内存管理 (续)
Memory Management Cont’d

钮鑫涛
南京大学
2024春

回顾页表

- 一级页表：

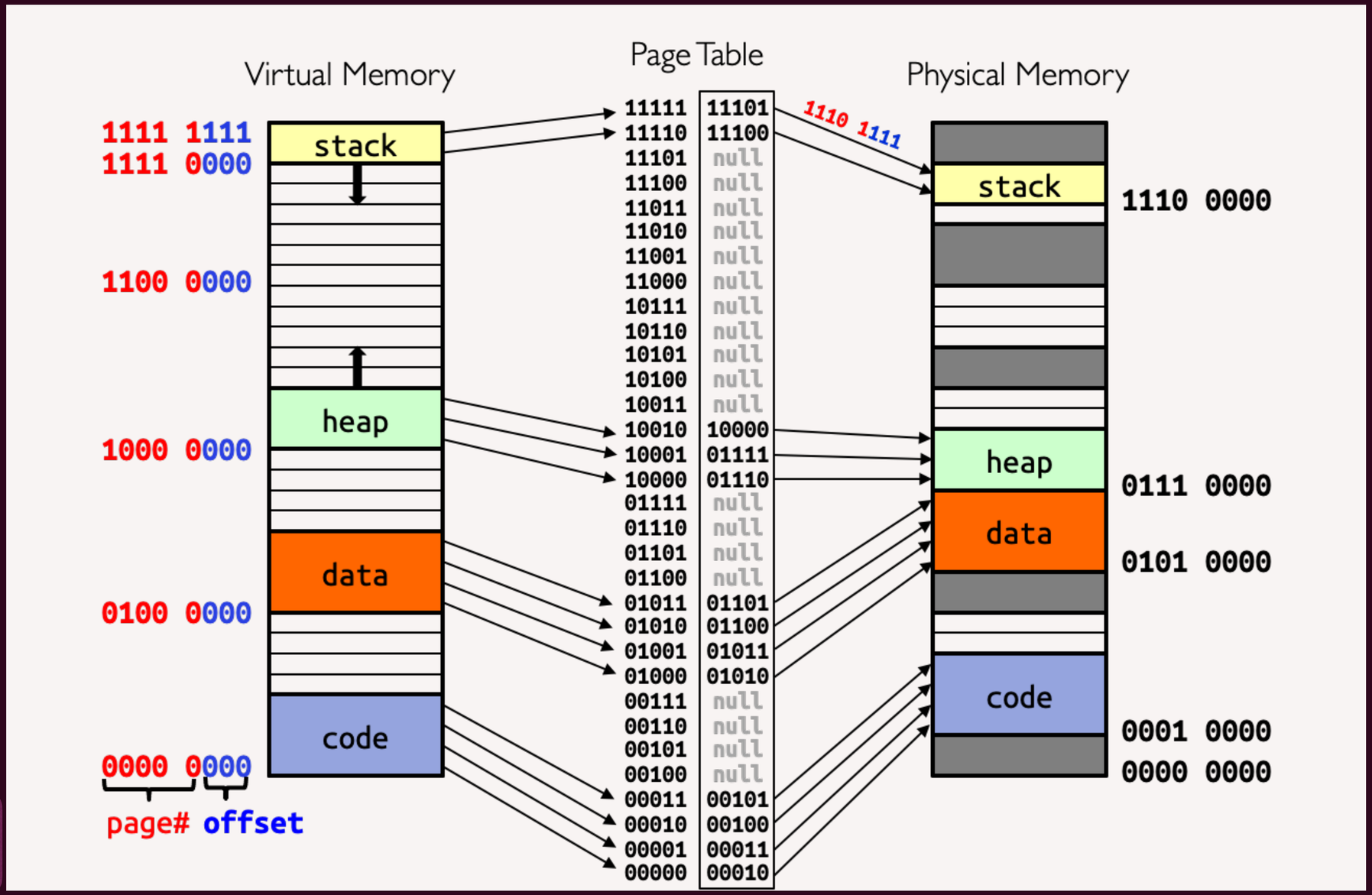
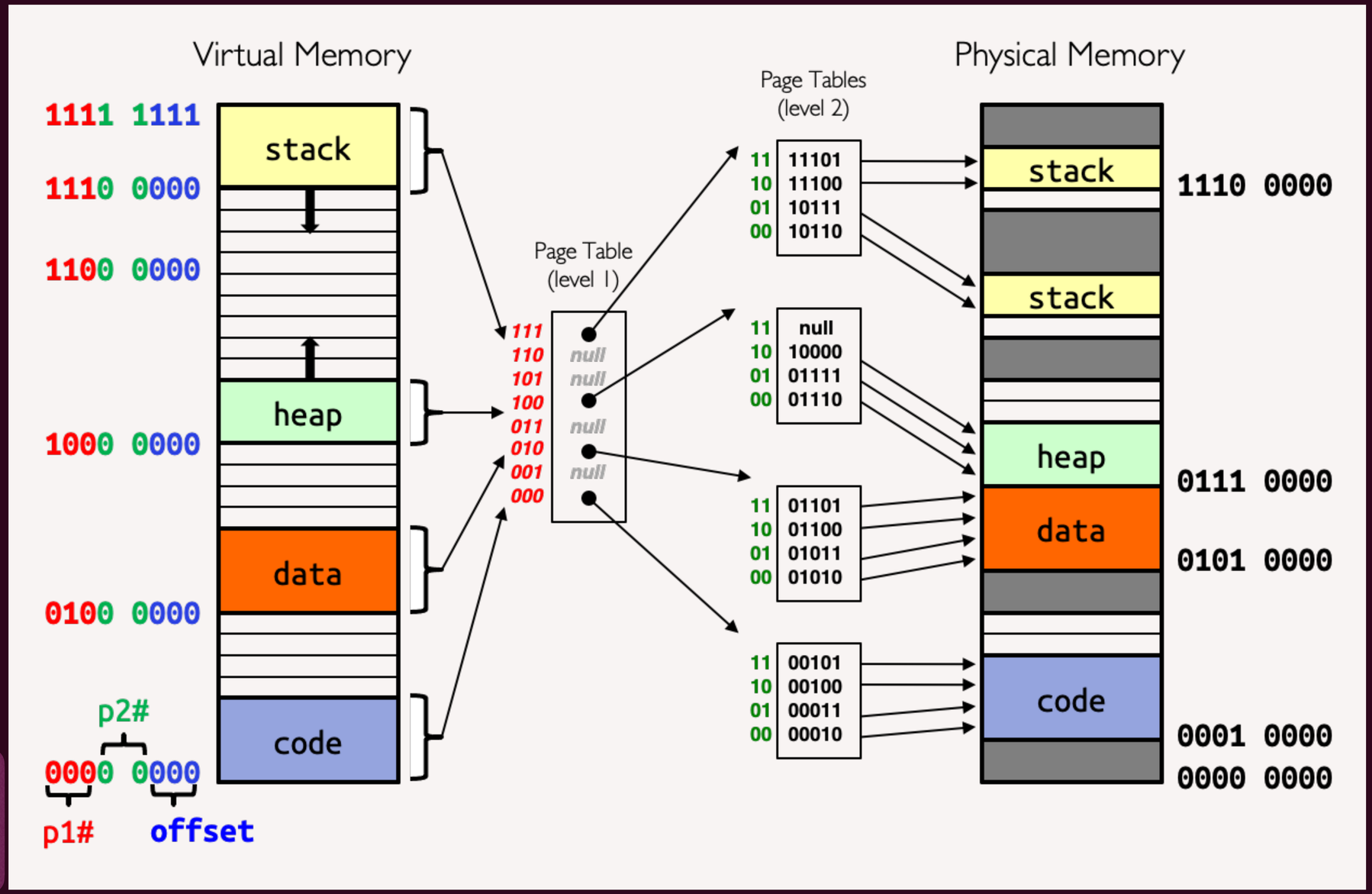


table size equal to # of pages in virtual memory!

