Chapter 09 虚拟内存

- **物理地址(Physical Address)**: 硬件物理内存中的每个字节的地址.
- **虚拟地址(Virtual Address)**: 虚拟内存中的地址.
- **地址空间(Address Space)**
- **虚拟内存(Virtual Memory, VM)**:

  存放在磁盘上的, 包含N个字节的 `字节数组` (连续的).

  每个字节都有一个唯一的虚拟地址, 作为到该数组的索引.

  磁盘上的数组**缓存在主存中**, 磁盘和主存之间以 `页` 作为缓存的基本传输单元.

  > `页` 其实就是高速缓存中的 `块` 在虚拟内存的情形下的特殊称呼.

  这个 `页` 被称为**虚拟页(Virtual Page, VP)**, 每个页具有固定大小 $P = 2^p$, 一般而言为4KB.

  类似地, 物理内存被分割为**物理页(Physical Page, PP)**, 大小同样为 $P = 2^p$.
- 整个虚拟内存中有三种 `虚拟页` :
  - `未分配` : VM还没有分配/创建的页, 因此没有任何数据和这些页相关联. 因此**不占用磁盘**空间.
  - `未缓存` : 已分配的页, 在磁盘中占用空间, 但并没有缓存在物理内存中.
  - `缓存的` : 已分配且缓存在物理内存中的页.
- 我们用 **`SRAM` 缓存**来称呼 `L1/L2/L3` Cache;

  用 **`DRAM` 缓存**来称呼虚拟内存系统的缓存, 它负责在主存中缓存磁盘中的虚拟页.
- 由于磁盘的速度远远低于DRAM, 为了效率, DRAM缓存有以下特点:
  - **虚拟页往往很大**, 通常为4KB~2MB. (由于一些设计上的考虑, 一般就是4KB, 即 2^{12} Byte)
  - 是**全相联的**, 即任何虚拟页可以被缓存在任何DRAM物理页中, 减少抖动的发生
  - 采用**写回**的策略, 即尽可能推迟向磁盘的写入.
- **内存管理单元(Memory Management Unit, MMU)**
- **页表(Page Table)**: 页表条目的数组, 存放在 `物理内存` 中, 负责将虚拟页映射到物理页.
- **页表条目(Page Table Entry, PTE)**:

  虚拟地址空间中的**每个** `页` **都**在页表的一个**固定偏移量处**有一个 `PTE` .

  每个 `PTE` 由一个 `有效位` 和一个 `地址字段` 组成. (思考这里的"有效"是对于DRAM而言的, 所以 `未被缓存` / `未分配` 都是某种"无效"的页)
  - 如果设置了有效位, 表示这个虚拟页在DRAM中, 所以PTE的地址字段解释为**DRAM中**相应物理页的起始位置.
  - 如果没有设置有效位, 表示这个虚拟页**不在**DRAM中.

    有两种情况: 要么是这个虚拟页根本没被分配; 要么是分配了但没有缓存在DRAM中.
    - 如果 `地址字段` 是是空地址 `null` , 那么就是没被分配;
    - 如果不是空地址, 那么地址字段解释为该虚拟页在**磁盘中**的起始位置.

- **页命中**:

  考虑一个具体的例子. 比如CPU想要读虚拟内存中的一个字, 且这个字在 `VP2` 中,

  那么地址翻译硬件把虚拟地址作为一个索引, 就会定位到页表中的 `PTE2` ,

  如果发现设置了有效位, 那么地址翻译硬件就知道 `VP2` 是缓存在内存中的了.

  于是它利用 `PTE2` 的 `地址字段` , 可以构造出这个字在 `DRAM` 的物理地址. 这叫做**页命中**.

- **缺页(page fault)**: 对DRAM缓存不命中的特别称呼.

  假如在刚才的例子中, 发现**没有设置**有效位, 但地址字段又不是 `null` , 就会触发一个缺页异常.

  内核会调用 `缺页处理程序` , 该程序会进行如下操作:

  1. 选择一个牺牲页, 比如牺牲页是 `VP4` , 那么内核就会修改 `VP4` 对应的 `PTE4` , 反映 `VP4` 将不再被缓存在主存中的这一事实.

