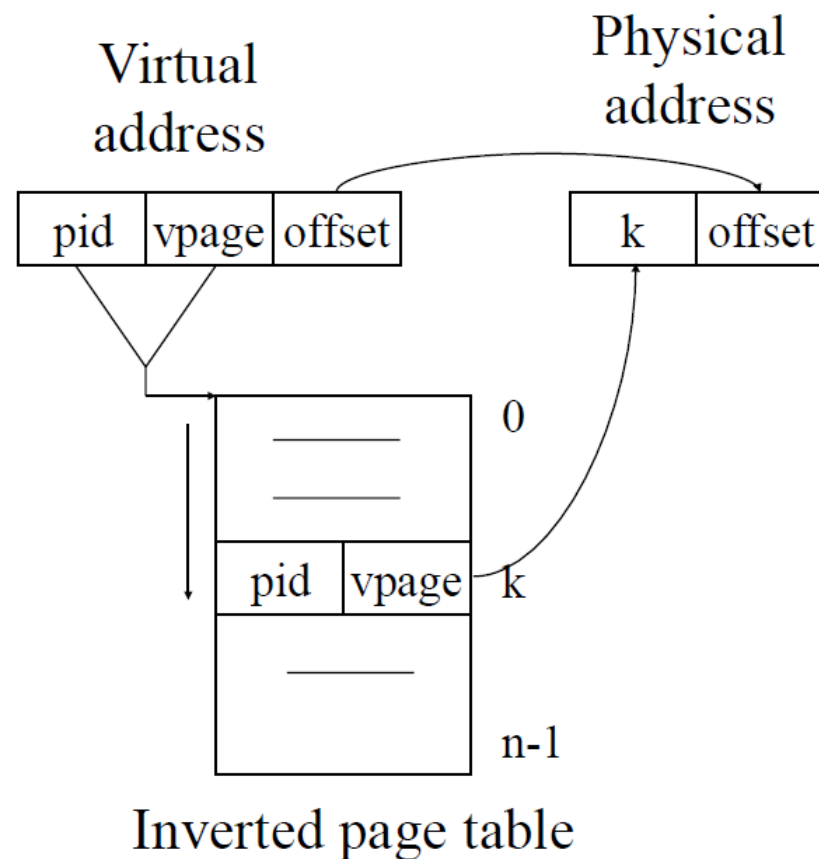
256-MB physical
memory has 2^{16}
4-KB page frames





反向页表

- 主要思想
 - 每个物理页一个PTE
 - 地址转换：哈希查找
 - $\text{Hash}(\text{Vpage}, \text{pid}) \rightarrow \text{Ppage\#}$
- 好处
 - 页表大小与地址空间大小无关
只与物理内存大小有关
 - 对于大地址空间，页表较小
- 坏处
 - 查找难（哈希冲突）
 - 管理哈希链等的开销





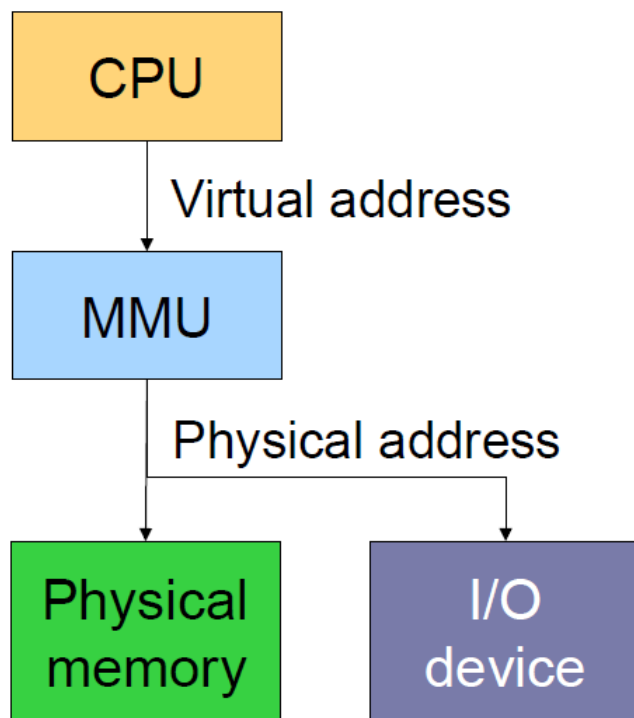
内容提要

- 计算机存储体系结构
- 虚存
 - 虚拟化
 - 保护
- 地址映射
 - 基址+长度
 - 分段
 - 分页
- MMU和TLB



地址转换实现

- 地址类别
 - CPU发出的是虚地址
 - 内存和I/O设备接收的是物理地址
- MMU职责
 - Memory Management Unit，负责虚地址到物理地址转换的硬件单元
 - 通常在片内实现，每个CPU有一个base寄存器和一个register寄存器
 - 虚存地址转换为物理地址，每条load和store指令都需要地址转换
 - 内存保护，检查地址是否有效
 - 特殊指令操作base和bound寄存器