回顾页表

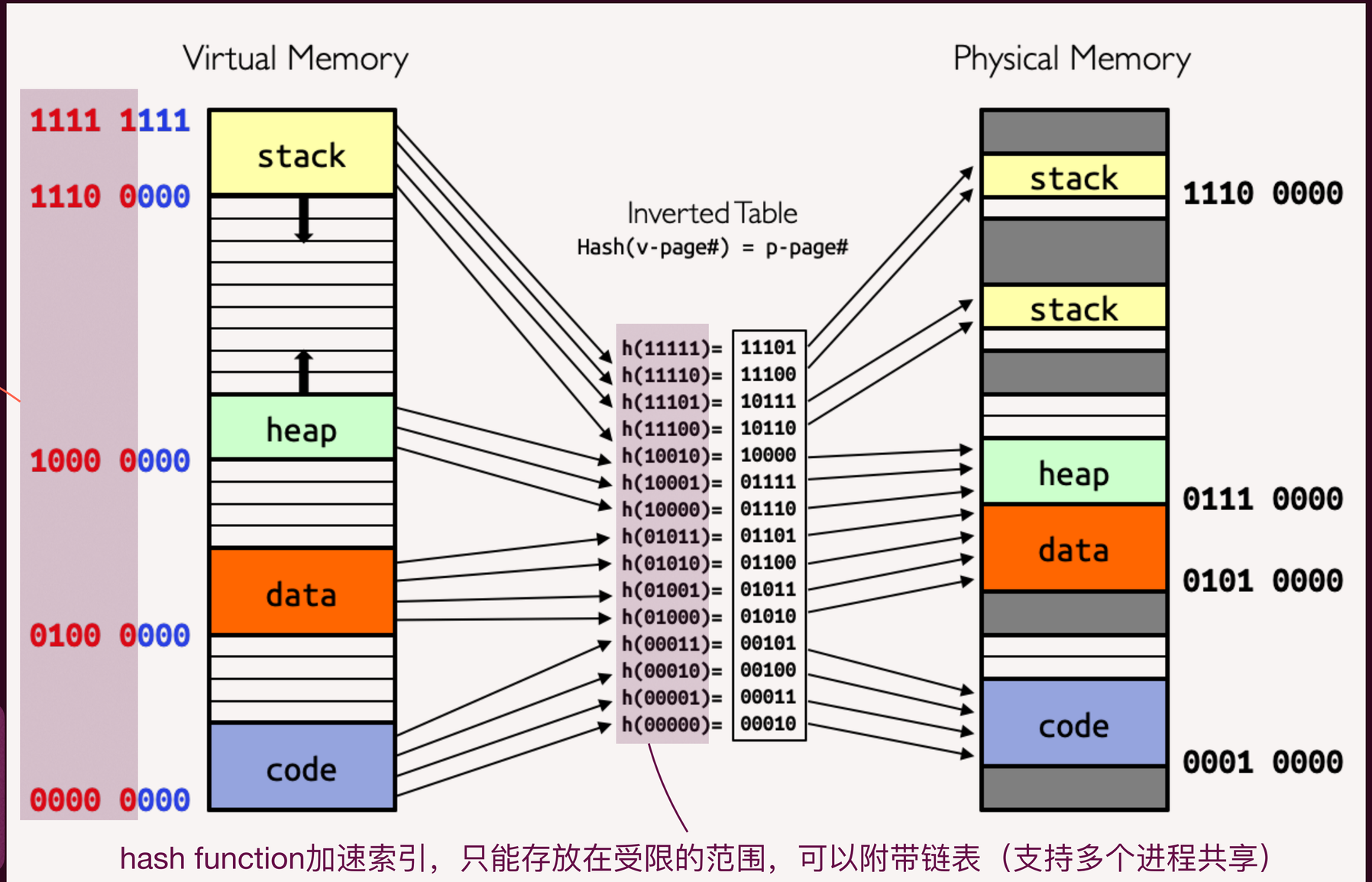
- 二级页表：



In best case, total size of page tables ≈ number of pages used by program virtual memory.

回顾页表

- 倒排（反置）页表（Inverted Page Table）：

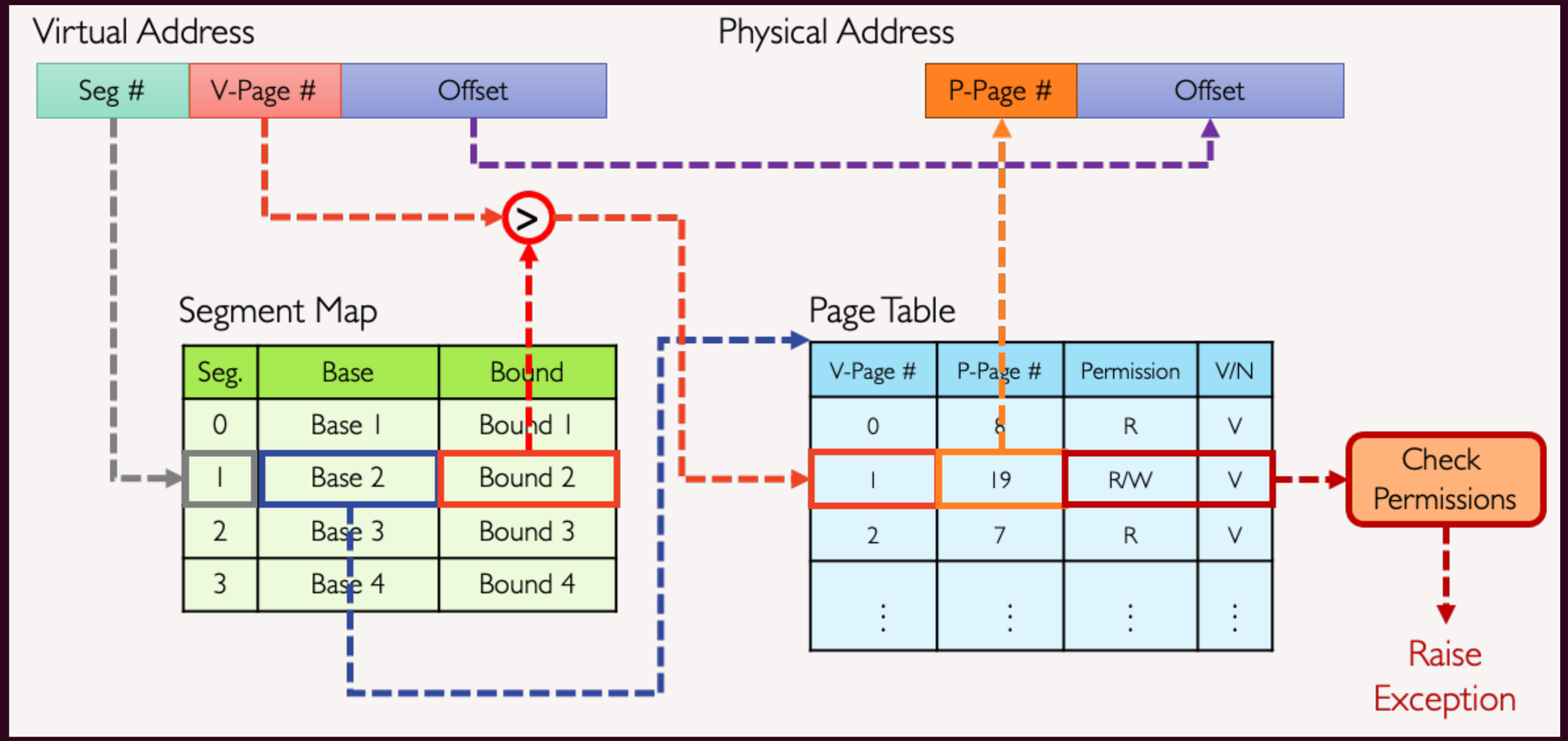


Total size of page table ≈ number of pages used by program in **physical memory**

hash function加速索引，只能存放在受限的范围，可以附带链表（支持多个进程共享）

段页（Segments and Pages）

- 段表和页表可以混合使用：



地址翻译比较

方法	优点	缺点
分段	快速上下文切换（段映射由CPU维护）	外部碎片
单级页表	无外部碎片 快速且易于分配	表的规模巨大 内部碎片
多级页表	表大小约为虚拟内存中的需要用的页面数量 快速且易于分配	每次页面访问涉及多次内存引用
倒排页表	表大小约为物理内存中的页面数量	需要复杂的哈希函数 页表没有缓存局部性

交换

交换（Swapping）

- 如果没有足够的空间来容纳所有进程怎么办?
  - 一个进程可以被暂时换出内存到后备存储
  - 然后在需要继续执行时再换回内存
  - 进程的总物理内存空间可以超过物理内存

交换

- 交换时间的主要部分是磁盘传输时间（与交换的内存量直接成正比）。
  - 系统维护一个就绪队列，里面是内存镜像在磁盘上的准备运行的进程。
  - 如果要放到CPU上的下一个进程不在内存中，需要换出一个进程并换入目标进程，这时上下文切换时间可能会非常高。
    - 比如一个100MB的进程交换到传输速率为50MB/秒的硬盘上，那么交换出（或交换入）时间 = 2秒
  - 如果进程正在等待I/O操作（I/O缓冲区在进程的地址空间中）？
    - 交换会产生负面影响（I/O失败或数据丢失）！一般涉及I/O操作的页需要被锁定在内存中，防止在操作完成前被换出。

交换

- 标准交换在现代操作系统中不再使用，但变异版本很常见（例如，Linux和Windows）。
  - 交换通常是禁用的。
  - 当分配的内存超过阈值（当可用内存极低时）才会启动交换。
  - 一旦内存需求减少到低于阈值时，交换再次被禁用。

交换

- 移动操作系统通常不支持交换。
  - 基于闪存（而不是磁盘）：空间较小；只能写入有限次数。
- 取而代之的是使用其他方法来释放内存：
  - iOS要求应用程序自愿释放已分配的内存。
    - 只读数据可以被丢弃并在需要时重新加载。
    - 如果应用程序未能释放内存，操作系统可以终止它们。
  - 类似地，Android在内存不足时会终止应用程序，但首先会将应用程序状态写入闪存以便快速重启。

如果单个进程本身超过物理内存?

- 计算机可以运行非常大的单个程序
  - 远远大于物理内存
  - 只要“活跃”的占用内存量适合物理内存，就可以运行
- 然而交换无法做到这一点
- 一个想法：不是所有内容都需要同时加载进内存!