     同时由于采用写回的策略, 如果内核发现 `VP4` 被修改了, 那么必须把它复制回磁盘.

     > 当然不是简单地复制回磁盘的 `ELF` 文件! 因为你总不会希望 `ELF` 文件执行完之后被修改了!
     >
     > 事实上是被复制到了一个叫 `交换文件` 的位置. 之后会讲到.

  2. 从磁盘把 `VP3` 复制到 `DRAM` 中的某个位置, 并修改 `PTE3` 为 `VP3` 在了主存中的物理位置.

- **页面调度(paging)**: 包括调入/调出. 这里的出/入是相对于 `DRAM` 而言的. 比如调入表示的是一个页从磁盘调入到 `DRAM` 中.
- **按需页面调度(Demand Paging)策略**: 只有当产生不命中时, 才会调入新的页面. 这是所有现代系统采取的策略.
- **分配页面**:

  比如我们调用了 `malloc` 函数, 导致分配了一个新的虚拟内存页, 比如是 `VP5` .

  那么就会在磁盘上创建一个 `VP5` 的空间, 然后更新 `PTE5` , 使它指向这个新创建的页面.

- 操作系统为每个进程提供了一个独立的页表.
- 多个虚拟页可以映射到同一个 `共享物理页` 上.
- 虚拟内存简化 `加载` 的过程:

  `加载器` 并不会真正地从磁盘向主存复制任何数据, 而是只是单纯地分配虚拟页, 并标记为无效的.

  当程序真正需要访问某个内存位置时, 相应的页面才会被复制到 `DRAM` 中.

- **内存映射(memory mapping)**: 将一组连续的虚拟页, 映射到某个文件中任意指定的位置.

  > 应用级内存映射: 使用 `mmap` 系统调用. 后面会讲到.

- 基于虚拟内存的**内存共享**:

  操作系统将**不同进程**的相应虚拟页面映射到**相同的**物理页面, 这样多个进程就共享了这部分数据/代码.

- 基于虚拟内存的**内存分配**:

  当程序请求 `堆空间` 时, 操作系统分配连续的k个虚拟内存页, 并把每个页都映射到一个物理页中.

  考虑到页表的实际运作方式, 这k个物理页是并不需要连续的, 而是可以任意分散在物理内存中.

- 基于虚拟内存的**内存保护**:

由于**CPU通过虚拟地址访问内存**, 所以每次内存访问都涉及读某个 `PTE` .

所以我们可以为 `PTE` 中添加额外的一些 `许可位` , 来表示该进程对相应的 `页` 的访问权限.

比如如果设置了 `SUP` 位, 那么该进程只有在 `内核模式(super)` 下才能访问这个 `PTE` 对应的 `页` .

另外还有 `R` `W` `X` 位, 表示读/写/执行的权限.

如果一条指令违反了这些许可条件, 则会产生**段错误(Segmentation Fault)**.

- 术语表:

缩写	含义	描述
N	$N = 2^n$	虚拟地址空间的地址总数
M	$M = 2^m$	物理地址空间的地址总数
P	$P = 2^p$	每个页的大小(字节)
VA	**Virtual Address**	**虚拟地址, 分成VPN和VPO两部分**
VPN	Virtual Page Number	虚拟页编号
VPO	Virtual Page Offset	页内偏移, 即在某个虚拟页中的字节偏移
TLB	Translation Lookaside Buffer	专门缓存PTE的Buffer(其实就是Cache)
TLBI	TLB Index	用于组选择(其实Index就是Set号)
TLBT	TLB Tag	用于行匹配
PA	**Physical Address**	**物理地址, 分成PPN和PPO两部分**
PPN	Physical Page Number	物理页编号
PPO	Physical Page Offset	页内偏移, 即在某个物理页中的字节偏移
CI	Cache Index	SRAM的组索引
CO	Cache Offset	SRAM的块偏移
CT	Cache Tag	SRAM的行标记

- **虚拟地址 --MAPPING--> 物理地址**: 本质是 `VPN` 到 `PPN` 的翻译, 因为 `VPO` 和 `PPO` 是相同的!
- **页表基址寄存器(Page Table Base Register, PTBR)**: 存储 `页表` 在 `DRAM` 中的基址(当然是**物理地址!** 否则就mapping套娃了!).