地址转换实现

- 操作系统职责
 - 内存管理
 - 新进程分配空间，结束的进程回收空间
 - 进程切换时上下文管理
 - 例如，保存当前进程的base-bound值，设置即将运行进程的base-bound值
 - 例如，设置即将运行进程的页表基址，例如x86 CR3寄存器，RISC-V satp寄存器
 - 异常处理
 - 内存越界访问、无效地址（例如base-bound不存在）等



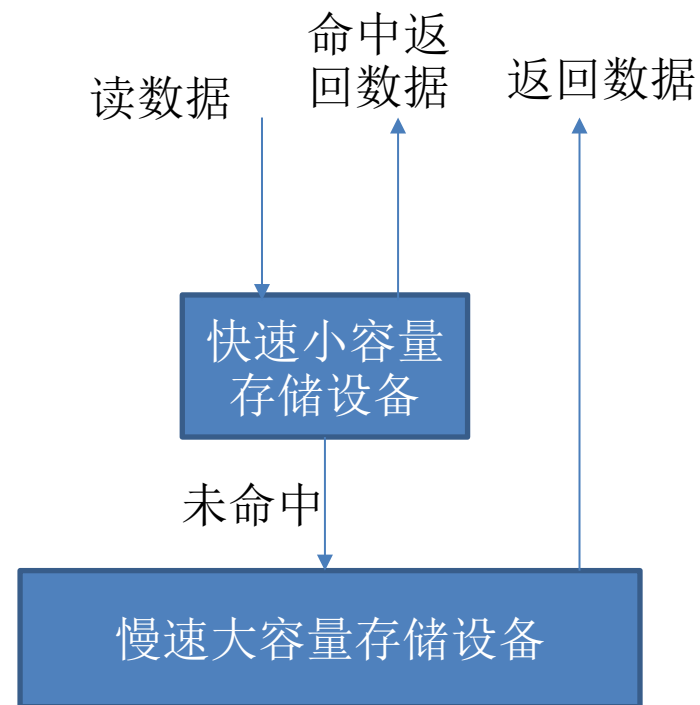
加速地址转换

- 程序只知道虚地址
 - 每个程序或进程的地址空间是 $[0, \text{max}-1]$
- 每个虚地址必须进行转换
 - 可能需要逐级查找多级页表
 - 页表保存在内存中，一个内存访问变成多个内存访问
- 解决办法
 - 用速度更快的部件来缓存使用最频繁的那部分页表项