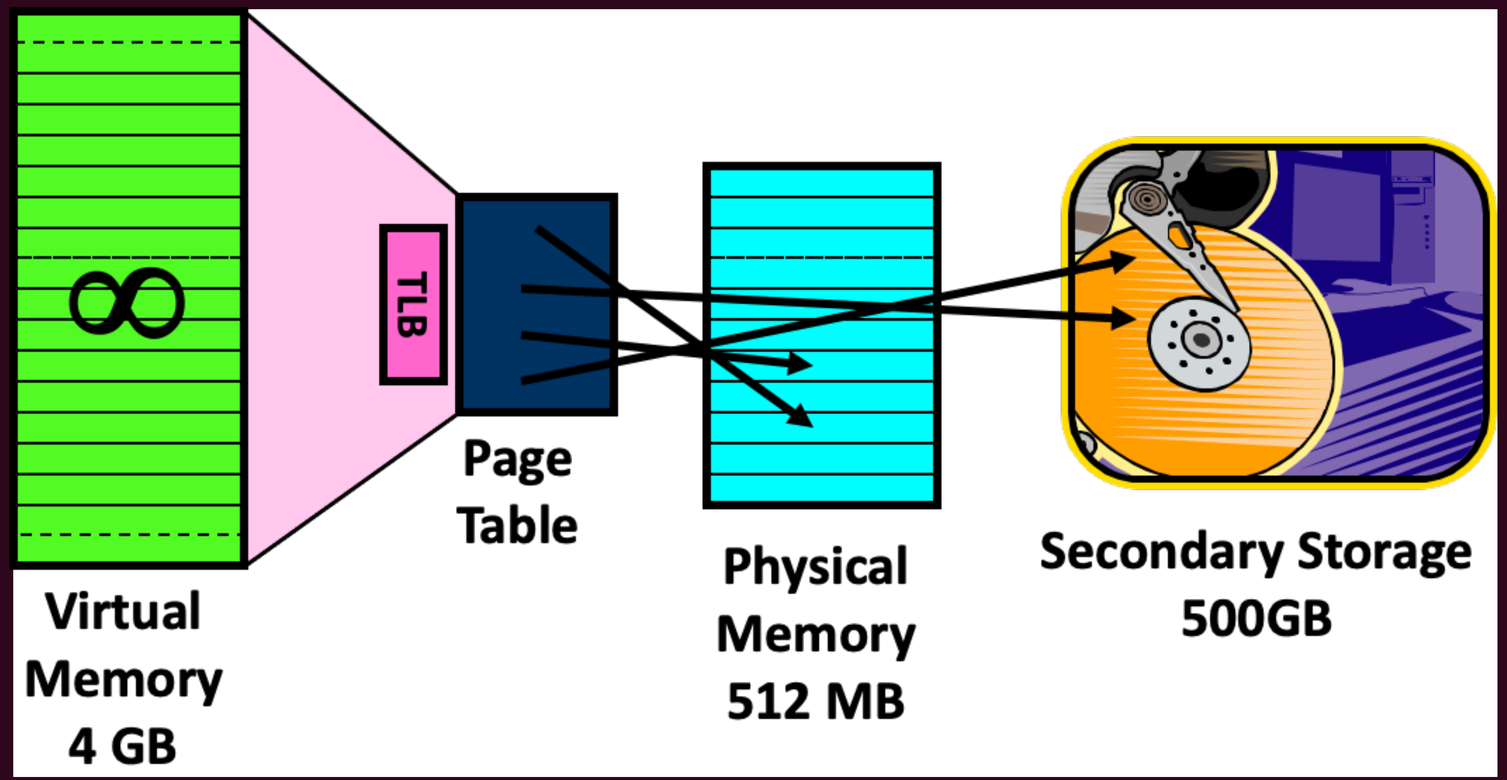
部分内存驻留

- 程序中的错误处理代码不会在每次运行时都被使用
  - 因此不需要它在整个持续时间内占用内存
- 数组可能分配得比所需要的大
  - int players[MAX_INTEGER];
- 程序地快速启动
  - 无需在运行之前加载整个程序

虚拟内存

虚拟内存

- 每个进程都有一个大地址空间的幻觉。
- 支持多个并发运行的进程使用大虚拟地址空间：只将常用的页面保留在内存中（此时内存可以看成是所有pages的一个cache）



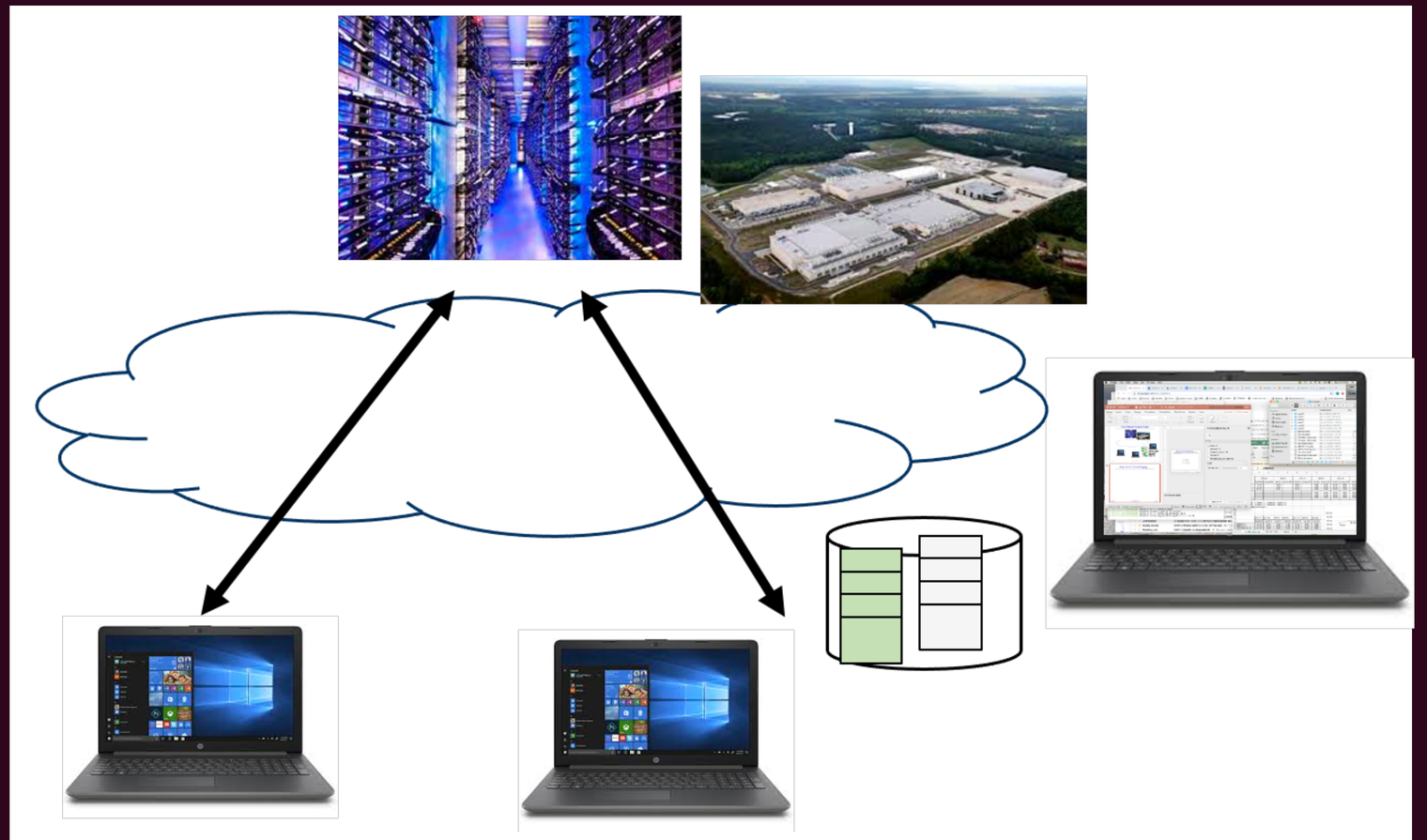
- 透明的间接层（页表）
  - 支持物理数据的灵活放置
    - 数据可以在磁盘上甚至是网络中的某处
  - 数据位置的变化对用户程序是透明的

虚拟内存

- 具备“**部分**”加载程序的执行能力
  - 程序不再受物理内存限制（可以运行无法完全放入物理内存的程序）
  - 每个程序在运行时占用更少的内存（可以同时运行更多的程序）
  - 加载或交换程序到内存中所需的I/O更少（每个程序启动更快）