  > `PTBR` 是一个 `CR` (Control Register, `控制寄存器`). 回忆 `GPR` (General Purpose Register)是指 `通用寄存器` .

- 对于一个n位的虚拟内存地址 `VPN|VPO` , 其中低p位是**虚拟页页内偏移(VPO)**, 高(n-p)位是**虚拟页号(VPN)**,
- High-level地看, MMU运作的逻辑是:

  首先利用 `VPN` 作为索引, 以 `PTBR` 存的地址为基址, 访问(处于高速缓存/主存中的)页表, 得到相应的PTE.

- 如果发现PTE**设置了**有效位, 说明命中了. 那么PTE的 `地址字段` 就是 `物理页号(PPN)`.
  得到了物理页号就好办了, 因为在虚拟页和物理页中, **页内偏移(Offset)**是一样的! 即 `PPO == VPO`.
  这样, 把 `PPN` 和 `VPO` 拼起来得到的 `PPN|VPO` 就是最终的物理地址.
- 如果PTE**没有设置**有效位, 那么引起 `缺页异常`, 控制传递到内核中的缺页处理程序.
  缺页处理程序会决定牺牲页, 并检查牺牲页是否被写过, 如果被写过, 就要更新回磁盘.
  然后把新的页面调入到DRAM, 更新PTE, 最后重新执行这条指令. 重新执行时将归结于命中的情形.

- 对于**SRAM**高速缓存, 大多数系统选择**物理寻址**.
  这是因为:
  首先, 高速缓存无需处理内存保护的问题, 因为权限检查是地址翻译过程负责的.
  此外, 使用物理寻址可以方便管理**多个进程**在高速缓存中缓存的块.

  > cpy: 我怎么感觉没得选? SRAM是对DRAM的缓存, 既然DRAM是物理寻址, 那SRAM就理应是物理寻址.

- **页表条目(PTE)**本身作为数据字, 当然也是**可以被缓存在SRAM**中的.
  所以MMU查阅PTE也是先向SRAM请求(通过物理地址 `PTEA` 进行请求).
  但这还是有点慢! 现在的系统会基于TLB进行优化加速🔄
- **TLB(翻译后备缓冲器, Translation Lookaside Buffer)**: MMU中用于缓存PTE的Cache. 是**虚拟寻址**(仅以VPN字段为"址")的.

  > 记忆: Translate 的时候, 先看看旁边(Lookaside)的Buffer里有没有缓存着你需要的PTE.

  当CPU想访问一个虚拟地址时, MMU首先接收到这个虚拟地址,
  然后通过 `虚拟地址` 中的 `虚拟页号(VPN)`, 访问TLB, 来取出相应的PTE(所以说TLB是虚拟寻址的), 得到物理页号(PPN).
  把PPN和VPO拼在一起, 就得到了可以用于访问SRAM/DRAM的物理地址.
  值得指出:
  1. MMU在不同的阶段使用 `虚拟地址[VPN|VPO]` 的不同位:
     - 在向TLB索要PTE时, 使用 `VPN`;
     - 在得到物理页号后计算物理地址时, 使用 `VPO`.
  2. TLB和之前学过的Cache有一点不同:
     之前的Cache的每一行都缓存**B个**字节, 而在TLB中, 每一行只会缓存**单个**PTE.
  3. 喂给TLB的地址并非整个虚拟地址, 而是只有 `VPN`. 而且 `VPN` 只被理解为 `TAG|SET` 两部分,
     这是因为每一行只存储单个PTE, 当然不涉及所谓的字节偏移.
  4. 当进程间上下文切换后, 由于TLB是虚拟寻址的, 所以其中的数据就不对了.
     解决方案是: 为TLB中的每一行再额外维护一个标记进程ID的字段, 当进行行匹配时, 除了tag, 还要检查这个字段.
  5. TLB并不是全相联的! 所以是有 `组选择` 的过程的.
  6. > 下面几行字是cpy写的, 而且非常存疑