缓存机制

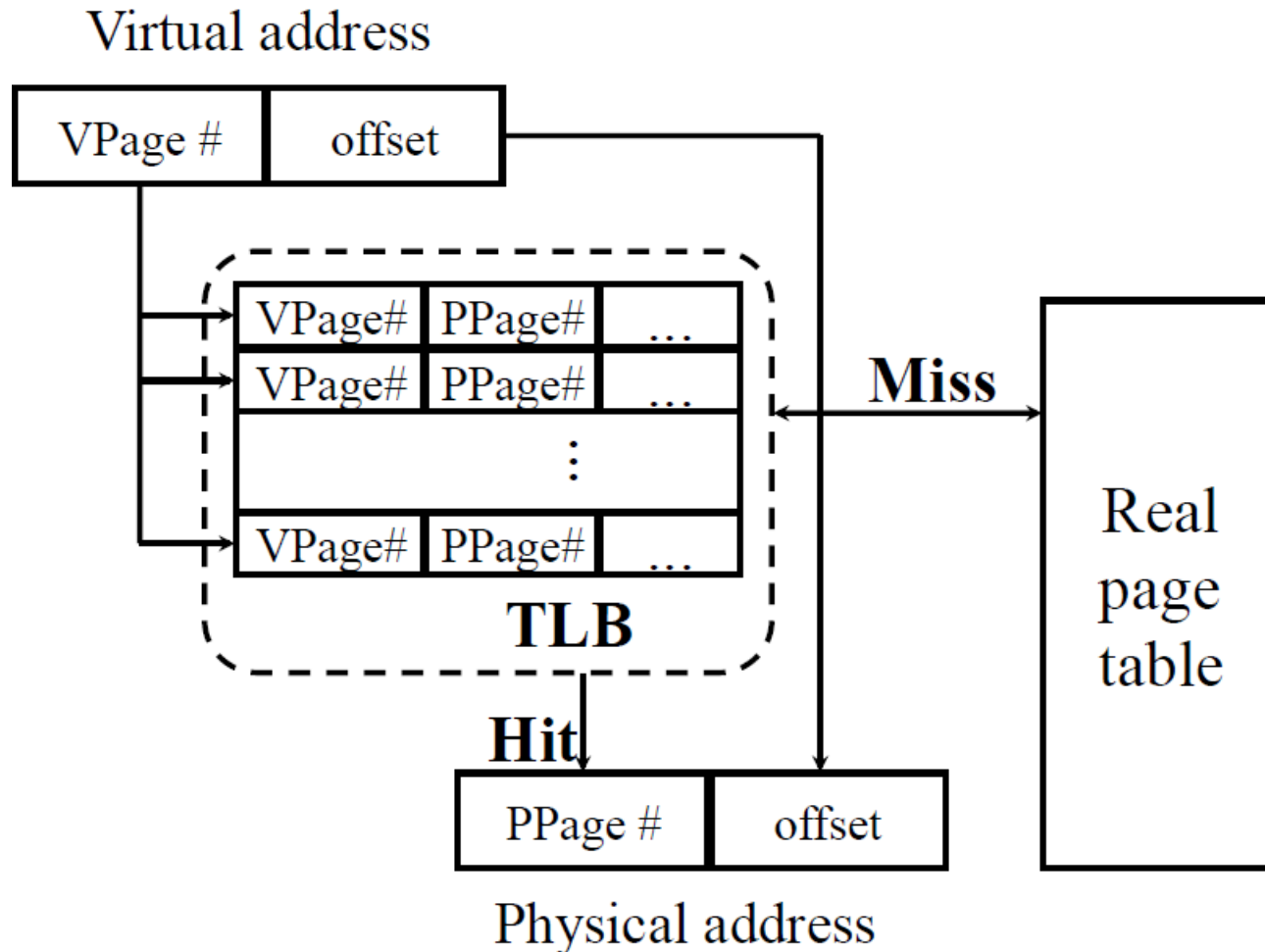
- 缓存是计算机系统的一种重要机制
 - 层次化存储体系结构
 - 用更快速的存储设备保存数据，以免每次从低速存储设备中读取数据
 - 缓存空间小于总数据量大小
 - 缓存命中时，从缓存中读取数据
 - 未命中时，从低速设备读取数据
 - 缓存满时，替换现有数据
 - CPU cache , page cache





TLB: Translation Look-aside Buffer

- 一种页表的缓存机制





TLB表项的格式

- 共有的（必须的）位
 - VP#（虚页号）：与虚地址进行匹配
 - PP#（物理页号）：转换后的实际地址
 - Valid位：标志此表项是否有效
 - 访问控制位：允许内核/用户访问，以及允许何种访问（read, write）
- 可选的（有用的）位
 - 进程标签（pid）
 - 访问标志位（R位）
 - 修改标志位（M位）
 - 缓存标志位



硬件控制的TLB

- CPU把一个虚地址VA给MMU进行转换
- MMU先查TLB: $VA = VP\# \parallel \text{offset}$
 - 将该虚页号同时与TLB中所有表项进行比较，硬件支持
- TLB hit（命中）：TLB里找到含VP#的表项
 - 如果有效（TLB的valid位=1），取表项中的物理页号
 - 如果无效（TLB的valid位=0），则等同于TLB miss
- 如果TLB miss（不命中）：TLB里没有含VP#的表项
 - MMU硬件在页表中进行查找，得到PTE
 - 将找到的PTE加载进TLB
 - 如果没有空闲表项时，替换一个TLB表项
 - 并取TLB表项中的物理页号



软件控制TLB

- CPU把一个虚地址VA给MMU进行转换
- MMU先查TLB: $VA = VP\# \parallel \text{offset}$
 - 将该虚页号同时与TLB中所有表项进行比较，硬件支持
- TLB hit (命中) : TLB里找到含VP#的表项
 - 如果有效 (TLB的valid位=1)，取表项中的物理页号
 - 如果无效 (TLB的valid位=0)，则等同于TLB miss
- 如果TLB 不命中 : TLB里没有含VP#的表项
 - 进入内核异常处理程序 (软件)，软件在页表中进行查找，得到PTE
 - 软件将该PTE加载进TLB
 - 如果没有空闲TLB表项，则替换一个TLB表项
 - 重新执行发生TLB不命中的指令



硬件控制 vs. 软件控制

- 硬件控制
 - 高效
 - 不灵活
- 软件控制
 - 简化MMU的逻辑，使得CPU芯片上更多面积用于缓存
 - 软件控制灵活
 - 可以使用反向页表，进行映射，处理大的虚地址空间
 - $\text{hash}(\text{Pid}, \text{VP\#}) \rightarrow \text{PP\#}$