近乎无限的内存！

- 如今：进程所需的运行部分甚至都不需要在“本地”！



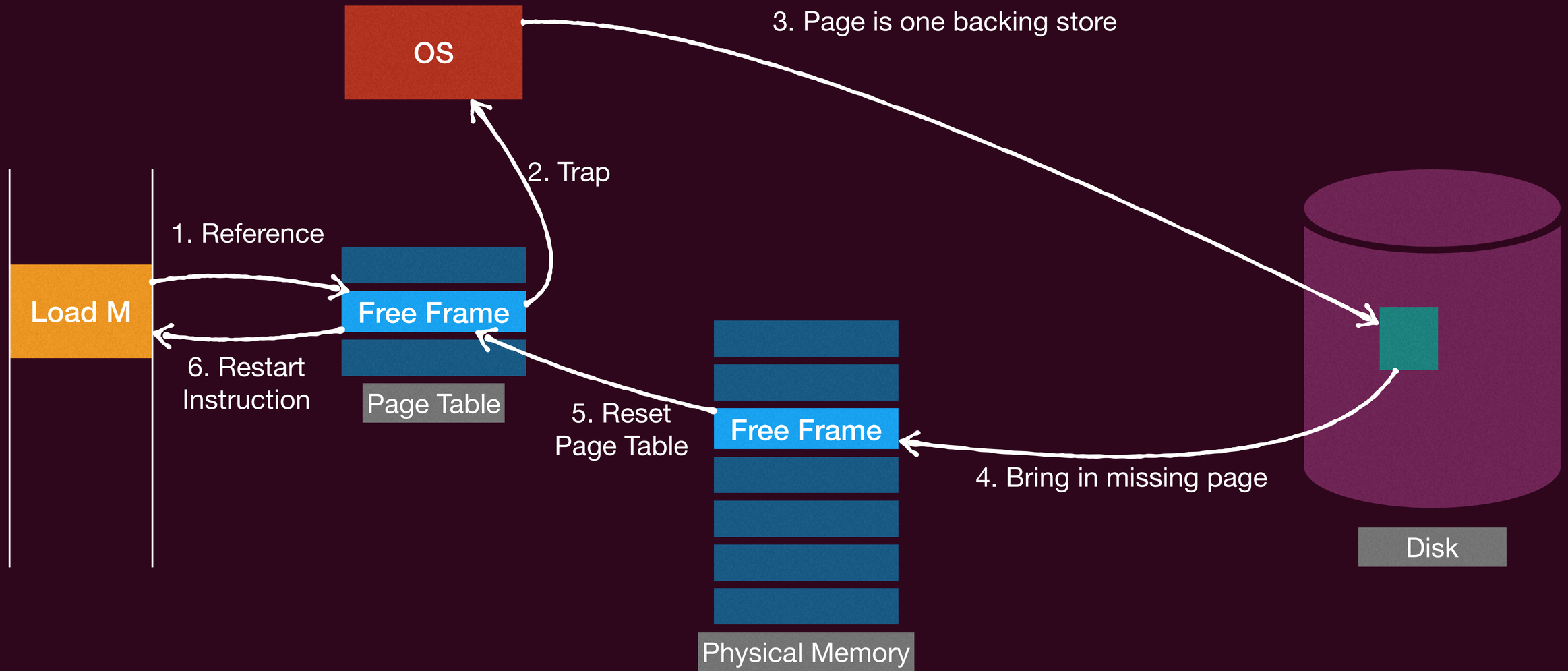
虚拟内存

- 使用存在/不存在位（在页表项中）来跟踪哪些页面存在于物理内存中。
- 当程序引用其地址空间的一部分时：
  - 如果页面在物理内存中，则直接进行地址转换
  - 如果不在，则发生**缺页异常（Page Fault）**，操作系统被调用来处理该异常：
    - 检测并将页面加载到内存中，然后重新执行指令(引用该地址空间的指令)

缺页异常(Page Fault)

- 缺页异常
  - CPU控制流传递
  - 提前注册缺页异常处理函数
- x86_64
  - 异常号 #PF (13)，错误地址在CR2

处理缺页异常


OS
3. Page is one backing store
2. Trap
1. Reference
Load M
Free Frame
Page Table
6. Restart
Instruction
5. Reset
Page Table
Free Frame
Physical Memory
4. Bring in missing page
Disk

具体流程

- 1. 硬件陷入内核，进行Context-Switch（将程序计数器保存在栈上，保存通用寄存器和其他易失性信息。）
- 2. 系统发现了这个事件是一个缺页异常，尝试确定所需的虚拟页面。
- 3. 一旦知道引发缺页异常的虚拟地址，系统检查地址是否有效，并且保护是否与访问一致。
- 4. 找到一个空闲（干净的）帧。
  - 如果没有空闲帧，则运行页面置换以选择一个victim(受害者)。
  - 如果所选帧是脏的，则将页面安排转移到磁盘，进行上下文切换，暂停引发异常的进程。

具体流程

- 5. 一旦帧变为干净状态，系统查找所需页面的磁盘地址，并安排磁盘操作将其调入（引发缺页异常的进程仍处于暂停状态）
- 6. 当磁盘中断指示页面已经到达时，更新页表，并将帧标记为正常状态
- 7. 将引发缺页异常的指令恢复到其原始状态，并重置程序计数器
- 8. 引发缺页异常的进程被调度，Context-Switch回去

性能

- 处理缺页异常的三个主要活动：
  - 服务中断：一般只需要几百条指令
  - 读取页面：需要大量时间
  - 恢复进程：需要少量时间
- 缺页错误率（$0 \leq p \leq 1$）：进程在内存中发现缺页异常的速率。
- 有效访问时间（Effective Access Time, EAT）：$(1-p)$ × 访存时间 + p × 缺页异常开销

性能

- 例子：
  - 内存访问时间 = 200 ns
  - 平均缺页异常服务时间 = 8 ms
  - EAT = (1 – p) x 200 + p x 8,000,000 = 200 + p x 7,999,800 ns
  - 如果 1,000 次访问中有一次引发缺页异常 (p = 0.001), 那么 EAT = 8200 ns (减速了 40 倍)
  - 如果想要性能降级 < 10%
    - 200 + p x 7,999,800 < 220
    - p < 0.0000025 (每 400,000 次内存访问中不到一次缺页异常)

缺页异常的其他用法

- 除了作为构建虚拟内存的一个机制外，缺页异常还有其他巧妙用法
  - Copy-On-Write
  - Zero-filled-on-Demand
  - memory-mapped files

Zero Pages

- 非常特殊的写时复制情况
  - 按需零填充 (Zero-Filled On Demand, ZFOD)
- 许多进程页是“空白”的
  - 所有的 bss（通常是指用来存放程序中未初始化的全局变量的一块内存区域）
  - 新的堆页面
  - 新的栈页面

Zero Pages

- 按需零填充实现：
  - 有一个系统范围内的全零帧
    - 所有的全0页帧都指向它
    - 标记为只读
    - 读取零是自由的
    - 但写入会导致缺页错误以及克隆操作

内存映射文件（Memory-Mapped Files）

- 将文件内容映射到进程的地址空间的机制。通过内存映射文件，可以将文件视为一块内存区域，而不需要显式地使用 `read()` 和 `write()` 的接口：
  - `mmap(add, len, prot, flags, fd, offset)`
  - 当进程访问映射区域时，如果所访问的页面尚未在内存中，则会发生页面错误。此时，操作系统会将相应的文件内容读取`read()`到内存中，以满足进程的访问请求。
  - 进程对映射区域的写操作会导致页面被标记为脏页，并在必要时通过 `write()`系统调用将页面内容写回文件。
  - 如果多个进程使用相同的 `mmap()`调用映射了相同的文件，它们将共享同一份文件内容，即它们的映射区域指向同一块内存区域，从而实现了共享内存的效果。

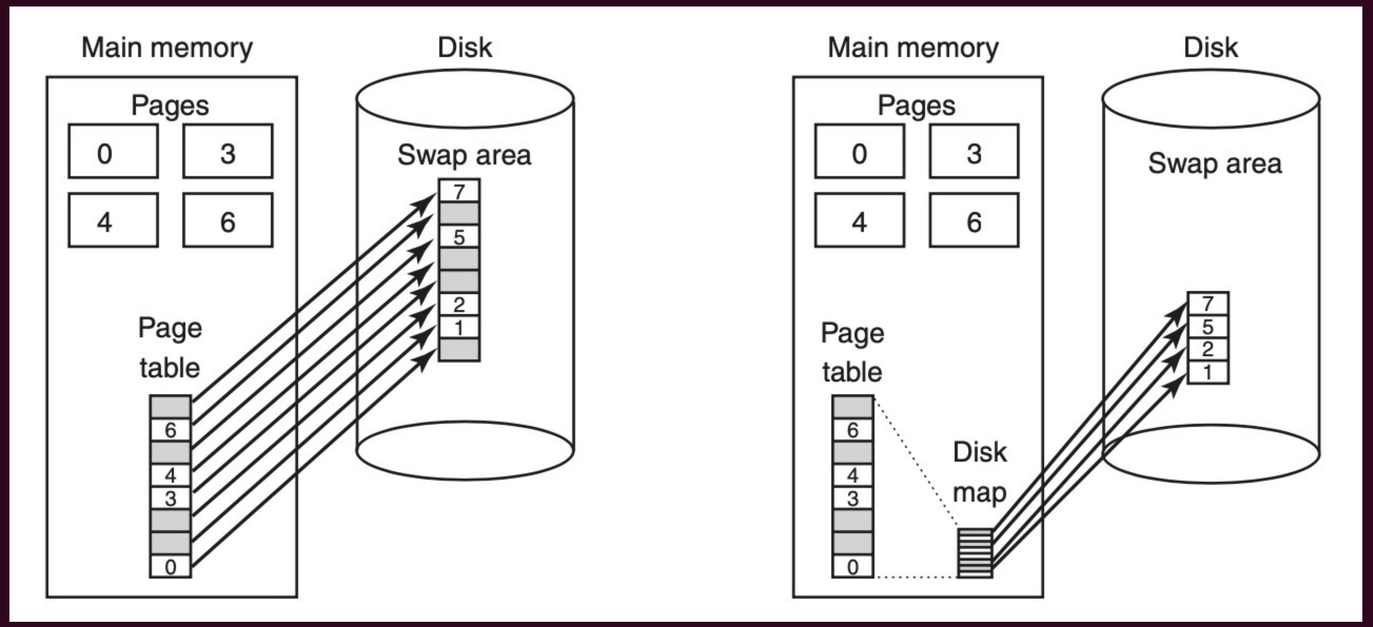
页面置换

何时将页面调入内存?