> TLB本质上也是DRAM的高速缓存, 其特殊性仅在于它只会缓存DRAM中的PTE数据!
> 也就是说, 即使是在下文多级页表的语境中, PTE也是接受一个4KB对齐(后12位为0)的虚拟页地址(的不包括后12位的部分), 然后返还一个物理页地址(同样不包括后12位的部分).
> **所以TLB作用可以理解为**: 喂给它一个4KB对齐的虚拟地址, 然后尝试返回一个4KB对齐的物理地址!(都不包括后12位的部分)

- **多级页表**:
  考虑一个64位地址空间的计算机, 如果页表要包含所有的虚拟地址, 那么整个页表将占据非常非常非常非常大的空间.
  然而其中却只有很少的一部分虚拟地址是我们会用到的. 大多数根本就是未分配的(比如中间的大片区域).
  于是引出多级页表的思想:
  - `一级页表` 中的PTE不再存储某个 `页` 的物理页号, 而是存储某个 `二级页表` 的基址(DRAM中的物理地址).
    `二级页表` 同理, ... ,
    直到最后一级的 `K级页表` , 才是存储某个 `页` 的物理页号.

    > 以一个具有2级结构的页表为例: `一级页表` 的PTE映射到大小为4MB的 `片` , 每一 `片` 包含1024个连续的 `页` .

  - 如果片i中的每个页面都是未分配的, 则一级页表中的 `PTE_i` 就是 `null` , 而不必真正地对应于某个二级页表. 这样就节省了这个二级页表的空间占用.
  - 在具有k级页表的系统中, 虚拟地址被划分为 `VPN1|VPN2|...|VPNK|VP0` , 每个 `VPN_i` 表示到第i级页表的 `索引` .
    只不过在前k-1级中, i级PTE存储的是第i+1级页表的 `基址` , 而只有第k级的PTE才会存储某个PPN或磁盘块的地址.

    > 注意寻址的两个要素 `基址` 和 `索引` 分别怎么得到的!
    > `基址` : 一级页表的在 `CR3` 中, 其余的在 `PTE` 中
    > `索引` : 在 `虚拟地址` 的各个划分字段中

    > 在现代64位系统的设计中, 无论哪一级的页表, 一个页表自身都恰好占用一个页(4KB)的大小.
    > 而根据设计, 每个PTE占8个字节, 所以每个页表(4KB)中PTE(8B)的数量为512(4K/8)个. 所以索引是从[0~511]
    > 所以 `VPN1|VPN2|...|VPNK|VP0` 的 `[VPNi]` 都是$\log_2$(4KB/8B)=9位的.
    > 因此具有4级页表的系统中, `VPN1|VPN2|VPN3|VPN4|VPN0` 的位数为 9 + 9 + 9 + 9 + 12 = 48 ,
    > 解释了为什么现在64位系统中的实际可用地址只有48位.

- cpy: 总结一下, 在只考虑由3个Cache: `TLB` , `SRAM` 和 `DRAM` 组成的系统时,

对于一个虚拟地址, 为了翻译成物理地址, 首先要得到 `PTE` : 先问TLB, 命中了就得到了, 如果没命中, 就去从SRAM/DRAM中拿.

得到PTE之后, 就可以计算出物理地址. 然后用这个物理地址先去问SRAM, 命中了就得到了, 没命中就去DRAM拿.

- 多级页表的各级PTE都可以被缓存到TLB中!

- Core i7 采用四级页表层次结构, 每个进程都有它私有的页表层次结构.

  并且规定与已分配了的页相关联的 `页表` 必须留驻在物理内存中. (虽然理论上 `页表` 本身是可以被换出到磁盘的)?感觉现有知识不能解释如何实现页表如何换出/换入.. 因为我们认为PTBR中和多级PTE中都是直接存放DRAM的物理地址, 而不能是磁盘地址...)