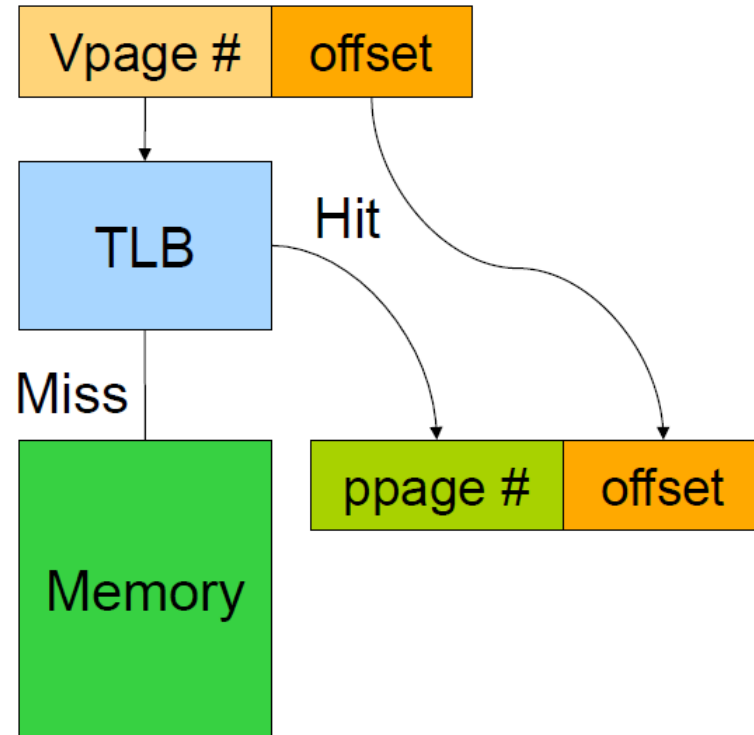
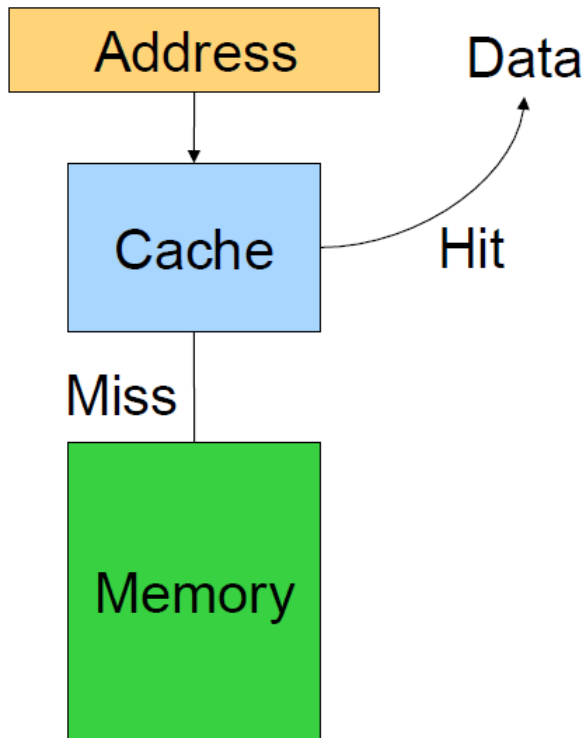


TLB设计问题

- 替换哪个TLB表项
 - 随机 或 最近最少使用算法 (伪LRU)
- 上下文切换时需要做什么
 - 有进程标签：修改TLB寄存器和进程寄存器的内容
 - 无进程标签：将整个TLB的内容置为无效
- 修改一个页表项时需要做什么
 - 修改内存中的PTE
 - 将对应的TLB表项置为无效 (TLB flush)
- TLB大小
 - 很小的TLB (64个表项) , 很好的TLB命中率
 - 不能太大 (不超过256个表项) , CPU的面积有限



CPU缓存 vs. TLB



- 相似之处

- 缓存一部分内存
- 不命中且满时，进行替换

- 不同之处

- 关联度：全关联，组关联
- 一致性：PTE修改



总结

- 虚存
 - 使得软件开发变得容易，而且内存资源利用率更高
- 进程地址空间
 - 分离地址空间能够提供保护和错误隔离
- 地址转换
 - 虚地址与物理地址
 - MMU



总结

- 地址映射
 - 基址+长度：简单，但有很大的局限性
 - 分段：有用，但太复杂
 - 分页
 - 页与页框
 - 页表与PTE
 - 大页表优化：分段+分页、多级页表、反向页表
- TLB
 - 加速地址转换的专门硬件
 - 但引入一致性问题