- 按需分页（Demand Paging） - 出现缺页异常后进行掉入页面
  - 最简单的方法
  - 为了提高效率，掉入是异步进行的：
    - 中断处理程序应快速响应 - 只需启动磁盘I/O并阻塞该进程，让其他进程运行。
- 预取(Prefetching)
  - 猜测即将使用哪些页面，因此提前将其调入内存
    - 往往是基于历史的缺页记录来预测

备份存储

- 当页面被换出时应该放在哪里?
  - 使用单独的交换空间（而不是文件系统）
    - 交换空间的I/O速度比文件系统的I/O快
    - 比文件系统需要更少的管理



页面置换(Page Replacement)

- 如果没有空闲的帧会发生什么?
  - 页面置换：找到内存中的一个页面，但实际上并未使用，将其换出。
    - 选择一个受害者帧（victim frame）进行驱逐。
    - 将所需的页面调入（新）空闲帧。
  - “页面置换”的问题也会出现在许多其他领域，比如内存缓存。

页面置换

- 当一个页面必须被换出时...
  - 更新页表：找到所有引用旧页面的页表项（因为帧可以共享），并将每个设置为不可见
  - 移除任何TLB条目
    - TLB关机：在多处理器系统中，必须从所有处理器的TLB中消除TLB条目
  - 将页面写回磁盘（如果需要，页表项中的脏位）
  - 重新启动引发陷阱的指令：（需要备份指令）

锁定页面

- 如果我们不想将某些页面换出怎么办?
  - 将页面固定到内存中以锁定
  - 有时必须将页面锁定到内存中
    - 总是将（部分）内核页面放入物理内存中
    - 用于从设备复制文件的页面必须锁定，以防止被选择用于驱逐（I/O）
    - 一个低优先级的进程交换入一个页面，然后一个高优先级的进程抢占并请求一个新的帧?
  - 需要小心使用

页面buffering

- 操作系统会等到内存完全满了吗?
  - 保留一组空闲帧（Buffering）以确保在需要时总有可用的帧在合适的时候
- 此外有一个交换页面的守护进程（后台进程）定期运行（类似于调度程序）
  - 如果空闲物理帧的数量 < "低水位标记"，则换出一些页面，直到数量达到"高水位标记"
  - 系统会一次性换出许多页面，以从低水平达到高水平。
    - 这样做是因为将大块数据写入磁盘更有效（批量传输）
    - 需要维护一个修改页面的列表，将页面写入其中并设置为非脏（Linux中的pdflush）

页面buffering

- 页面交换的守护进程可以以低优先级调度
  - 利用空闲时间准备未来的工作
- Linux 交换守护进程是一个名为 `kswapd` 的进程

页面置换策略

- 哪个帧应该被替换?
- 目标是实现最低的缺页异常率（尽量减少从磁盘获取页面的次数）。
  - 如果选择一个不常被使用的页面，系统性能会更好。
  - 如果删除一个频繁使用的页面，它可能很快就需要被重新调入。

页面置换策略

- 哪个帧应该被替换?
  - 通过在特定的内存引用串上运行算法（模拟），计算缺页异常的数量（命中率/未命中率）来评估算法。
  - 结果取决于可用帧的数量。
    - 一个足够大（小）的帧数将具有高（低）的命中率。

First-In-First-Out (FIFO)

- 替换最老的页面
  - 丢弃可能没有人再感兴趣的页面
  - 使用一个FIFO队列来跟踪页面的老化（aging）程度

随时间的页面引用

内存中的页面

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	**5**	5	5	5	**4**	4
	2	2	2	2	2	2	**1**	1	1	1	**5**
		3	3	3	3	3	3	**2**	2	2	2
cold-start miss			**4**	4	4	4	4	4	**3**	3	3
# Page Fault = 10											

问题：最老放入的可能是需要被频繁使用的！

Belady异常(Belady's Anomaly)

- Belady异常：增加帧数反而可能会降低命中率（帧数越少，缺页异常越少）😱!
  - 给定4个帧，有10个缺页异常
  - 给定3个帧，有9个缺页异常

随时间的页面引用

内存中的页面

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	**4**	4	4	**5**	5	5	5	5	5
	2	2	2	**1**	1	1	1	1	**3**	3	3
		3	3	3	**2**	2	2	2	2	**4**	4
# Page Fault = 9											

Belady异常

- Belady's Anomaly会发生在**任何**页面替换算法中（比如随机替换），只要它不遵循“栈算法”属性（stack algorithm property）。
  - 之后的LRU（最近最少使用）和Optimal（最佳）算法始终遵循栈算法属性，因此它们永远不会受到Belady's Anomaly的影响。
- 栈算法属性确保当页面帧数量增加时，先前存在的页面集合始终是当页面帧更多时存在的页面集合的子集。换句话说，随着页面帧数的增加，先前存在的页面应始终保留在内存中。
  - 如果没有这个性质，则可能会导致先前存在于内存中但由于增加页面数反而被删除这个页面，如果正好这个页面是一个需要频繁使用的页面，那么...

Optimal (MIN)

- 替换那些在最长时间内不会被使用的页面
  - 尽可能推迟不愉快的换页事件
  - 但这个方法只存在理想中，我们无法预知未来！
    - 因此通常用于衡量算法的性能（和最优的差距有多大）

随时间的页面引用

内存中的页面

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	1	**4**	4
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	3	3	3	3	3	3
			4	4	4	**5**	5	5	5	5	5
# Page Fault = 6											

Least Recently Used (LRU)

- 替换那些在最长时间内没有被使用的页面
  - 使用历史而不是未来：很长时间没有被使用的页面可能会保持长时间未使用
  - 基于局部性原理

随时间的页面引用

内存中的页面

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	1	1	1	1	1	**5**
	2	2	2	2	2	2	2	2	2	2	2
		3	3	3	3	**5**	5	5	5	**4**	4
			4	4	4	4	4	4	**3**	3	3
# Page Fault = 8											

Least Recently Used (LRU)

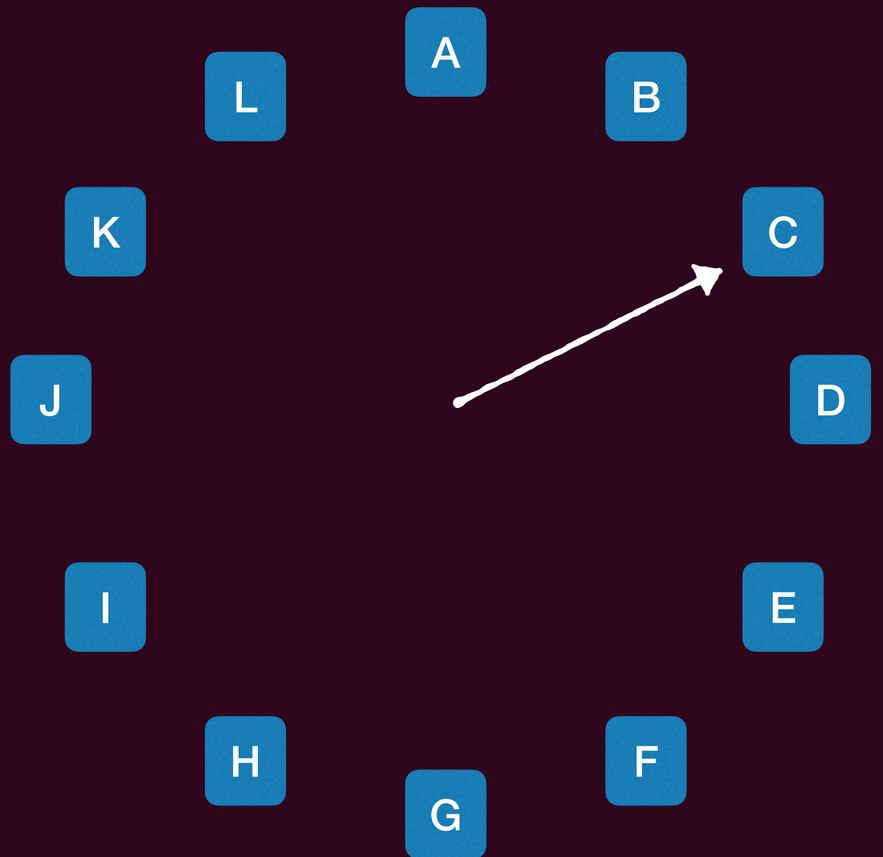
- 为了跟踪哪些页面最近最少被使用（将每个页面的最后一次使用时间与之关联），操作系统必须在每次内存引用时进行一些计时工作。
- 计数器实现
  - 每个页面都有一个对应的计数器项。
  - 每次页面被引用时，通过硬件将时钟寄存器的值复制到计数器中。
  - 当需要更换页面时，查看计数器以找到最小的值。
- 开销太大！

近似LRU (Approximating LRU)

- 通过引用位来近似LRU（即二次机会算法，Second Chance）：寻找一个在最近的时钟周期内没有被引用的老页面。
  - 系统中每个页面有一个引用位（R）。
  - 每当引用页面（即读取或写入），引用位被设置为1（由硬件完成）。
  - 如果要被替换的页面：
    - R = 1：将引用位设置为0；将其放在FIFO队列的末尾；并检查下一个页面。
    - R = 0：替换它。
  - 如果所有页面都被引用，那么第二次机会等于FIFO。