  `CR3` 寄存器作为PTBR寄存器, 存储第一级页表的基址.

  `CR3` 是进程上下文的一部分, 因此在上下文切换时, `CR3` 的值要被恢复.
- Linux虚拟内存系统:
  - 虚拟内存被组织称一些 `区域` (或称作 `段`)的集合, 一个**区域(area)**就是**已分配的**虚拟内存的 `连续片(chunk)` .

    > 比如代码段, 数据段, 堆, 用户栈, 共享库段 等等都是不同的区域.

  - 每个已分配的虚拟页面一定保存在某个区域中. 换言之, 如果某个虚拟页不属于任何一个区域, 则不能被进程引用.
  - 内核为系统中的每个进程维护一个单独的 `任务结构(task_struct)` , 包含或指向内核运行该进程时所需的一切信息,

    包括: PID, 栈指针, ELF文件名, PC值等等. 其中一个条目指向 `mm_struct` , 它描述虚拟内存的当前状态.

    在 `mm_struct` 中, 存在 `pgd` 和 `mmap` 两个字段, `pgd` 存储第一级页表的基址, 而 `mmap` 指向一个 `区域结构链表` .

    这个链表中的每个结点都描述了当前虚拟地址空间的一个区域, 包括区域的起讫位置, 读写权限等信息.
  - 当MMU翻译某个虚拟地址而触发缺页异常时, 内核通过检查 `区域结构链表` 来判断这个虚拟地址是否落入某个区域,

    从而可以辨别出是否是一个非法地址(**段错误**). 以及是否有权限访问这个区域(比如只读代码段就是无权写入的, **保护故障**).

- **内存映射(memory mapping)**:

  为了**初始化**某个**虚拟内存区域**的内容, Linux将一个 `虚拟内存区域` 和一个 `磁盘上的对象` 关联起来.
- 虚拟内存区域可以映射到两种类型的 `对象` :
  - **普通文件**: 例如一个可执行目标文件.

    文件区(section)被分成页大小的片, 每一片包含一个虚拟页的**初始内容**.

    由于页面是按需调度的, 直到CPU第一次通过虚拟地址引用这个区域内的虚拟页时, 相应的数据才会真正进入物理内存.

> 当使用execve()新执行一个程序 `a.out` 时, `代码区` 和 `数据区` 就映射到了 `a.out` 文件的 `.text` 和 `.data` 区.

- **匿名文件**: 由内核创建, 所有bit为0的文件.
  当第一次引用这样一个区域内的虚拟页面时, 不会实际地发生磁盘到内存的数据传送. 因为只需要单纯用0来覆盖相应物理页即可.

  > (?不过gxt说: 是有一个单独的全0页, 然后同样使用copy-on-write技术映射, 当写入时才复制出单独的一份.)

  > 当使用execve()新执行一个程序 `a.out` 时, `bss区域` 就是请求全0的, 所以被映射到 `匿名文件` , 文件大小则包含在 `a.out` 中.
  > 而栈和堆区域也是映射到匿名文件, 初始长度为0.

- 无论是用哪种方式初始化, 只要一个虚拟页面被初始化了, 它就会在一个由内核维护的 `交换文件` **(swap file)**中换来换去.
  (?怎么具体理解) `交换文件` 的大小限制着当前运行着的进程能够分配的虚拟页面的总数.
- **共享对象**:
  如果一个进程A将一个 `对象` 映射到虚拟内存的一个 `区域` , 并且作为 `共享对象` 时,
  进程A对这个区域的写操作也会对其它映射这个对象的进程B可见. 这个区域也叫做 `共享区域` .
  而且***这些变化会反映在磁盘上的原始对象上***.
  内核利用文件名的唯一性, 可以判断某个 `共享区域` 也页是否已经在物理内存中了, 从而在内存中只需要存放共享对象的**一个**副本.
- **私有对象**:
  如果进程A将某个区域映射到一个 `私有对象` , 则进程A对这个 `对象` 的改变对其它进程不可见,
  而且***这些变化不会反映在磁盘的原始对象上***.
  当多个进程的私有区域映射到同一个私有对象时, 使用 `写时复制(copy-on-write)` 技术,
  只有当一个进程试图**写**私有区域时, 才会真正在物理内存中为这个进程单独复制出一个副本.