二次机会算法

- 高效替代Second Chance算法：时钟算法
  - 将所有页面帧以时钟形式放在一个循环列表中（避免在队列上移动页面）



发生缺页异常时，检查当前指向的页面。
采取的操作取决于R位的值：

- R = 0：驱逐该页面
- R = 1：将R=0并将指针向前移

N次机会时钟算法

- 第N次机会算法：给每个页面N次机会
  - 操作系统为每个页面保持一个计数器：计数扫过的次数
  - 发生页面错误时，操作系统检查使用位：
    - 1：清除使用位并清除计数器（在最后一次扫过时被使用）
    - 0：增加计数器；如果计数=N，则替换页面
  - 这意味着时钟指针需要扫过N次而页面没有被使用后才会替换该页面

N次机会时钟算法

- 我们如何选择N值?
  - 选择较大的N值？更接近LRU（最近最少使用）算法
  - 为什么选择较小的N值？更高效
- 脏页面怎么办?
  - 替换脏页面需要额外的开销，所以在替换前给脏页面一次额外的机会?
    - 一种方法：
      - 对于干净页面，使用N=1
      - 对于脏页面，使用N=2（当N=1时写回磁盘）

Not Recently Used，NRU算法

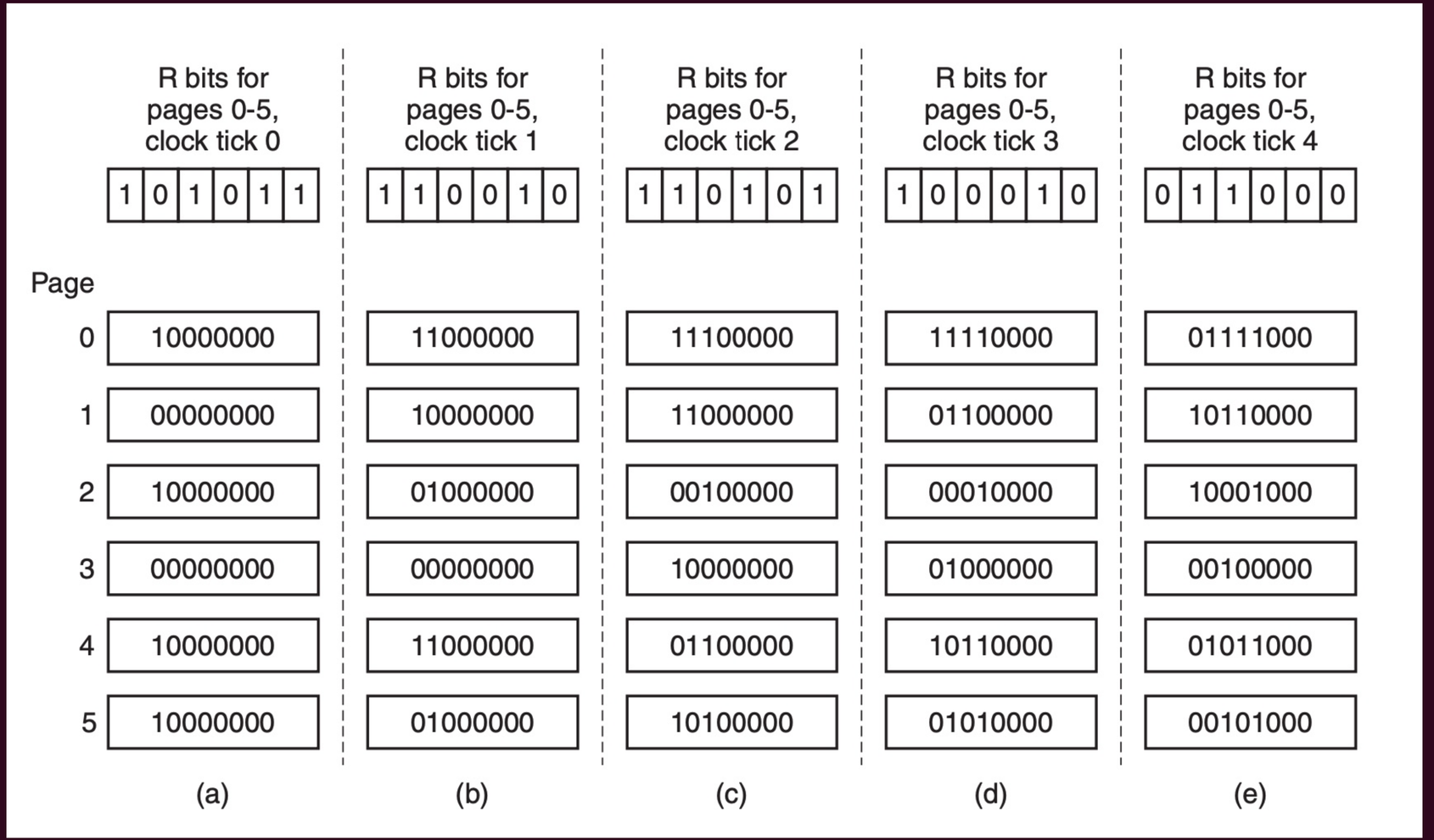
- 通过两个状态位—引用位和修改位来近似LRU（即Not Recently Used，NRU算法）：为那些被**修改**的页面赋予更高的优先级以减少I/O负担（增强的第二次机会）。
  - (0, 0)：既不是最近使用的也没有被修改过（最适合替换）
  - (0, 1)：不是最近使用的但被修改过（不太理想，需要在替换前写出）
  - (1, 0)：最近被使用但是干净的（但可能会很快再次被使用）
  - (1, 1)：最近被使用且被修改过（可能会很快再次被使用，并且需要在替换前写出）
- 从最低编号的非空类中随机移除一个页面（可能需要多次在循环队列中搜索）

老化算法Aging

- 通过额外的引用位来近似LRU（即Aging算法）：保持一个软件计数器来追踪每个页面被引用的次数，并替换计数最小的页面。
  - 每个页面关联一个软件计数器
  - 在每个时钟中断时
    - 对每个页面的计数器进行右移1位操作（类似于时间/2）
    - 并将R位添加到最左边（类似于新引入的刷新了时间）
  - 较大的值表示最近使用的页面
    - 选择01110111页面进行替换，而不是11000100页面

老化算法Aging

- Aging算法



抖动（Thrashing）

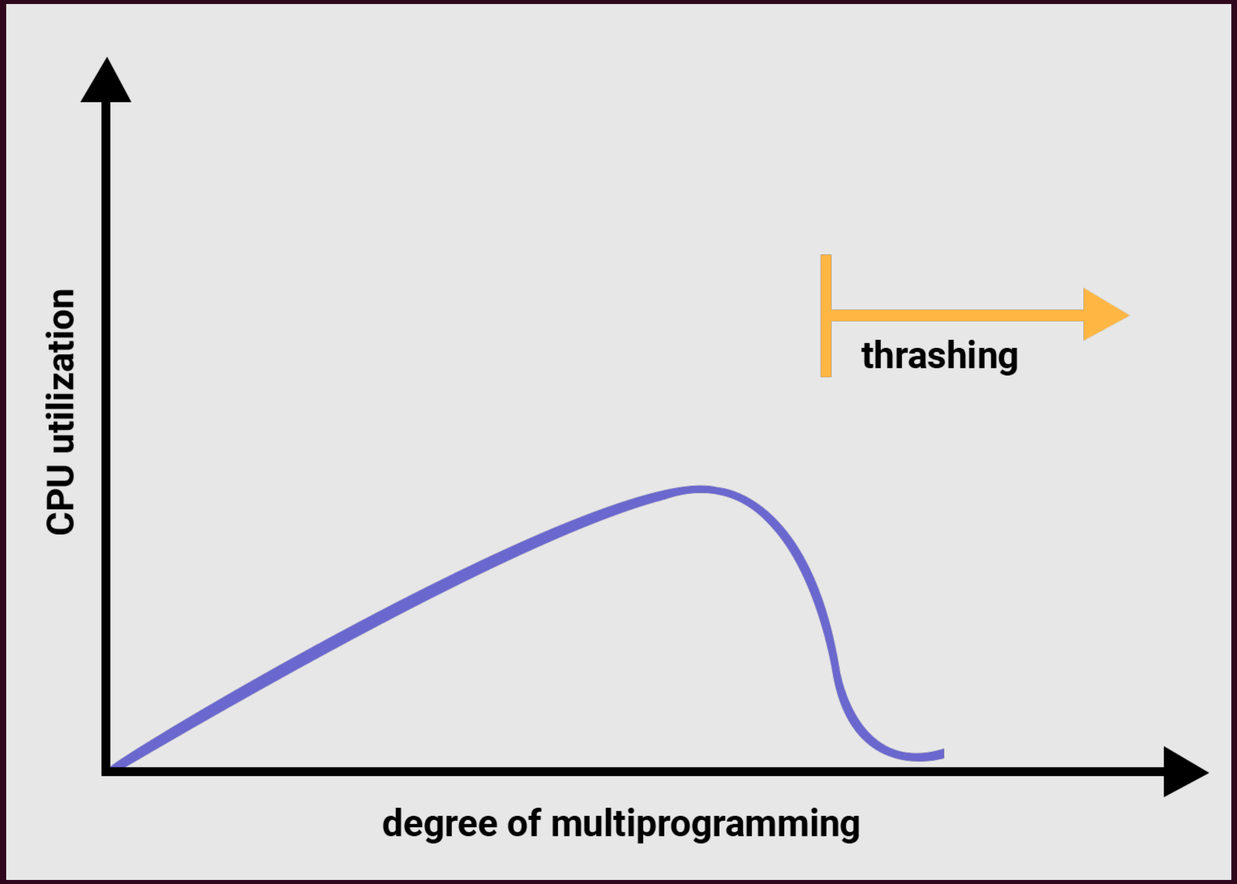
- 如果一个进程没有足够的页面，缺页率可能会非常高。
  - 假设引用页面编号顺序为0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3，但有三个可用帧，当运行FIFO / LRU…… 此时，
- 此时大部分 CPU 时间都被用来处理缺页异常
  - 等待缓慢的磁盘 I/O 操作
  - 仅剩小部分的时间用于执行真正有意义的工作

抖动（Thrashing）

- 调度器甚至还会造成问题加剧
  - 等待磁盘 I/O导致CPU利用率下降
  - 调度器载入更多的进程以期提高CPU利用率
    - 本来物理页面就不足的局面“雪上加霜”
  - 触发更多的缺页异常、进一步降低CPU利用率、导致连锁反应

抖动

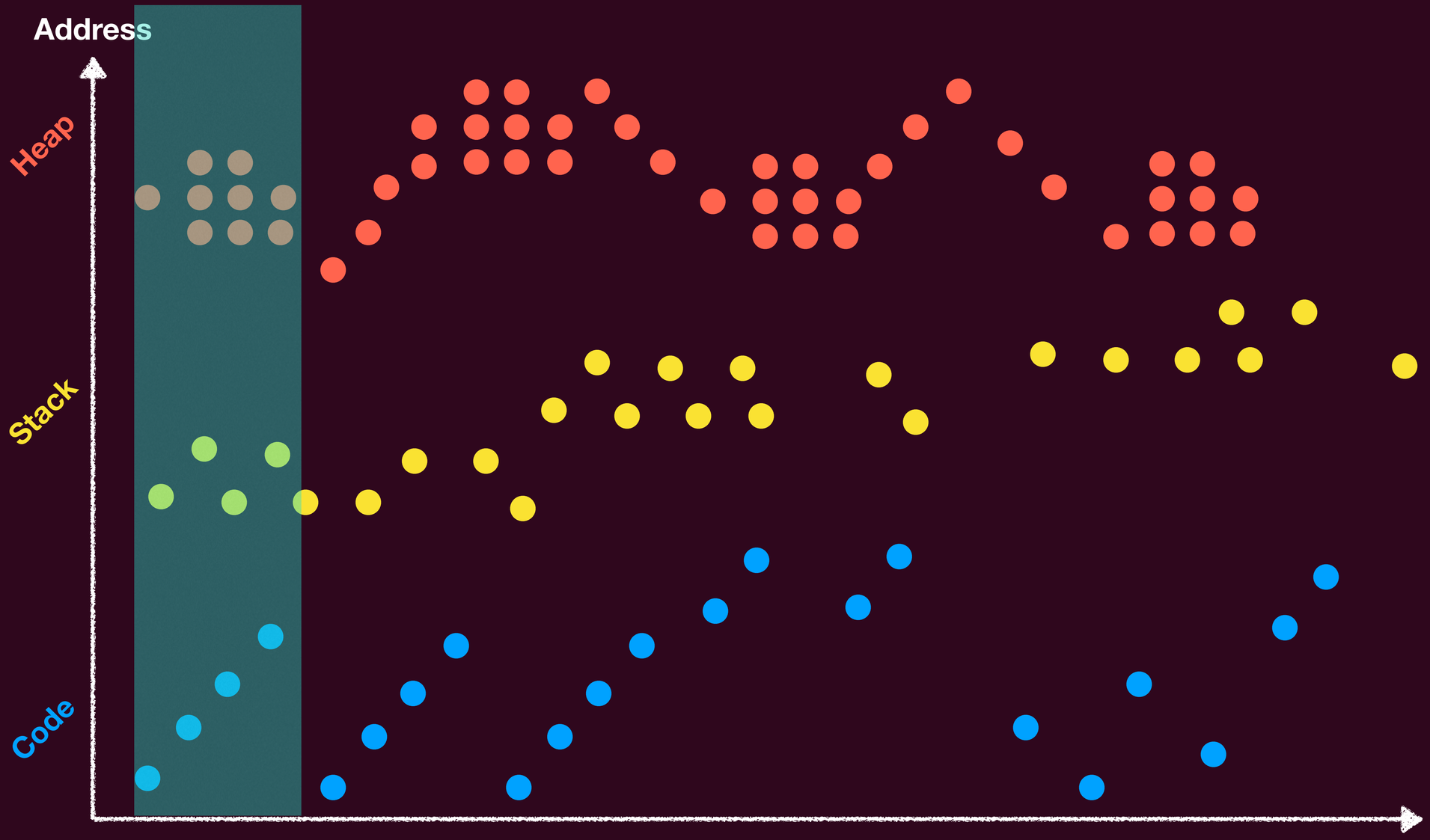
- 抖动（也叫颠簸）：一个进程花费所有时间在页面间进行交换（大多数引用导致缺页异常）



局部性原理（Principle of Locality）

- 时间局部性（Temporal locality）：相同的内存位置在不久的将来会再次被访问。
- 空间局部性（Spatial locality）：未来将会访问附近的内存位置。
- 当一个进程执行时，它会从一个局部转移到另一个局部

Principle of Locality



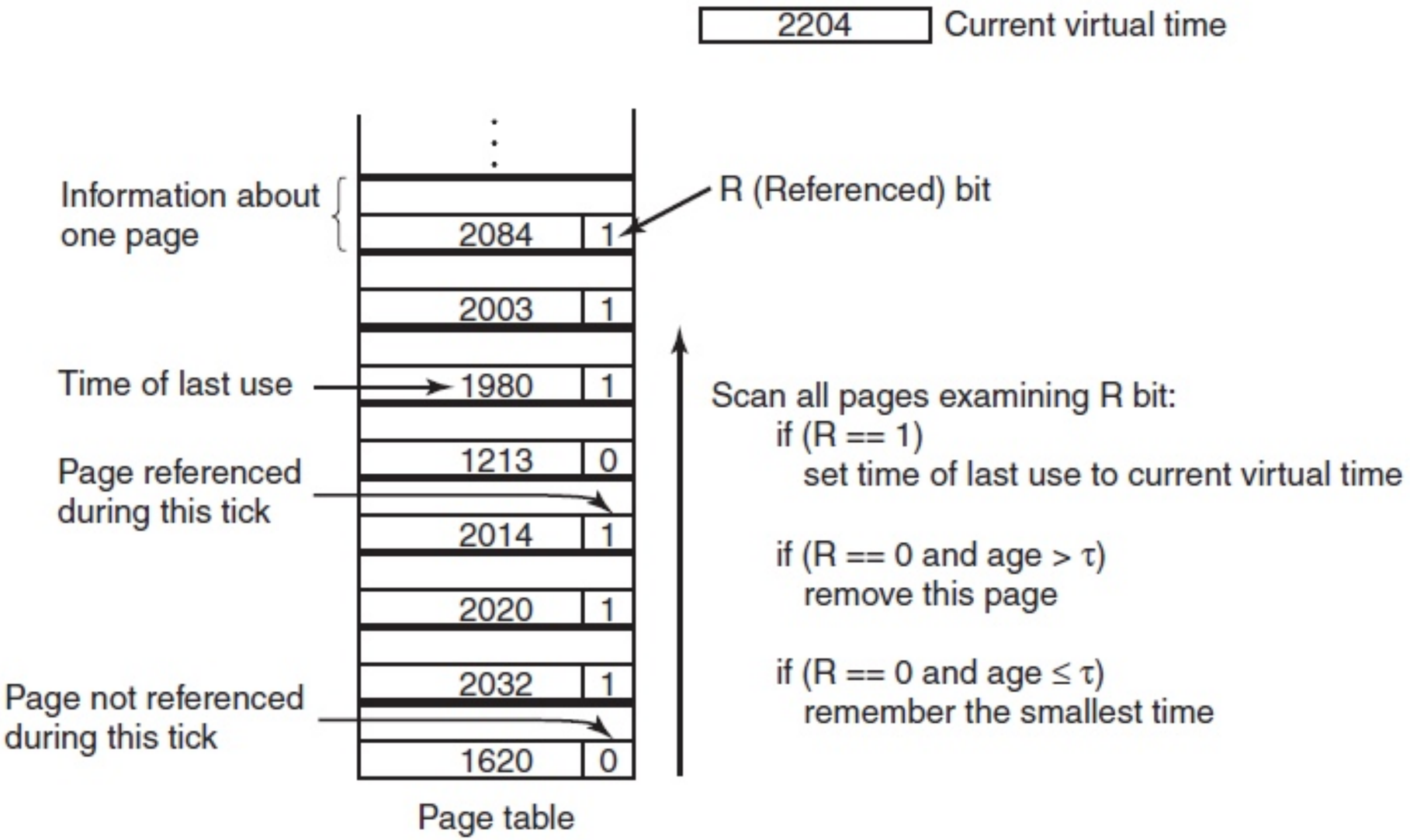
工作集（Working Set）模型

- 工作集（Working Set）：
  - 其在时间段(t-x,t)内使用的内存页集合也被视为其在未来(下一个x时间内)会访问的页集
  - 如果整个工作集都在内存中，那么进程将运行而不会引起太多缺页异常，直到它进入另一个执行阶段。
  - 如果可用内存太小，无法容纳整个工作集，则会发生抖动。
- all-or-nothing模型
  - 进程工作集要不都在内存中，否则全都换出
  - 需要跟踪每个进程的工作集，并确保在运行之前将其加载到内存中。
  - 大大降低缺页异常率