  > COW技术通过以下方式实现:
  > 1. 先将区域对应的**PTE**都标记为 `只读` , 并在内核的 `区域结构体` 中设置 `私有的写时复制` 标记.
  > 2. 然后当一个进程试图 `写` 这个区域时, 就会因为PTE的 `只读` 标记而触发 `故障处理程序` ,
  >   故障处理程序(内核态)注意到这个区域是 `私有的写时复制` 的, 所以知道了应该在物理内存中创建这个页面的副本,
  >   然后更新PTE指向这个新的副本, 并设置权限为 `可写` . 然后重新执行造成故障的指令即可.

这种技术延迟了私有对象的复制, 直到(因为写操作)不得不复制时才复制, 尽量减少了对物理内存资源的占用.

> `fork()` 就利用了 `写时复制技术` 来实现的父子进程的内存私有:

> 把父子进程中的每个页面的PTE都标记为只读, 并且每个区域结构都标记为 `私有的写时复制` .
>
> 这样, 当父/子进程任何一个进行 `写` 操作时, COW机制才会触发新页面的创建.

- `mmap()` : 用户级的内存映射:

  Linux进程可以使用mmap()函数创建新的 `虚拟内存区域` , 并将某个对象映射到该区域.

  ```
  1  void* mmap(void* start, size_t length, int prot, int flags, int fd, off_t offset);
  ```

  `mmap` 函数要求内核创建一个新的虚拟内存区域:

  - 最好(只是一个对内核的建议)从 `start` 地址开始(事实上我们通常传入 `NULL`),
  - 并把文件描述符 `fd` 指定的磁盘对象的一个 `"连续的片"` 映射到这个虚拟内存区域,

    这个 `"连续的片"` 由 `offset` 和 `length` 描述: 即从距文件 `fd` 偏移 `offset` 个字节开始的总共 `length` 个字节.
  - 参数 `prot` (protect的缩写)指定映射的虚拟内存区域的访问权限. 如 `PROT_EXEC` 表示这个区域的所有页面可被当做指令来执行.
  - 参数 `flags` 是一个位向量, 描述被映射对象的类型. `MAP_ANON` , `MAP_PRIVATE` , `MAP_SHARED` 位分别表示匿名/私有/共享对象.
  - 如果调用成功, 则返回指向 `映射区域` 的指针. 否则返回MAP_FAILED(-1).

  `mmap` 的应用:

  - 比如把一个文件输出到 `stdout` . 优势是不需要把任何数据先复制到User层, 而是在OS层就传输了.

- `munmap()` : 删除虚拟内存的区域.

  ```
  1  int munmap(void* start, size_t length)
  ```

  对已删除区域的引用会导致段错误.

下面的讨论和上面几乎无关, 因为我们将讨论一个运行在虚拟内存中的算法, 不涉及物理内存和虚拟内存的关系.

这个算法的特殊性在于它会利用虚拟内存中一个特定的区域"堆", 满足应用程序对动态内存分配的需求.

- **动态内存分配器**: 维护一个进程的**虚拟内存**区域, 称为**堆(heap)**.
  - `堆` 是请求二进制0的, 向上增长. 内核维护一个变量 `brk` (break的缩写), 指向堆的顶部. (类似于 `rsp` 之于栈的意味)
  - `分配器` 把堆空间视为一组不同大小的 `块(block)` 的集合, 所有的 `块` 分为两种, `已分配的` 和 `空闲的` .
- C标准库的 `malloc(size)` 函数:
  - 在成功分配时返回一个指针, 指向大小至少为size字节的内存块. (失败时返回NULL, 并设置 `errno`)
  - 这个指针在32位下总是8的倍数, 在64位下总是16的倍数.
  - `malloc` 不保证它返回的内存 `块` 是清零的.