跟踪工作集w(t, x)

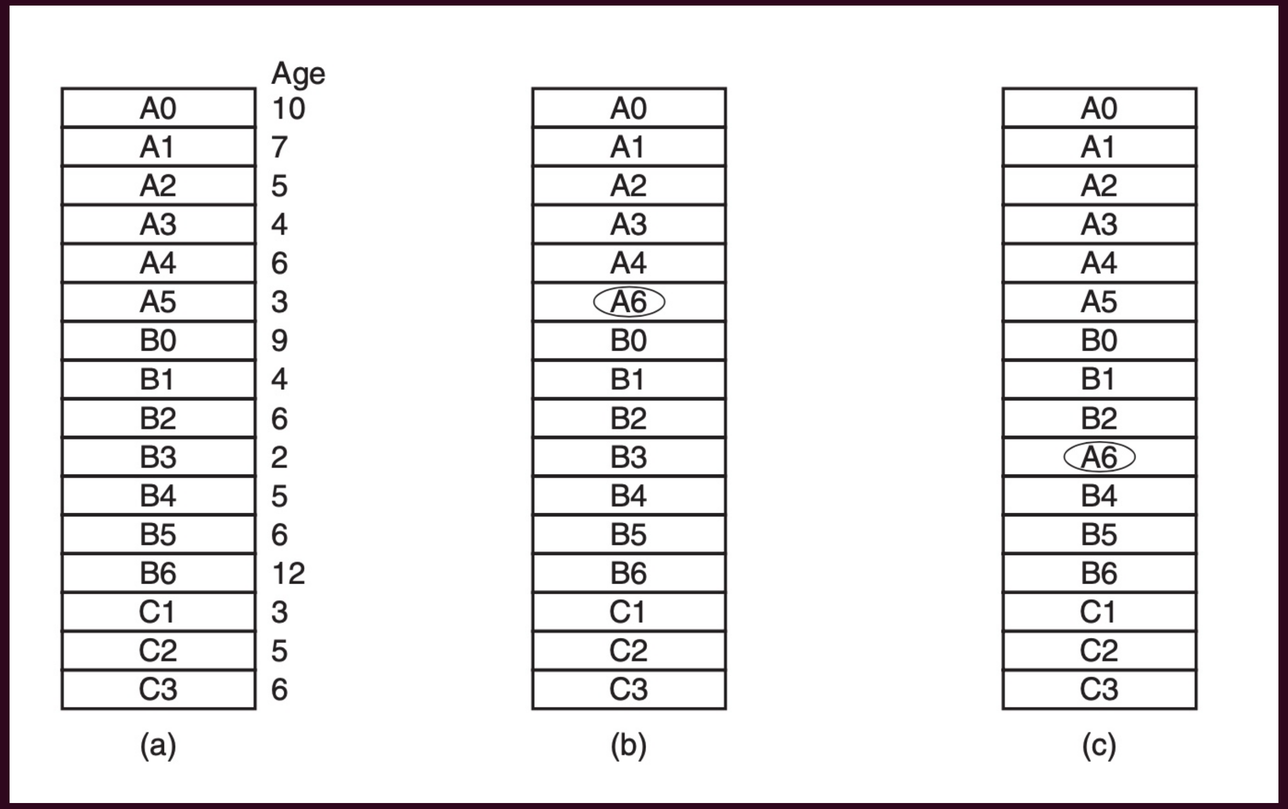
- 工作集时钟中断固定间隔发生，处理函数扫描内存页
  - 访问位为1则说明在此次tick中被访问， 记录上次使用时间为当前时间
  - 访问位为0(此次tick中未访问)
    - Age = 当前时间 – 上次使用时间
    - 若Age大于设置的x，则不在工作集
  - 将所有访问位清0
    - 注意访问位(access bit)需要硬件支持

工作集算法


2204
Current virtual time
Information about one page
R (Referenced) bit
2084
1
2003
1
Time of last use
1980
1
1213
0
Page referenced during this tick
2014
1
2020
1
2032
1
Page not referenced during this tick
1620
0
Page table
Scan all pages examining R bit:
if (R == 1)
set time of last use to current virtual time
if (R == 0 and age > τ)
remove this page
if (R == 0 and age ≤ τ)
remember the smallest time

本地和全局策略

- 替换页帧是全局（从所有进程中选择）还是本地（只挑自己的页帧）？



Select a frame of the smallest age for A6

Local Replacement

Global Replacement

本地和全局策略

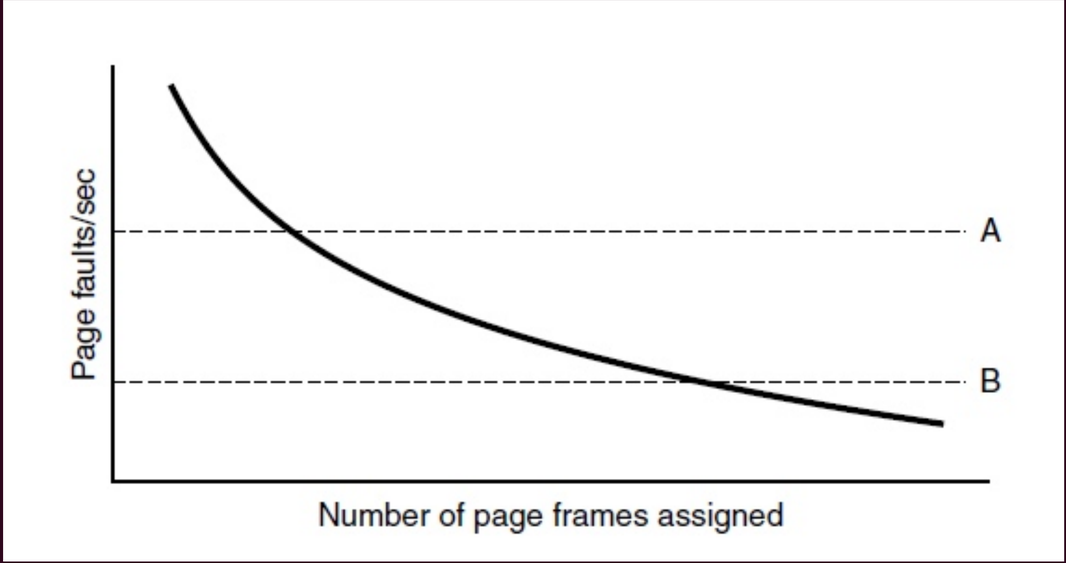
- 本地页面置换：每个进程从其分配的帧集中选择受害者
  - 每个进程的帧数固定分配
  - 每个进程的性能更加一致
  - 但可能导致内存利用不足
- 全局页面置换：从分配给任何进程的帧中选择受害者
  - 每个进程的帧数可变
  - 吞吐量更大，因此更为普遍

帧分配

- 采用全局页面置换时，操作系统必须不断决定为每个进程分配的页面帧数：
  - 平均分配：为每个进程分配相等的份额。
  - 比例分配：根据进程大小进行分配（需要为每个进程分配一定数量的帧）。
  - 按照优先级分配（优先级高的）
- 这些静态的帧数分配无法解决不断变化的动态需求

帧分配

- 缺页异常频率（PFF）算法用于帧分配
  - 缺页异常率 = 每秒平均缺页数
  - 分配帧以使进程的PFF相等
  - 仍然需要临时将某些进程完全交换出去（某些进程需要更多内存，但没有进程需要更少内存）



如果实际速率太低，则进程丢失帧
如果实际速率太高，则进程获得帧

Put it all together

操作系统对分页的支持

- 进程创建
  - 确定程序和数据的大小并创建页表
  - 为页表分配空间并初始化
  - 为磁盘上的交换区域分配空间并初始化
  - 在进程表中记录有关页表和交换空间的信息

操作系统对分页的支持

- 进程执行
  - 为新进程重置内存管理单元（页表基址寄存器）
  - 上下文切换：清除TLB（除非它是带有标记的）
  - 可选地，将进程的一些或所有页面调入
- 页面守护进程（Page Daemon）
  - 大部分时间处于休眠状态，但定期唤醒以检查内存状态，并主动准备待驱逐的页面

操作系统对分页的支持

- 缺页异常
  - 找到所需的页面，并在磁盘上定位该页面
  - 找到一个可用的页面帧来放置新页面（必要时替换旧页面）
  - 将所需的页面读入页面帧
  - 备份程序计数器以再次执行指令
- 进程终止
  - 释放页表、页面和交换空间
  - 共享页面只能在使用它们的最后一个进程终止时释放

程序优化

- 分页是一个“操作系统问题”吗?
  - 程序员能减少工作集的大小吗?
- 局部性取决于数据结构
  - 数组鼓励顺序访问
    - 多次引用同一页
    - 可预测的访问下一个页面
- 编译器和链接器也能帮忙
  - 不要把一个函数分散到两页上
- 这些优化方式的效果可能非常显著

总结

- 内存的抽象
  - 屏蔽物理的局限（无限空间、连续、独享）
- 实现机制：地址翻译
  - 需要硬件和OS配合
- 底层内存的管理：连续分配、分段、分页
  - 利用TLB缓存
- 实现虚拟内存的工程问题：
  - 换页、工作集

阅读材料

- [OSTEP]13、14、15、16、17、18、19、20章
- [现代操作系统]第4章