    > 虽然 `堆` 的 `页` 在初始化时是请求全0的, 但当分配器把一个之前 `free()` 过的 `块` 再分配时, 这个 `块` 就不能保证是全0的了!

- `sbrk(incr)` 函数将内核的 `brk` 指针增加 `incr` 来扩展堆.
  - 如果成功, 则返回 `brk` 的旧值, 否则返回 `-1` , 并设置 `errno` 为 `ENOMEM` .
  - `incr` 取0是合法的, 含义是返回 `brk` 的当前值.
  - `incr` 取负数甚至也是合法的, 含义是收缩堆.
- C标准库的 `free(void* ptr)` 函数:
  - `ptr` 参数必须是一个由 `malloc` 系列函数分配获得的 `已分配块` 的 `起始地址` .
  - 它没有返回值, 所以不会告诉应用出现了错误. 这有时是非常棘手的.

- 分配器的**要求**:
  - 可以处理任何的 `malloc()` 和 `free()` 序列.
  - 立即响应请求, 而不能比如为了性能, 把一些请求缓存后重新排列.
  - 分配器用到的任何数据结构也必须存放在 `堆` 中. (这是为了书上没细说cpy也没搞懂的可拓展性着想)
  - 对齐 `块` , 使得返回的地址能满足任何类型的数据对象的对齐要求.

    - 如果一个块已经是 `已分配块` 了, 就不能再被修改或移动! (这是当然的, 否则应用程序拿到的指针就不对了啊! 数据也不对了!)
- 分配器的**追求:**
    - 尽可能高的 `吞吐率` : 即让单位时间能完成尽可能多的分配/释放请求.
    - 尽可能高效的 `内存利用率` : 因为内存空间是有限而宝贵的(因为已分配的虚拟内存页总数受 `交换空间` 大小限制).
        - **峰值利用率(Peak Utilization)**: 计算公式: `总的有效载荷(aggregate payload)` 的最大值 / `堆大小` 的当前值
- **碎片(Fragmentation)**: 有尚未被使用的内存, 但不能满足当前的分配请求. 分为两种情况:
    - 内部碎片: `已分配块` 的大小会比 `有效载荷` 大一些, 比如用于对齐, 或存放一些分配器算法所需的数据结构.
    - 外部碎片: 空闲内存 `合计` 满足这个分配请求, 但没有 `单独的一整个` 空闲块可以满足这个请求.
- 实现分配器需要思考的几个问题:
    - `空闲块组织` : 如何记录空闲块?
    - `放置` : 面对一个分配请求, 如何选择一个合适的空闲块用来分配?
    - `分割` : 当新分配的块放置到某个空闲块后, 这个空闲块如果有剩余的部分, 如何处理?
    - `合并` : 如何处理一个刚刚被free掉的块?

- **隐式空闲链表**: 所谓隐式, 是指所有的 `块` 是通过 `块头部` 中的 `大小字段` 隐含地连接着的. 堆空间的每个字节都将属于某个 `块` .

  为了方便, 可以在最后分配一个设置了分配位, 但大小为0的 `终止头部` , 用做结束标记. 可以简化算法的实现.
    - **关于对齐**: 注意所谓对齐不是指 `块头部` 8字节对齐, 而是块头部紧随着的 `有效载荷` 的首字节需要对齐!
    - **放置策略**: 首次适配 / 下一次适配 / 最佳适配
    - **合并空闲块**(处理假碎片): 立即合并 / 推迟合并
        - 带边界标记的合并(同时设置 `头部` 和 `脚部`).
        - 优化的带边界标记的合并(只有 `空闲块` 需要设置脚部, `已分配块` 只需在头部设置一个bit用于标记前边的块是否为空闲块)

实现一个简单的分配器:

- 在本节中, **字**是4字节, **双字**是指8字节. 这与Intel的称呼是不同的(回忆在机器代码中, Intel称2字节为一个word).
- **64位干净(64-bit clean)的**: 我们实现的分配器应该能够**不加修改地**在32位和64位进程中运行.
- 我们基于一个 `堆模拟器` 来实现分配器. 这个模拟器使用C标准库的 `malloc` 函数分配 `MAXHEAP` 大小的空间, 作为模拟堆的最大大小.
- 我们要实现 `mm_init()` , `mm_malloc(size)` , `mm_free(*ptr)` 这三个函数.
- 堆开始于一个双字(8字节)对齐的边界. 每次扩展堆时也应该以双字(8字节)为基本单位.

- **显式空闲链表**: 考虑到我们只需要维护空闲块, 且空闲块内部的有效载荷区域是可以被分配器利用的.

我们维护 `空闲块` 组成的双向链表, 而指针就存放在有效载荷区.

这使得 `首次适配` 策略下, 分配一个块的时间从O(块总数)降低到了O(空闲块总数).

不过也让 `最小块大小` 必须足够大, 以包含所需要的指针.

- **分离的显示空闲链表**:

  分离存储(segregated storage): 维护多个空闲链表, 每个链表中的块具有大致相等的大小. 具体而言, 是每个链表负责某个大小范围.

  比如可以划分为 {1},{2},{3,4},{5~8},{9~16},...,{1025~2048},{2049~4096},{4097~+∞}

  - 简单分离存储: 略.
  - **分离适配**:

    分配器维护一个空闲链表的**数组**. 每个空闲链表都是一个 `大小类` . 被组织称显式(或隐式???)链表.

    每个链表里包含这个 `大小类` 中的大小不同的块.

    - 当得到一个分配请求时, 首先确定这个请求所属的 `大小类` , 从而知道对哪个链表尝试进行 `首次适配` .

      如果成功找到了一个合适的块, 就分配这个块, 如果能分割就分割, 把分割之后的块分配到相应 `大小类` 的链表.

      如果没有找到合适的块, 就**搜索下一个大小类对应的链表**.

      如果所有链表都没有合适的块, 就向内核请求额外的 `堆空间` , 从这片内存中分配出一个块, 剩余部分放置到对应的大小类中.
    - 当得到一个释放请求时, 要尝试执行合并, 然后把得到的块插入到相应的空闲链表中.

MallocLab中, cpy的实现思路:

- 思路为分离适配(P605)

  分成11个链表, (各自容纳的范围定义在RANGE_i中)

  由于堆地址可以用32位编码, 所以绝对地址用(堆基址+相对地址)来表示
- 对于 `malloc(size)` 请求, 首先计算asize, 即这个请求所对应的块大小(包含header, footer和alignment的总大小).

  然后去对应的FreeBlockList去搜索, 如果找不到就搜索下一个大小类的空闲链表.
  - 如果没搜到, 那就调用new_FreeBlock(), 扩展堆, 并且只扩展asize个字节, 用于容纳这个分配请求.

    这里我们不尝试进行合并/分割, 因为这个newFreeBlock的大小是刚刚好的. 没有分割的余地, 没有合并的必要.
  - 如果搜到了, 那么就首先把header和footer设置好.

    然后从对应的FreeBlockList[index]中删除这个FreeBlock.

    如果发现这个FreeBlock的大小减去asize还剩下至少MIN_BLOCK_SIZE(16)个字节, 那就分割这个块.
- 对于 `free(p)` 请求,

  先把这个已分配块设置为Free的,

  然后看看能不能和前后的合并.

  合并的时候, 如果前/后块被合并掉了, 别忘了从相应的FreeBlockList[]中删除!

  然后把这个(尝试合并后的)块加入到对应的FreeBlockList[index].

包括: 设置FreeBlockList[index]的值(我们暂时选用直接插入到最前面), 设置之前 `最前面块` 的prev指针

`prev` 指针可以让删除时, 更方便让定位前驱成为O(1)的(?)

- 对于 `realloc(p, size)` 请求,
  - 如果p对应的块的block_size比size大, 那么尝试分割, 然后直接返回p即可.
  - 如果p对应的块的block_size比size小, 那么直接 new_p = malloc(size), 然后copy_data(), 然后free(p), 然后return new_p.